A practical introduction to real-time systems for undergraduate engineering

Douglas Wilhelm Harder
Jeff Zarnett
Vajih Montaghami
Allyson Giannikouris

The lead author would like to thank Maryse (May) Beauregard for proof reading this text throughout her term and to Colin MacPherson, among many other students, who made suggestions for how the original course could be modified. He would also like to thank the many MTE graduates and employees at Clearpath Robotics, Inc., especially Ilya Baranov. Additional suggestions for scheduling and the use of CAN bus and the networking and sockets sections areb based on notes provided by Andrew Morton.

Finally, and most importantly, the lead author would like to thank the many authors who have published research and textbooks on this and related fields as well as the many authors and editors of associated Wikipedia articles. While many of those papers and texts are excellent references, few are appropriate as a textbook for a junior undergraduate course in real-time systems, hence the reason for authoring this text.

Typographic conventions

This text uses a 10 pt Times New Roman font where *italics* indicates new terms and names of books. `9 pt Consolas` is used for program listings and console commands with output, and within paragraphs for keywords, variables and function names. Section titles are in Constantia.

Disclaimer

This document is intended for the instruction and examination of MTE 241 *Introduction to Computer Structures and Real-time Systems* at the University of Waterloo. The material in it reflects the authors' best judgment in light of the information available to them at the time of preparation. Any reliance on this document by any party for any other purpose is the responsibility of such parties. The authors accepts no responsibility for errors or omissions, or for damages, if any, suffered by any party as a result of decisions made or actions based on the contents of this text for any other purpose than that for which it was intended.

This draft is, unfortunately, still incomplete in many respects.

Printed in Canada.

A practical introduction to real-time systems for undergraduate engineering

Douglas Wilhelm Harder, Jeff Zarnett,
Vajih Montaghami and Allyson Giannikouris
University of Waterloo

Version 0.2018.07.31

To Sherry E. Robinson, Bill S. Lin and Jakub Dworakowski

Preface

This is an introduction to real-time systems for engineering students who are not focused on computer or software engineering.

This document is intended for MTE 241 *Introduction to Computer Structures and Real-time Systems.* The material in it reflects the authors' best judgment in light of the information available to them at the time of preparation. Any reliance on this document by any party for any other purpose are the responsibility of such parties. The authors accepts no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.

Contents

1 Introduction to real-time systems

This is a course that will introduce various computer structures and real-time systems. The topics this course will look at are

1. describing real-time systems,
2. considering appropriate programming languages for real-time, embedded and operating systems,
3. looking at the organization of a computer,
4. describing static memory allocation,
5. describing dynamic memory allocation, specifically those appropriate for real-time systems,
6. explaining threads and tasks,
7. scheduling these tasks,
8. dealing with hardware interrupts,
9. synchronizing the execution of tasks,
10. generalizing synchronization to resource management,
11. avoiding deadlock,
12. facilitating inter-task communication,
13. creating systems that are fault tolerant,
14. describing operating systems,
15. simulating the execution of real-time systems,
16. verifying that correctness of systems,
17. dealing with file management,
18. efficient data management,
19. considering issues with virtual memory and caching,
20. digital signal processing,
21. an introduction to digital control theory,
22. security, and
23. looking at what is ahead.

We will begin with our introduction by

1. describing what a real-time system is,
2. looking at a case study of anti-lock braking systems,
3. describing the components of a real-time system, including the environment, hardware and software, and
4. reviewing a brief history of real-time systems.

We will begin by describing a real-time system.

1.1 What is a real-time system?

Most of the software you've used to date has been interactive: it responds to your commands. Interactive software is always subject to delays. Surely you have experienced that feeling of waiting over a second for a word processor to respond to you entering a single keystroke, or the mouse taking a split second longer to respond than would make it seamless. We will define such systems as follows:

Definition: *General-purpose systems* (*hardware* and *software*) are tangible and intangible components of computer systems where operations are not subject to performance constraints. There may be desirable response characteristics, but there are no hard deadlines and no detrimental consequences other than perhaps poor quality of service if the response times are unusually long.

In contrast with general-purpose systems, *real-time* systems are meant to monitor, interact with, control, or respond to the physical environment. The interface is through sensors, communications systems, actuators, and other input and

output devices. Under such circumstances, it is necessary to respond to incoming information in a timely manner. Delays may prove dangerous or even catastrophic. Consequently, we will define a real-time system as one where

1. the time at which a response is delivered is as important as the correctness of that response, and
2. the consequences of a late response are just as hazardous as the consequences of an incorrect response.

Those requirements that describe how the system should respond to a given set of inputs (both from sensors and messages received from communication systems) given the current state of the system and what the expected outputs (both signals to actuators and messages sent through communication systems) and changes of state of the system are described as *functional requirements*. Other requirements are collectively described as *non-functional* requirements, and these include requirements concerning safety, performance and security, as described in Table 1

Table 1. Descriptions of non-functional requirements of real-time systems.

Non-functional requirement	Description	Example
Safety	This deals with operational responses by the system that protect the system prevent the system from coming into harm.	It has been determined that an increase in engine temperature can be dealt with by reducing the throttle if the increase is detected within 5 s; consequently, if the temperature sensor is checked at least once every 2.5 s, even in the worst case, the temperature will not exceed a critical value for more than 5 s.
Performance	The deals with either timing of responses or throughput necessary to protect the system from harm or other non-desirable outcomes.	It may be required that the fire-suppression system must be activated within 10 ms of the detected light intensity of an optical beam dropping below 95 %, or it may be required that a drone must be able to accept and process ten inputs from various sensors per second including the processing of video frames.
Fault tolerance	The ability to protect the system from harm resulting from design faults.	A quadcopter drone that is able to continue flying even if one of its four engines fails would be more fault tolerant than one that fails as soon as one of the engine fails. A drone that immediately attempts to land safely in the event of an engine failure and communicate its location would be *failsafe*.
Robustness	The ability to protect the system from harm resulting from external interference and perturbations.	Any communication between drones or other tasks is subject to natural interference that may cause the received message to differ from the message that was originally sent. A robust system could detect and correct such introduced faults.
Scalability	The ability to perform reasonably in an environment with added load.	If suppose ten drones cooperated on a task and require 1 ms/s to communicate while performing the task. If all drones were required to communicate with all other drones, one hundred drones attempting a similar task would spend 10 ms/s communicating; meanwhile, if the drones were divided into ten groups of ten each with one drone designated as a *leader*, after which only the leaders communicate, communication may be reduced to as little as 2 ms/s.
Security	This describes the operation of the system to prevents the system from intentional harm, including harm that may cause the operation of the system to be inconsistent with the intentions of the user.	One hundred drones performing a search-and-identify mission of an escaped convict cannot be interfered with in such a manner as to allow the non-detection of the convict or an intentionally false identification of the location of the individual. Similarly, one hundred drones engaged in a performance at a public event cannot be redirected to cause harm to the audience.

Other non-functional requirements may include availability, configurability and regulatory compliance. Real-time systems are not meant to be *fast*, per se; instead, they should be just fast enough to ensure that all functional requirements and non-functional requirements including, but not limited to, performance requirements.

Some examples of real time systems include:

1. transportation: control systems for and traffic control of vehicles, ships, aircraft and spacecraft;
2. military: weapons system, tracking and communications;
3. industrial processes: control for production including energy, chemical and manufacturing using robotics;
4. medical: patient monitoring, defibrillation and radiation therapy;
5. telecommunications: telephone, radio, television, satellite, video telephony, digital cinema and computer networks;
6. household: monitoring and control of appliances; and
7. building management: security, heating, ventilation, air conditioning and lighting.

We will look at anti-lock braking systems as a case study of both hardware and software real-time systems. However, as time is a central component of any real-time system, we will quickly first define time and embedded systems.

1.1.1 What is time?

Time is a natural phenomenon where one "second" is

> the duration of 9192631770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the caesium 133 atom at rest at a temperature of 0 K,

as defined by the *Bureau international des poids et mesures.* With the exception of the kilogram, all other units are defined relative to the second. Atomic clocks are used to measure time, and *coordinated universal time* (UTC) is an international standard for time. Your systems will, however, be using quartz clocks, where a quartz crystal is carved to vibrate at 2^{15} Hz = 32768 Hz when an electric field is placed across it. A 5-bit digital counter will overflow once per second as it counts the oscillations. With 86400 s/day, such clocks tend to drift less than 1 s/day and therefore different systems will have different times even if they start synchronized (more expensive crystals will have less drift). In the chapter on fault tolerance and robustness, we will look at techniques for synchronizing clocks between systems.

1.1.2 What are embedded systems?

Elicia White definition of an embedded system is

> a computerized system that is purpose-built for its application.

The purpose-built includes both hardware and software components. Software for embedded systems is usually written on general-purpose computers running integrated development environments (IDEs) using cross-compilers: compilers that produce machine instructions for processors other than the processor running the IDE. An embedded system should usually be considered an *object* within a larger system. The embedded system should have well defined functionality that allows it to be replaced by another system that adheres to the same specification.

The challenges of writing applications for embedded systems include constraints such as

1. cost,
2. correctness (the system must be close to error free),
3. main memory availability (random-access memory or RAM),
4. code size restrictions (read-only memory (ROM) or flash memory),
5. processor speed,
6. power consumption, and
7. available peripherals.

As you have seen in your study of algorithms and data structures, there is often a trade-off between speed and memory; for example, a doubly linked list requires $\Theta(n)$ more memory, but allows many $O(n)$ run-time operations in a singly linked list to now run in $\Theta(1)$ time. Similarly, trade-offs can be made between the above constraints. Other concerns with developing applications on embedded systems include

1. uncertainty as to whether issues are software or hardware,
2. the possibility of software errors causing damage to hardware, and
3. the systems tend to be remote; that is, access and maintenance (including upgrades) tend to be non-trivial issues.

None of these are concerns with software development for general-purpose processors.

1.2 Case study: anti-lock braking system

From physics, you may recall that static friction is stronger than dynamic friction. When trying to stop a vehicle in a very short distance or on a slippery surface, it is possible for the wheels to lock and stop rotating. When this happens, the vehicle begins to skid (dynamic friction) and loses traction. This also means the driver no longer has control over the vehicle—a dangerous situation. If the wheels do not lock up, the driver will not only have control while stopping, but the vehicle will also stop in a shorter distance.

A skilled driver can ascertain the maximum amount of brake force that can be safely applied without causing a skid. This technique is called threshold braking. It is a very difficult technique to learn and use, especially in a situation where emergency braking is needed.

Anti-lock braking systems (ABSs) were first developed in the late 1920s for aircraft as a mechanism for preventing skidding during landings, as skidding will significantly reduce the lifetime of the tires and skidding in wet conditions can lead to dangerous situations. While threshold braking is possible in smaller systems such as automobiles, it is exceptionally difficult in aircraft. The entire ABS was hydraulic using a flywheel and valve that would under differential spin cause pressure to bleed from the brakes, allowing the wheels to unlock but continue to apply a braking force.

In 1958, an anti-lock brake was built for a motorcycle where it reduced stopping distances on slippery surfaces by as much as 30 %. In the 1960s, such a system was built for automobiles. In both cases, the product never went into mass production.

Computerized anti-lock braking systems were introduced by Chrysler in 1971 and it was an option available for many luxury models for the next decade. It was first introduced as a standard feature in the 1985 Ford Scorpio, for which it was awarded the European Car of the Year Award in 1986.

In addition to speed sensors and hydraulic valves, modern ABS interfaces with a central electronic control unit (ECU). The ECU is an embedded system comprised of a number of computer modules that control various aspects of the car. The ECU today includes one or more microcontrollers, a clock, memory, both analog and digital inputs, and output drivers, while communication is usually through a CAN (controller area network) bus. ISO 26262 *Road vehicles—functional safety* is a standard that directs the development process of such modules.

Starting in late 2009, the National Highway Traffic Safety Administration (NHTSA) began receiving complaints concerning brake problems on the Toyota Prius that manifested itself as a *short delay* in regenerative braking when hitting a bump; consequently increasing the stopping distance. This was solved via a software update; however, it is not clear from the literature as to whether it was a hardware bug, or if the necessary correction could be done in software.

Note that for the microcontroller of ABS, faster is not better. A design that meets the required specified deadlines is all that is sufficient. Reliability is a much greater factor than performance. Once a design for a system such as ABS is developed, unlike desktop or mobile computer programs, there will be no need to revisit the design every year. In fact, the incentives point the other way: the system works and any change introduces the possibility of error.

1.3 Components of real-time systems

The defining characteristic of any real-time system are the timing requirements: not only must the system respond correctly to inputs, it must do so within a specified amount of time. Such requirements can generally be categorized as either

1. absolute requirements where the response must occur at defined deadlines, and
2. relative requirements where the response must occur within a specified period of time following an event.

The consequences of failing to satisfy deadlines allows one to describe real-time systems as

1. *hard real-time* where failure to meet a deadline results in a failure and any response—even if correct—following the deadline has no value,
2. *firm real-time* where failure to meet the occasional deadline will not result in a failure yet any response following a deadline has no value, but such a failure will result in a degradation of quality of service, and
3. *soft real-time* where the value of a response drops following the passing of a deadline, but the response is not wasted.

In the first two cases, if it can be determined *a priori* that the deadline will not be satisfied, it may be better to not even begin to calculate the response. More complex real-time systems will likely consist of subsystems from each of these three categories.

A real-time system is always interacting with the physical world, and a model of a real-time system, as described by Michal A. Jackson, includes the system itself, the environment and the interface. Connecting the system and the environment are input (e.g., sensors), output (e.g., actuators) and bi-directional flow of information (e.g., communication channels). These components invariably are physical in nature and thus, while providing information to the system, they are also part of the environment. This high-level approach is shown in Figure 1-1.

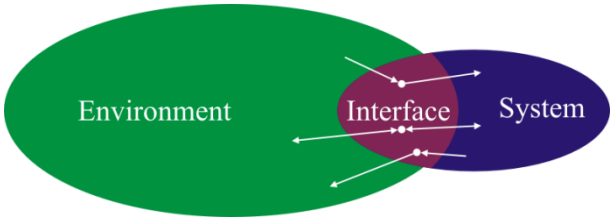


Figure 1-1. A model of a real-time system.

The system and interface will usually be comprised of both hardware and software; however, the last may be excluded in a purely mechanical or electrical system; however, this book will focus on those systems using a software-driven controller. Never-the-less, many of the lessons you take out of this book will have analogous applications in either pure mechanical or electro-mechanical systems. Reasons for using software to control real-time systems include:

1. the development costs are significantly lower (tools and developers are more readily available),
2. the software can be verified to be correct, and
3. maintenance can be easier as it may require only a software update.

The expense, however, is that the unit cost will be higher, as each unit will require a microcontroller and an appropriate power source. Despite this additional cost, approximately 99 % of processors made today are for embedded systems, many of which are real-time systems. We will discuss these three aspects next.

1.3.1 The environment

The environment that the real-time system is in is beyond the control of the engineer and it must, therefore, be modelled. A real-time system can be tested in a simulated environment driven by the model and it can be validated to work under the most extreme circumstances presented by the model. If the model, however, is inaccurate, any subsequent system

may fail (as the real situation may be more demanding than the model suggested) or be excessively expensive (scenarios the system was set up to handle—costing developer time and possibly more expensive hardware—never occur). Modelling the environment is beyond the scope of this text.

1.3.2 Real-time hardware

The hardware of a software-driven real-time system first must be predictable. While this is likely obvious for any microprocessor, this also applies to sensors, actuators, other input and output devices, and communication systems.

Counter-intuitively, many of the advances in processor technology make it more difficult determine predictability: instruction pipelining, branch prediction, virtual memory and caching pose serious challenges for determining the timing behaviour of a system. These enhancements were designed to make the processor perform faster (under most circumstances), not more predictably. We will discuss some of these in a later topic.

The hardware must also be reliable and fault tolerant as well as controller driven; that is, it must be able to interact with the processor through a communication bus. Devices will require both polling and interrupt support. These concepts will also be discussed in Chapter 8 of this book.

Devices will be connected to the processor through one or more communication busses. Any shared bus will result in competitions for that resource that will degrade performance and make timing behaviour more difficult to ascertain. Furthermore, any interactions through a communications channel (wireless, Ethernet, etc.) also make for challenges in creating real-time systems (there are real-time protocols such as real-time transport protocol (RTP) as opposed to transmission control protocol (TCP), but these require additional support).

One observation is that there is no requirement for the hardware to be fast. It only needs to be fast enough as is necessary to control the expected environment in the desired manner. Consider, for example, the 8-bit Freescale RS08 microcontroller, which is a descendant of the Motorola 6800. It has only one data register: an 8-bit *accumulator*; it uses a 14-bit address register which allows for a maximum of 2^{14} = 16 KiB of main memory, and the maximum processor speed is 20 MHz—200 times slower than modern general-purpose processors. The unit cost is on the order of 50 cents and less in bulk.

Hardware failures in real-time systems usually result in malfunctioning equipment, and the system may or may not be able to recover from such failures. An interesting example of a variation of a hardware failure from which a recovery was possible was in 2010, when Voyager 2, which was 13 light-hours away from Earth, experienced a communications failure. This was narrowed to a problem where "[a] value in a single memory location was changed from a 0 to a 1"[1]. Fortunately, this could be solved with a reset of the memory; although it took over a day to determine that this solution was successful.

1.3.3 Real-time software

While there are issues that affect the predictability of hardware, the timing characteristics of hardware, never-the-less, tend to be easier to quantify. If the characteristics of a device are not adequate, it is possible to search other products. The jungle of possible software implementations of the same algorithms are, however, more varied. Therefore, the first two-thirds of this course will focus on real-time software systems: dealing with the challenges posed in devising algorithms that satisfy the timing constraints of real-time systems. A small real-time system may contain only one processor and a few hundred lines of code, while the projected estimates for the mid-1980s space station "Freedom" ran closer to 20 million lines of Ada.

[1] Veronia McGregor of the Jet Propulsion Laboratory quoted in "NASA Finds Cause of Voyager 2 Glitch" , May 18, 2010 by Irene Klotz.

There are two configurations for real-time systems, programs where access to resources is

1. direct through machine instructions, and
2. indirect through an intermediate operating system that mediates such requests.

Whether or not there is an operating system mediating requests for resources, it is necessary to manage the resources available to programs. In this course, we will consider the management of such resources, including:

1. the processor,
2. main memory,
3. peripheral resources,
4. synchronization between tasks, and
5. file systems.

We will conclude the course by showing that the cumulative efforts we have made in managing these resources can be bundled into a single operating system *kernel* that executes in a protected environment which prevents executing programs from accidentally corrupting main memory or accessing other resources currently engaged in other tasks.

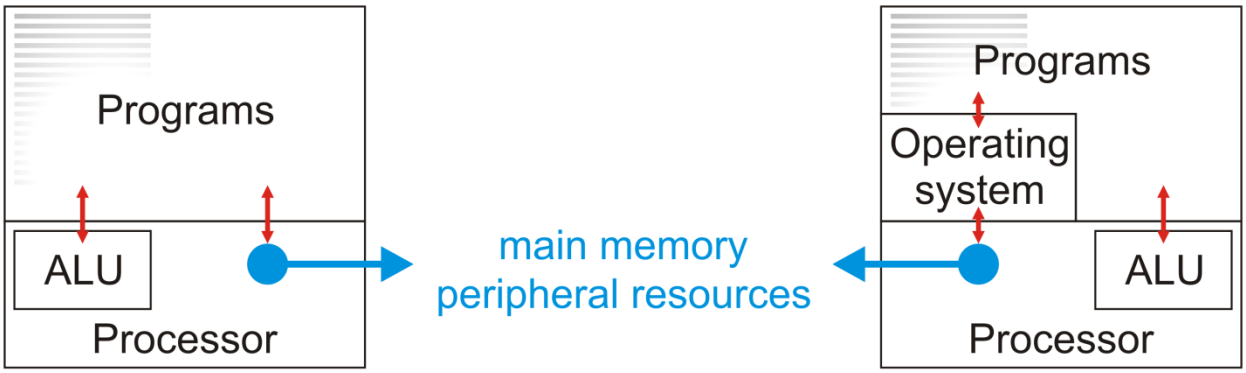


Figure 1-2. Configuration of smaller embedded systems versus larger embedded and general-purpose systems.

Numerous failures, apart from software errors (*bugs*), in real-time systems can be described as being the result of

1. race conditions,
2. unexpected environmental conditions, and
3. failures in the model.

The majority of this text will look at avoiding race conditions through synchronization and deadlock avoidance, but we will also look at software simulation and verification.

A race condition occurs when the response of the system (hardware or software) depends on the timing or sequencing of events or signals initiated by independent tasks, but where at least one of the responses is undesirable. These are non-deterministic bugs that are often difficult to find, as it may be very difficult to recreate the exact circumstances causing the failure; hence the alternate name, *Heisenbug*.

To give some examples of race conditions, suppose two individuals are driving their cars down a three-lane highway, one in the left lane and the other in the right, and each wishes to change into the middle lane. This is only a problem if both cars are in line with each other and both drivers want to make the lane change in the same five-second window. This is exasperated by factors such as lighting conditions, the alertness of the drivers, the presence of distractions, some drivers only checking the middle lane for traffic, some drivers checking first and then signalling, while others signalling first and then checking (ideally, you check, then signal and then check again), and yet others may not check, or not signal, or not do either.

Another example of a race condition is when you agree to meet someone at a building at a specific time, but when you get there, you realize that you could be meet at either the front or the back entrance. Staying at one entrance could see both of you waiting indefinitely long, but going from one entrance to the other may have both of you miss each other if you both within the same 20-second window decide to take two different paths between the two possible meeting points (after all, you could walk through the building, clockwise around the building or counter-clockwise around the building). This is less of an issue today, so long as everyone's mobile phone is charged.

We will look at three examples of how race conditions:

1. killed patients in the Therac-25 killed,
2. almost ended the adventures of the Mars rover "Spirit" before the end of the first month, and
3. affect circuit and the benefits of circuit simplification.

We will start with Therac-25.

1.3.3.1 Therac-25

A race condition in the response of the Therac-25, a radiation therapy machine produced by Atomic Energy of Canada Limited (AECL), to operator instructions led to patients being given 100 times the expected radiation. This was the result of a race condition in which if the operator issued an instruction too soon after a previous instruction, the system was still responding to the first command and therefore ignored the second without any notification that it was doing so. Three patients died as a result.

1.3.3.2 The Mars rover "Spirit"

On January 4th, 2004, the Spirit rover set down on Mars to begin its 90-sol (Martian day or 1.027 Earth days) mission of exploring the planet surface. It would go on to communicate information back to the Earth for a total of 2210 sols, ending on March 22nd, 2010. However, a race condition due to a failure in modeling and an unexpected environmental condition may have catastrophically curtailed its mission to a mere 16 sols.

Figure 1-3. The Martian rover "Spirit" (from NASA).

The rover has a processor, 120 MiB of RAM and 256 MiB of flash memory, part of which contained files relevant to the operating system and 230 MiB of which are dedicated to a flash file system that stores data produced by the various instruments and cameras. The operating system is Vx-Works version 5.3.1 by Wind River Systems, a real-time OS that was compiled with flash file system extension. For the file system to work, however, critical information must be stored in appropriate data structures in main memory (this will be discussed in Chapter 17). Everything was fine, except for a sequence of unlikely events, which was not anticipated by the software designers.

After the rocket carrying Spirit launched on June 10th, 2003, it was determined that there were serious issues with the existing software. During the trip, new files were uploaded to the rocket carrying the rover and then installed on the rover itself. Everything seemed good to go. They even simulated Spirit in operation for 10 sols to ensure that this new installation would not cause any problems. However, the new installation added approximately a thousand extra files and directories compared to the original software.

On sol 15 (15 Martian days after landing), a utility was uploaded to Spirit to delete the obsolete files and directories, but only one of the two components was received; therefore, a second transmission was scheduled for sol 19. On sol 18, however, the rover's scientific instruments and cameras were busy collecting data and creating data, and instructions were sent to add these new files into the flash file system. Only now, the flash memory system made a request for additional memory, but the old files and directories occupied the remaining memory, so the request for additional memory could not be fulfilled. The system did what it was designed to do if there was a failure: reset. This is more or less what most people do at home when their computer fails to respond, but in this case, the reset was automatic.

So the operating system reset, as directed. On start-up, it tries to mount the flash file system, this results in a memory request which is, again, denied. So the system resets again and again... This cycle of resets ended most communications with Earth and posed a serious problem for Spirit: it could not go to sleep at night, and therefore its system was overheating and the battery was running low. The operators on Earth even sent the command SHUTDWN_DMT_TIL (*shutdown, dammit, until*—someone had a sense of humor) in hopes of putting Spirit to sleep, to no avail; unbeknownst to the operators, the reset sequence had priority, even over the shutdown command.

With no additional information, it was assumed that Spirit was in a reset cycle (there may have been other causes, for example, a solar event (solar flare or storm) had occurred just prior to Spirit's silence, but a reset cycle was the only one that they allegedly could do anything about), and this would point to a problem in either the flash memory system, the EEPROM (Electrically Erasable Programmable Read-Only Memory), or a hardware failure. Fortunately, the software programmers included two features that allowed a recovery: a window of time was inserted between resets that allowed commands to be received, and it was possible to issue a command to boot without installing the flash file system. At this point, on sol 21, they were finally able to issue the command to give Spirit the sleep it required.

For the next two weeks, every Martian morning, a command was sent to wake up and reset without loading the flash file system. Utilities were uploaded to manipulate the flash memory directly without loading the file system. This caused some corruption, but some information was recovered, including a photograph of the Rock Abrasion Tool (RAT) (shown in Figure 1-4), and more importantly a log of every event leading up to, and including, the failed request for additional memory. Once the system was stable, an exception-handler utility was developed that would recover more gracefully from an allocation error than simply triggering a reset.

Figure 1-4. The RAT.

Incidentally, the Opportunity rover landed on Spirit's sol 21—only hours after they were finally able to put Spirit to sleep. This summary is compiled from information appearing in Ron Wilson's *The trouble with Rover is revealed* (http://www.eetimes.com/document.asp?doc_id=1148448) and Mark Adler's blog entry and presentation *Spirit Sol 18 Anomaly* (http://hdl.handle.net/2014/40546).

1.3.3.3 Logic expression simplification

Another example of a race condition, but Consider the circuit shown in Figure 1-5. From predicate logic, the result should always be equal to zero.

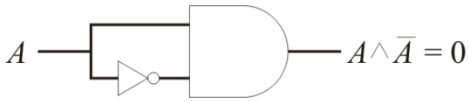


Figure 1-5. A simple circuit with one input and output.

Unfortunately, with each circuit element, there is a slight delay as to how long it takes a change to propagate to the output. Consequently, the actual timing diagram of the voltages looks like what you see in Figure 1-6.

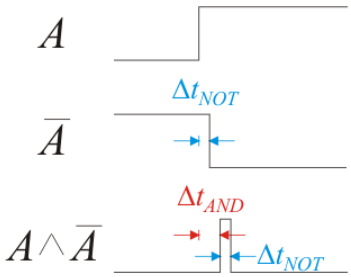


Figure 1-6. The timing diagram of the circuit in Figure 1-5.

Thus, the output, rather than being a constant 0 V, it exhibits a spike (a window of short duration where the output is not zero). Any circuit, however, that expects a clean 0 V may react adversely to the spike if this is not accounted for. To minimize the number and impact of such transient intermediate states, Karnaugh maps are used to simplify Boolean expressions such as:

$$ABCD + AB\overline{C}\overline{D} + AB\overline{C}D + A\overline{B}CD + A\overline{B}C\overline{D} + A\overline{B}\overline{C}\overline{D} + \overline{A}\overline{B}C\overline{D} + \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}BCD + \overline{A}BC\overline{D}$$
$$= AB\overline{C} + \overline{A}BC + ACD + \overline{B}\overline{D}$$

1.3.3.4 Summary of real-time software and race conditions

We've discussed some situations where the sequence in which events occur can result in problems. Such conditions are called *race conditions*. Later in the course, we will look at solutions to such problems, at least in software.

1.3.4 Summary of the components of real-time systems

Thus, a software-controlled real-time system will work in an environment of the physical world, interfaced through hardware and administered by software. This course will focus on the software component of real-time systems.

1.4 The history of real-time programming

Programming real-time systems arose in parallel with the construction of large commercial and government systems in the 1960s. In his 1965 text *Programming Real-time Computer Systems*, James Martin discusses issues such as dynamic scheduling, dynamic core allocation, allocation of priorities, multi-programming, interrupts, queues, overloads, multi-processing, communication lines, random-access files, supervisory programs, communication with other computers, high reliability, duplexing and switchover, fall-back, programming test, problem of programmer coordination, design problems, and monitoring the programming progress. All of these issues remain associated with real-time programming today. At that time, the larger real-time systems included air defence, telephone switching, airline reservations and the space program and these often grew faster than programming paradigms could keep up. It was only in 1965 that Edsger Dijkstra proposed the concept of a semaphore, a variable used for controlling access to a shared resource (we will examine this in a later topic in great detail), to deal effectively with synchronization—synchronization and concurrency is not even a significant topic in Martin's book.

With the introduction of semaphores (special flags) and other innovative ideas, issues such as mutual exclusion and serialization could now be dealt with in a manner that could be proved to be correct. One major step forward was with the United States government requirement of a language designed for real-time and embedded applications; the result was the programming language Ada. Furthermore, the greater availability and lower cost of processors made it desirable to shift control out of hardware and into software—not without failures—to reduce development costs. Finally, in the last two decades, real-time systems have moved into the realm of mass-produced consumer products and thereby providing significantly more investment in developing real-time systems in the commercial industries.

1.5 Topic summary

In this topic, we introduced real-time systems, we looked at a case study of the development of anti-lock braking systems, we described the relationship between the environment, hardware and software in a real-time system and looked at two situations where race conditions may lead to issues in real-time systems through race conditions.

Problem set

1.1 In one sentence, what differentiates a real-time computer system from a conventional computer system?

1.2 Recall that form your algorithms and data structures course that it is often possible to speed up an algorithm if you are willing to store more information. While this leads to often more complex functionality and increased development costs, such options are often taken in conventional computer systems. Why would you have to be more careful about such trade-offs when you are dealing with an embedded system?

1.3 There are two requirements for an anti-lock braking system (ABS):

1. the vehicle must slow down, and
2. the tires cannot skid.

Without specific numbers, what are some of the timing requirements for such a real-time system? Why does releasing the pressure on the brakes actually decrease the braking distance?

1.4 Suppose that the ABS component of a brake system fails, how should the system respond? Why?

1.5 Draw a block diagram of an ABS system.

1.6 Section 1.3.2 describes the characteristics of the Freescale RS08 microcontroller. It has only one data register—an 8-bit *accumulator*. All operations involve either modifying this register, writing to the register, or saving the value to a memory location. Any binary operation requires that one of the operands be located in main memory where it is fetched using direct or indirect addressing, possibly with an offset. Is this reasonable for a system where the majority of the operations involve calculating statistics based on input from a sensor, or would it be better to get a system that has two or more registers?

2 Real-time, embedded and operating-system programming languages

In this topic, we will look at real-time programming languages and characteristics of such languages to help ameliorate the likelihood of faults occurring. We will also consider characteristics of programming languages appropriate for embedded systems and for programming operating systems. We will then consider the programming language we will use in this class: C.

2.1 Programming languages

What language should we use for real-time systems, and why is C so prevalent? Why do we not use, for example C++, C#, Pascal, Ada, Java or another programming language for our projects? We would prefer a language that is appropriate for

1. real-time,
2. embedded, and
3. operating

systems development. On any project, it is necessary, however, to standardize the language.

Rule 1 of the Jet Propulsion Laboratory (JPL) coding standard specifies that the when C is used, any programs must adhere to the ISO/IEC 9899-1999(E) standard. By specifying the standard to which source code must adhere to, this ensures

1. any compliant compiler can be used,
2. the source code can be analyzed by tools meant to verify and , and
3. any competent programmer is able to work with the code as necessary.

The source code may not rely on behaviors not specified in the standard. By avoiding compiler- or platform-specific behavior, it avoids issues of portability and code reuse in future projects.[2]

We will discuss these development domains, followed by a discussion of other software design techniques applicable to real-time systems. We are not assuming that an operating system is necessarily in place. Instead, we will investigate the various structures and modules necessary to accomplish goals in real-time systems, and we will conclude the course by observing that these structures and modules are sufficiently common and critical that they can be placed into a protected environment which a user cannot accidentally or even deliberately interfere with. Many vendors will provide real-time operating systems (RTOSs) which you can use off-the-shelf; however, in this course, you will understand how those structures and modules are designed and how they work so that when you use a vendor product, you will understand what is going on *under the hood*.

A technician (programmer, electrician, construction worker, etc.) should know how to use a tool or package, an engineer should know how that tool or package works.

We will look at

1. programming paradigms,
2. ideal characteristics of programming languages for given systems, and
3. other software programming techniques.

[2] Throughout this text, we will be referencing many of the rules from the JPL *Institutional Coding Standard for the C Programming Language*. A primary object of the Jet Propulsion Laboratory, in addition to many other objects, is the design, construction and operation of planetary robotic spacecraft.

2.1.1 Programming paradigms

Before we start looking at programming languages, it is likely useful to compare procedural languages with object-oriented languages. We will proceed historically through

1. structured programming,
2. procedural languages,
3. data abstraction,
4. object-oriented languages and
5. design patterns.

These reflect the evolution of software engineering, each contributing to the previous. Initially, software was programmed entirely in assembly, and it was only with the introduction of COBOL that programming became abstracted from the machine instructions. This further lead to data abstraction and behavioral abstraction, together with the identification of common problems with recognized solutions. There are other programming paradigms such as functional programming, logic programming and aspect-oriented programming, but procedural and object-oriented are the two most commonly found in real-time systems and embedded systems development.

2.1.1.1 Structured programming

The concept of *structured programming* is based on the structured-programming theorem which says that

1. blocks of code executed in sequence,
2. a Boolean-valued condition selectively executing one of two blocks of code (*conditional statements* or *if statements*), and
3. repeatedly executing a block of code until a Boolean-valued condition is false (*repetition statements* or *loops*)

is sufficient to express any computable function. Prior to this, especially when programming was within the realm of assembly language, you could expect to see, for example, code that looks like:

```
void insertion_sort( int *array, int n ) {
    int i, j, tmp;
    i = 0;

    start:
    if ( ++i == n )
        return;

    tmp = array[i];
    j = i - 1;

    loop: if ( array[j] > tmp )
        goto copy;

    array[j + 1] = tmp;
    goto start;

    copy: array[j + 1] = array[j];

    if ( --j >= 0 )
        goto loop;

    array[0] = tmp;
    goto start;
}
```

The *structured programming paradigm* tries to improve the quality and reduce the development and maintenance time of programming by requiring the user to restrict flow control to:

1. blocks of instructions,
2. conditional statements, and
3. repetition statements.

The primary goal being to combine a series of statements into *blocks* meant to be executed as a unit and avoiding statements such a `goto` that allow for a myriad of execution sequences, also known as *spaghetti programming* due to the myriad of crossing paths of execution, as highlighted in Figure 2-1.

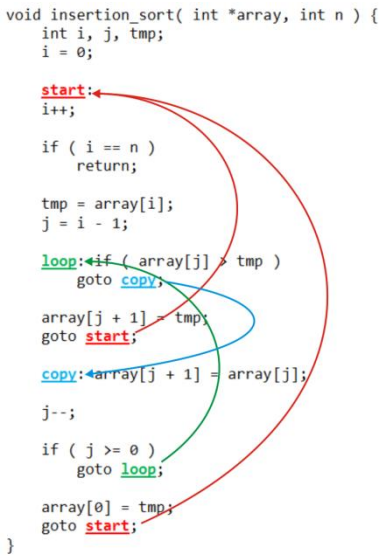


Figure 2-1. The basis for the term *spaghetti programming*.

The above implementation of insertion sort would be written as

```
void insertion_sort( int *array, int n ) {
    int i, j, tmp;
    bool found;

    for ( i = 1; i < n; ++i ) {
        tmp = array[i];
        found = false;

        for ( j = i; !found && (j > 0); --j ) {
            if ( array[j - 1] > tmp ) {
                array[j] = array[j - 1];
            } else {
                found = true;
            }
        }

        array[j] = tmp;
    }
}
```

The cost, however, is not zero: even with all optimizations turned on, the structured programming approach for this problem contains 5 % more instructions and is 10 % slower than the unstructured approach. The benefits of structured programming, however, in terms of readability and understandability and therefore reduced development and maintenance costs severely outweigh these negligible costs. When the arguments for structured programming were first put forward, there was a significant outcry and it was years before the benefits were recognized by the programming community as a whole.

One paradigm that uses structured programming is *procedural programming*.

2.1.1.2 Procedural programming

The C programming language uses the *procedural programming paradigm*. Structured programming is achieved through the solution of a problem being specified by a sequence of computational steps that must be performed on input data to transform it into a format that gives a solution to the given problem. Thus, each problem can be specified by

1. the format of the data that is necessary to solve the problem (including any necessary assumptions on the state of that data), and
2. the transformation on that data in order to produce the solution to the problem.

The primary mechanism for solving a problem is a *function* that takes data as input (*parameters*) and returns data as a solution to the problem (the *return value*). When a function is called, it is passed specific input, or *arguments*. By compartmentalizing problem-solving to function calls, this allows for the re-use of existing code and thus reducing code duplication.

Consider Gaussian elimination: this is a near ubiquitous algorithm used in most solutions requiring linear algebra. The idea is so simple that most programmers will implement it in-place—in one application, there were 16 separate implementations scattered throughout the libraries—however, the algorithm is subject to numerical instabilities that can be resolved by selective use of pivoting and scalar multiplication. For example, for sufficiently small α, $\begin{pmatrix} \alpha & 1 \\ 1 & 2 \end{pmatrix}\mathbf{x} = \begin{pmatrix} 1 \\ 3 \end{pmatrix} \equiv \begin{pmatrix} \alpha & 1 \\ 0 & 2+\frac{1}{\alpha} \end{pmatrix}\mathbf{x} = \begin{pmatrix} 1 \\ 3+\frac{1}{\alpha} \end{pmatrix} \not\equiv \begin{pmatrix} \alpha & 1 \\ 0 & \frac{1}{\alpha} \end{pmatrix}\mathbf{x} = \begin{pmatrix} 1 \\ \frac{1}{\alpha} \end{pmatrix}$, where $\mathbf{x} \approx \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ for the first two, and $\mathbf{x} \approx \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ for the last. Recall that $2 + 10^{16}$ can be stored exactly using double-precision floating-point numbers, but $3 + 10^{16}$ is stored as $4 + 10^{16}$ due to rounding. Of the 16 implementation, all but two had bugs related to floating-point computations.

In general, any large problem can be solved by sequentially solving conceptually easier sub-problems. Consequently, structured programming is achieved by identifying computational steps that solve simpler problems that are necessary to solve the larger problem and then implementing functions to solve the smaller sub-problems.

For example, the quicksort function can be described as

1. taking as input an array of unordered items that can be linearly ordered, and
2. reordering the items so that they are sorted according to the linear order.

This problem can be solved by taking the following computational steps:

1. if the size of the array being sorted is less than 20, call a non-recursive sorting algorithm;
2. otherwise, sort the list as follows:
    a. choosing a pivot and removing it from the list (choose the median of the first, middle and last entries of the array being sorted; place the other two accordingly);
    b. starting from the front and back,
        i. finding the next item greater than the pivot,
        ii. finding the previous item smaller than the pivot,
        iii. swapping the two

        until the entries are partitioned into those less than the pivot and those greater than it;
    c. placing the pivot between the two partitions; and
    d. calling quicksort recursively on both partitions if the partitions are not empty.

Many of these steps describe sub-problems that could be solved in many different ways; consequently, many of them could be written as functions that are called from the quicksort algorithm.

```
// The argument 'array' is an array of entries from a to b - 1
// Those entries are sorted so that array[k] <= array[k + 1] for k = a, ..., b - 2
```

```
void quicksort( int *array, int a, int b ) {
    if ( b - a < 20 ) {
        insertion_sort( array, a, b );
    } else {
        pivot = find_pivot( array, a, b );

        int low = a + 1, high = b - 2;

        while ( true ) {
            low  = find_next( array, pivot, low, b );
            high = find_previous( array, pivot, a, high );

            if ( low < high ) {
                swap( array, low, high );
            } else {
                break;
            }
        }

        reinsert_pivot( array, pivot, high, b );

        quicksort( array, a, high );
        quicksort( array, high, b );
    }
}
```

Now, each of the sub-problems may be solved in a similar manner: describing the form of the input, and what must be performed in order to solve the sub-problem. This can be repeated as often as necessary.

In order to make software development reasonable and tractable, in general, a function should solve one and only one problem, and any sequence of steps that could collectively be described as solving a well identifiable sub-problem should be factored out into a function. The benefits include:

1. easier maintenance,
2. the ability to quickly switch algorithms (for example, using insertion sort instead of quicksort in very specific applications),
3. allows easier division of labor—different teams of developers can be assigned very specific tasks, and
4. if the sub-problem appears elsewhere, the same solution can be used in both locations to solve both problems (reducing costs of development, testing and maintenance).

A number of rules from the JPL coding standard apply to the procedural approach to programming, including Rule 25::

> "Functions should be no longer than 60 lines of text and define no more than 6 parameters."

and including Rule 14:

> "The return value of non-void functions shall be checked or used by each calling function, or explicitly cast to (void) if irrelevant."

While not used in embedded or real-time systems, the printf command for displaying text on the console does have a return value: an integer equal to the number of characters that were printed. A failure to print to the screen would result in either 0 (nothing was printed) or a number smaller than the expected number of characters. In most cases, programmers are not interested in the return type of printf, and thus, following this standard, we would see code such as

```
(void) printf( "Hello world!\n" );
```

indicating to the reader that there is a return value, but that that return value is being explicitly ignored.

2.1.1.3 Data abstraction

Another concept in software engineering is that of data abstraction, or *abstract data types* (ADTs). At the basic level, designers think of integers, real numbers, alphabets, Boolean values and references to other objects. Programming languages provide primitive data types that represent these values, and the programmer must choose the appropriate type that sufficiently approximates the abstract concepts. As primitive data types such as `short`, `int` and `long` all use fixed amounts of memory, they can only represent integers on a finite range, while `float` and `double` represent real numbers with different amounts of precision (the former being useful only for graphics and the later sufficient for scientific computation). The `char` represents a very small ASCII alphabet with 256 characters while Unicode allows the representation of significantly larger alphabets and writing systems (ԿԽ, բարեւ Ձեզ, হ্যালো, 您好, მიესალმები, γεια σας, हेलो, שלום, नमस्ते, こんにちは, ಹಲೋ, ជំរាបសួរ, 안녕하세요, ສະບາຍດີ, ഹലോ, हेलो, सत श्री अकाल, здравствуйте, ආයුබෝවන්, ഹലോ, హలో, สวัสดี and hello).

More complex abstractions are described as compositions of more primitive types. Designers only need then to discuss the problem at the appropriate level of abstraction. For example, in arranging plans for an evening with your friends, you us simply *message* them. There is no need to understand the underlying implementation, you need to understand the interface provided by your computer or smart phone. Similarly, software engineers will think in terms of abstractions, such as lists, queues, stacks, sorted lists and priority queues. Software engineers then design data structures or data types that implement these abstract concepts. A sorted list that is not likely to change is best stored as an array, but one that is to be modified is better stored as a search tree. Whether or not a B+-tree or an AVL tree or a red-black tree is used for a sorted list depends on the requirements, just like whether a binary heap or a leftist heap is used for a priority queue, again, depends on the requirements. No data structure is ideal of all situations and it is up to the designer to choose the appropriate representation.

The ADTs you have seen prior to this course include

1. sets (a collection of unique objects),
2. bags (sets with repetition),
3. lists (an explicitly ordered sequence of items),
4. strings (an ordered sequences of characters from an alphabet),
5. stacks (last-in—first-out),
6. queues (first-in—first-out),
7. sorted lists (an implicitly ordered sequence of items),
8. priority queues (highest-priority-first) and
9. graphs (vertices and edges).

Lists can be implemented using linked lists or arrays, and we will refer to the first entry as the *front* and the last entry as the *back*. This is an appropriate time to discuss terminology we will use throughout this text.

	Insert	**Remove**	**Next or First**	**Last**
Stack	push	pop	top	bottom
Queue	enqueue	dequeue	head	tail
Linked list	push front or back	pop front or back	front	back

Different texts will use different terminology, including *pushing* and *popping* from queues.

2.1.1.4 Object-oriented programming

Object-oriented (OO) programming is usually built on top of the procedural programming paradigm. This combines data abstraction with the procedural problem-solving paradigm. The characteristics of an OO language are

1. encapsulation,
2. inheritance, and
3. polymorphism.

To summarize what you have seen previously:

1. In an object-oriented programming language, the focus of the design is on *encapsulated* and related data and the operations and queries that can be performed on that data in a structure usually referred to as a *class*.
2. In addition to the relationship between the data stored within a class, it is also possible to specify ordered relationships between classes where one class is said to be *derived* from another if it contains a superset of the data stored in the *parent* class (the derived class *inherits* the data of the parent class) and that in addition to possibly including new operations and queries, operations and queries in the parent class may be either *inherited* or may be redefined (*overwritten*).
3. This inheritance relationship defines either a partial order (resulting in a directed acyclic graph (DAG) of classes) or a hierarchical ordering (defining a tree of classes). When an operation or query is made on a particular object, it traces back from the location of the class in the DAG or tree until it finds the first redefinition or the original implementation of the operation or query, whichever is first, a behavior described as *polymorphism*.

Object-oriented programming became adopted in larger software projects as the focus is on well-defined collections of data and operations that can be performed on that data. One issue OO languages is that there is a computational overhead in implementing polymorphism. For example, Java implements all three aspects, but polymorphism is applied in C++ only through the use of the `virtual` keyword.

Note: private members are not immune from malicious attacks. Consider the following C++ code:

```
class X {
    private:
        int x;
    public:
        X( int xp ):x( xp ) {}
        void get_x() { return x; }
};

int main() {
    X a( 3 );

    std::cout << a.get_x() << std::endl;    // This prints the initial value, 3

    int *p_ax = reinterpret_cast<int *>( &a );
    *p_ax = 5;

    std::cout << a.get_x() << std::endl;    // The value is now 5

    return 0;
}
```

All encapsulation does is prevent honest programmers from accidently accessing or modifying the internal structure of a class.

2.1.1.5 Design patterns

The concept of a *design pattern* was adopted from architecture: well defined solutions to common and reoccurring problem arising in the field. Computer architecture and software designers have compiled numerous patterns for which there are recognized reusable and efficient solutions. The benefit of design patterns is that there are often numerous other means of solving such problems, each of which have negative characteristics.

One such design pattern is a *singleton*, a class for which there is only ever one instance of that class. One may consider many ways of implementing such a class, but the most reasonable C++ implementation is as follows:

```
class Singleton {
    private:
        static Singleton *p_instance;
        Singleton() { }

    public:
        static Singleton *get_instance();
};

Singleton *Singleton::p_instance = nullptr;

Singleton *Singleton::get_instance() {
    if ( Singleton::p_instance == nullptr ) {
        Singleton::p_instance = new Singleton();
    }

    return Singleton::p_instance;
}

int main() {
        Singleton *ptr = Singleton::get_instance();

        return 0;
}
```

Thus, the only way to access the single instance of the class is to call the `get_instance` member function, and as the constructor is private, only the member functions defined in this class can access and assign to that member variable. Without encapsulation, this cannot be done securely in C as it can be in C++. A summary of design patterns in Gamma et al. is made available by Jason McDonald at his web site[3]. Some patterns that are possibly of interest to mechatronics students are listed in the following table.

Chain of responsibility	Avoid coupling the sender of a request to its server by giving more than one server a chance to handle the request. Chain the receiving servers and pass the request along the chain until a server handles it. For example, a request could be sent to a chain of robots and the first one available would service the request.
Observer	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Iterator	A mechanism for providing access to the entries of a container without exposing the underlying implementation of that container. For example, is an implementation of a list ADT using an array, a linked list or a linked list of arrays?
Proxy	Suppose means of storing some (usually large) components of an object in secondary memory as opposed to in main memory, loading the component only when required.
Abstract factory	A means of creating instances of related objects without specifying the actual class to which it belongs. For example, you may have to deal with different graphical user interfaces (GUIs) in different systems. Each will have classes for windows and other graphical widgets, but you would rather not have to declare two constructions of essentially the same display.

[3] http://www.mcdonaldland.info/files/designpatterns/designpatternscard.pdf

2.1.1.6 Summary of programming paradigms

We have described structured programming and the procedural programming paradigm of C. This was followed by the concept of data abstraction and its central role in object-oriented programming in Java. The C++ and Ada programming languages contains elements of both procedural and object-oriented programming in their design philosophies—you can easily use both approaches. We finished with the concept of design patterns—recognized solutions to common problems—and described some examples that may be applicable to engineering disciplines in general. There are other programming paradigms, including functional, logic and aspect-oriented programming, but most embedded systems use (as this book will) structured procedural programming with data abstraction and encapsulation with design patterns indicating best practices.

2.1.2 Ideal characteristics of programming languages

Not all programming languages allow the same ease of performing certain tasks; each enforces certain programming practices on the code being generated. Consequently, there are programming languages that may be ideal for one type of problem, while other programming languages may be better suited to other problem domains.

For example, Matlab is an excellent programming language for solving problems involving linear algebra, while Maple is excellent for solving symbolic mathematical problems.

We will look at the desirable characteristics for programming

1. real-time systems,
2. embedded systems, and
3. operating systems.

2.1.2.1 Characteristics of a real-time programming language

Ideally, a real-time programming language will have mechanisms built into its design in order to facilitate

1. data encapsulation,
2. exception handling,
3. synchronization (including mutual exclusion and serialization),
4. concurrency, and
5. message passing.

The Ada programming language was designed in the 1970s specifically with these goals in mind. At the other end of the spectrum, none of these concepts are built into the C programming language: if any are to be used, they must be built into libraries which are then called through appropriate functions. Other real-time programming languages include Modula and Modula-2, and there is a real-time specification for Java, a programming language that implements the first four of the above five mechanisms.

If production and maintenance costs, rather than performance, are of primary concern as, a programming language like Java is likely appropriate. After all, if a toaster fails to pop up the toast after 40 seconds and does so after 41 seconds, it is hardly a tragedy. There is a real-time specification for Java (RTSJ) and this environment has been recently used for numerous aspects of the South Korean T-50 trainer jet, shown in Figure 2-2. RTSJ is used in the multi-function display set, the heads-up display, the mission computer, the mission planning and support system, fire control, as well as other components.

Figure 2-2. The T-50 trainer (photo by Kentaro Iemoto).

As another alternative, Ada is a programming language resulting from an initiative by the United States Department of Defense. It is a programming language designed from the ground up to support concurrency (for example, tasks and synchronous message passing) and to work in real-time and embedded environments. As another example, routines that are designed to run in parallel are described as *tasks*, and *functions* are routines called by executing tasks. In C, both are implemented as functions, where concurrency is achieved by simply telling the appropriate library to "begin executing this function as if it was a parallel task". In Ada, tasks have separate declarations from functions—you cannot accidently call a task from a function, and you cannot accidently start executing a function as a parallel task.

Despite the increase of popularity of object-oriented programming languages, their overhead can be detrimental. Specifically, the aspects of inheritance and polymorphism, and also function encapsulation (the overhead of which can be ameliorated but not entirely eliminated with inline) results in unpredictable and inefficient systems. The problem is dramatically worse in systems with garbage collection; the garbage collector runs whenever the system decides it is appropriate. This is, from the perspective of the programmer, utterly unpredictable. However, object-oriented programming languages can be recommended for soft and firm real-time systems.[4] Of course, in a system such as the T-50 trainer, the cost of processors that are perhaps 100 % (or more) faster is probably insignificant when contrasted with the possible savings in software development and maintenance in using a language such as Java. That is to say, sometimes the cheaper decision is just to buy more or better hardware.

In an anecdote replayed by Laplante, a real-time system was developed using C++ at the insistence of the design team. The system was developed and was functional; however, upon the inclusion of additional features, it was found to be impossible to meet specified deadlines. One client engaged an outside vendor to implement the system in C together with hand-optimized assembly language for the most critical sections. The low-level procedural correspondence between C and assembly allowed this to occur, whereas this was not possible with the higher-level object-oriented approach of C++. However, individual data points should not be used to draw sweeping conclusions.

2.1.2.2 Characteristics of an embedded systems programming language

Programming languages for embedded systems, in general, must produce compact and efficient code. They must allow for access to peripherals and exhibit close ties to the underlying hardware. Assembly language as well as C and its descendants are well suited for embedded systems.

[4] Laplante and Ovaska, p.165.

With the focus of a language such as C on procedural programming, it is possible to include inline assembly instructions; for example, taken from the Keil web site,

```
extern void test();

void main( void )  {
    test ();

#pragma asm
    JMP   $  ; endless loop
#pragma endasm
}
```

Thus, while other alternatives exist, C is still a ubiquitous programming language for embedded systems. There is even an extension to the C language, *Embedded C*, that is useful for writing embedded code (see http://www.engineersgarage.com/tutorials/emebedded-c-language).

Additionally, embedded systems tend to be smaller than many applications, and therefore are still manageable without much additional overhead. Consequently, the framework provided by higher level languages like C++ may not be worth the additional costs. The C programming language was designed to write operating systems, and many of the structures we will examine will have parallels in operating systems; it has even been described as a "portable assembly language". Consequently, it is appropriate at this level.

Every year, VDC Research polls embedded systems developers as to which programming languages they use to develop such systems. Normally, they document all programming languages used, however, in 2014 they contrasted the change from 2008 to 2013. Java and C#, each running on virtual machines, are seeing significantly more prominence, while the workhorses of embedded systems, C and Assembly, are seeing a decrease in market share (see Figure 2-3). Recall that, at best, virtual machines run similar code with a slow-down of at least 300 %.

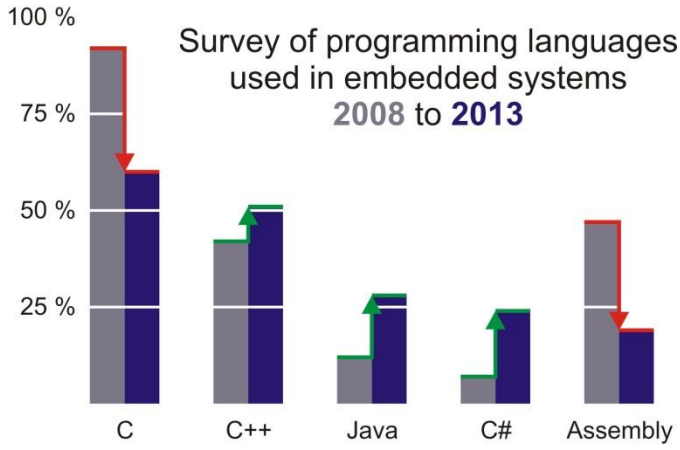


Figure 2-3. Percent of programming languages used in embedded systems
(with multiple responses, totals are greater than 100 %). Recreated from
http://electronicdesign.com/embedded/developers-discuss-iot-security-and-platforms-trends.

2.1.2.3 Characteristics of an operating system programming language

The C programming language was designed to implement the kernel and other components of the Unix operating system. The C programming language was originally written to allow Unix to be portable across many platforms; consequently, many of its design decisions were made in order to allow it act as an abstraction of a processor. Additionally, compiled C tends to be more compact than other programming languages. In a sense, the original C compliers were essentially interpreters. For example, the reason C has auto-increment (`++i` and `i++`) and auto-decrement (`--i` and `i--`) operators

is that there are assembly language instructions and related flags for these very operations, especially when associated with array indexing. Otherwise, there would be no point to these, and we would be left with

```
i = i + 1;    // ++i;
i = i + a;    // i += a;
```

Most languages that do not descend from C do not support such operators on the argument that they don't add anything to the programming language—they really are cues for the compiler.

Of course, another part is sheer momentum: experienced C programmers are more readily available than, for example, ESL (embedded systems language) programmers.

Why not use C++, as data encapsulation and exception handling are built into the language? While an appeal to authority can sometimes form the basis of a fallacious argument, the following quote by two Linux kernel developers, Linus Torvalds and Richard Gooch, respectively, seem appropriate:

> "*Trust me: writing kernel code in C++ is a bloody stupid idea. The fact is, C++ compilers are not trustworthy. The whole C++ exception handling thing is fundamentally broken. It's especially broken for kernels. Any compiler or language that likes to hide things like memory allocations behind your back just isn't a good choice for a kernel.*"

> "*My personal view is that C++ has its merits, and makes object-oriented programming easier. However, it is a more complex language and is less mature than C. The greatest danger with C++ is in fact its power. It seduces the programmer, making it much easier to write bloatware. The kernel is a critical piece of code, and must be lean and fast. We cannot afford bloat. I think it is fair to say that it takes more skill to write efficient C++ code than C code. [Developers] will not know the various tricks and traps for producing efficient C++ code.*"

As an example, the first author of this text implemented the Dormand-Prince algorithm for approximating solutions to systems of initial-value problems. This algorithm is adaptive, so the number of steps (the size of the output array) is not known prior to completing the algorithm. When implemented in Matlab, all memory allocation and deallocation is performed by the Matlab interpreter. In C++, the first solution is to use the Standard Template Library (STL) `vector` class; however, in C, no such structure exists, so it was necessary to come up with an appropriate intermediate structure: in this case, a linked list of arrays. Then, only at the end, was a pass made to copy all data in these arrays into a single array of the appropriate size. The C implementation was significantly faster than the C++ version using the vector class, and twenty times faster than the Matlab version.

2.1.2.4 Summary of ideal characteristics

The C programming language is appropriate for embedded and operating systems, but it lacks the desirable characteristics for real-time systems. Never-the-less, it is still the most common programming language used in such situations, and we will consequently use it in this class. In time, it seems that object-oriented programming languages such as C++ and Java (languages that impose a layer of *data abstraction* on the procedural programming paradigm used by C) will become dominant. C++ is likely more appropriate when tighter code is required while Java is available if minimizing development and maintenance costs takes highest priority.

2.1.3 Other software programming techniques

As described previously, an object-oriented programming language is often described as implementing data structures that include

1. encapsulation (data hiding),
2. inheritance, and
3. polymorphism.

In fact, object-oriented programming actually deals more with the latter two concepts. Encapsulation is a software programming technique used to ease maintenance. We will discuss through this course how encapsulation can be used as a technique of disciplined programming that will help you develop maintainable code, even if it is not enforced by the programming language using visibility specifiers such as *public*, *protected* and *private*. Note that inheritance and polymorphism are not required characteristics of a real-time programming language listed at the start of Section 2.1.2.1.

Other software techniques applicable to embedded systems include

1. abstraction and
2. modularity.

Abstraction separates the concepts or ideas regarding the functionality from the concrete implementations. For data structures, the implementation is hidden behind an interface used by programmers so that the programmer can use, for example, a stack to provide the expected behavior without worrying about the details. Systems can have abstraction layers where, for example, a network can be divided into multiple layers where programmers of each layer need only understand the interface of the immediate adjoining layers. Consider the Open Systems Interconnection (OSI) model of a network:

1. application,
2. presentation,
3. session,
4. transport,
5. network,
6. data link, and
7. physical.

An application need only understand the interface of the presentation, and at each step, the message and its address is appropriately packaged and modified until it is finally sent to its destination on the physical network. At the other end, the application merely accepts the package in an appropriate form as returned by a function in the presentation layer interface.

Modular programming involves the separation of the functionality of a system into self-contained, independent and interchangeable modules. Each module contains only the functionality required to execute the desired level of abstraction. This allows a *separation of concerns*, where the programmer of one module need not concern him or herself with the details of the other modules (very helpful in 4^{th}-year design projects). We will be using modularity in this course, too. The tasks that are necessary to control and allocate resources on a computer can be broadly broken into categories to deal with ideas such as

1. dynamic memory allocation,
2. task execution and scheduling,
3. synchronization,
4. message passing, and
5. file systems.

For example, in Linux, it is possible to swap different modules for any of these required categories. The default scheduler in Linux (we will look at this later), is not a real-time scheduler; however, you can swap that scheduler out and install a module with a real-time scheduler (one that selects the next task to execute in $\Theta(1)$ time).

2.1.4 Summary of real-time programming languages

C is more desirable as an operating system and embedded systems programming language as opposed to a real-time programming language; however, the availability of appropriate libraries, the fact that C programmers are ubiquitous, and the availability of compilers on all platforms, make it the *de facto* programing language for many embedded systems. This does not mean that it is a *better* programming language; Ada is more appropriate, it is simply less accessible. We also discussed other software techniques applicable to designing embedded systems such as the applicability of object-oriented design techniques and abstraction and modularity.

2.2 The C programming language

We will now continue with looking at many of the features in the C programming language and compare and contrast them with characteristics of the C++ programming language you have already learned. We will see:

1. C does not have classes, only structures,
2. it is possible to do generic programming in C, only it is less safe or very complicated,
3. a discussion of pass-by-reference in C,
4. it is possible to emulate object-oriented programming without the encapsulation provided in C++,
5. it has header files that are important to understanding a system,
6. the functioning of the pre-processor,
7. there is a relationship between the order of structures and memory allocation,
8. there is both explicit memory allocation and deallocation,
9. bit-wise operations,
10. bit-fields in C99,
11. µVision4 specifics,
12. there are other places to find help.

These topics are discussed here, but let us start with a cartoon from XKCD (http://xkcd.com/371/ used for academic purposes):

2.2.1 No class, just structure…

C++ is an object-oriented language and therefore the primary mechanism for creating aggregate types is the class. A class is a collection of data (member variables) associated with a collection of functions that operate on that data (member functions). The interface is usually through public member functions while the actual implementation is hidden behind an opaque barrier of private and protected member variables and functions. There are many good reasons to use objects; however, that and other features of C++ are not necessarily the most appropriate for embedded systems.

As a brief review of C++, in C++, we have global and member functions, the second being associated with a class. There are also global, local and member variables, the second being associated with a function calls and the third being associated with instances of classes.

```
// Global variables and functions

class Class_name {
    private:
        // Private member variables (member variables are usually private)
        // Private member functions (helper functions)
    public:
        // Public member functions (interface)
};

// A global function
int main( void ) {
    // Local variables
}
```

A function or variable that is shared is said to be *static*. For example:

1. A static local variable is one that is shared by all calls to that function,
2. A static member function or member variable of a class is one that is not bound to any specific instance of that class.

Think of static variables and functions as global variables and functions that can only be accessed in the associated function or class.

```
// A global function
void f( void ) {
    // The number of times this function is called
    static int size = 0;
    ++size;
}

class Class_name {
    private:
        // Private member variables (member variables are usually private)
        // Private member functions (helper functions)
    public:
        // A static member variable, or "class variable"
        static double PI = 3.1415926535897932385d;
};
```

In C, there are no classes with member variables and member functions, only structures and functions. A `struct` is a `class` without any visibility restrictions, without associated member functions (and therefore without polymorphism) and without inheritance.

```
struct pair {
       int first;
       int second;
};

struct single_node {
    void               *p_entry;
    struct single_node *p_next;
};
```

The member variables of the structure are called *fields* and instances of structures are commonly referred to as *records*. Unlike C++ classes, where you can declare an instance of a class by just using the class name, in C, you must always use the `struct` modifier, as is demonstrated by the pointer to the next node in the single node structure.

```
struct pair coordinate;
struct single_node new_node;
```

To simplify this, C uses a concept known as *type definitions* (`typedef`) that allows you to use this definition in place of the full type name or description; for example,

```
typedef struct {
    int first;
    int second;
} pair_t;

typedef struct single_node {
    void               *p_entry;
    struct single_node *p_next;
} single_node_t;
```

By convention, types in C are suffixed with an `_t`. Recall that a singly linked list (or *single list* for short) usually consists of a head pointer (storing the address of the first node), a tail pointer (storing the address of the last node as a means of optimizing insertions at the end of the linked list), and a counter. Here is a single list structure:

```
typedef struct {
    single_node_t *p_head;
    single_node_t *p_tail;

    size_t size;
} single_list_t;
```

Note: for any container, *size* will refer to the number of items that the container is holding, while *capacity* will refer to the maximum number of items that the container can hold.

Here we have a pre-defined type `size_t` (defined in `stddef.h`) that is used to store the number of entries in the linked list. The type `size_t` is an unsigned integer able to store a number sufficiently large to capture the maximum size of an object. This would be at least 2 bytes on a 16-bit computer, 4 bytes on a 32-bit computer and 8 bytes on a 64-bit computer. The use of `size_t` eliminates potential portability problems.

2.2.2 More than the sum of its parts

Consider the following structure:

```
#include <stdio.h>

struct demo {
    char a;
    int b;
    char c;
    short d;
    char e;
    int f;
    long g;
};

int main() {
        struct demo x;

        printf( "%p   %d\n", &x,     sizeof( x )   );
        printf( "%p   %d\n", &(x.a), sizeof( x.a ) );
        printf( "%p   %d\n", &(x.b), sizeof( x.b ) );
        printf( "%p   %d\n", &(x.c), sizeof( x.c ) );
        printf( "%p   %d\n", &(x.d), sizeof( x.d ) );
        printf( "%p   %d\n", &(x.e), sizeof( x.e ) );
        printf( "%p   %d\n", &(x.f), sizeof( x.f ) );
        printf( "%p   %d\n", &(x.g), sizeof( x.g ) );
        return 0;
}
```

Consider the output:

```
0x7fffbcc535e0   32
0x7fffbcc535e0   1
0x7fffbcc535e4   4
0x7fffbcc535e8   1
0x7fffbcc535ea   2
0x7fffbcc535ec   1
0x7fffbcc535f0   4
0x7fffbcc535f8   8
```

Why is this true if 1 + 4 + 1 + 2 + 1 + 4 + 8 = 21? This is a consequence of the compiler trying to optimize access time to memory. While memory is byte addressable, most computers will read multiples of bytes, or *words*, and a *word boundary* will be at multiples of the word size, so if a field spans one of these boundaries, it will require two fetches to access it, as opposed to one. The compiler option `gcc -fpack-struct` will minimize the space required by the structure:

```
0x7fffbcc535e0   21
0x7fffbcc535e0   1
0x7fffbcc535e1   4
0x7fffbcc535e5   1
0x7fffbcc535e6   2
0x7fffbcc535e8   1
0x7fffbcc535e9   4
0x7fffbcc535ed   8
```

These two memory maps are shown in Figure 2-4.

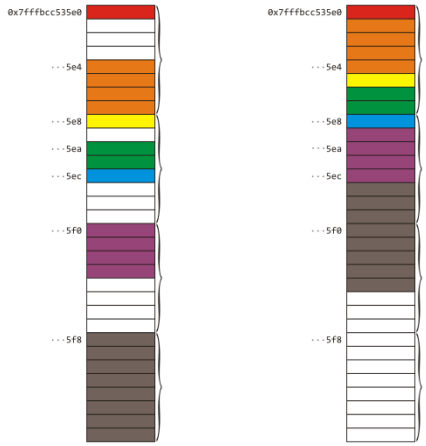


Figure 2-4. Memory map of a structure: one optimized and the other packed.

The justification for this layout deals with the design of the data bus (the connection between the computer and main memory), a topic that we will see in the next chapter.

2.2.3 Generics in C

One point you may have noticed above is that we don't have anything in C that resembles templates from C++ or generics in Java. Instead, we are forced to create data structures that simply store addresses where the type is left unspecified, namely `void *`. The notation is slightly confusing for a new C programmer, as

```
void f() { … }
```

defines a function that does not have a return value, but

```
void *f() {…}
```

is a function that returns a pointer where the type of that pointer is unspecified (that is, it is just a 16-, 32- or 64-bit address, depending on the system)—it is not a pointer to nothing.[5] We will look at using the pre-processor later.

2.2.4 Static and dynamic memory allocation

Now, we must consider the difference between the following two declarations:

```
single_list_t sl;
single_list_t *p_sl = (single_list_t *) malloc( sizeof( single_list_t ) );
```

In the first case, the compiler knows the size of a single list and allocates the appropriate amount of memory (somewhere). In the second case, the compiler knows the size of a pointer (4 bytes on a 32-bit processor and 8 bytes on a 64-bit processor, and of course, 2 bytes on a 16-bit processor and 1 byte on an 8-bit processor). The memory for the second single list must later be returned to the operating system. To return the memory, call

```
free( p_sl );
```

Memory is *byte addressable*. This means that each byte has a unique address and if you want to access or change a single bit, you must first access the byte containing that bit. If the bit is changed, then the entire byte must be written back to the address.

[5] There is another alternative, colloquially referred to as *hacking the pre-processor*. If you're interested, please read Andrei Ciobanu article at http://andreinc.net/2010/09/30/generic-data-structures-in-c/ for a brief introduction.

Why byte addressable? Why not 7-bit addressable or 6-bit addressable? This largely historical:

1. 7 bits is sufficient for coding the English alphabet with one parity bit,
2. 8 bits is sufficient for coding European languages (ä, ö, ü, ß, é, â, ç), and
3. 256 colors is usually sufficient for a gray-scale image.

Later, we will see the concept of block addressability. For example, on a hard drive, each block (usually 4 KiB, but possibly smaller or larger) has its own address and no more. If you want to access a byte on a hard drive, you must first load the block containing that byte into the hard drive, modify the byte, and if necessary write the entire block back to the hard drive.

The next question is how many addresses are there? In general, an address will be a multiple number of bytes. An *n*-bit computer will be able to address 2^n unique bytes. Consequently:

1. A 8-bit processor will be able to access $2^8 = 256$ bytes,
2. A 16-bit processor will be able to access 2^{16} bytes or 64 KiB,
3. A 32-bit processor will be able to access 2^{32} bytes or 4 GiB, while
4. A 64-bit processor will be able to access 2^{64} bytes or 16 million TiB.

Note: $2^{10} = 1024 \approx 1000 = 10^3$. Thus, $2^{32} = 2^2 \times 2^{30} = 2^2 \times (2^{10})^3 \approx 4 \times 1000^3 = 4$ billion.

It is colloquial to call 2^{10} as one kilobyte (kB) and 2^{32} as four gigabytes using metric prefixes; however, these are powers of 10, not powers of 2. Consequently, I will use 10 kB and 2 GB to represent 10 000 bytes and 2 billion bytes, respectively, while 10 KiB and 2 GiB to represent 10 240 and 2^{31} bytes, respectively.

Now, let's observe something interesting:

```
#include <stdlib.h>
#include <stdio.h>

int main( void ) {
    int exit_value;

    int m = 4;

    int *p_n = (int *) malloc( sizeof( int ) );

    if ( p_n == NULL ) {
        exit_value = EXIT_FAILURE;
    } else {
        *p_n = 5;

        printf( "The address of 'm':          %p\n", &m   );
        printf( "The value of 'm':            %d\n",  m   );
        printf( "The address of 'p_n':        %p\n", &p_n );
        printf( "The value of 'p_n':          %p\n",  p_n );
        printf( "The value stored at 'p_n':   %d\n", *p_n );

        free( p );

        exit_value = EXIT_SUCCESS;
    }

    return exit_value;
}
```

When we compile and run this on a 32-bit computer, we get the output:

```
$ gcc example.c
$ ./a.out
The address of 'm':          0xfffffd73c
The value of 'm':            4
The address of 'p_n':        0xfffffd730
The value of 'p_n':          0x16d1a38
The value stored at 'p_n':   5
```

You will notice a few things here:

1. The local variables are stored close to the *end* of memory,
2. The local variables are stored next to each other, but
3. The memory allocated by `malloc` is somewhere else.

You will also recall that `malloc` must find and allocate the memory so that no one else can either overwrite it, or even view it.

In a few topics, we will look at how understanding how the program works, what the operating system does, and discover a few things about operating systems.

Thus, the programmer must be aware of how much memory is required. This introduces the unary operator `sizeof( datatype )` which will return the memory required by the given data type. For example,

1. `sizeof( int )` usually equals 4 (representing four bytes),
2. `sizeof( float )` always equals 4 (representing eight bytes),
3. `sizeof( double )` always equals 8 (representing eight bytes), and
4. `sizeof( single_node_t )` (comprised of two pointers) equals 8 on a 32-bit machine (every pointer is 4 bytes) and equals 16 on a 64-bit machine (every pointer is 8 bytes).

Note, the only requirement in the specification is that the following must be true:

```
2 <= sizeof( short int )
  4 <= sizeof( int ) && sizeof( short int ) <= sizeof( int )
       sizeof( int ) <= sizeof( long int )
  sizeof( long int ) <= sizeof( long long int )
```

Aside: Note that `sizeof` is an operator, not a function. It must be able to determine the size of the type at compile time. This is slightly confusing, as `sizeof int` is invalid—one must use `sizeof( int )`—but `return 0` is just as valid as is `return( 0 )`.

This ambiguity as to how large various integer data types are has led many lower-level tools and utilities to create a set of specified types:

```
typedef signed char   S8;          typedef char               U8;
typedef short         S16;         typedef unsigned short     U16;
typedef int           S32;         typedef unsigned int       U32;
typedef long long     S64;         typedef unsigned long long U64;
```

If we only use these defined types, `S8` through `U64`, then if we port our code to a different compiler where, perhaps `char` is signed by default, we would only have to change the first line. Common alternate type definitions include:

```
typedef signed char    int8_t;     typedef char                uint8_t;
typedef short         int16_t;     typedef unsigned short     uint16_t;
typedef int           int32_t;     typedef unsigned int       uint32_t;
typedef long long     int64_t;     typedef unsigned long long uint64_t;
```

These are defined in the header file `stdint.h`, together with other useful definitions, including for example

```
#define        INT8_MAX     0x7f
#define        INT8_MIN    ( -INT8_MAX - 1)
#define        UINT8_MAX   (2*INT8_MAX + 1)

#define        INT16_MAX     0x7fff
#define        INT16_MIN    (           -INT16_MAX -     1 )
#define        UINT16_MAX   (2U*__CONCAT(INT16_MAX, U) + 1U)
```

This is emphasized in Rule 17 of the JPL coding standard, which says that

> "typedefs that indicate size and signedness should be used in place of the basic types."

2.2.5 Pass-by-reference in C

The C programming language does not allow pass-by-reference. Consequently, the following C++ example cannot be written in C:

```
void increment( int &n ) {
    ++n;
}
```

If we write

```
void increment( int n ) {
    ++n;
}
```

and call this with

```
int i = 5;
increment( i );
```

this, does not change the value of the argument `i`, as the value of the argument is copied to the parameter `n`. While the parameter is changed, the original argument is left unchanged.

Instead, we can solve this by passing the address of the object to be changed, for example

```
void increment( int *p_n ) {
    ++(*p_n);
}
```

and call this with

```
int i = 5;
increment( &i );
```

As a first approximation, any pass-by-reference in C++ can be converted into a pass-by-value in C by:

1. replacing the `&p` in the formal parameter with `*p`,
2. replace any instance of the actual parameter `p` in the function call with `*p`, and
3. replace any arguments `q` with `&q`.

The benefits of pass-by-reference in C++ do however include

1. transparency (you don't have to explicitly use `&q` in the calling sequence),
2. temporary objects can be passed (those created but not assigned to local variables), and
3. it is easier to work with references and therefore it is less likely to result in bugs (is, for example, `*n++` the same as `(*n)++` or `*(n++)`?).

The following swaps two 32-bit integers:

```
void swap( U32 *p_a, U32 *p_b ) {
    U32 t;

     t = *p_a;
    *p_a = *p_b;
    *p_b =  t;
}
```

The following swaps two arbitrary sized objects:

```
void swap( void *p_a, void *p_b, size_t n ) {
    int i;
    char t;
    char *p_char_a = (char *) a;
    char *p_char_b = (char *) b;

    for ( i = 0; i < n; ++i ) {
        t = p_char_a[i];
        p_char_a[i] = p_char_b[i];
        p_char_b[i] = t;
    }
}
```

Note: if you want to pass a pointer by reference, you would use

```
typename **pp_obj;
```

In the calling function, you would pass the address of the pointer, and in the function updating the parameter, you would assign a pointer to `*pp_obj`. Note that Rule 26 of the JPL coding standard says that

"The declaration of an object should contain no more than two levels of indirection."

Thus, a declaration such as

```
typename ***ppp_obj;
```

meaning the address of the pointer that stores the address of a pointer to an object, should generally not be used. If you dealing with the address of a pointer storing the address of a two-dimensional array (a matrix), this suggests it would be clearer to define the two-dimensional array within a structure, and to then pass the address of the pointer to the structure. This would significantly clarify code for the reader.

2.2.6 An object-oriented approach in C

Let's start writing a function to work on a singly linked list as if it was in C++. Let's start with the push front function:

```
int push_front( void *p_new_entry ) {
    single_node_t *p_new_node = (single_node_t *) malloc( sizeof( single_node_t ) );
```

The compiler does not allow us to arbitrarily assign pointers to different objects being assigned without explicitly telling the compiler that that is what we want to do, consequently, we must *cast* the returned pointer from malloc as a pointer to a single node: `(single_node_t *)`.

Next, we must initialize the fields:

```
    p_new_node->p_entry = p_new_entry;
    p_new_node->p_next = ...
```

Normally, we would assign the next pointer of the new node to be placed at the front of the linked list to be address of the node currently at the front of the linked list, but which linked list?

In C++, when a member function is called on an instance, the address of the object it is called on is implicitly passed as the pointer `this`. In this case, however, we have no such luck: we must explicitly pass the address of the object.

```
bool single_list_push_front( single_list_t *const p_this, void *p_new_entry ) {
    bool success;
    single_node_t *p_new_node = (single_node_t *) malloc( sizeof( single_node_t ) );

    if ( p_new_node == NULL ) {
        // No memory...
        success = false;
    } else {
        p_new_node->p_entry = p_new_entry;
        p_new_node->p_next = p_this->p_head;
        p_this->p_head = p_new_node;

        if ( p_this->size == 0 ) {
            p_this->p_tail = p_new_node;
        }

        ++( p_this->size );
        success = true;
    }

    return success;
}
```

As we do not have the new operator, which automatically calls a constructor, we may have to do our own initialization. This is often done with an `init()` function that must be called separately:

```
void single_list_init( single_list_t *const p_this ) {
    p_this->p_head = NULL;
    p_this->p_tail = NULL;
    p_this->size = 0;
}
```

We would now do the following:

```
int main( void ) {
    single_list_t sl;
    single_list_init( &sl );

    // Use the single list with, for example, single_list_push_front( &sl, ... );

    return EXIT_SUCCESS;
}
```

or

```
int main( void ) {
    single_list_t *p_sl = (single_list_t *) malloc( sizeof( single_list_t ) );
    init( p_sl );

    // Use the single list with, for example, push_front( p_sl, ... );

    free( p_sl );

    return EXIT_SUCCESS;
}
```

Note: this brings us to another convention. You will note that we use

```
single_list_t *p_sl;
```

and we do not use either of

```
single_list_t* p_sl;
single_list_t * p_sl;
```

The first of these alternates would make the most sense: "`p_sl` is a *pointer to a single list*". Unfortunately, this suggests that `single_list_t*` is a type, and it is not:

```
single_list_t* p_sl1, p_sl2;
```

declares `p_sl1` to be a pointer, but `p_sl2` to be simply a single list.

Thus, we will read my convention as "`p_sl` is a *pointer* that stores the address of a single list". This is the same notation used in the Keil operating system.

If all instances of a class are to be allocated dynamically, we could combine both memory allocation and initialization into a single function:

```
single_list_t *single_list_alloc() {
    single_list_t *p_list = (single_list_t *) malloc( sizeof( single_list_t ) );

    p_list->p_head = NULL;
    p_list->p_tail = NULL;
    p_list->size = 0;

    return list;
}
```

This will not work, however, if any single list is to be declared statically (either as a global or local variable).

2.2.7 Header files

Up until now, you've dealt with a header file and a source file. A few comments on terminology:

1. The signature of a function is called a *declaration*, while
2. The signature together with a function body is a *definition*.

In a project, you may have a number of source files with associated header files, such as:

my_module.h

```
#ifndef CA_UWATERLOO_DWHARDER_MY_MODULE
#define CA_UWATERLOO_DWHARDER_MY_MODULE

// Definitions and macros to be used by anyone
// using this package

#define N 100
#define F8(x) F((x), NULL, 0, 255 )
#define F16(x) F((x), NULL, 0, 65535 )

// Type definitions and structures to be used
// by anyone using this package

typedef unsigned char U8;
typedef signed char   S8;

typedef struct my_struct {
    // Fields...
} my_struct_t;

// The declarations of functions that are
// defined in the source file

void f( int, my_struct_t *, int, int, int );
int g( int, int );

#endif
```

my_module.c

```
#include "my_module.h"
// Any headers required to compile this file
#include <stdio.h>
#include <math.h>
#include "my_other_module.h"

// Definitions and macros to be used inside
// this file only

#define ERROR_LIMIT 5
#define MAX(x, y) ((x) >= (y) ? (x) : (y))

// The declaration of functions that are only
// used and defined in this source file and
// required for compilation

void swap( int *, int * );

// Function definitions

void f( int n, my_struct_t *p_ms, int low, int hi ) {
    // Performing a task
}

int g( int x, int x ) {
    // Performing another task
}
```

For modules meant to be used in other programs or modules, they will often be compiled into object files which will then be included in the compilation of other functions that require them. For example:

```
$ ls
main.c   my_module.c   my_module.h   my_other_module.c   my_other_module.h
$ gcc -c my_module.c
$ gcc -c my_other_module.c
$ ls
main.c              my_module.c         my_module.h         my_module.o
my_other_module.c   my_other_module.h   my_other_module.o
```

We will now compile and execute a source file that has a global `int main(…)` function.

```
$ gcc -o executable_name main.c my_module.o -lm
$ executable_name
executable_name: Command not found.
$ ./executable_name
..running running running..
```

Note that the file name of the executable is executable_name, but just typing that at the prompt will not automatically execute that file. If you type ls, however, it seems to work. This is because the shell (terminal interface) has a user-defined list of places that it will for executable files (type

```
$ echo $PATH
```

if you want to see where it looks), and if it doesn't find it in one of those directories, it will stop searching. Thus, if you want to execute a file that is not in the path, you must explicitly give a path either from root "`/`" or from the current directory, "`.`"; for example, both of these would work:

```
$ /home/dwharder/mte241/executable_file
$ ./executable_file
```

If you do not include an output file, the default name of the executable will be `a.out` (for *assembler output*, which is technically wrong, as it is the linker output).

You will note we must use `-lm` to link the math library (which includes the implementation of `double sin(double)`), but we didn't have to link to a library containing `printf`. This is because, as a general

rule, any header file prefixed by "`std`" is automatically linked in, and you must explicitly use `-nostdlib` if you don't want it linked. For every other library, e.g., `libname.so`, you must include it in the linking process with `-lname`. For the most part, you won't have to worry about this in this course, as the IDE will take care of all of this.

For your information, all functions in `stdio.h` are in `libc.so` and all functions in `math.h` are in `libm.so`.

The reason for this discussion is that we will be using the Keil RTX (for Real-Time eXecutive) real-time operating system (RTOS) that comes with our Keil evaluation board and the μVision4 IDE. Consequently, it will be useful for you to understand how the forest of header files are all related. Once you start looking at the library, you will find a number both header and source files related to the operating system. These are shown in Figure 2-5.

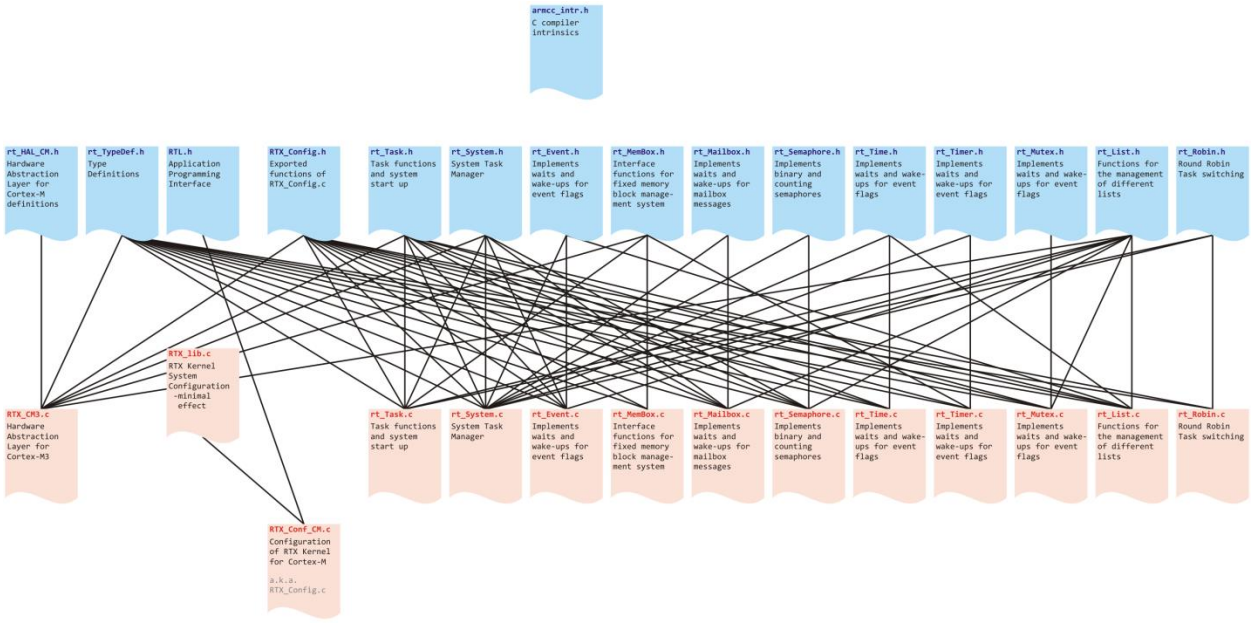


Figure 2-5. Header and source files for the Keil RTX RTOS.

We'll look at two just to note what the files contain, `rt_Mailbox.h` and `rt_Semaphore.h`, reprinted here for academic purposes:

```
/*----------------------------------------------------------------------------
 *      RL-ARM - RTX
 *----------------------------------------------------------------------------
 *      Name:    RT_MAILBOX.H
 *      Purpose: Implements waits and wake-ups for mailbox messages
 *      Rev.:    V4.70
 *----------------------------------------------------------------------------
 *      This code is part of the RealView Run-Time Library.
 *      Copyright (c) 2004-2013 KEIL - An ARM Company. All rights reserved.
 *---------------------------------------------------------------------------*/

/* Functions */
extern void      os_mbx_init(     OS_ID mailbox, U16 mbx_size );
extern OS_RESULT os_mbx_send(     OS_ID mailbox, void *p_msg,    U16 timeout );
extern OS_RESULT os_mbx_wait(     OS_ID mailbox, void **message, U16 timeout );
extern OS_RESULT os_mbx_check(    OS_ID mailbox );

extern void      isr_mbx_send(    OS_ID mailbox, void *p_msg );
extern OS_RESULT isr_mbx_receive( OS_ID mailbox, void **message );

extern void      os_mbx_psh( P_MCB p_CB, void *p_msg );

/*----------------------------------------------------------------------------
 * end of file
 *---------------------------------------------------------------------------*/
```

```
/*----------------------------------------------------------------------------
 *      RL-ARM - RTX
 *----------------------------------------------------------------------------
 *      Name:    RT_SEMAPHORE.H
 *      Purpose: Implements binary and counting semaphores
 *      Rev.:    V4.70
 *----------------------------------------------------------------------------
 *      This code is part of the RealView Run-Time Library.
 *      Copyright (c) 2004-2013 KEIL - An ARM Company. All rights reserved.
 *---------------------------------------------------------------------------*/

/* Functions */
extern void      os_sem_init(  OS_ID semaphore, U16 token_count );
extern OS_RESULT os_sem_send(  OS_ID semaphore );
extern OS_RESULT os_sem_wait(  OS_ID semaphore, U16 timeout );

extern void      isr_sem_send( OS_ID semaphore );

extern void      os_sem_psh( P_SCB p_CB );

/*----------------------------------------------------------------------------
 * end of file
 *---------------------------------------------------------------------------*/
```

We have standard headers and footers, and a sequence of functions that perform various operations. You will not be expected to memorize or understand all of this at this point, but by the end of the course, you will have a good idea as to the purpose of each of these files. At the top of Figure 2-5 is a header file that is included by default in each compilation and below this, to the right, are eleven header files with corresponding source files. These files are for the operating system; however, other files are microprocessor specific, such as

`LPC17xx.h`
: CMSIS Cortex-M3 Device peripheral access layer header file for NXP LPC1768 and related devices.

`system_LPC17xx.c`
: CMSIS Cortex-M3 Device System source file for NXP LPC1768 and related devices.

2.2.8 Further help

One of the best books on the market for programming in C is *Practical C Programming* by Steve Oualline, or—as it is better known—the "*Cow Book*. Another excellent text—especially for this course is the 2007 Springer Verlag on-line text by Parab, Shelake, Kamat and Naik (PSKN), *Exploring C for Microcontrollers: A Hands on Approach*, which uses the Keil development environment. These are shown in Figure 2-6.

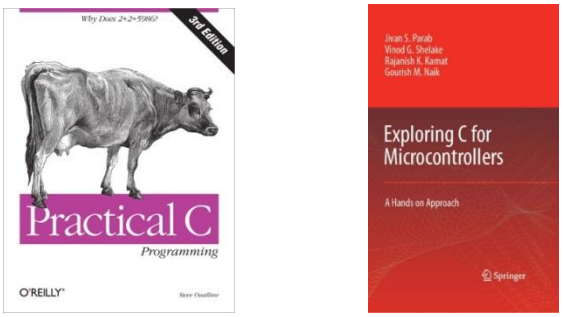


Figure 2-6. Practical C Programming from O'Reilly, Inc., and Exploring C for Microcontrollers from Springer-Verlag.

There are additional web sites available from the various manufacturers, including Keil, ARM and NXP Semiconductors.

2.2.9 Bit-wise operations

You have likely been taught bit-wise operations, but you might not be sure what they're useful for.

Let's take as an example, a set of five different Boolean-valued flags that control the state of an operating system. We could define five global variables of the appropriate type:

```
#include <stdbool.h>

bool flag_all;
bool flag_directory;
bool flag_long_name;
bool flag_recursive;
bool flag_no_backups;
```

Unfortunately, this occupies five bytes. Instead, we could use a single byte:

```
#define ALL         (1 << 0)
#define DIRECTORY   (1 << 1)
#define LONG_NAME   (1 << 2)
#define RECURSIVE   (1 << 3)
#define NO_BACKUPS  (1 << 4)

unsigned char flags;
```

Now, if you want to access the flag for `LONG`, use

```
if ( flags & LONG_NAME ) { … }
```

If you want to set the flag for `RECURSIVE` to true, use

```
flags |= RECURSIVE;
```

If you want to set the flag for `ALL` to false, use

```
flags &= ~ALL;     // Bit-wise NOT
```

Now, if you had a tri-value flag (one that holds values of `TRUE`, `FALSE` or `FAIL`), you could just use the next two bits:

```
#define USER        (3 << 5)
#define USER_FALSE  (0 << 5)
#define USER_TRUE   (1 << 5)
#define USER_FAIL   (2 << 5)
```

Now, however, we would have to do a little more work

```
if ( (flags & USER) == USER_TRUE ) { … }
```

Bit shifting and bit-wise AND can be used to extract components of a number:

```
int x = 1561710820;         // 01011101000101011101000011100100
int y = (x >> 5) & 511;     // 00000000000000000000000010000111
```

Note that 511 is $2^9 - 1$ or `(1 << 9) - 1` or 111111111_2.

2.2.10 Bit-fields in C99

An addition to the 1999 standard for the C programming language was an internalization of the concept of a bit field. The number of bits that a particular field takes up is specified by a trailing colon followed by a positive integer indicating the number of bits. This removes the need to access the bits through individual bit-wise operations. Taking the examples in Section 2.2.8 and re-interpreting them as bit-fields, we have the following code:

```
#include <stdio.h>
#include <stdbool.h>

#define FALSE 0
#define TRUE  1
#define FAIL  2

typedef struct {
        bool all       : 1;
        bool directory : 1;
        bool long_name : 1;
        bool recursive : 1;
        bool no_backup : 1;
        bool user      : 2;
} flag_t;

int main( void ) {
        flag_t my_flags;

        my_flags.all  = TRUE;
        my_flags.user = FAIL;

        my_flags.all  = FALSE;
        my_flags.user = TRUE;

        return 0;
}
```

Note that `true` and `false` are defined in `stdbool.h`.

2.2.11 switch statements versus function pointers

As described by Nigel Jones[6], often C programmers will use a switch statement under what appear to be appropriate conditions: a particular variable may take on one of a fixed number of values, and it is possible to describe the response to each possible value. On the surface, the switch appears to be very appropriate; however, the compiler may choose to compile a switch statement into

1. a calculated jump necessitating the possibility of wasted space,
2. an if-else-if chain

or an appropriate combination thereof. A switch statement may be reasonable if the range of possible values is contiguous and the blocks of code to be executed are similar in size, but Nigel points out that even a small change in the code base may result in a significant change in performance due to the compiler choosing an alternate approach for encoding the switch. An alternative, to a switch statement, at least with the range of values are contiguous or equally spaced is to use function pointers. Suppose, for example, that you have already declared four functions

```
void response0( void );
void response1( void );
void response2( void );
void response3( void );
```

You may now initialize an array

```
void (*function_array[4])( void );

function_array[0] = &response0;
function_array[1] = &response1;
function_array[2] = &response2;
function_array[3] = &response3;
```

and now you may call

```
function_array[2]();
```

In a case where a variable *n* may take on one of four possible values from 0 to 3, you may now replace the switch statement

```
switch( n ) {
    case 0:  // response 0...
        break;
    case 1:  // response 1...
        break;
    case 2:  // response 2...
        break;
    case 3:  // response 3...
}
```

with a function call from the table:

```
function_array[n]();
```

The execution is now well defined, and changes to the code will not affect the run time of responses not related to the one being modified.

[6] See http://embeddedgurus.com/stack-overflow/category/efficient-cc/.

2.2.12 µVision4 specifics: variable declarations in functions

You may be familiar with C++ where you can declare variables at any point, and this in general also holds in C; however, older versions of the compiler that ships with µVision4 require all local variables to be declared at the top of the function.

2.2.13 The pre-processor

You've already seen the `#include` pre-processor directives, and we've touched on #define to give a definition to an *identifier* (any token starting with an underscore or a letter followed by any number of underscores, letters or numbers). Note that in C, you must still specify the `.h` at the end of library files (for example, `#include <stdio.h>`). When C++ introduced namespaces, they moved to the convention that, for example,

```
#include <iostream.h>  // Access the deprecated version without namespaces
#include <iostream>    // Access the new version
```

There are, however, other features that allow for conditional inclusion of source code to be sent to the compiler

```
#ifdef IDENTIFIER
// ..
#else
// ..
#endif
```

For example, if you are developing an embedded system, it may have to compile for numerous microcontrollers, so you may want to

```
#if defined LPC17xx
  #include "rtx_lpc17xx.h"
#elif defined EFM32
  #include "rtx_efm32.h"
#else
  #error "no target specified at the command line"
#endif
```

Now, you can choose which header files are included in the compilation based on the arguments:

```
$ gcc example.c -DEFM32

$ gcc example.c -DLPC17xx
```

If you forget to specify the target, you get the error:

```
$ gcc example.c
example.c:8:2: error: #error "no target specified at the command line"
```

If you want to actually give an identifier a value, use

```
$ gcc example.c -Dthe_answer=42
```

The line

```
#if defined
```

is used so commonly that it is abbreviated to

```
#ifdef
```

with a similar definition of `#ifndef`. While you can also *undefine* an identifier you no longer want used; for example,

```
#undef EFM32
```

removes the definition of `EFM32` (if any), Rule 22 of the JPL coding standard states that

"`#undef` shall not be used."

Finally, you can define a macro—essentially, an in-line function:

```
#define MAX(x, y) ((x) <= (y)) ? (x) : (y))
```

Now, if you call `MAX( var_1, var_2 )`, the preprocessor replaces this with

```
((var_1) <= (var_2)) ? (var_1) : (var_2))
```

You'll notice that there are a lot of parentheses there: this is to ensure that expressions such as `MAX(a+b, c/d)` work. If you used

```
#define UNPROTECTED_MAX(x, y) x <= y ? x : y
```

then `UNPROTECTED_MAX( addr_1 & 1020, addr_2 & 1020 )` will do the simple substitution of

```
addr_1 & 1020 < addr_2 & 1020 ? addr_1 & 1020 : addr_2 & 1020
```

which the compiler will interpret as

```
addr_1 & (1020 < addr_2) & 1020 ? addr_1 & 1020 : addr_2 & 1020
```

Another example is

```
#define DEG_TO_RAD(x) ((x) * 0.01745329251994330d)
```

If you did not place parentheses around the `x`, then `DEG_TO_RAD( angle + 90 )` would be replaced by

```
angle + 90 * 0.01745329251994330d
```

We will often see macros used to specify default values of parameters in C:

```
#define F1( a ) f( (a), 0, NULL )
#define F2( a, b ) f( (a), (b), NULL )

int f( int a, int b, int *p_c ) {
   // Performs tasks...
}
```

You may even see

```
#define INIT_STACK( x ) stack_t x; stack_init( &x )
```

Now, if you have

```
INIT_STACK my_stack;
```

this is replaced with

```
stack_t my_stack; stack_init( &my_stack );
```

Note that you can use comments after your defined identifiers if necessary—these are not substituted into the source code.

```
#define false  0
#define true   (!false)   // Negated 'false'
```

Use the `-E` option in gcc to see the output of a source file after running only the preprocessor.

2.3 Software engineering and development

The practice of software engineering is the systematic, disciplined and quantifiable application of engineering, scientific or algorithmic principles and experience to research, design, develop, test, document, operate and maintain economically viable software solutions to problems that concern the safeguarding of the client, public or environment. This section will be expanded to give an introduction to some basic software engineering practices followed by a section on good programming practices.

2.3.1 Software engineering and the software development cycle

The development of software is fundamentally different from many other engineering processes, as

1. software is not material, and therefore can become arbitrarily complex, and
2. software rarely ends in a final product, as it can easily be upgraded or replaced.

Consequently, this results in a more cyclical approach to development, often where subsequent versions can be released on a regular cycle. Unfortunately, the complexity of software, and its implied promise of solving problems that cannot be easily solved in more traditional fields of engineering, results in numerous issues, for example:

> "The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time."[7]
>
> Tim Cargill

We will quickly look at a reasonable model for the software development cycle and then consider good programming practices to complement the design cycle.

Note that much of the information in these slides is derived from associated Wikipedia pages.

2.3.1.1 Problem definition

First, it is necessary to explicitly define the problem we are trying to solve. This should be a brief statement that indicates to all parties what is required.

[7] Jon Bentley, "Programming pearls: Bumper-Sticker Computer Science". Communications of the ACM 28 (9): 896–901, 1985.

2.3.1.2 Software requirements

Given the problem definition, it is next necessary to determine what is required of the solution. This includes either the capabilities to or conditions imposed on the solution in order to either

1. solve the problem, or
2. comply with a regulatory or legal constraint (terms within contracts, standards, regulations, statutes).

To determine the requirements, it is necessary to communicate with those who have an interest in seeing a solution to the given software problem. Such individuals or groups are referred to as *stakeholders* and these may include the client or sponsors, the development team together with their management and executives, and possibly any group that either operates, benefits from, is adversely affected by, markets, or regulates the solution.

In order to determine the requirements, the following steps should be followed:

1. elicitation,
2. analysis,
3. specification,
4. validation, and
5. management.

2.3.1.2.1 Elicitation

The first step is to approach the stakeholders and, through either discussion or observation, to gather or discover what is require the solution to do or solve. At this point, the requirements will be a collection of disparate and possibly contradictory statements about the solution. Following this, it is necessary to analyze these statements.

2.3.1.2.2 Analysis

Once the requirements have been collected, it is necessary to analyze to restate them clear and unambiguous terms using consistent language. A requirements may be described as either a

1. functional requirement if it describes an expected response of the solution given specified conditions and inputs,
2. performance requirement if it describes the run-time or memory requirements,
3. ancillary requirement if it describes a quality of the solution including qualities describing
    a. the execution, such as reliability, security, availability or usability, and
    b. evolution, such as maintainability, extensibility, portability and scalability.

The overlap between functional and ancillary requirements is not necessarily clear or distinct. Instead, it provides a This can be helped through the development of *stories*. These describe how the purpose of the software through various examples. These may include one or more *narratives* that describe why the solution is necessary:

As a ___*role*___, I want ___*feature*___ so that ___*benefit*___.

For each such narrative, one or more *scenarios* is presented in order to describe a requirement of the solution:

Given ___*situation and context*___ and when ___*conditions and events*___,
then ___*list of responses and outcomes*___.

requirement that places an absolute quantifiable bound on some characteristic of the solution is often said to be a *constraint*.

2.3.1.2.3 Specification

A requirements specification is a deliverable that presents the requirements in a single document.

2.3.1.2.4 Summary of software requirements

Once the software requirements and specifications are completed, they should be validated to ensure that the solution proposed will in deed solve the engineering problem of the user or client. Once this is completed, we can proceed to project planning.

2.3.1.3 Project planning

The project plan should include:

1. a statement of work, including
    a. a list of features,
    b. a description of deliverables, and
    c. an estimate as to the effort required for each deliverable;
2. a list of resources required to complete the project;
3. a work-breakdown structure indicating those resources (human and material) that will be developed to each component of the project;
4. a project schedule, and
5. a risk plan.

Following the project plan, we may proceed to the design.

2.3.1.4 Design

The design document will give both a high-level design together with detailed design plans for each component.

2.3.1.5 Code development

As the design document takes shape, it is possible to start development, which includes the authoring, testing and verification of the code. At this point, as the code is written, it is essential to write *unit tests* that check that the specifications are being implemented correctly. A unit test, when it passes, ensures that some component of the specification is being satisfied, and it should be possible to link the test to the specification.

2.3.1.6 Integration

As the components and deliverables are developed, they must be integrated with other components and deliverables. This will lead to further integration testing and verification.

2.3.1.7 System verification and validation

Once all the components are developed and integrated, it will be necessary to perform system-wide testing, verification and validation. Following this step, the solution can be deployed.

2.3.1.8 Maintenance

Following the deployment, additional changes will continue to be performed on the code base to address issues such as

1. corrective maintenance to fix the code where it does not meet the specification (bug fixing),
2. adaptive maintenance where evolving technology must be adapted to,
3. perfective maintenance where improvements are made, and
4. preventative maintenance where future problems are anticipated and corrected.

With corrective maintenance, it is important to write *regression tests*, that fail if the fix is not in place, and pass when the fix is in place.

2.3.1.9 Summary of software engineering

Software engineering is the process of beginning with the requirements of the user or client and following a systematic approach to determining the problem and the requirements, a specification, project plan and design document, upon

which the code development will be based, after which there will be integration and integration testing, followed by system testing, verification and validation, after which the solution is deployed. Following this is the inevitable maintenance required to correct, perfect or adapt the code base,.

2.3.2 Good development practices

Good coding practices consist of an informal collection of heuristics, guidelines and rules that have been observed to correlate with improved software quality. We will look at a number of such best practices, once this section is completed.

2.3.2.1 Good programming practices

While everyone can appreciate the following code, which prints all twelve verses of the song "The Twelve Days of Christmas"[8], here modified to allow compilation in modern gcc compilers without additional options, it might be somewhat difficult to first find and then fix the one spelling mistake:

```
#include <stdio.h>
main(int t,int _,char *a){return!0<t?t<3?main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a)):1,t<_?
main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l+,/n{n+,/+#n+,/#;#q#n+,/+k#;*+,/'r :\
'd*'3,}{w+K w'K:'+}e#';dq#'l q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';\
d}rw' i;# ){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' iwk{KK{nl]!/w{%'l#\
#w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c ;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w!nr'\
/ ') }+}{rl#'{n' ')#}'+}##(!!/"):t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_
,a+1):0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-\
O;m .vpbks,fxntdCeghiry"),a+1);}
```

This code—which uses such interesting programming techniques such as recursive calls to `main(…)`, using the underscore `_` as a variable name, and nested `?:` operations—has become famous, or infamous, among programmers. That which is so wrong is so admired for its wrongness, or, more colloquially, like "Plan 9 from Outer Space", it's so bad, it's good.

Writing such code does have one significant benefit: job security for the author. Unfortunately, writing production code like this will also lead to the phenomenon of writing new code rather than trying to fix or maintain an existing code base. We will look at some of what are considered the best coding practices, including:

1. commenting your code,
2. choosing good function and variable names and using naming conventions,
3. consistent indentation,
4. restricting the line length to 80 characters,
5. grouping related statements,
6. limiting the scope of variables,
7. always using braces for control structures, and
8. avoiding heavily nested control statements.

We will begin with comments.

[8] In an e-mail from Jim Coplien to his colleagues on Tuesday, December 22, 1998.

2.3.2.1.1 Comments

You have heard over and over again that you should "comment your code". Comments are to aid the reader in understanding the purpose of the code, not what it does. The most useless comments translate the coded into English:

```
++i;   // Add 1 to i

// Loop from j = 0 to n - 1
for ( int j = 0; j < n; ++j ) { ...

// If k = 3, do something
if ( k == 3 ) { ...
```

Comments should explain the purpose of the code:

```
// Loop over the rows in the matrix
for ( j = 0; j < n; ++j ) { ...

// First deal with the special case when k == 3, ...
if ( k == 3 ) { ...
```

2.3.2.1.1.1 Comment blocks versus single-line comments

C and C++ have two styles comments. Any text that appears between a /* and an ending */ represents a comment block and may span multiple lines. Such comments cannot be nested: any /* that appears in a comment block is treated as part of the comments. When documenting code, comments are often highlighted

```
/*****************************************************************
 * Enter your comments here
 *  - documentary, 70 characters in width
 *****************************************************************/
```

All asterisks other than the first and last are purely decorative, but allow the reader to easily distinguish between comments and code. Do not, under any circumstances, try to create clean comments by closing the comment block on the right,

```
/*****************************************************************
 * NEVER DO THIS                                                 *
 *  - the extra work discourages programmers                     *
 *    from updating the comments.                                *
 *****************************************************************/
```

If it is necessary to divide your code base into sections, it is relatively easy to do so in a way that quickly attracts the attention of the programmer reading the code:

```
/*****************************************************************
 * ************************************************************* *
 * *              Class declarations and definitions             * *
 * ************************************************************* *
 *****************************************************************/
```

The other style of comments are single-line comments, where all text following a double-slash and up to the end of the current line is treated as a comment.

```
// This is a single-line comment
```

Such comments are useful for short comments found inside of class, structure and function definitions, however, judgement should be used when deciding whether to append such comments to the end of a line, or to keep such comments on separate lines. For example, in documenting a class or structure, by appending information to the end of each line, the reader is still able to scan both the types and the fields or member names.

```
typedef struct {
   size_t  dimension;                // matrix dimension of a square n x n matrix
   size_t  max_off_diagonal_entries; // maximum number of off-diagonal entries in the matrix
   size_t *a_row_index;              // array of the index of each row's first off-diagonal entry
   size_t *a_column_index;           // array of the column of each off-diagonal entry
   double *a_diagonal_values;        // an array of the n diagonal entries
   double *a_off_diagonal_values;    // an array of all off-diagonal entries of the matrix
} sparse_matrix_t;
```

When documenting a function in this manner, it will lead to a maintenance disaster:

```
        if ( empty() ) {
           result = obj;   // An object larger than the item requested was not found
                           //  - return the item that was passed to flag this situation
        } else if ( value() <= obj  ) {             // First, there is seldom much space over here
           result = right()->next( obj );           // Second, this is a maintenance disaster, as
        } else {                                    // any change to a block of code will require
           Type tmp = left()->next( obj );          // the spacing to be adjusted.
                                                    // Additionally, if lines of code are added,
           result = ( tmp == obj ) ? value() : tmp; // this will break the comments, increasing
        }                                           // the frustration of the programmer even
                                                    //  more.
```

Comments that describe the functionality of code or explain a specific deviation should generally be at the same level of indentation as the code that is being described.

A significant benefit of including only single-line comments within classes, structures and functions is that any portion of the code can be temporarily commented out through the use of comment blocks without having to worry about nested comments. For example, it is always possible to comment out an entire function:

```
        /* Comment out the an entire function
        bool is_upper_triangular( sparse_matrix_t *this ) {
            // Local variable declarations and initializations
            bool return_value = true;

            // These comments are ignored
            for ( i = 1; i < this->dimension; ++i ) {
                // Some other code
            }

            return return_value;
        }
        */
```

or just one component:

```
        bool is_upper_triangular( sparse_matrix_t *this ) {
            // Local variable declarations and initializations
            bool return_value = true;

            // A description of what this for loop does...
            for ( i = 1; i < this->dimension; ++i ) {
                /* Comment out this conditional statement
                if (
                    this->a_row_indices[i] < this->a_row_indices[i + 1]
                    && this->a_column_indices[this->a_row_indices[i]] < i
                ) {
                    // This comment is ignored
                    return_value = false;
                    break;
                } */
            }

            return return_value;
        }
```

In some cases, the single-line comments are even used in documenting functions and classes:

```
////////////////////////////////////////////////////////////////////
// Enter your comments here
//  - documentary, 70 characters in width
////////////////////////////////////////////////////////////////////
```

or for code division:

```
////////////////////////////////////////////////////////////////////
/ ////////////////////////////////////////////////////////////// /
/ /              Class declarations and definitions              / /
/ ////////////////////////////////////////////////////////////// /
////////////////////////////////////////////////////////////////////
```

This has the benefit that entire sections of code can be simultaneously commented out with a single block comment.

2.3.2.1.1.2 Classification of comments

An excellent source every programmer and engineer should read is Bernhard Spuida's "The fine art of commenting". His first observation is one that all engineers should be aware of:

> "All time saved by not commenting during the coding process is made up more than twice at least by inserting comments and providing documentation after the coding process is finished."

Failure to add comments what-so-ever will repay itself tenfold during code maintenance and future development, as the programmers who are fixing or extending an existing code base will take that much more time to understand the existing code. He classifies comments based on their purpose:

1. documentation,
2. functionality, and
3. explanatory.

To begin, every function, class, structure and in some cases even variables (to be discussed later) needs to be documented, and this documentation must be standardized. Aspects that may be described include:

1. purpose,
2. requirements,
3. arguments and return values,
4. hardware dependencies, and
5. known weakeness (often in the form of a *to-do* list).

Some information that may be stored in the version control system, including:

1. creation date,
2. author's name, and
3. the change history,

however, any changes to the functionality must always be refected in the documentary comments.

First and foremost, one aspect that will be very different when programming real-time systems where sensors pass information from the environment and ultimately are assigned to variables. It is important that the declaration of the variable has comments that give the reader a clear understanding of the significance of the variable[9], including

1. a description of what the variable stores,
2. the units of the variable,
3. the resolution of the least-significant bit (LSB), and
4. the dynamic range of the variable (smallest and largest values).

For example, a temperature sensor reading or the angular speed of a while may be commented as follows:

```
unsigned short outside_temperature;   // Ambient temperature (degrees Fahrenheit)
                                      // Range:   0x0000 - 0x07ff
                                      //        -20 degF - 144 degF
                                      // 1 LSB = 0.08 degF
                                      // 32 degF = 0x028a

signed int wheel_lf_angular_speed;    // Angular speed of the left front wheel (rad/s)
                                      // Range:    0xff000001 - 0x00ffffff
                                      //           -161 rad/s - 161 rad/s
                                      // 1 LSB = 0.0000096
```

In addition to commenting the significant of variables, it is also useful to explain to the reader what problem the code is attempting to solve, and how is it solving that problem. Comments should not simply re-iterate what the code is doing, they should enlighten the reader as to why it is doing this. This now begs the question: what is a useful comment? For example, recall that a binary tree is a node-based data structure where

1. each node contains a value and two pointers to left and right nodes,
2. one node is designated the *root node*,
3. if the left or right nodes are not null pointers, those nodes are called *children* of the given node,
4. a *path of length N* is a sequence of nodes (n_0, n_1, …, n_N) where n_{k+1} is a child of n_k,
5. there is a unique path from the root node to each node in within a tree and the length of that path is the *depth* of the node (the root node having a depth of 0), and
6. given any node *n*, the collection of all nodes *m* such that there is a path (*n*, …, *m*) is the sub-tree rooted at *n*.

A binary search tree is a binary tree where each node within the tree

1. all nodes in the left sub-tree have values less than or equal to the value stored at the root node,
2. all nodes in the right sub-tree have values greater than or equal to the value stored at the root node, and
3. both the left and right sub-trees are themselves binary search trees.

[9] "Real world variables" by Nigel Jones: http://embeddedgurus.com/stack-overflow/2013/01/real-world-variables/

Given a binary tree, we can now ask a question such as: "What is the next-largest element of a given value?" One implementation of such an algorithm in C++ is:

```
// Return the object if no next-largest value is found
template <typename Type>
Type Binary_search_node<Type>::next( Type const &obj ) const {
    Type result;

    if ( empty() ) {
        result = obj;   // Return the object
    } else if ( value() <= obj  ) {              // If the value is less than or equal to
        result = right()->next( obj );           // the object, get the next-largest
    } else {                                     // object from the right tree;
        Type tmp = left()->next( obj );          // otherwise, get the next value from
                                                 // the left tree and if no larger value
        result = ( tmp == obj ) ? value() : tmp; // is found there, return this value
    }

    return result;
}
```

These comments are little better than

```
++i; // Increment i
return 0; // Return 0
```

They say what the code is doing, but even a mediocre programmer can understand this.

Instead, the above function is so short, it would be better to comment in the description:

```
/*****************************************************************************
 * template <typename Type>
 * Type Binary_search_tree<Type>::next( Type const &obj ) const
 *
 * In a binary search tree, find the next-largest object of the argument
 *  - If no next-largest entry is found, return the argument 'obj'
 *  - There can be duplicate entries in the tree
 *
 * Given any node, there are three possibilities:
 *   1. We are at an empty node, in which case, there is no next-largest
 *      object--return the argument.
 *   2. The value of the entry is less-than-or-equal-to the argument, thus
 *      if there is a next larger entry, it must be in the right sub-tree
 *   3. The value of the entry is greater than the argument,
 *       - query the left sub-tree to find the next-largest entry
 *       - if a next-largest entry is found, return it,
 *       - otherwise, this must be the next-largest entry, so return the value
 *****************************************************************************/

template <typename Type>
Type Binary_search_node<Type>::next( Type const &obj ) const {
    Type next_entry;

    if ( empty() ) {
        // An empty sub-tree has no next-largest entry
        next_entry = obj;
    } else if ( value() >= obj  ) {
        // The right sub-tree must contain the next-largest entry
        next_entry = right()->next( obj );
    } else {
        assert( value() > obj );

        // Query the left sub-tree for the next-largest entry
        Type next_entry_in_left = left()->next( obj );

        // If none is found, this is the next-largest;
        // otherwise, return what is found
        next_entry = ( next_entry_in_left == obj ) ? value() : next_entry_in_left;
   }

    return next_entry;
}
```

Note how the structure of the comments reflects the structure of the code? Alternatively, you could write:

```
 * Given any node, there are three possibilities. If we are at an empty node, in
 * which case, there is no next-largest object, so return the argument. If the
 * value of the entry is less-than-or-equal-to the argument, thus if there is a
 * next larger entry, it must be in the right sub-tree. Finally, the value of
 * the entry is greater than the argument, so query the left sub-tree to find the
 * next-largest entry and if a next-largest entry is found, return it, otherwise,
 * this must be the next-largest entry, so return the value.
```

This might be an excellent explanation in a text book, but associating the comments with the source code is never-the-less difficult. Writing comments is, in many ways, an art form, and always remember that **you** are likely going to be the programmer who is looking at this code, only one week from now, you've forgotten what it is you were doing when you wrote it.

For more complex routines, you may want to describe the functionality of any conditional or looping statements immediately prior to those statements and any initialization statements required for those flow-control statements to

execute. In general, end-of-line comments tend to describe what that line does. Most programmers can figure out what a line of code does; what you want to do is explain what the code is trying to accomplish.

Or in short: comments should explain *why*, not what.

2.3.2.1.2 Names and naming conventions

As observed by Guido van Rossum and others, code is more often read than written, and therefore readability is essential. In order to enhance readability, consistent naming conventions are used, so we will discuss

1. word concatenation,
2. variable names,
3. type names, and
4. function names.

We will begin by looking at concatenating multiple words to form type, variable and function names.

2.3.2.1.2.1 Word concatenation

Every symbol in most programming languages consists of a letter or underscore followed by a sequence of zero or more letters, underscores or digits. These are usually used to represent words, and consequently, there is always the issue as to how to join a string of words together without the availability of a space. There are four approaches described in the following table.

Convention	Description	Examples
juxtaposition	join the words	`stacksize pushfront singlelist protonmass`
snake_case	join with intermediate undercores	`stack_size push_front single_list proton_mass`
camelCase	capitalize subsequent words and join	`stackSize pushFront singleList protonMass`
ALL CAPS	capitalize all leters and join with underscores	`STACK_SIZE PUSH_FRONT SINGLE_LIST PROTON_MASS`

Both snake_case and camelCase may be described as lower or upper depending on the capitalization of the first letter of the first word. Upper camelCase is sometimes referred to as PascalCase.

More recently developed programming languages have created standard libraries that choose one of these conventions, and programmers who adopt the language maintain the convention.

	Variables	Constants	Functions	Types
Java	lower camelCase	ALL CAPS	lower camelCase	upper camelCase
C#	lower camelCase	ALL CAPS	upper camelCase	upper camelCase
Matlab	juxtaposition	juxtaposition	juxtaposition	juxtaposition
Python	lower snake_case	ALL CAPS	lower snake_case	lower snake_case

In C and C++, however, there is no universal convention for word concatenation. Lower snake_case is usually used, except for constants which are ALL CAPS, and in C++, classes use upper snake_case; however, if a significant portion of a development team is derived from, for example, programmers more familiar with Java, they may adopt a blend of lower and upper camelCase.

As to which is preferable, at least one paper, "An Eye Tracking Study on camelCase and under_score Identifier Styles" by Sharif and Maletic presented at the 18th IEEE International Conference on Program Comprehension shows that snake_case is more easily recognized than camelCase. Consequently, we will use lower snake_case throughout this publication.

2.3.2.1.2.2 Name components and length

Any name should be sufficiently descriptive to allow a casual reader with some familiarity with the project to easily discern the purpose of either the type, variable or function. Consequently, some guidelines you should consider following include:

1. Names should be meaningful, meaning not too short as to be ambiguous, but not so long as to be difficult to read, so for a variable describing the maximum queue length in the last ten minutes, `maximum` is likely too short, `max_queue_size` is likely reasonable if the time is inferred, but `maximum_10_minute_queue_size` is likely too long.
2. Different names should differ by more than just a few letters, as
   ```
   double average_medial_node_depth;
   double average_medium_vertex_depth;
   ```
   are too similar.
3. Acronyms and initialisms can be made use of, but not excessively, as this may lead to opaque variable names.
4. While `l0`, `lO`, `IO`, `lo`, `Io`, `Ol`, `O1`, `OI`, `ol`, `o1` and `oI` are all valid names, don't use them. The Consolas typeface uses a slashed zero, but many typefaces (including Courier New, Arial, and Times New Roman) make it difficult to distinguish between 0 and O.
5. If you must use shorter variable names,
   a. use *i*, *j*, *k*, *n*, *m* for integers, and if you must (because you are implementing a given formula), use `ell` for *l*,
   b. use *x*, *y*, *z* for floating-point numbers, and
   c. use *t* for time, *d* for distance, etc.

   Most readers will be confused if he or she sees a statement like
   ```
   double i = array[x];
   ```

All of these are guidelines and not hard-and-fast rules, and there are always circumstances where the strict adherence to these guidelines will have negative consequences.

2.3.2.1.2.3 Type names

A type, be it a structure or a class, should be a singular noun. If it is a container of nodes, it could be either a `node_pool_t` or `node_list_t`, but do not use `nodes_t`. In C, the naming convention is to give structures a type name using lower snake_case followed by an `_t`. This is only a convention, but if you choose not to follow it, other C users will become frustrated with your packages. In C++, the naming convention for classes is to use a capitalized snake case word, such as `Node_pool` or `Node_list`. As a counterexample, the Keil RTX RTOS uses `U8`, `U16`, … and `S8`, `S16`, … to represent types for unsigned and signed 8-, 16-, etc. bit integers, in contrast with the more usual `uint8_t`, `uint16_t`, … and `int8_t`, `int16_t`, ….

2.3.2.1.2.4 Variable names

Variables store data, and therefore should also be represented by nouns. When deciding how to name variables, you should take into consideration that

1. variables store data, and thus the name should be a noun that reflects the data stored,
2. consistency is important, do not use `num_entries` in one location and `array_count` elsewhere,
3. indices for arrays, however, can be reduced to common letters such as `i`, `j` and `k`,
4. fields in structures should not repeat the name of the structure; for example, for a stack structure, the number of items in the stack should be `size` and not `stack_size`, and
5. where applicable, names can be suffixed by the units associated with the field; for example, `wait_time_ms`, `ambient_temp_degC`, `speed_cm_per_min` and `freq_kHz`.

In the C++ standard template library, the number of items in a container is represented by `size` and the number of items the container can hold is presented by `capacity`. Arrays should either be always singular, or always plural. Which is preferable is up to the reader, but be consistent; for example, the two alternatives are shown here:

```
// We have arrays of vertices and edges

typedef struct {
    vertex_t *p_verticies;
    size_t num_vertices;
    edge_t *p_edges;
    size_t num_edges;
} graph_t;

graph_t layouts[10];

// Now, however, referring to a single vertex
// from a single layout appears to suggest
// a plurality

layouts[5].p_verticies[7];
```

```
// Here, they suggest a pointer to one object

typedef struct {
    vertex_t *p_vertex;
    size_t num_vertices;
    edge_t *p_edge;
    size_t num_edges;
} graph_t;

graph_t layout[10];


// We are now referring to vertex 7 of
// layout 5.

layout[5].p_vertex[7];
```

It is conventional to use `p_` to prefix a pointer, as we have previously used. Whle this may seem unnecessary, it ensures the reader is aware of the significance of the variables. For example, you would never use the `.` operator on a pointer, and you would never use `->` on an instance.

```
void init_stack( stack_t *s, size_t n ) {
    s->size = 0;
    s->capacity = n;
    s->entries = malloc( n*sizeof( void * ) );
}
```

```
void init_stack( stack_t *const p_this, size_t n ) {
    p_this->size = 0;
    p_this->capacity = n;
    p_this->p_entries = malloc( n*sizeof( void * ) );
}
```

2.3.2.1.2.5 Function names

Functions perform operations, and therefore the name should reflect the action being performed. Hence, functions are often verbs or verb phrases using the imperative. A single verb, such as `evaluate(…)`, however, is often insufficient to describe the action of a function. (Of course, `analyze_this(…)` may be quite reasonable, so long as it is paired with another function `die_another_day( _00_t * )`.)

In some cases, there are common prefixes for certain actions, such as for determining a state, or fetching or setting a field

Operation	Prefix	Example	Abbreviated form
Determining a state	`is_`	`bool is_empty()`	`bool empty()`
Fetching a field	`get_`	`size_t get_capacity()`	`size_t capacity()`
Setting a state	`set_`	`void set_name( char * )`	Usually not abbreviated
Calculating a value	`find_`	`double find_min_path()`	`double min_path()`

In C, where functions are not intrinsically linked to structures, it is usual to prefix the name of the action by the structure type. Thus, associated with `stack_t` structure would be `stack_init()`, `stack_push()`, `stack_pop()` and `stack_empty()`. With C++ classes, it is, on the other hand, considered inappropriate to repeat the class name in the names of the member functions. Thus, the member functions of the `Stack` class would be `push()`, `pop()` and `empty()`.

2.3.2.1.2.6 Summary of names and naming conventions

A reasonable choice of names and naming conventions is critical to the readability of source code. Consistency is as important as clarity, and having a common naming convention on a project will greatly aid in reducing confusion among developers. It is always critical that when one is in the employ of a company or under contract, that one false the conventions adopted by the company or client. There may be issues with their naming convention, but consistency is ultimately a more important factor in reducing the costs of any project. In one situation, one author was the only person

in a company able to debug code that had previously been submitted by a long-since-gone developer, for all the variable names were in German and with abbreviations and the use of juxtaposition, it was even difficult to use a dictionary to determine the significance of particular variable names.

2.3.2.1.3 Consistent indentation and alignment of braces

There are two reasonable mechanisms for aligning braces for functions and blocks of code:

```
if ( condition ) {
    // statement body
} else if ( condition ) {
    // alternative body
} else {
    // other alternative body
}

for ( init; condition; iteration ) {
    // looping statement body
}

while ( condition ) {
    // while body
} while ( condition );

do {
    // while body
} while ( condition );

void f( parameters ) {
    // function body
}
```

```
if ( condition )
{
    // statement body
}
else if ( condition )
{
    // alternative body
}
else
{
    // other alternative body
}

for ( init; condition; iteration )
{
    // looping statement body
}

while ( condition)
{
    // while body
}

do
{
    // while body
}
while ( condition );

void f( parameters )
{
    // function body
}
```

This lead author prefers the one on the left—it is more compact and the additional spacing does not necessarily help readability; however, that is an esthetic choice. If you work at any business organization, you are required, however, to follow the style adopted by that corporation (or, at least, your group within that business organization).

2.3.2.1.4 Restrict your line length

Newspapers have narrow columns as it allows the reader to easily scan the articles, for under a certain width, the mind can comprehend the entire line without moving left-to-right and back again. Once the line length becomes too long, the eye must scan continually left and right, sometimes requiring scrolling, all actions that distract the reader from comprehending the code itself.

There are numerous reasonable mechanisms for breaking long lines. For example, the line

```
for ( SingleList<double>::iterator itr = list.begin(); itr != list.end(); ++itr ) {
    // Loop body...
}
```

will extend beyond 80 characters, but could be broken as

```
for ( Single_list<double>::iterator itr = list.begin();
      itr != list.end(); ++itr ) {
    // Loop body...
}
```

Note that the break is sufficiently indented to line it up with the start of the contents of the for loop.

Similarly, if a function has many parameters, or parameters with long names, it is useful to break it across more than one line; for example,

```
double *analyze_least_squares( double *p_data, size_t count,
                               int degree, bool ignore_outliers,
                               bool running_count, double *p_bounds,
                               bool linear );
```

In general, if you must break an arithmetic or logical expression, try to break on the lower priority operators, and therefore the right is preferable to the left:

```
double x = a + b                 double x = a
             * c;                        + b*c;
```

Similarly, place the relevant operator at the start of the line, as it may, otherwise, be assumed; for example, on the left, the reader may miss the negative sign and assume that addition is used, as opposed to subtraction:

```
double x = a + b - c + d -       double x = a + b - c + d
           e + f;                            - e + f;
```

In the following conditional statement, the spacing has been adjusted so that the reader can easily deduce the similarities and differences between the two operands:

```
if ( ( sum && (A.off_diagonal[aj] != -B.off_diagonal[bj] ))
  || (!sum && (A.off_diagonal[aj] !=  B.off_diagonal[bj] )) ) {
    ++count;
}
```

Many editors allow the user to set how many characters a tab is displayed as, and consequently, any attempt to use tabs for any additional indentation beyond the indentation necessary to group code within a block may result in undesirable results under different tab lengths. For example, the reader used spacing to identify to the reader the relationship between the values, where the tab characters are represented in yellow (with one tab equalling eight spaces):

```
	for ( int n = m; n < count; ++n ) {
		table[jy][m] =
			(   table[jy][m - 1] -   table[iy][m - 1])
		       /(x_values[jx]      - x_values[ix]);
	}
```

Changing the tab length to 4 spaces would result in code that seems more confusing than enlightening:

```
    for ( int n = m; n < count; ++n ) {
        table[jy][m] =
            (   table[jy][m - 1] -    table[iy][m - 1])
               /(x_values[jx]         - x_values[ix]);
    }
```

Consequently, restricting the code length to 80 characters, while based on the width of older terminals such as the VT100, is still relevant today to aid in readability. That we can extend our code beyond 80 characters does not mean we should.

2.3.2.1.5 Grouping related statements

Within a function, statements that are performing related operations within a function should be grouped together. For example, the following function adds a new item onto a queue and logs the transaction (storing the identifier of the object, the size of the queue, and the current time):

```
void data_queue_push( queue_t *const p_this, data_t *p_entry ) {
    if ( p_this->size == p_this->capacity ) {
        queue_double_capacity( p_this );
    }

    ++( p_this->tail );
    ++( p_this->size );

    if ( p_this->tail == p_this->capacity ) {
        p_this->tail = 0;
    }

    fprintf( p_this->logfile, "enqueue: %s %d %d\n",
             p_entry->id, p_this->size, time( NULL ) );
    p_this->p_entries[p_this->tail] = obj;
}
```

The statements, however, are not grouped, as the three statements related to placing the new object at the tail have other operations (updating the size and logging the transaction) interspersed between them. Grouping the three statements distinguishes the operations being performed, and allows for more reasonable comments:

```
void data_queue_push( queue_t *const p_this, data_t *p_entry ) {
    // Double the capacity if the queue is full
    if ( p_this->size == p_this->capacity ) {
        queue_double_capacity( p_this );
    }

    // Place the new object at the new tail of the queue and update the size
    ++( p_this->tail );

    if ( p_this->tail == p_this->capacity ) {
        p_this->tail = 0;
    }

    p_this->p_entries[p_this->tail] = obj;

    ++( p_this->size );

    // Log the event
    fprintf( p_this->logfile, "enqueue: %s %d %d\n",
                                       obj->id, p_this->size, time( NULL ) );
}
```

2.3.2.1.6 Limiting the scope of variables

Variables can be declared within any block of code delimited by braces, and their scope is restricted to that block—it is not possible to access a variable before it is declared or once it goes out of scope. It is sometimes tempting to simply declare all variables at the start of a function, but this has two consequences:

1. it is not possible for the compiler to reuse memory, and
2. a programmer might inadvertently access a variable when it was not meant to be used.

By limiting the scope of a variable, the reader can get an idea as to where the variable is meant to be used, thereby helping understand its purpose. If you are using C++ or C99, make use of the declaration of variables within the `for` statement, where the following two are equivalent for scope purposes:

```
{
    int i;

    for ( i = 0; i < n; ++i ) {
        // Do something...
    }
}
```

```
for ( int i = 0; i < n; ++i ) {
    // Do something...
}
```

The following is an implementation of the fast Fourier transform in C++, with comments removed. On the left, all local variables are declared at the start of the function, while on the right, variables are declared to limit their scope, with the most important declaration being that of the variable k in the second for loop,

```
static void FFT( std::complex<double> *p_data, size_t n ) {
    size_t i, j, k, m;
    std::complex<double> w, wn, tmp;
    double const PI = 4.0*std::atan( 1.0 );

    for ( i = n/2; i >= 1; i /= 2 ) {
        w = 1.0;
        wn = std::exp(std::complex<double>( 0.0, -PI/i ));


        for ( j = 0; j < i; ++j ) {
            for ( k = j; k < n; k += 2*i ) {
                m = k + i;
                tmp = p_data[k] + p_data[m];
                p_data[m] = (p_data[k] - p_data[m])*w;
                p_data[k] = tmp;
            }

            w *= wn;
        }
    }

    for ( i = 0, j = 0; i < (n - 1); ++i ) {
        if ( i < j ) {
            tmp = p_data[j];
            p_data[j] = p_data[i];
            p_data[i] = tmp;
        }

        for ( k = n/2; k <= j; k /= 2 ) {
            j = j - k;
        }

        j += k;
    }
}
```

```
static void FFT( std::complex<double> *p_data, size_t n ) {
    double const PI = 4.0*std::atan( 1.0 );


    for ( size_t i = n/2; i >= 1; i /= 2 ) {
        std::complex<double> w = 1.0;
        std::complex<double> wn =
            std::exp( std::complex<double>( 0.0, -PI/i ) );

        for ( size_t j = 0; j < i; ++j ) {
            for ( size_t k = j; k < n; k += 2*i ) {
                int m = k + i;

                std::complex<double> tmp = p_data[k]
                                         + p_data[m];
                p_data[m] = (p_data[k] - p_data[m])*w;
                p_data[k] = tmp;
            }

            w *= wn;
        }
    }

    for ( size_t i = 0, j = 0; i < (n - 1); ++i ) {
        if ( i < j ) {
            std::complex<double> tmp = p_data[j];
            p_data[j] = p_data[i];
            p_data[i] = tmp;
        }

        size_t k;

        for ( k = n/2; k <= j; k /= 2 ) {
            j = j - k;
        }

        j += k;
    }
}
```

The most significant benefit is that because the indexing variable `k` of the last for loop is declared outside the loop, this indicates to the reader that the variable will be used beyond the scope of the loop; whereas without the declaration, a reader may simply assume that `k` is restricted to the body of the loop.

2.3.2.1.7 Always use braces for any control structure, even if you don't have to

While this could have been indicated before, it is of such relevance that it is included as a separate point. In C, if a conditional or looping statement has a body that consists of only a single statement, one does not require braces around the body. Consequently, these are valid examples of C code:

```
if ( n != 0 ) p = b/n;
```

```
for ( i = 0; i < 10; ++i )
    sum += i;
```

Unfortunately, this short cut is the cause of numerous errors, as programmers may, for example, include another statement in the body, forgetting to include braces. It is better to use the more conventional

```
if ( n != 0 ) {
    p = b/n;
}
```

```
for ( i = 0; i < 10; ++i ) {
    sum += i;
}
```

2.3.2.1.8 Avoid deeply nested control statements

When you come across code that is deeply nested control statements, it becomes difficult to understand the interrelationship. In this case, we are attempting to determine if a sparse matrix (using the new Yale sparse matrix format) is symmetric.

```
for ( i = 1; i < p_matrix->M; ++i ) {
    for ( j = p_matrix->row[i]; j < p_matrix->row[i + 1]; ++j ) {
        if ( p_matrix->column[j] > i ) {
            break;
        }

        ++count;

        for ( k = p_matrix->row[p_matrix->column[j] + 1] - 1;
              k >= p_matrix->row[p_matrix->column[j]]; --k ) {
            if ( p_matrix->column[k] == i ) {
                if ( p_matrix->off_diagonal[k] == p_matrix->off_diagonal[j] ) {
                    break;
                } else {
                    return false;
                }
            } else if ( p_matrix->column[k] < i ) {
                return false;
            }
        }
    }
}
```

The inner body could, instead, be broken into a separate function, and one conditional can be simplified to a simple return statement:

```
bool verify_row_symmetry( Matrix *p_matrix, size_t i, size_t j ) {
    size_t k;

    for ( k = p_matrix->row[p_matrix->column[j] + 1] - 1;
          k >= p_matrix->row[p_matrix->column[j]]; --k ) {
            if ( p_matrix->column[k] == i ) {
                return p_matrix->off_diagonal[k] == p_matrix->off_diagonal[j] );
            } else if ( p_matrix->column[k] < i ) {
                return false;
            }
        }
    }
}
```

and now the original code is, while still obscure with comments, more understandable:

```
for ( i = 1; i < p_matrix->M; ++i ) {
    for ( j = p_matrix->row[i]; j < p_matrix->row[i + 1]; ++j ) {
        if ( p_matrix->column[j] > i ) {
            break;
        }

        ++count;

        if ( !verify_row_symmetry( p_matrix, i, j ) ) {
            return false;
        }
    }
}
```

2.3.2.1.9 Use #error, #warning and assertions

The C programming language allows both `#error` and `#warning` to flag exceptions during pre-processing, and assertions to check conditions at run time, but only when debugging, and not for production code. The pre-processor instructions simply take strings as arguments and print those strings to standard error; for example,

```
#if some-condition
#error some condition is not met
#endif

#ifdef DEBUG
#warning debugging statements turned on
#endif
```

Additionally, both C and C++ have the `assert.h` and `cassert` libraries, respectively. This contains a macro `assert(…)`, which takes a Boolean-valued argument. If the argument returns `false`, the assertion will terminate the program and write the filename, line number, function name and the expression that failed to standard error. This can be used to ensure that, for example, the first conditions of a conditional statement have not been significantly modified, or perhaps the second block assumes the negation of the first block. Here, we may deduce the negation of the first statement using De Morgan's rule.

```
#include <assert.h>

// Some code...

if ( ((a < b) && (c == d)) || (e <= f) ) {
    // Do something...
} else {
    assert( (a >= b) || (c != d)) && (e > f) );
    // Do something else...
}
```

```
#include <stdlib.h>

// Some code...

if ( ((a < b) && (c == d)) || (e <= f) ) {
    // Do something...
} else if ( (a >= b) || (c != d)) && (e > f) ) {
    // Do something else...
} else {
    exit( 0 );
}
```

Functionally, these are equivalent (although the second does not give information as to what failed), but assertions have additional benefit that they can be removed. By declaring

```
#define NDEBUG
```

this directs the compiler to ignore all debugging statements, including assertions. Consequently, during testing, we may leave `NDEBUG` undefined, but when it comes time for compiling the final product, `NDEBUG` can be defined, and all these tests are turned off, meaning the (hopefully) unnecessary checks are turned off, thereby resulting in faster code. In the right-hand side, the second test will always be performed, both during development and in production.

Assertions can be used, for example, to ensure that arguments or return values fall within specific ranges, or have specific properties. If during development, if these values ever fall outside those specified, an error will be raised with additional information as to where the assertion failed.

There is only one issue with assertions: the tests can never contain side effects: variables cannot be assigned, and any function calls cannot change the state of any data within the system, for once the assertions are *turned off*, these side effects will no longer occur, resulting in different behavior.

The use of assertions is highlighted in Rule 16 of the JPL coding standard, which says

> "Assertions shall be used to perform basic sanity checks throughout the code. All functions of more than 10 lines should have at least one assertion."

2.3.2.1.10 Use parentheses to indicate precedence

In secondary school, you were made aware of BEDMAS (brackets, exponentiation, division-and-multiplication, and addition-and-subtraction). Rather than evaluating operators strictly from left to right, inserting parentheses whenever needed, this allowed us to write polynomials as

$$ax^3 + bx^2 + cx + d$$

instead of

$$a\left(x^3\right) + \left(b\left(x^2\right) + \left(cx + d\right)\right)$$

which is what would be required if there was no precedence apart from brackets, in which case, the bare-bones equation would mean

$$\left(\left(\left((ax)^3 + b\right)x\right)^2 + c\right)x + d$$

which is almost certainly not what we meant…

We could tabulate this precedence rule in the following table:

1	2	3
a^b	$a \cdot b$	$a + b$
	$a \div b$	$a - b$

Similarly, C has operator precedence, but there are many more operators, and thus, one could memorize the table, where all operators are binary unless an explicit `a` or `x?y:z` is indicated.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
→	←	→	→	→	→	→	→	→	→	→	→	←	←	→
	`++a`												=	
	`--a`												+=	
`a++`	`+a`												*=	
`a--`	`-a`	*			<								/=	
`a()`	`!a`	/	+	<<	<=	==	&	^	|	&&	||	`x?y:z`	%=	,
`a[]`	`~a`	%	-	>>	>=	!=							<<=	
.	`(type)a`				>								>>=	
->	`*a`												&=	
	`&a`												^=	
	`sizeof a`												|=	

The arrow indicates if the operator is

1. left associative where
   `a + b - c + d` means `((a + b) - c) + d` and
   `a[]()->b` means `((a[])())->b`; and
2. right associative where `-!*a` means `-(!(*a))`.

Unfortunately, it becomes very difficult to read certain statements if you simply rely on precedence; for example, the following is taken from an encryption algorithm which we will cover later in this book:.

```
v1 += (v0 << 4 ^ v0 >> 5) + v0 ^ sum + k[sum >> 11 & 3];
```

Your goal is to use the above table to determine which of the following is the correct interpretation:

```
v1 += ( (v0 <<  4) ^ (v0  >> 5)) + (v0 ^  sum) + k[(sum >>  11) & 3 ];
v1 += (((v0 <<  4) ^ (v0  >> 5)) + v0) ^ (sum  + k[ sum >> (11  & 3)]);
v1 += (((v0 <<  4) ^ (v0  >> 5)) + v0) ^ (sum  + k[(sum >>  11) & 3 ]);
v1 += (  v0 << (4  ^  v0) >> 5)  + (v0 ^  sum) + k[(sum >>  11) & 3 ];
v1 += (( v0 << (4  ^  v0) >> 5)  + v0) ^ (sum  + k[(sum >>  11) & 3 ]);
v1 += ( (v0 <<  4) ^ (v0  >> 5)) + (v0 ^  sum) + k[ sum >> (11  & 3)];
v1 += (( v0 << (4  ^  v0) >> 5)  + v0) ^ (sum  + k[ sum >> (11  & 3)]);
v1 += (  v0 << (4  ^  v0) >> 5)  + (v0 ^  sum) + k[ sum >> (11  & 3)];
```

If you haven't figured it out by now, you will find your answer in a subsequent chapter. If you can appreciate the frustration of this exercise, you will remember the easier precedence rule suggested by Steve Oualline:

> Multiplication, division and modulo (%) come before addition and subtraction;
> put brackets around everything else.

This is also strongly emphasized in the JPL coding standard, under Rule 18 which says

> "In compound expressions with multiple sub-expressions, the intended order of evaluation shall be made explicit with parentheses."

2.3.2.2 Use interfaces for data structures

With an object-oriented programming language like Java or C++, encapsulation is part of the programming language: private member variables are simply not visible to tasks that use that data structure. It is often tempting, however, when returning to C to manipulate the data structures directly. For example, suppose you have a linked list and in your program, you are aware that you must immediately insert an object into that list. Normally, this would require two function calls:

```
single_list_t *lst = (single_list_t *) malloc( sizeof( single_list_t ) );
single_list_init( &lst );
single_list_push_front( &list, whatever_value );
```

You may reason as follows: why call two functions which first initialize the linked list to one that is empty, and then add a push front that also must check whether or not the current linked list is empty; why not just initialize the linked list to one that is not empty, thereby requiring fewer steps:

```
single_list_t *p_lst = (single_list_t *) malloc( sizeof( single_list_t ) );
p_lst->p_head = (single_node_t *) malloc( sizeof( single_node_t ) );
p_lst->p_head->p_value = whatever_value;
p_lst->p_head->p_next = NULL;
p_lst->p_tail = p_lst->p_head;
p_lst->size = 1;
```

If speed is that essentially, then implement an initialization routine that initializes a singly linked list with one item:

```
void single_list_init( single_list_t *const p_this ) {
    p_this->p_head = NULL;
    p_this->p_tail = NULL;
    p_this->size = 0;
}

void list_init_push( single_list_t *const p_this, void *p_first_item ) {
    p_this->p_head = (single_node_t *) malloc( sizeof( single_node_t ) );
    p_this->p_head->p_value = p_first_item;
    p_this->p_head->p_next = NULL;
    p_this->p_tail = p_this->p_head;
    p_this->size = 1;
}
```

Similarly, do not simplify statements

```
if ( !single_list_empty( p_lst ) ) {
    printf( "The first entry is at %p\n", single_list_front( p_lst ) );
}
```

to ones that immediately access the data structure:

```
if ( p_lst->p_head ) {
    printf( "The first entry is at %p\n", p_lst->p_head->p_value );
}
```

Certainly the second is faster, and perhaps demonstrates your superior understanding of the C programming language, but the function calls signal to another reader your intentions, as opposed to what you are actually doing:

1. If the singly linked list is not empty, print the address of the object at the front of the linked list.
2. If `p_lst->p_head` is not null, print what is at `p_lst->p_head->p_value`.

2.3.2.3 Functions are not only for code reuse

Functions can be used to reduce duplication, and as a general rule, if you find yourself writing code twice, that's often acceptable, but if you are writing the same code a third time, it is time to factor out that code as a function. But this is not the only purpose of functions: a function should perform a specific task.

Essentially, you can always ask yourself: "Does a significant block of code in a function body perform a complex operation that 1. produces either nothing or a single value, and 2. performs a task that can be summarized in a few words?" If so, that block of code is an excellent candidate for factoring out into a separate function

Consider this stack function that doubles the capacity of the internal array if the current one is too small:

```
void stack_push( stack_t *const p_this, int value ) {
    if ( p_this->size == p_this->capacity ) {
        int i;
        int *p_old_entries = p_this->p_entries;

        p_this->capacity *= 2;
        p_this->data = (int *) malloc( p_this->capacity*sizeof( int ) );

        for ( i = 0; i < p_this->size; ++i ) {
            p_this->p_data[i] = p_old_entries[i];
        }

        free( p_old_entries );
    }

    p_this->p_entries[p_this->size] = value;
    ++( p_this->size );
}
```

Over 50 % of the function is doing something other than pushing the new value onto the stack—instead, this entire block of code could be factored out as a separate function. Additionally, the condition `p_this->size == p_this->capacity` indicates what the code is doing, but creating a `stack_full` function indicates what the programmer intended.

```
void stack_double_capacity( stack_t *const p_this ) {
    int i;
    int *p_old_entries = p_this->p_entries;

    p_this->capacity *= 2;
    p_this->p_entries = (int *) malloc( p_this->capacity*sizeof( int ) );

    for ( i = 0; i < p_this->size; ++i ) {
        p_this->p_entries[i] = p_old_entries[i];
    }

    free( p_old_entries );
}

bool stack_full( stack_t *const p_this ) {
    return ( p_this->size == p_this->capacity );
}

void stack_push( stack_t *const p_this, int value ) {
    if ( stack_full( p_this ) ) {
        stack_double_capacity( p_this );
    }

    p_this->data[p_this->size] = value;
    ++( p_this->size );
}
```

Now, each function can be tested individually to determine if it works and the functionality of the actual push function is now quite clear.

As another example, when this author wrote a memory-allocation algorithm, he first sat down and wrote in pseudo code what he wanted to achieve:

```
void *memory_alloc( unsigned int n ) {
    unsigned int bucket, N, offset;

    // If the memory request is 0 or too large, immediately return
    // Convert the request to the correct bin number
    // Find a non-empty bucket containing available memory
    //  - if none is found, return NULL
    // Get a block of memory of the appropriate size
    // Set that block as a having been allocated and put back the unused portion
    // Return the physical address to the user
}
```

Even though none of these operations would ever be used elsewhere, the author converted each of these to a function and the final algorithm looked as follows:

```
void *memory_alloc( size_t bytes_requested ) {
    void *p_allocated_block;
    size_t bin_idx, N, offset;

    if ( (bytes_requested == 0) || ((bytes_requested + BUFFER_SIZE) > MAX_SIZE) ) {
        p_allocated_block = NULL;
    } else {
        // Find if a bin has an available block of memory
        //  - the most appropriate bin may be empty
        bin_idx = bytes_requested_to_bin_index( bytes_requested );
        bin_idx = find_non_empty_bin( bin_idx );

        if ( bin_idx == NO_AVAILABLE_BIN ) {
            p_allocated_block = NULL;
        } else {

            // Acquire a block of memory from the available bin
            memory_block = pop_bin( bin_idx );
            set_alloc( memory_block, 1 );
            split( memory_block, bytes_requested );

            p_allocated_block = memory_block_to_user_address( memory_block );
        }
    }

    return p_allocated_block;
}
```

Had all this code been placed into a single function, it would have spanned four pages, but as it is now, the execution of this function is much more readable, and each function performs a separate task which can be independently verified. When this code was developed, once all the functions were verified as working correctly, because the logic in the top-level routine was already correct, the allocation function worked flawlessly with the first execution. Had the memory allocation been written as a single function, no testing could be done until the body of the function was substantially complete, in which case, tracking down the plethora of bugs would be quite difficult.

2.4 Summary of real-time programming

In this topic, we've looked at failures in real-time systems, and considered mechanisms that can be used to overcome faults in programming languages. We've also looked at desirable characteristics of programming languages for both real-time systems and for operating-system kernels, and given justification for using C. Next, we have discussed some of the aspects of C that are important to this course together with a comparison and contrast with the implementation of data structures in C and C++. We then looked at some of the best programming practices as applied to the authoring of C, followed by an introduction to software engineering practices.

Problem set

2.1 Why do you think that there was such an outcry raised the insistence that structured programming be used by software developers as opposed to allowing software developers us traditional programming techniques based on writing optimal code.

2.2 Structured programming only requires conditional statements and condition-controlled iterative statements (loops). Thus, statements such as `break`, `continue` and `goto` do not constitute structured programming. Comment on the following different implementations as to whether or not it is worth breaking structured programming. Assume that the condition is initially false and once it is true, it remains true.

```
for ( i = 0; i < n; ++i ) {
    // Code block 1

    if ( condition ) {
       break;
    }

    // Code block 2
}
```

```
for ( i = 0; i < n && !condition; ++i ) {
    // Code block 1

    if ( !condition ) {
        // Code block 2
    }
}
```

```
for ( i = 0; i < n; ++i ) {
    // Code block 1

    if ( condition ) {
       continue;
    }

    // Code block 2
}
```

```
for ( i = 0; i < n; ++i ) {
    // Code block 1

    if ( !condition ) {
        // Code block 2
    }
}
```

```
for ( i = 0; i < n; ++i ) {
    // Code block 1

    for ( j = 0; j < n; ++i ) {
        //Code Block 2

        if ( condition ) {
            goto label;
        }

        // Code block 3
    }

    // Code block 4
}

label:  // Code block 5
```

```
for ( i = 0; i < n && !condition;; ++i ) {
    // Code block 1

    for ( j = 0; j < n && !condition; ++i ) {
        //Code Block 2

        if ( !condition ) {
            // Code block 3
        }
    }

    if ( !condition ) {
        // Code block 4
    }
}

label:  // Code block 5
```

2.3 The course notes shows both unstructured and structured implementations of insertion sort. Why do you think that the unstructured implementation compiles to a smaller set of instructions?

2.4 Procedural programming is based on describing functions where you specify:

1. the input data and its state, and
2. the transformation performed on the data and state.

How does this differ from object-oriented programming?

2.5 You can think of a function as the consequence of a sentence with an action verb. Take the following description of Dijkstra's algorithm and determine which components could be written as functions?

Loop:

1. Initialize a table setting the distance to each vertex as *infinity* and flag each vertex as unvisited.
2. Set the distance to the initial vertex as 0.
3. Find the unvisited vertex *v* with the shorted distance to it.
4. If no such vertex is found, we are finished.
5. For each unvisited neighboring vertex *w* of *v*,
    a. Calculate the recorded distance to *v* and the weight of the edge between *v* and *w*.
    b. If this calculated distance is less than the recorded distance to vertex *w*, update the recorded distance to *w*.
6. Flag the vertex *v* as visited.
7. Return to Step 3.

3 Computer organization

A program is a sequence of instructions. In order to execute the program, at the very least, you require two *resources*:

1. a processor, and
2. main memory.

The processor will execute the individual instructions and main memory is required to allow access to the instructions and allow access to and modification of data. For a computer to do something useful, it requires additional resources. In general, we will refer to any other resources as *devices* and the executing program will communicate with the processor through *device controllers* that are either accessed directly through:

1. specific instructions,
2. memory mapping (associating locations in memory with registers in the device controller), or
3. device drivers.

Additional resources may generally be classified as

1. storage devices: hard-disk, floppy-disk, flash, tape, and optical drives;
2. input devices: keyboards, mice, touch-sensitive screens, and microphones;
3. output devices: terminal screen (monitor), speaker, and printer; or
4. communication devices: serial and parallel ports, USB and Ethernet.

Note, however, today most devices connect to a computer through a USB port. Even keyboards are no longer purely input devices—settings and LEDs may be controlled by the processor.

Relevant to the material in this text, we will view a general-purpose computer or microcontroller as

1. one or more processors, each with possibly multiple cores, with each being able to execute instructions independently,
2. main memory, storing instructions and information necessary for computation, and
3. device controllers to communicate with other devices and computers.

We will begin by explaining why the processor and main memory are so central to computers, and then we will continue to look at other aspects. Before we explain why, let us look at the design of most processors today, by describing

1. Turing machines,
2. processor registers,
3. processor architecture,
4. main memory, and
5. operating systems.

We will now look at these.

3.1 The Turing machine

In 1936, prior to the first programmable computer being built (the German Z3 and British Colossus were developed independently in 1941 and 1943, respectively), Alan Turing defined the *Turing machine*. It is comprised of four parts:

1. The machine itself is in one of a finite set of states.
2. An infinite tape divided into frames, each of which could hold a single character in an alphabet. The tape can be accessed via a head that points to one frame on the tape and can read from and write to that frame as well as being able to move to either the next or previous frame.

3. There is a program that is a sequence of instruction. Each instruction maps a pair consisting of the current state and the letter stored in the frame currently under the head to a triplet consisting of a new state for the machine, a character to be written to the frame, and an instruction to either move to the previous frame, stay at the current frame, or move to the next frame of the tape.

Figure 3-1 shows a Turing machine and if the set of symbols is {0, 1, ␣} and the set of states is {b, c, d, e}, then transition table (instructions)

Current		Next		
State	Symbol read	State	Symbol written	Direction
b	␣	c	0	right
c	␣	e	␣	right
e	␣	f	1	right
f	␣	b	␣	right

will create an unending sequence of 0–␣–1–␣–0–␣–1–␣–⋯ Currently, the state in the figure is e and the symbol is a blank, so with the next transition, the third row indicates will set the state to f, write a 1 and move the head right.

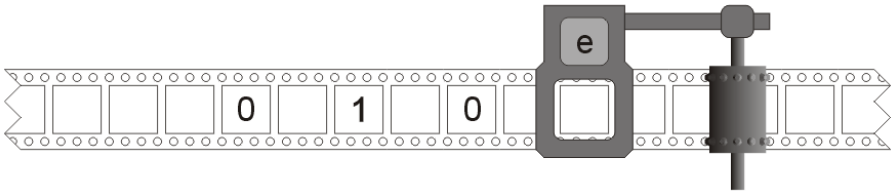


Figure 3-1. A mechanical Turing machine.

This sounds like a painfully tedious way of programming, but what is critical here is the subsequent Turing-Church conjecture: If an algorithm exists to solve a given problem, that algorithm can be implemented on a Turing machine.

We will see that the components of a Turing machine are built into our current-day computers with the following correspondences where

1. the state of the system is maintained through registers,
2. the infinite tape is main memory, and
3. instructions are assembly instructions that manipulate main memory and the registers created through the compilation of programming languages.

We will look at each of these components, and then give a quick overview of operating systems.

3.2 Register machines

The processor on most computers and microcontrollers contain a number of *registers*, each of which can store a fixed number of bits.

1. Some of these are data registers storing *words*; that is, the largest unit of data on which the arithmetic-logic unit can operate. In a 64-bit computer, a word is 64 bits, for example.
2. Others may store addresses that refer to locations in main memory (usually 16-, 32- or 64-bit addresses, although the microcontroller Freescale 68HC08 has only 13-bit addresses).

Many processors do not differentiate between data and address registers. There are other registers that the processor will use, including:

1. a program counter (PC) that stores the address of the next instruction to execute, and
2. a status register that stores information about the most recent instruction.

When you execute a command like,

```
++i;
```

the compiler will determine whether or not the value of that variable is already stored in a register or if it is a value stored in main memory (there are only a small number of registers, and a function may have many local variables). If the variable is stored in a register, it will simply increment that register. If it is not, that variable is stored somewhere in main memory, so it will first copy the value from main memory into a register and then add one to it. In either case, the compiler may or may not write that value back to main memory.

The status register will be updated to reflect such things as:

1. Is `i` now zero?
2. Is `i` positive or negative?
3. Did adding one to `i` cause a carry (unsigned) or an overflow (signed)?

Each of these Boolean flags would be stored by a single bit in the status register. Beyond this, we will not delve too much further into the functioning of the processor. As a mechatronics student, you will be using the processor as a tool; most of you will not be designing processors. Never-the-less, it is useful to understand why these two components are essential to programming.

3.2.1 Instructions versus data

The Turing machine makes a distinction between instructions and data where

1. instructions are to be executed in a specific sequence by the processor and are generally considered to be immutable, while
2. data is to be accessed by the processor and instructions use the data as operands and it should be possible to change these values.

The distinction between instructions and data will allow us to make different decisions when designing the architecture of a computer.

Consequently, one could envision a system with one set of memory being distinct from each other. For example, consider the Atari 2600, shown in Figure 3-2. The machine instructions are stored in read-only memory on cartridges purchased separately by the consumer, while the device itself only had random-access memory for run-time data.

Figure 3-2. An Atari 2600 with separate instruction memory (ROM stored on cartridges) and data memory (RAM). (Wikipedia users Evan-Amos and Locke Cole)

This setup, where instructions are stored separately from data, is described as a *Harvard architecture*.

3.2.2 Word size

The size of a data register said to be a *word* and these are usually 4, 8, 16, 32 or 64 bits. This generally defines the largest integer data type that can be operated on via the arithmetic-logic unit (ALU). Most processors in desktop and laptop computers today have 64-bit words. Some processors have 128-bit words, but these are rare. The LPC1768 has a word size of 32, but microcontrollers—depending on the application—may also have word sizes of 16, 8 and even 4 bits (such as the Epson S1C60 family of microcontrollers, the Amtel MARC4 and the EM Microelectronics EM6682 as used in a Braun electric toothbrush).

3.2.3 The registers in the LPC1768

The microcontroller we are using is the LPC1768. We will look at the registers in this specific processor, including

1. the general-purpose registers and
2. some of the special registers.

3.2.3.1 General-purpose registers

The general-purpose registers in the LPC1768 may store either data or addresses. These are identified as R0, R1, …, R15 and each of these can hold 32-bits. The ALU can only perform arithmetic or Boolean logic operations on values that are stored in these registers. In the examples, the italicized integers *m* and *n* represent values from 0 to 15, and any other italicized identifiers represent numbers. For example, one instruction is

```
ADD Rm, Rn;             % Rm = Rm + Rn
```

In some cases, you can specify the value that is to be added:

```
ADD Rm, #const;         % Rm = Rm + const
```

Normally, if you add two numbers and the result is greater than the largest number that can be stored, you get an overflow, in which case, the result will yield an unexpected value.

For example, consider the sum 89 + 42 as 8-bit signed integers.

```
 1011001
+ 101010
10000011
```

This number is, however, negative as the leading bit is '1', so to determine the value of this, we apply 2's complement to get `01111101`, or –125. There are special commands that perform *saturation* arithmetic

```
QADD Rm, Rn;            % Rm = (Rm + Rn > MAX_VALUE) ? MAX_VALUE : Rm + Rn
```

where if the sum of two values is larger than the largest value that can be stored, the result is that largest value, as opposed to the general behavior to wrap to the smallest value. For example, the sum of two 8-bit signed integers 89 + 42 would be `0111111`, or 127, which is the largest 8-bit signed integer.

As an example of another machine instruction that is likely not to be used by most C++ compilers is the logical `BIC`, or *bit-clear*, instruction

```
BIC Rm, Rn;             % Rm = Rm & ~Rn -- clear the bits in Rm of any 1 in Rn
```

This is not a course in assembly language programming; however, these maintain the *state* of the processor.

Note: One thing you may appreciate here is why writing in an assembly language usually results in faster code (at least for small blocks of code). Even if you wrote:

```
x &= ~y;
```

not all compilers would reduce this to a single instruction. Instead, they may make a copy of `y`, take a bit-wise complement, and then proceed to perform a bit-wise AND with `x`. Some compilers do have numerous algorithms for examining code to determine whether or not two or more instructions could be replaced by a smaller set of instructions, but inappropriate optimizations themselves cause problems, as we will see later. Additionally, many general-purpose compilers will simply not take into account machine-specific instructions, instead preferring to restrict the instructions generated to an almost universal subset.

To copy a value stored in one register into another, use the *move* command:

```
MOV Rm, Rn;              Rm =  Rn
MVN Rm, Rn;              Rm = -Rn
```

In addition to using these registers to store data, they can also store addresses. This is necessary to load and save the values stored in registers from and to main memory, respectively. The command

```
LDR Rm, [Rn, #offset];  Rm = *(Rn + offset)
```

loads the value stored at the address R*n* plus the offset into the register R*m*. Any arithmetic or logic operations will be transformed by the compiler into such a set of instructions.

Note: On many systems, the size of an address may be different from the size of a word. For example, the Motorola 68000 ("68k") has 16-bit data registers (the word size is 16 bits) but its address registers can hold 24 bits, that is, it can access up to 2^{24} memory locations. With each memory location being one byte, the maximum memory is 16 MiB. On such computers, the data registers are separate from the address registers, and so they are identified, respectively, as D0, D1, D2, …, and A0, A1, A23, …. In this case, the *width* of the data bus is 16 bits and the *width* of the address bus is 24 bits.

Of the sixteen general-purpose registers on the LPC1768, they may still be distinguished based on their use:

1. Registers R0 through R7 are *low registers* and are used by instructions that only allow three bits to specify the register.
2. Registers R8 through R12 are *high registers* and are used by instructions that allow four bits to specify the register.
3. R13 and R14 are involved in function calls, where
    a. R13 is a *stack pointer* (also MSP or PSP) and is used to track the values of parameters, local variables and other related information, and
    b. R14 is a *link register* (also LR) and stores the address where the function should return on the completion of the execution of that function.
4. R15 is the *program counter* (also PC) and it stores the address of the next instruction to be executed. You can execute a *goto* by changing the value of the program counter.

We will see more about registers R13 and R14 when we discuss static memory allocation.

3.2.3.2 Special-purpose registers

Beyond these general-purpose registers, addition memory is required for code execution. There are five additional special-purpose registers in the LPC1768 that contain various types of information:

1. The program status register is 32 bits that is subdivided into three groups:
    a. The application program status register (APSR), which is five bits that store details from the execution of the last instruction:
        i. Was it a negative number?
        ii. Was it zero?
        iii. Did a carry occur when adding two unsigned integers?
        iv. Did an overflow occur when adding two signed integers?
        v. Did a saturation occur when performing saturation arithmetic?
    b. The interrupt program status register (IPSR), which is used when external devices need to communicate with the processor.
    c. The execution program status register (EPSR), which stores the exception the processor is handling (if any).
2. There are three registers that deal primarily with interrupts, including PRIMASK (1 bit), FAULTMASK (1 bit) and BASEPRI (8 bits), and these will be discussed in Chapter 8.
3. The control register (CONTROL) is two bits that are used to provide a protected environment in which an operating system can execute (that is, when you have an operating system).

All of these values store the *state* of the processor at any one time. If there are no changes to main memory, then if we save the values of the registers, we can shut the processor down or do something else, and if we then restore all of the registers to the saved values, the next instruction will execute as if nothing happened in between.

3.2.3.3 Summary of the LPC1768

We've described quickly some of the registers used in the LPC1768. We will at some point discuss the state of a processor. This includes the values of all the registers in the processor.

3.2.4 Summary of register machines

We have briefly described register machines, the definition of a *word*, and how instructions affect the values of registers. We will now proceed to discuss the second aspect of a Turing machine: main memory. If you are interested, you could consider sitting in on a course such as ECE 222 *Digital Computers*:

> Computer organization. Memory units, control units, I/O operations. Assembly language programming, translation and loading. Arithmetic logic units. Computer case studies.

3.3 Main memory

Another aspect required by a Turing machine is some form of long-term memory that will, in our case, usually be represented as main memory. We will quickly describe some aspects of main memory as they relate to this course and the use of microcontrollers. Again, we will not often use main memory, at a low level, but rather an abstract level.

3.3.1 Addressing

Nominally, it would be easiest if the contents of a register could be copied back-and-forth between the processor and main memory through a single operation. Thus, one would expect that each word would be given its own address in main memory, in which case, main memory would simply be a sequence of words. For example, Figure 3-3 shows the first six words of memory on a 32-bit processor.

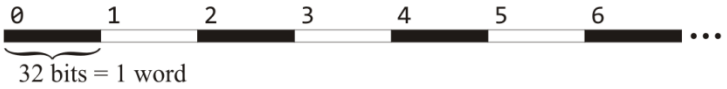


Figure 3-3. Individually addressed words on a 32-bit processor.

As an address is just another number, these are also stored in memory, so just like a word size, each processor will have its own address size, where the number of bits determines the largest number of words that can be accessed (if an address is *n* bits, up to 2^n words can be accessed). However, the fact that computing was developed primarily in the English-speaking United States and ASCII uses 8 bits to store a letter of the English alphabet (together with numbers, symbols and special characters), it was convenient to give each 8 bits (called a byte) its own address. Such memory is said to be *byte addressable*, as is shown in Figure 3-4.

Figure 3-4. Byte addressable memory.

Never-the-less, 16-, 32- and 64-bit processors will group bytes together, and therefore, even though you can specify byte 7, the processor will load the word containing the byte. For example, on a 32-bit processor, the bytes would be grouped into intervals of four bytes (representing one word), as is shown in Figure 3-5.

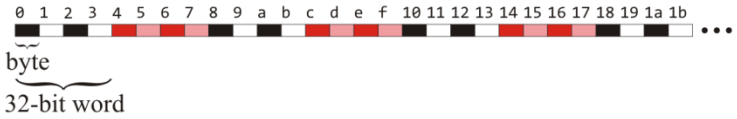


Figure 3-5. 32-bit words within byte addressable memory.

To contrast the near ubiquitous byte addressability, the low-power 4-bit microcontroller EM6682 is nibble (or half-byte) addressable—each four bits has its own address.

Note that the value stored in a byte may be represented by two hexadecimal numbers, from `0x00` to `0xff`, where the "`0x`" indicates that the following number is in hexadecimal. For example, in ASCII, the letters 'A' through 'Z' run from `0x41` to `0x5a` while 'a' through 'z' run from `0x61` to `0x7a`, and the numbers '0' through '9' run from `0x30` to `0x39`. To access individual bits, we must use bit-wise operations as was previously discussed.

While the address and word size are the same on the LPC1768, this isn't true in general. For example, the earlier Motorola 68000s (also known as the M68k) had 16-bit words, but as memory is byte addressable, having 16-bit addresses would restrict the size of memory to 2^{16} bytes or 64 KiB. This was an insufficient amount of memory, and therefore 24 bits were used for addresses, and therefore the maximum amount of memory would be 2^{24} = 16 MiB (in 1979, 64 KiB of RAM cost around $400 so 16 MiB would be upwards of $100,000, not adjusted for inflation).

The next question is how is data transferred between the registers and main memory. This is done through a *data bus*. The width of the data bus is the number of bits that can be transferred in parallel between registers and main memory. An example of these is shown in Figure 3-6.

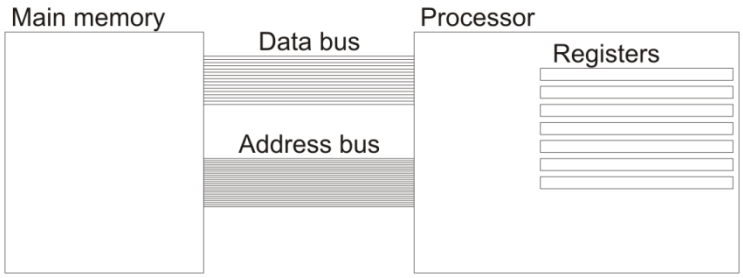


Figure 3-6. The data and address buses between the processor and main memory.

The size of the address bus must equal the size of the address registers. These are used to indicate the address of where data that is to be either read or written. The size of the data bus often equals the size of a register (the word size); however, this may not always the case. In the M68k, the data bus is 16 bits while the word size is 32 bits. Consequently, it requires two separate instructions to load a word from main memory; this, however, significantly reduces the cost of the processer (reducing the number of pins, simplifying the traces, etc.)—a significant issue when dealing with embedded systems.

In addition to separate data and address buses, there is a third control bus that is used to signal:

1. that main memory is being read,
2. that main memory is being written to, and
3. the number of bytes being read or written.

In the last case, a 32-bit data bus could be used to write, for example, 8 or 16 bits instead of all 32. This is shown in Figure 3-7.

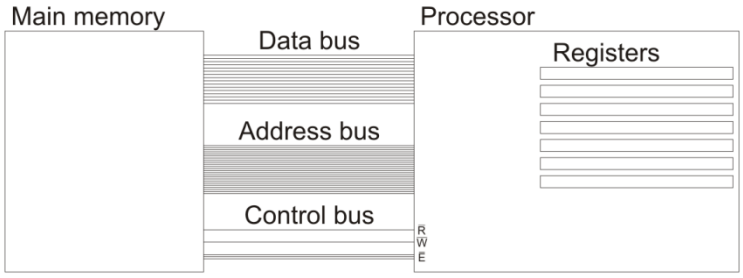


Figure 3-7. The data, address and control buses.

If the word size and address size is different, the processor will require separate data and address registers. However, once we get to 32-bit and 64-bit processors, it is often easier to have the word size equal the address size with both data and address buses being 32 bit. Therefore, a 32-bit processor (including the LPC1768 microcontroller we are using) will also have 32-bit addresses, thus restricting memory to 2^{32} bytes or 4 GiB. A 64-bit processor will have 64-bit addresses, and thus as many as 16 777 216 TiB can be addressed.

Note: most computing today does not require 64-bit processors, at least with respect to the size of a word—32-bit integers are more than sufficient for almost all applications, so moving around 64 bits is unnecessarily expensive. The greatest benefit, perhaps, for 64-bit processors is that now a double-precision floating-point number (`double`) can be loaded or saved in a single fetch from or store to main memory, respectively. The primary benefit is in the address space: a 64-bit address can access 16 777 216 TiB of main memory, whereas 32-bit computers were restricted to 4 GiB of main memory.

In general, we will represent addresses as hexadecimal numbers. For example, a 24-bit address will run from `0x000000` to `0xffffff` and a 32-bit address will run from `0x00000000` to `0xffffffff`. Most of our examples will use 32-bit addresses; however, examples on Linux systems may have 64-bit addresses. The structure of busses between processors and memory is covered in other text on microprocessor interfacing.

One consequence of the discrepancy between word size and bytes is that, for example, a 32-bit machine will not load one byte at a time; rather, given an address, it will load four bytes, those bytes with addresses ending with bits `00`, `01`, `10` and `11`. Consequently, if you will recall from the previous topic, the compiler is very careful to separate out fields in a structure so as to avoid poor placement. Suppose that a 4-byte integer is stored at addresses `0xf3230255` through `0xf3230258`. In this case, the last two bits are `01`, `10`, `11` and `00`, and this will require two fetches from memory: one for the 32 bits stored at `0xf3230254` and one for the 32 bits stored at `0xf3230258`. Additional instructions will be required to combine the two values in a register. Such processors are said to be *word aligned*.

Note: word alignment is not always required (for example, processors in the x86 line) but microprocessors will tend to have word alignment as it simplifies the system.

3.3.2 Byte order

Another peculiarity that results from having byte addressability is the question: in a 32-bit integer, which byte comes first. Of course, the "obvious" answer is the most significant byte or *biggest* part of the integer. For example, one would expect that `0x12345678` would be stored as four consecutive bytes with the values `0x12`, `0x34`, `0x56` and `0x78`. However, suppose you are adding two integers: the arithmetic-logic unit does not have the hardware to automatically add 32-bit numbers. Instead, this is converted into the adding four byte-sized integers, possibly with carries. In that case, would you not want to add the least significant bytes first, then the next least significant, and so on? For this, and many other reasons, some processors store the *littlest* byte first.

Systems that store the most-significant (or "biggest") byte first are said to be *big endian*, while systems that store the **l**east-significant (or "*littlest*") byte first are said to be ***l**ittle endian*.

Thus, using big endian, one billion, or $111011100110101100101000000000_2$ would be stored in main memory as

00111011	10011010	11001010	00000000

while using little endian, it would be stored as

00000000	11001010	10011010	00111011

Intel uses little endian while Motorola uses big endian. ARM processors allow you to decide which endian format is used.

This only matters if you are doing byte-wise operations on data types that are greater than one byte in size.

3.3.3 Accessing memory

The actual connection between the processor and main memory is often through additional registers that are directly connected to the data and address buses.

1. the memory address register (MAR) contains as many bits as the width of the address bus, and an address (or general) register is copied to this register prior to memory being read or written to,
2. the memory data register (MDR) contains as many bits as the width of the data bus, and
    a. if memory is being written to, the data to be written to memory is first copied to this registers, otherwise
    b. if memory is being read, once the read operation is complete, this register will contain the value in memory, which can then be copied to another register in the processor.

Once these two registers are ready, the control lines will be appropriately signaled and the main memory controller will deal with accessing and either reading or writing to the specified memory address.

3.3.4 Buses

The connection referred to in the previous sections as a bus (from *omnibus*, Latin for "for all") is, in general, used to connect most peripherals with the processor. In general, you can think of a bus as a single communication system between components in a computer. A computer may have more than one bus, but the added cost is often prohibitive. Instead, all components (the processor, main memory, and other peripherals) will communicate via the bus and, as only one device can use the bus at any one time, protocols must be in place to ensure only one device uses the bus at a time. For further details on buses, see any text on microprocessor interfacing.

3.3.5 Memory allocation

Any program that wants to run requires memory. The available memory must be in some way allocated when it is required. In the simplest case, only a single task may be running on a processor, in which case, all memory can be used however that task implements it. Normally, however, memory is a shared resource between many tasks, and there must be some central mechanism for allocation that memory to tasks as the memory is required. At the same time, there must be a mechanism for collecting that memory once it is no longer required, such as after the termination of a task.

In some cases, it is possible for the compiler to determine the location or relative location of the memory allocations required for a task. For example,

1. the instructions comprising the program can be placed in a single segment called the *code segment* (sometimes called a *text segment* as a book is read-only), and
2. any constants (numeric constants, strings, etc.) or static variables required by a program can be placed together in subsequent segment of memory called the *data segment*.

Thus, memory may be allocated by the compiler for these two segments, as shown in Figure 3-8.

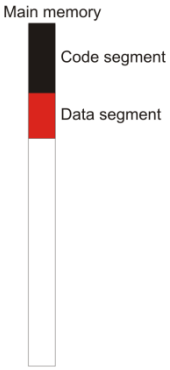


Figure 3-8. Memory allocation of code and data segments.

There is, however, another case where the compiler can determine how much memory is required:

When a function is called, the compiler knows how many local variables it has, how many parameters it has, and the arguments that were passed. Consequently, it should be possible to determine how much memory is being used by a function at any one point. This is another situation where the memory allocation is determined by the compiler, on a per function basis. As function calls have a stack-like behaviour (function A calls function B, and when function B returns, it returns to function A), this can also be exploited with respect to memory allocation (the memory required for function B is allocated immediately following the memory allocated for function A, and when function B returns, its memory can be deallocated). We will describe this more in detail in the next topic; however, we will see that we can visualize such allocation as if it were on a growing stack, as shown in Figure 3-9.

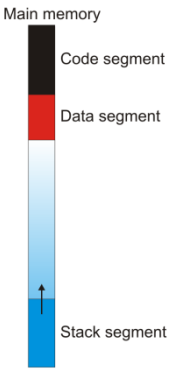


Figure 3-9. Memory allocation with a stack segment.

The one case where the compiler cannot deal with memory allocations is when it deals with any interaction with another party: when the compiler generates the code, it does not know how many or when messages are received, or how many documents are being generated by a user, or how many clients are requesting a particular resource. In these cases, there needs to be some other mechanism for memory allocation that can occur at run-time, or *dynamically*. We will see how we can take a single large block of memory (we will call it a *heap*) which can be dynamically allocated as necessary. This is a not-necessarily contiguous region which we can visualize as growing from the data segment, as shown in Figure 3-10.

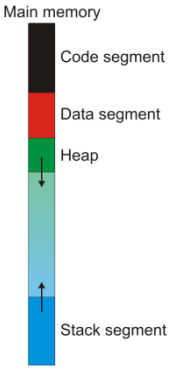


Figure 3-10. Memory segmentation including the dynamic heap.

3.3.6 Summary of main memory

This topic briefly introduced some of the more obvious issues that are relevant to main memory:

1. most processors use byte addressing although fetches may be word aligned,
2. words may have their bytes ordered from most significant to least significant (big endian) or least significant to most significant (little endian) byte order,
3. accessing memory is through the memory address and data registers (MAR and MDR),
4. buses connect the processor and main memory (as well as other peripherals), and
5. a high-level description of memory allocation.

Once we discuss other peripherals, we will look at other issues such as memory-mapping and direct memory access. We will now, however, look at the high-level relationship (architecture) of the computer.

3.4 Processor architecture

A processor with registers that hold the state, with access to main memory, and where instructions transition the current state into a new state is equivalent to a Turing machine. Consequently, the core functionality you require to execute an algorithm is the processor and main memory; everything else is for utility or convenience. Thus, the core of any computer is the processor and its memory, as shown in Figure 3-11.

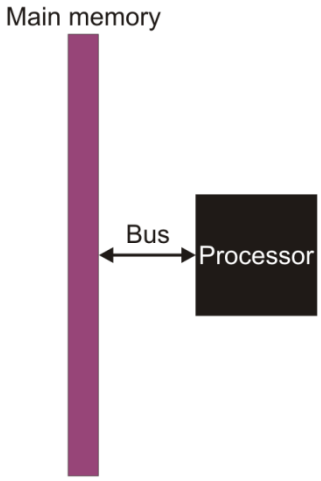


Figure 3-11. A processor and memory: the critical components of a computer.

The manner in which the components of a computer are connected is described as the architecture. We will describe the

1. design of microprocessors and microcontrollers,
2. Harvard architecture,
3. von Neumann architecture, and
4. Cortex-M3 architecture.

We will start by describing the difference between microprocessors and microcontrollers and then look at the various architectures of connecting these.

3.4.1 Microprocessors and microcontrollers

Desktop and laptop computers have separate processors and memory: a processor may be replaced by a faster one, while more memory can be added to the memory banks. For an embedded system, having separate processors and memory, however, has sufficiently many drawbacks that it is often better for producers to make microcontrollers that contain both a processor and main memory all on the same integrated circuit; that is, a collection of electric circuits (resistors, capacitors, inductors, transistors, diodes, etc. connected by traces) on a single plate of a semiconducting material, usually silicon. A microcontroller will also have additional peripherals (for example, system clocks, non-volatile memory (ROM) and other communication interfaces) built into the same integrated circuit whereas the same would be found, perhaps, on the motherboard of a general-purpose computer.

3.4.1.1 Microprocessor

A microprocessor is essentially a register machine with limited capabilities, including:

1. an arithmetic-logic unit in order to perform integer arithmetic and Boolean operations,
2. a floating-point unit for performing floating-point arithmetic (not always present),
3. a system clock, the cycle of which times the execution of instructions, and
4. a control unit that regulates the operations of the processor.

Previously, floating-point units were separate integrated circuits (chips), but today they are usually integrated into the same chip. Communication to other devices is through a bus or other pins.

Prior to microprocessors, the central processing unit of a computer would have consisted of circuit boards with hundreds if not thousands of interconnected circuits. Reducing the manufacturing of the processor to handful (or one) integrated circuit greatly reduced the costs. By not integrating main memory onto the chip, the unit cost was further reduced and greater flexibility was provided for the end user. Examples include the Intel x86, i386 and x86-64 families of microprocessors and the Motorola 6800, 68K and PowerPC families of microprocessors.

3.4.1.2 Microcontrollers

Suppose we wanted to build an embedded system. We could create a board with a microprocessor, but that board would also have to contain a number of other integrated circuits, including (at the very least)

1. main memory for dynamically changing variables,
2. flash memory or read-only memory for instructions and constants,
3. a real-time[10] clock, and
4. some form of input and output.

Ultimately, there is a significant cost involved per embedded device to combine these integrated circuits on a printed circuit board (PCB). This would involve numerous additional costs in design, quality assurance, testing and maintenance; for example, see Figure 3-12.

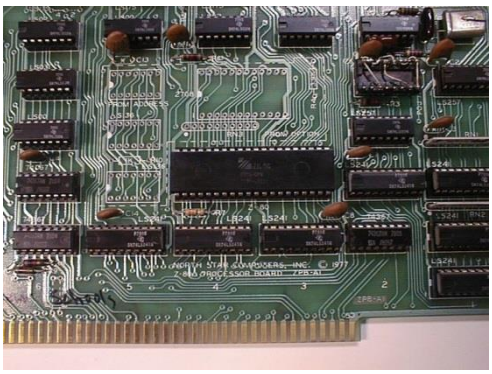


Figure 3-12. Multiple integrated circuits on NorthStar Horizon Z80 processor board (photograph by Wikipedia user Deron Meranda).

Instead, a microcontroller (MCU or μC) contains a significant number of components on the same die that would otherwise be peripheral integrated circuits in, for example, a desktop or laptop computer. The first microcontroller was designed in 1971 at Texas Instruments (TI) by the engineers Gary Boone and Michael Cochran. The 4-bit TMS 1000 included, in addition to the processor, read-only memory, read/write memory and a clock on one integrated circuit.

A system-on-chip (SOC) is usually used to refer to more powerful microcontroller, often with sufficient resources to run general operating systems such as Linux. A familiar example of a SOC is the Broadcom BCM2835 that forms the core of the Raspberry Pi, shown in Figure 3-13. This SOC includes a 700 MHz ARM1176JZF-S processor, a VideoCore IV graphics processing unit (GPU) and 256 or 512 MiB of RAM, but unlike the LPC1768, the Raspberry Pi does not have a real-time clock. Broadcom Corporation uses a model similar to that of ARM Holdings plc in that it licences the design of the Raspberry Pi to manufacturers.

[10] It is exceptionally unfortunate that *real-time* here means *actual-time* in contrast with the timer that signals the cycles of the processor.

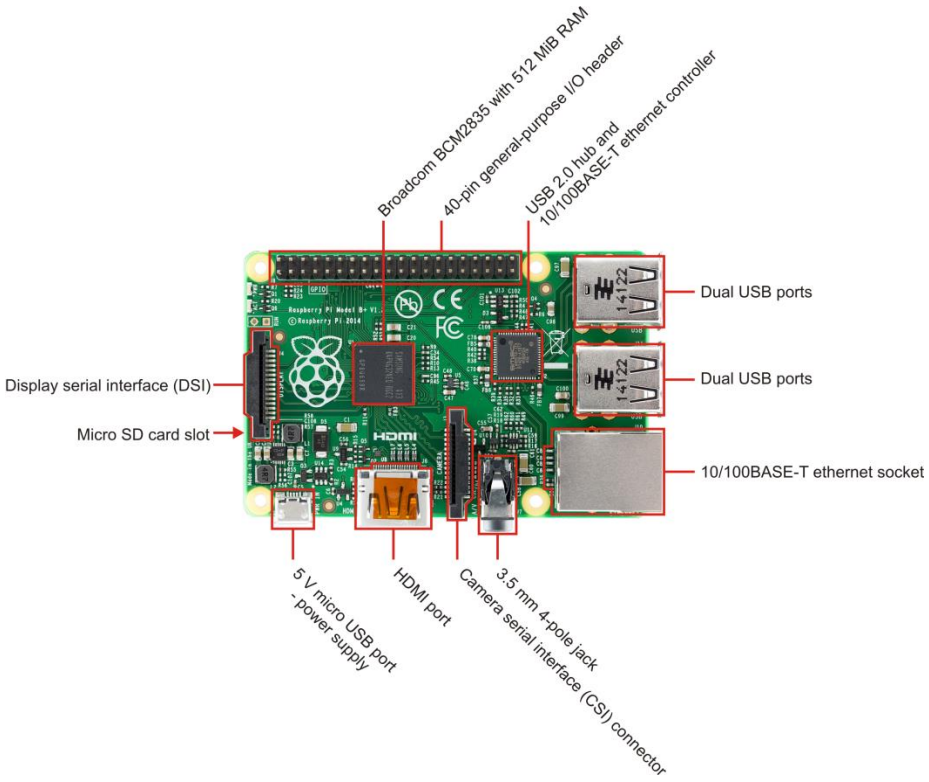
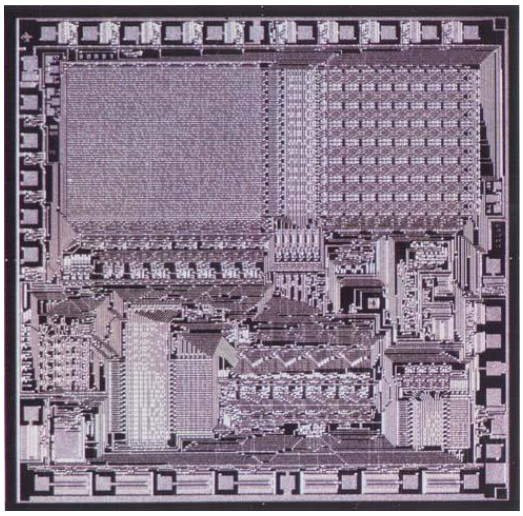


Figure 3-13. Components of a Raspberry Pi B+, augmented from a photograph by Lucas Bosch.

A digital-signal processor is a microprocessor dedicated to measuring, filtering or compressing analog signals in real time by converting the input analog signal into a digital signal, performing the appropriate operations, and converting the output back into an analog signal. Some microcontrollers have digital-signal processing hardware built into the chip.

In *State of the Art: A Photographic History of the Integrated Circuit* (see http://smithsonianchips.si.edu/augarten/), Stan Augarten describes the first microcontroller: the TMS 1000 by Texas Instruments. At the time that he wrote his book, 1983, it was the most widely used *computer-on-a-chip*, as well as being the first to integrate RAM, ROM and I/O onto a single chip together with a microprocessor. The team that designed this chip was led by Gary Boone and Michael Cochran, but while they developed the chip in 1971, rather than making it available as a consumer item, its first use was in a calculator introduced in 1972. The version of this chip shown below contains a 128 bytes ROM in the top-left quadrant, a 32 byte RAM in the top-right quadrant, with the arithmetic-logic unit, controller, and other components below. The actual size is 0.310 cm × 0.363 cm, or .



3.4.1.3 What is firmware?

Firmware refers to software that is critical for the operation of hardware, and is therefore usually stored in read-only memory (ROM) which may be read directly or may be loaded into main memory when the system is turned on. In most hardware architectures, a firmware program is first loaded into main memory as part of setting up an infrastructure for the program to execute. Once this is completed, the program counter is set to the address of the first instruction and the program begins executing. In smaller systems, usually embedded, the program itself may be written into read-only memory. This removes the loading process and therefore the set-up time is reduced. As part of the boot process, instructions in ROM may perform task such as:

1. performing a power-on test,
2. reading configuration parameters from CMOS memory, and
3. loading a *bootstrap loader* from a *boot sector* of a *boot device* into main memory.

This bootstrap loader will now load a second-stage loader (*e.g.*, GNU GRUB, BOOTMGR, Syslinux), which in turn will load an operating system. Firmware is in many cases upgradable, but this requires additional hardware to flash the existing ROM.

3.4.2 Harvard architecture

An *architecture* describes, at a high level, the parts of a computer and their relationships. The *Harvard architecture* is based on the design of the Harvard Mark I computer, designed in 1939 and built in 1944, shown in Figure 3-14 where instructions and data reside in separate memory and are accessed via separate buses.

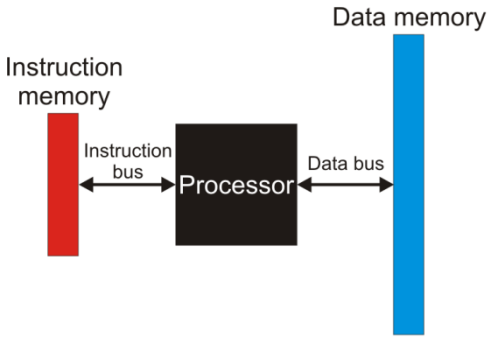


Figure 3-14. The Harvard high-level architecture.

The first programmable computer was developed by the German civil engineer Konrad Zuse also used this approach. His "Z3" electromechanical computer had the instructions read from tapes (this computer was also Turing complete). As another example of a Harvard architecture, the 4-bit EM6682 has a 4-bit data bus and a 4-bit address bus, but with 72 (> 2^6) instructions, it requires two fetches per instruction, or two cycles per instruction (CPI).

An alternative architecture appeared a few years later.

3.4.3 von Neumann architecture

The Harvard architecture was the approach with many early computers, and it was not until 1945 when John von Neumann published his *First Draft of a Report on the EDVAC* that lead to an architecture that saw a single main memory which would contain both instructions and data. This came to be known as the von Neumann architecture; however, this was based on the work of researchers both at Princeton University and elsewhere. This is the architecture used in most computers and microcontrollers today.

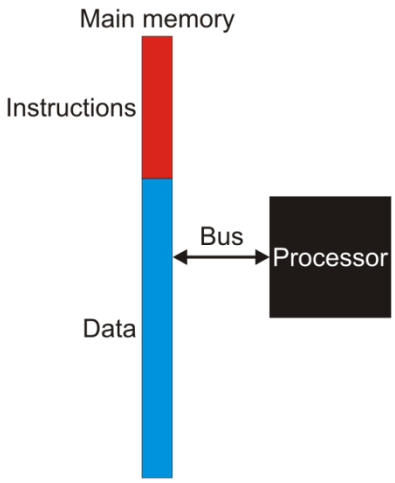


Figure 3-15. The von Neumann high-level architecture.

Having a single bus connecting the processor to main memory has as its consequence that instructions cannot be fetched simultaneously with data. Consequently, this can severely restrict performance.

Aside: for your interest only, the following is from the seminal paper written by John von Neumann.

It is evident that the machine must be capable of storing in some manner not only the digital information needed...but also the instructions which govern the actual routine to be performed on the numerical data... Hence there must be some organ capable of storing these program orders. There must, moreover, be a unit which can understand these instructions and order their execution.

Conceptually we have discussed above two different forms of memory: storage of numbers and storage of orders. If, however, the orders to the machine are reduced to a numerical code and if the machine can in some fashion distinguish a number from an order, the memory organ can be used to store both numbers and orders. The coding of orders into numeric form is discussed in 6.3 below.

If the memory for orders is merely a storage organ there must exist an organ which can automatically execute the orders stored in the memory. We shall call this organ the *Control.*

3.4.4 The Cortex-M3 architecture

The microcontroller we will be working with, the NXP (from *Next Experience*) LPC1768 microcontroller is based on the Cortex-M3 architecture, a design that blends the von Neumann and Harvard architectures: all instructions and data are stored in main memory, but part of main memory is accessible by a second instruction bus. If the entire program can be fit into this sub-section, instructions may be fetched simultaneously with data instructions. While this leads to a much more complex architecture, it reduces the effect of the von Neumann bottleneck, at least with respect to fetching instructions.

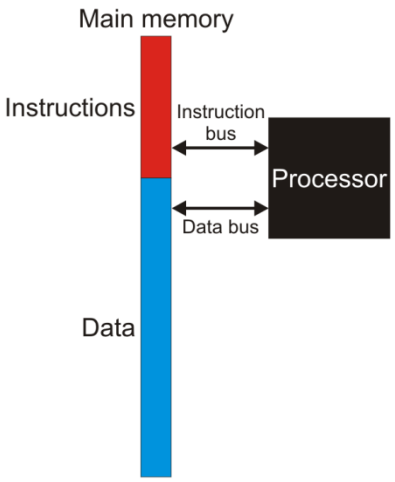


Figure 3-16. The architecture used in the Cortex-M3.

3.4.5 Architecture summary

This concludes a brief overview of various architectures, specifically the architecture used by the microcontroller in our lab. Next we will describe the purpose of an operating system.

3.5 Operating systems

Consider again this program:

Program 1. Static and dynamic memory allocation.

```
#include <stdlib.h>
#include <stdio.h>

int main( void ) {
    int m = 4;

    int *p_n = (int *) malloc( sizeof( int ) );
    *p = 5;

    printf( "The address of 'm':          %p\n", &m   );
    printf( "The value of 'm':            %d\n",  m   );
    printf( "The address of 'p_n':        %p\n", &p_n );
    printf( "The value of 'p_n':          %p\n",  p_n );
    printf( "The value stored at 'p_n':   %d\n", *p_n );

    free( p );

    return 0;
}
```

The program is running, and memory has been allocated by the operating system. Following this, instructions are sent to the terminal to print the results, and the terminal, in turn, makes those results appear in a window in a graphical user interface. But what is an operating system?

The operating system is a manager for the resources available on a computer.

The operating system is not a graphical user interface, it is not a command line interface, it is a collection of data structures and functions that manage the resources available. These resources include:

1. available processors or cores,
2. main memory,
3. other storage devices (secondary memory),
4. input devices,
5. output devices, and
6. communication devices.

We will focus on the allocation and effective use of the two primary resources in this course, namely the processors and main memory, and how these may be effectively used in real-time situations.

3.5.1 Why do we need an operating system?

In short, we don't always need one. You can load a program into memory of the LPC1768 microprocessor that you will be using in your laboratories, ensuring that the first instruction is at memory location `0x00000000`, and when you reset the processor, it will begin executing your program. This is all taken care of by compiler and loader the µVision4 integrated development environment (IDE). To understand the purpose of operating systems, let's review a history of the development of computing.

The first programmable computers worked as follows: you loaded a program (initially by rewiring the computer and later with punch cards), and then ran it. When the program finished (or time ran out), you would collect the output and the next program would be executed.

There are, of course, benefits to such an environment: it is very fast, as the program is the only executable that is running. The program has access to all of memory, all resources available to it, etc. There are issues, however:

1. most real-world systems do not require such speed, and

2. it was found a lot of time was spent waiting for input or output.

For example, suppose you write such a program and you are waiting for input from some device. How do we communicate with that device? Originally, a device may have been directly connected to the processor with specific instructions for communicating with the device. In this case, the device may have a bit which, when set, indicates that data is ready to be read off of that device.

```
if ( device_A_ready() ) {
    int value = device_A_read();
    process_A( value );
}
```

If you are specifically waiting for the device, you may try something like:

```
while ( !device_A_ready() );  // Loop

int value = device_A_read();
process_A( value );
```

In the second case, the processor could spend a significant amount of time essentially doing nothing as it waits. It would be really nice if the device could flag the executing program to signal it had data, but what happens to the program if it was running?

Looking ahead: Later, we will look at two different solutions. First, we will discuss the use of *communication buses* to allow the processor to transfer data between it, memory and other devices. Second, we will look at the use of *hardware interrupts* that will allow devices to signal that some goal has been accomplished. We will briefly describe hardware interrupts here, but only as a brief overview.

When a processor is executing a program, suppose that there was processor support to do the following:

1. when a device signals that it is ready for data, the processor saves the state of the processor,
2. another function is called that can deal with the device that has signaled it is ready,
3. the function deals with the data appropriately, and
4. the processor is returned to the exact state it was in immediately prior to the signal.

In this case, the processor would continue executing as if nothing happened. The next instruction would execute in exactly the same manner as if nothing had happened. This is because a processor is *deterministic*. If two processors are in the same state, they will continue execution in the same manner. The only time that it will affect execution is if the data accessed from the device, at some point, affects the next instruction.

Fortunately, we will see that this is such an elegant solution that most processors have support for such a mechanism.

Now, you could code for this, but such code would have to be very carefully written: any error in saving or restoring the state of the processor would result in *non-deterministic* errors. For example, the program could run perfectly well nine times out of ten, or 99 times out of 100 or 999,999 times out of one million, but if the signal occurred at exactly the wrong time, it could result in an incorrect result. Trying to find such bugs is exceptionally tedious work: in one such case, a program was run on every computer (in the background) at a place I worked repeatedly over the course of a few days before it finally crashed and produced a usable *core dump* (an error report) that could be investigated.

Now, suppose one function is necessarily waiting for a response from a device. For example, suppose that a function is executing and it then requests that a particular block from a hard drive be loaded into main memory. The run-time of such a request depends on the speed of the disk. A disk spinning at 7200 RPM will have an average seek time of approximately $\frac{1}{\frac{7200\text{ RPM}}{60\text{ s/min}}} \Big/ 2 = \frac{1}{240\text{ s}^{-1}} = 0.00416\overline{6}\text{ s} \approx 4\text{ ms}$ with a worst case of 8 ms. In 8 ms, a 3 GHz processor

could execute 24 million instructions. Certainly if a function is waiting that long, would it not be more appropriate to allow another function to start executing? For example, if the request started the process of copying information from the hard drive to the address specified.

```
char *p_memory_address;  // the address to where the block will be loaded
hard_drive_load_block( block_id, p_memory_address );

while ( !hard_drive_ready() ) {
    // Perform other tasks
}

char value = memory_address[k]; // Access the kth byte
```

Again, this would have to be coded very carefully to ensure that other tasks can be performed. Wouldn't it be easier if we could just start executing another function until the block was copied? For example, what happens if it is essential that we access the value as soon as it is loaded from the hard drive? If our loop was not well designed, it might take some time to finish execution until the next check.

The ability to switch between two functions is called *multiprogramming*. This was first done in 1954 on a computer called the LEO III (Lyons Electric Office).

Multiprogramming is just one of many ways of sharing the processor between multiple tasks all wanting to execute on that processor. Others include time sharing and real-time systems. We will look at these later, but we will group all of these together as multitasking.

Note: Processors for microcomputers appear to have capped out at approximately 3 GHz. This is for a number of reasons, but a good discussion is available here:

http://www.technologyreview.com/view/421186/why-cpus-arent-getting-any-faster/

Essentially, there are other bottlenecks that are more critical at this point, including power (too hot), memory (access is too slow) and instruction-level parallelism (optimizations and pipelining).

3.5.2 Uses of operating systems

An operating system is a manager of resources of a computer and it is written to deal with exactly such issues. Rather than embedding all of the resource management software into your program, the operating system deals with issues such as

1. dynamic memory allocation,
2. device communications, and
3. multitasking.

Note that, in essence, these three cover the gambit of available resources:

1. main memory (dynamic memory allocation),
2. available processors or cores (multitasking),
3. other storage, input, output, and communication devices (device communication).

This is a standard engineering approach to any problem: divide the larger problem into independent sub-problems and develop solutions for each of the sub-problems: the issue of resource management has been factored out of the programming problem.

3.5.3 Linux, POSIX and the Keil RTX RTOS

When Unix was first developed, it evolved into a number of different flavors, each developed by separate vendors. While each underlying operating system was reasonably portable, code written for one flavor would likely require significant rewriting to run on a different flavor of Unix. Consequently, the Portable Operating System Interface for Unix (POSIX) standard was created. This defined a common interface so that any program that accessed the operating system strictly through this interface could (theoretically) be run on any other platform. Linux implements the POSIX interface and we will use this heavily during our lectures as examples. In the lab, you will be able to contrast this with the Keil RTX RTOS, which you will be using in the laboratories.

POSIX (Portable Operating System Interface) is a collection of IEEE standards specified for maintaining compatibility between operating systems. POSIX defines

1. the application programming interface (API), and
2. command line shells and utility interfaces.

Originally, as the name suggests, POSIX was targeted at providing cross-platform compatibility between variants of the Unix operating system, but it is now also implemented in numerous other systems.

3.5.4 Real-time operating systems

The original Linux scheduler (the program that decided what runs next) could, in its worst-case scenario consider every single process that could be scheduled. Thus, the run time was linear (O(n)) in the number of tasks. Such a response time is not real-time. The first criteria for a real-time operating system is:

> **All services provided by the operating system must have bounded as well as reasonable and consistent response times and memory requirements.**

Real-time operating systems will also, in general, provide two other services:

1. A mechanism to ensure that the most critical process is the one that is currently executing, and
2. A mechanism for dealing with requests from other devices.

Throughout this course, we will investigate all of these.

3.6 Computer organization summary

We have discussed the concept of a register machine, described a Turing machine, considered multiple architectures possible for computers and microcontrollers, and then considered the purpose of operating systems, both in general and for real-time systems.

Problem set

3.1 In general, resources could be classified as those where:

1. information is uni-directional, either
    a. flowing to the processor, or
    b. flowing from the processor; and
2. information is bi-directional.

The classifications 1a and 1b are usually referred to as input and output, respectively. Why do we break the second classification into *storage devices* and *communication devices*? After all, isn't a storage device just something that is communicated with?

3.2 Given a Turing machine where our infinite tape is as follows:

	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	

where the *head* is located at the cell denoted by pink. The state of the machine can be either A or B, and currently the state is A. The instructions are as follows:

Current state		**Operation**		
Value under head	**State**	**Value to write**	**New state**	**Move head**
0	A	1	B	R
0	B	0	A	R
1	A	0	B	R
1	B	1	A	R

What does this program do to the tape after instructions are executed 14 times?

You should get a tape as follows, with the head at the last shown cell.

	0	0	0	1	0	1	0	1	0	1	1	0	1	0	1	0	1	1	

3.3 Most desktop and laptop processors have 64 registers. The 6800 has two registers. Is it possible to have a processor with just a single data register? (Your argument should use the requirements of the Turing machine.)

3.4 Which of the following is the correct definition of the *word* size of a processor?

1. It is the width of the bus: the amount of data that can be transferred between the processor and main memory.
2. It is the width of the data registers: the amount of data that can be operated on by a single instruction.
3. The width of the bus equals the width of the data registers, so the word size is both of these.

3.5 The address bus is 20 bits wide. What is the maximum amount of main memory that can be accessed by such a bus?

3.6 If the word size on a processor is 16, 32 or even 64 bytes, why do we still keep memory that is byte addressable (as opposed to word addressable)?

3.7 What are the benefits of having the word size equal the address size? Why might such a requirement be detrimental to the cost of an inexpensive embedded microcontroller?

3.8 What does the following test?

```
#include <stdbool.h>

bool test() {
    int a = 1;
    char *p_char_a = (char *)( &a );

    return (*b_char_a) == 0;
}
```

3.9 What is the difference between a von Neumann architecture and the Harvard architecture?

3.10 What is primary drawback of the von Neumann architecture? Why is this not so much an issue with a desktop or laptop?

3.11 What is the primary benefit of the Harvard architecture with respect to power consumption?

3.12 What is the difference between Linux/Unix and POSIX?

3.13 Does a real-time operating system need to be fast?

4 Static memory allocation

All real-time and embedded systems require data acquisition from sensors, and that data must be stored and processed. Data can be categorized in terms of either temporary or persistent:

1. temporary data is that which must be reacted to, but once the action is performed, the data is no longer required, and
2. persistent data is that information that is being collected by the system for long-acquisition or subsequent data transfer to another system.

In either case, data should be moved as seldom as possible. Ideally,

1. temporary data is read into *local memory* and then discarded or into *global memory* or *dynamic memory* where it is subsequently overwritten or the memory is released and reused, while
2. persistent data should only be read into *global memory* or *dynamic memory*, and if it is to be transferred to another system, it should be transferred from that memory.

Memory is allocated in one of two ways:

1. In some cases, the compiler can make decisions about where to allocate memory. It may be either at an absolute address or at a relative address, but the need for such memory must be discernable from the code at compile time, and this is termed *static memory allocation*. The absolute addressing includes global and static local variables, while relative addressing is used for the local variables of functions.
2. In others, the requirement for memory cannot be determined at compile time. For example, when you open a new document in a word processor, this requires memory; however, the compiler cannot be aware of that. Consequently, this requires memory allocation at run time, or *dynamic memory allocation*.

This topic will look at static memory allocation, specifically how memory is allocated on the call stack, and will conclude with an error-handling mechanism that allows you to return to a pointer other than the most recent function call.

Terminology: When you define a function, the *parameters* are the variables that are to be passed into the function. When you make an actual function call, however, you are passing *arguments*. Therefore, in the function

```
double fabs( double x ) {
    return ( x < 0 ) ? -x : x;
}
```

the variable `x` is a parameter of the function; the behavior of the function will change based on its value, and therefore it *parameterizes* the function call. On the other hand, when you now call this function, you pass an argument to the function:

```
printf( "%f\n", fabs( sin( 1005.2343 ) ) );
```

In this case, the return value of the sine function is the argument to the absolute value function.

4.1 The requirements of a function

The operation of a function requires, at a minimum: locations to store:

1. arguments,
2. local variables, and
3. a return value.

These must be passed to the function, and as a function may be called recursively, each function call requires a different location in memory. In addition, as a function may be called from multiple locations, the processor must know where to return to when the function call returns; that is, you must store

4. the program counter as it was immediately prior to the function call.

Now, consider the nature of function calls. Suppose we want to calculate the sine of a complex number z. This requires us to calculate the cosine, sine and exponential of three real numbers, the calls to both sine and cosine will involve a call to a floating-point absolute value function, as is shown in Figure 4-1. These form a tree of function calls, but the only functions we must keep track of those on the path from the initial function call to the currently executing function. When we return, the path is shorted by one, and when another function is called, that path is extended by one.

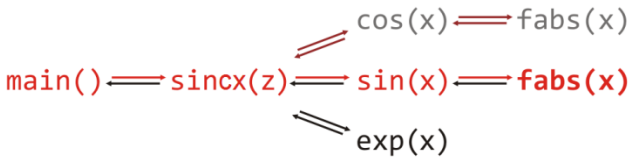


Figure 4-1. Calculating the sine of a complex number.

Thus, this mimics the behaviour of a stack (see Figure 4-2): the memory required in main is at the bottom of the stack, the memory required for the call to the complex sine is next, followed by the memory required for a double-precision floating-point sine, followed by a call to the absolute value function. If the absolute value function wanted to call another function, it could use the next available memory.

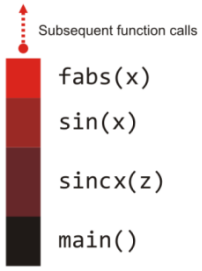


Figure 4-2. The function call stack.

Now, because the memory required for each function call changes, we need to track

5. a *stack pointer* to the current top of the stack.

However, there are two variables involved here: the amount of memory required for arguments changes from function call to function call (as with `printf`) and the memory required for local variables changes, also. Thus, we require a second pointer,

6. a *frame pointer* that separates arguments from local variables.

Now, usually both the stack pointer and frame pointer are stored in registers, however, the value of these registers must be temporarily stored as subsequent calls are made. Thus, with each function call, in addition to storing the old program counter, we will also have to store

7. the old stack pointer, and
8. the old frame pointer.

In addition, the new function will require the use of registers—but when the function call is made, the registers are storing values being used by the previous function. Thus, we must also store

9. the previous values of any registers used.

Later, we will see how the Cortex-M3 manages to avoid requiring both a stack and a frame pointer.

Thus, a function call looks like what is shown in Figure 4-3. In this image, the most recent function call is displayed in vivid color, while the previous function call is grayed.

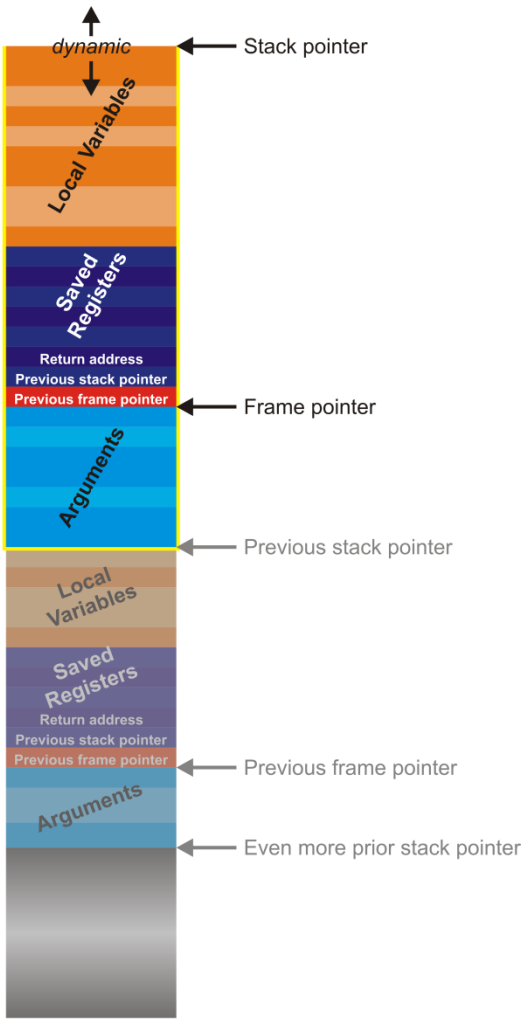


Figure 4-3. A function call.

When the function returns, it must place the return value in an expected location. In this case, the most obvious point is right on top of the previous stack pointer, as is shown in Figure 4-4. Note that you will never see the return value at this location: when the function returns, this will either

1. be assigned to a variable and copied to that location,
2. become the argument of another function call or operation, or
3. be ignored.

The last case happens quite often: `printf` returns the number of characters printed—how often have you ever inspected this value?

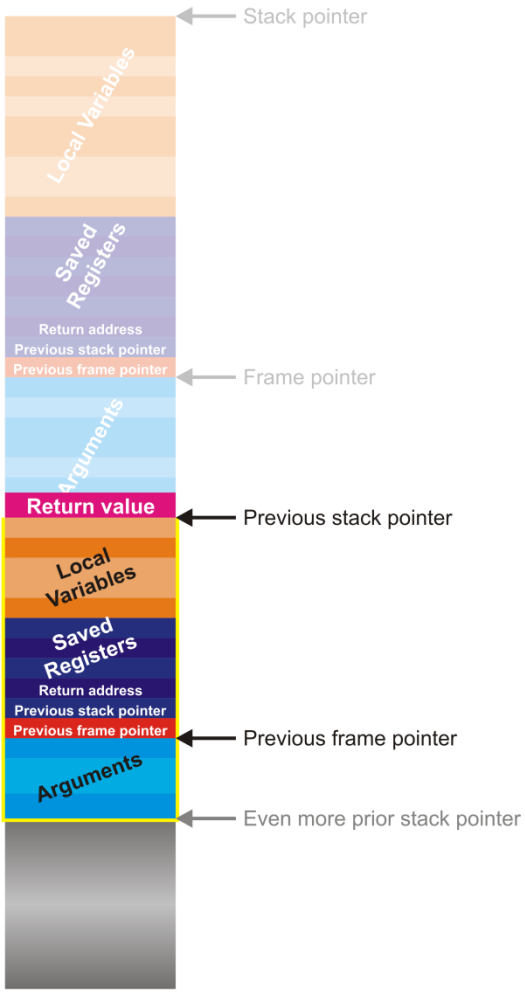


Figure 4-4. A function call with the return value of the function that just returned.

Once the return value is copied to an appropriate location, the function may continue growing or shrinking the memory required for local variables, as is shown in Figure 4-5.

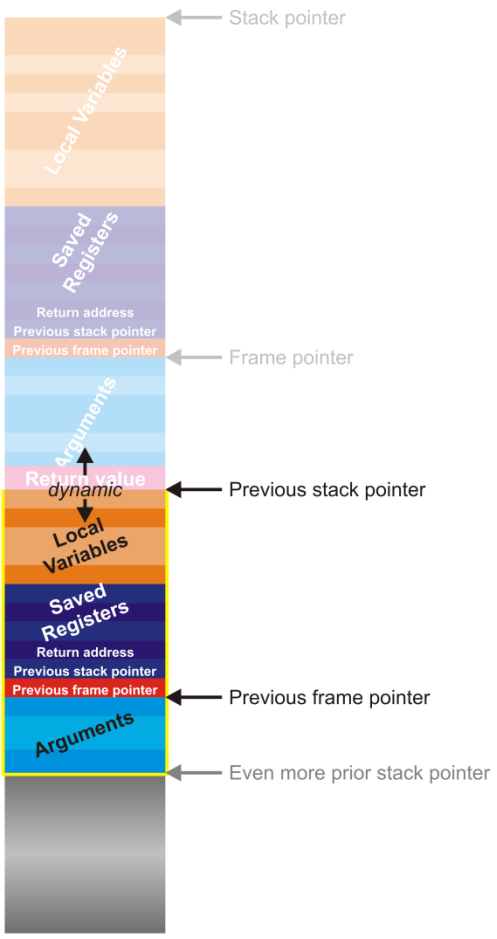


Figure 4-5. Returning to dynamically changing amounts of local variables.

To view that local variables can be dynamic in size, consider the following function:

```
void f( void ) {
    local i;
    printf( "%p\n", &i );
}

void g( void ) {
    int i;

    for ( i = 1; i < 10; ++i ) {
        int array[i*i];
        f();
    }
}

int main( void ) {
    g();
    return 0;
}
```

With each subsequent call, additional memory is allocated for the array, and the previous memory is reused, as the previous array went out of scope.

4.2 The Cortex-M3 design

The Cortex-M3 is designed to work as an embedded system, and therefore numerous assumptions can be made. First, there is not likely going to be a significant number of parameters to functions. Also, it is assumed that functions will very quickly require the use of their parameters. Consequently, arguments are not passed through the call stack, but rather, they are passed through the first four registers. (If more arguments are required, the address of those arguments must be passed as one of the four registers.) Thus, the functions know where the parameters are stored when the function call is made. Similarly, the return value is stored in a register. The compiler will deal with storing the values of the registers on the calling function's call stack. This allows a single stack pointer to be used. Later we will see that there are two stack pointers, but one is to allow devices peripheral to the computer to interrupt the execution of the processor.

4.3 Set jump and long jump

The `setjmp` and `longjmp` features in C provide a mechanism that is more primitive than the throw and catch of C++. The following two examples show how `longjmp` returns to the location

```
#include <stdio.h>



void second( int n ) {
    printf( "    start of second\n" );

    printf( "    end of second\n" );
}

void first( int n ) {
    printf( "  start of first\n" );
    printf( "   calling second\n" );
    second( n );
    printf( "   finished calling second\n" );
    printf( "  end of first\n" );
}

int main( void ) {
    int i = 0;

    printf( "start of main\n" );

    while ( i < 3 ) {
        ++i;
        printf( " calling first\n" );
        first( i );
        printf( " finished calling first\n" );
    }

    printf( "end of main\n" );

    return 0;
}
```

```
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buffer;

void second( int n ) {
    printf( "  start of second\n" );
    longjmp( buffer, n );
    printf( "  end of second\n" );
}

void first( int n ) {
    printf( " start of first\n" );
    printf( "   calling second\n" );
    second( n );
    printf( "  finished calling second\n" );
    printf( " end of first\n" );
}

int main( void ) {
    int i = 0;

    printf( "start of main\n" );

    while ( setjmp( buffer ) < 3 ) {
        ++i;
        printf( " calling first\n" );
        first( i );
        printf( " finished calling first\n" );
    }

    printf( "end of main\n" );

    return 0;
}
```

The change in behaviour between normal function calls and `longjmp` is clear from the output:

```
start of main
 calling first
  start of first
   calling second
    start of second
    end of second
   finished calling second
  end of first
 finished calling first
 calling first
  start of first
   calling second
    start of second
    end of second
   finished calling second
  end of first
 finished calling first
 calling first
  start of first
   calling second
    start of second
    end of second
   finished calling second
  end of first
 finished calling first
end of main
```

```
start of main
 calling first
  start of first
   calling second
    start of second
 calling first
  start of first
   calling second
    start of second
 calling first
  start of first
   calling second
    start of second
end of main
```

Note however that Rule 11 of the JPL coding standard says that "[t]here shall be no calls to the functions `setjmp` or `longjmp`".

Later in this course, we will discuss error-handling mechanisms where this will be applied.

4.4 Summary of static memory allocation

The allocation of global and static local variables is dealt with quite easily by the compiler; however, the compiler can also set up the mechanism to make the allocation of memory required by local variables, and other aspects of function calls. The Cortex-M3 makes certain assumptions about parameters allowing them to be passed in registers. The C programming language `setjmp` and `longjmp` allow you to travel back down the call stack to a prearranged location.

Problem set

T.B.W.

5 Dynamic memory allocation

The last topic looked at static memory allocation. We will now proceed to dynamic memory allocation, the situation when memory is being allocated or deallocated by various tasks at run time. We will consider the interface of an abstract dynamic memory allocator (Dynamic Memory allocator ADT[11]), and then look at a number of implementations of this abstract data type. We will consider the appropriateness of the different implementations for real-time systems.

The relevance of studying dynamic memory allocation to real-time systems is related to the non-functional requirements of safety, performance and scalability. Dynamic memory allocation is often complex and some approaches are simply incapable of delivering memory with guarantees as to the run time, thereby violating safety. Additionally, the approaches must be reasonably fast with the appropriate data structures supporting the process in a way that does not have serious consequences if the number of allocations or deallocations suddenly increases.

Thus, we will look at:

1. the abstraction of a dynamic memory allocation scheme,
2. various allocation strategies,
3. the memory allocation schemes in FreeRTOS, and
4. comments on other features in memory allocation schemes.

We will start with defining an abstract dynamic memory allocator.

5.1 Abstract dynamic memory allocator

An abstract dynamic memory allocator is a container that maintains a pool of memory and that satisfies, where possible, requests for memory and receives allocated memory when returned by its user. The interface for such an ADT has at least two signatures:

```
void *allocate_memory( size_t n );
```
Allocate a block of *n* bytes of memory, returning a pointer to the address of the first byte.

```
void deallocate_memory( void *p_mem_block );
```
Return the block of memory allocated at the address *mem_block* back into the memory pool.

Note that, in general, it is not possible to return a part of a block of memory, and generally the allocator records the size of the block that was allocated. In C++, this interface is provided through the `new` and `delete` operators; however, these are coupled together with the initialize and destruction of the instances of classes by appropriate calls to constructors and destructors, respectively.

Other possible interfaces include:

```
void *allocate_clear_memory( size_t n );
```
Like `allocate_memory`, but sets all bits to zero in the block that is allocated.

```
void *reallocate_memory( void *p_memory, size_t n );
```
Allocate *n* bytes of memory either by expanding the memory allocated at address *mem*, if possible, or allocate new memory while copying over the contents at *mem* into that new memory. In either case, a pointer is returned to the first byte of that block of reallocated memory.

Note that C++ does not offer these in conjunction with their `new` and `delete` interface.

[11] Abstract Data Type, see Section 2.1.1.3.

Recall the difference between static and dynamic memory allocation:

Static	allocated at compile or design time	deterministic	allocation and deallocation are performed during the initialization and termination of processes
Dynamic	allocated at run time	stochastic	allocation and deallocation occurs during the execution of the process

We already saw previously that static memory allocation, when used in conjunction with function calls and returns, may be performed efficiently using a stack. This is not the case with dynamic memory allocation.

With respect to deallocation of memory, dynamic memory allocation may either

1. require manual deallocation by the developer, or
2. the system may perform automatic deallocation.

We will describe each of these here and then address the issue of garbage collection.

5.1.1 Manual allocation management

The first case is exemplified by C and C++: an explicit call to `free(…)` or `delete …` must be made. With such a scheme, the programmer is in complete control of any dynamic memory allocation. The drawback is that it is error-prone for developers, some of whom may not be entirely aware of the consequences of failing to delete memory. For example, a common scenario in which memory is allocated but never deallocated occurs when memory has been allocated by one part of the program and passed to another, but not deleted by the other task.

There are four common sources of error that we must be aware of:

1. pointers that store addresses of memory that have not yet been initialized are referred to as *wild pointers*,
2. pointers that store addresses of memory that has been freed are referred to as *dangling pointers*,
3. the same memory being freed multiple times, and
4. memory that is allocated but not appropriately deallocated when it is no longer needed; that is, a *memory leak*.

5.1.1.1 Wild pointers

After memory is allocated, but before it is first used, the content of that memory is usually random–unknown junk values. Consequently, if the pointer is used as if it is referring to an initialized object, interesting things may or may not occur—especially on different platforms or with different parallel events.

Consider, for example, a singly linked list[12] used as follows:

```
single_list_t *p_list = (single_list_t *) malloc( sizeof( single_list_t ) );
single_list_push_front( p_list, 42 );
```

If the memory allocated all happens to contain zero, this will function perfectly: any variable storing the size will have a value of zero, and the address of the head pointer will also be zero (`NULL`). However, if this code is run on another machine, the memory may not be zeroed, in which case, it may appear that the linked list has a non-zero number of objects. For example, if it is determined in the above case that the linked list is not empty, then a tail pointer would not be updated when the node containing 42 is inserted.

[12] See Section 2.2.6.

The solution is straight-forward: ensure that each call to `malloc(…)` is immediately associated with a call to an initializer.

```
single_list_t *p_list = (single_list_t *) malloc( sizeof( single_list_t ) );
single_list_init( p_list );

single_list_push_front( p_list, 42 );
```

C++ solves this problem by having the new operator immediately call the constructor. Consequently it is not possible to allocate memory without initializing it (assuming of course that the initializer is correctly implemented).

Failing to correctly initialize objects in C++ is a non-trivial problem in an algorithms and data structures course where development is done in Windows but testing is done on Linux. In Windows, most memory is zero anyway, so an incorrectly implemented constructor appears to work. If a student does not test their code in Linux, they will never discover the error until they get their grade.

This can be solved in C using macros:

```
#define SINGLE_LIST( list ) single_list_t list; \
                            single_list_init( &list )

#define SINGLE_LIST_P( p_list )                                                 \
    single_list_t *p_list = (single_list_t *) malloc( sizeof( single_list_t ) ); \
    single_list_init( p_list )
```

Now our code looks like:

```
SINGLE_LIST( list );
single_list_push_front( &list, 42 );
```

or

```
SINGLE_LIST( p_list );
single_list_push_front( &p_list, 42 );
```

5.1.1.2 Dangling pointers

Avoiding dangling pointers can be solved by always assigning a pointer the value `NULL` after the memory has been freed:

```
free( ptr );
ptr = NULL;
```

If you want to assign these pointers to `NULL` only during development (where use of a dangling pointer can be caught during testing), but not in production code (where there is an unnecessary assignment), this can be done as follows:

```
free( ptr );
#ifdef DEVELOPMENT
    ptr = NULL;
#endif
```

A reference to an address that has been deallocated has non-deterministic consequences: the operating system may

1. still flag that memory as allocated, so no issues occur,
2. cause the program to crash, or
3. have reallocated that memory to the same task, but it is now being used for a different purpose.

The last is the most detrimental, as the other data structure can be corrupted.

5.1.1.3 Freeing the same location more than once

One possible consequence of dangling pointers is that they may be freed multiple times. This can have very different results, but usually one of two events will occur:

1. the allocator will cause the program to stop execution,
2. the memory may have since been allocated again, in which case, you would free memory that was not meant to be freed, or
3. *heap corruption*—the heap is in an inconsistent state and operations that mange it will be unpredictable.

Again this is a matter that can be resolved by having as few persistent variable storing addresses and ensuring that when a call to `free(…)` is made, all of those variables must be set to null.

5.1.1.4 Memory leaks

The primary cause of a memory leak is when the last reference to memory is lost by the application. In C and C++, this may happen in one of two ways:

1. The last pointer assigned the memory location is a local variable that then goes out of scope (often when a function returns), or
2. The last pointer (local, member or global) assigned the memory location is overwritten.

In either case, because the last value storing the address is lost, it is now impossible to call either `free(…)` or `delete …` to indicate to the operating system that the memory is no longer required. Consequently, as long as the application is running, the operating system will simply assume that the memory is being used by the application.

Aside: We will see later that when a program exits (or is terminated), any allocated memory is deallocated by the operating system—there is no permanent loss of that memory. However, what happens if the memory leak is in the operating system itself? An interesting article on this is *Finding and Fixing NT Memory Leaks* by Paula Sharick.

http://windowsitpro.com/systems-management/finding-and-fixing-nt-memory-leaks

As the aside mentions, a memory leak in an operating system can be detrimental; however, there are other instances where memory leaks can be more serious than one in an application being run:

1. in an embedded system where memory is more limited as compared to what one would expect from a desktop or laptop system,
2. in an embedded system that is meant to execute for an extended period of time (even years),
3. when memory may be shared by multiple processes and where the termination of one of these processes does not necessarily cause the memory to be collected, and
4. in a device driver.

Numerous programs and tools are available to help find memory leaks. In ECE 250 *Algorithms and Data Structures*, students are given an overloaded `new` and `delete` operators that track memory allocations and deallocations and provide specific details about any memory that has currently not been deallocated.

5.1.1.5 Summary of issues with memory deallocation

Manual memory deallocation has many issues; however, it is also the most efficient if it is done correctly.

5.1.2 Automatic allocation management

Automatic allocation management has two aspects:

1. automatic initialization, and
2. garbage collection.

We will discuss both these here:

5.1.2.1 Automatic initialization

In C, allocation of memory and initialization are two separate operations, often in the form:

```
Type *p_entry = (Type *) malloc( sizeof( Type ) );
type_init( p_entry, … );
```

Accidently accessing a pointer to an object that has not been initialized is a form of memory corruption. In C++, the inclusion of a constructor prevents this: as soon as the memory is allocated, the constructor is called on the object prior to `new` returning a pointer to the calling function:

```
Type *p_entry = new Type( … );
```

In languages such as Java and C#, variables defined to be primitive data types are automatically initialized to zero.

5.1.2.2 Garbage collection

This second case is exemplified by the garbage collectors in programming languages such as Java and C#. Each time a reference to an object is assigned, additional work is done by the runtime environment to track references to allocated memory. We will describe

1. two algorithms for dealing with garbage collection, and
2. some of the issues with garbage collections.

5.1.2.2.1 Garbage collection algorithms

There are two mechanisms for dealing with garbage collection:

1. reference counting, and
2. tracing algorithms.

We will look specifically at the Boehm–Demers–Weiser garbage collector for C.

5.1.2.2.1.1 Reference counting

The simplest form of garbage collection is reference counting: track how many references store the address of a particular object and whenever one of those references is assigned a new value, decrement the count for the previous value and increment the count for the assigned value. Whenever the count for an object is decremented to zero, delete the object. Essentially, each time an assignment is done to either a pointer or reference; this must be replaced by a call to change the allocation tree. This, of course, makes every assignment more expensive than one would expect. An example of reference counting is shown in Figure 5-1.

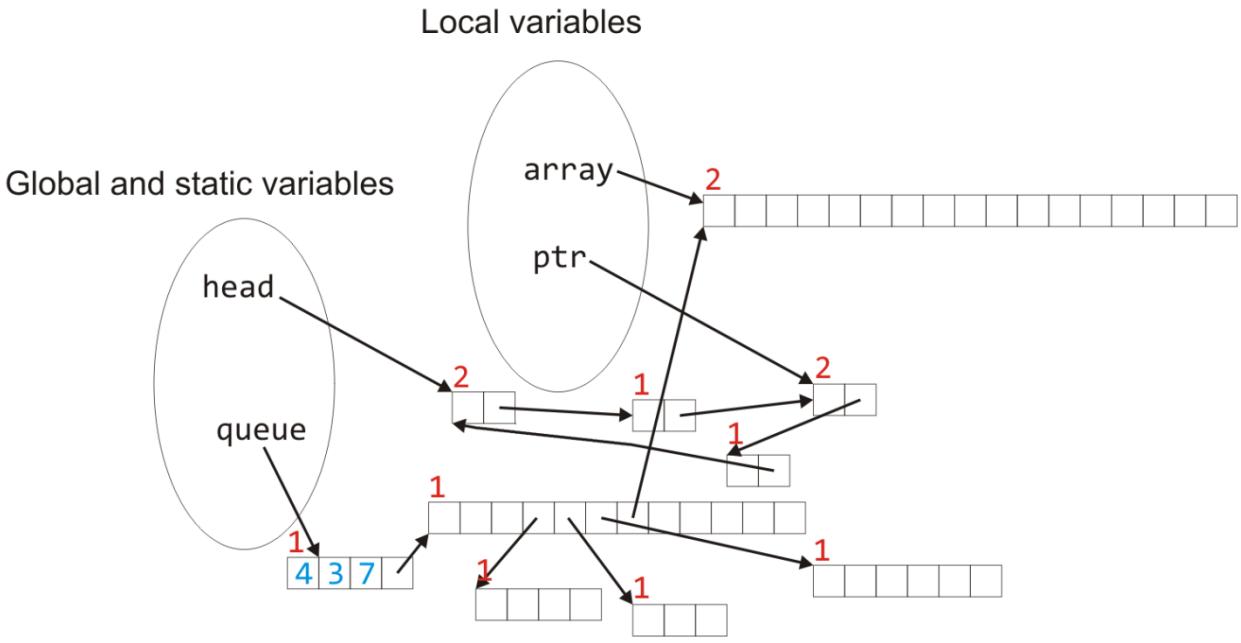


Figure 5-1. Reference counting for a collection of assigned blocks of memory.

In this case, if the global variable `queue` is set to `NULL`, the reference count for the data structure is decremented to zero, so its memory is deallocated, but not before the reference count of the array storing the queue has its reference count decremented. When the array is marked for deallocation, each of the arrays pointed to in the array also have their reference count decremented, and thus 3 of those 4 arrays can also be deallocated (one of them still has a reference with the local variable `array`).

It is possible to implement reference counting in C++ by creating a class that behaves like pointers but where operators such as

1. the unary dereference operator *,
2. the assignment operator =,
3. the auto increment and decrement operators ++ and --,

are overloaded.

Issues with reference counting include:

1. cycles cannot be detected (what if `head` and `ptr` are set to `NULL`),
2. it requires $\Theta(n)$ additional memory where n is the number of pointers or references, and
3. it is not real-time, as the reference count of multiple objects may have be decremented even if none of them are eligible for garbage collection.

Tracing algorithms solve the first problem.

5.1.2.2.1.2 Tracing algorithms

One problem with reference counting is that a data structure such as the cyclic link list pointed to by `head` in Figure 5-1 still has internal references even if, for example, `head` and `ptr` are reassigned. Such a structure would not be garbage collected. As an example of another garbage collection algorithm, we will consider *mark-and-sweep algorithm* where garbage collection is only run when a request for memory is made and there is no available memory; thus, unreferenced memory may remain marked as allocated. These algorithms track all global and local variables that store references and each allocated block of memory is associated with a bit. When the algorithm is run,

1. all bits are set to zero,
2. each memory block referred to by a global or local variable is *marked* (the bit is set to 1),
3. the first time each block is marked, this algorithm is run recursively and any memory blocks referred to within this memory block are themselves marked.

Thus, we perform a depth-first traversal of a directed graph, and all blocks that are connected to the set of global and local variables referencing objects are therefore marked. We continue then to sweep through all allocated blocks of memory and free all those that are not marked. Other garbage collection algorithms (such as the *mark-compact algorithm*) are based on this mark-and-sweep algorithm.

Note that this approach makes it unsuitable for real-time systems, as the behaviour is unpredictable. If garbage collection occurs at the wrong moment in time, this could cause the system to miss a deadline. For example, suppose a task requests memory when it has another 10 ms of computation time to complete an operation that must meet a deadline in 12 ms. Even if the garbage collection cycle is only 5 ms, the deadline will be passed. In soft and even firm real-time systems, such sporadic delays may be acceptable, but they would be unacceptable in a hard real-time system.

5.1.2.2.1.3 Garbage collection in C

The Boehm–Demers–Weiser non-real-time garbage collector (see http://www.hboehm.info/gc/) can be implemented in most C programs by

1. installing and including the library with `#include "gc.h"`;
2. initializing the garbage collection with a call to `GC_INIT()`;
3. replacing all calls to `malloc(…)` with calls to
    a. `GC_MALLOC(…)` if the object itself may contain pointers, or
    b. `GC_MALLOC_ATOMIC(…)` if the object does not contain subsequent pointers;
4. replacing all calls to `realloc(…)` with calls to `GC_REALLOC(…)`; and
5. remove all calls to `free(…)`.

You can access the size of the heap with `GC_get_heap_size()`.

5.1.2.2.1.4 4 Summary of garbage collection algorithms

Garbage collection schemes generally fall into one of the two described categories: reference counting and tracing algorithms, the second being far more prevalent and most algorithms today are based on the mark-and-sweep algorithm.

5.1.2.2.2 Issues with garbage collection

One issue with garbage collection is that references to allocated memory may remain in data structures even if they are not accessible. For example, a stack that is used to perform a depth-first traversal of a tree will store addresses of nodes within the tree; however, if the stack remains in scope and any global or local variable referring to the tree is reassigned or goes out of scope, then there will still be entries in the stack that refer to nodes within the tree until either

1. the stack is no longer referenced to, or
2. the stack is reused to perform a depth-first traversal on a different tree.

The easiest solution is that any data structure that is used to implement an algorithm on a data structure should have a shorter life span than the data structure itself; however, if this is not possible, references in such intermediate data structures should be assigned null when they are no longer logically part of the data structure. For example, consider the following example of a class in Java:

```
public class Stack {
    private int capacity;
    private Object[] array;
    private int size;

    public Stack( int s ) {
        capacity = s;
        array = new Object[capacity];
        size = 0;
    }

    public int size() {
        return size;
    }

    public void push( Object obj ) {
        array[size] = obj;
        ++size;
    }

    public void pop() {
        --size;
    }

    public Object top() {
        return array[size - 1];
    }
}
```

Suppose we perform a depth-first traversal of a tree using a stack:

```
// Allocate memory for a stack to be used for traversals
Stack s = new Stack( 100 );

// Executing...
General_tree tree = new General_tree();
// Add children here

// Perform traversal
s.push( tree.root() );

while ( !s.empty() ) {
    General_tree t = s.pop();
    // Push any children of 't' onto the stack
}

root = null;
// We're finished, right?
```

At this point, we should be fine, right? `root` is set to `null` and therefore it and all its descendants can be garbage collected. Unfortunately, no, because the entries of the array in the stack may still be assigned even if they mean nothing to the stack itself (the next time it is used, those entries will be overwritten). Instead, we must remove all references to objects temporarily stored in containers when those objects are removed from the containers:

```
public void pop() {
    --size;
    array[size] = null;
}
```

For other information about garbage collection, read

Java theory and practice: Garbage collection and performance: *Hints, tips, and myths about writing garbage collection-friendly classes* by Brian Goetz, available at

http://www.ibm.com/developerworks/library/j-jtp01274/

5.1.2.2.3 Summary of garbage collection

In this topic, we have discussed garbage collection algorithms and some issues that may affect the efficacy of a garbage collection algorithm.

5.1.2.3 Summary of automatic allocation

Automatic allocation and deallocation are two separate issues: C implements neither, C++ implements automatic initialization, and with the Boehm–Demers–Weiser garbage collector, it is possible to implement garbage collection in C, but initialization must still be performed.

5.1.3 Summary of abstract dynamic memory allocation

An abstract dynamic memory allocator will, at the very least, have an interface that allows tasks to request memory from the pool and to return memory to the pool. Most embedded systems will have manual deallocation, but it is possible to have a reference counting scheme whereby each allocated object is associated with a count, thereby allowing blocks of memory to be deleted. This has its own weaknesses and the execution of the garbage collector to find these blocks of available memory is expensive with respect to run time.

5.2 Allocation strategies

The allocation of memory by the operating system can be either fixed partition or variable partition. Like static and dynamic allocation, fixed partitioning is simpler to implement, but it has numerous restrictions, the most significant of which is internal fragmentation. We will look at using:

1. fixed block sizes,
2. variable block sizes,
3. a composition of these two schemes, and
4. other advanced memory allocation schemes.

5.2.1 Fixed block sizes

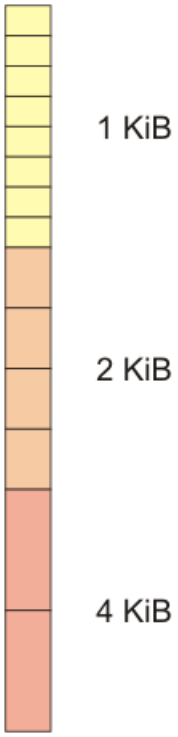
One possibility is to have a fixed block size. In this case, all blocks of memory allocated are the same size; if the request is less than one block, a full block is allocated anyway. If a request is for, say, 3.7 blocks, the memory returned will be 4 blocks.

5.2.1.1 One size of blocks

In an embedded system, it may only be necessary to provide memory for a data structure such as a linked list. In this case, a memory allocation strategy is very straight-forward: create a linked list (you can think of it as a stack—we will only be pushing and popping from the front) and cast each block as if it was a pointer and store the address of the next block of available memory. The last block of memory would have store the address `NULL`, and when a node is deallocated, it would be prepended to the front of the linked list. An implementation of this is available at https://ece.uwaterloo.ca/~dwharder/icsrts/Keil_board/dynamic/

5.2.1.2 Fixed size blocks

With a fixed-sized-block strategy, memory is initially divided into partitions, each of which may be assigned.



Generally, these can be allocated and deallocated in Θ(1) time. We will discuss a strategy we will use again in this class. One may, for example, keep either an array or linked list of the addresses of the unassigned partitions with one data structure per block size.

```
// Global variables for the operating system
int partition_count[3] = {8, 4, 2};
size_t partition_size[3] = {1024, 2048, 4096};
single_list_t addresses[3];

// Initialization
void memory_init() {
    char *p_base_address = 0x039a8000;
    char *p_working_address = p_base_address;

    for ( int i = 0; i < 3; ++i ) {
        for ( int j = 0; j < partition_count[i]; ++j ) {
            addresses[i].push_front( p_working_address );
            p_working_address += partition_size[i];
        }
    }
}
```

Now, whenever memory is required, we just need pop the next address off of the appropriate linked list and return it:

```
void *malloc( size_t n ) {
    void *p_allocated_memory = NULL;

    for ( int i = 0; i < 3; ++i ) {
        if ( n <= partition_size[i] &&  addresses[i].size() > 0 ) {
```

```
            p_allocated_memory = addresses[i].pop_front();
            break;
        }
    }

    return p_allocated_memory;
}
```

Note that this automatically allocates the smallest possible partition that satisfies the request. Freeing memory is similarly meticulous:

```
void free( void *p_address ) {
    if ( p_address != NULL ) {
        char *p_working_address = p_base_address;

        for ( int i = 0; i < 3; ++i ) {
            p_working_address += partition_size[i] * partition_count[i];

            if ( p_address < p_working_address ) {
                addresses[i].push_front( p_address );
            }
        }
    }
}
```

In this case, however, the use of the linked list unnecessarily wastes memory; we don't need this because either a partition is

1. assigned, in which case, it will be used to store whatever the requesting process requires of it, or
2. not assigned, in which case, this is memory we can use for another purpose; for example, a linked list.

Essentially, each partition could store in its first location the address of the next.

```
// Global variables for the system
int partition_count[3] = {8, 4, 2};
size_t partition_size[3] = {1024, 2048, 4096};
char *ap_addresses[3];

// Initialization
void memory_init() {
    char *p_base_address = 0x039a8000;
    char *p_working_address = p_base_address;
    int i, j;
    void *p_address;

    for ( i = 0; i < 3; ++i ) {
        ap_addresses[i] = p_working_address;

        for ( j = 0; j < partition_count[i]; ++j ) {
            p_address = p_working_address;
            p_working_address += partition_size[i];

            if ( j == partition_count - 1] ) {
                p_address = NULL;
            } else {
                p_address = p_working_address;
            }
        }
    }
}
```

```
void *malloc( size_t n ) {
    void *p_return_value = NULL;
    int i;

    for ( i = 0; i < 3; ++i ) {
        if ( n <= partition_size[i] &&  list_size( &( ap_addresses[i] ) ) > 0 ) {
            p_return_value = list_pop_front( &( ap_addresses[i] ) );
            break;
        }
    }

    return p_return_value;
}

void free( void *p_address ) {
    if ( p_address == NULL ) {
        return;
    }

    char *p_working_address = p_base_address;

    for ( int i = 0; i < 3; ++i ) {
        p_working_address += partition_size[i] * partition_count[i];

        if ( p_address < p_working_address ) {
            list_push_front( &( ap_addresses[i] ), p_address );
            break;
        }
    }
}
```

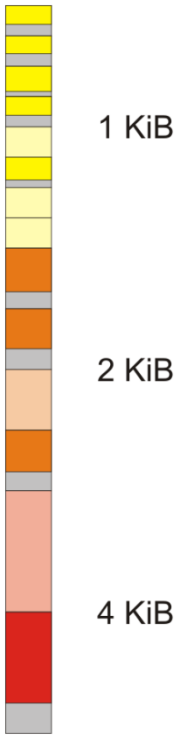
Issue: if another process writes outside of its partition, this could corrupt the unused partitions.

Note that Rule 24 of the JPL coding standard says that "There should be no more than one statement or variable declaration per line." In the above example, both looping variables are declared together, as there is nothing gained by defining them separately. The two previous lines, however, declare and initialize two variables of type `char *` separately.

In an embedded system, if it is known at least approximately how much memory is required and in what amounts, it may be reasonable to apply such a simple scheme. Many microprocessors come with a fixed amount of main memory, so there may be no requirement to economize on main memory use if other factors already require the given chip to be used. In addition, today, main memory is significantly cheaper than it was even a decade ago.

5.2.1.3 Internal fragmentation

One significant issue with fixed partitions of memory is that may factor against its use is that not all memory requests may require the full block of memory; however, this does not prevent the entire partition from being assigned. These bits of unused memory are termed *internal fragmentation*. In the next figure, allocated blocks are in solid colors with representative internal fragmentation shown in gray.



Just do demonstrate, collecting the allocated-and-used, unallocated, and internal fragments in this example, we find that 20 % of the memory is "lost" to internal fragmentation; it is not storing anything useful, yet cannot be assigned to a new memory request either.

Note that a fixed partition strategy can be used to augment a dynamic and variable partition strategy. For example, if it is known that a significant but variable number of blocks of a specific size `BLOCK_SIZE` may be required for a project, it may be easier to allocation `n * BLOCK_SIZE` bytes initially and use a scheme similar to the one above for fast allocation and deallocation of these blocks. This would require additional functions:

```
void *fix_malloc( size_t )
void  fix_free( void * )
```

This may be useful in, for example, a video gaming application where speed is necessary but where the number of partitions may vary quickly.

5.2.1.4 Summary for fixed block sizes

A memory allocator that uses only fixed block sizes can be very fast: all operations are $\Theta(1)$. Unfortunately, this may result in internal fragmentation, where a block significantly larger than the requested memory is allocated for a given request. The extra memory is said to be an internal fragment and it cannot be used until the entire block is released. Variable sized allocators can deal with this situation, but it also results in additional overhead in terms of run time and the possibility of *external fragmentation* (blocks too small to allocate).

5.2.2 Variable-sized-block strategies

With a fixed partition strategy, memory is initially divided into partitions, each of which may be assigned. We will look at a number of approaches.

5.2.2.1 Bitmaps

It is possible to divide memory into M n-bit units, and then to create a bit-array of size M storing the status of each unit (0 for unallocated, 1 for allocated). In this case, $100/(n + 1)$ % of the memory is used for the bitmap, so if the unit size is 4 bytes, the bitmap occupies approximately 3 % of memory; while if the unit is 16 bytes, the bitmap occupies approximately 0.8 % of memory.

Issues include internal fragmentation, although now the average wasted memory per allocation will be only $n/2$ bits (in addition to the memory of the bitmap itself), and finding a block of m bytes requires one to find a sequence of $\frac{8m}{n}$ zeros.

A sample bitmap is shown here where gray indicates unallocated (0) and red indicates allocated (1).

While bitmaps may not be appropriate for dealing with real-time memory allocation (finding large contiguous blocks of memory can be very slow—especially if memory is fragmented), they are a candidate for secondary memory—especially when there is no requirement for the blocks to be contiguous.

5.2.2.2 Linked lists

An alternate approach is to consider some form of linked list. This linked list could be stored in one of two ways: as a separate linked list, or as suggested above, by embedding the linked list into the memory that is either free or allocated. We will take the second approach.

Thus, as an initialization, the linked list would contain a single entry. Assuming there are 4 KiB of memory available starting at address `0x00003000`, it would be a single list with one block. This would be prefixed with a header with eight bytes, where:

1. four bytes stores the size (4088 = 4096 – 8), and
2. the next four initially points to `NULL`.

How do we track an allocation of 512 bytes? We split the block of size 4096 into two blocks: one of size 520 (512 + 8) and one of size 3576 (of which 3568 = 3576 – 8 is available). The address of the 9th byte in the allocated block is retuned. If our memory started at address `0x3000`, this would be `0x00003008`. In addition, we may use one bit to flag whether the block is allocated or deallocated.

Question: Is it better to store both the size and the memory location of the next block? After all, can we not just add the size of the current block to determine the location of the next?

1. Allocating extra memory requires more space, but
2. Having to calculate an offset with each step of the linked list is an unnecessary operation, and this may be detrimental.

Suppose we have 1 MiB of memory, and we expect, on average, 1000 allocations. The additional memory for this header would be just under 1 % of the available memory.

Question: Do we require a singly linked list or a doubly linked list?

Suppose we want to possibly merge a deallocated block back together with adjoining entries. In this case, it would be necessary to look both forward and back in the linked list to determine whether or not it can be merged with the block immediately before and/or the block immediately after.

5.2.2.3 External fragmentation

One significant issue with any variable sized memory allocator is that blocks must be broken up into smaller sub-blocks in order to accommodate requests. This may lead to a situation where the allocated memory is scattered between blocks of available memory, and as allocated blocks are freed, they are now returned to the pool in a "checkerboard" pattern, as shown in Figure 5-2.

Figure 5-2. Allocated memory (interspersed between available memory.

Consequently, while *N* bytes may be available, it may not be possible to satisfy a request for *N* bytes because there is no single contiguous block of size *N* or greater. This can lead to situations where even moderately-sized requests cannot be satisfied. Such a situation is referred to as *external fragmentation*. There are two possible solutions that can alleviate such a situation:

1. coalescence, and
2. allowing small amounts of internal fragmentation.

We will discuss both of these here.

5.2.2.3.1 Coalescence

When memory is deallocated, the allocator can determine whether or not there are nearby blocks that are also available, in which case, the two available blocks are coalesced into a single larger available block. In this case, the memory in Figure 5-2 would have larger blocks available, as shown in Figure 5-3.

Figure 5-3. The memory allocated in Figure 5-2 with adjacent available blocks coalesced.

This, however, may require significantly more overhead either at allocation time or at deallocation time as the available blocks must be sorted in some manner, and maintaining an ordered list will always require some overhead.

5.2.2.3.2 Allowing internal fragmentation

If a request is made for a block of size *n* bytes, but the block that is being allocated is slightly larger, say *n* + 10 bytes, it might be better to tag the entire block as being allocated where the 10 bytes left over constitutes a form of *internal fragmentation*. This might be more efficient than splitting the block into two blocks, one of size *n* and one of size 10, and then trying to coalesce them together when the one is deallocated. It is unlikely that there would be a request for memory of size 10 or less so the second block would probably never be allocated anyway. This is even more useful in an embedded system if it is known that there will never be requests for memory of size less than *m* bytes, in which case, it is pointless make such a split.

5.2.2.3.3 Summary of external fragmentation

External fragmentation is an issue that needs to be dealt with in variable-sized allocators. Two general means of alleviating this issue are to coalesce two available blocks back together as a single available block, and to not allow blocks to be split below a minimum size (resulting, however, in internal fragmentation). We will consider these when we consider the various allocation schemes.

5.2.2.4 Basic variable-sized allocation schemes

As soon as the first deallocation occurs, unless that deallocation is immediately adjacent to the final block of available memory, we now have some external fragmentation of available memory. Thus, with any subsequent allocation, there is a question of which block of available memory should be sub-divided in order to accommodate the request.

There are four algorithms we will look at:

1. first fit,
2. next fit,
3. best fit, and
4. worst fit.

5.2.2.4.1 *First fit*

First fit starts at the beginning of memory and finds the first block large enough to accommodate the request. Once a block is found, it is divided into two blocks, one allocated and the other still unallocated. It is a relatively fast algorithm. The run time is O(*n*) where *n* is the number of unallocated blocks.

Problem: We are iterating through all blocks of both allocated and unallocated memory. If we are searching only for the next available block of unallocated memory, why do we waste time stepping through allocated blocks?

Solution: We could have two doubly linked lists: one for all blocks, and another for unallocated blocks only. Thus, we would have `previous_unallocated_block`, `previous_block`, `next_block`, and `next_unallocated_block`. These two fields are, however, not necessary when a block is allocated, so it does not affect the header size of an allocated block; however, it will require that unallocated blocks be at least a minimum size.

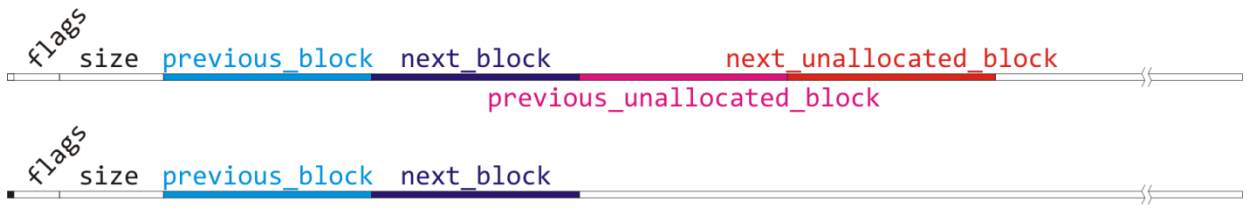


Figure 5-4. When a block is unallocated, the memory is used to store pointers to the previous and next unallocated blocks,
while when the memory is allocated, all memory beyond the immediate previous and next blocks is available to the user.

5.2.2.4.2 *Next fit*

One issue with first fit is that it will quickly shrink the initial blocks of available memory into chunks that may be too small to allocate and this may result in a number of small unallocated blocks at the start of memory. Consequently, an alternative is *next fit*. Rather than always starting from the start of memory, searching for an available block, the allocator tracks the last block that was sub-divided and with the next request for memory, begins by checking that block. Like first fit, the run time is still O(*n*) where *n* is the number of unallocated blocks.

Note: We will describe first fit and next fit as *sequential fits*, as any implementation requires the collection of unallocated blocks to be iterated through sequentially.

5.2.2.4.3 *Best fit*

As an alternative, consider finding the smallest possible block that can satisfy the request? This would require searching through all available blocks—an operation which could be potentially Θ(*n*)!

Problem: How could we reduce this to Θ(ln(*n*))? We could keep the blocks in order of size, but this would still require walking through the list—an O(*n*) solution. Recall that the blocks themselves represent nodes, and what node-based data structure was used for storing linearly ordered data?

Solution: How about an AVL tree or a red-black tree? Once again, we could use the unallocated memory portion to store information relevant to either tree structure.

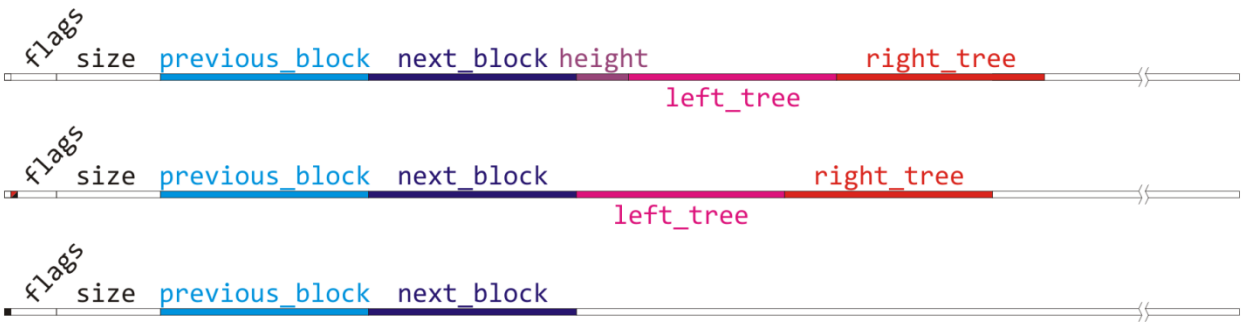


Figure 5-5. An AVL node (storing the height) and a red-black tree (storing a single bit with the color), together with a block when either is allocated where all memory beyond the two block pointers is available to the user.

Now, the best fit runs in $\Theta(\ln(n))$ time—possibly even better than first fit or next fit which will run in $O(n)$ time.

One problem with best fit is that it tends to leave significant fragments of unusably small memory around—an extreme case of external fragmentation. If we wanted to leave the largest possible hole, we could use the opposite strategy.

5.2.2.4.4 Worst fit

Suppose instead we now instead always allocate any new memory in the largest possible block of unallocated memory. The unallocated component of the block will be large; hopefully large enough that it can be used for another memory allocation later (as opposed to being too small to be useful).

Problem: This requires us to keep a sorted list from largest to smallest, but we are only interested in ever accessing the largest of these. What data structure can we use to keep track of these?

Solution: A max heap could be used here. If we want to maintain the smallest possible blocks size for unallocated blocks, we could use a leftist heap (a binary heap structure); however, if more memory is available, we could use either a binomial or even Fibonacci heap structure.

Note: We will describe best fit and worst fit as *branching fits*, as any implementation requires the collection of unallocated blocks to be stored in a tree-based data structure.

5.2.2.4.5 Summary of linked list memory management

We have seen four techniques for allocating memory through linked lists. Such allocation strategies are not, however, always appropriate for real-time systems. In each case, it may be necessary to iterate through many available blocks before one is found. Thus, it would be very difficult to ensure a (small) upper bound for the time it takes to allocate memory using such strategies.

5.2.2.5 Summary of variable-sized allocation strategies

Variable-sized allocation strategies are more flexible than fixed-sized strategies. They are more likely to be implemented as a linked list, though it is possible to use some form of bitmap. A consequence of variable-sized allocations is that external fragmentation may occur, a problem that can be partially fixed by coalescence or allowing some internal fragmentation. Four strategies for where to allocate a memory request are first-fit, next-fit, best-fit and worst-fit. The next topics look at more advanced schemes.

5.2.3 Advanced memory allocation algorithms

We will look at several other advanced memory allocation algorithms, including:

1. quick-fit,
2. binary buddy,
3. Doug Lea's malloc,

4. half-fit,
5. two-level segregate fit, and
6. smart memory allocator.

5.2.3.1 Quick fit

It might be reasonable to keep additional lists for common requests: for example, if a particular data structure is known to require blocks of size 1024, then any available block that is of size 1024 up to perhaps 1152 could be additionally stored in a separate list. When a request of size 1024 occurs, it might be simpler to allocate the entire block out of that list, regardless of size, and accept the balance as internal fragmentation.

5.2.3.2 Binary buddy

Binary buddy is a scheme that, at its simplest, divides memory into 2^h blocks of K bytes each. Memory can be allocated in blocks of size K, $2K$, $4K$, and up to 2^hK. Blocks can only be allocated, however, along integral multiples of 2^k. For example, if $h = 3$, blocks can be allocated only in the following according to the following shown in Figure 5-6.

Figure 5-6. Possible groupings of blocks allocatable using binary buddy when $h = 3$.

The name comes from the restriction that when a block of memory is split into two, this creates two *binary buddies*, and when it comes time to coalesce deallocated blocks together, a block can only be coalesced with its "buddy". This allows for a very easy scheme of not only dividing memory, but more importantly being able to coalesce deallocated memory into larger available blocks, and this can be achieved using a perfect binary tree and doubly linked lists. To initialize the system:

1. Create an array of $h + 1$ doubly linked lists, but we only require a head-pointer—no tail pointer or counter is necessary. The k^{th} linked list is associated with blocks of size $2^k N$ for $k = 0, 1, \ldots, h$. The memory blocks themselves can be used to store the next and previous pointers required for a doubly linked list.
2. Place the entire memory block into the h^{th} linked list, with the remaining linked lists being empty.
3. Create a perfect binary bit-tree (*allocation tree*) of height h stored as an array where the bits indicate whether a particular block or a portion thereof has been allocated. Set all the bits to 0 indicating that there are no allocated or partially-allocated blocks.

This allocation tree and the association between the bits and the blocks they represent shown in Figure 5-7.

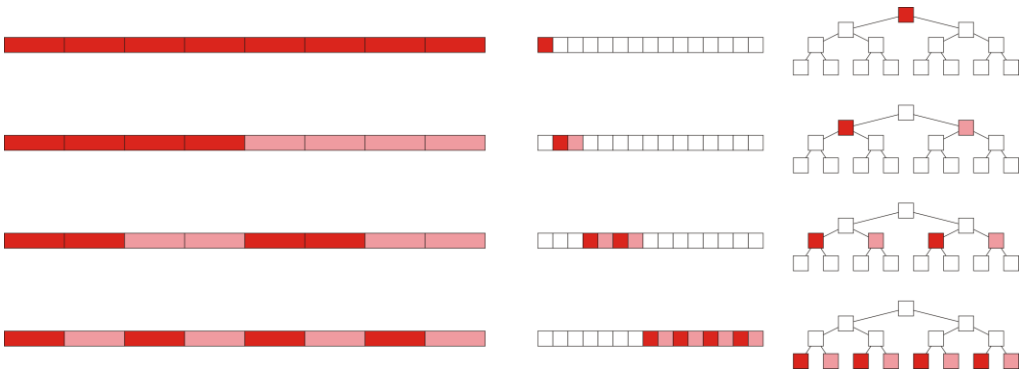


Figure 5-7. Each of the blocks and their corresponding bits in the perfect binary bit-tree.

On a 32-bit system, this would require $4(h + 1) + 2^{h-2}$ bytes, so if $h = 10$, this would require 300 bytes allowing up to $2^{10} = 1024$ separate blocks to be allocated. In the limit, this will require approximately $\frac{100}{4K}\%$ additional memory, and if $K = 32$ bytes, the overhead would be less than 1% so long as $h \geq 10$.

After initialization, our memory-allocation scheme is initialized, we have the set-up shown in Figure 5-8.

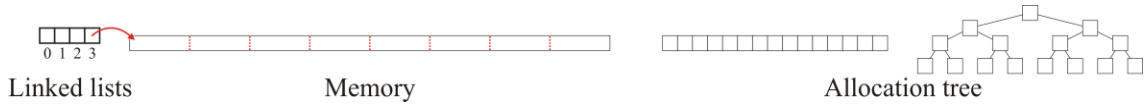


Figure 5-8. Binary-buddy memory allocation scheme following the initialization when $h = 3$.

Now, when a request for n bytes of memory comes, let k be the smallest power of 2 such that $n \leq 2^k K$.

1. If the k^{th} linked list is empty,
    a. if all linked lists from $k + 1$ to n are empty, deny the request—the requested memory cannot be allocated,
    b. otherwise, let $k^* > k$ be the next non-empty linked list and iterate j from k^* down to $k + 1$:
        i. pop an unallocated block from the j^{th} linked list, flag it as allocated in the allocation tree, and push both halves onto the $(j - 1)^{th}$ linked list in reverse order;
2. pop a block off of the k^{th} list, mark the corresponding bit in the binary bit-tree to 1.

Thus, the run time of an allocation is $O(h)$. For example, if a request for a block of size $n \leq K$ is made, the state of our memory-allocation scheme would be what is shown in **Error! Reference source not found.**.

Figure 5-9. The state of our memory allocation scheme after one block of size K was allocated.

Now, if a request is made for a block of memory $n \leq K$, $K < n \leq 2K$ or $K < n \leq 2K$, such a block could be immediately allocated by popping a block off of the corresponding list and flagging it as allocated. You will note that the scheme ensures only the smallest possible block is ever broken into two, ensuring that larger unallocated blocks are not unnecessarily split in two.

If this is followed by a request for a block requiring size $2K$, followed by two additional requests for blocks of size K, we would see the state change as indicated in Figure 5-10.

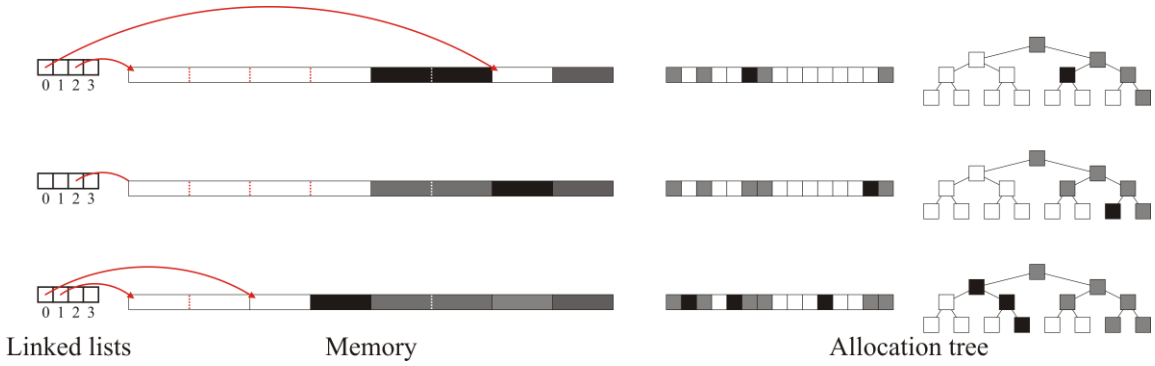


Figure 5-10. The state after allocating blocks of size $2K$, K and K, in that order.

When a block of memory is freed, it is flagged as being unallocated. To determine the size of the freed block, it is necessary to start at the corresponding leaf node for that block and walk up the allocation tree until a bit is found indicating that that block is allocated. Next comes the most useful aspect of the binary-buddy allocation scheme: we can now check the sibling of the freed block, and if it too is unallocated, we can coalesces the two into a single unallocated block. Removing an unallocated sibling from its linked list in $\Theta(1)$ time is why a doubly linked list is required. The coalesced block may also be subsequently merged with any unallocated sibling. Thus, we have the following algorithm:

1. Assume that the size of the deallocated block is K, and starting at that leaf node in the allocation tree, move up the tree until a node is found that is flagging the corresponding block as being allocated. Let the depth of that node be k; and
2. iterate j from $h - k$ to 1, where
    a. if the sibling of the node is also flagged as allocated, we are finished,
    b. otherwise
        i. pop the sibling from the j^{th} linked list,
        ii. flag the parent as unallocated, and
        iii. push the parent block onto the $(j - 1)^{\text{th}}$ linked list.

In our example, if the deallocation of blocks followed the same order in which they were allocated, the first block two deallocations would only see those blocks placed into their respective linked lists, but the third block deallocated would see it coalesce with its sibling, and that block itself would again be coalesced with its sibling. Finally, the deallocation of the last block allocated would see coalescence occur until the entire block of memory is indicated as being free. These are shown in Figure 5-11. Recall that freed blocks would be placed back at the front of their respective linked lists, and in the last two cases, the freed block is coalesced with the surrounding blocks.

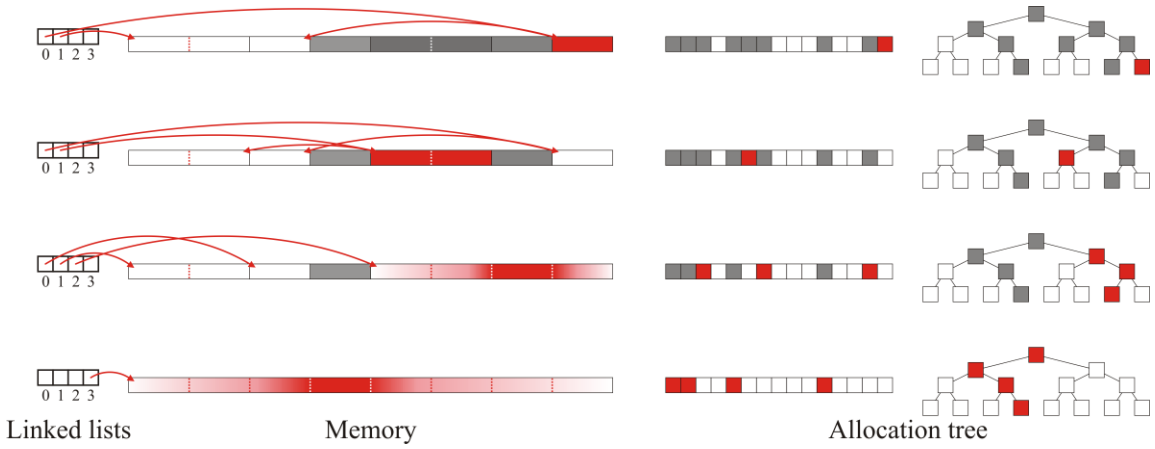


Figure 5-11. The blocks deallocated in the same order they were allocated in Figure 5-9 and Figure 5-10.

The most difficult aspect of this implementation is that of the allocation tree. You may recall that from the implementation of a binary heap stored as an array, the root would be located at array entry 1 and the children at array entry j would be located at entries $2j$ and $2j + 1$, while the parent would be located at entry $j \div 2$ using integer division, which rounds down. In order to translate this into bits, we must do a little more work, as follows: if `allocation_tree` is an array of 2^{h-2} bytes (`unsigned char`) the bit corresponding to entry `j` in an array is in byte `j >> 3` and in bit `1 << (j & 7)`, we could then perform the following operations:

Operation	Source code
check the bit	`allocation_tree[j >> 3] & (1 << (j & 7))`
set the bit	`allocation_tree[j >> 3] |= (1 << (j & 7))`
clear the bit	`allocation_tree[j >> 3] &= ~((unsigned char) (1 << (j & 7)))`

Similarly, if `p_block_address` contained the address of a block being freed, the corresponding leaf block could be found by calculating

```
((size_t) (p_block_address – p_memory_block))/block_size,
```

where `p_memory_block` is the address of the first block in memory, and `block_size` what we have been referring to as *K*. Issues with this scheme include

1. internal fragmentation, and
2. the run time of both allocation and deallocation is O(*h*).

It is, never-the-less, a reasonable algorithm. Improvements can be made if, for example, it is known that it is only necessary to allocate blocks of size *K* or 2*K*, one could create a *forest*, essentially bypassing the unnecessary task of splitting the entire block of memory. In our scenario, if we restricted allocation to blocks of size *K* or 2*K*, our initialized system would already appear as shown in Figure 5-12.

Figure 5-12. Binary buddy with restricted allocation size.

Again, internal fragmentation is an issue, but the run time is now Θ(1). Such characteristics make binary buddy a reasonable choice for a real-time system.

5.2.3.3 Doug Lea's malloc

For allocations of 256 bytes or larger, this allocator uses best fit. If a tie exists, the least-recently used block is chosen. This is achieved by having a number of bins linking available blocks of either exact or approximate increasing size (16, 24, 32, …, 512, 576, 640, …, 2^{31}). Adjacent freed blocks are coalesced into larger blocks.

For allocations under 256 bytes (*small allocations*), if a perfect fit is not found, it uses a next fit algorithm beginning at the location of the most recent small allocation. The motivation for this second approach requires us to take a small diversion.

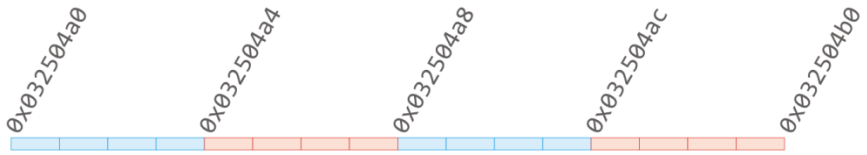
It has been observed that approximately 10 % of code is executed 90 % of the time in any application. Most of the other 90 % of the code is initialization, clean up, dealing with special cases, etc. One consequence of this is the introduction of caching. Main memory is not as fast as a processor, so if a processor was to read the next instruction from memory, it would have to wait many cycles doing nothing before that instruction is read (sometimes called *processor stalling*). This is a consequence of processor speed increasing faster than that of main memory. One solution was to introduce smaller amounts of faster memory called caches. Main memory is divided into, for example, 4 KiB pages and a cache holds a fixed number of frames for these pages. When an access to main memory is made, it is checked whether or not the page is in a frame of the cache. If so, it is immediately read; otherwise we have a *page miss*, the page is loaded from main memory into a frame of the cache, and computing continues.
If processors were accessing addresses randomly scattered throughout memory, such a scheme would be useless; however, it is the above observation that makes caches a reasonable strategy.

Now, for small allocations, these could come from some node-based data structure such as a tree. It would be much more preferable, given the design of caches, to have all such allocations in the immediate vicinity of other allocations so that as many as possible can fit into one page of memory.

Note: There are now multiple levels of caches, each one faster than the previous. The slowest, a level-1 cache (L1) has the largest frames, while the faster caches (levels 2, 3 and even 4, or L2, L3 and L4) will have smaller frames.

For more information on Doug Lea's malloc, see http://gee.cs.oswego.edu/dl/html/malloc.html. Reading this document, you will see that we have exited the world of computer science and entered the realm of computer engineering: there are many different requirements and constraints imposed by hardware, and there is no *ultimate* or *ideal* solution. Instead, minimizing space and time requirements, there are others compromises that must be struck in order to most efficiently deal with hardware design.

One example: Be willing to allow internal fragmentation to avoid alignment issues (even though main memory is byte addressable, 32- and 64-bit processors will read words at a time (4 or 8 bytes). For optimal performance, allocations should be aligned with these addresses.



Caveat: the smallest allocable block is 16 bytes in 32-bit systems and 24 bytes in 64-bit systems. Therefore, any application making significant use of allocations significantly smaller than these sizes should consider using an alternate approach to memory allocation.

5.2.3.4 Half fit

This is a dynamic memory allocation algorithm that is designed for real time systems. With any real time system, it is necessary to know the worst-case execution time and this is an algorithm that allows this.

Strategy: available blocks are placed into bins storing sizes in the range 2^k, …, $2^{k+1} - 1$ for $k = 0, 1, 2, \ldots$. A bit-vector is used to record which bins are empty. When a request comes in, a block is taken from the bin where that request is guaranteed to fit. Thus, any request of size 9 to 16 bytes would take a block from the bin storing blocks of size 16 to 32. No searching is performed. If the bin is empty, the next available bin is used. If necessary, the block is divided into allocated and free parts, the free part being reinserted into the appropriate bin. During deallocation, the freed block is coalesced with any surrounding block that is also unallocated. Unlike binary buddy, this requires the coalescing of at most three blocks.

This prevents very small fragments from being left over, but there is also the issue that for a sufficiently large request, there may be a block which could satisfy the request, but that request is denied. For example, there may be a free block of size 1100, but a request for a block of size 1050 would examine the bin for blocks of size 2048, …, 4095.

See Takeshi Ogasawara, *An Algorithm with Constant Execution Time for Dynamic Storage Allocation*.

5.2.3.5 Two-level segregated fit

This design has two levels of bins: the higher level is in powers of two, while each of these bins in turn is divided into 2^M bins for some reasonably small value of M. Each of these bins stores available blocks of that size. If a request comes in for a specific amount of memory, the most-significant bit indicates the overall bin, and the next M bits indicates the second-level bin. For example, a request for

$$618 = \mathbf{1}\mathbf{0011}01010_2 \text{ bytes}$$

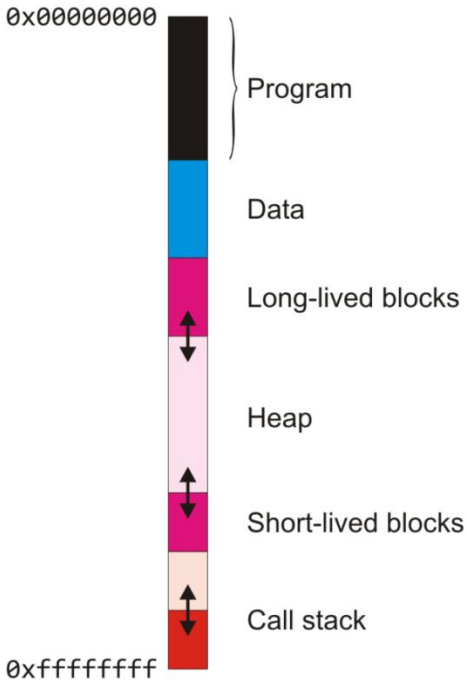
would examine bin 9 at the higher level and there it would check bins 3, 4, 5, …, 15 (where $M = 4$). Of course, if bin 9 was empty, we would proceed to look at bins 10, 11, etc.

Where possible, freed bins are coalesced. This scheme allows the wasted memory to be reduced to a minimum while still allowing $\Theta(1)$ allocation and deallocation.

See M. Masmano et al., *TLSF: a New Dynamic Memory Allocator for Real-Time Systems*.

5.2.3.6 Smart fit

This algorithm uses a novel approach: divide allocations into short-lived and long-lived blocks, growing a separate heap for each. The authors suggest that two factors can be looked at determine which category a request falls into: the size of the request and the number of allocation events.



There are other features which you can read about in Ramakrishna et al., *Smart Dynamic Memory Allocator for Embedded Systems*.

5.2.3.7 Summary of advanced memory allocation strategies

Algorithms such as first-, next-, best- and worst-fit all have weaknesses akin to those of linked lists. Additional structures such as those observed can often help improve the run time of memory allocators.

5.2.4 Summary of allocation strategies

We have discussed how bitmaps and linked lists can be used for allocating memory, and how one consequence of variable sized blocks is external fragmentation. We first looked at four simple algorithms and then we also considered six more advanced algorithms.

5.3 Case study: FreeRTOS

The real-time operating system FreeRTOS (see http://www.freertos.org/) comes with five dynamic memory allocation schemes, existing in the files `heap_1.c` through `heap_5.c`. Before we go through these, we should consider how casting works in C. Suppose we have a data structure:

```
typedef struct block_link {
    struct block_link *p_next_free;   /* The next free block in the list. */
    size_t size;                      /* The size of the free block. */
} block_link_t;
```

Suppose that each of the fields is four bytes in size. In that case, suppose we take an arbitrary pointer `ptr` and it is pointing to an arbitrary location in memory, as is shown in Figure 5-13.

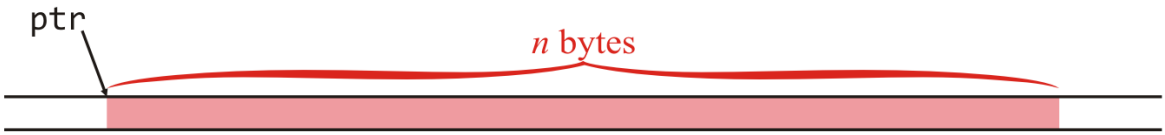


Figure 5-13. A pointer storing the address of a block of memory.

If we now cast that pointer

```
block_link_t *p_cast_ptr = (block_link_t *) p_ptr;
```

then the compiler will treat this as a record of 8 bytes, as sis shown in Figure 5-14.

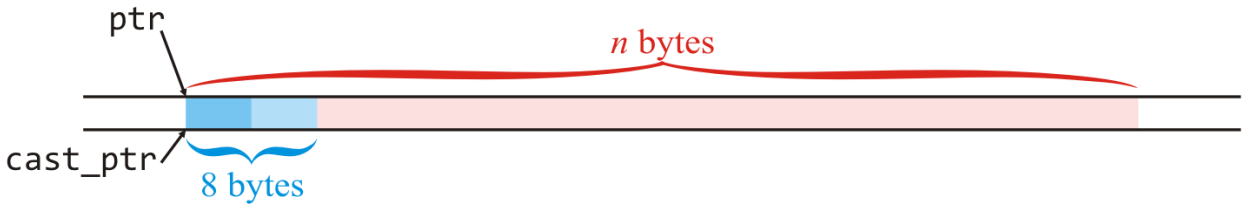


Figure 5-14. A block of memory cast as a specific record.

Thus, if we now assign:

```
p_cast_ptr->p_next_free = NULL;
p_cast_ptr->size = 42;
```

then those values will be stored in the first eight bytes overwriting whatever was there previously, as is shown in Figure 5-15.

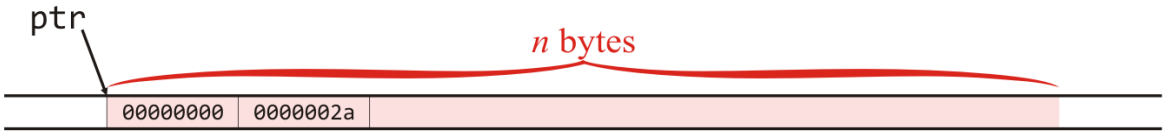


Figure 5-15. The memory in Figure 5-13 with the first eight bytes overwritten with NULL and 42.

Problem: What happens if the block of memory is less than the size of the structure?

5.3.1 Allocation only

The allocation strategy used in `heap_1.c` simply allocates memory from an array and never allows any form of deallocation:

1. First it checks to determine whether or not the allocated memory should be aligned with the word size of main memory. If so, it increases the size of *n* to a multiple of the word size.
2. Secondly, it ensures that the memory allocated neither exceeds the total memory that can be allocated nor causes an overflow.
3. Finally, if configured, it sets a hook (the error handling mechanism for FreeRTOS whereby this user-defined function is called) if the allocation failed.

While apparently trivial, it is best for numerous embedded systems where

1. all tasks are created and
2. all memory including that for queues and semaphores is allocated

when the system boots. Therefore memory deallocation will never be needed. As deallocation is unnecessary, there is no need for an implementation of the code to handle it. One implementation is here:

```
void *pvPortMalloc( size_t n ) {
    void *p_block = NULL;

    #if portBYTE_ALIGNMENT != 1
        if ( n & portBYTE_ALIGNMENT_MASK ) {
            n += ( portBYTE_ALIGNMENT - ( n & portBYTE_ALIGNMENT_MASK ) );
        }
    #endif

    vTaskSuspendAll(); {
        if ( ( ( xNextFreeByte + n ) < configTOTAL_HEAP_SIZE ) &&
            ( ( xNextFreeByte + n ) > xNextFreeByte ) ) {
                p_block = &( xHeap.ucHeap[ xNextFreeByte ] );
                xNextFreeByte += n;
        }
    }  xTaskResumeAll();

    #if ( configUSE_MALLOC_FAILED_HOOK == 1 )
        if ( p_block == NULL ) {
            extern void vApplicationMallocFailedHook( void );
            vApplicationMallocFailedHook();
        }
    #endif

    return p_block;
}
```

5.3.2 Best fit without coalescence

This simply allocates memory using a simple best fit strategy where it maintains a list of unallocated blocks sorted in the order of the size. When a block is deallocated, it is simply placed back into the list. No attempt is made to coalesce adjacent free blocks. When a block is allocated, the linked list structure is left untouched.

```
typedef struct block_link {
    struct block_link *p_next_free;   /* The next free block in the list. */
    size_t size;                      /* The size of the free block. */
} block_link_t;
```

The code is reasonably straight-forward; however we will look at how the programmers used a macro to avoid an unnecessary function call while still maintaining functional independence and coherence. The names of fields, parameters and local variables have been simplified for clarity.

```
#define prvInsertBlockIntoFreeList( p_block ) {                                          \
    p_block_link_t *p_itr;                                                               \
    size_t s;                                                                            \
                                                                                         \
    s = p_block->size;                                                                   \
                                                                                         \
    for ( p_itr = &xStart; p_itr->p_next_free->size < s; p_itr = p_itr->p_next_free ) { \
        /* Just iterate to the correct position. */                                      \
    }                                                                                    \
                                                                                         \
    p_block->p_next_free = p_itr ->p_next_free;                                          \
    p_itr->p_next_free = p_block;                                                        \
}
```

In C++, this could be avoided by using inline functions. Visually, the blocks are stored as is shown in Figure 5-16. The list is actually a linked list with sentinels where the first (`xStart`) and last (`xEnd`) nodes are dummy nodes with the first

having a size set to 0 and the last having a size set to the size of allocated memory. Thus, any allocated block will always be larger than the first and less than or equal to the last in size.

Figure 5-16. Storage of blocks in heap_2.c in FreeRTOS.

Such a scheme can be used if most of the blocks dynamically allocated and deallocated after system initialization falls within a fixed number of block sizes. If blocks of memory for data structures such as queues may have arbitrary size, this scheme will quickly result in significant amount of fragmentation.

5.3.3 Standard library malloc and free

The third implementation creates a thread safe wrapper of the standard library implementations of `malloc` and `free`.

```
void *pvPortMalloc( size_t n ) {
    void *p_block;

    vTaskSuspendAll(); {
        p_block = malloc( n );
        traceMALLOC( p_block, n );
    } xTaskResumeAll();

    #if ( configUSE_MALLOC_FAILED_HOOK == 1 )
        if ( p_block == NULL ) {
            extern void vApplicationMallocFailedHook( void );
            vApplicationMallocFailedHook();
        }
    #endif

    return p_block;
}

void vPortFree( void *p_block ) {
    if ( p_block != NULL ) {
        vTaskSuspendAll(); {
            free( pv );
            traceFREE( pv, 0 );
        } xTaskResumeAll();
    }
}
```

5.3.4 First fit with coalescence

In this implementation, we use a first-fit model, but also, the unallocated blocks are stored in address order. Consequently, at the same time a block is being deallocated, it can be checked with its neighbors to determine whether or not it can be coalesced. The schemes are similar to those already described, so we will focus on coalescence. Consider the memory allocated in Figure 5-17. Here, those blocks marked in red are allocated; those in black are unallocated. A linked list joins those that are unallocated and each block (allocated or unallocated) has a header with the size of the block and a pointer, which is only used if the block is unallocated.

Figure 5-17. First-fit with coalescence.

When an allocated block is freed, one would walk through the linked list until you have pointers to both the unallocated block immediately preceding the freed block in memory, and the one immediately following the freed block:

1. If all three are contiguous, they will be joined into a single block (so if the block of size 40 is unallocated, it would be merged with the two surrounding blocks forming one of size 16 + 40 + 24 = 80).
2. If the previous block is contiguous with the deallocated block, those two would be merged (so if the second block of size 16 is deallocated, it would be merged with the block of size 24).
3. If the next block is contiguous with the deallocated block, the would be merged (not shown).
4. Otherwise, the deallocated block becomes a link in the linked list (so if the block of size 56 is deallocated, it becomes another node in the linked list).

The routine for doing the coalescence is quite straight-forward and readable:

```
static void prvInsertBlockIntoFreeList( block_link_t *p_block ) {
    block_link_t *p_itr;
    uint8_t *p_previous;

    for( p_itr = &xStart; p_itr->p_next_free < p_block; p_itr = p_itr->p_next_free ) {
        // Find right block
    }

    p_previous = (uint8_t *) p_itr;

    // Test if the block can be merged with the previous block
    //  - increase the size of the previous block
    //  - treat the previous block as if it is the one being freed

    if ( (p_previous + p_itr->size) == (uint8_t *) p_block ) {
        p_itr->size += p_block->size;
        p_block = p_itr;
    }

    p_previous = (uint8_t *) p_block;

    // Test if the block can be merged with the following block
    //  - if we're at the end, just point to the trailing sentinel
    //  - increase the size of the block being freed
    //  - have block being freed point to the block after the following block

    if ( (p_previous + p_block->size) == (uint8_t *)( p_itr->p_next_free ) ) {
        if ( p_itr->p_next_free != pxEnd ) {
            p_block->size += p_itr->p_next_free->size;
            p_block->p_next_free = p_itr->p_next_free->p_next_free;
        } else {
            p_block->p_next_free = pxEnd;
        }
    } else {
        p_block->p_next_free = p_itr->p_next_free;
    }

    // Update the next pointer of the block prior to the block
    // being freed, but only if we are not filling a gap

    if ( p_itr != p_block ) {
        p_itr->p_next_free = p_block;
    }
}
```

The source code had `(uint8_t *) p_itr->p_next_free` , so, very quickly, is this equivalent to:

1. casting `p_itr` as a pointer to an byte and then accessing the field `p_next_free`, or
2. accessing the field `p_next_free` and casting it as a pointer to a byte?

That is, is it `((uint8_t *) p_itr)->p_next_free` or `(uint8_t *)( p_itr->p_next_free )` ?

Almost no thought will make it clear that the first is absurd in this case—`uint8_t` is a primitive data type and not a structure, it has no fields—but what if it was a structure? The notation

```
(uint8_t *)( p_itr->p_next_free )
```

while, arguably unnecessary for an experienced programmer, is still easier to immediately recognize without additional thought.

5.3.5 First fit with coalescence over multiple regions

This final version is identical to that described in the previous section, only it does not require the block of available memory to be one large contiguous block. The available memory can itself be separated throughout the memory of the system.

5.3.6 Summary of the case study

In summary, FreeRTOS includes five possible schemes for dynamic memory allocation. Only the first, the most trivial, is real-time while the other tasks are O(*n*) in the number of blocks that have been deallocated.

5.4 Other features: clearing and reallocation

In addition to `malloc`, there are two other related functions: `calloc` (clear allocate) and `realloc` (reallocation).

By default, asking for memory through malloc will have the operating system find an appropriate block of memory, mark it as allocated, and return a pointer to the first address of that block. The contents of that block, however, are not modified. It contains whatever data may have previously been in that memory, which may or may not be meaningful or bogus data. The call `calloc` sets all the bits in that block to zero.

Suppose you allocated an array of size *n*, but then realize later you require an array of size $n + m$. Normally, this would require you to create a new array, copy the information over, and then destroy the old array. However, what if there is memory available immediately after the memory currently allocated? Could not the operating system just expand the block of memory that has been allocated? The `realloc` command attempts to do this. If it is successful, the allocated memory is expanded. If that memory is not available, the operating system finds a sufficiently large block of memory and copies over the contents of the original array into that larger block (for example, using `memcpy`).

```
#include <stdio.h>
#include <stdlib.h>

int main( void ) {
        int i;

        int *p_data = (int *)malloc( 10*sizeof( int ) );
        printf( "The address of 'p_data' is %p\n", p_data );

        for ( i = 0; i < 20; ++i ) printf( "%d ", p_data[i] );
        printf( "\n" );

        for ( i = 0; i < 10; ++i ) p_data[i] = i*i;

        p_data = (int *)realloc( p_data, 20*sizeof( int ) );
        printf( "The address of 'p_data' is %p\n", p_data );

        p_data = (int *)realloc( p_data, 1000000*sizeof( int ) );
        printf( "The address of 'p_data' is %p\n", p_data );

        for ( i = 0; i < 20; ++i ) printf( "%d ", p_data[i] );
        printf( "\n" );

        free( p_data );

        return 0;
}
```

The output is

```
$ gcc example.c
$ ./a.out
The address of 'p_data' is 0xacaa010
0 0 0 0 0 0 0 0 0 0
The address of 'p_data' is 0xacaa010
0 1 4 9 16 25 36 49 64 81 135121 0 0 0 0 0 0 0 0 0
The address of 'p_data' is 0x2b696c62a010
0 1 4 9 16 25 36 49 64 81 135121 0 0 0 0 0 0 0 0 0
$
```

The first ten entries happen to be zero, but when realloc is called and the block of memory is expanded, it happens that the next entry is non-zero: `00000000 00000010 00001111 11010001`

5.4.1 Reallocation in C++ and the move constructor and move assignment operator

Can you call `realloc` in C++? Generally, no: recall that when memory is allocated, it is also necessary to call any constructors. As objects may have pointers to themselves, it may not be possible to simply copy the memory over. The vector class in C++ will allocate new memory and move the objects over; indeed, in C++-11, there is a new move constructor and a new move assignment operator—two functions that may assume that the previous objects are being destroyed anyway. A copy constructor may have to make a deep copy of larger data structures while the move constructor may be much easier. For example:

The copy constructor for a binary search tree would have to make a complete copy of the entire tree—an $\Theta(n)$ operation. The move constructor, however, would only have to copy over the address of the root and a few other variables—an $\Theta(1)$ operation.

5.5 Summary of dynamic memory allocation

The previous topic was on memory allocation. We discussed static allocation—memory that can be allocated by the processor either as (1) global variables or static local or member variables allocated in a region adjacent to the instructions; or (2) as local variables allocated relative to a frame on a call stack. Dynamic memory allocation is more complex and has issues associated with it that are not applicable to static memory allocation.

We have looked at a number of memory allocation algorithms spanning a wide gambit of ideas and approaches. Numerous data structures are used to try to efficiently allow the allocation of memory in the shortest time and with minimal internal and external fragmentation.

Note that it is not necessary to have an operating system in order to perform dynamic memory allocation. Recall that a microprocessor need have only a single executing task.

Problem set

5.1 What is the minimal interface required for a dynamic memory allocator ADT?

5.2 Recall that the difference between $O(n)$ and $\Theta(n)$ is that the first describes a situation in which the worst-case run time is linear, but the system may end early, and the second describes a situation where the run-time is always linear. For each of the following, determine which is the most appropriate Landau symbol to use:

1. find the maximum entry in an array,
2. find the average value of the entries in an array,
3. find if an array contains a specific entry,
4. find if an array contains an entry in a given range, and
5. determine if the entries of an array are monotonically increasing.

5.3 What are the run times of the following algorithms:

4. first-fit,
5. best-fit,
6. worst-fit, and
7. next fit.

5.4 Explain why it may be prudent to have separate memory allocators in an embedded system. For example, one providing fixed-sized blocks of memory

5.5 Can the run-time of a memory allocation scheme ever be worse than $\Theta(n)$ where n is the number of unallocated blocks?

5.6 The dynamic memory heap grows normally from one end of an available block of memory. How could you use this larger block of memory to have two separate allocation schemes?

5.7 In the previous question, we considered growing two different heaps. For one, you may have a more simple memory allocation scheme running in $\Theta(1)$ time, while the other is $O(n)$ in the number of unallocated blocks. For example, one may allow the allocation of arbitrarily sized blocks, while the other allocates only fixed-sized blocks. What could be done if no more memory is available?

5.8 List some of the advantages of using an automatic memory allocation scheme (using garbage collection) as opposed to a manual memory allocation scheme (using explicit calls to allocate and deallocate memory).

5.9 A reference counting scheme may be appropriate for a real-time system, as incrementing or decrementing the number of pointers that store the address of a block of memory can be done in $\Theta(1)$ time. Unfortunately, this may also requires the user to explicitly set some pointers to `NULL` once they are no longer required (such as in a cyclic list). Can this still be considered *automatic memory management*?

5.10 What are some constraints on the memory allocation requirements of a real-time system if that system is to use some form of reference counting scheme?

5.11 What is the most significant issue of a mark-and-sweep based algorithm for garbage collection with respect to real-time systems? If a mark-and-sweep algorithm was to be used, what additional requirement would you have to have on any task that is associated with hard deadlines?

5.12 In Java, you can only access an entry in an array by using the indexing operator; for example, you can only ever use `array[9]` to access the 10th entry in the array. In C++, you could do something more interesting such as

Java

```
for ( int i = 0; i < 10; ++i ) {
     p_array[i] = 0;
}
```

C

```
int *p_itr = p_array; // the variable will iterate
                      // through the array

for ( i  = 0; i < 10; ++i ) {
     *p_itr++ = 0;
}
```

What are some pitfalls that Java can avoid by requiring the user to always access array entries using an array index (with respect to garbage collection)?

5.13 Explain how you would implement a `realloc` function in the half-fit memory allocation scheme.

5.14 Explain how you would implement a `realloc` function in the binary buddy memory allocation scheme.

5.15 Starting with an initial contiguous block of 64 KiB, perform the memory allocation:

1. allocate 28 KiB,
2. allocate 10 KiB,
3. allocate 15 KiB,
4. deallocate 10 KiB,
5. allocate 8 KiB,
6. allocate 3 KiB,
7. deallocate 8 KiB,
8. allocate 7 KiB,
9. deallocate 28 KiB, and
10. allocate 20 KiB,

using first-, next-, best- and worst-fit algorithms, both with coalescing of adjacent blocks and non-coalescence of adjacent blocks. If an algorithm cannot allocate a given block, attempt to allocate that block immediately following any subsequent deallocations.

Note that with next fit, if a block is broken into two (the first half allocated, the second half not), the next request for memory inspects the second half first.

5.16 The best-fit algorithm in FreeRTOS uses a sorted list, but that sorted list uses a linear ordering. Create a data structure that casts a node as one in a balanced binary search tree. Note that you have a choice between red-black trees and AVL trees where with

1. red-black trees, you only require one additional bit to indicate the *color* of the node (red or black); and
2. AVL trees, you must store the heights of the left and right sub-trees.

The latter normally requires a field that can hold numbers as large as 2 ln(*n*) where *n* is the maximum number of nodes in the tree (this overestimates the maximum height of an AVL tree with *n* nodes). Therefore, four bits could be used to store the maximum height of an AVL tree with up to $n = 1808$ nodes. Better yet, because the *difference* in height is really all that matters, instead, we could come up with a different scheme:

1. $0 = 00_2$ indicates the node is balanced,
2. $1 = 01_2$ indicates the node is right-heavy, and
3. $2 = 10_2$ indicates the node is left-heavy.

Thus, if the height of the left sub-tree was increased by one as a result of an insertion:

1. if the balance was 0, the balance becomes 2,
2. if the balance was 1, the balance becomes 0, and
3. if the balance was 2, the tree is now AVL unbalanced.

Thus, an AVL tree can be represented with only two additional bits (as opposed to storing the height of each node).

5.17 The worst-fit algorithm requires a heap. A leftist heap is a node-based data structure that requires references to both children; however, both finding a node within any heap with *n* nodes is usually an O(*n*) operation. If we are implementing worst fit without coalescence, this is not an issue: we are only putting nodes into the tree and popping the top. If we want to coalesce adjacent memory blocks when one is deallocated, however, we have a problem: we must remove the nodes from the heap. What additional information is required in the heap for this purpose?

How many addresses does your scheme require for each entry in the heap?

6 Threads and tasks

Previously, when you have created an executable, the function `main( void )` begins running and it may call additional functions which call other functions, and so on; however, such an execution path is serial: at any one time, only one instruction is being executed by the processor. This makes, for example, the run-time analysis of programs relatively easy, as we are reduced to solving a recurrence relation, most of which fall into a small collection that can be solved by the master theorem. However, this leads to a number of issues.

With respect to the non-functional requirements of real-time systems, tasks allow a system to be scalable and ensures reasonable performance by allowing each task to focus on one purpose, with scheduling ensuring that the appropriate task runs at the appropriate time. At the same time, creating a proper framework for organizing and managing tasks ensures scalability.

Thus, we will look at how

1. you would implement an embedded system using a single sequence of execution,
2. threads are created in various systems and what information is required,
3. threads may be used to solve problems,
4. how we can track the relationship between threads, and
5. the `volatile` keyword in C.

We start with the weaknesses of having just a single sequence of execution.

6.1 Weaknesses in single threads

Consider the following example which tries to describe how an embedded system could be implemented with a single thread:

```
int main( void ) {
    int i, j;
    queue q1, q2;

    queue_init( &q1 );
    queue_init( &q2 );
    init();

    for ( i = 0; 1; ++i ) {
        if ( sensor_1_ready() ) {
            queue_push( &q1, get_sensor_value_1() );
        }

        for ( j = 0; j < 10; ++j ) {
            if ( sensor_2_ready() ) {
                queue_push( &q2, get_sensor_value_2() );
                break;
            }
        }

        emergency_flag = process_data( q1, q2 );

        if ( emergency_flag ) {
            critical_response();
        }

        if ( communications_ready() ) {
             while ( !queue_empty( &q1 ) ) {
                 send( queue_front( &q1 ) );
                 queue_pop( &q1 );
             }
        }

        if ( (i & 127) == 0 ) {
```

```
                check_system_stability();
            }
        }
    }
```

Here, two sensors are being read whenever they are ready, a response occurs if an issue arises from the processing of that data, the data is sent to some destination if the communications port is ready, and every-so-often, check that the system is still stable (once every 128 cycles), perhaps taking steps to rectify the situation.

There are numerous weaknesses in this arrangement:

1. The system check is of low priority, but what happens if the system check occurs just when an issue that requires an immediate response occurs?
2. Suppose the system check usually takes only 20 μs, but
3. It seems also that the communications system is not really of high priority, so sending a packet containing data may result in, again, the system not responding to a more critical situation.

Now, this is with only two sensors and one response: what happens if there are multiple sensors and multiple possible responses based on input from those sensors? We cannot continue to easily make the system both responsive and more complex. Instead, we will try to break the problem down into individual tasks and to then execute each task separately. There are numerous independent tasks that are going on, including:

1. getting data from the sensors,
2. determining if there is a critical situation requiring an immediate response,
3. sending data off site, and
4. periodically checking system stability.

Let's discuss how to do this.

6.2 Creating threads and tasks

Remember that `main(…)` is nothing more than a function. When we start executing a thread, it is necessary start executing something, but what?

The easiest way to start a new task is to say: run this function, but run it as a new task.

The general mechanism of creating multiple threads is to pass an initialization function a pointer or reference to another function that is to be executed not as a function call, but as a second parallel *thread* of computation or *task*. The term *thread* is appropriate, as each thread is, in itself, still a sequential sequence of instructions being executed.

The terms threads and tasks will be used interchangeably in this course, as such independent sequences of execution are called

1. "threads" in operating systems such as Linux, the `pthread.h` library, and in Java, but
2. "tasks" in the Keil RTX RTOS and other embedded systems.

The name is related to the focus of application: from an operating systems point-of-view, the focus is on independent execution, while in embedded systems, the focus is on achieving separate goals. We will use both terms.

Thus,

> a *task* is a sequence of instructions that may be executed independent of other such tasks or threads on one or more processors.

Separate threads will therefore have

1. separate states of the processor (registers),
2. some means of differentiating them (thread identifiers), and
3. additional information.

If two threads or tasks are being executed on the same processor, we will require some mechanism of choosing (that is, scheduling) which is to be executing at any one time. This is the subject of the next topic. Now we will focus on the purpose and generation and use of tasks and threads. At this point, we will now look at the generation of

1. threads in POSIX,
2. threads in Java,
3. tasks in the Keil RTX RTOS, and
4. threads in the CMSIS-RTOS RTX.

Following this, we will see applications of multiple tasks and threads.

6.2.1 Threads in POSIX

In POSIX, the signature for the command for generating a new thread is

```
pthread_create( pthread_t *p_thread,
                const pthread_attr_t *p_attributes,
                void *(*start_routine)( void * ),
                void *p_arguments );
```

where

1. `thread` is a pointer to a `pthread` identifier and identifier will be assigned a value when the thread is created,
2. the attributes may specify various characteristics about the thread, but this can be `NULL` to use the default values,
3. the start routine is any function that takes a single untyped pointer that returns an untyped pointer, and
4. the argument that is passed to the function is passed as the fourth argument.

Any arguments to the thread that is to begin executing as a separate thread must therefore be reinterpreted as a pointer, usually to a structure of some sorts. When the thread exits, it may want to return information to the thread that created it. As an example, we may have two arguments and two return values. In this case, we would define structures such as:

```
typedef struct {
    typename param1;
    typename param2;
} parameters_t;

typedef struct {
    typename ret1;
    typename ret2;
} return_t;
```

Next, to create the thread, we create an instance of our parameters (making sure that they remain in scope for the full duration of the existence of the thread being created).

```
parameters_t args;

args.param1 = some_value;
args.param2 = another_value;

pthread_t thread_id;  // Will be assigned in pthread_create
```

```
pthread_create( &thread_id, NULL, function_name, &args );

// This function and the created function are now running in parallel
```

Neither thread is guaranteed to be the one that continues running once `pthread_create` returns.

The function that is to be run as a separate thread must be expecting a pointer to its arguments:

```
void *function_name( void *p_void_arg ) {
    // The argument is an arbitrary pointer (a "void pointer")
    //  - cast it to a pointer to an instance of 'parameters_t'

    parameters_t *p_args = (parameters_t *)p_void_arg;

    // Now you can use p_args->param1 and p_args->param2

    while ( 1 ) {
        // Infinite loop
    }
}
```

If the thread is to exit, we can call `pthread_exit(…)`:

```
void *function_name( void *p_void_arg ) {
    // the argument is an arbitrary pointer (a "void pointer")
    //  - cast it to a pointer to an instance of 'parameters_t'

    parameters_t *p_args = (parameters_t *)p_void_arg;

    // Now you can use p_args->param1 and p_args->param2

    // If you want to exit, you can call 'pthread_exit(...)'
    //  - you must point to something that exists outside this thread
    return_t* p_ret = (return_t *) malloc( sizeof( return_t ) );  // See footnote 13
    p_ret->ret1 = value;
    p_ret->ret2 = value;

    pthread_exit( p_ret );
}
```

One possibility for returning a value is for the creating thread to have a local variable and then pass the address of that local variable as an argument. The second thread can then modify the value of that local variable.

If the thread exits, the calling thread must, at some point, join with it:

```
pthread_create( &thread_id, NULL, function_name, &args );

// This function and the created function are now running in parallel

void *p_void_ret;

pthread_join( thread_id, &p_void_ret );

return_t *p_ret = (return_t *)p_void_ret;
```

[13] Under best practices, it is undesirable—or forbidden—to allocate memory dynamically at any time after task initialization. Ideally, the memory for the return value should have been allocated at the time the task was initialized.

```
// You can now use ret->ret1 and ret->ret2
```

The `pthread_join(…)` will not return until after `pthread_exit(…)` is called by the created thread. Consequently, if the created thread never exits, the function `pthread_join(…)` will never return.

As a short-cut, if a thread has no return values, it need not call `pthread_exit(…)`. Instead, it simply returns `NULL` and the second argument of `pthread_join(…)` is `NULL`

6.2.2 Threads in Java

We will take a minute to look at how parallel threads can be run in Java. In C, for a file to be convertible into an executable, it must have a `int main(…)` function. In Java, each file holds exactly one class and the name of the file must be the name of the class. For a class to be executable, it must have a `public static void main(…)` method.

file_name.c

```
int main( int argc, char *argv[] ) {
    // Some code

    return EXIT_SUCCESS;
}
```

ClassName.java

```
public class Class_name {
    public static void main( String[] args ) {
        // Some code
    }
}
```

In C, any function can be executed as a thread. In Java, however, only classes that have a `public void run()` method can be executed and it is that method that is run.

file_name.c

```
int run( void *void_arg ) {
    arg_t *arg = (arg_t *)void_arg;
    // Some code
}
```

Somewhere else:

```
pthread_t t;
arg_t args = …;
pthread_create( &t, NULL, run, &args );
```

ClassName.java

```
public class ClassName implements Runnable {
    ClassName(…) {
        // Constructor
    }

    public void run() {
        // Some code
    }
}
```

Somewhere else:

```
Thread t = new Thread( new ClassName(…) );
t.start();
```

For the new thread to execute, the thread class `start()` function looks in the class for the `run()` function and calls it. Now, it could simply always look for the `run()` function each time a new thread is created, but this could be quite problematic: a newbie programmer may come along and say "Hey, this is a silly function name. Let's change it to `public void start()`." The `.java` file would still compile into a `.class` file, and it would only be later on that the error would be caught.

Note that in C, the arguments are passed through an additional argument to `pthread_create`, while in Java, any arguments would be passed to the constructor of the class. Difference instances of the `ClassName` class could be passed different parameters in the constructor.

Now, to execute a class as a new thread requires the existence only one function; however, suppose a class, such as a graph data structure, has a large number of members. In this case, each time, you'd have to check whether or not all the methods were implemented, and this could cause problems.

Java's solution is to introduce *interfaces*. All an interface is a collection of signatures, and if you state that a class *implements* that interface, it must have implementations of all the signatures; if it doesn't, the compilation fails. Now, any class needing that interface need only check if the class has that interface. The class could not compile if there was a missing implementation or a changed name.

Note that one of the major design decisions around Java was to create a version of C++ that improved on many of the error-prone aspects of C++. For example: they removed pointers; you could no longer use public: and private: labels, instead, visibility each method had to be identified individually; it was a truly object-oriented programming language where all classes are derived from an ultimate Object class using only single inheritance, etc. The introduction of interfaces was only one more of these adjustments to reduce errors in programming and development.

Note the difference between the object-oriented design and the procedural design:

Procedural	Object-oriented
The function `pthread_create` is called	A `Thread` object is created
A reference to the thread is created as a thread identifier passed as an argument	The object created is the reference to the thread
The characteristics of the thread are specified in a pointer to a structure	The characteristics of the thread are additional arguments to the thread constructor
A function pointer is passed as an argument	An instance of a class implementing the `Runnable` interface is passed as an argument to the thread constructor
The arguments to the function are passed as an untyped pointer	The arguments to the thread being created are passed in the constructor of the runnable class

6.2.3 Tasks in the Keil RTX RTOS

In the Keil RTX, task generation is similar to that of POSIX `pthread` library, but there are four options:

1. The task does not take arguments, such as

```
#include <rtl.h>

int main( void ) {
    OS_TID task_id;

    task_id = os_tsk_create( task_name, priority );

    if ( task_id == 0 ) {
        // task was not created
    }

    // Continue executing

}

__task void task_name( void ) {
    // Executing...
}
```

2. The task takes arguments,

```
#include <rtl.h>

int main( void ) {
    OS_TID task_id;
    argument_t *p_arguments;

    // Initialize p_arguments;

    task_id = os_tsk_create_ex( task_name, priority, p_arguments );

    if ( task_id == 0 ) {
        // task was not created
    }

    // Continue executing
}

__task void task_name( void *p_void_arg ) {
    // Executing...
}
```

3. The task does not take any arguments, but the call task passes a stack defined in user space, and
4. The task takes arguments, and the calling task passes a stack defined in user space.

Recall that each thread will require its own function call stack. In the first two instances, the operating system provides each of the tasks with their own call stack. In the latter two, the user can choose a different sized call stack and pass it in. We will discuss the difference between *user space* and *kernel space* later in the course.

We will discuss priority at a future point in time when it comes to scheduling tasks where one may be more *important* than another.

6.2.4 Threads in the CMSIS-RTOS RTX

There are multiple operating systems for the same hardware, and there is a more primitive operating system for all Cortex-M microcontroller called the CMSIS-RTOS RTX. Like POSIX, CMSIS is a common interface to interacting with any Cortex-M microcontroller and it stands for *Cortex Microcontroller Software Interface Standard*. (Recall that POSIX stands for *Portable Operating System Interface for Unix*). The CMSIS-RTOS RTX does not have all the features of the Keil RTX and it does not have the concept of a *task* and the scheduler has many fewer *priorities*.

```
// Thread creation in the CMSIS-RTOS RTX
#include "cmsis_os.h"

void thread_name( void const *p_arguments );

osThreadDef( thread_name, osPriorityNormal, 1, 0 );

void main( void ) {
    osThreadId thread_id;
    thread_id = osThreadCreate( osThread(thread_name), NULL);

    if ( thread_id == NULL ) {
        // thread was not created
    }

    // Executing...

    osThreadTerminate( thread_id );
}

// Thread creation in the Keil RTX
#include <rtl.h>

int main( void ) {
    OS_TID tsk_id;

    tsk_id = os_tsk_create_ex( task_name, priority, p_arguments );
    // Continue executing

}

__task void task2( void ) {
   ..
}
```

6.2.5 Summary of thread and task creation

We have looked at some aspects of thread and task creation and thread interaction. Next, we will look at the purposes behind threads. We will learn about priorities later.

6.3 Applications of threads and tasks

We have previously discussed the concepts of the procedural programming paradigm. One requirement of this paradigm is that a function (or procedure) performs exactly one well defined task with defined input and a defined transformation on that input. When you consider all the different goals of the initial example we gave of an embedded system implemented as a single loop, this strongly suggests we are doing something wrong here, too.

A thread or tasks preforms a sequence of instructions that may be performed, for the most part, independent of other sequences of instructions achieving a well-defined goal. Threads or tasks may, however, still share information throughout execution.

Threads and tasks can be used to

1. solve different problems posed by the system,
2. break down a larger problem into smaller problems, and
3. specifically, solving a divide-and-conquer algorithm.

The benefit of breaking problems into independent threads and tasks is that they can be executed, at least potentially, in parallel. Such parallelism may be achieved through either:

1. having multiple cores or dependent processors,
2. having independent processors possibly remote from each other, and
3. artificially through task scheduling.

For the balance of this talk, we will introduce the first two, while in the next topic, we will discuss scheduling.

6.3.1 Parallel execution

Consider a repetition loop where each iteration is independent of the others. In this case, such a program could be run in parallel: in the extreme case, each iteration could be run on a separate processor, and the execution time would be reduced to the execution time of one statement. In most real-time applications, there is less of an emphasis on parallel computation, but it is useful to consider at least a few results.

Suppose that a particular task can be executed in 5 s, and this contains a loop that executes a significant number of times in such a way that it can be parallelized. If the initialization and finalization code requires 400 ms and the code within the loop is negligible, the most such a block of code could be sped up is by a factor of $\frac{5}{0.4} \approx 12.5$. It is not possible to do better than this, and it will not be possible to achieve this limit, either. However, if you had two processors, the time required would be 400 ms for the initialization and finalization, and the remaining instructions normally taking 4.6 s would be performed on two processors, thereby taking only 2.3 s, for a total of 2.7 s, or a speed up of 46 %.

Amdahl's law gives the theoretical maximum improvement of a system if you improve only one component of it. If you apply this to parallel computation, you are only able to improve that component that is parallelizable. The application of this law, in this case, is that if you use *n* processors and *B* is the proportion of time that is strictly serial, so the time to execute with *n* processors is:

$$T(n) = T(1)\left(B + \frac{1}{n}(1-B)\right)$$

In our example here, $B = \frac{0.4}{5} = 0.08$ and $T(2) = T(1)\left(B + \frac{1}{n}(1-B)\right) = 2.7$. If we continue to double the number of processors, we get the following run times:

Processors	Run time
1	5
2	2.7
4	1.55
8	0.975
16	0.6875
32	0.54375
64	0.471875
128	0.4359375

Note the diminishing returns: as you throw more and more processors at the problem, the improvement becomes negligible. In the next topic, we will see how parallel computation can significantly improve the run time of a divide-and-conquer algorithm.

6.3.2 Divide-and-conquer algorithms

In your algorithms and data structures course, you have already seen a number of divide-and-conquer algorithms, possibly including:

1. binary search,
2. quicksort, and
3. merge sort.

There are numerous other applications of this type of algorithm, including:

1. fast integer multiplication,
2. fast matrix-matrix multiplication, and
3. fast Fourier transform.

All of these are recursive functions; that is, the function calls itself. Let's look at merge sort as implemented in C.

Note that the divide-and-conquer strategy can often be applied even if it does not benefit the overall run-time. Consider, for example, a matrix-vector multiplication. If

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{pmatrix} \text{ and } \mathbf{v} = \begin{pmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{pmatrix},$$

then we may reduce the multiplication of an $n \times n$ matrix and an n-dimensional vector to four multiplications of $n/2 \times n/2$ matrix and an $n/2$-dimensional vector. Except where $\mathbf{A}$ has a very special shape (such as is the case with the discrete Fourier transform allowing us to find an $n \ln(n)$ algorithm), the run time will always be $\Theta(n^2)$.

First, it is wasteful to use such algorithms if the size of the list being sorted is small, say around 30. The overhead of the additional function calls is too large; thus we use a fast in-place sorting algorithm, such as insertion sort, which sorts the entries of the array from index a to index $b - 1$.

```
void insertion_sort( double *p_array, size_t a, size_t b ) {
    size_t j, k;
    bool found = false;
    double tmp;

    for ( k = a + 1; k < b; ++k ) {
        tmp = p_array[k];

        for ( j = k; !found && j > a; --j ) {
            if ( p_array[j - 1] > tmp ) {
                p_array[j] = p_array[j - 1];
            } else {
                found = true;
            }
        }

        p_array[j] = tmp;

    }
}
```

We use insertion sort because its runtime is $\Theta(n + d)$ where d is the number of *inversions* in the list and where $d = O(n^2)$. In the worst case, insertion sort is $\Theta(n^2)$, but if the number of inversions is small, it will be a very fast algorithm.

Now we can continue implementing merge sort. Recall that in merge sort,

1. if the list is under a certain size, call insertion sort;
2. otherwise,
    a. divide the list in two,
    b. call merge sort recursively on both halves, and
    c. merge the resulting lists.

Having taken our previous advice, we note that the merging process is essentially distinct from the algorithm, so we write a separate function.

```
#include <assert.h>

// Merge the entries from a to b - 1 and from b to c - 1
void merge( double *p_array, size_t a, size_t b, size_t c ) {
    assert( a <= b && b <= c );

    size_t i = 0, j = a, k = b;
    double *p_sorted_array = (double *) malloc( (c - a)*sizeof( double ) );

    while ( j < b && k < c ) {
        if (p_array[j] <= p_array[k] ) {
            p_sorted_array[i] = p_array[j];
            ++j;
        } else {
            p_sorted_array[i] = p_array[k];
            ++k;
        }

        ++i;
    }

    for ( ; j < b; ++i, ++j ) {
        p_sorted_array[i] = p_array[j];
    }
```

```
    for ( ; k < c; ++i, ++k ) {
        p_sorted_array[i] = p_array[k];
    }

    for ( i = 0, k = a; k < c; ++i, ++k ) {
        p_array[k] = p_sorted_array[i];
    }

    free( p_sorted_array );
}
```

Now we can implement merge sort:

```
void merge_sort( double *p_array, size_t a, size_t c ) {
    assert( a <= c );

    if ( c - a < USE_INSERTION_SORT ) {
        insertion_sort( array, a, c );
        return;
    }

    size_t b = a + (c - a)/2;

    merge_sort( p_array, a, b );
    merge_sort( p_array, b, c );
    merge( p_array, a, b, c );
}
```

Note: Why do we use

```
size_t b = a + (c - a)/2;
```

instead of

```
size_t b = (a + c)/2;
```

This is actually more relevant to mechatronics students than the average programmer: the sum `(a + b)` may overflow, so you may get some interesting results. For example, suppose your type was only eight bits and you call merge sort:

1. with `merge_sort( array, 0, 180 )`, the mid-point *b* is (0 + 180)/2 = 180/2 = 90, but
2. the second recursive call is `merge_sort( array, 90, 180 )` and 90 + 180 = 270 > 256, so the arithmetic-logic unit would calculate the mid-point *b* to be 7.

Now, let us use parallel routines to perform each recursive call separately.

For our merge sort routine, we must pass a pointer to the array and the initial and one-past-the-final positions. Thus, we must define a structure that holds all of these arguments:

```
typedef struct {
    double *p_array;
    size_t a;
    size_t c;
} interval_t;
```

Now, a user doesn't want or care that we are using a parallel algorithm, so we will instead provide an interface for the user that is more natural. Also, we do not have to create a separate thread at this point, because there is only one task being performed:

```
void merge_sort( double *p_array, size_t n ) {
    interval_t arg;
```

```
    arg.p_array = p_array;
    arg.a = 0;
    arg.c = n;

    merge_sort_internal( &arg );
}
```

Our internal merge sort must first recast our argument as a pointer to our argument structure. Following that, we calculate the mid-point and then prepare the arguments for our recursive calls.

```
void *merge_sort_interal( void *p_void_arg ) {
    // the argument is an arbitrary pointer (a "void pointer")
    //  - cast it to a pointer to an instance of 'interval_t'

    interval_t *p_arg = (interval_t *)p_void_arg;

    if ( ( p_arg->c - p_arg->a ) < USE_INSERTION_SORT ) {
        insertion_sort( p_arg->array, p_arg->a, p_arg->c );
    } else {
        size_t b = p_arg->a + (p_arg->c - p_arg->a)/2;

        interval_t arg1, arg2;

        arg1.p_array = p_arg->p_array;
        arg1.a = p_arg->a;
        arg1.c = b;

        arg2.p_array = p_arg->p_array;
        arg2.a = b;
        arg2.c = p_arg->c;

        pthread_t other_thread;

        // Create a thread to sort the second half
        pthread_create( &other_thread, NULL, merge_sort_interal, &arg2 );
        merge_sort_interal( &arg1 );

        // Wait for them to finish
        pthread_join( other_thread, NULL );

        merge( p_arg->p_array, p_arg->a, b, p_arg->c );
    }

    return NULL;
}
```

Note that we are using a recursive algorithm to demonstrate threads; however, in general, recursive algorithms should never be used in an embedded system. This is echoed in Rule 4 of the JPL coding standard:

"There shall be no direct or indirect use of recursive function calls."

The execution is shown in Figure 6-1. Depending on the various run times, there could be up to five threads running in parallel.

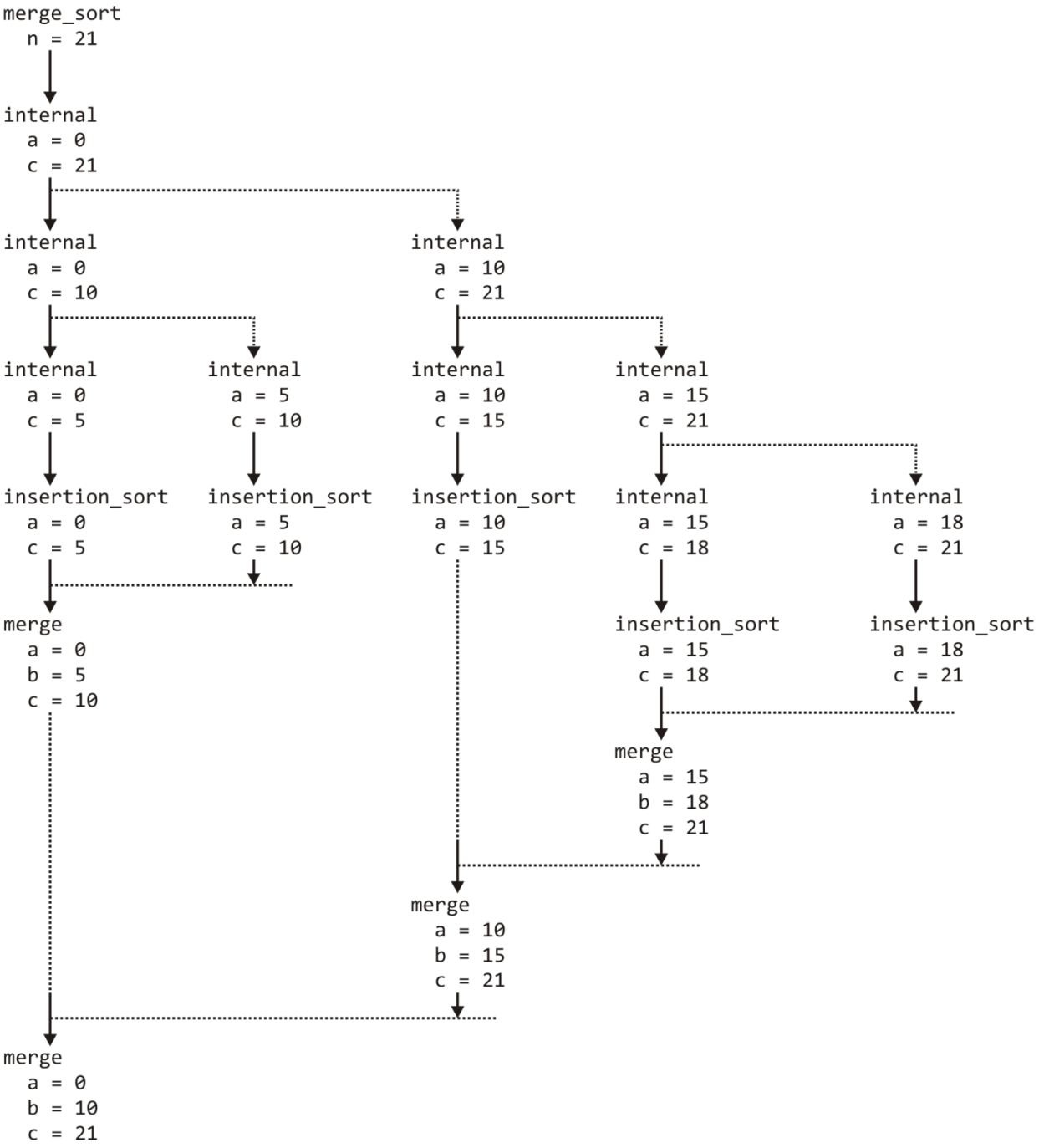


Figure 6-1. Using merge sort to sort an array of size 21 in parallel.

In your algorithms and data structures course, you saw the run time of merge sort was $\Theta(n \ln(n))$; however, if the separate threads are running on separate cores, the run time is now $\Theta(n)$.

6.3.3 Independent versus dependent tasks

The this section, we have looked at using threads and tasks to accomplish independent duties that allow where each thread can run independently of the others. The only dependence is that the parent thread or task must wait for any descendants to finish. In such a case, it is likely possible that separate threads and tasks can be run in parallel if we have multiple processors or cores. Usually, however, tasks cannot run independently of others. For example, tasks may have to share data or other resources, and this can result in the corrupting the data structure (as we saw when two tasks simultaneously attempt to modify a linked list). We will look at the issues of synchronization in Chapter 9 and deadlock in Chapter 11.

6.3.4 Application of threads and tasks

We have looked at three applications of threads and tasks:

1. parallel execution,
2. divide-and-conquer algorithms, and
3. accomplishing independent tasks.

In all three cases, the threads and tasks ran in parallel and were independent, thereby allowing for parallel processing. With sufficiently many cores or processors, algorithms such as quicksort and merge sort can execute in $\Theta(n)$ time. Next we will see, at a high level, how to maintain threads, and in subsequent chapters we will look at issues of synchronization, resource sharing, deadlock and inter-process communication.

6.4 Maintaining threads

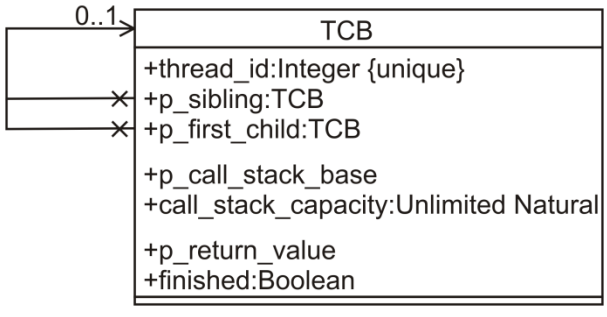
With multiple threads and tasks, in general, we need some form of mechanism for handling these information associated with these. For example, first, whatever mechanism we device to create and handle threads, that mechanism will have to track

1. thread identifiers, and
2. the relationships between the threads.

The first task to start executing is known as a *base thread* or *base task*. That thread usually has an identifier equal to `0`. Any thread or task that is spawned by another is a *child thread* or *child task*, and the thread or task that spawned that child is the latter's *parent*. As every task can only be spawned by on other task, and there is only one base task, the relationship is clearly hierarchical and may therefore be represented as a tree structure.

The thread-creation mechanism will have to maintain this relationship, thus, each thread will require a record associated with it. For example, we may consider a data structure as follows:

```
typedef size_t tid_t;

typedef struct tcb {
    tid_t thread_id;

    struct tcb *p_sibling;
    struct tcb *p_first_child;

    void *p_call_stack_base;
    size_t call_stack_capacity;

    void *p_return_value;
    bool finished;
} tcb_t;
```



Such a data structure is called a *thread control block* (TCB) as appropriate. We will discuss each of these components:

1. the thread identifier,
2. the tree of child threads,
3. the stack, and
4. the return value, if any.

The routine creating threads would track a table of all threads that are currently executing and other routines may access these entries.

6.4.1 Memory allocation for threads

In many embedded systems, it is often easier to pre-allocate sufficient memory for the thread control blocks (TCBs) corresponding to the maximum number of threads to be run at any one time. Thus, there will be some mechanism

```
// Global variables
tcb_t *p_base_tcb;         // the address of the TCB for the base thread
size_id thread_count = 1;  // the base thread

bool create_thread( tid_t *p_tid, void *(*start_routine)( void * ), void *p_arguments ) {
    bool success;

    if ( thread_count == THREAD_CAPACITY ) {
        success = false;
    } else {
        ++thread_count;

        tcb_t *p_new_tcb = next_available_tcb();

        success = true;
    }

    return success;
}

void exit_thread() {
    // Kill all descendent threads
    //  - how this is done is beyond the scope of this course...

    --thread_count;
}
```

Note: While we may not understand right now, this is potentially unsafe code. In our topic on synchronization and mutual exclusion, we will discuss this in greater detail and describe the options for making this safe.

6.4.2 The thread identifier

The thread identifier is unique for each thread and is a means of identifying the thread from others. The base thread is usually assigned an identifier of `0` and each subsequently created thread is assigned the next largest value.

```
// Global variables
tcb_t *p_base_tcb;              // the address of the TCB for the base thread
size_id thread_count = 1;       // the base thread
tid_t next_available_tid = 1;  // the base thread was tid 0

bool create_thread( tid_t *p_tid, void *(*start_routine)( void * ), void *p_arguments ) {
    bool success;

    if ( thread_count == THREAD_CAPACITY ) {
        success = false;
    } else {
        ++thread_count;

        tcb_t *p_new_tcb = next_available_tcb();

        p_new_tcb->thread_id = *p_tid = next_available_tid;
        ++next_available_tid;

        success = true;
    }

    return success;
}
```

In POSIX, the identifier has type `pthread_t`. On the RTX for the Keil board, the identifier is

```
typedef U32 OS_TID;   // Defined in RTL.h
```

that is, an unsigned 32-bit integer. With a 16-bit integer, this would allow for a maximum of 65536 tasks before looping—a restriction that may be undesirable in an embedded system where sub-tasks may be continually created and destroyed. With 32 bits, this allows for 4.3 billion threads with unique identifiers.

6.4.3 The hierarchy of children threads

Each thread may have many children. The wrong way to implement such a situation would be to have an array of child threads or a separate linked list of threads, as each additional memory allocation will be unnecessarily expensive and costly. Instead, we make an observation:

1. a linked list of children requires only a pointer to the head of the linked list, and
2. the children can be ordered.

Thus, each thread can be assigned a children pointer and a sibling pointer where

1. the children pointer stores the address of the TCB of the first child, and
2. each child stores a pointer to the next child in the list—its sibling.

Each thread would be allocated such a record. For the *base thread*, the parent would be `NULL`, while, for all other threads, this would contain the address of the TCB associated with the thread that created it. As a single thread may have multiple children,

1. the children pointer will store the address of the first child, and
2. the sibling pointer of that child will store the address of the next child.

If there are no children, the children pointer will be `NULL`, and the last child in the list will have its sibling pointer set to `NULL`.

For example, suppose:

1. the base thread `0` first created two child threads `1` and `2`,
2. child `1` created two child threads `3` and `4`,
3. the base thread creates a third child thread `5`, and
4. that child thread creates a child thread `6` of its own,

then the resulting TCBs would look as shown in Figure 6-2

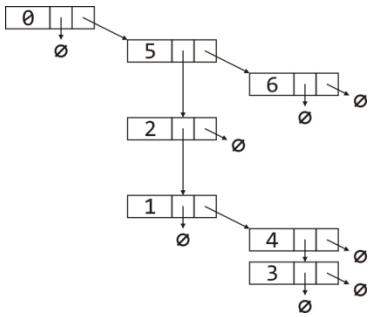


Figure 6-2. The hierarchy of thread control blocks for a base thread and five descendants.

We will require a pointer to the TCB of the currently executing thread. We will then update this appropriately.

```
// Global variables
tcb_t *p_base_tcb;             // the address of the TCB for the base thread
tcb_t *p_running_tcb;          // the address of the TCB for the executing thread
size_id thread_count = 1;      // the base thread
tid_t next_available_tid = 1;  // the base thread was tid 0

bool create_thread( tid_t *p_tid, void *(*start_routine)( void * ), void *p_arguments ) {
    bool success;

    if ( thread_count == THREAD_CAPACITY ) {
        success = false;
    } else {
        ++thread_count;

        tcb_t *p_new_tcb = next_available_tcb();

        p_new_tcb->thread_id = *p_tid = next_available_tid;
        ++next_available_tid;

        // The new thread has no children
        p_new_tcb->p_first_child = NULL;

        // Prepend this new TCB onto the linked list of children
        p_new_tcb->p_sibling = p_running_tcb->p_first_child;
        p_running_tcb->p_first_child = p_new_tcb;

        success = true;
    }

    return success;
}
```

If we wanted to iterate through all the threads, we could use a post-order depth-first traversal using a stacks storing addresses.

```
if ( p_base_tcb->p_first_child != NULL ) {
    stack_t dft;
    stack_init( &dft, MAX_TCB_HEIGHT + 1 );

    stack_push( &dft, p_base_tcb->p_first_child );

    while ( stack_top( &dft )->p_first_child != NULL ) {
        stack_push( &dft, stack_top( &dft )->p_first_child );
    }

    while ( !stack_empty( &dft ) ) {
        tcb_t *p_top = (tcb_t *) stack_top( &dft );
        stack_pop( &tcb_1 );

        if ( p_top->p_sibling != NULL ) {
            stack_push( &tcb_1, p_top->p_sibling );

            while ( stack_top( &dft )->p_first_child != NULL ) {
                stack_push( &dft, stack_top( &dft )->p_first_child );
            }
        }

        // Deal with and access the thread associated
        // With the TCB pointed to by 'p_top'
        //  - do not do anything before manipulating the TCB until after the
        //    stack has been rearranged, as we must still access the TCB
    }

    stack_destroy( &dft );
}

// Deal with 'p_base_tcb' as necessary or appropriate
//  - this may be different from other TCBs
```

This could also be used to iterate through all descendants of a particular thread. Normally, when doing either a depth-first traversal using a stack or a breadth-first traversal using a queue, in the worst case, the capacity of the data structure must be the maximum number of threads, even if the height is significantly less. This could be prohibitively expensive in an embedded system, and the above implementation requires that the memory allocated only equal the maximum height of the thread-hierarchy tree. This is possible as follows:

1. If the base thread has a first child thread, push it onto the stack, and
2. repeatedly push the first child of the current top of the stack onto the stack.
   – The stack contains the depth-first traversal down the left side of the tree.
3. Then, while the stack is not empty:
   a. Get a pointer to the current TCB at the top of the stack, and pop the top of the stack.
   b. If the current TCB has a sibling,
      i. push the sibling on top of the stack, and
      ii. repeatedly push the first child of the top of the stack onto the stack.
      – The stack now continues the depth-first traversal down the left side of the sibling.
   c. Manipulate the current TCB.
4. Finally, deal with the base thread, if necessary. Often, this will be handled separately from its descendants.

It is important to not manipulate the TCB prior unit after the stack has been adjusted (as the modifications could include killing the thread and freeing its TCB), otherwise, if either `p_first_child` or `p_sibling` are changed, the traversal

could become corrupted. To demonstrate how this works, the first image in Figure 6-3 shows the initialization, the second image is when the first node is manipulated, and the third image is the state of the stack while the second node is being manipulated. Colors are used to indicate the various nodes and their location in the stack.

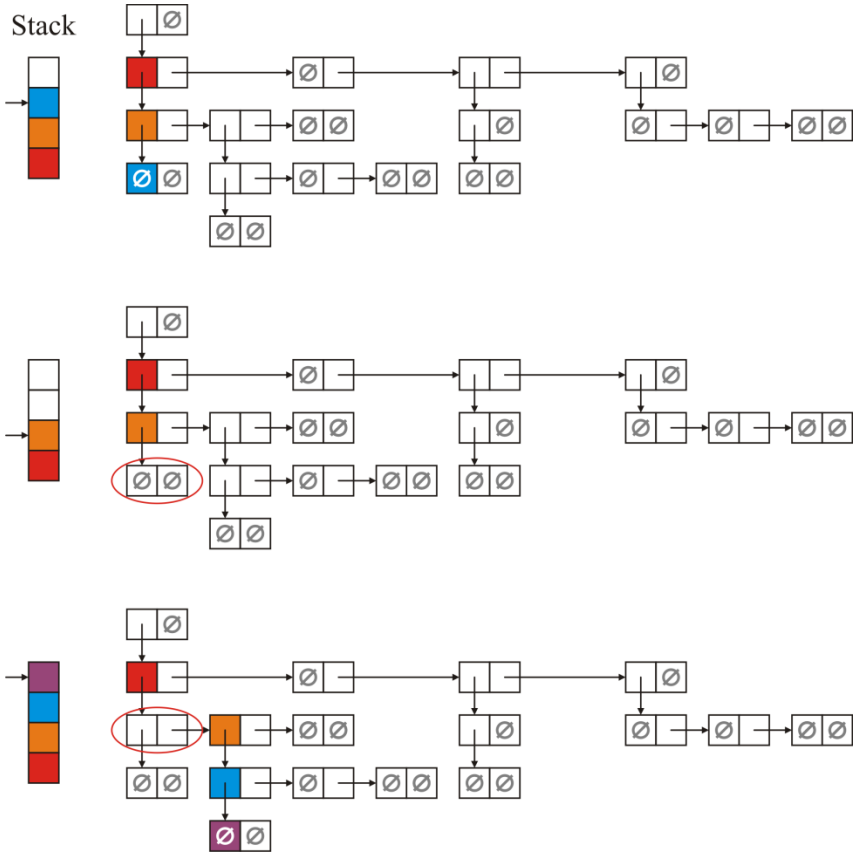


Figure 6-3. The initialization and first two steps of a depth-first post-order traversal of a thread-hierarchy tree. The next node visited would be the purple node in the third image.

This is normally more difficult with a general tree where each node tracks all of its children.

6.4.4 The call stack

The base and the size of the call stack must be assigned by function creating the thread. In an embedded system, there may be a block of memory available for function stacks, or as in the Keil RTX, the user could allocate the memory for the stack and pass it into the thread creation function. As this is secondary to our purposes here, we will assume that there is some mechanism for assigning such stacks.

Recall from our discussion on architectures that an executing task has a call stack. In this case, however, each task must have its own call stack. How can we achieve this?

There are two solutions:

1. Virtual memory is an advanced design we will revisit in Chapter 18, but it is not one that is used in most operating systems with real-time requirements.
2. Fix the size of each of the stacks to a maximum amount.

In Unix, you can either limit the stack size or you can make it *unlimited*—essentially, use as much memory as is accessible, as much as 2^{32} bytes on a 32-bit processor and 2^{64} bytes on a 64-bit processor, assuming of course that you have that much hard drive space (yes, hard drive as virtual memory can swap pages out of main memory onto a hard

drive for temporary storage)—or you can restrict it to a certain amount, and if the stack grows beyond that point, the process is killed.

```
$ limit
cputime      unlimited
filesize     unlimited
datasize     unlimited
stacksize    10240 kbytes
coredumpsize 0 kbytes
memoryuse    unlimited
vmemoryuse   unlimited
descriptors  1024
memorylocked 32 kbytes
maxproc      200
$ limit stacksize 1024
 …
stacksize    1024 kbytes
 …
$ limit stacksize unlimited
…
stacksize    unlimited
 …
```

General-purpose operating systems can achieve unlimited stack sizes through *virtual memory* (a topic that will be discussed later, but also one that is a bane of real-time systems). As for fixing the stack size to a maximum possible amount, there are two approaches here:

1. Statically allocate memory for a fixed number of tasks, where each task is allocated one of the blocks of memory when it is created.
2. Dynamically allocate memory for a task when it is created.

Dynamic allocation allows different tasks to have different stack sizes, and if all references to memory within the stack are relative to the base, it is even possible to dynamically change the stack size at run time. For example, in µVision4, if you edit the file `startup_LPC17xx.s`, you can either search through the file and find the appropriate line:

```
; <h> Stack Configuration
;   <o> Stack Size (in Bytes) <0x0-0xFFFFFFFF:8>
; </h>

Stack_Size      EQU     0x00000200

                AREA    STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE   Stack_Size
__initial_sp
```

or, you may note the *Configuration Wizard* tab at the bottom, as shown in Figure 6-4.

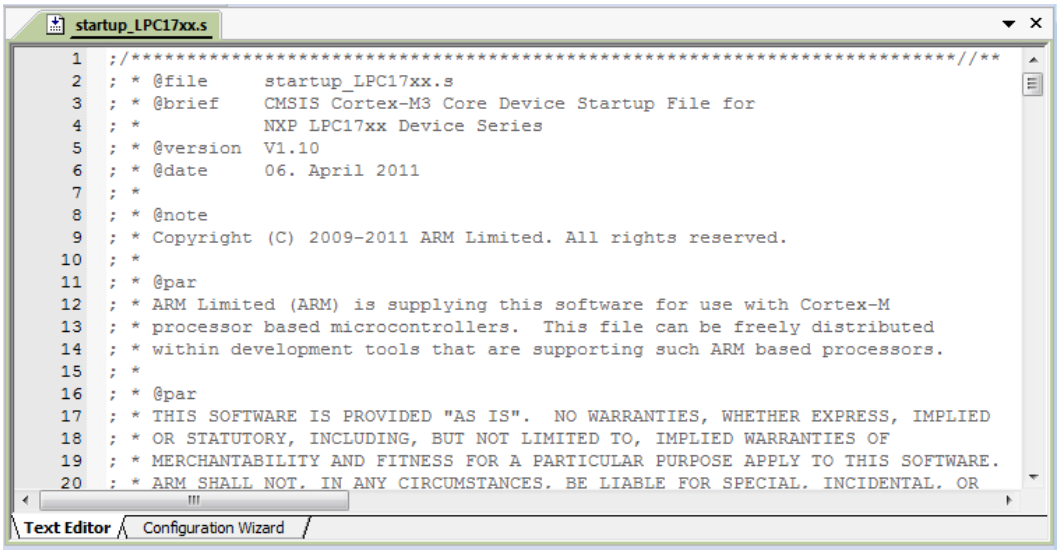


Figure 6-4. Editing `startup_LPC17xx.s` from µVision (ARM Ltd. and ARM Germany GmbH), reproduced here for academic purposes.

This allows you to edit the stack and heap sizes in a more convenient interface.

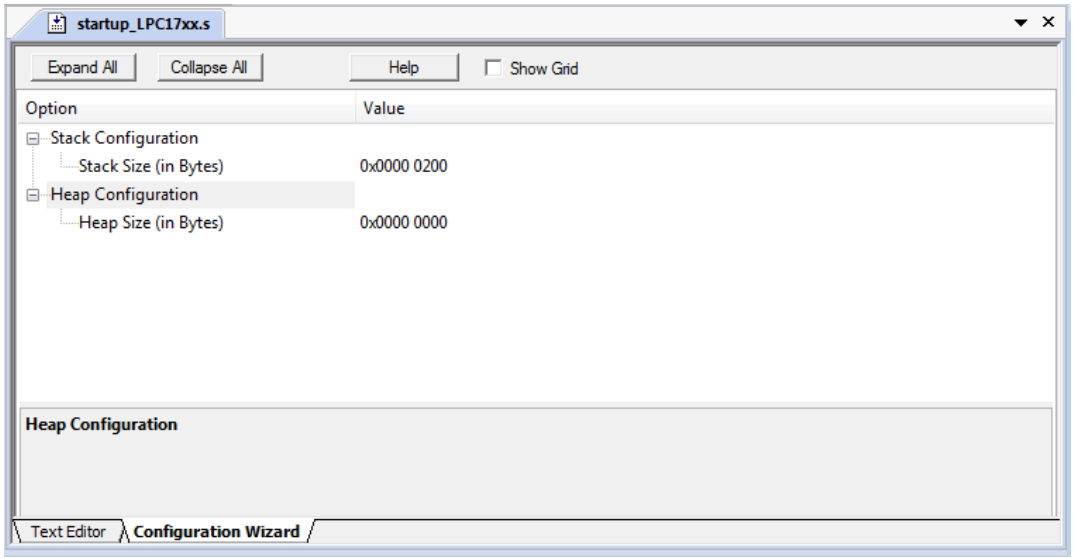


Figure 6-5. The configuration wizard from µVision (ARM Ltd. and ARM Germany GmbH), reproduced here for academic purposes.

Now, each task has, in this case, 200 bytes, and if that amount is exceeded, the tasks is killed. You may note the second line: heap size. As we discussed above, if you want something other than the default, you can also pass in your own stack.

6.4.5 Return values

When a thread wants to return, there must be somewhere to temporarily store that information. For this, we will include two fields:

1. a *finished* flag indicating whether or not the task has exited, and
2. if it has, a pointer to the returned data type.

When the thread exits, these will be set:

```
// Global variables
tcb_t *p_base_tcb;             // the address of the TCB for the base thread
tcb_t *p_running_tcb;          // the address of the TCB for the executing thread
size_id thread_count = 1;      // the base thread
tid_t next_available_tid = 1;  // the base thread was tid 0

bool create_thread( tid_t *p_tid, void *(*start_routine)( void * ), void *p_arguments ) {
    bool success;

    if ( thread_count == THREAD_CAPACITY ) {
        success = false;
    } else {
        ++thread_count;

        tcb_t *p_new_tcb = next_available_tcb();

        p_new_tcb->thread_id = *p_tid = next_available_tid;
        ++next_available_tid;

        // The new thread has no children
        p_new_tcb->p_first_child = NULL;

        // Prepend this new TCB onto the linked list of children
        p_new_tcb->p_sibling = p_running_tcb->p_first_child;
        p_running_tcb->p_first_child = p_new_tcb;

        p_new_tcb->finished = false;

        success = true;
    }

    return true;
}

void exit_thread( void *p_return_value ) {
    --thread_count;

    // Kill all descendent threads
    //  - how this is done is beyond the scope of this course...

    p_running_tcb->p_return_value = p_return_value;
    p_running_tcb->finished = true;
}
```

When the parent thread is ready to get the return value from the child thread, it calls

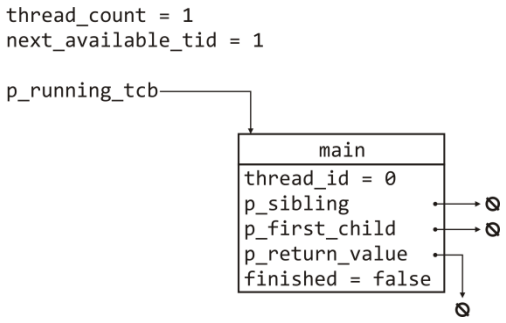
```
void *join_thread( tid_t tid ) {
    void *p_child_return_value;

    tcb_t *p_child_tcb = p_running_tcb->p_first_child;

    while ( p_child_tcb != NULL && p_child_tcb->thread_id != tid ) {
       p_child_tcb = p_child_tcb->p_sibling );
    }

    // The child is not found
    if ( p_child_tcb == NULL ) {
        p_child_return_value = NULL;
    } else {
        // 'p_child_tcb' now stores the address of the appropriate child's TCB
        while ( !( p_child_tcb->finished ) ) {
            // Let the child run
            // How???  Covered in the next topic on scheduling
        }

        // Remove the child from the list of children and
        // deallocate memory for the p_child_tcb

        p_child_return_value = p_child_tcb->p_return_value;

        free_tcb( p_child_tcb );
    }

    return p_child_return_value;
}
```

6.4.6 Visual example

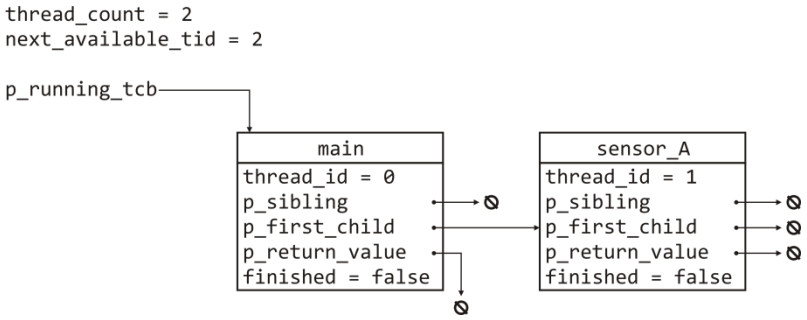
Suppose we execute the following code using our sample instructions

```
void sensor_A( void * );
void sensor_B( void * );

int main( void ) {
    tid_t thread_id_A, thread_id_B;
    create_thread( &( thread_id_A ), sensor_A, NULL );
    create_thread( &( thread_id_B ), sensor_B, NULL );

    // Do something...

    void *return_A = join_thread( thread_id_A );
    void *return_B = join_thread( thread_id_B );

    return EXIT_SUCCESS;
}
```

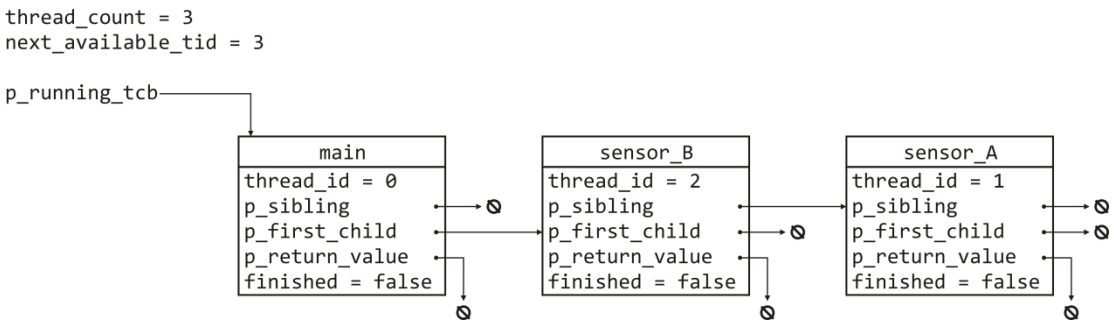
Immediately prior to the creation of the first child thread, the base thread is described in its TCB.



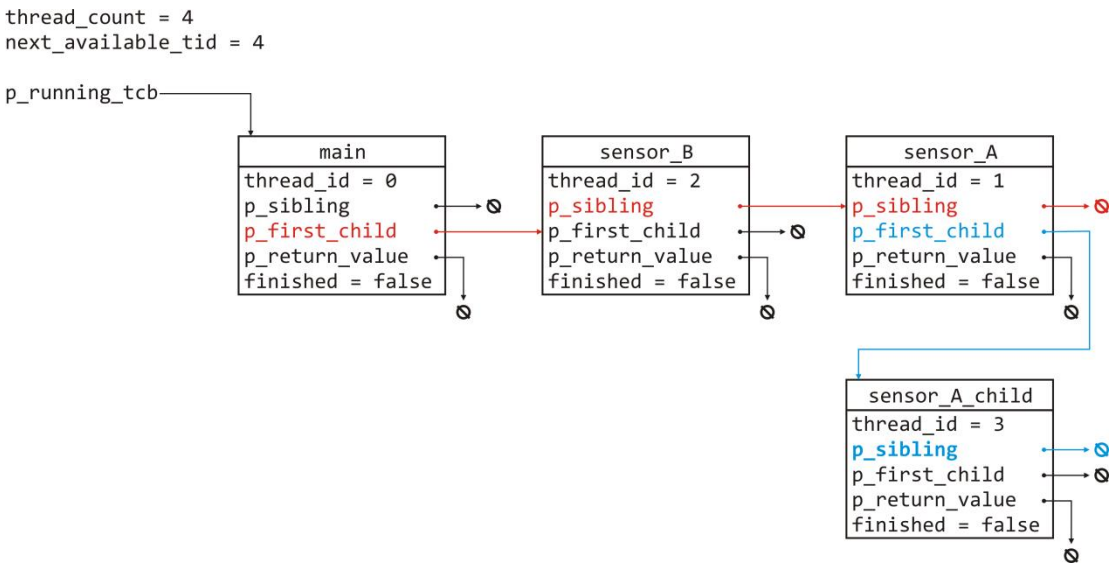
Next, as the first child is created:



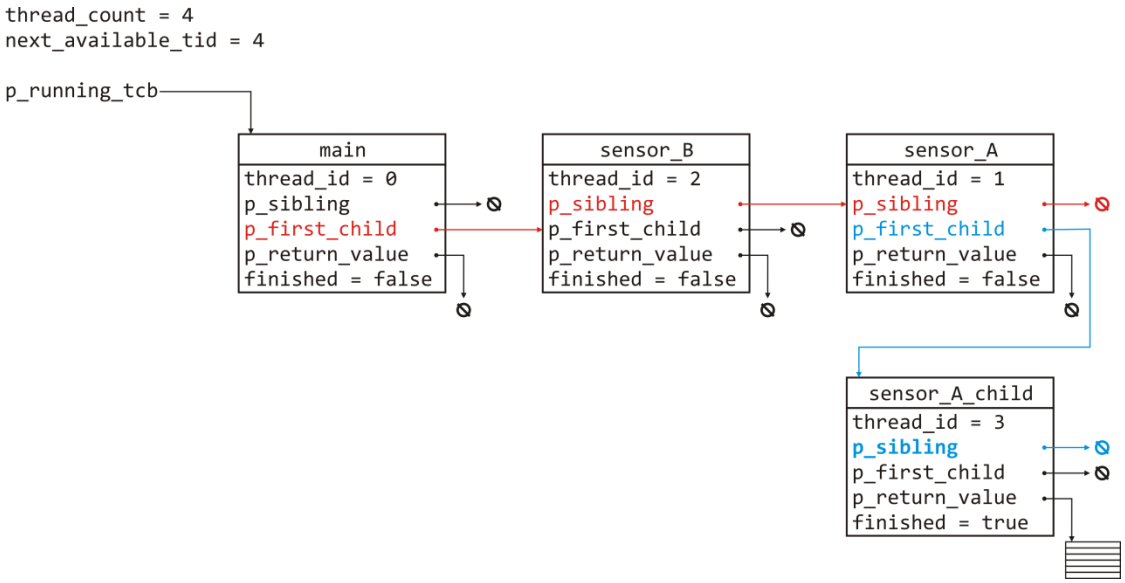
At some point, there may be the opportunity for `sensor_A` to execute. At some point, however, execution will return to the base thread, and the second child will be created. When the second thread is created, it is pre-ended to the linked list:



Suppose that while `sensor_A` is executing, it creates a child thread (let us call it `sensor_A_child`). In this case, its

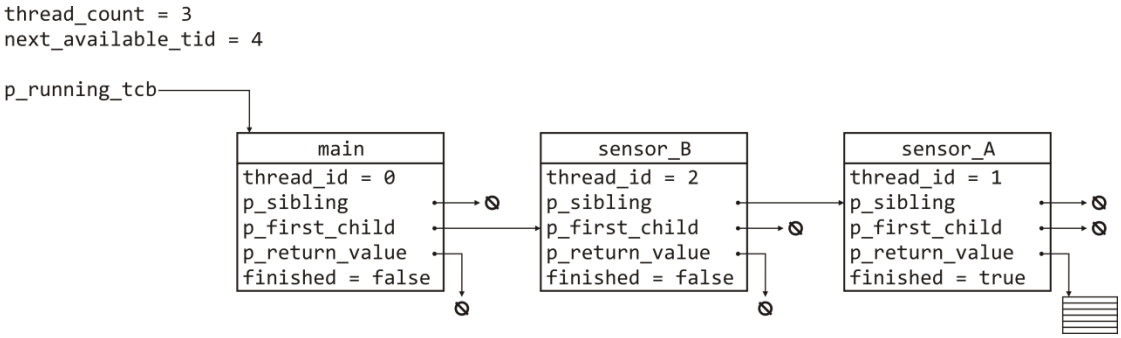


If we assume that the thread `sensor_A_child` exits, it will call `exit_thread( void * )`, passing it a return value, and setting the finished flag to `true`.

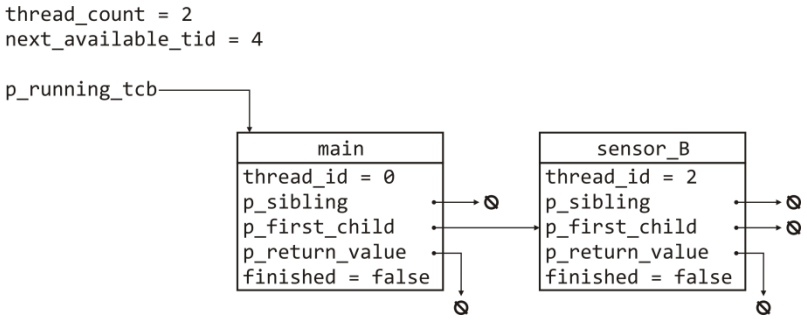


When `sensor_A` calls to join, it will receive that return value, free the memory for TCB block, reduce the thread count, and as it has no more children, set its `p_first_child` to `NULL`.

At this point, all three threads will execute. When threads `sensor_A` and `sensor_B` finish executing, they will call `exit_thread( void * )` and pass it a return value. For example, if `sensor_A` finishes first, it will pass `exit_thread(…)` the address of a data structure with the return value. This will be stored and the `finished` field will be set to true.



When the base thread final calls `join_thread(…)`, that stored return value will be returned from the `join_thread(…)` function and assigned to the corresponding variable. At this point, that TCB would be taken out of the linked list (`p_sibling` for `sensor_B` would be set to `NULL`), the block of memory will be deallocated and the thread count would be reduced to two.



Note that in a real-time system, it is likely that `sensor_A` or `sensor_A` will never return—they will loop indefinitely reading and passing data from the corresponding physical sensor.

6.4.7 Case study: the TCB in the Keil RTX RTOS

The task control block (TCB) must store all information necessary about executing tasks. In the Keil RTX RTOS, the TCB is defined in `rt_TypeDef.h`, reproduced here for academic purposes:

```
typedef struct OS_TCB {
    /* General part: identical for all implementations.                    */
    U8      cb_type;                /* Control Block Type                  */
    U8      state;                  /* Task state                          */
    U8      prio;                   /* Execution priority                  */
    U8      task_id;                /* Task ID value for optimized TCB access  */
    struct OS_TCB *p_lnk;           /* Link pointer for ready/sem. wait list   */
    struct OS_TCB *p_rlnk;          /* Link pointer for sem./mbx lst backwards */
    struct OS_TCB *p_dlnk;          /* Link pointer for delay list             */
    struct OS_TCB *p_blnk;          /* Link pointer for delay list backwards   */
    U16     delta_time;             /* Time until time out                 */
    U16     interval_time;          /* Time interval for periodic waits    */
    U16     events;                 /* Event flags                         */
    U16     waits;                  /* Wait flags                          */
    void    *p_msg;                 /* Direct message passing when task waits  */
    U8      ret_val;                /* Return value upon completion of a wait  */

    /* Hardware dependant part: specific for ARM processor                 */
    U8      full_ctx;               /* Full or reduced context storage     */
    U16     priv_stack;             /* Private stack size, 0= system assigned  */
    U32     tsk_stack;              /* Current task Stack pointer (R13)    */
    U32     *stack;                 /* Pointer to Task Stack memory block      */

    /* Task entry point used for uVision debugger                          */
    FUNCP   ptask;                  /* Task entry address                  */
} *P_TCB;
```

Thus, the type `P_TCB` is a pointer to a TCB. In `RTL.h`, we have the additional definition

```
#define OS_TCB_SIZE     48
```

You will note that the order of the fields was specifically chosen to align with 32-bit words, as shown in Figure 6-6.

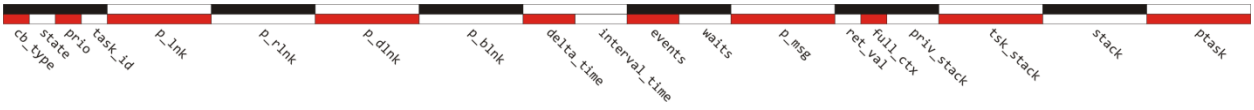


Figure 6-6. The layout of the `OS_TCB`.

As discussed in Topic 2, the compiler will explicitly align the 2-byte and 4-byte fields to line up with the word size. It is quite easy to reorder these fields so that the default memory occupied by this structure is 72 bytes and not 48 bytes. Forcing the compiler use a sub-optimal compact format, accessing fields that spanned a word boundary would require two fetches.

Consequence: When working in embedded systems or at any other time where memory is at a premium, even the order of fields in a structure will have an impact on memory use.

Note that `OS_TID` is 32 bits, but the internal field `task_id` is only eight bits. Thus, we may deduce that while individual tasks are assigned unique identifiers, internally, we can have at most 256 tasks executing at once (the RTX actually restricts you to only 250 active tasks).

6.4.8 Summary of maintaining threads

In this topic, we have considered how we can, at the bare rudiments of how we could maintain a relationship between threads, their children, etc. We will continue to build on this structure in the subsequent topic on scheduling.

6.5 The `volatile` keyword in C

Consider the following code:

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

int global_variable = 0;

void read_global( void * );
void write_global( void * );

int main( void ) {
        pthread_t thread1, thread2;

        pthread_create( &thread1, NULL, (void *) &read_global, NULL );
        pthread_create( &thread2, NULL, (void *) &write_global, NULL );

        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);

        printf( "Exiting main...\n" );
        return EXIT_SUCCESS;
}

void write_global( void *p_void_arg ) {
        int i;

        for ( i = 3; i >= 0; --i ) {
                printf( "%d\n", i );
                sleep( 1 );
        }

        global_variable = 1;
        printf( "Exiting writer...\n" );
}

void read_global( void *p_void_arg ) {
        while ( global_variable == 0 ) {
                // Do nothing
        }

        sleep( 1 );
        printf( "Exiting reader...\n" );
}
```

What this does—and we'll get into thread later—is there is a shared global variable `global_variable`. The two functions `write_global` and `read_global` run in parallel. The first waits 10 seconds and then changes the value of the global variable, the second waits for the global variable to change. When we compile and execute it, it works as expected:

```
$ gcc test.c -lpthread
$ ./a.out
3
2
1
0
Exiting writer...
Exiting reader...
Exiting main...
$
```

No problem, so let's do it again, but this time with some optimizations turned on:

```
$ gcc -O test.c -lpthread
$ ./a.out
3
2
1
0
Exiting writer...
```

and it appears to be hanging—it doesn't exit. Why? The optimizer looked at:

```
while ( global_variable == 0 ) {
    // Do nothing
}
```

and said—nothing in the body of this while loop is changing the value of the variable `global_variable`, so just change it to:

```
if ( global_variable == 0 ) {
    while ( true ) {
        // Do nothing
    }
}
```

After all, nothing in the body of this while loop is changing the value of the variable, so why check it each time? The problem is, the variable is being changed, but not in this loop. We must give the optimizer a hint that the variable `global_variable` may change elsewhere, and we do so by flagging it as *volatile*:

```
volatile int global_variable = 0;
```

Now the code executes as expected. This will become relevant in subsequent chapters when we start looking at communication between tasks and peripheral devices.

6.6 Summary of threads and tasks

In this topic, we have considered the concept of tasks and threads, how to create them, and even some limited form of synchronization, whereby one thread must wait for another to finish. We have considered the issues of parallelization of algorithms and creating divide-and-conquer algorithms through multiple threads, the additional overhead required, maintaining the relationship between the threads and those that they spawn. Finally, we looked at the `volatile` keyword.

Problem set

6.1 In creating a thread or task, this requires:

6. some mechanism of sending back information about the thread or task identifier,
7. options regarding the creation of the thread,
8. arguments to be passed to the thread,
9. memory allocated for a call stack, and
10. the address of the task to be executed.

How are these satisfied in:

1. The POSIX thread library,
2. Java,
3. the Keil RTX RTOS, and
4. the CMSIS-RTOS RTX.

6.2 In class, we have considered two mechanisms to reducing function calls:

1. the use of the `inline` keyword, and
2. function macros.

A simple function such as

```
inline void set_prev_mem( unsigned int offset, unsigned int new_prev ) {
    unsigned int old_prev = get_prev_mem( offset ) << OFFSET_BITS;
    ((mem_block *)(p_mem_base + (offset << MIN_POWER)))->mem_list ^= old_prev;
    ((mem_block *)(p_mem_base + (offset << MIN_POWER)))->mem_list
        |= new_prev << OFFSET_BITS;
}
```

or

```
#define SET_PREV_MEM( offset, new_prev ) {                                    \
    unsigned int old_prev = get_prev_mem( offset ) << OFFSET_BITS;            \
    ((mem_block *)(p_mem_base + (offset << MIN_POWER)))->mem_list ^= old_prev;  \
    ((mem_block *)(p_mem_base + (offset << MIN_POWER)))->mem_list             \
        |= new_prev << OFFSET_BITS;                                           \
}
```

While the second is still prevalent in source code associated with embedded systems, what are the benefits of using the `inline` keyword?

6.3 A task to apply a filter to data normally requires an initial calculation requiring 2 ms followed by a computation that requires 512 ms. How many processors do you require if the task must be completed in 100 ms? How many processors do you require if the task must be completed in 10 ms?

6.4 Why are the fields in the `OS_TSB` structure specifically ordered in the manner they are?

6.5 What are the benefits of maintaining the

6.6 Compare the run-time of `malloc` and `calloc` with respect to the number of bytes n that are required. You may assume that internally they use an $O(\ln(k))$ where k is the number of tasks.

6.7 Your real-time system uses a best-fit memory allocation scheme that runs in $\Theta(n)$ time where n is the number of deallocated blocks. You consider this acceptable, as only one function occasionally deallocates the blocks allocated to it. Over time, however, the run time of the allocation scheme appears to cause deadlines to be missed. What might be a possible cause (there are at least two scenarios you might consider)?

6.8 A real-time system for allowing multiple tasks to run must allow a maximum of N separate tasks to run regardless of any other consideration. How would you deal with the dynamic memory allocation of the task control blocks?

6.9 Create two structures for a thread that is tasked to watch an arbitrary number of sensors that return readings on one value. The sensors are identified by 16-bit unique identifiers and not all sensors may be operational at any time and sensors may be added to or taken out of the network; hence the need to pass the list of identifiers. That task must then perform a number of statistical operations on the data for a specified period of time and return five values: the average, standard deviation, skewness of the data as well as the minimum and maximum values.

6.10 In which component of the compiler is the `volatile` keyword necessary?

6.11 If there is only one task executing, does it make sense to use the `volatile` keyword?

6.12 In Section 6.4.3, we showed how a post-order depth-first traversal could be performed. How could you include a pre-order traversal where information about ancestors is collected prior to traversing the descendants and then passed down.

7 Scheduling

Up to now, we've discussed different tasks being performed, and we've assumed that they are operating independently on separate cores or processors: each task is executed independently of the other and, in our example, each parent task waited for its children to complete, at which point, it could carry on. We will, however, answer a few questions:

1. What does the parent merge sort thread do as it waits for its child to finish?
2. In general, what does the system do if a thread is waiting for input or a communication?

In a real-time system, when there are many tasks that could potentially execute simultaneously, it is often critical that one specific task should be executing at a point in time. For example, if a drone is attempting to execute a complex manoeuvre that may result in the drone crashing if the control of the rotors is not carefully maintained, the tasks involved with secondary duties such as downloading data should not be monopolizing the processors. Scheduling is primarily a question of safety: ensuring that the tasks that should be running are the ones that are currently executing on the processors. Additionally, intelligent designs of scheduling algorithms ensure that performance requirements can be met.

Thus, we will begin by reviewing the issues of having tasks wait on others, followed by an introduction to multitasking. We will then consider the more efficient non-preemptive scheduling algorithms followed by an examination of the less efficient but significantly more adaptable non-preemptive scheduling algorithms. We will conclude by considering some of the many issues associated with scheduling real-time systems.

7.1 Background: waiting on tasks and resources

The merge sort routine, while waiting for its child to complete, could go into a busy-waiting loop (the parent is just staring intently at the child waiting for it to finish). This could be a waste of the processor, but if nothing else is happening, why not? Consider waiting for a block of memory stored on a hard drive to be loaded into main memory. This takes about 10 ms, during which time a 3 GHz processor could execute 30 million instructions. Again, busy-waiting would seem to be sub-optimal. Worse, consider a program such as a word processor or an editor, waiting for a user to strike a key. With 100 % processor utilization with no useful work, this is now going beyond absurd.

Some programs may not require significant input or output, apart from reading data during the initialization and writing the results at the end. If the limiting resource for a program is the speed of the processor, the program is said to be *processor bound*, or *CPU bound*. For example, approximating solutions to initial-value problems is likely to be processor bound.

In other cases, a task may require constant and significant access to main memory. Consider a program designed to simulate the behavior of the AMD G3 processor. This involves solving a system of a million-and-a-half equations in an equal number of unknowns. Even though the representing matrix is sparse, this still requires over 36 MiB for the coefficients of the matrix, and each entry will be accessed with each step of the simulation. A task where the limiting resource is access to memory is said to be *memory bound*.

Many tasks, however, require significant access to additional resources. If the dominant factor in limiting the execution of a task is any other resource other than the processor or access to main memory, the task is said to be *I/O bound*. This other resource could be a user, or it could be another device connected to the processor.

Summary: A task is said to be

1. *processor bound* or *CPU bound* if there is an inverse relationship between processor speed and computation time,
2. *memory bound* if the limiting factor is accessing data in main memory, and
3. *I/O bound* if the limiting factor is the response of resources other than the processor and main memory (waiting for a user, another device, etc.)

7.2 Introduction to multitasking

Of course, you are familiar with how a computer can run multiple tasks simultaneously: you have your web browser open, while you are editing a document, all while you're doing what you really want to do, and that is to play *Quake Live*. At the same time, you can run an IVP solver, a processor-bound program, that takes a second to run. You have obviously got more tasks in the system than there are physical or logical processors. Yet, although you have so many tasks running,, no task freezes up, nor do you have to explicitly program in switches to other programs.

By 1961, the developers of the LEO III business computer recognized that if one executing program is waiting for a peripheral device (for example, input or output), it should be possible to start executing a different program. Suppose that Thread A is executing

```
// Executing...
var = get_sensor_value();
// Process the result...
```

Suppose that `get_sensor_value()` is called, but the sensor is not yet ready to transmit the information. We could wait, but that wastes processor time. Instead, we could start executing a second thread, Thread B. The issue here is that at some point, we will want to continue executing Thread A, so, in order to do this, we must store the state of the processor:

```
get_sensor_value() {
    while ( !sensor_1_ready() ) {
        // Save the state of the processor as it is right now
        //  - this includes all registers, including the PC
        scheduler();
        // Continue executing...
    }

    return access_sensor( ... );
}
```

Having done so, we can now launch a second thread. We will say that the currently executing thread is being *put to sleep* and the second thread that we are scheduling is being *woken up*. However, what does this *scheduling* function do? The scheduler requires at least two pieces of information:

1. what are the other threads that could be scheduled, and
2. what was the state of the thread when it was put to sleep (that is, if it has run at all)?

We will look at both of these next.

7.2.1 What can be scheduled?

If we are simply allowing threads to continue executing until either they finish or until they make a request that cannot be immediately satisfied (accessing a busy sensor, waiting for a communication system to become available, or retrieving information from a hard drive—a request that can take hundreds of thousands of processor cycles), and we are only interested in throughput (that is, getting the maximum use of the processor), then all we really need is a linked list of threads:

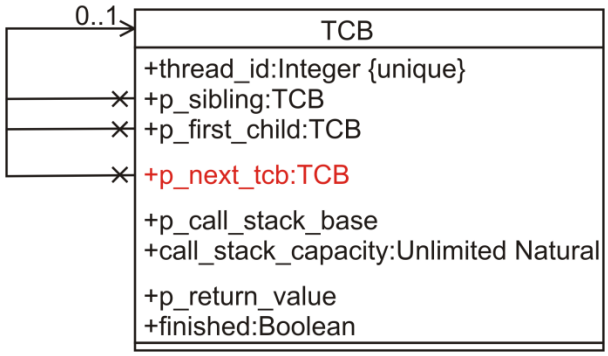
```
tcb_t *p_ready_queue_head;
tcb_t *p_ready_queue_tail;
```

Inside the system initialization, we would initialize the base thread TCB and set:

```
p_ready_queue_head = p_base_tcb;
p_ready_queue_tail = p_base_tcb;

scheduler();
```

Now, in order to generate a linked list, we require a next pointer. We can do this by adding another field to the TCB:

```
typedef struct tcb {
    tid_t thread_id;

    struct tcb *p_sibling;
    struct tcb *p_first_child;

    struct tcb *p_next_tcb;

    void *p_call_stack_base;
    size_t call_stack_capacity;

    void *p_return_value;
    bool finished;
} tcb_t;
```



You will notice that the fields allow for the same thread to be contained in numerous linked lists, all associated with different purposes:

1. one linked list of siblings, and
2. another list of threads waiting for the processor.

Now, each time a thread is created, that thread can be appended to the end of the linked list:

```
bool create_thread( tid_t *p_tid, void *(*start_routine)( void * ), void *p_arguments ) {
    bool success;

    if ( thread_count == THREAD_CAPACITY ) {
        success = false;
    } else {
        ++thread_count;

        tcb_t *p_new_tcb = next_available_tcb();

        p_new_tcb->thread_id = *p_tid = next_available_tid;
        ++next_available_tid;

        // The new thread has no children
        p_new_tcb->p_first_child = NULL;

        // Prepend this new TCB onto the linked list of children
        p_new_tcb->p_sibling = p_running_tcb->p_first_child;
        p_running_tcb->p_first_child = p_new_tcb;

        p_new_tcb->finished = false;

        // Place the argument onto the call stack in the right location
        insert_argument( p_new_tcb->stack, p_arguments );
        p_new_tcb->PC = start_routine;

        // Append the thread to the end of the list of ready threads
        p_new_tcb->p_next_tcb = NULL;
        p_ready_queue_tail->p_next_tcb = p_new_tcb;
        p_ready_queue_tail = p_new_tcb;

        // Now return to the calling thread
        // - the new thread will be executed when the processor becomes available

        success = true;
    }

    return success;
}
```

Incidentally, the TCB for the Keil RTX, reproduced here for academic purposes, contains such a link:

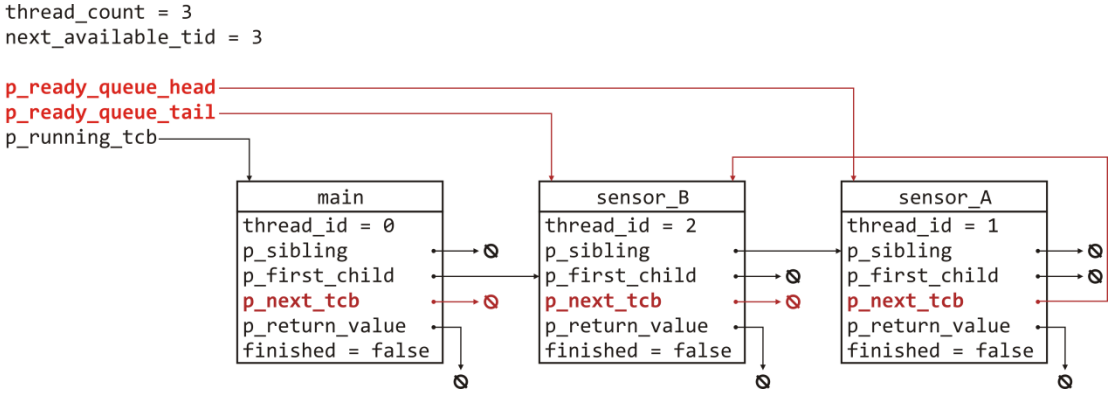
```
typedef struct OS_TCB {
    /* General part: identical for all implementations.                      */
    U8     cb_type;                /* Control Block Type                     */
    U8     state;                  /* Task state                             */
    U8     prio;                   /* Execution priority                     */
    U8     task_id;                /* Task ID value for optimized TCB access */
    struct OS_TCB *p_lnk;          /* Link pointer for ready/sem. wait list  */
    struct OS_TCB *p_rlnk;         /* Link pointer for sem./mbx lst backwards */
    struct OS_TCB *p_dlnk;         /* Link pointer for delay list            */
    struct OS_TCB *p_blnk;         /* Link pointer for delay list backwards  */
    // ...
} *P_TCB;
```

7.2.2 Visual example

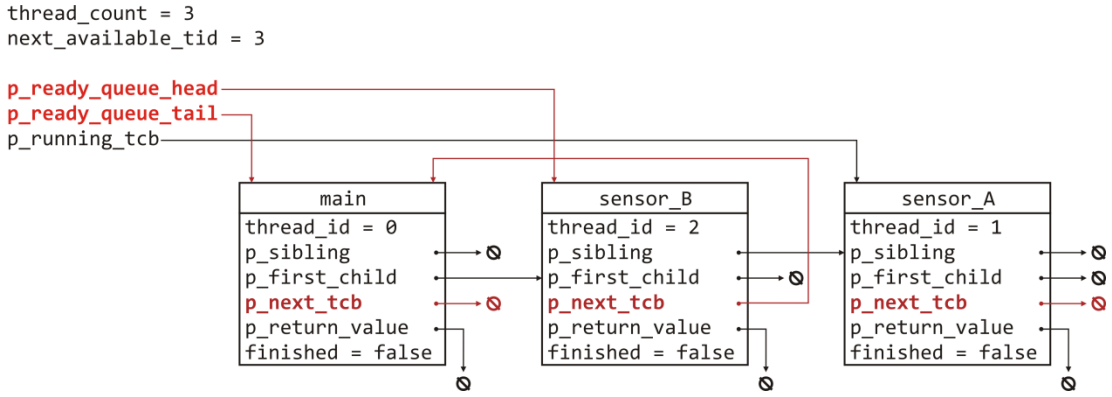
Let us take the same sample code from Section 6.4.6.

```
void sensor_A( void * );
void sensor_B( void * );

int main( void ) {
    tid_t thread_id_A, thread_id_B;
    create_thread( &( thread_id_A ), sensor_A, NULL );
    create_thread( &( thread_id_B ), sensor_B, NULL );

    // Do something...

    void *return_A = join_thread( thread_id_A );
    void *return_B = join_thread( thread_id_B );

    return EXIT_SUCCESS;
}
```

Let us suppose that the base thread successfully executed the creation of both threads and continued to run. In this case, it would remain the running thread, but there would now be two items on the ready queue. As sensor_A was created first, it would be at the head of the ready queue.



Suppose now that there comes a time for another thread to be executed. In this case, the currently running thread would be pushed onto the back of the ready queue and the thread at the head of the ready queue would start executing.



If now there is another opportunity to change which thread is executing, `sensor_A` would be put back on the end of the queue, and `sensor_B` would begin executing. In this manner, it is always possible to determine which task is currently executing, and which tasks are waiting to execute.

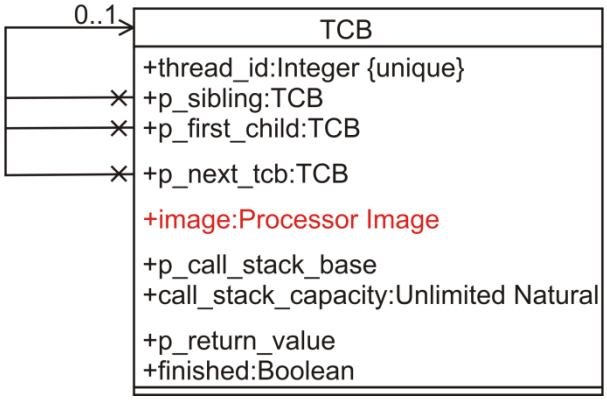
7.2.3 Storing an image of the processor—saving the register values

One important thing to recognize at this point is that when the scheduler is called, it will pick whichever thread is at the front of the linked list; however, when the thread that is currently being put to sleep, it will be woken up again at some point in the future, but all the registers must have all the exact same values as at the moment the thread was put to sleep. Therefore, the scheduler must also store the state of the processor when the task switch occurs, and it will do so through a sequence of assembly language instructions that will store the data structure to memory. For this, we need someplace to store this. For example, if this was C, we could create a structure;

```
typedef struct {
    uint32_t R0, R1, R2, R3, …, R14;
    uint32_t PC;
    unit32_t PSR, PRIMASK, FAULTMASK, BASEPRI, CONTROL;
} processor_image_t;
```

Thus, each TCB would have another field for this information:

```
typedef struct tcb {
    tid_t thread_id;

    struct tcb *p_sibling;
    struct tcb *p_first_child;

    struct tcb *p_next_tcb;
    processor_image_t image;

    void *p_call_stack_base;
    size_t call_stack_capacity;

    void *p_return_value;
    bool finished;
} tcb_t;
```



Now, let's look back at the call we made:

```
get_sensor_value() {
    while ( !sensor_1_ready() ) {
        // Save the state of the processor as it is right now
        //  - this includes all registers, including the PC
        scheduler();
        // Continue executing... *
    }

    return access_sensor( ... );
}
```

Notice that we should be able to carry on executing after the function call. In fact, the call to the scheduler is just one more function call—one that we can essentially *undo*.

Thus, when we call the scheduler, we can set it up so that as soon as we start the program counter, it is as if the call to scheduler had just returned:

```
// Copy registers Rk to p_running_tcb->image.Rk for k = 0, ..., 12
// Modify the other registers R13 (SP), R14 (LR) and R15 (PC)
//  - we can now make use of R0 through R12 to make these computations
// Copy the special registers
```

Now, when we restore this state, the last thing we will do is set the PC, in which case, it will continue executing the code above at the point marked by a star *. Now we must

```
void scheduler() {
    // Copy registers Rk to p_running_tcb->image.Rk for k = 0, ..., 12
    // Modify the other registers R13 (SP), R14 (LR) and R15 (PC)
    //  - we can now make use of R0 through R12 to make these computations
    // Copy the special registers

    if ( p_ready_queue_head == NULL ) {
        // Restore only those registers that were used to execute the
        // condition of this if statement
        // Set the PC to p_running_tcb->image.PC
        //  - you don't need a return statement--this automatically "returns"
    }

    // Push the current thread onto the queue
    p_ready_queue_tail->p_next_tcb = p_running_tcb;
    p_ready_queue_tail = p_running_tcb;

    // Pop the front thread off of the queue
    p_running_tcb = p_ready_queue_head;
    p_ready_queue_head = p_ready_queue_head->p_next_tcb;

    // Context switch...
    // Copy the information from p_running_tcb->image back to the registers,
    //  - Be sure to set the program counter last.
}
```

At this point, we will simply continue cycling through the available threads. When the thread that was executing

```
get_sensor_value() {
    while ( !sensor_1_ready() ) {
        // Save the state of the processor as it is right now
        //  - this includes all registers, including the PC
        scheduler();
        // Continue executing... *
    }

    return access_sensor( ... );
}
```

is loaded again, it will continue executing and check if the sensor is ready yet. If it is, it will access the sensor, otherwise it will—once again—call the scheduler.

7.2.4 Description of tasks

In the previous section, we focused on threads. However, now that we are focusing on real-time algorithms, in general, each thread will be designed to accomplish some form of task. Consequently, we will use a change in terminology to emphasize a change in paradigm. We will describe tasks by

1. how often they run, and
2. how much time they require when they are running.

We will first, however, begin by classifying the tasks we are to schedule.

7.2.4.1 Classification of tasks

We will divide tasks into one of four categories:

1. fixed-instance,
2. periodic,
3. aperiodic, and
4. sporadic

tasks. We will discuss each of these here.

7.2.4.1.1 Fixed-instance tasks

A fixed-instance task executes a fixed number of times (usually just once), often for initialization or clean-up purposes. We will disregard fixed-instance tasks in most of our analysis, as we will be considering the steady state of a real-time system.

This does not mean to suggest such tasks should be ignored; however, either they meet their deadlines or they do not. If they do not, the code must be re-examined to determine what can be sped up to satisfy, for example, the initialization deadline.

> One example relayed to me from an engineering student was during their co-op placement, the boot sequence for a particular real-time system installed on a transit system took five minutes, whereas the original system took only seconds. This was an irritation for the staff, but once in a while, the system would fail while the system was in use and this would lead to frustrations on the part of passengers, as well, thereby degrading the quality of service.

7.2.4.1.2 Periodic tasks

Periodic tasks are those that must be run with a fixed cycle. Real-time embedded systems often involve periodic tasks due to the prevalence of control applications requiring cyclic operations: tasks which must be run periodically for the system to function; for example, in streaming video requires numerous tasks to be performed, say, producing a frame of output twenty-four times per second. A complex system could have tasks that are hard, firm or soft. How can we tell

whether or not periodic tasks that are designed into the system can be scheduled? Usually we consider period tasks as a pair (c_k, τ_k) where τ_k is the period and c_k is the worst-case computation time during that period (a task may, for example, check a sensor and, in general, do nothing unless a value it has read has exceeds a critical threshold). In a real-time system, it is necessary that each task runs during its period, and therefore we will consider the end of each period to be a deadline.

If all the tasks have the same period, it is easy to determine as to whether or not the periodic tasks will normally overload the system, but what happens if different tasks have different periods? In this case, we can calculate the processor utilization,

$$U = \sum_{k=1}^{n} \frac{c_k}{\tau_k} .$$

This represents the long-term average processor utilization. If $U > 1$, the system is *overloaded*; that is, the processor cannot guarantee that it will be able execute all the tasks in such a way to satisfy all deadlines. If $U \leq 1$, we will have at least two scheduling algorithm that will schedule the tasks in such a way so as to meet all their deadlines (see Section 7.2.5.3). As an example, consider the three periodic tasks (2, 6), (1, 4), (4, 12). We could schedule them as shown in Figure 3-8.

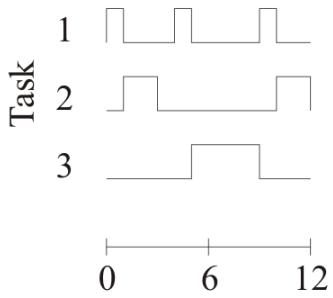


Figure 7-1. The scheduling of three tasks.

This is only one of many possible schedules for these three tasks. Note that there is a period during the 4th millisecond (we'll assume milliseconds) where no tasks are scheduled to execute. If a sporadic task occurred, it could be executed during this period of time; otherwise, the idle task will fill that time.

Note: We will consider this beyond the scope of this course, but in some cases, the deadline will be before the end of the period. For example, if a task is required to fetch data from a sensor, and that data must be available to other tasks, getting that data right at the end of a period may be futile, and therefore, such a task may be described by a triplet (c_k, d_k, τ_k) where d_k is width of the time interval at the start of the period during which the task must complete its execution. If you take ECE 455 *Embedded software*, you will see this generalization.

7.2.4.1.3 Aperiodic tasks

Aperiodic[14] tasks are tasks that respond to events that occur at irregular intervals. There is no minimum time interval between two events that could require the same response, and therefore it very difficult to require such responses to be hard real-time—there is no guarantee that in any timer interval, too many of such events would overload the system. Such tasks are usually responding to events signaled by either *interrupts* external to the microcontroller. Such tasks must be scheduled in such a way as to ensure that any soft real-time aperiodic task should never cause a firm or hard real-time task to miss its deadline, and any firm real-time aperiodic task should never cause a hard real-time task to miss its deadline.

[14] aperiodic, *adj*, Not periodic; without regular recurrence. From the Oxford English Dictionary.

An example of an aperiodic task is one that responds to requests in a client-server model. You may expect an arrival rate of 3 requests per second, there is still a 1.2 % chance that eight or more requests appear in a single second. It would be very difficult to respond to such requests in a hard real-time fashion, and if the system is firm real-time, occasionally, some requests may simply be denied.

7.2.4.1.4 Sporadic tasks

Sporadic[15] tasks are aperiodic tasks that usually require a firm or hard real-time response. For a system to be able to respond to such events, they be described by a pair (c_k, τ_k) where τ_k is the minimum inter-arrival time (the minimum interval between such events) and c_k is the worst-case computation time required to respond to the event. Sporadic tasks, by their nature, may *overload* the system: that is, it is not possible to schedule all the tasks so that all meet their deadlines; however, it must be decided which firm and hard real-time tasks will miss their deadlines. If a sporadic task overloads the system, either it will not be accepted, or it may require that a currently executing task will miss its deadline. If the tasks that will miss their deadline are firm or hard real-time, they will not even be scheduled.

An example of a sporadic task is on that responds to an alert sensor that monitors the temperature of a target once per 100 ms. Because the real-time clocks on the sensor and the microprocessor may drift, it is not guaranteed that the clocks will be perfectly synchronized; however, if the sensor sends an alert that the temperature has exceeded a critical value, it will not read the temperature again for another 100 ms, time during which the real-time system can respond.

When a sporadic task is ready to run, it will be treated as if it is a periodic task.

Note: Again, beyond the scope of this class, as with periodic tasks, the deadline may occur prior to the end of the minimum inter-arrival time. For example, with our example of a sensor checking a temperature, if it is found the temperature is outside the required interval, the response may have to be quicker than simply as often as the sensor is sampling the temperature. Again, we will then describe that sensor as the triplet (c_k, d_k, τ_k) where d_k is width of the time interval at the start of the interval during which the task must complete its execution. If you take ECE 455 *Embedded software*, you will see this generalization.

7.2.4.1.5 Summary of task classification

We will describe tasks as either fixed-instance, periodic or non-periodic. In the latter case, aperiodic tasks can occur at any time, and are therefore usually require soft real-time responses, while sporadic tasks require firm or hard real-time responses, and to deal with these, it is necessary that there is some minimum time interval between such events.

[15] sporadic, *adj*, appearing, happening, etc., now and again or at intervals; occasional. From the Oxford English Dictionary.

7.2.4.2 Estimating worst-case execution times

Before we can determine whether or not a scheduling algorithm will allow all tasks to satisfy their deadlines (be they periodic or sporadic), we must be aware of the execution time. In order to determine whether a deadline will be met or missed, we must estimate how long a task may run. Most tasks do not exhibit uniform run times. In the majority of cases, where a task is, for example, inspecting an environmental condition, may simply record the data; however, occasionally, the task may have to react to a situation that has been observed. Thus, we must estimate for each task the worst-case execution time (WCET) for each task and determine whether or not all deadlines can still be met under such circumstances. We will represent this worst-case execution (or *computation*) time by c. There are two techniques in order to estimate the WCET, including

1. an analysis of the source code, and
2. estimation from empirical evidence.

Incidentally, the word "empirical" comes from the ancient Greek word for "experience", namely, *εμπειρία* or *empeiria*.

A significant amount of research has gone into estimating the run-time of an algorithm through an examination of the source code; however, this tends to over-estimate the run time. For example, most compilers will involve some form of optimization and most processors implement pipelining where, under certain circumstances (independence), instructions can be run in parallel. The consequence of over-estimating the worst-case execution times is that the microcontroller designated for the system will be too powerful, and therefore cost too much and may consequently use more power, thereby degrading the lifetime of, for example, the battery.

By executing the tasks under simulated conditions, one must ensure the model is a reasonable approximation of reality; otherwise, the observations made under testing conditions may vary greatly (and usually underestimate) the actual performance in a working system. In order to do this, however, you must have an understanding of statistics.

Suppose, for example, we ran a hundred tests and we plotted a histogram with a bin width of 1 ms, as is shown in Figure 7-2.

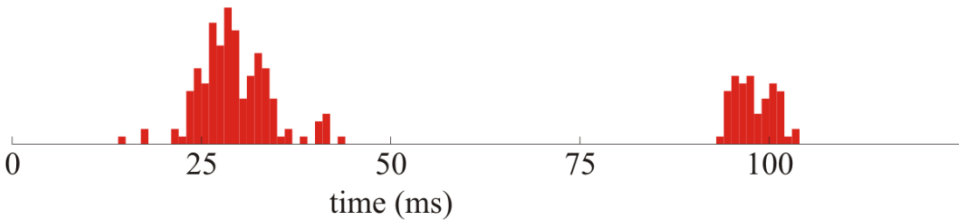


Figure 7-2. A histogram of execution times of a task.

On the left-hand side, we may use a single normal distribution to estimate the worst case execution time, as is shown in Figure 7-3. This suggests it may be reasonable to use a normal distribution.

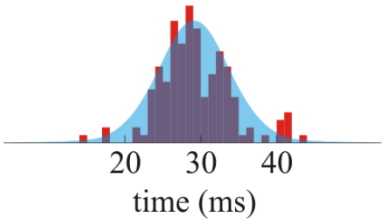


Figure 7-3. A normal distribution superimposed over the samples.

We could, for example, calculate the mean and sample standard deviation:

$$\bar{x} = \hat{\mu} = \frac{1}{n}\sum_{k=1}^{n} x_k$$
$$s = \sqrt{\frac{1}{n-1}\sum_{k=1}^{n}\left(x_k - \hat{\mu}\right)^2}$$

and then determine whether or not we want 99 % confidence ($\hat{\mu} + 2.326s$), 99.9 % confidence ($\hat{\mu} + 3.090s$), or 99.99 % confidence ($\hat{\mu} + 3.719s$) on estimating the worst-case execution time. For further details on the coefficients, look up *one-sided confidence intervals*.

Such a blunt approach may, however, overestimate the worst-case run time of the average behaviour of this task. Instead, there appear to be three *humps*, in which case, we may be observing three essentially different paths if computation, as shown in Figure 7-4.

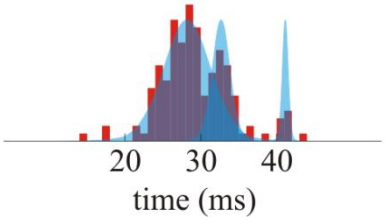


Figure 7-4. The distribution as three distributions.

In this case, we may have to only consider the worst-case of the last *hump*, which may not be as extreme as when we consider all three in aggregate.

The other data, however, does not even appear to be bi-normal. Instead, it might be better to approximate that with a uniform distribution.

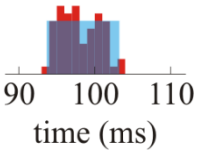


Figure 7-5. An apparently uniform distribution of run-times.

In this case, the estimation of the lower and upper bounds is more complex. First, we must estimate the two end-points of the uniform distribution on [*a*, *b*], so given the *n* points $t_1, \ldots, t_n$, we can estimate the endpoints with:

$$\hat{a} = \min\{t_1,\ldots,t_n\} - \frac{\max\{t_1,\ldots,t_n\} - \min\{t_1,\ldots,t_n\}}{n-1} \text{ and}$$
$$\hat{b} = \max\{t_1,\ldots,t_n\} + \frac{\max\{t_1,\ldots,t_n\} - \min\{t_1,\ldots,t_n\}}{n-1}.$$

Second, these have standard deviations of

$$s = \sqrt{\frac{n\left(\hat{b}-\hat{a}\right)}{\left(n^2-1\right)\left(n+2\right)}};$$

therefore, like before, we may find estimators of the upper bond with 99 % confidence ($\hat{a} - 2.326s$ and $\hat{b} + 2.326s$), 99.9 % confidence ($\hat{a} - 3.090s$ and $\hat{b} + 3.090s$), or 99.99 % confidence ($\hat{a} - 3.719s$ and $\hat{b} + 3.719s$). For example, the data

213.011, 215.912, 211.197, 210.2126, 215.779, 211.237, 209.900, 207.724
214.327, 212.309, 214.816, 210.6839, 216.169, 216.160, 217.415, 211.525

comes from a uniform distribution on [204.71, 217.83]. The estimators of these end points are $\hat{a} = 204.4666$ and $\hat{b} = 218.1576$ with a standard deviation of s = 0.2098, and therefore 99 % confidence intervals of these two parameters are [–4.9545, –3.9787] and [7.6697, 8.6455].

Important: You are not being shown this with the expectation that you will memorize this. Instead, this is demonstrating that there are valid statistical mechanisms for estimating such values—don't just guess.

Other factors can significantly affect the run time, including caches and virtual memory. These two topics will be ignored at this point, but we will examine these at the end of the course.

7.2.4.3 Periodic and sporadic tasks with additional deadlines

Both periodic and sporadic tasks may be described by a pair (c_k, τ_k), being the worst-case computation time and either the period or the minimum inter- arrival time. In many cases, this is sufficient to describe such tasks: so long as the tasks complete execution within τ_k time, the tasks have met their deadline. In some cases, however, a task may have an additional deadline prior to the end of the period.

Suppose a real-time task is periodic τ_k, but within each period, there is a deadline d_k, prior to which it must complete its computation. For example, it may occur that a task completes at the end of its period, but is then immediately scheduled at the start of the next period, as shown in Figure 7-6.

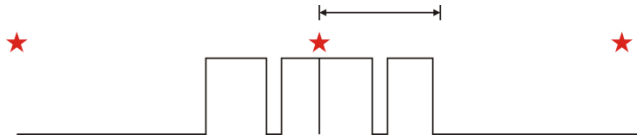


Figure 7-6. A close-to minimum separation between completion times for a task scheduled under RM scheduling.

Thus, the minimum inter-completion time may be as small as c_k. If the result of the computation is begin processed by an independent system, the time c_k may be insufficient to process that information before the next result arrives. Consequently, it may be necessary to impose additional deadlines d_k where $c_k \le d_k \le \tau_k$, where if $d_k = \tau_k$ for each task, we are reduced to RM scheduling.

To show this, suppose that with the task shown in Figure 7-6, the deadline was $d_k = \frac{3}{4}\tau_k$. In this case, the worst-case inter-completion time would be significantly longer, as shown in Figure 7-7.

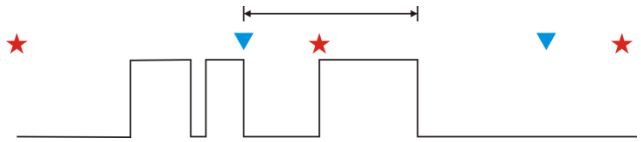


Figure 7-7. The worst-case inter-completion time for a task with a deadline marked with inverted triangles.

More generally, the inter-completion time is now $c_k - d_k + \tau_k$ as opposed to just c_k. We will represent a periodic task with a deadline by the triplet (c_k, d_k, τ_k).

Similarly, a sporadic task may be known to have an inter-arrival time of no less than τ_k, but the system must respond to that event significantly before the next event. For example, a warning sensor set to signal the system if the temperature exceeds a critical value may have be designed to not signal more often than once every 100 ms, but once the signal is received by the real-time system, the task responding to that event must signal the corresponding actuators within 15 ms. For this sporadic event, $\tau_k = 100$ ms but $d_k = 15$ ms.

When we consider scheduling algorithms, we will consider the simpler case where $d_k = \tau_k$ for all tasks, and then we will look at scheduling algorithms where the deadline occurs prior to the end of the period, $d_k < \tau_k$.

7.2.4.4 Example: Periodic garbage collection

Previously, we discussed how garbage collection in a real-time system was not realistic given an algorithm such as mark-and-sweep. A task on track to meet a deadline may subsequently miss that deadline if, during a request for memory, the garbage collection routine is called as there is insufficient memory to satisfy a given request for memory. As the mark-and-sweep algorithm performs a graph traversal followed by a traversal of all allocated memory, it is difficult to determine an upper bound on the run time—especially in a complex system (the only system where garbage collection makes much sense, anyway).

One promising approach is to treat the garbage collector like any other periodic task. The garbage collector is a task with a period τ_{GC} and worst-case execution time c_{GC}, and it will be scheduled like any other periodic task. Throughout this topic, we will investigate the *Metronome* garbage collection algorithm.

D.F. Bacon, P. Cheng and V.T. Rajan. *The Metronome: A simpler approach to garbage collection in real-time systems*.

Figure 7-8. A metronome by Niki Odolphie.

7.2.4.5 Summary of the description of tasks

As a general overview, we have classified tasks as being periodic or sporadic. We have also identified an *idle task* as one that can be scheduled if nothing else is currently waiting to execute.

7.2.5 State and state diagrams

Consider the above program again: suppose we had four threads we were switching between. Once every four cycles, the above program is loaded, it checks that the sensor is still not ready, and it is immediately suspended again. Is this what we want? As an alternative check, consider the joining of threads:

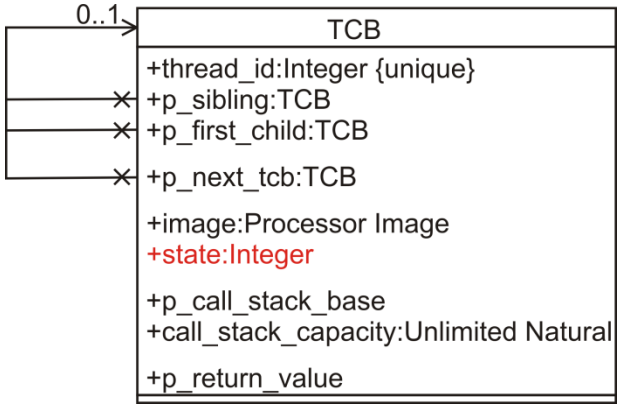
```
void *join_thread( tid_t tid ) {
    void *p_child_return_value;
    tcb_t *p_child_tcb = p_running_tcb->p_first_child;

    while ( p_child_tcb != NULL && p_child_tcb->thread_id != tid ) {
       p_child_tcb = p_child_tcb->p_sibling );
    }

    // The child is not found
    if ( p_child_tcb == NULL ) {
        p_child_return_value = NULL;
    } else {
        // 'p_child_tcb' now stores the address of the appropriate child's TCB
        while ( !( p_child_tcb->finished ) ) {
            scheduler();
        }

        // Remove the child from the list of children
        // Deallocate memory for the p_child_tcb

        p_child_return_value = p_child_tcb->p_return_value;

        free_tcb( p_child_tcb );
    }

    return p_child_return_value;
}
```

Why should we even try to load the parent if the child thread isn't finished (or *exited*)? Suppose we could flag a thread as "not being ready to run" because it is waiting on some other task? In that case, what could we do?

```
#define BLOCKED 0
#define READY   1
#define RUNNING 2
#define ZOMBIE  3
```

We will then include a state in our TCB, but additionally, we can also remove the `finished` field from the TCB, as well, as that is implied by the state being `ZOMBIE`.

```
typedef struct tcb {
    tid_t thread_id;

    struct tcb *p_sibling;
    struct tcb *p_first_child;

    struct tcb *p_next_tcb;
    processor_image_t image;
    int8_t state;

    void *p_call_stack_base;
    size_t call_stack_capacity;

    void *p_return_value;
    bool finished;
} tcb_t;
```



Thus, the only threads that are on the linked list are those that are *ready to run*. Thus, each time a thread

1. is placed on the ready queue, the state is set to `READY`,
2. is selected for execution, the state is set to `RUNNING`,
3. is waiting on something else beyond its control, the state is set to `BLOCKED`,
4. exits, its state is set to `ZOMBIE`[16].

We can even consider transitions between these states:

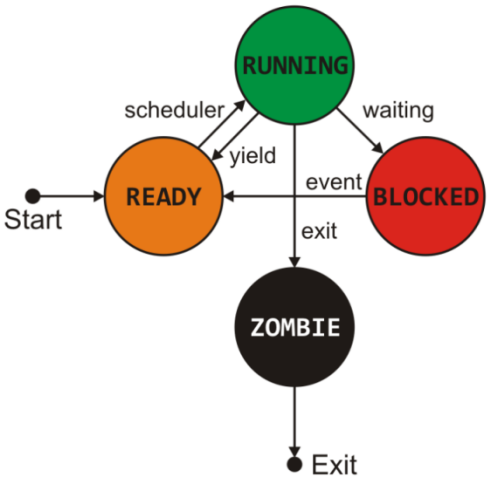


Figure 7-9. State diagram for our four states.

A state that exits but has not yet been joined by its parent remains in the `ZOMBIE` state. Once it is joined, the memory for that thread (the `TCB`) can be cleaned up.

Now, there is one issue we haven't discussed yet: how do we deal with moving a blocked state to a ready state (that is, how do we move it back onto the ready list)? To accomplish this, we will require multiple queues, but as every thread is only ever in one state (and therefore only on one queue), we can always reuse the same next pointer. Thus, we will discuss

1. a `TCB` queue data structure, and
2. the application of this to maintain state.

7.2.5.1 A `TCB` macro-based queue data structure

Previously, we only had a ready queue, and therefore all we did was have a head pointer and a tail pointer, and we simply manipulated these. If that is all that is necessary, great; however, as soon as there are two places where you want to implement the same abstract approach, it makes sense to create a common data structure.

```
typedef struct {
    tcb_t *p_head;
    tcb_t *p_tail;
    // No count is necessary
} tcb_queue_t;
```

Now, all we need do is implement a number of associated functions. As speed is necessary, it may be better to implement these as macros:

[16] A *zombie* is, *[i]n the West Indies and southern states of America, a soulless corpse said to have been revived by witchcraft; formerly, the name of a snake-deity in voodoo cults of or deriving from West Africa and Haiti.*

```
#define TCB_QUEUE_INIT ( queue ) {                      \
    (queue).p_head = NULL;                              \
    (queue).p_tail = NULL;                              \
}

#define TCB_QUEUE_INIT_AND_ENQUEUE( queue, tcb ) {      \
    (queue).p_head = (tcb);                             \
    (queue).p_tail = (tcb);                             \
    (tcb).p_next_tcb = NULL;                            \
}


#define TCB_QUEUE_EMPTY( queue ) ((queue).p_head == NULL)
#define TCB_QUEUE_FRONT( queue ) ((queue).p_head)

#define TCB_ENQUEUE( queue, tcb ) {                     \
    (tcb).p_next_tcb = (queue).p_head;                  \
    (queue).p_head = (tcb);                             \
                                                        \
    if ( (queue).p_tail == NULL ) {                     \
         (queue).p_tail = (tcb);                        \
    }                                                   \
}

#define TCB_DEQUEUE( queue ) {                          \
    if ( (queue).p_head == (queue).p_tail ) {           \
         (queue).p_tail = NULL;                         \
    }                                                   \
                                                        \
    ((queue).p_head)->p_next_tcb = NULL;                \
    (queue).p_head = NULL;                              \
}
```

You may ask yourself: "Why are we learning marcos? Can't we use `inline` functions?"

Yes, using the keyword `inline` to signal to the complier that it should replace a function call with the code itself is recommended for several reasons, most importantly: the compiler can check both the types of the arguments and the type of the return value before inlining the function. However, there is still a lot of legacy code that still uses macros.

You may note the use of `(queue).p_head` as opposed to `queue.p_head`. This is because macro substitutions are performed lexically in place. Thus, suppose that the queue was already a pointer. In this case, we would call

```
TCB_QUEUE_FRONT( *p_queue )
```

but this would be translated to

```
(*p_queue.p_head)
```

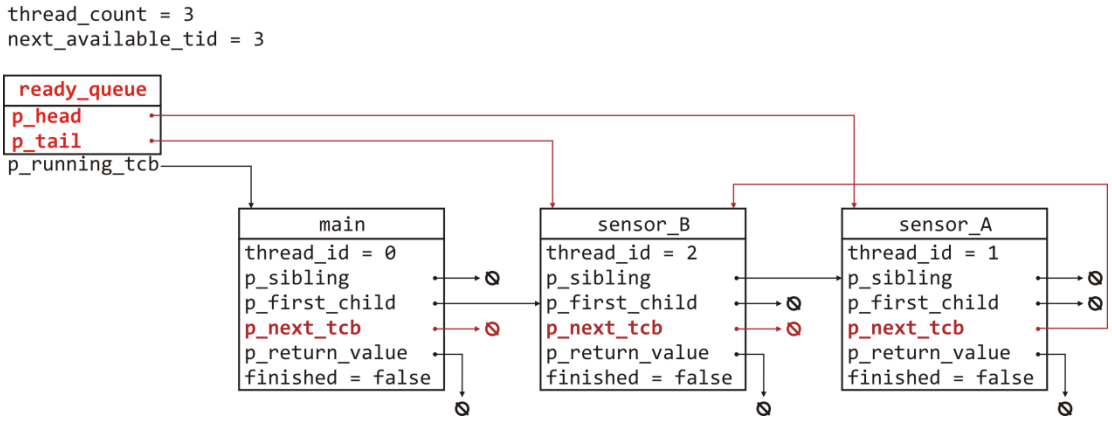
which would be interpreted by the compiler as `(*(p_queue.p_head))`, as the *dot* or *field-access* operator (`.`) has higher precedence than dereferencing (unary *). By wrapping the macro argument in parentheses,

```
#define TCB_QUEUE_FRONT( queue ) ((queue).p_head)
```

then the above is interpreted correctly as `((*p_queue).p_head)`.

Incidentally, the awkwardness of `(*queue_ptr).p_head` likely led to the inclusion of the *arrow* or *dereferencing field-access* operator (`->`), thus allowing you to use `p_queue->p_head`.

If we were to update our visual example in Section 7.2.2, the variable ready_queue would now have two fields storing the addresses of the first and last entries in the ready queue.



7.2.5.2 Using the state

We could now redefine some of our previous data structures:

```
tcb_queue_t ready_queue;
TCB_ QUEUE_INIT_AND_ENQUEUE( ready_queue, p_base_tcb );
```

and functions:

```
void scheduler() {
    // Copy registers Rk to p_running_tcb->image.Rk for k = 0, ..., 12
    // Modify the other registers R13 (SP), R14 (LR) and R15 (PC)
    //  - we can now make use of R0 through R12 to make these computations
    // Copy the special registers

    if ( TCB_QUEUE_EMPTY( ready_queue ) ) {
        // Restore only those registers that were used to execute the
        // condition of this if statement
        // Set the PC to p_running_tcb->image.PC
        //  - you don't need a return statement--this automatically "returns"
    }

    // Push the currently running thread onto the queue,
    // but only if it is still tagged RUNNING
    if ( p_running_tcb->state == RUNNING ) {
        p_running_tcb->state = READY;
        TCB_ENQUEUE( ready_queue, p_running_tcb );
    }

    // Pop the front thread off of the queue
    p_running_tcb = TCB_QUEUE_FRONT( ready_queue );
    p_running_tcb->state = RUNNING;
    TCB_DEQUEUE( ready_queue );

    // Context switch...
    // Copy the information from p_running_tcb->image back to the registers,
    //  - Be sure to set the program counter last.
}
```

We could, similarly, modify `create_thread(…)`:

```
bool create_thread( tid_t *p_tid, void *(*start_routine)( void * ), void *p_arguments ) {
    bool success;
    if ( thread_count == THREAD_CAPACITY ) {
        success = false;
    } else {
        ++thread_count;
```

```
        tcb_t *p_new_tcb = next_available_tcb();

        p_new_tcb->thread_id = *p_tid = next_available_tid;
        ++next_available_tid;

        // The new thread has no children
        p_new_tcb->p_first_child = NULL;

        // Prepend this new TCB onto the linked list of children
        p_new_tcb->p_sibling = p_running_tcb->p_first_child;
        p_running_tcb->p_first_child = p_new_tcb;

        // Place the argument onto the call stack in the right location
        insert_argument( p_new_tcb->stack, p_arguments );
        p_new_tcb->PC = start_routine;

        // Append the thread to the end of the list of ready threads
        p_new_tcb->state = READY;
        TCB_ENQUEUE( ready_queue, p_new_tcb );

        // Now return to the calling thread
        // - the new thread will be executed when the processor becomes available

        success = true;
    }

    return success;
}
```

Now, each sensor would have a queue, and if a thread is waiting on a sensor, instead of placing it on the ready queue, place it on the queue for the sensor.

```
tcb_queue_t sensor_1_queue;
TCB_QUEUE_INIT( sensor_1_queue );
```

Then, the scheduler would check sensors and place threads onto the ready queue:

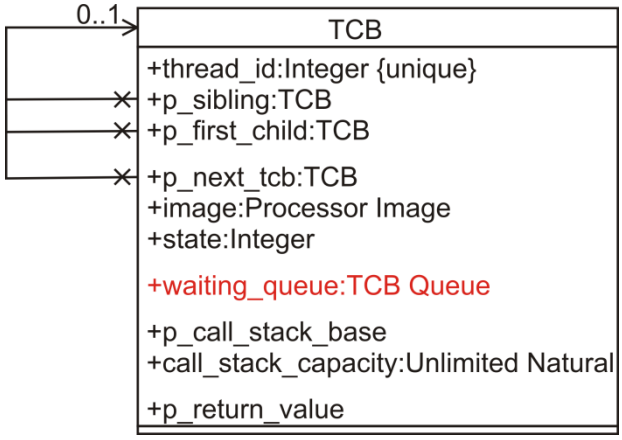
```
void scheduler() {
    // Copy registers Rk to p_running_tcb->image.Rk for k = 0, ..., 12
    // Modify the other registers R13 (SP), R14 (LR) and R15 (PC)
    //  - we can now make use of R0 through R12 to make these computations
    // Copy the special registers

    // Check if a sensor's queue is not empty
    if ( !TCB_QUEUE_EMPTY( sensor_1_queue ) ) {
        // If the sensor is ready:
        //  - pop the front from the queue
        //  - change the state to READY
        //  - enqueue it onto the ready queue

        if ( sensor_1_ready() ) {
            tcb_t *p_tcb = TCB_QUEUE_FRONT( sensor_1_queue );
            TCB_DEQUEUE(sensor_1_queue );

            p_tcb->state = READY;
            TCB_ENQUEUE( ready_queue, p_tcb );
        }
    }

    if ( TCB_QUEUE_EMPTY( ready_queue ) ) {
        // Restore only those registers that were used to execute the
        // condition of this if statement
        // Set the PC to p_running_tcb->image.PC
```

```
        //  - you don't need a return statement--this automatically "returns"
    }

    // Push the current thread onto the queue
    p_running_tcb->state = READY;
    TCB_ENQUEUE( ready_queue, p_running_tcb );

    // Pop the front thread off of the queue
    p_running_tcb = TCB_QUEUE_FRONT( ready_queue );
    p_running_tcb->state = RUNNING;
    TCB_DEQUEUE( ready_queue );

    // Context switch...
    // Copy the information from p_running_tcb->image back to the registers,
    //  - Be sure to set the program counter last.
}
```

Note that if multiple threads are waiting on a sensor, only one thread would be made ready. If multiple threads require access to the same result from a specific sensor, techniques for synchronizing (Chapter 7) should be used, instead, with only one thread actually waiting on the sensor itself.

Similarly, any thread that is waiting on a child to finish could be blocked and the waiting parent could be assigned to a pointer in the call to join. Then, when an exit occurs, the exiting thread would check if the parent is waiting on it to exit, and if so, it would set the state of the parent to ready and enqueue the parent on the ready queue.

```
typedef struct tcb {
    tid_t thread_id;

    struct tcb *p_sibling;
    struct tcb *p_first_child;

    struct tcb *p_next_tcb;
    processor_image_t image;
    int8_t state;

    tcb_queue_t waiting_queue;

    void *p_call_stack_base;
    size_t call_stack_capacity;

    void *p_return_value;
} tcb_t;
```



This queue would be initialized inside

```
bool create_thread( tid_t *p_tid, void *(*start_routine)( void * ), void *p_arguments ) {
    bool success;

    if ( thread_count == THREAD_CAPACITY ) {
        success = false;
    } else {
        ++thread_count;

        tcb_t *p_new_tcb = next_available_tcb();

        p_new_tcb->thread_id = *p_tid = next_available_tid;
        ++next_available_tid;

        // The new thread has no children
        p_new_tcb->p_first_child = NULL;

        // Prepend this new TCB onto the linked list of children
        p_new_tcb->p_sibling = p_running_tcb->p_first_child;
        p_running_tcb->p_first_child = p_new_tcb;

        TCB_QUEUE_INIT( p_new_tcb->waiting_queue );

        // Place the argument onto the call stack in the right location
        insert_argument( p_new_tcb->stack, p_arguments );
        p_new_tcb->PC = start_routine;

        // Append the thread to the end of the list of ready threads
        p_new_tcb->state = READY;
        TCB_ENQUEUE( ready_queue, p_new_tcb );

        // Now return to the calling thread
        // - the new thread will be executed when the processor becomes available

        success = true;
    }

    return success;
}
```

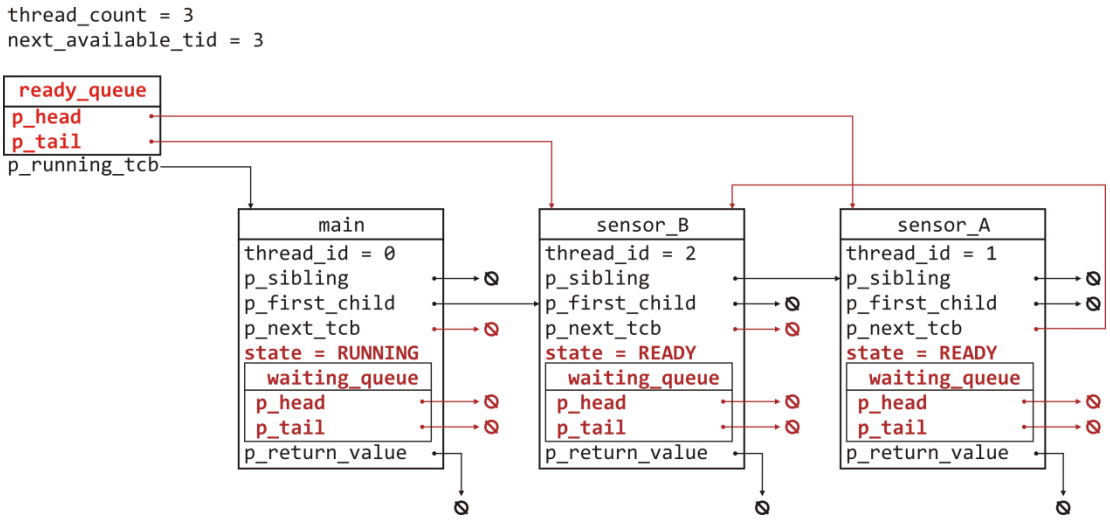
Now, inside join, if the child has not yet finished (that is, it is not yet in the `ZOMBIE` state), the parent puts itself on the child's waiting queue:

```
void *join_thread( tid_t tid ) {
    void *p_child_return_value;

    tcb_t *p_child_tcb = p_running_tcb->p_first_child;

    while ( p_child_tcb != NULL && p_child_tcb->thread_id != tid ) {
       p_child_tcb = p_child_tcb->p_sibling );
    }

    // The child is not found
    if ( p_child_tcb == NULL ) {
        p_child_return_value = NULL;
    } else {
        // 'p_child_tcb' now stores the address of the appropriate child's TCB

        // If the child is not ready, block this thread
        if ( p_child_tcb->state != ZOMBIE ) {
            TCB_ENQUEUE( p_child_tcb->waiting_queue, p_running_tcb );
            p_running_tcb->state = BLOCKED;
            scheduler();
        }

        // Remove the child from the list of children
        // Deallocate memory for the p_child_tcb

        p_child_return_value = p_child_tcb->p_return_value;

        free_tcb( p_child_tcb );
    }

    return p_child_return_value;
}
```

Now, when the thread exits, we must wake up any waiting parent.

```
void exit_thread( void *p_return_value ) {
    --thread_count;

    // Kill all descendent threads
    //  - how this is done is beyond the scope of this course...

    p_running_tcb->state = ZOMBIE;

    p_running_tcb->p_return_value = p_return_value;

    // If a parent is waiting for this thread to exit, wake it up
    if ( !TCB_QUEUE_EMPTY( p_running_tcb->waiting_queue ) ) {
        tcb_t *p_waiting_tcb = TCB_QUEUE_FRONT( p_running_tcb->waiting_queue );
        TCB_DEQUEUE( p_running_tcb->waiting_queue );
        p_waiting_tcb->state = READY;
        TCB_ENQUEUE( ready_queue, p_waiting_tcb );
    }

    scheduler();
}
```

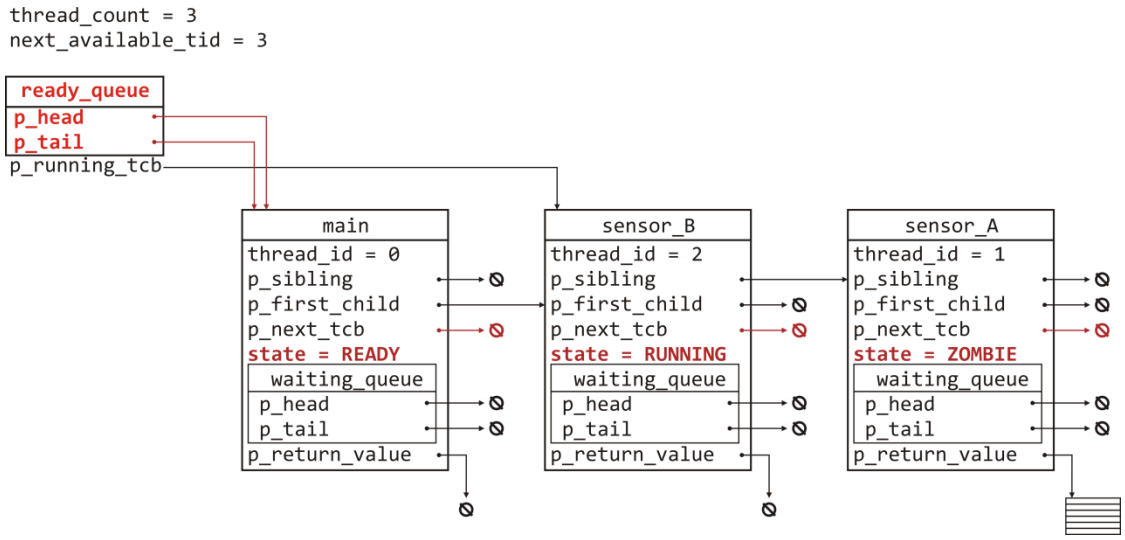
Let us continue with our visual example in Section 7.2.2, where for now the base thread is running.



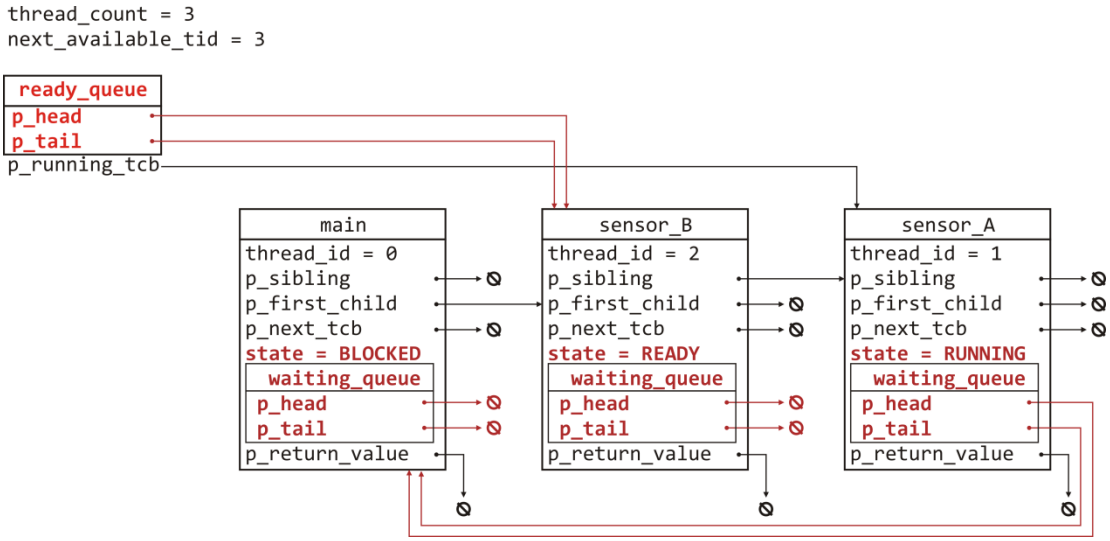
We will now look at what happens if:

1. `sensor_A` calls `exit_thread(…)` first before the base thread calls `join_thread(…)`, and
2. the base thread calls `join_thread(…)` first.

First, if `sensor_A` begins executing and then exits before the corresponding join is called, the state will be set to `ZOMBIE`, the return value will be stored, and the next available task will begin executing. This is shown here:



If, however, the base thread joins first with `sensor_A`, it will place itself into a blocked state, enqueue itself onto the waiting queue for `sensor_A` and then the next ready thread would begin executing, in this example, `sensor_A`.



Now, when `sensor_A` exits, it will set its state to `ZOMBIE` and save the return value, unblock the base thread, and place that thread onto the ready queue. It will not put itself back onto the ready queue, but instead, `sensor_B` will begin executing. When it comes time for the base thread to execute, it will continue the execution of `join_thread(…)`, which will fetch the now-stored return value, free up the memory for the tcb of `sensor_A`, and return the address of the return value.

7.2.5.3 The idle thread: if nothing else is ready

What happens if nothing is ready? Does the scheduler just continually check if sensors are ready? If a response from a sensor is critical, this may be one solution, but another solution is to have an *idle thread*. If nothing else can run, execute this thread instead. Such a thread cannot wait on any other tasks, so it may perform some background checks and then yield the processor every so often by calling the `scheduler()` routine (so that in our example, the scheduler can check the sensors again).

In order to implement such a scheme, we would have to introduce a new state:

```
#define IDLE 4
```

This is the constant state of the idle thread, so it will never be enqueued on the ready queue. The scheduler will choose this thread if and only if the ready queue is empty. A pointer to the TCB of such a thread would be assigned to a global variable, similar to `p_base_tcb` and `p_running_tcb`.

Now, what do we do if there are no tasks waiting to be executed? Do we do nothing (loop)? One possibility is to *stop* (or *halt*) the processor, in which case, there will be a corresponding savings in energy and a reduction in heat production. The processor could be woken up by a hardware interrupt either by an event occurring (a device signalling the processor) or by a clock independent of the processor.

Alternatively, we can implement an *idle task* that is executed any time no other tasks are available. The idle task would perform background duties that do not require resources—perhaps checking values, calculating statistics for subsequent analysis, etc. A constant global variable `p_idle_tcb` could store the address of this idle task's task control block, and if there is nothing else to execute, the idle task is scheduled.

Just because it is called the idle task, however, does not mean that it does nothing useful. In fact, it may be advantageous to use this otherwise unused processor time in some way. For example, when you install SETI@home (the *search for extra-terrestrial intelligence at home*), it replaces the idle task on your computer with one that analyzes radio signals.

More realistically, an example of what an idle task could do in a real-time system is collect statistics about trends within data. For example, in a control system, it may be necessary to respond whenever a parameter exceeds a critical value. There may be a standard response which will always solve the problem, but the intensity of the response could be tempered based on prior trends. For example, the response in the first case in Figure 7-10 may be significantly less drastic than that in the last case.

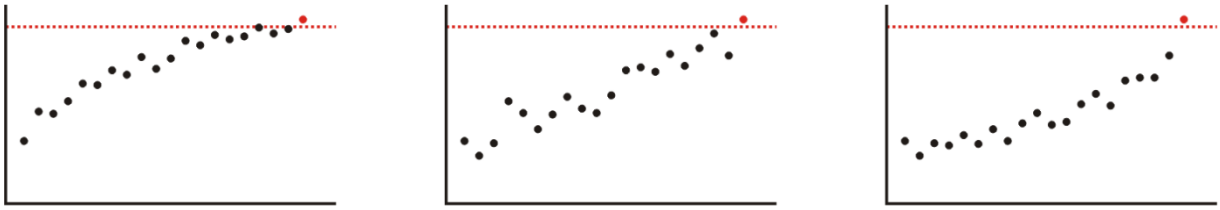


Figure 7-10. Possible trends toward reaching a critical value.

Alternatively, the idle task may search for cases where a preventative response may be more desirable.

Note that in a non-preemptive situation, the idle task must yield the processor periodically, usually with a relatively small period.

In his blog on EmbeddedGurus[17], Nigel Jones as a few recommendations on what an idle task can perform:

1. acting as a watchdog, checking the state of other systems of if deadlock has occurred,
2. saving power by putting the microcontroller to sleep or by using other power-saving strategies,
3. calculating the load on the system (for example, which tasks are running for how long and how often) and using this for subsequent analysis of the performance of the system,
4. performing a check on read-only flash memory (later, we will introduce a cyclic redundancy check (or CRC), which can be used to do this using read-only operations, or
5. performing a check on main memory, although this will require both read and write operations on main memory, and thus may introduce race conditions that must be dealt with.

Such a list is not exhaustive, but gives the most commonly used options.

7.2.5.4 Some observations

In our design above, we still have a few observations we must make:

1. We no longer worry about polling to see if a child is finished; however, we still have to continue polling the sensor to see if it is ready.
2. This is safe code so long as we do not have any interruptions in our execution. While the strategy above works, it will fail as soon as we consider anything other than polling for determining the state of sensors and other peripherals; the code is subject to errors we will discuss in the topic on synchronization.
3. The above design is not appropriate for real-time systems:
    a. Our list of ready threads is a first-come—first-served (FCFS) queue, but does not have any concept of a thread's importance. What happens if a very important thread must begin executing?
    b. Suppose that a parent queue is performing a time-sensitive task, but it is waiting on a child to exit, and is therefore placed the end of the ready queue. When the child finally exits, the parent will have to wait its turn—possibly missing its deadline.

[17] See http://embeddedgurus.com/stack-overflow/2013/04/idling-along/.

Therefore, the approach above is very inappropriate for real-time systems, but it is an introduction to some of the operations that must be performed when handling multiple tasks.

7.2.5.5 Case study: states in various systems

The state diagram for the CMSIS-RTOS RTX is similar but slightly more complicated, as shown in Figure 7-11.

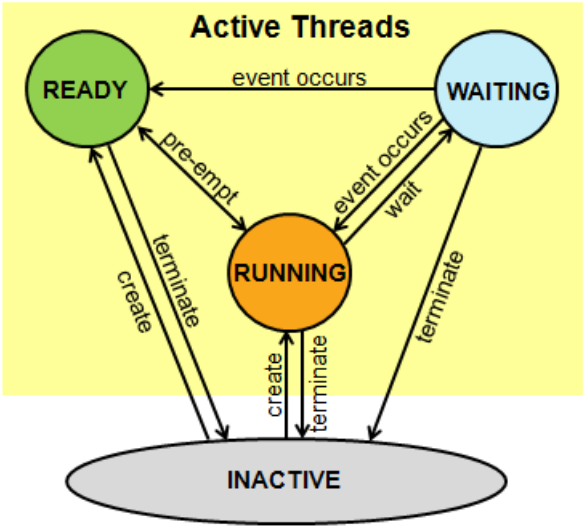


Figure 7-11. The CMSIS state diagram (from http://www.keil.com/pack/doc/cmsis/RTOS/html/modules.html).

Note, however, that there are additional transitions.

The states for Unix processes (threads with resources) is even more complex, as shown in Figure 7-12.

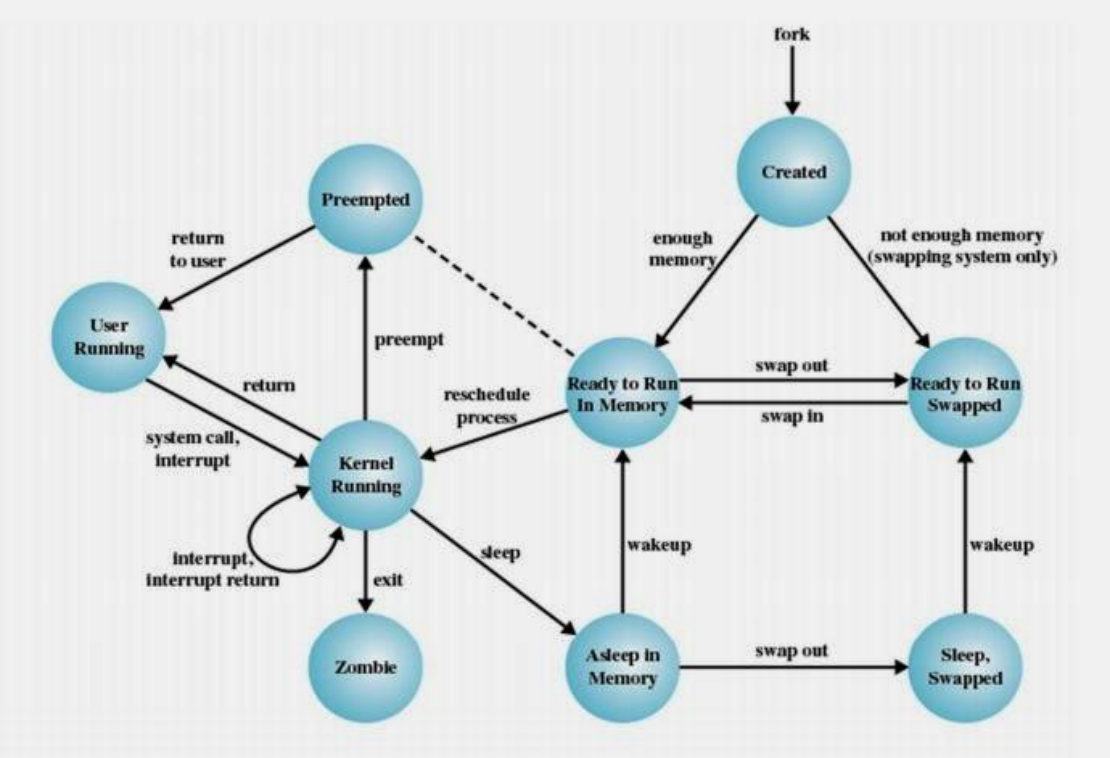


Figure 7-12. The state diagram for Unix processes
(from http://opensourceforgeeks.blogspot.ca/2014/03/processes-and-threads-in-linux.html).

7.2.5.6 Summary of the state

In our design above, we have described how we can use the state to implement multiprogramming more efficiently than polling. We have looked at how we the various aspects can interact, and some of the issues that arise from this naïve approach. We will continue by looking at timing diagrams.

7.2.6 Timing diagrams

Now that we are talking about multiple tasks are executing on a single processor, it is sometimes convenient and instructive to show the execution graphically. This is done through *timing diagrams*. For example, in Figure 7-12, you see that Task 1 runs for 7 s, Task 2 for 4 s, and Task 3 for 9 s.

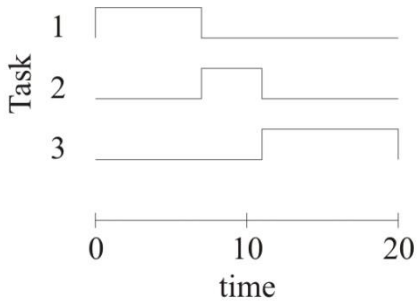


Figure 7-13. A timing diagram.

Now, timing diagrams in analog systems or where the exact switches between tasks are relevant will often show a transition from task executing to the next, as demonstrated in Figure 7-14. In general, this will not be necessary

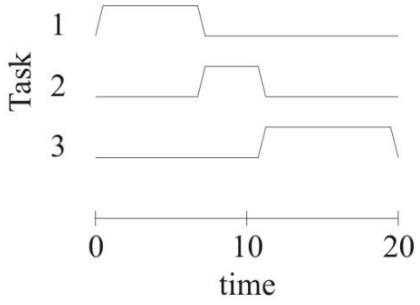


Figure 7-14. A timing diagram with transitions included.

Transitions do, however, allow multiple tasks to be shown on a single timeline.

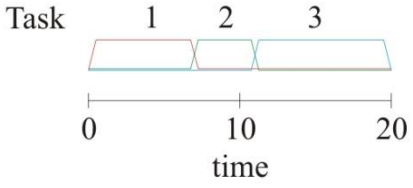


Figure 7-15. Tasks with transitions on a single timeline.

Now, if a task is starting at the initial time on the scale, we indicate a transition from a non-running status to running; otherwise, we assume the task has been executing in the past already. Similarly, if a task ends execution at the final time on the scale, we assume it has ended; otherwise, we assume that it continues executing. These are shown in Figure 7-16.

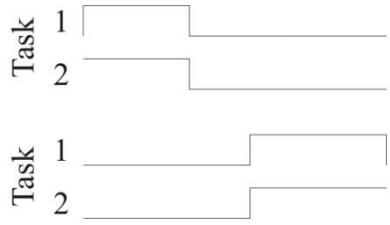


Figure 7-16. Timing diagrams with Task 1 starting at the initial time, while Task 2 is already running at the initial time, and where Task 1 terminates at the final time while Task 2 continues running after the final time.

We will be using timing diagrams periodically to describe how various algorithms work.

7.2.7 Timing interrupts

We will discuss interrupts in the next topic in greater detail, but one issue we must examine right now is that of a timing interrupt. We have already discussed how a task or thread could be put to *sleep* by having its state set to `READY` and then having its TCB enqueued on the ready queue. So far, this has only occurred as a response to the parent inquiring as to whether the child is finished executing. Such a change of state is the result of the thread or task making a request that cannot be currently satisfied, but given time, the child will finish executing.

Another possibility might be a signal from the real-time clock[18]. In the morning, you are sleeping, and the clock interrupts your sleep. Alternatively, if you have an appointment at 2:00, and Google Navigate realizes you must: walk to your car and then drive your car to the appointment, it will calculate the expected travel time and interrupt whatever you are doing at 1:35 to notify you that you must leave to get to your appointment on time.

Similarly, you can get a real-time clock to *interrupt* whatever task or thread is currently executing. The program counter (PC) of that task or thread is stored, and the program counter is set to the first instruction of a special function called an *interrupt service routine* (ISR). We will discuss this later, but for now, we will assume that it is possible for such a routine to do various simple tasks such as:

1. manipulate the ready queue,
2. to either
    a. return to currently executing task or thread, or
    b. change the state of the currently executing task to `READY` and place it onto the ready queue and call the scheduler.

In the next topic on hardware interrupts, we will generalize this; however, we will look briefly at timing interrupts on the Keil RTX RTOS. In the Keil RTX RTOS, each time the clock fires an interrupt, the function

```
// SysTick interrupt service routine
void SysTick_Handler( void )  {
        // Do something...
}
```

is executed. We will discuss interrupt service routines (ISRs) later, but each time the clock counts down to 0, a timing interrupt is fired, and the operating system responds by halting the current task and calling the corresponding ISR. In the design of the Keil RTX RTOS, the ISR associated with a timing interrupt is always named `SysTick_Handler`. A default handler is defined, but if the developer defines another function with the same name, the developer's version takes precedence.

The clock frequency is stored in the variable

[18] Please remember, the "real-time" in "real-time clock" simple means that it is a clock keeping track of the *actual* time. It has nothing to do with the same phrase in "real-time systems".

```
uint32_t SyStemCoreClock;      // ticks per second
```

which stores the frequency of the timer. The function

```
uint32_t SysTick_Config( uint32_t ticks )
```

initializes the `SysTick` timer and its interrupt. The argument is the number of ticks between interrupts, and as `SystemCoreClock` stores the frequency (ticks per second), you can use calls such as

```
SysTick_Config( SystemCoreClock/100 );   // Generate an interrupt every 10 ms
SysTick_Config( SystemCoreClock/1000 );  // Generate an interrupt every millisecond
SysTick_Config( SystemCoreClock/10000 ); // Generate an interrupt every 100 us
```

Now, the main oscillator on the MPC 1760 is a 12 MHz quartz oscillator and in general, you will be using this oscillator to drive the processor. The oscillator driving the LPC1768 must fall into one of two ranges:

1. 1 MHz to 20 MHz, or
2. 15 MHz to 25 MHz.

and thus you would select the first range. To generate the frequency of the processor, you must select which clock is used: you can either use the main oscillator, but you can also use the internal RC oscillator or the real-time clock oscillator. In any case, the input frequency F_{in} must be between 32 kHz and 50 MHz.
To boost this to the speed of the processor, a phase-locked loop (PLL) is used, with integer multiplier selection M_{sel} and a divider selection N_{sel}. The frequency of the PLL is calculated as $2\frac{M_{\text{sel}}}{N_{\text{sel}}}F_{\text{in}}$ and therefore to have a processor speed of 400 MHz with the main oscillator, one can choose $M_{\text{sel}} = 50$ and $N_{\text{sel}} = 3$. The processor speed must be in the range 275 MHz to 550 MHz. All of this can be configured in the file `system_LPC17xx.c`.

7.2.8 Summary of multiprogramming and non-preemptive scheduling

In this section, we described multiprogramming, where when one thread is waiting on a resource, such as

1. a child thread to finish, or
2. a sensor to be ready

It is possible to halt execution of that thread and start another independent thread. There are some very simple non-preemptive scheduling algorithms we can use for multiprogramming, but they are not appropriate for real-time systems. Next, we will consider interrupts to the currently executing task or thread.

7.3 Non-preemptive scheduling algorithms

We will introduce scheduling algorithms by looking at five approaches to scheduling a collection of single-instance tasks, each with known computation times c_k and deadlines d_k. The last four algorithms are called *fair* in the sense that they attempt to minimize a measurable quantity that may be said to describe a reasonable system. These five algorithms are:

1. timeline,
2. first-come—first-served,
3. shortest-task next,
4. earliest-deadline first, and
5. least-slack first.

In each case, any time the processor is available, a task will be scheduled to execute and that task will continue executing until it completes and exits. Unfortunately, we will see that the first is prohibitive, the next two are unacceptable for real-time systems, but the last two will appear to have promise for real-time systems.

In each of our three cases, we will schedule the four tasks.

Table 2. Tasks with computation times c_k and deadlines d_k.

Task	Computation time (ms)	Deadline (ms)
A	40	76
B	6	84
C	14	35
D	20	37

We will assume that the tasks have arrived in the order they are listed here.

7.3.1 Timeline scheduling

Given n tasks, there are $n!$ different ways of scheduling those tasks. Thus, timeline scheduling is where a human being sits down and devises an acceptable schedule. In our example, there are 4! = 24 different possible schedules, and thus a designer could sit down and use whatever heuristics he or she chooses to come up with an acceptable schedule. In the above example, of the 24 schedules, only two schedules allow all tasks to meet their deadlines.

Unfortunately, timeline scheduling is cost-prohibitive and can only be performed apriori. Thus, we must look at scheduling algorithms that can select tasks at run time based on quantitative criteria.

7.3.2 First-come—first-served scheduling

Our first definition of fairness will be to schedule the tasks in the order in which they arrive; that is, in a first-come—first-served (FCFS) ordering. This is a common definition of fairness which is used in many client-server models. If you walk into a bank, you expect to be served before someone who walks in after you (that is, unless they have an appointment). Similarly, there is seldom a good reason for a web server to satisfy requests in anything other than a FCFS order. Of course, FCFS may not always work: during a 1979 Who concert selling general admission seating, eleven fans died in a crush to reach the *best* seating. Today, most venues are required by law to sell specific seats to clients, and FCFS only applies to the order in which clients can select their seats.

The implementation of the FCFS algorithm is quite straight-forward. The ready queue is an actual queue, and new tasks are pushed onto the end of the queue, while each time the processor is ready, a task is dequeued from the front of the queue. If the queue is ever empty, the idle task is run. Suppose the idle task is executing and a hardware interrupt occurs.

The interrupt handler will execute the appropriate interrupt service routine, and when it is finished, it will call the scheduler, which will then check:

```
if ( p_running_tcb == P_IDLE_TCB ) {
    p_running_tcb = pop_tcb();
} else {
    if ( waiting_size == 0 ) {
        // Do nothing
    } else {
        TCB_ENQUEUE( p_running_tcb );
        p_running_tcb = pop_tcb();
    }
}

context_switch();
```

Thus, if we ever come across a situation where there is nothing else ready to run, the idle task will be scheduled.

If we apply FCFS to the four tasks mentioned above, we note that Tasks C and D miss their deadlines, as is shown in Figure 7-17.

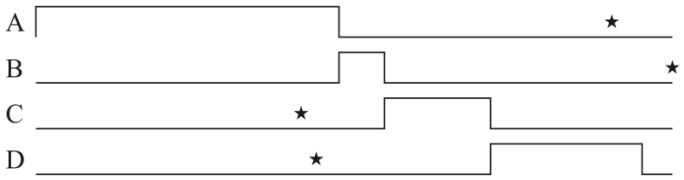


Figure 7-17. Scheduling four tasks using FCFS. Deadlines are marked with stars.

Next, we will look at shortest-job next.

7.3.3 Shortest-job next scheduling

A more global definition of fairness is to minimize the average wait time. You have already experienced this in a grocery store: suppose two customers are waiting to be served and one has two items and the other has one hundred items. The service time for the individual with milk and cookies is 30 seconds (0:30), while the service time for the individual throwing a party is perhaps 10 minutes (10:00). If the milk-and-cookies client goes first, she is finished after 30 seconds, and the other client had to wait 30 seconds for the server to finish with the milk and cookies, and thus will served after 10:30. Thus, the average wait time of the two is 5:30. If the party animal goes first, he is finished after 10:00, and the milk-and-cookies client must wait 10 min and her service is finally finished after 10:30; hence, the average wait time is 10:15.

The shortest-job next criteria specifies that the next task to be executed is the one that has the shortest computation time. This will minimize the average wait time, and thus all tasks benefit.

As with the FCFS algorithm, shortest job next could be easily implemented by using a binary min-heap using an array; however, just as efficient, would be to implement some form of leftist heap. A leftist heap is a node-based binary tree where each node tracks its *null-path length*—that is, the length of the shortest path from that node to a descendent that is not a full node.

1. insertions are always performed in the right sub-tree, and
2. if after an insertion, the right sub-tree has a larger null-path length, the two children are swapped so that the left sub-tree now has the larger null-path length; while
3. deleting the minimum (top) is performed by inserting the right sub-tree into the left sub-tree.

It can be shown that operations on this data structure are also $\Theta(\ln(n))$, just like a binary min-heap, only now we do not have to pre-allocate memory for the data structure. Only now, we would be required to have two pointers in our TCB.

To prove that shortest-job next minimizes the average wait time, we observe that if Task *i* is executed before Task *j*, then the computation time of Task *i* must be added onto the wait time of Task *j*. Thus, we wish to find an order $k_1, k_2, \ldots, k_n$ such that it minimizes

$$\frac{1}{n}\sum_{i=1}^{n}\left(\sum_{j=1}^{k} c_{k_j}\right) = \frac{1}{n}\sum_{i=1}^{n}\left((n-i+1)c_{k_i}\right).$$

Minimizing the average wait time is equivalent to minimizing the sum itself, so we will multiply by *n* and expand the product to get that we must minimize

$$\sum_{i=1}^{n}\left((n-i+1)c_{k_i}\right) = (n+1)\sum_{i=1}^{n} c_{k_i} - \sum_{i=1}^{n} i c_{k_i} .$$

The left-hand sum is a constant—it is the sum of the computation times, so this cannot change. The other, however, depends on the ordering of the computation times, and so as to minimize the sum, we must maximize

$$\sum_{i=1}^{n} i c_{k_i} .$$

Suppose we have two tasks with computation times 5 and 10. If we run the shorter task first ($k_1 = 1$ and $k_2 = 2$), we get $1\cdot 5 + 2\cdot 10 = 25$, but if we run the longer task first ($k_1 = 2$ and $k_2 = 1$), we get $1\cdot 10 + 2\cdot 5 = 20$. Thus, running the shorter task first maximizes this sum, and therefore minimizes the sum $\sum_{i=1}^{n}\left((n-i+1)c_{k_i}\right)$. In general, however, consider the i^{th} and j^{th} tasks to be run. Their contribution to this sum is

$$i c_{k_i} + j c_{k_j} .$$

To see which must have the larger computation time, c_{k_i} or c_{k_j}, we will add zero in the form $ic_{k_j} - ic_{k_j}$ to get

$$i c_{k_i} + \left(i c_{k_j} - i c_{k_j}\right) j c_{k_j} = i\left(c_{k_i} + c_{k_j}\right) + (j-i)c_{k_j} .$$

On the right hand side, the left-hand term is constant: it is *i* times the sum of the two computation times, but the right-hand term is maximized if c_{k_j} is larger, that is, the task with the longer computation time is run second.

Now, if we consider any two tasks, we can always increase the sum $\sum_{i=1}^{n} i c_{k_i}$ by ensuring that the longer computation time comes second, and therefore the computation times must be sorted. ■

Never-the-less, using shortest-task first, we note that Tasks A and D miss their deadlines, as is shown in Figure 7-18.

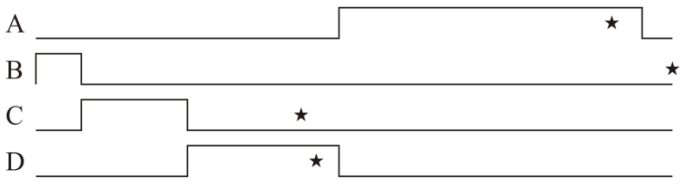


Figure 7-18. Scheduling four tasks using shortest-task first. Deadlines are marked with stars.

7.3.4 Earliest-deadline first scheduling

As another scheduling algorithm, consider that each task has a deadline, so begin executing that task that has the earliest deadline first (EDF). This algorithm is fair based on the immediate need for the processor, and airports will use a hybrid scheduling algorithm for security checks: if your flight doesn't leave for more than half an hour, you wait in the FCFS queue, while if your plane is leaving earlier, you are processed based on who's flight is leaving the earliest—they have the earliest deadline to catch their plane. With this algorithm, each of the four tasks meet their deadline as shown in Figure 7-19, and this is the algorithm we will become central when we begin looking at preemptive scheduling of periodic, sporadic and fixed-instance tasks.

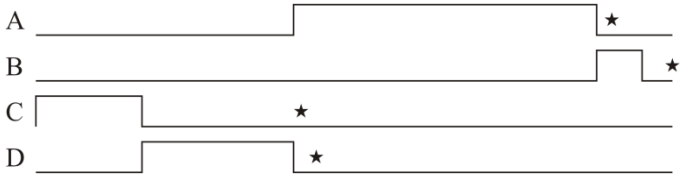


Figure 7-19. Scheduling tasks using earliest-deadline first scheduler. Deadlines are marked with stars.

We will see later that if tasks can be scheduled such that all tasks meet their deadlines, then EDF will create a schedule that will allow all tasks to meet their deadlines. Like the shortest-job next scheduler, EDF can be implemented using a priority queue, only now the priority is the deadline.

7.3.5 Least-slack first scheduling

One final scheduling strategy, and one that is similar to EDF, is least-slack first (LSF). Slack refers to how much time a task can wait before it must start running. If we consider Tasks C and D, we note that while Task C has an earlier deadline, Task D must start earlier to meet its deadline, as shown in Figure 7-20.

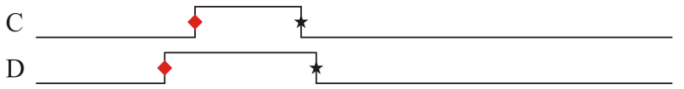


Figure 7-20. Tasks C and D with their deadlines marked with stars and the latest possible start time marked with diamonds.

Thus, if we subtract the computation time from the deadline, we get the slack ($s_k = d_k - c_k$) for each task, as shown in Table 3.

Table 3. Tasks with computations times c_k, deadlines d_k and latest start times (or *slack*) s_k.

Task	Computation time (ms)	Deadline (ms)	Latest start time (or *slack*) (ms)
A	40	76	36
B	6	84	78
C	14	35	21
D	20	37	17

Under this algorithm, the only difference would be that Task D starts, and is then followed by the scheduling of Task C, as shown in .

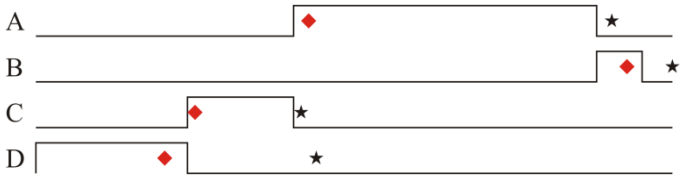


Figure 7-21. Scheduling tasks using least-slack first scheduler. Latest start times are marked with diamonds.

Like EDF, all tasks meet their deadlines with LSF, and if it is possible to schedule the tasks so that all tasks meet their deadlines, LSF will create a schedule that allows all tasks to meet those deadlines.

One important point is that by just looking at the EDF, we may schedule a task that will never-the-less miss its deadline. By focusing on LSF, if the latest start time has already passed, in a firm real-time system, we might as well not even begin executing the task—it's deadline will be missed and thus whatever result it produces will be worthless, and in a hard real-time system, if it is known a critical task will miss its deadline, steps can be taken to ameliorate any damage from the missed deadline.

7.3.6 Summary of non-preemptive scheduling algorithms

We have looked at five scheduling algorithms where each task is scheduled and allowed to continue until it finishes execution. Each algorithm attempts to be *fair* based on certain criteria:

1. what makes the designer happy,
2. which task has been waiting the longest,
3. how can we minimize the average wait time,
4. which task has the earliest deadline, and
5. which task has the earliest start time?

Of these five scheduling algorithms, only the first and last two ensure that if it is possible for tasks to meet their deadlines, that the tasks will be scheduled in such a way to allow all tasks to meet their deadlines. Consequently, we will focus on these algorithms in real-time systems.

7.4 Preemptive scheduling algorithms

Real-time systems that are not purely responsive (say, a child's toy that that responds only after someone activates a pressure or motion sensor) will have periodic tasks, if nothing else, to check that the state of the system is not corrupted. Therefore, scheduling algorithms will have to take into account the idea of preemption. We will begin with motivation for using preemptive scheduling, and then we will look at various preemptive algorithms, including:

1. timeline,
2. round-robin,
3. deadline-based (earliest-deadline first and least-slack first), and
4. priority-based (rate monotonic)

scheduling. In any real time system, it is usually necessary that any scheduling algorithm runs in $\Theta(1)$ time, although if the number of tasks is small, an $O(\ln(n))$ algorithm is also potentially acceptable.

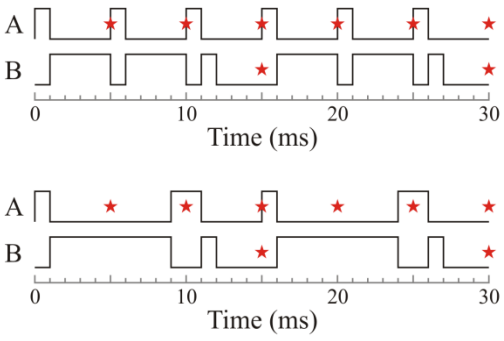
7.4.1 Motivation

In any but the simplest real-time systems, there will be tasks that run periodically, and in firm or hard real time systems, those tasks that do run periodically must execute once per period, thus the end of each period can be seen as the deadline of the task. Additionally, it is also not possible for periodic tasks to start execution before their period. For example, a task running 24 times per second meant to digitize the image captured by a camera cannot start executing prior to when the image is captured, and must complete its task before the image refreshes with the start of the next period. If any instance of that task starts too early or misses its deadline, the system is compromised. However, non-pre-emptive scheduling algorithms cannot work. Consider a system that has two periodic tasks:

Task	Computation time (ms)	Period (ms)
A	1	5
B	9	15

It is possible to schedule these tasks, but only if Task B is interrupted at some point between 5 ms and 9 ms to allow Task A to run during its second period. Two possible schedules follow the rules:

1. First, Task A is given priority: if Task A is ready to execute and Task B is currently executing, Task B is interrupted and Task A is scheduled.
2. Second, a switch is only made if absolutely necessary—that is, we will switch only if not switching will cause a task to miss a deadline. That is, a switch is made only if the slack time of a task goes to zero.



Beyond the scope of this introductory text, but something to think about, in real-time systems, it might be undesirable to have a task run back-to-back. For example, in the case of capturing an image at 24 fps, there is a small window during which the CCD is refreshed, and therefore unavailable. Thus, in reality, in addition to each task having a period, it may also have a start time and a deadline within that period: the task must run 24 times per second, but within any period, it

must not start running until 7 ms has passed, and it must complete execution 10 ms before the end of the period, thus leaving $0.024\dot{6}$ s during which the task can execute.

Another issue that must be considered is that we really have no choice but to rely on timing interrupts. Recall that when we are scheduling algorithms, the computation time is usually only the worst-case computation time. Thus, while a task may normally require only 1 ms, if it must as a response to a change send out a message, it may run for 3 ms, thus our scheduling algorithm must take this worst-case scenario into consideration. Unfortunately, it is when a system is responding to external events that most tasks are going to executing close to their worst-case computation time. Thus, it is generally not possible to simply rely on tasks yielding the processor. In addition, suppose that Tasks A and B in our example are the only critical tasks, but there are other background non-real-time tasks that can optionally compute if the critical tasks are not running (at least 0.2 s/s are available). If these tasks are executing, they must be halted if either of the critical tasks becomes ready (enters its next period). Such non-real-time tasks could periodically yield the processor, but and the interval between yields may have to be quite small to ensure the critical tasks have an opportunity to start executing, thus producing a significant overhead.

Consequently, without preemption and timing interrupts, a developer would be seriously hampered as to how. The cost of preemption, however, is the cost of the necessary context switch and the addition overhead of scheduling the necessary timing interrupts. We will now continue looking at various preemptive scheduling algorithms and consider their appropriateness for real-time systems.

7.4.2 Timeline scheduling

As with our non-preemptive schedulers, it is always possible to design a schedule by hand. In such cases, the designer can come up with a schedule that, for example, guarantees a minimum number of context switches between tasks while still ensuring that all tasks meet their deadlines. It is also a lot easier for a human to take into account subtle dependencies between the various tasks. For example, two periodic tasks may check each second to take an appropriate response if the system is too hot or too cold, respectively. Each task may, in the worst case execute for 20 ms, but normally they simply check the temperature (1 ms) and if everything is okay, they yield the processor. Consequently, while the cumulative worst-case run time may be 40 ms, in reality, the worst-case total computation time of the two tasks will be only 21 ms.

Thus, given a collection of n periodic tasks (c_k, τ_k), if the utilization does not exceed unity, it follows that we can always schedule these tasks over a period of the least common multiple (lcm) of the periods, that is, the lcm(τ_1, τ_2, τ_3, …, τ_n), a value that is also described as the *super period*. If this is a relatively small number, it might well be a reasonable approach. We will consider a number of examples and we will then consider issues when statically scheduled tasks come into conflict with sporadic tasks.

7.4.2.1 Example of timeline scheduling

Consider an example of three tasks (1, 6), (9, 14) and (4, 21), and as the sum $\frac{1}{6}+\frac{9}{14}+\frac{4}{21}=1$, we could, by hand, create a schedule over the super period of lcm(6, 14, 21) = 42 ms. For example, we could determine at any point, which task that is ready to execute using the earliest-deadline-first algorithm, as is shown in Figure 7-22.

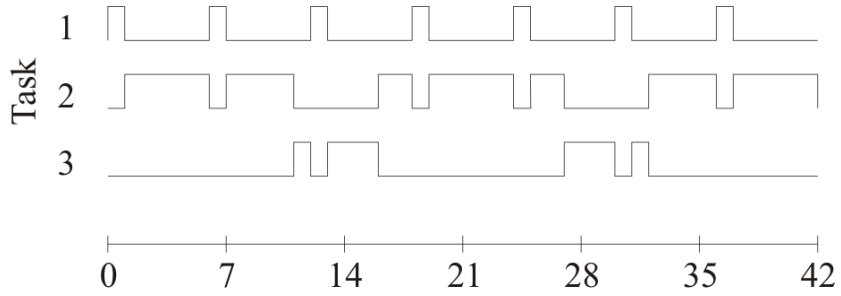


Figure 7-22. Scheduling tasks (1, 6), (9, 14) and (4, 21) using earliest-deadline first.

However, this is not as optimal as an alternate choice, shown in Figure 7-23, which reduces the number of context switches from 18 to 12, thereby reducing unnecessary overhead. Most tasks execute without interruptions.

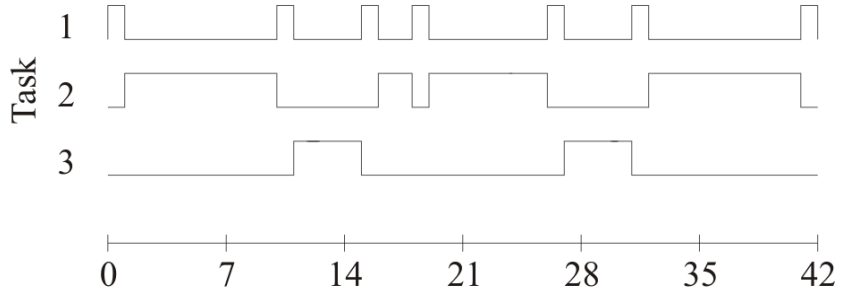


Figure 7-23. An alternate schedule for the three processes in Figure 7-22.

Thus, timing interrupts can be scheduled for 0, 1, 10, 11, 15, 16, 18, 19, 26, 27, 31, 32 and 41 ms each 42 ms interval. When the interrupt occurs, the scheduler looks up the next task. As each task will not usually require its full worst-case computation time, when a task yields the processor early, the idle process can execute, perhaps making various checks or other operations.

7.4.2.2 Large super periods

Timeline scheduling of periodic tasks is not always reasonable, especially if the period As an example, the lcm of the periods of the three tasks (4, 21), (13, 22) and (8, 65) is 30030, and the tasks are schedulable, as $\frac{4}{21}+\frac{13}{22}+\frac{8}{65}=\frac{27161}{30030}\leq 1$. However, for the scheduler to keep such a large table may be prohibitively expensive.

In this case, it might be easier to slightly increase the times, but to create a lower lcm. In this case, we could use (5, 22), (13, 22) and (9, 66), and $\frac{5}{22}+\frac{13}{22}+\frac{9}{66}=\frac{21}{22}\leq 1$, and thus device the schedule with the super period of 66 ms shown in Figure 7-24.

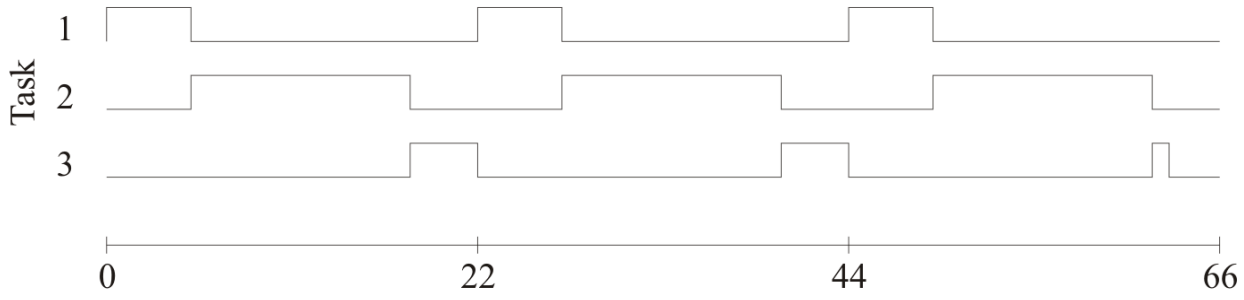


Figure 7-24. One of many possible schedules for the three tasks (5, 22), (13, 22) and (9, 66).

Note that with the augmented times, we always increased implied the processor utilization. In some cases, however, it may not be possible to change the periods; for example, the period of a task translating images must match the frequency of 24 fps. In others, the original tasks may be schedulable, while the augmented tasks are not (for example, (4, 10) and (1, 6) and (6, 14)).

7.4.2.3 Issues with timeline scheduling

Suppose we have timeline scheduled a set of periodic tasks but these are now placed into a real-time environment where sporadic tasks must occasionally be executed. In this case, some tasks may be required to miss their deadlines. How do we decide which tasks will miss their deadlines? In more complex systems, periodic tasks may appear dynamically at run time and therefore cannot be scheduled, or given a number of periodic tasks, perhaps only a subset may be executing at any one time (depending on, say, external conditions such as light, temperature or other factors). Another issue with timeline scheduling that the entire schedule must be recalculated if either a new task is added to or removed from the set of executing tasks, or the worst-case computation time or periods of any of the tasks change.

7.4.2.4 Summary of timeline scheduling

Timeline scheduling makes it possible to determine the times at which each of the tasks runs, but this requires that the scheduling be done during the early development phase. If there is a change to the number of tasks that are required, the scheduling will have be redone, as well.

7.4.3 Round-robin and other fair schedulers

The simplest preemptive scheduler is termed *round-robin*. This is more of an idea than an algorithm, as it can be combined with most other scheduling algorithms. Basically, one issue with FCFS is that whichever task comes last, it may have to wait a significant period of time prior to it being scheduled. If we are in a multi-user system where some tasks are IO-bound and others are processor-bound, if the IO-bound tasks are associated with users sitting at terminals, it would be exceptionally disruptive if a 5-second processor bound task started running. Round-robin (RR) essentially says that a task is allowed to run at most *n* units of time (called a *time slice*) before it is pre-empted and placed back into the ready queue. Each time a task is scheduled, a timing interrupt is set at the appropriate time in the future, and if the task yields the processor early, the associated timing interrupt is cancelled, a new task and a new timing interrupt are scheduled.

The implementation is quite straight-forward: just prior to the scheduler switching to the next task to be scheduled, the scheduler arranges for a timing interrupt in *n* units of time. If the task uses its entire time interval, it's state is set to READY and it is placed back into the ready queue (unless there is nothing else to run, in which case, nothing is done). If the task finishes before the interrupt (either because it exits or it is waiting on a resource), the scheduler resets the time of the next timing interrupt. Such a scheduler would run in $\Theta(1)$ time.

We cannot, however, use this round robin for real-time systems, as it does not take into account any deadlines. It is, however, are a basis for the schedulers used in general-purpose systems where concepts such as fairness are relatively important—indeed, the name of the current scheduler for Linux is the *Completely Fair Scheduler* by Ingo Molnár.

Aside: The Completely Fair Scheduler (CFS) is not $\Theta(1)$, unlike the scheduler it replaced; however, the previous $\Theta(1)$ scheduler did replace another scheduler that was $O(n)$. The CFS uses a red-black tree where tasks are inserted based on a linear ordering of time spent executing. The left-most node is therefore that task that has spent the least time executing and will be the task scheduled next. When it finishes execution, it will be reinserted into the tree with its updated execution time—something that is likely to place it elsewhere into the tree. Note that interactive applications (editing code, for example) often spend a lot of time waiting for the user to strike a key, and so while they have very short execution times, they don't use the processor that much, either, so other longer-executing tasks will have an opportunity to execute.

7.4.4 Deadline-based preemptive scheduling

We will now look at two algorithms that are optimal in the sense: if tasks can be scheduled in such a way that it is possible for all tasks to meet their deadlines, then both of these will find such a schedule:

1. earliest-deadline first, and
2. least-slack first.

We will describe both of these, but will also note that while being *optimal*, they are not usually used in practice.

7.4.4.1 Earliest-deadline first

As we described previously, we could adapt the earliest-deadline first algorithm to work with periodic tasks. This will, however, require timing interrupts, as tasks may become ready to execute at the start of their next period. Thus, at any time, the task that is executing is that task that is both ready and has the earliest deadline. This algorithm is optimal in the sense that if tasks can be scheduled in such a way for all tasks to meet their deadlines, this scheduler will find such a schedule.

Proof:

We will restrict our proof to periodic tasks, but we can also consider sporadic tasks as being periodic. We will assume that all tasks started running at time $t = 0$. Suppose that tasks $(c_1, \tau_1), \ldots, (c_n, \tau_n)$ have a utilization $\sum_{k=1}^{n} \frac{c_k}{\tau_k} \leq 1$ but EDF fails to schedule them. Thus, there must be some task $(c_{\text{crit}}, \tau_{\text{crit}})$ that first misses its period at time P. Therefore, all other tasks up until this point have earlier deadlines and thus each task has run m_k times where $m_k \tau_k \leq P$ but $(m_k + 1)\tau_k \geq P$. Specifically, $m_{\text{crit}} \tau_{\text{crit}} = P$. Thus, $\sum_{k=1}^{n} m_k c_k > P$, and therefore

$$\sum_{k=1}^{n} \frac{m_k c_k}{P} > 1.$$

Now, as $m_k \tau_k \leq P$, $\frac{m_k c_k}{P} \leq \frac{m_k c_k}{m_k \tau_k} = \frac{c_k}{\tau_k}$, and thus we would conclude that $\sum_{k=1}^{n} \frac{c_k}{\tau_k} > 1$, which is a contradiction. ■

When a task is scheduled, the scheduler will have to determine whether or not prior to the task's worst-case computation time that another task will become ready (its next period starts) and that other task has an earlier deadline. If so, the scheduler will have to schedule an interrupt.

If during execution, a new periodic or sporadic task is created, after the infrastructure for the task is completed, the scheduler will have to determine whether or not the new task has an earlier deadline than the currently executing task. If so, rather than returning to the currently executing task, the scheduler will have to

If a task, for whatever reason, is known not to be able to meet its deadline (the deadline is in d ms, but the remaining worst-case computation time is still $c > d$, we have two choices:

1. if the task is soft-real time, we can still schedule it, but
2. if the task is firm or hard real-time, there is no point in scheduling it, so instead remove that task and allow other tasks to execute.

7.4.4.2 Least-slack first algorithm

Like EDF, we can also schedule periodic and sporadic tasks using least-slack first (LSF), and like EDF, it is optimal in the sense that if $\sum_{k=1}^{n} \frac{C_k}{\tau_k} \leq 1$ then LSF will schedule the tasks. Thus, the only difference may occur if a short-running task with an earlier deadline may be executed after a longer-running task with a later deadline so long as the later task has less *slack* time prior to its deadline. Hwang, Choi and Kim demonstrate that this algorithm is better under certain conditions than earliest-deadline first when dealing with multiple processing units.

7.4.4.3 Implementation of deadline scheduling algorithms

The FCFS scheduling algorithm uses a simple queue, while the shortest-job next scheduling algorithm would use a priority queue, where the priority is inversely proportional to the computation time (the shorter the run time, the higher the priority). The earliest-deadline first algorithm would also use a *ready priority queue*, but now the task at the top of the heap is that task is the task with the earliest deadline. However, we require a little more effort: when a periodic task finishes its execution, it must decide whether:

1. it is immediately ready to execute again, or
2. the current period is not yet finished.

In the former case, the task can be immediately enqueue it back on the priority queue. In the latter, we must instead have a second priority queue: one that stores those tasks, the periods of which have not yet begun. We will call this one the *waiting priority queue*. Now, before we switch to the next task to be executed from the ready priority queue, we must schedule a timing interrupt from the real-time clock. The time of that interrupt will be the time the task on top of the waiting priority queue becomes ready.

When the timing interrupt occurs, the interrupt service routine for the real-time clock will do the following:

1. it will continue to check to top of the waiting priority queue and pop all tasks that are now ready to run—these tasks will be placed onto the ready priority queue, and
2. it then compares the deadline of the interrupted task and the task at the front of the ready priority queue:
    a. if the interrupted task still has the earliest deadline, return to its execution, otherwise
    b. if the task at the front of the ready priority queue has a deadline earlier than the interrupted task, set the current task to READY, enqueue it on the ready priority queue, and then switch to the task currently at the top of the ready priority queue.

Thus, at all times, the task that is running is that task with the earliest deadline. Note that in our analyses, we often ignore the overhead of this additional work. Unfortunately, if every task completes before the start of the next period, this will require as many timing interrupts as there are scheduled tasks to execute.

Note that if the task at the top of the waiting priority queue is a hard or firm real-time task and it is observed that it will miss its deadline, the scheduler can decide whether or not to schedule that task.

7.4.4.4 Optimal schedulers and periodic tasks with overloads

Suppose that tasks are being scheduled with either earliest-deadline first (EDF) or least-slack first (LSF) in a system that is periodic. In this case, it is entirely possible to schedule all of the tasks using earliest-deadline first in such a way to ensure all tasks execute so long as the utilization does not exceed 1.

What happens if the system is overloaded? Suppose that the utilization $U > 1$. In this case, earliest-deadline first makes no guarantees that any of the tasks will be scheduled so as to meet their deadline; however, which deadline is passed cannot be determined apriori. In fact, in the long term, if a system remains overloaded with a utilization $U > 1$, then each task will be scheduled, on average, as if it had a new period $\tau_k' = U\tau_k$ (Cervin et al., 2002) . Thus, if one task was

expected to run as (3, 16) and the utilization was 1.125, the long term behaviour of the task would be more as if it was (3, 18); however, we cannot predict in advance which tasks will meet and which will miss their deadlines.

7.4.4.5 Real-time issues with optimal deadline scheduling algorithms

While both these can schedule tasks in such a way as to satisfy all deadlines if the tasks are schedulable, this may not always be the case. Consider a system which has periodic tasks, each with computation times and periods; however, there are also *sporadic* tasks that must be started in response to the real-time environment. When such sporadic tasks appear, the system may become temporarily overloaded, in which case some tasks will miss their deadlines. Unfortunately, there is no way of determining *a priori* which tasks will miss their deadlines. Some tasks may have hard deadlines, while others have either firm or soft deadlines, in which case, it would be most desirable to ensure that those with hard deadlines are scheduled while tasks with soft or even firm deadlines are the ones that are missed. We will introduce a concept of priority in the next section as a mechanism for scheduling tasks.

7.4.4.6 Summary of optimal deadline scheduling algorithms

Optimal deadline scheduling algorithms always ensure that the highest priority task is executing; however, if there is an overload, there is no means of determining which task will miss its deadline. To solve this problem we will next consider priority-based scheduling.

7.4.5 Priority-based preemptive scheduling algorithms

To solve some of the issues with deadline scheduling, we will consider a different approach to scheduling: using priorities. We will define priorities and consider data structures that can be used to ensure that the highest priority task is the one that is scheduled next. We will look at two algorithms that use priorities:

1. rate-monotonic (RM) and
2. deadline-monotonic (DM)

scheduling. We will then discuss the issue of jitter with respect to these two scheduling algorithms, together with the possibility that it may not be possible to exactly prescribe priorities apriori in a dynamic system.

7.4.5.1 Priorities

An alternate approach to scheduling is to assign each task a priority, in which case, in a real-time system, the task that is running is the task that can run that has the highest priority. In general, priorities will be represented by integer values, usually restricted to a range, say 0 to 255 or –20 to 20, where lower numbers represent higher priorities than larger numbers.

We will consider

1. a classification of how priorities are assigned, and
2. the implementation of priority schedulers.

Note: Previously, we discussed fair algorithms. With priority scheduling, if there are multiple tasks at the same priority, a simple fair algorithm may be most appropriate for choosing which task to implement next. Suppose, for example, we use FCFS. If one of the two should be necessarily running before the other, then we have an issue with the design: the priorities were not set correctly.

7.4.5.1.1 Classification of priorities: fixed and dynamic

In a real-time system, priorities can be either *fixed* or *dynamic*:

1. A fixed priority is one that is determined at during the design phase, either explicitly chosen or the result of an algorithm, and
2. A dynamic priority is one that can be changed at run time.

We will look at rate monotonic and deadline monotonic scheduling algorithms, both of which assign fixed priorities to the tasks, and then we will see how earliest-deadline first can be implemented using dynamically changing priorities.

7.4.5.1.2 Implementation of priority schedulers

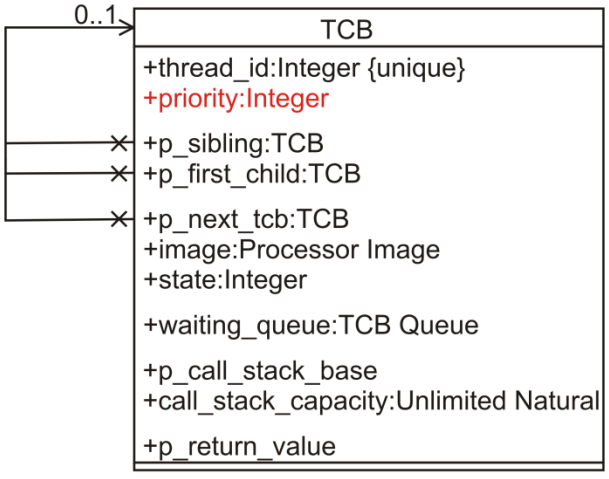
Almost all real-time systems use priorities to distinguish which tasks should be run. The RTX real-time operating system stores each task's priority in the task control block:

```
typedef struct OS_TCB {
    /* General part: identical for all implementations.                    */
    U8     cb_type;                 /* Control Block Type                  */
    U8     state;                   /* Task state                          */
    U8     prio;                    /* Execution priority                  */
    U8     task_id;                 /* Task ID value for optimized TCB access  */
    struct OS_TCB *p_lnk;           /* Link pointer for ready/sem. wait list   */
    struct OS_TCB *p_rlnk;          /* Link pointer for sem./mbx lst backwards */
    // ...
} *P_TCB;
```

As you can observe, the priority is an eight-bit unsigned integer, and therefore allows priorities from 0 to 255; however, users are restricted to priorities from 1 to 254, with 0 reserved for critical operating system tasks and 255 reserved for the idle task.

For our data structure, we could add a similar field:

```
typedef struct tcb {
    tid_t thread_id;
    int8_t priority;

    struct tcb *p_sibling;
    struct tcb *p_first_child;

    struct tcb *p_next_tcb;
    processor_image_t image;
    int8_t state;

    tcb_queue_t waiting_queue;

    void *p_call_stack_base;
    size_t call_stack_capacity;

    void *p_return_value;
} tcb_t;
```



We will consider two implementations:

1. using an array of queues, and
2. using a heap-based priority queue

and then we will consider some issues with using heap-based structures.

7.4.5.1.2.1 Using an array of queues

Suppose we have an array of m queues, one for each of m priorities from 0 to $m - 1$ (we will assume $m \leq 32$). In this case, we enqueue each task in the queue associated with its priority. In tracking the non-empty queues, we will use a 4-byte bit-vector, where each bit is set to 1 if the corresponding queue is non-empty with the highest priority queue being associated with the most significant bit:

```
tcb_queue_t ready_queues[NUM_PRIORITIES];
uint32_t non_empty_queue_bit_vector;

void init_ready_queues() {
    int i;

    for ( i = 0; i < NUM_PRIORITIES - 1; ++i ) {
        TCB_QUEUE_INIT( task_queue[i] );
    }

    // Enqueue the idle task
    TCB_QUEUE_INIT_AND_ENQUEUE( task_queue[NUM_PRIORITIES - 1], p_idle_tcb );
    non_empty_queue_bit_vector = 1 << (NUM_PRIORITIES - 1);
}
```

When we enqueue a new task, we simply enqueue the task on the queue corresponding to the priority of the task. We also set the corresponding bit of the non-empty queue bit-vector to 1:

```
void push_ready_queues( tcb_t *p_tcb ) {
    p_tcb->p_next_tcb = NULL;

    // Enqueue the task and set the corresponding bit to 1
    TCB_ENQUEUE( ready_queues[p_tcb->priority], p_tcb );
    non_empty_queue_bit_vector |= 1 << p_tcb->priority;
}
```

To dequeue, your first thought that this becomes an O(m) problem, as we must search for a non-empty queue; however, on many processors, there is a machine instruction that returns the number of zeros following the last bit set to 1; for example, the CTZ (count trailing zeros) instruction on ARM. Because the idle task is always ready if it is not running, there will be at least one non-zero bit in the bit-vector. After the appropriate task is dequeued and the queue is empty, the corresponding bit is cleared:

```
tcb_t *pop_ready_queues() {
    size_t top_priority = __builtin_ctz( non_empty_queue_bit_vector ); // gcc specific

    assert( top_priority < NUM_PRIORITIES );

    tcb_t *p_popped_tcb = TCB_QUEUE_FRONT( ready_queues[top_priority] );
    TCB_DEQUEUE( ready_queues[top_priority] );

    if ( TCB_QUEUE_EMPTY( task_queue[top_priority] ) ) {
        non_empty_queue_bit_vector &= ~(1 << p_tcb->priority);
    }

    return p_popped_tcb;
}
```

Note that by always requiring the existence of an idle task removes many edge cases that must normally be checked in a queue data structure need not be dealt with here; for example, the queue is never empty. Even if a count trailing zeros instruction is not available, it is possible to do this reasonably efficiently in software, as demonstrated at the Bit Twiddling Hacks website: https://graphics.stanford.edu/~seander/bithacks.html. If the number of priorites exceed 32, the above approach can be modified but becomes more cumbersome; for such a situation, we will look at another approach using binary min heaps.

7.4.5.1.2.2 Using a heap structure

One reasonably fast implementation of priorities is through the use of a heap structure. Recall previously in our discussion on best-fit and worst-fit memory allocations that a leftist heap allowed us to efficiently store data in linear order, although we could also use a binary min-heap. The latter would require more memory Now the scheduling time is O(ln(*n*)) where *n* is the number of tasks in the ready queue.

7.4.5.1.2.3 Using heaps with lexicographical ordering

One significant issue with heap structures is that neither a binary min-heap nor a leftist min-heap guarantees FCFS ordering. For example, suppose tasks 1, 2, 3, …, all at the same priority, arrive in sequence. After the third task is enqueued, there is an alternating sequence of dequeues and enqueues. In this case, the sequence of tasks dequeued is 1, 3, 4, 5, 6, …, and Task 2 is never given an opportunity to execute. Now, while such a possibility is significantly diminishing, as it requires the operations to occur in alternating order, it is still an issue with respect to a real-time system: a task at the current highest level of priority may never be scheduled to execute: that is, it becomes starved.

A solution may be derived from a dictionary: the concept of a lexicographical ordering. In this ordering, any word starting with 'a' comes before any word starting with 'b', 'c', or any other character. Words are lexicographically ordered.

> lexicon, *n*, a word-book or dictionary; chiefly applied to a dictionary of Greek, Hebrew, Syriac, or Arabic.
>
> Oxford English Dictionary (OED at www.oed.com)

The characteristics of a lexicographical ordering is that each character, in order, is linearly ordered: 'a' < 'b' < 'c' < ⋯ < 'z'. We could apply this to, for example, pairs of linearly ordered items. For example, we could define a pair of integers $(m_1, m_2) < (n_1, n_2)$ if

1. $m_1 < n_1$, or
2. $m_1 = n_1$ and $m_2 < n_2$.

For example, (1, 2) < (1, 3) < (2, 1) < (2, 7) < (3, 4), *etc*. The only subtle difference between integers and letters of the alphabet is that we cannot write down, for example, all two-letter "words" that use integers instead of characters.

Thus, we could have a global counter *n* that is incremented each time a task is enqueued into the scheduler, and the priority of the task with priority *p* becomes the hybrid lexicographical priority (*p*, *n*). Thus, in the above scenario, assuming that all tasks had priority 3, as they are enqueued, they take on the lexicographical priority (3, 1), (3, 2), (3, 3), *etc*. and thus they will be dequeued in the exact same order in which they are enqueued.

7.4.5.1.3 Summary of priorities

We have described the concept of a priority and considered how to create a scheduler that can deal with priorities. Next, we will consider rate-monotonic scheduling.

The use of priorities to schedule tasks is exceptionally prevalent in real-time operating systems—most schedulers will use a priority without any other consideration. Later, we will see how priorities can lead to certain consequences (deadlock and starvation), so while it is very useful from the point-of-view of scheduling, it does not come without additional issues.

7.4.5.2 Rate-monotonic scheduling

This section will introduce fixed-priority rate-monotonic (RM) scheduling, describe a situation where the scheduler fails, and then considers a formula for determining when a set of periodic tasks is schedulable. Next, we will review an older-but-common formula found in many text books. We will conclude by seeing that when all periods are mutually harmonic, RM scheduling is as optimal as earliest-deadline first. We will also consider how the algorithm works when we allow over overloads.

7.4.5.2.1 Description of rate-monotonic scheduling

One serious issue with earliest-deadline-first as a means of scheduling periodic tasks is that almost all real-time operating systems are priority based. Consequently, implementing an earliest-deadline first scheduler using priorities very quickly becomes prohibitive. For example, suppose that two tasks ready to execute currently have priorities 7 and 8, respectively, as the first task has an earlier deadline than the second. If prior to either being scheduled, another task becomes ready with an intermediate deadline, it will be necessary to adjust the priority of at least one of the two scheduled tasks. Thus, such an operation is O(*n*) in the number of tasks. Additionally, as tasks are ready to execute, on average, they will have later deadlines than existing tasks, resulting in a cycling through of possible priorities. If the number of priorities is relatively small (as with the RTX real-time operating system—only 254), such adjustments will be necessary on a regular basis. Consequently, it is necessary to come up with a scheduling algorithm that can make use of the priority scheduling available in most operating systems.

Suppose we want to dynamically schedule *n* periodic tasks (c_k, τ_k) so that all the tasks meet their deadlines. An alternate scheduler is the rate-monotonic scheduler that requires that at any time, of all tasks that could be executed, the one that is executing is the one with the shortest period. Once that task has finished execution, execute the next task that is ready to execute with the next shortest period. By this definition, this is a fixed scheduler as the periods are already known apriori. The means of translating this scheme into a priority scheduler is by simply assigning priorities corresponding to the periods. Thus, given the three tasks (1, 4), (3, 7), and (3, 10), the first would have highest priority and the latter would have the lowest priority. An attempt to use this schedule is shown in Figure 7-25.

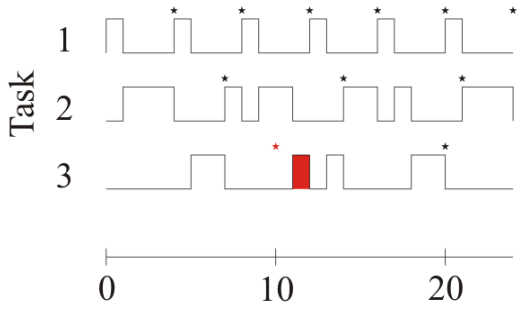


Figure 7-25. Scheduling three tasks using rate-monotonic scheduling.

As is highlighted, we quickly note that Task 3 fails to meet its deadline, a problem that could easily have been resolved if Task 1 had completed prior to starting the execution of the second cycle of Task 2—something which would have happened had we been using earliest-deadline first, as the utilization, *U*, is less than one:

$$\frac{1}{4}+\frac{3}{7}+\frac{3}{10}=\frac{137}{140}\leq 1.$$

7.4.5.2.2 Determining schedulability

As a result of the previous example, it is clear that not all schedulable periodic tasks are schedulable with rate-monotonic scheduling. Currently, there is no real-time test that can determine whether or not a collection of tasks is schedulable using rate-monotonic scheduling. Thus, we must rely on tests that may occasionally give an incorrect answer. In statistics, a test that is not always correct may have two *types* of errors:

1. a type I error (a false positive), indicating that a collection of tasks is schedulable when it is in fact not, and
2. a type II error (a *false negative*), indicating that a collection of tasks is not schedulable when it is, never-the-less, schedulable.

Any real-time formula for schedulability must

1. be easy to calculate ($\Theta(1)$ if it is being determined whether or not a new task can be include in a set of already schedulable tasks),
2. never result in a type I error (that is, a false positive) as this may result in missed deadlines, and
3. minimize the number of type II errors (that is, reject as few as possible collections of tasks that are RM schedulable.

One such formula for which we will offer a proof is a collection of *n* tasks is schedulable if

$$\prod_{k=1}^{n}\left(1+\frac{c_k}{\tau_k}\right)\le 2\,.$$

Proof:

Without loss of generality, assume that the period of the shortest task is $\tau_1 = 1$, and for other tasks, assume that the period is the period of the previous task plus the computation time of that previous task, so

$$\tau_k = \tau_{k-1} + c_{k-1} \text{ or } \tau_n = 1 + \sum_{k=1}^{n-1} c_k\ .$$

This represents the worst-case scenario, where no task will be able to execute more than once and where the utilization is minimized. Thus, we have that all tasks must execute before the end of the period of the first task, or

$$\sum_{k=1}^{n} c_k \le 1,$$

as no task will have an opportunity to execute if this sum exceeds unity. Now let us look at the product

$$\prod_{k=1}^{n}\left(1+\frac{c_k}{\tau_{k-1}+c_{k-1}}\right) = \left(1+c_1\right)\left(1+\frac{c_2}{1+c_1}\right)\left(1+\frac{c_3}{1+c_1+c_2}\right)\left(1+\frac{c_4}{1+c_1+c_2+c_3}\right)\cdots\left(1+\frac{c_n}{1+c_1+\cdots+c_{n-1}}\right)$$

If you begin multiplying out successive terms in the product, you get

$$\begin{aligned}\prod_{k=1}^{n}\left(1+\frac{c_k}{\tau_{k-1}+c_{k-1}}\right) &= \left(1+c_1+c_2\right)\left(1+\frac{c_3}{1+c_1+c_2}\right)\left(1+\frac{c_4}{1+c_1+c_2+c_3}\right)\cdots\left(1+\frac{c_n}{1+c_1+\cdots+c_{n-1}}\right)\\ &= \left(1+c_1+c_2+c_3\right)\left(1+\frac{c_4}{1+c_1+c_2+c_3}\right)\cdots\left(1+\frac{c_n}{1+c_1+\cdots+c_{n-1}}\right)\end{aligned}$$

and thus, by induction, we

$$\prod_{k=1}^{n}\left(1+\frac{c_k}{\tau_{k-1}+c_{k-1}}\right) = 1+\sum_{k=1}^{n} c_k\ ,$$

but by assumption, this cannot exceed unity, and thus, $\prod_{k=1}^{n}\left(1+\frac{c_k}{\tau_k}\right)\le 2$. ■

A schedulability test for any fixed-priority algorithm exists, but it cannot in any sense be computed in real time. Interested readers can look at M. Joseph and P. Pandya's article "Finding response times in a real-time system" in the Computer Journal, 29(5) on pp.390-5, 1986.

In a dynamic system, where periodic tasks occasionally start or finish execution, it is now possible to determine whether or not a new task will be schedulable. If the new tasks increases this product to a value greater than 2, that task is rejected.

The worst-case scenario described above suggests how we can most easily go about scheduling tasks using RM scheduling by hand:

1. First order the tasks so that $\tau_1 \le \tau_2 \le \cdots \le \tau_n$.
2. Then, starting with the first, draw the required number of periods and schedule that task at the start of each period. It has the highest priority, so when it can run, it must run.
3. Then, for each successive task, draw the required number of periods and schedule the corresponding task to run as early as you can during each period. If there is any overrun, mark it accordingly.

7.4.5.2.3 An older formula

When RM scheduling was first proposed (Liu and Layland, 1973), it was determined that the tasks are schedulable if

$$U=\sum_{k=1}^{n}\frac{c_k}{\tau_k}\le n\left(2^{1/n}-1\right).$$

This formula is less desirable than the formula presented above. For example, when n = 2, the tasks are only guaranteed to be schedulable if $\frac{c_1}{\tau_1}+\frac{c_2}{\tau_2}\le 2\left(\sqrt{2}-1\right)\approx 0.828$, or if the system remains below 82.8 % utilization. This is a consequence of the plot of the worst-case utilization $\frac{c_1}{\tau_1}+\frac{\tau_1-c_1}{\tau_1+c_1}$ as c_1 varies from 0 to τ_1, shown in .

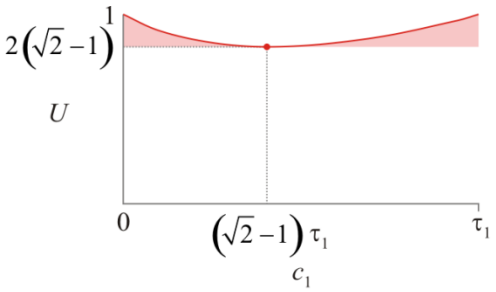


Figure 7-26: Best-case utilization in the worst-case scenario for two tasks.

The minimum occurs when $c_1=\left(\sqrt{2}-1\right)\tau_1$, and thus this formula will reject new tasks even if they are RM schedulable, for example, tasks (1, 11) and (9, 12) are RM schedulable, and $\left(1+\frac{1}{11}\right)\left(1+\frac{9}{12}\right)=\frac{21}{11}\le 2$, and yet

$\frac{1}{11}+\frac{9}{12}=\frac{37}{44}=0.84\overline{09}>2\left(\sqrt{2}-1\right)\approx 0.8284$, thus this second formula would fail to identify this pair of tasks as being RM schedulable. In the example in Section 7.4.5.2.1, $\frac{1}{4}+\frac{3}{7}+\frac{3}{10}=\frac{137}{140}\approx 0.979\gg 3\left(\sqrt[3]{2}-1\right)\approx 0.779$.

As the number of tasks grows, we should consider what happens to the right-hand side of the formula. Using L'Hôpital's rule, we have

$$\begin{aligned}\lim_{n\to\infty} n\left(2^{1/n}-1\right) &= \lim_{n\to\infty}\frac{2^{1/n}-1}{\frac{1}{n}}\\ &= \lim_{n\to\infty}\frac{\frac{d}{dn}\left(2^{1/n}-1\right)}{\frac{d}{dn}\frac{1}{n}}\\ &= \lim_{n\to\infty}\frac{-\frac{2^{1/n}\ln(2)}{n^2}}{-\frac{1}{n^2}},\\ &= \lim_{n\to\infty} 2^{1/n}\ln(2)\\ &= \ln(2)\approx 0.693\end{aligned}$$

so for any system with a large number of tasks, a utilization no greater than 69 % guarantees the tasks are schedulable using rate-monotonic scheduling. As a general rule, most schedulers do not try to calculate the right-hand side and instead simply use ln(2). The appendix shows an algorithm that can reasonably approximate the correct algorithm with 16-bit fixed-point precision. In Figure 7-27, for a pair of the benefits of the previously described formula become very apparent.

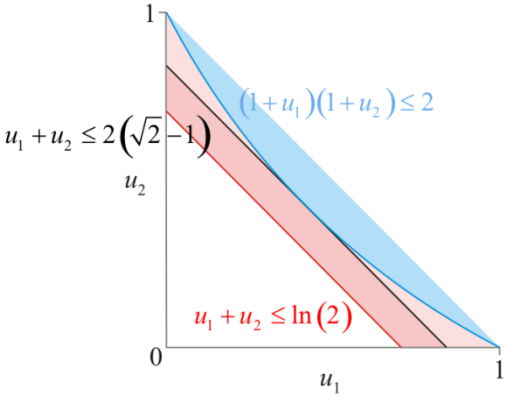


Figure 7-27: Given two utilizations, the blue region indicates pairs that can be scheduled using earliest-deadline first. The light-blue curve shows all pairs of tasks that the multiplicative method flags as schedulable, the black line indicates tasks that are accepted by the additive formula, and the light-pink region are those pairs accepted by the multiplicative formula but
rejected by the additive formula, and the dark pink region are those pairs rejected by the simplifed $u_1 + u_2 \le \ln(2)$.

The same image for three utilizations is shown in Figure 7-28. Here, a quadrant shows all tasks schedulable with earliest-deadline first, a light-blue convex surface shows all tasks that would be accepted by our multiplicative formula, the dark

surface below that includes all that have a utilization less than $3\left(\sqrt[3]{2}-1\right)$, and the red surface shows those tasks with a utilization less than ln(2).

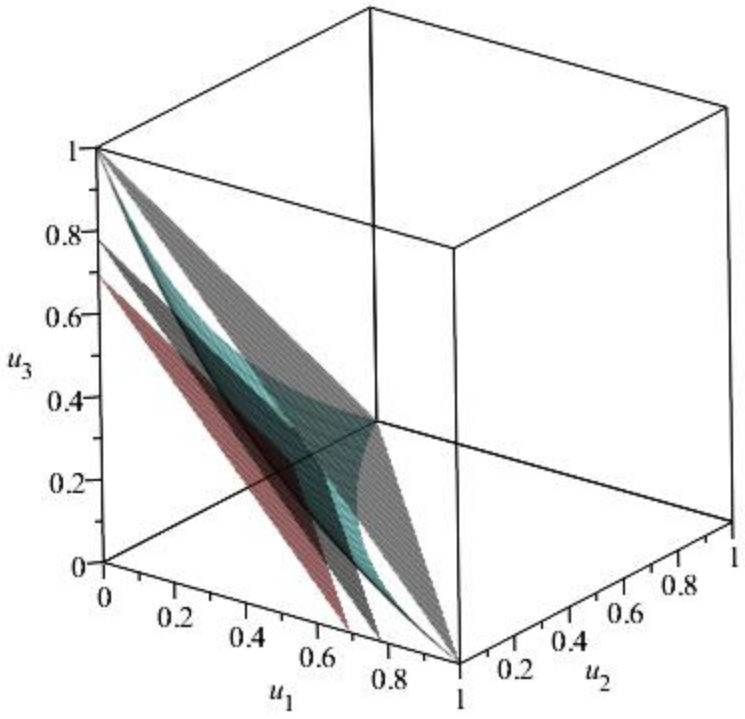


Figure 7-28: Given three utilizations, the upper light-grey plane marks the quadrant of all tasks that can be scheduled using earliest-deadline first, while the blue surface shows all triplets of tasks flagged as schedulable by the multiplicative formula. Below that a darker surface bounding those triplets flagged as schedulable by the additive formula, and below that are those triplets where the utilization does not exceed ln(2).

Next, we will look at a specific situation where RM scheduling will schedule any collection of tasks so long as the utilization does not exceed 1.

7.4.5.2.4 Mutually harmonic periods

As an observation, it was determined that RM scheduling can still, with reasonable success, schedule tasks so long as the utilization does not exceed 88 % (Lehoczky et al., 1989); however, this is not appropriate for hard deadlines.

If all periods of the tasks are in a harmonic relation with each other (that is, they are *pairwise harmonic*), then RM will schedule them up to 100 % utilization. Two periods are said to be *in harmonic relation* if one is an integer multiple of the other. For example, (1, 3), (2, 6) and (4, 12) are three tasks with periods that are in harmonic relation with each other and therefore RM will schedule these despite the utilization being 1, as shown in Figure 7-29.

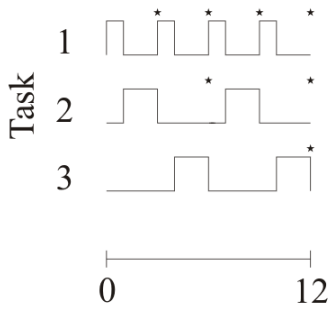


Figure 7-29. Three tasks with utilization 1 but with mutually harmonic periods. The stars indicate the start of periods.

Proof:

Assume that all tasks have periods that are mutually harmonic and that the sum of the utilizations does not exceed unity. First, order the different periods $\tau_1 < \tau_2 < \cdots < \tau_m$ and let $c_{k,1}, c_{k,2}, \ldots, c_{k,m_k}$ be the computation times of the tasks with period τ_k. Over a period of τ_k, all tasks with shorter intervals have higher priority, and therefore their utilization must be exactly

$$u_{k-1} = \sum_{j=1}^{k-1} \frac{\sum_{i=1}^{k_j} c_{j,i}}{\tau_j}.$$

As all tasks with period τ_k now must be able to execute, as they have the highest priority, as the total utilization $U \leq 1$. If they were not able to execute, this would imply that $U > 1$, which contradicts our assumption. ■

In real systems, there is not necessarily a *best possible* period for the cycles required by a controller. Consequently, it makes sense to ensure the periods are in harmonic relation to each other to ensure that all deadlines will be satisfied.

Aside: Why the term *harmonic relation* instead of *integer multiples*? This comes from *harmony* in music. Consider the image in Figure 7-30.

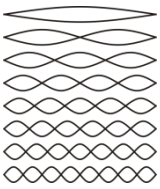


Figure 7-30. Seven harmonics of a base frequency.

Note that all pairs of periods must be harmonic—it is not enough that the shorter periods are harmonic with a larger period. Consider, for example the four tasks (5, 24), (7, 30), (21, 40) and (2, 60) with utilization one where all periods are harmonic with one of length 120. RM scheduling, however, fails, as can be seen in Figure 7-31.

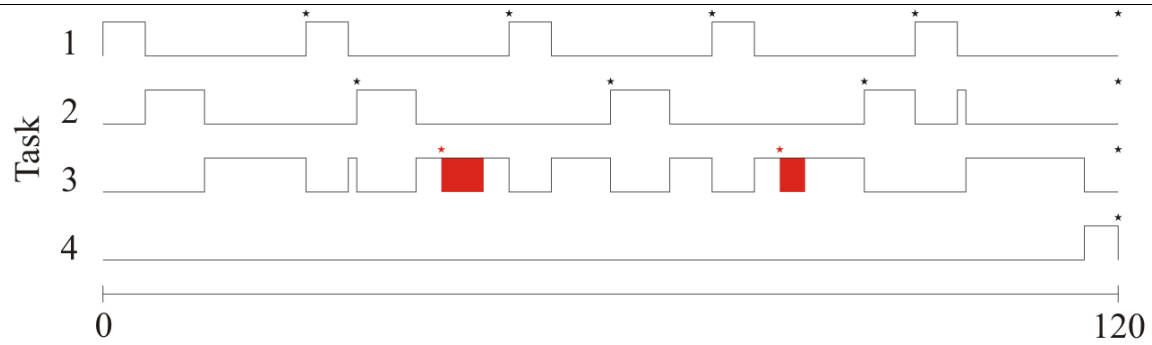


Figure 7-31. Missed deadlines with RM scheduling when periods are not mutually harmonic. Stars denote the beginning of periods.

Note: A collection of n tasks is pairwise harmonic if there is a sequence of integers $m_1, m_2, \ldots, m_M$ such that for each task, there is an integer i_k such that the period of $\tau_k = \sum_{k=1}^{i_k} m_k$. For example, the sequence of integers associated with the periods 5, 10, 30, 120 and 360 is 5, 2, 3, 4, and 3. You can visualize the harmonics in the image shown in Figure 7-32.

Figure 7-32. Mutually harmonic tasks with periods 5, 10, 30, 120 and 360.

7.4.5.2.5 RM scheduling with overloads

With rate-monotonic scheduling, if an overload occurs, the task with the lowest priority, that is, the highest period, is likely to miss its deadline. In the worst case, the longer-period tasks may simply never be scheduled leading to *starvation*—that is, the tasks are never scheduled.

Recall that with earliest-deadline first (EDF) scheduling, in an overloaded system, on average, the tasks simply appear to execute as if they were on a larger period.

The justification for using RM over EDF is that most operating systems have schedulers that are strictly priority based. Consequently, there is significantly less overhead with RM.

7.4.5.2.6 Summary of rate-monotonic scheduling

Rate-monotonic scheduling is a reasonable scheduling algorithm where the priority is inversely proportional to the period. Unlike earliest-deadline first scheduling, it does not guarantee that the tasks are schedulable if the processor utilization is too high; however, it is a simple algorithm and often there will be more than enough cycles available to allow this algorithm to be used.

Next, we will look at

7.4.5.3 Deadline-monotonic scheduling

RM scheduling only works if the deadlines match the end of the period, but if the deadlines are scheduled prior to the end of the period, RM scheduling may fail, as is demonstrated by the case (2, 4, 4) and (1, 2, 5), where even though the utilization is $U = 0.5 + 0.2 = 0.7$, the first task is immediately scheduled, and by the time the second task is scheduled, it has missed its deadline. If the priorities were reversed, however, it would be clear that both tasks meet all their deadlines, as shown in Figure 7-33.

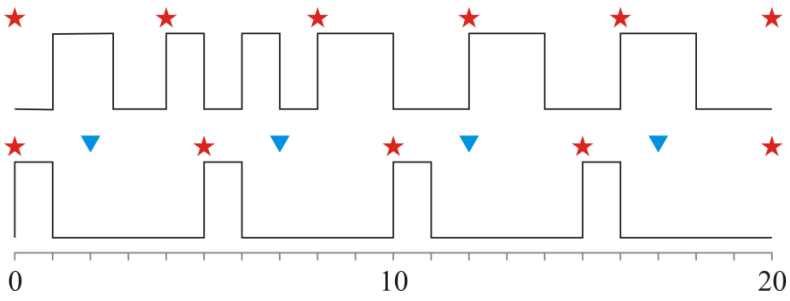


Figure 7-33. Two tasks (2, 4, 4) and (1, 2, 5) with higher priority given to the second task.

This leads to a second scheduling algorithm, known as deadline-monotonic (DM) scheduling. Priority is assigned based on the deadlines, as opposed to being assigned on the period. Because processor utilization can only increase as the

deadlines approach the periods, as a consequence, the formulas that determine schedulability with rm scheduling also determine whether a set of tasks with deadlines is schedulable, that is

$$\prod_{k=1}^{n}\left(1+\frac{c_k}{d_k}\right)\leq 2\ .$$

Like RM scheduling, DM scheduling is an optimal fixed-priority scheduling algorithm—no other algorithm that fixes priorities can do better.

To give a brief example, consider the following tasks: (2, 4, 5), (1, 9, 10) , (2, 10, 15). In this case, the product

$$\left(1+\frac{2}{4}\right)\left(1+\frac{1}{9}\right)\left(1+\frac{2}{10}\right)=2\,,$$

and thus the tasks are schedulable using DM. If we schedule them over the super period, we see that the tasks are easily schedulable, as shown in Figure 7-34.

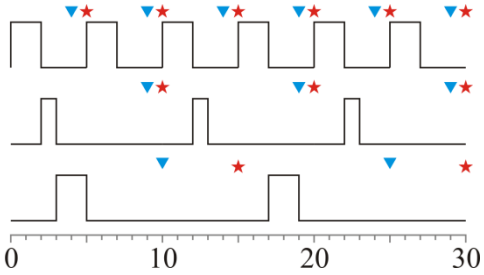


Figure 7-34. The scheduling of three tasks (2, 4, 5), (1, 9, 10) , (2, 10, 15).

Unfortunately, the actual utilization is significantly lower:

$$\frac{2}{5}+\frac{1}{10}+\frac{2}{15}=\frac{19}{30}\,.$$

Unfortunately, this is the cost of having earlier deadlines. Consider the three tasks (1, 2, 20), (1, 2, 20) and (2, 3, 20). These three tasks are simply not schedulable, even though the processor utilization would 20 %. In this scenario, however, a the tasks could be schedulable if either one of the tasks has its deadline sufficiently relaxed or if the processor speed was increased by a factor of 33 % (the multiplicative schedulability test would not indicate this, but a quick look at the numbers indicates that they are schedulable.

7.4.5.4 Dealing with jitter

With RM and DM scheduling, the highest-priority tasks will execute periodically; however, lower priority tasks will not have that luxury—at any time, they may be interrupted or delayed as a result of a the execution of a higher-priority task.[19] Consequently, the lower the priority, the greater the uncertainty (and thus higher *jitter*) associated with the execution of such tasks. Suppose, however, with such a task there is an overwhelming requirement to, for example, update in a periodic manner, a memory location storing the output of that particular execution or to transfer information to memoryless actuator that requires periodic signals. If the task itself has a long period, the uncertainty could be an unsolvable problem, with separate responses arriving as distantly from as little time as $2c$ to as much time as $2\tau - c$.

[19] This section is based on comments in the TimeSys manual *The Concise Handbook of Real-Time Systems*, version 1.3, 2002, p.41

In order to deal with such a case, one common solution is to split the lower-priority Task A into two separate tasks: the first, Task A, does the computation, while the second shadowing task, Task A′, has the exact same period, but has an artificially high priority—possibly higher than even the task with the shortest period. In this case, Task A′ simply copies the result from, or passes the result on from the execution of Task A in the previous period.

While, in general, such a shadowing task will have a very short computation time *c′*, but never-the-less, it may delay the execution of another shorter-period (and therefore higher priority) task. Consequently, the algorithms for determining schedulability of tasks using RM may fail—a collection of tasks may be RM schedulable, but the additional delay caused by an interruption by the shadowing task may cause a higher priority task to miss its deadline. The most straight-forward means of correcting for this is to increase the computation time of each higher priority task by *c′*. As the runtime of such shadowing tasks should be relatively brief, this may not have that significant an affect. The task and its shadow task can be considered to be a single task with their combined computations.

An example of a how a high priority task and a lower priority task together with a shadowing task may interact is shown in Figure 7-35.

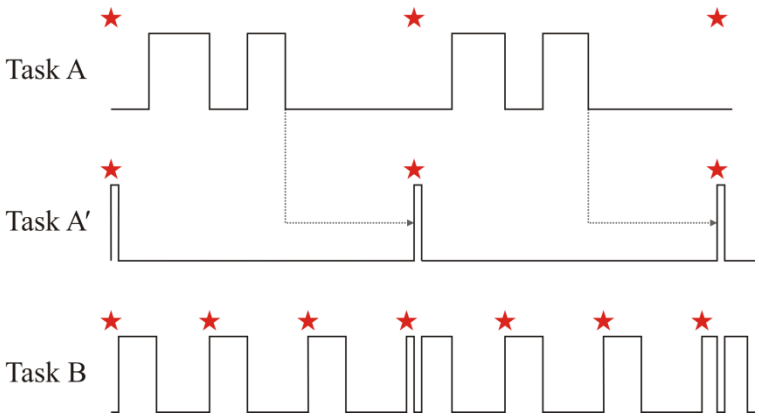


Figure 7-35. The scheduling of a low priority Task A and its higher priority shadow Task A′ executed at the same time as a higher-priority Task B. Note that on two separate occasions, Task B is interrupted by the shadowing task. Stars indicate the boundaries of the periods.

Of too many tasks requiring shadowing tasks, however, must result in our algorithm giving too many false negatives; that is, the test for schedulability returns false even when the tasks are RM schedulable.

7.4.5.5 Restricted priority levels for RM and DM scheduling

For algorithms such as RM and DM scheduling require that tasks with different periods or deadlines are given different priorities. Unfortunately, all priority-driven systems have an integral and finite number of priorities, usually powers of two up to 256, and in some cases, the number of priority levels may be further reduced by other aspects of the design of the system.[20] In other cases, the exact periods or deadlines may not be known apriori, as aspects of the system, including the hardware, may change. Consequently, it may be necessary to have a general algorithm that assigns a task a priority based on its period or deadline.

Suppose that lower and upper bounds on the periods are τ_{lower} and τ_{upper}, respectively, and there are *N* available priorities from 0 to *N* – 1. We cannot simply divide this interval into *N* equally spaced intervals (using a formula such as $N\frac{\tau-\tau_{\text{lower}}}{\tau_{\text{upper}}-\tau_{\text{lower}}}$), for if the lower and upper bounds were 0.01 s and 10 s, respectively, and 10 priority levels were

[20] This section is also based on comments in the TimeSys manual *The Concise Handbook of Real-Time Systems*, version 1.3, 2002, pp.38-40

available, almost all tasks with period less than 1 s would be given the same (highest) priority, and thus a task with period 10 ms would have the same priority as a task with period 500 ms, even though the . As it is likely desirable for a task with period τ to have a higher priority than another task with period 2τ, this suggests an exponential drop off. Thus, a formula such as

$$N\left(\frac{\ln\left(\frac{\tau}{\tau_{\text{lower}}}\right)}{\ln\left(\frac{\tau_{\text{upper}}}{\tau_{\text{lower}}}\right)}\right) = N\left(\frac{\ln(\tau)-\ln(\tau_{\text{lower}})}{\ln(\tau_{\text{upper}})-\ln(\tau_{\text{lower}})}\right)$$

When $\tau = \tau_{\text{lower}}$, the numerator is zero, and when $\tau = \tau_{\text{upper}}$, the ratio is one. Thus, in our example with ten priorities with periods ranging from 0.01 s to 10 s, the distribution of the priorities would be as shown in Figure 7-36, with the boundary points being

0.01, 0.01995, 0.03981, 0.07943, 0.1585, 0.3162, 0.6310, 1.259, 2.512, 5.012, 10.

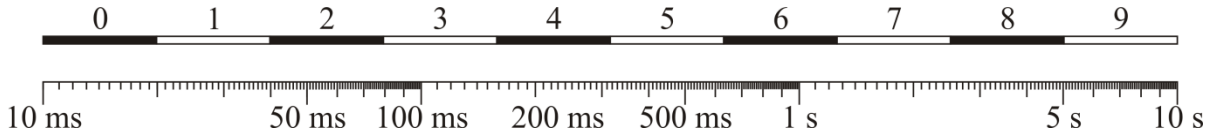


Figure 7-36. Dividing the interval [0.01 s, 10s] into ten exponentially growing sub-intervals for priority levels.

More realistically, if this time interval was logarithmically divided for 256 priorities, the highest ten priorities would span 3.1 ms bounded by

0.01, 0.01027, 0.01055, 0.01084, 0.01114, 0.01144, 0.01176, 0.01208, 0.01241, 0.01275, 0.01310,

while the lowest ten priorities would span 2.365 s with

7.635 s, 7.844 s, 8.058 s, 8.279 s, 8.505 s, 8.738 s, 8.977 s, 9.222, 9.475 s, 9.734 s, 10 s.

Note: In most mathematical libraries, the `log` function returns the natural logarithm, ln(*n*), while the `log10` function returns the common logarithm, $\log_{10}(n)$. Fortunately, because all logarithms are scalar multiples of each other, as $\log_b(n) = \frac{\ln(n)}{\ln(b)}$, it follows that the above ratios are the same regardless of the base of the logarithm. The natural logarithm is used simply because the most natural function to use in a mathematical library is `log`.

An issue with using such a formula to assign priorities is that tasks may be scheduled out of order, and thus this may affect the ability for each task to meet its periodic deadlines. Without providing evidence, the TimeSys concise handbook recommends a minimum of five bits (32 priorities) for the priority if it is to be implemented in hardware, and eight bits (256 priorities) if priority is implemented in software.

7.4.5.6 Summary of priority-based preemptive scheduling algorithms

In this topic, we have introduced the concept of priorities and priority-based schedulers. Specifically, we looked at the RM scheduling algorithm, but we also considered how to implement EDF using a priority-based scheduler. Today, almost all real-time operating systems have schedulers that use priority to determine which task to execute next.

7.4.6 Summary of preemptive scheduling algorithms

We considered four preemptive scheduling algorithms: the design-phase (and usually manual) timeline scheduling, round robin and other fair schedulers, the optimal earliest-deadline first scheduling algorithm, and rate-monotonic scheduling based on fixed priorities. Using priorities to implement a scheduler is so quick and straight-forward that it is

essentially ubiquitous throughout the real-time operating system industry, and thus we needed to consider some of the issues when trying to implement EDF using priorities.

7.5 Issues with scheduling

We will briefly describe a few other issues with scheduling before continuing with the next topic:

1. missing deadlines,
2. non-pre-emptible tasks, and
3. scheduling with multiple processors.

We will briefly look at these next.

7.5.1 Missing deadlines

Suppose a task will miss its deadline. In this case, if the task is either hard or firm, it is better to not schedule the task at all. In a hard real-time system, it may be better to schedule exception handling tasks to deal with the missed hard deadline, and for firm tasks, it is likely better to allow other tasks to run. After all, if in a video broadcast, a frame requires 10 ms of processing time to decode it, but it must be shown in 8 ms, it is better to skip that frame and use the processing time to allow the system to catch up again.

7.5.2 Scheduling non-pre-emptible tasks

With EDF and RM, we must assume that the tasks are pre-emptible; that is, it is possible to halt the execution of one task to begin the execution of another. We will see later that there are certain situations, especially related to the allocation of resources, where it is not possible to pre-empt a lower priority task in favor of a higher priority task. In these cases, there is no known optimal scheduler. We will investigate these later.

7.5.3 Multicore and multiprocessor processing

Up until now, we have considered only a single processor. With the cost of multi-core and multiple processor microcontrollers coming down, such devices will become more common in the future. One example is the XMOS XCore XS1 architecture for a 32-bit RISC microprocessor designed for embedded systems and the XCore XS1-G4 is a quad-core microprocessor built using this architecture. Another is the NXP LPC43xx family of dual-core microcontrollers with an Cortex-M4F core paired with a Cortex-M0 core.

Therefore, we will digress briefly to discuss multiprocessing: the use of either

1. multiple cores, or
2. multiple processors

to accomplish a task, and now, more than one task can run simultaneously. In the first case, there always the benefit of possible shared access to memory; however, the second may require more support for the tasks to share memory. We will, however, consider the effects of having multiple tasks executing in parallel.

7.5.3.1 States and TCBs

As soon as you have multiple threads that can be executed simultaneously, one must consider whether or not one requires one queue per processor or core, or to have one universal queue and schedule the tasks from there. If the systems are sufficiently remote (such as, on separate systems trying to work together), these really no longer become an issue, as the only way communication will occur is through messaging.

7.5.3.2 Multiprocessor scheduling

This course focuses on uniprocessor scheduling; however, there is significant research into multiprocessor scheduling and you will see more of this if you take ECE 455 *Embedded Software*. There are two approaches to multiprocessor scheduling:

1. partitioned scheduling, and
2. global scheduling.

In the first, the tasks and threads are divided among the processors, and each is scheduled separately (reducing the problem to uniprocessor scheduling). The second requires that task and threads are migrated between processors, requiring significantly more overhead; however, even ignoring this overhead, in a paper *Static-priority scheduling on multiprocessors*, Björn Andersson et al. consider an multiprocessor extension to RM scheduling for equivalent processors. The worst case utilization is one third of the total capacity. They also show that "no static-priority multiprocessor scheduling algorithm (partitioned or global) can guarantee schedulability for a periodic task set with utilization higher than one half…"

In addition to the time of scheduling tasks, there is the additional factor of deciding which processor or core to schedule the tasks on and the cost of migrating a task from one processor or core to another. Shortest-job next will reduce the overall wait time of all tasks whether there is one or multiple cores; however, this is not useful for real-time systems. We will look at three scheduling algorithms:

1. least-slack first for fixed-instance tasks,
2. rate-monotonic—first-fit for periodic tasks, and
3. earliest-deadline first for periodic tasks.

7.5.3.2.1 Least-slack first

Least-slack first (LSF) (also least-laxity first or LLF) is optimal for fixed-instance tasks and appears to be slightly better than earliest-deadline first in this case. LSF is not optimal for periodic tasks when there are multiple processors, but this is still an area of research.

7.5.3.2.2 Rate-monotonic—first-fit periodic scheduling

Rate-monotonic—first-fit (RMFF) is a fixed-priority partitioned scheduler that says that a task is assigned to the first processor that still allows that task to be scheduled together with all other tasks that have previously been assigned to that processor. In this case, schedulability is guaranteed if

$$\sum_{k=1}^{n} \frac{c_k}{\tau_k} \le m\left(2^{1/2} - 1\right) \approx 0.41m\,.$$

7.5.3.2.3 3 Earliest-deadline first periodic scheduling

Earliest-deadline first (EDF) is optimal when all tasks have unit execution times—that is, all tasks are of the form (1, τ_k). It may, however, not be possible and therefore there are other conditions whereby it is possible to schedule tasks. For example, we will consider conditions that are *sufficient* for a feasible schedule existing.

Recall that if *x* is necessary for *y*, it follows that if *x* has not occurred or is false, then *y* will not occur or will also be false. In prepositional logic, $\neg x \rightarrow \neg y$, or equivalently, $y \rightarrow x$. That is, if *y* occurred or if *y* is true, then *x* must have occurred or *x* is also true.
Similarly, if *x* is sufficient for *y*, then *x* occurring or if *x* is true, then *y* will occur or *y* will also be true. In prepositional logic, $x \rightarrow y$, or equivalently, $\neg y \rightarrow \neg x$. That is, if *y* has not occurred or if *y* is false, then *x* must also not have occurred or *x* is also false.

Note that *x* is necessary for *y* is equivalent to saying *y* is sufficient for *x*.

The wording can vary. The following are six statements are similar ways of stating that "a function is continuous" is necessary for "a function to be differentiable":

1. A function must be continuous for it to be differentiable.
2. A function must be non-differentiable for it to be discontinuous.
3. If a function is differentiable, it is continuous.
4. If a function is not continuous, it is not differentiable.
5. It is necessary that a differentiable function is continuous.
6. It is necessary that a function is continuous for it to be differentiable.

On the other hand, the following six statements are similar ways of stating that "a function is differentiable" is sufficient for "a function to be also continuous":

1. A function that is differentiable is also continuous.
2. A function that is not continuous is not differentiable.
3. If a function is differentiable, it is continuous.
4. If a function is not continuous, it is not differentiable.
5. It is sufficient that a function is differentiable for it to be continuous.
6. It is sufficient that a function is not continuous for it to be not differentiable.

Note that "*x* is necessary for *y*" is the same as saying "*y* is sufficient for *x*". A function being continuous is necessary for the function being differentiable, and a function being differentiable is sufficient for it to be continuous.

First, the utilization must be less than *n*; that is

$$U = \sum_{k=1}^{n} \frac{c_k}{\tau_k} \leq n .$$

If we assume our units are sufficiently small that each time unit is an integer, then if we define $T = \gcd(\tau_1, ..., \tau_n)$, then it is sufficient for a feasible schedule to exist if $T \frac{c_k}{\tau_k}$ is an integer for each $k = 1, \ldots, n$.

Recall that, for rational numbers, assuming you have a common denominator,

$$\gcd\left(\frac{a}{q}, \frac{b}{q}, \frac{c}{q}\right) = \frac{\gcd(a,b,c)}{q} .$$

For example, consider the tasks (24, 40), (8, 20), (4, 10), (16, 40) and (3, 15). The sum of the utilizations is 2, and therefore there is a possibility that these can be scheduled: the gcd of the denominators is 5, and

$$\frac{24 \cdot 5}{40} = 3, \frac{8 \cdot 5}{20} = 2, \frac{4 \cdot 5}{10} = 2, \frac{16 \cdot 5}{40} = 2, \frac{3 \cdot 5}{15} = 1$$

are all integers. In this case, the first and fourth tasks can be scheduled on one processor while the 2nd, 3rd and 5th tasks can be scheduled on the second, as shown in Figure 7-37.

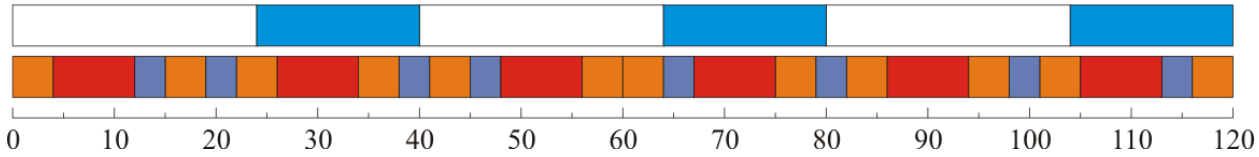


Figure 7-37. Solution to five tasks on two processors.

7.5.3.2.4 Summary of multiprocessor scheduling

The theory for multiprocessor scheduling is not as developed as it is for uniprocessor scheduling; however, as the price of multicore microcontrollers and microcontrollers come down, and as the demand for more processing in embedded applications increases, more research will likely result in further developments and performance guarantees.

7.5.3.3 Summary of multiprocessor and multicore processing

The lowering costs of multiple core processors and multiple processor boards will inevitably begin to become a significant aspect of embedded and real-time systems, even if they that common today. There are scheduling algorithms for such systems that are generalizations of uni-core/processor scheduling algorithms, but they do not have the guarantees when implemented on multiple cores or multiple processors.

7.5.4 Summary of issues with scheduling

We have quickly covered some issues with respect to scheduling, including missed deadlines, the reality that some tasks are not pre-emptible, and the state of having multiple cores or multiple processors (multiprocessing).

7.6 Summary of scheduling

In this topic, after giving an introduction to the topic, we've considered multiprogramming and non-preemptive scheduling as well as multitasking/multithreading and preemptive scheduling. We concluded by looking at other issues with scheduling.

Problem set

7.1 In what way are the FCFS and shortest job next algorithms fair?

7.2 Why are these algorithms not suitable for real-time systems?

7.3 FCFS may be most suitable for some client-server models; however, suppose that there are significant differences between the times it takes to service the different clients. What could you do if you can predict the service time of an incoming request for a service and

1. there is only one server, or
2. there are multiple servers.

7.3 The initialization of a server requires that seven tasks be executed where some tasks must be in specified orders and some tasks have deadlines. Use timeline-scheduling to create an acceptable schedule of these algorithms. Do not interrupt any of the tasks:

Task	Run-time (ms)	Deadline (ms)	Dependencies
1	1.2	-	
2	0.7	-	1
3	0.5	-	1
4	0.4	-	1
5	0.9	5.0	2, 3
6	1.1	4.0	4
7	0.6	-	4

7.4 Define the earliest-deadline first algorithm for selecting a task.

7.5 How could you modify the earliest-deadline first algorithm if there are dependencies, as suggested in Question 7.3?

7.6 Is the set of tasks in the following table schedulable using earliest-deadline first (EDF) scheduling?

Task	Run-time (ms)	Deadline (ms)
1	1.3	1.5
2	0.6	2.0
3	0.5	2.5
4	0.7	3.0
5	0.2	3.5

7.7 Why are we guaranteed that the set of periodic tasks (with their worst-case computation time and periods) in the following Table is schedulable using EDF scheduling?

Task	c_k (ms)	τ_k (ms)
1	3	11
2	2	12
3	3	15
4	4	20

7.8 Suppose we have an additional task that must have a period of 14 ms. What is the longest its worst-case computation time can be?

7.9 Suppose we have an additional task that has a worst-case computation time of 5 ms. What is the shortest period that task could have?

7.10 Suppose you had a set of tasks that were divided into hard, firm and soft real-time. If the hard real-time tasks do not overload the system but the inclusion of the firm and soft real-time tasks od overload the system, how would you decide which tasks to execute?

7.11 Describe two techniques for determining the worst-case execution time.

7.12 How could you determine the worst-case execution time of the following piece of code? The worst-case run-time of the functions is given in the comments.

```
void f() {
        if ( condition ) {
                g();  // 35 ms
        }

        do {
                h();  // 1 ms

                if ( condition ) {
                        i();  // 25 ms
                        break;
                }
        } while ( condition );

        j();  // 4 ms
}
```

Suppose that it is known that the loop will run at most 6 times. What is the worst-case execution time?

7.12 What is the criteria for rate-monotonic scheduling?

7.13 RM scheduling bases the priority on the length of the period. Could you come up with a scheduling algorithm based on the worst-case execution time?

7.14 The LPC1768 microcontroller allows you change the clock speed of the processor. Suppose that you had a system that had twenty periodic and sporadic tasks for which the utilization was 0.035 when the system clock is at a maximum and you are using RM scheduling. What could you do to reduce the cost of the deployment of your system?

7.15 Suppose you have a system that is schedulable using RM scheduling and the clock rate on the microcontroller is only at 50 % of the maximum clock speed, but a decision has been made to use a real-time garbage collection service similar to *Metronome* which must run for 1 ms every 10 ms. What is the simplest solution to incorporating this scheduler?

7.16 Is the set of tasks with utilizations 0.3, 0.16, 0.15, 0.12 and 0.02 guaranteed to be RM schedulable?

7.17 Schedule the tasks in the following table for a period of 100 ms.

Task	c_k (ms)	τ_k (ms)
1	3	10
2	4	25
3	3	20
4	3	25
5	1	50

7.18 Schedule the tasks in the following table using EDF and RM scheduling.

Task	c_k (ms)	τ_k (ms)
1	4	20
2	3	15
3	2	15
4	2	10

7.19 Suppose the periods were reduced by 20 % in Question 7.18. Again, attempt to schedule the tasks using both EDF and RM scheduling.

7.20 Find the maximum integer value of *n* such that this set of three tasks is schedulable using EDF and RM scheduling, respectively.

Task	c_k (ms)	τ_k (ms)
1	3	8
2	2	10
3	*n*	12

7.21 Why is it that the scenario outlined to provide the lower bound on tasks with arbitrary periods not applicable to the scenario where all tasks have periods that are mutually harmonic.

7.22 Find two tasks with different periods, both with utilization of 0.45 such that they are guaranteed to be schedulable using RM scheduling and justify your answer.

7.23 Find two tasks with different periods, both with a utilization of 0.5 such that the two are not RM schedulable and explain why.

7.24 Find two tasks with different periods that are not harmonic and where both tasks have a utilization of 0.49 but where the two tasks are still RM schedulable.

7.21 If we were to implement EDF scheduling using priorities, it would be necessary to continually modify those priorities. Suppose that we have the following three tasks:

Task	c_k (ms)	τ_k (ms)
1	1	5
2	3	8
3	5	12

Initially, Tasks 1, 2 and 3 would have priorities 0, 1 and 2, respectively. How would these be updated if over time to ensure the highest priority task (lowest value) is the one with the earliest deadline?

In the case of a tie, there are two scenarios:

1. If a task is running and another task becomes ready, but they both have the same deadline, keep running the same task; and
2. If two tasks have the same deadline but currently none is running, you may choose either one.

7.22 In EDF and RM scheduling, if a task is currently running and another task with the same deadline or priority becomes ready, why would it be easier to keep the current task running?

7.23 Consider the following overloaded system:

Task	c_k (ms)	τ_k (ms)
1	1	5
2	3	10
3	6	10
4	8	20

What is the expected behavior if these are scheduled with EDF scheduling? What is the expected behavior if they are scheduled using RM scheduling?

7.24 Suppose that you have the following system which you would like to schedule with RM scheduling; however, the utilization is close to unity. How could you do to schedule these?

Task	c_k (ms)	τ_k (ms)
1	1	4
2	2	9
3	3	12
4	4	18

Could you adjust the periods to 4, 8, 12, 16? Could you adjust the periods to 5, 10, 10, 20?

7.25 Suppose that a sensor created a single value every 10 ms, and a task was responsible for reading this value every 10 ms. In this case, the task should likely use a period of $\tau_k = 10$. Suppose that it was necessary—to ensure a harmonic periods—to change the period to $\tau_k = 9$? Consider this under the following circumstances:

1. the sensor has only one register and
    a. the task has the highest priority, versus
    b. the task is one of the lower priority tasks; versus
2. the sensor has two registers in which it can store a previous not-yet-read value.

7.26 Given the following four pairs of tasks, does the multiplicative test for RM schedulability indicate that these are RM schedulable? Are any of the pairs of tasks flagged as schedulable using the additive test?

(1, 0.12), (2, 1.52)
(1, 0.60), (2, 0.49)
(1, 0.51), (2, 0.63)
(1, 0.46), (2, 0.72)

7.27 Suppose lower priority task of each pair shown in the previous question is sensitive to jitter, and thus a shadow task is created for each. Suppose that by splitting off the shadow task, it decreases the computation time of the task by 0.01, but the shadow task itself has a run-time of 0.01. Are these tasks still flagged as being RM schedulable using the multiplicative test?

8 Hardware interrupts

To this point, we have discussed polling, where the processor will query whether a device is ready to send or receive information. This is usually done through querying a register on the device controller, where that register will either contain a value indicating that it is ready to transmit (for a sensor), ready to receive (for an actuator), busy, or in some other state. Normally, a single *device driver* is written to perform such actions so as to ensure modularity.

Unfortunately, the processor seldom knows when the device is ready to transmit (perhaps the device is periodically reading a value, in which case, the scheduler can have a task or thread read the device; however, if the clocks on the two systems are not perfectly synchronized, the task or thread may miss a value). To guarantee that no value is ever missed, the task or thread must have a period strictly less than that of the sensor. This can be inefficient.

Interrupts are a mechanism whereby a device can signal the processor that it is ready to send or receive information. This allows for significant improvements for non-functional performance requirements, as the executing tasks need not concern themselves with checking the state of the various devices. An interrupt allows a device to essentially raise their hand, so-to-speak, indicating that the processor. This, however, can lead to serious issues with respect to the non-functional requirements of safety: the processor may be executing a critical task and if a device were to halt execution of that task, there may be serious consequences. Thus, we will also discuss strategies that allow the processor to deal with interrupts efficient and to also ignore interrupts under certain conditions.

The evolution of the handling of interrupts has been significant over the years, and so we will only discuss the current state—it would be too much to describe the growth. Thus, we will consider

1. the mechanism of interrupts,
2. how lower-priority interrupts can be ignored if higher-priority interrupts are currently in progress,
3. how to wait for interrupts,
4. designing both time-triggered and event-triggered systems,
5. the critical watchdog timer, and
6. a brief overview of the implementation of interrupts.

Finally, we will conclude with a scenario of what happened on the Apollo 11 mission and where a well-designed real-time system prevented a stream of insignificant interrupts from seriously affecting the landing.

8.1 Sources of interrupts

Now, there are numerous devices that may want to interrupt the processor, including two we have already discussed:

1. the real-time clock, and
2. a watch-dog timer.

Other devices that may communicate with, for example, the LPC1768 include:

1. any of the four UARTs (universal asynchronous receiver/transmitters),
2. CAN (controller area network) bus,
3. pulse-width modulators,
4. other peripherals through one of three I²C (inter-integrated circuit) buses (two-wire serial BUS developed by Philips in the early 1980's allowing for simple and efficient control of applications; widely used in embedded systems applications),
5. SPI data flash memory, and
6. brown-out detection.

There are two broad classifications of hardware interrupts:

1. Those generated internally by the processor, and
2. those generated by external devices.

Internal hardware interrupts include division-by-zero errors, memory protection. We will, however, focus on the servicing of externally generated hardware interrupts.

8.2 The mechanism of interrupts

Another approach, however, is to have the device flag the processor to indicate that it is ready. In this case, we must accomplish the following:

1. interrupting the processor,
2. stop the execution of the current task or thread,
3. execute code relevant to the interruption, and
4. return to the normal execution of code.

8.2.1 Interrupting the processor

In order to interrupt the processor, it is necessary that a device sends a signal to the processor. As there are multiple devices that likely want to interact with the processor, there are two approaches:

1. a single line used by all of the devices, or
2. a line dedicated for each device.

The first is, of course, less expensive in terms of hardware; however, at this point, it must be determined which device signaled the interrupt. The second requires significantly more hardware, but allows a much more immediate and targeted response.

When the processor is signaled, this is referred to as an interrupt request (IRQ).

8.2.2 Halting execution

When an IRQ occurs, the processor may be in one of two states:

1. executing one or more instructions, or
2. in a sleep state.

In the first case, the processor must continue executing all instructions that are currently in the *pipeline*. In the latter, the processor must be woken up from its sleeping state. This is one of our many possible sources of *jitter*, the variation in response of a real-time system.

Modern processors can, where possible, execute instructions in a number of steps. For example, the Cortex-M3 has a three-stage pipeline:

1. with the first cycle, the instruction is fetched (Fe) from memory,
2. the instruction is decoded (De), and
3. the instruction is executed (Ex) and results are written to a register.

This allows up to three instructions to be executing simultaneously. On other systems, Stage 3 is broken into two steps:

3a. the instruction is executed, and
3b. the result is written to a register.

Pentium 4 cores have 31-stage pipelines, were each step is relatively trivial, requiring a minimum of hardware to execute each stage.

At this point, one of two approaches must be taken, as it will be necessary to begin executing other code to respond to the interrupt (the *interrupt service routine* (ISR)):

1. only selected registers, most importantly the program counter (PC) and status register (SR), are saved to a stack, or
2. the complete state of the processor is automatically saved on a stack.

There are benefits to both approaches, as we will discuss.

The first allows for very fast responses: the function called to deal with the interrupt must take responsibility to leave the processor in the exact same state that it found it in. This requires that at least parts of the ISR to be written in assembly; however, this also ensures that the absolute minimal amount of work required is performed: if the ISR only requires two data registers and one address register, it only needs to store and then restore those values. This is the approach taken by the developers of the Motorola 68000-based processors. Each ISR would have to store the state of the processor and restore it prior to returning from the interrupt.

The second approach, used in μVision, allows the ISRs to be written in C. This reduces development costs but increases the run-time costs. We will focus on this second approach, as it is the one you will be using in the project.

8.2.3 Selecting the interrupt service routine (ISR)

At this point, the processor must determine which interrupt service routine (ISR) should be called. Most modern systems use an approach called an *interrupt vector*. This is an array of function pointers, and it selects that entry of the array corresponding to the interrupt in question. The manner in which these are assigned is through special names made available to the programmer together with some trickery.

It is usually a good strategy that whether or not an interrupt is intended to be used in a system, that each interrupt is associated with an ISR that runs in an infinite loop. The purpose of this is for debugging purposes: if there is an interrupt occurring that is not being properly handled by the system, this will cause a significant lag in the system and can be easily caught. For example, μVision IDE defines such a default ISR for each such interrupt. Once the development code is replaced with production code, in general, all (apparently) unnecessary ISRs are replaced with a simple return. If they are called, they do nothing, so they will still slow the system down, but not significantly.

For each type of interrupt, the μVision IDE for the MCB1768 provides a name to which you can assign a function, for example, if you were intending to use the analog-to-digital converter, you would define:

```
void ADC_IRQHandler( void ) {

    #ifdef DEVELOPMENT
    while( 1 ) {
        // Infinite loop
    }
    #endif
}
```

The name `ADC_IRQHandler` will be specific to the interface you are using. For μVision, the approach is to have a name associated with the device appended with `_IRQHandler`. To view all available interrupts for the MCB1768, see `startup_LPC17xx.s`.

For the Cortex-M3, the interrupt vector table starts at memory location `0x00000004` and is preceded by only the initial value of the stack pointer (SP). It is possible to relocate this table (including the stack pointer) elsewhere by assigning the vector table offset register (`VTOR`).

The first two locations are the addresses of the routines for a system reset and for non-maskable interrupts (NMI).

The list of all identifiers associated with ISRs is listed in `startup_LPC17xx.s`, reproduced here for educational purposes, with some of the more relevant ones identified.

```
__Vectors       DCD     __initial_sp              ; Top of Stack
                DCD     Reset_Handler             ; Reset Handler
                DCD     NMI_Handler               ; NMI Handler
                DCD     HardFault_Handler         ; Hard Fault Handler
                DCD     MemManage_Handler         ; MPU Fault Handler
                DCD     BusFault_Handler          ; Bus Fault Handler
                DCD     UsageFault_Handler        ; Usage Fault Handler
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     SVC_Handler               ; SVCall Handler
                DCD     DebugMon_Handler          ; Debug Monitor Handler
                DCD     0                         ; Reserved
                DCD     PendSV_Handler            ; PendSV Handler
                DCD     SysTick_Handler           ; SysTick Handler

                ; External Interrupts
                DCD     WDT_IRQHandler            ; 16: Watchdog Timer
                DCD     TIMER0_IRQHandler         ; 17: Timer0
                DCD     TIMER1_IRQHandler         ; 18: Timer1
                DCD     TIMER2_IRQHandler         ; 19: Timer2
                DCD     TIMER3_IRQHandler         ; 20: Timer3
                DCD     UART0_IRQHandler          ; 21: UART0
                DCD     UART1_IRQHandler          ; 22: UART1
                DCD     UART2_IRQHandler          ; 23: UART2
                DCD     UART3_IRQHandler          ; 24: UART3
                DCD     PWM1_IRQHandler           ; 25: PWM1
                DCD     I2C0_IRQHandler           ; 26: I2C0
                DCD     I2C1_IRQHandler           ; 27: I2C1
                DCD     I2C2_IRQHandler           ; 28: I2C2
                DCD     SPI_IRQHandler            ; 29: SPI
                DCD     SSP0_IRQHandler           ; 30: SSP0
                DCD     SSP1_IRQHandler           ; 31: SSP1
                DCD     PLL0_IRQHandler           ; 32: PLL0 Lock (Main PLL)
                DCD     RTC_IRQHandler            ; 33: Real Time Clock
                DCD     EINT0_IRQHandler          ; 34: External Interrupt 0
                DCD     EINT1_IRQHandler          ; 35: External Interrupt 1
                DCD     EINT2_IRQHandler          ; 36: External Interrupt 2
                DCD     EINT3_IRQHandler          ; 37: External Interrupt 3
                DCD     ADC_IRQHandler            ; 38: A/D Converter
                DCD     BOD_IRQHandler            ; 39: Brown-Out Detect
                DCD     USB_IRQHandler            ; 40: USB
                DCD     CAN_IRQHandler            ; 41: CAN
                DCD     DMA_IRQHandler            ; 42: General Purpose DMA
                DCD     I2S_IRQHandler            ; 43: I2S
                DCD     ENET_IRQHandler           ; 44: Ethernet
                DCD     RIT_IRQHandler            ; 45: Repetitive Interrupt Timer
                DCD     MCPWM_IRQHandler          ; 46: Motor Control PWM
                DCD     QEI_IRQHandler            ; 47: Quadrature Encoder Interface
                DCD     PLL1_IRQHandler           ; 48: PLL1 Lock (USB PLL)
                DCD     USBActivity_IRQHandler    ; 49: USB Activity interrupt to wakeup
                DCD     CANActivity_IRQHandler    ; 50: CAN Activity interrupt to wakeup
```

This list will, of course, differ with the number of peripherals available. Each of these is assigned a default value; however, through some compiler trickery, if you declare a function to have the same name, it will use your version.

8.2.4 Characteristics of interrupt service routines (ISRs)

An interrupt service routine should be as short as possible, for three reasons; to

1. allow the system to quickly return to normal operation,
2. minimize the possibility of other interrupts arriving (to be discussed next), and
3. reduce the probability of a programming bug during the ISR.

In general, ISRs should not call functions, but if it is necessary to call a function, such functions must be *re-entrant*—after all, such a function could have been interrupted. Any function that uses either static or global variables is non-re-entrant; for example, consider this simple swap function that uses a global variable:

```
int tmp;

void swap( int *x, int *y ) {
    tmp = *x;
    *x = *y;
    *y = tmp;
}
```

If an interrupt occurred between `tmp` being assigned and its value being retrieved, then if that ISR also called this function, the value of `tmp` would be overwritten. The function `printf(…)` is non-re-entrant—you will be using the debugger if you are attempting to correct an error in an ISR.

If it is necessary to call a non-re-entrant function from an ISR, it is necessary that all calls to such functions be wrapped in blocks of code where all interrupts are turned off and then back on again once the non-re-entrant function has returned—probably a bad idea in real-time systems.

```
__disable_irq(); {    // Disable Interrupts
    // Call non-re-entrant function
    printf( "Hello world!\n" );
} __enable_irq();     // Enable Interrupts
```

8.2.5 Returning from an interrupt

When an interrupt service routine finishes execution, it must return to normal operation. This will not be a simple function return, however, as the previous function was interrupted. Thus, many processors will have instruction sets that include a special return-from-interrupt instruction (`rti`). This instruction will restore the system to normal execution. For µVision, this is taken care of by the compiler—you do not have to worry about this.

One issue with interrupts, however, is that the interrupt may now wake up a task or thread that has a higher priority than the one that was interrupted. In this case, it is necessary

1. to place the currently executing task back onto the ready queue (setting its state to `READY`), which involves saving the state of the processor immediately prior to the interrupt firing (information that may be located in various locations, depending on how this is done), and
2. change the state of the previously blocked high-priority task or thread to `RUNNING`, and changing the state of the processor and stacks as if that was the task or thread that was interrupted. Once the return-from-interrupt is issued, it will be as if that was the task that was executing.

Now, in order to change the executing task, it is necessary access registers and the various stacks. Of course, this can only be done through assembly instructions and access to the task/thread control blocks (TCBs).

With the setup in µVision, where ISRs can be written in C, this is no longer an option, so now we must take a different approach to ensuring the highest priority task is scheduled as soon as possible. In this case, we must revert to using round-robin scheduling. While round-robin scheduling is generally not required for real-time systems (it adds additional overhead to the system), if round-robin scheduling is used, then any task that is made ready as a result of an ISR (say, by posting to a semaphore), the highest priority task will have to wait no more than one time slice before it is scheduled. Now, this is not ideal, but it gives an upper bound as to how long a high priority task must wait before it is scheduled. Thus, in order to reduce development time, it is necessary to increase deployment costs: the processor speed will have to be slightly higher, for example, which in turn increases power consumption. In some cases, this may be acceptable.

8.2.6 A simple application

Consider yourself sitting in front of a computer typing at the keyboard using a word processor in a windows interface. As the processor is waiting for you to do something, it waits and does nothing. All of its threads are waiting on some form of input. Each time you strike a key, the keyboard signals an interrupt, and the operating system launches an interrupt service routine (ISR) which accesses the key that was pressed and then determines which window is *in focus*—that is, which window is active. It will then determine whether or not there is a thread associated with that window waiting on an interrupt from the keyboard. As one is waiting, they value of the keystroke is stored somewhere and that task is woken up and placed on the ready queue. When the ISR executes a return-from-interrupt, and the currently executing task keeps running. As it is likely that the current executing task is the idle task, it soon yields the processor and the task waiting for the keystroke is scheduled. It accesses the character, places it into an appropriate entry of a data structure for the document, displays that keystroke on the screen, and then issues another wait on the next keystroke.

Alternatively, if you move a trackball or mouse, each movement of sufficient size signals an interrupt. The operating system determines that a movement was made and then stores the characteristics of the movement (speed and direction). It then wakes up a thread that deals with displaying the mouse on the screen. On the return-from-interrupt, that thread redraws the cursor and determines whether a signal must be sent to the currently active window. If so, that second thread is woken up. That currently active window may itself have a thread waiting on a trackball or mouse movement.

8.2.7 Summary of the mechanism of interrupts

In this topic, we have gone through the steps of how an interrupt is implemented, and looked at a brief example.

8.3 Ignoring and nested interrupts

When an interrupt is executing, there is nothing to theoretically prevent another interrupt from occurring. In a real-time system, it may be more important to respond to this second interrupt than to continue servicing the current interrupt—the currently executing interrupt may be nothing more a sensor indicating that it has some data for the system, while the incoming interrupt may be a signal that requires immediate response: a parameter of the system has gone outside its allowable range.

There are two categories of interrupts:

1. interrupt requests (IRQs), and
2. non-maskable interrupts (NMI).

The latter are interrupts that require an immediate response and can never be disabled. These include

1. non-recoverable hardware errors, including
    a. non-recoverable internal system chipset errors,
    b. corruption in system memory including parity and ECC errors, and
    c. data corruption detected on system and peripheral buses;
2. system debugging and profiling; and
3. system resets.

The most straight-forward means of dealing with this is to include with each interrupt a flag, and when an interrupt occurs, that flag is set. If a flag is set while an ISR is executing, when the ISR returns, that flag is checked to determine whether or not any other interrupts have occurred. If another interrupt has occurred in the interim, the corresponding ISR is then called, rather than returning to normal execution. This, of course, requires that any ISR signals an acknowledgment that it has serviced the interrupt.

Another approach is to allow interrupts to interrupt ISRs. The immediate problem, however, is that an incoming interrupt may be of less significance than the interrupt that is currently being handled: how do you flag the relative importance of the various interrupts?

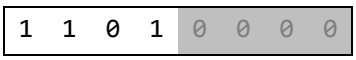
In this case, like tasks and threads, interrupts are given a priority, and at any time, only higher priority interrupts are allowed to interrupt a currently executing ISR.

Type	Exception Number	IRQ Number	Exception type	Priority	Handled using
processor core exceptions or internal interrupts	1	n/a	Reset	–3	
	2	–14	NMI	–2	
	3	–13	HardFault	–1	Fault handlers
	4	–12	MemManage	Configurable	
	5	–11	BusFault		
	6	–10	UsageFault		
	11	–5	SVCall		System handlers
	14	–2	PendSV		
	15	–1	SYSTic		
device-specific exceptions or external interrupts	16	0	IRQ		ISRs
	17	1			
	⋮	⋮			

In the Cortex-M3, there are three registers associated with interrupts:

1. The PRIMASK register is 1 bit and when set, it allows NMI and hard fault exceptions. All other exceptions are blocked.
2. The FAULTMASK register is 1 bit and when set, it allows only NMI. All other exceptions are blocked.
3. The BASEPRI register is 8 bits and stores a positive value so that interrupts with that priority and lower (equal or higher values) are not allowed. Recall that the lower the interrupt priority number, the higher the actual priority. Negative priorities are the highest and are reserved for NMI and hard fault exceptions.

The Cortex-M3 allows each vendor to choose the number of bits for the priority of an interrupt, requiring a minimum of three bits. The LPC1768 microcontroller uses four. Now, in order to make the various platforms somewhat compatible, ARM uses an interesting trick: the byte used to store the priority stores the relevant bits as the most significant bits. Thus, the interrupt priority of 13 (in a range from 0 to 15) would be stored as:



Thus, if this code is ported to the TI Tiva Cortex-M4F, the priority becomes 6 (in a range from 0 to 7):

Thus, a reasonable distribution of priorities still exists.

By default, the priority of all interrupts is zero.

Setting priority is done through the Cortex Microcontroller Software Interface Standard (CMSIS) with a command

```
NVIC_SetPriority( IRQn_Type IRQn, uint32_t )
```

where the first argument is an enumerated type:

```
enum   IRQn_Type {
    NonMaskableInt_IRQn =   -14,  // Exception  2:  non-maskable interrupt
```

```
    HardFault_IRQn =        -13,  // Exception  3:  hard fault interrupt
    MemoryManagement_IRQn = -12,  // Exception  4:  memory management interrupt
    BusFault_IRQn =         -11,  // Exception  5:  bus fault interrupt
    UsageFault_IRQn =       -10,  // Exception  6:  usage fault interrupt
    SVCall_IRQn =            -5,  // Exception 11:  SV call interrupt
    DebugMonitor_IRQn =      -4,  // Exception 12:  debug monitor interrupt
    PendSV_IRQn =            -2,  // Exception 14:  Pend SV interrupt
    SysTick_IRQn =           -1,  // Exception 15:  System Tick interrupt
    WWDG_STM_IRQn =           0,  // Device interrupt 0: window watchdog timer interrupt
    PVD_STM_IRQn =            1   // Device interrupt 1:  PVD through EXTI line detect
}
```

where

1. negative `IRQn` values represent processor core exceptions, or internal interrupts, and
2. positive `IRQn` values (including `0`) represent device-specific exceptions, or external interrupts.

The first device-specific interrupt is associated with the watchdog timer.

For all interrupts, you can request the priority with

```
uint32_t NVIC_GetPriority( IRQn_Type );
```

With the exception of the non-maskable (NMI) and hard fault (HardFault) interrupts, it is possible to set the priority of each interrupt.

Commands to modify characteristics of external interrupts include:

```
void NVIC_ClearPendingIRQ( IRQn_Type );
void NVIC_DisableIRQ( IRQn_type );
void NVIC_EnableIRQ( IRQn_Type );
uint32_t NVIC_GetActive( IRQn_Type );
   // Returns 0 if not active, 1 if active, or active and pending
uint32_t NVIC_GetPendingIRQ( IRQn_Type );
   // Returns 0 if not pending, 1 if pending
void NVIC_SetPendingIRQ( IRQn_Type );
```

An external interrupt is pending between the time that the IRQ arrives and the ISR executes a return-from-interrupt. An external interrupt is active between the time the ISR is started and a return-from-interrupt is executed. It is still active even if the interrupt is pre-empted by a higher priority interrupt.

The last related function is request for a system reset:

```
void NVIC_SystemReset( void );
```

Such a function would be used if it is determined that the system is in an inconsistent state and there is no apparent path to recovery. A failure with a task can be solved by killing the task and respawning that task, but a failure in the memory allocator, scheduler or any other aspect of the system associated with resources, the only solution may be to restart the system. This may occur due to a race condition that was not anticipated or one that was not properly handled.

8.4 Waiting for an interrupt

Up until now, we've discussed handling interrupts; however, how can we *wait* for an interrupt to occur? One solution is polling:

1. A global variable is shared by the interrupt service routine (ISR) and the task or thread wanting to wait on the interrupt and this global variable is false (yielding the processor if it is not ready),
2. The task or thread continually checks that variable to see if it is set to true,
3. When the ISR services the routine, it accesses the device, copies whatever information is necessary and then sets the shared global variable to true.

At this point, the next time the task or thread is scheduled, it the value of the shared global variable will be true. It can then access the information, and deal with it appropriately. Note that this shared global variable will have to be declared `volatile`.

In the next topic, we will look at using a better technique for waiting on interrupts: the use of semaphores that put the task or thread to *sleep* until the ISR is run.

8.5 System design

We now have a second approach to dealing with external events: allowing the software system query sensors or have the sensors signal the software system. These are referred to as *time-triggered* and *event-triggered* systems, respectively.

8.5.1 Time-triggered versus event-triggered systems

There are two means by which a system can be designed, through

1. time-triggered responses, and
2. event-triggered responses.

Most complex systems are a hybrid of these, but systems can be entirely time-driven (periodically turning on and off a light) or entirely event-triggered (often a pure interrupt-driven system).

8.5.1.1 Time-triggered systems

When we sample sensors that report on the state of the surrounding environment, it is necessary to sample the environment sufficiently often to ensure we do not miss any events. Recall that even with sporadic events, there must be a minimum time between such events if we are to have any hope of controlling the system: critical events that can happen arbitrarily closely together in time that require responses with fixed hard deadlines will always have the possibility of overloading the system.

If an event occurs with a maximum frequency of f Hz, the Nyquist-Shannon sampling theorem says we must sample the response with a frequency of $2f$ Hz, or once every $1/2f$ seconds. This is discussed in greater depth in Chapter 20 on digital signal processing and is a significant topic in any text or course on the subject.

If there are multiple events with different frequencies, one may simply:

1. sample each event in a separate thread, or
2. sample all events at the highest frequency.

If the frequencies are sufficiently close, the second solution is probably sufficient, but if there are vast differences in the frequencies, unnecessary work may be performed for the less frequent events. For example, Figure 8-1 shows three events being sampled at twice each frequency. If all three events are sampled at the highest frequency, there is significant and unnecessary processor utilization, as shown in Figure 8-2.

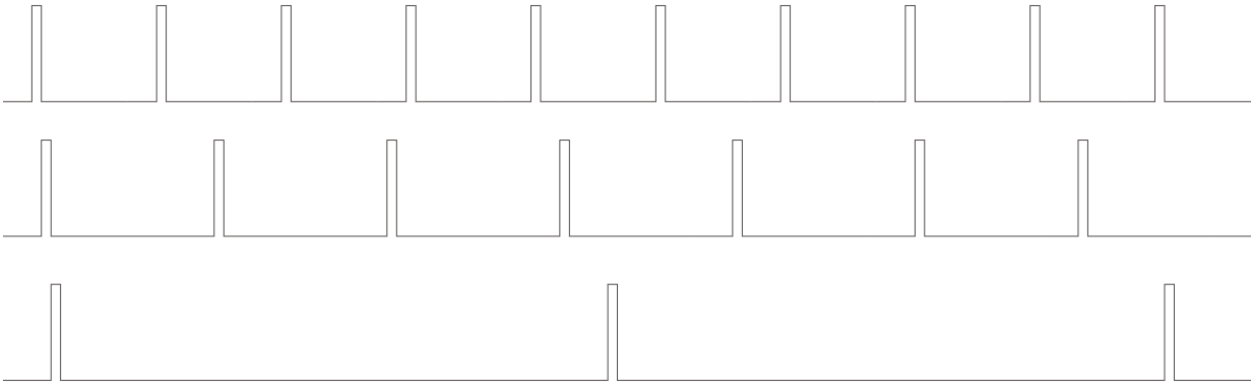


Figure 8-1. Sampling three events, each at their own frequency.

Figure 8-2. Sampling three events at the highest frequency.

Time-triggered events will continually take observations and respond according to the events as changes occur. This is suitable for dealing with systems where parameters describing the environment are highly variable and all parameters are of equal significance. Time-triggered events will provide robust responses during times of overload, but when there is no load (no events of significance), there is the potential for wasted resources.

8.5.1.2 Event-triggered systems

With hardware interrupts, we can now respond to events when the sensors or external systems determine there is an issue to be dealt with. Each such event, however, requires a interrupt service routine (ISR) to execute, possibly pre-empting an even higher priority task that is currently executing.

Responses to such events depend on the environment, and when such events are irregular, such an approach is desirable. When there is no load on the system, event-triggered events ensure economic use of resources, but when the system is overloaded, it is unlikely that the system will be designed to respond appropriately.

8.5.1.3 Hybrid systems and summary

Most systems will be hybrids of both these approaches. We will, however, cover in the next section pure interrupt-driven systems.

8.5.2 Pure interrupt-driven systems

Consider any child's toy that has a microprocessor in it to respond to actions of the child. Such a system would consist of, as far as the program is concerned, sensors, actuators, speakers, a few LEDs perhaps, and other devices that may visually, aurally or physically stimulate the child. Such devices often need only respond to the actions caused by the child. Additionally, to keep costs as low as possible (and to provide as good an experience as possible for the child), the processor should not be continuously running. Instead, it need only respond to interrupts, and when the system is not responding to an interrupt, the processor should go to sleep in a low-power mode. While an interrupt is being responded to, the system may also store information about actions that were performed during the response (so as to not repeat the same thing again the next time) and even gage the response of the child (learning what appears to illicit positive responses in the child, and what appears to bore the child by the child not doing any follow on actions).

Pure interrupt-driven systems are not restricted to toys: consider any appliance that is not meant to be continually operating, such as a toaster, coffee maker, microwave, etc. A coffee maker may have a timer, but as we will see with the

Cortex-M3, the timer can be programmed to interrupt at a specific point in the future; consequently, the processor itself could be put to sleep. Numerous other industrial robots are purely interrupt driven.

The Cortex-M3 has a nice feature that allows the processor (including the NVIC) to be put into a deeper sleep allowing an optional and peripheral low-power *Wake-up Interrupt Controller* (WIC) to respond to interrupts. This optional controller is enabled when the `DEEPSLEEP` bit of the *System Control Register* (SCR) is set to 1. When an interrupt is detected by the WIC, it will wake up the processor and restore its state before the NVIC can then service the interrupt. In deep sleep mode, consequently, the response to interrupts will take more clock cycles.

8.5.3 Summary of time- versus event-triggered systems

This section has described how the response of a system can generally be classified as either a time-triggered response or an event-triggered response.

8.6 Watchdog timers

In our introduction, we discussed the concept of a watchdog timer (WDT). As described, this WDT would simply interrupt via a request to reset the system. This is an unmaskable interrupt and therefore must always succeed.

One issue, however, is that this may not always be the optimal solution: it is not always the case that you must immediately reset the system if a watchdog timer goes off. A more layered approach is to allow a sequence of events to occur:

1. The first time the watchdog timer fires, call an ISR to determine if there is a problem, and if that problem can be fixed. We will discuss in Topic 11 on deadlock means of determining whether or not the system is in a state where all tasks or threads are currently waiting with no possibility of any of them executing. If we can detect deadlock, there are means of fixing this without resorting to resetting the system.
2. If this does not resolve the situation, and the WDT fires again, again, rather than resetting the system, it may be more prudent to take actions such as killing all soft real-time or non-real-time threads and tasks.
3. Finally, if the system still does not respond, the third time the WDT fires, the system is reset.

Incidentally, WDTs are becoming necessary: even if the software is bug-free, hardware may still fail. Today, processors are being built on such a scale that cosmic rays may flip an on-chip bit.

In a story from Ed VanderPloeg, as written by Jack Ganssle in an article "Great Watchdog Timers For Embedded Systems" at http://www.ganssle.com, reproduced here for educational purposes:

> "The world has reached a new embedded software milestone: I had to reboot my hood fan. That's right, the range exhaust fan in the kitchen. It's a simple model from a popular North American company. It has six buttons on the front: 3 for low, medium, and high fan speeds and 3 more for low, medium, and high light levels. Press a button once and the hood fan does what the button says. Press the same button again and the fan or lights turn off. That's it. Nothing fancy. And it needed rebooting via the breaker panel.
>
> "Apparently the thing has a micro to control the light levels and fan speeds, and it also has a temperature sensor to automatically switch the fan to high speed if the temperature exceeds some fixed threshold. Well, one day we were cooking dinner as usual, steaming a pot of potatoes, and suddenly the fan kicks into high speed and the lights start flashing. 'Hmm, flaky sensor or buggy sensor software', I think to myself.
>
> "The food happened to be done so I turned off the stove and tried to turn off the fan, but I suppose it wanted things to cool off first. Fine. So after ten minutes or so the fan and lights turned off on their

own. I then went to turn on the lights, but instead they flashed continuously, with the flash rate depending on the brightness level I selected.

"So just for fun I tried turning on the fan, but any of the three fan speed buttons produced only high speed. 'What "smart" feature is this?', I wondered to myself. Maybe it needed to rest a while. So I turned off the fan and lights and went back to finish my dinner. For the rest of the evening the fan & lights would turn on and off at random intervals and random levels, so I gave up on the idea that it would self-correct. So with a heavy heart I went over to the breaker panel, flipped the hood fan breaker to & fro, and the hood fan was once again well-behaved."

"For the next few days, my wife said that I was moping around as if someone had died. I would tell everyone I met, even complete strangers, about what happened: 'Hey, know what? I had to reboot my hood fan the other night!' The responses were varied, ranging from 'Freak!' to 'Sounds like what happened to my toaster...' Fellow programmers would either chuckle or stare in common disbelief.

"What's the embedded world coming to? Will programmers and companies everywhere realize the cost of their mistakes and clean up their act? Or will the entire world become accustomed to occasionally rebooting everything they own? Would the expensive embedded devices then come with a *reset* button, advertised as a feature? Or will programmer jokes become as common and ruthless as lawyer jokes? I wish I knew the answer. I can only hope for the best, but I fear the worst."

8.7 Implementation of interrupts

Interrupts are normally dealt with through two steps:

1. a mechanism for detecting an edge when no common ground is shared, and
2. an edge-detection circuit.

The most common technique for detecting a signal is to use a NPN bi-polar junction transistor (BJT), where a signal from the attached device is amplified significantly, as shown in Figure 8-3.[21] Such an arrangement is described as an *open collector*. When an interrupt occurs, the voltage across the BJT drops, and does so very quickly.

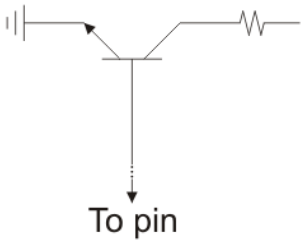


Figure 8-3. A NPN BJT open collector.

The benefit of this design is that it is possible to detect a signal from multiple devices, as shown in Figure 8-4.

[21] Thanks to a conversation with Dr. David Nairn.

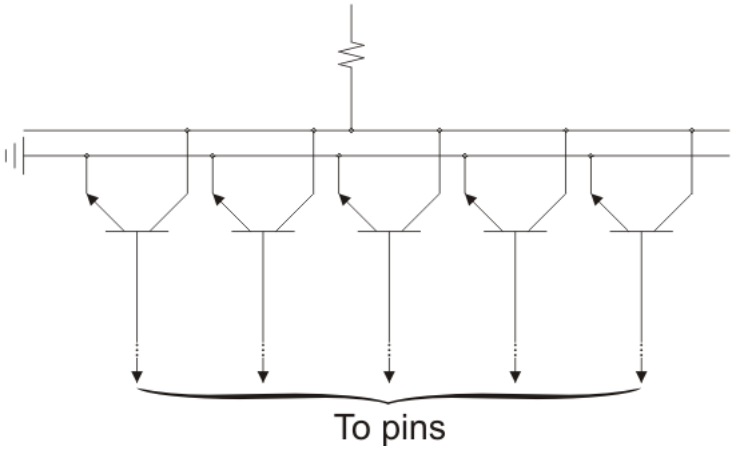


Figure 8-4. Multiple collectors.

Now, devices can either be added to or removed from this arrangement—in any case, it will still continue to function.

The next step is edge detection: we have the voltage across the BJT dropping very quickly. We must now signal the processor that an edge has been detected requiring us to convert the edge into a signal; something that can be done using a D flip-flop, although to achieve internal synchronization (the incoming interrupt is asynchronous), subsequent processing may be required.

8.8 Apollo 11: an interrupt overload

During Apollo 11's lunar descent, the Primary Guidance, Navigation and Control System (PGNCS) had been expected to be operating at approximation 85% utilization. One peripheral attached to this system was the rendezvous radar; a system required to coordinate the rendezvous of the Lunar Module and the Command and Service Module. An error in the checklist manual resulted in switch for the rendezvous radar to be incorrectly positioned during the descent. Because the rendezvous radar used a separate 800 Hz AC source than the one used by the PGNCS for timing reference and the two sources were not phase locked, the random variations in the phase of the signal caused the rendezvous radar to appear to be dithering as opposed to being stationary. Each phantom movement generated an interrupt that had to be handled, resulting in an additional 13% utilization. The system was now at approximately 98% utilization, but still functioning.

During the descent, Buzz Aldrin twice requested that the PGNCS calculate and display the difference between altitude sensed by the radar and the computed altitude. This additional 10% utilization resulted in an overload and returned an *executive overflow* alarm. Fortunately, the software used priority scheduling and with the overload, it immediately killed lower priority tasks including the two requests made by Aldrin. Consequently, despite these problems, the Lunar Module was issued a "go" and a few minutes later, Aldrin and Neil Armstrong became the first humans to land on the Moon. The Director of the Apollo Flight Computer Programming was Margaret H. Hamilton.

The problem was not unknown: it was a known and documented hardware bug. As it only happened once, it was considered to not be critical and therefore was not dealt with prior to the launch—there were sufficiently more important issues to which to attend.[22]

Incidentally, you may be interested in the composition of the computer on the Apollo 11. The system had 2 KiB of RAM and 32 KiB of ROM. The computer ran at 1.024 MHz and it had four 16-bit general-purpose registers together we a few special-purpose registers. The operating system could run eight tasks using a non-pre-emptive multi-tasking scheduler. Tasks were required to yield the processor.

[22] Taken in part from the NASA Apollo 11 Mission Report, http://history.nasa.gov/alsj/a11/A11_MissionReport.pdf and a letter published in Datamation, March 1, 1971 from the Director of Apollo Flight Computer Programming, Margaret H. Hamilton of the MIT Draper Laboratory, entitled "Computer Got Loaded".

You can read more about this in the article *How powerful was the Apollo 11 computer?* by Grant Robertson. See http://downloadsquad.switched.com/2009/07/20/how-powerful-was-the-apollo-11-computer/.

8.9 Summary hardware interrupts

In this topic, we have looked at sources of interrupts, and how interrupts can be handled by a processor or microcontroller. We looked at a simple application of interrupts, and then discussed the mechanism of interrupts, and saw how we can either ignore or use nested interrupts. This was followed by a discussion of time-triggered and event-triggered interrupts, watchdog timers and an example of how interrupts are implemented. We concluded with a cautionary story of how interrupts, had they not been properly handled, may have resulted in the cancellation of the Moon landing at the time that the Lunar and Command Modules were already orbiting the Moon.

Problem set

8.1 Why is it necessary to finish executing all instructions that are currently in the pipeline before servicing an interrupt?

8.2 Why is a processor with a significantly shorter pipeline more desirable for a real-time system, as opposed to one that has significantly longer pipeline.

8.3 Would it be best to describe an interrupt line as a data line, an address line, or a control line?

8.4 When an interrupt occurs, does the processor use the currently executing task's call stack in order to store the current state of the processor? (Hint: consider that the call stack is fixed in size.)

8.5 In any function call that deals with the ready queue (for example, the scheduler), should all interrupts be turned off? Is there any other mechanism of protecting the ready queue, for example, by using semaphores?

8.6 In light of the previous question, should a non-maskable interrupt ever access the ready queue?

8.7 While user mode is almost universal, why do you think that kernel mode is sometimes called supervisor mode and at others monitor mode?

8.8 Why can you not switch between user mode and kernel mode with the execution of a single instruction?

8.9 In your own words, explain how software interrupts work.

9 Synchronization

Given multiple tasks that are attempting to coordinate or *synchronize* (Greek: *same time* cf. synonym *same name*) their activities, this can lead to numerous issues simply because all are accessing memory. There are two forms of synchronization we will investigate:

1. mutual exclusion: preventing two tasks from accessing the same data, and
2. serialization: having two events occur one-after-the-other.

In the first case, *mutual exclusion* generally refers to where only one task is allowed to access a specific memory location (or other resource) at a time. If another task wants access to the same memory or resource, it must wait. The most obvious example is where two tasks may want to use the same printer or write to the same document simultaneously. The second refers to having events occur in specified chronological orders: Task B cannot continue executing until Task A has finished its execution.

In this topic, we will look at

1. the need for synchronization,
2. Petri nets as a graphical means of describing synchronization,
3. synchronizing through token passing,
4. test-and-reset and poling,
5. semaphores,
6. problems in synchronization, and
7. automatic synchronization.

The one means by which synchronization is not to be implemented is by hard-coding wait times, under the assumption that the problem has been solved by another task within the programmed time. This is emphasized in Rule 7 of the JPL coding standard:

> Task synchronization shall not be performed through the use of task delays.

In this standard, they include the comment that "Specifically the use of task delays has been the cause of race conditions that have jeopardized the safety of spacecraft. The use of a task delay for task synchronization requires a guess of how long certain actions will take. If the guess is wrong, havoc, including deadlock, can be the result."

We will, however, begin by looking at why synchronization is required.

9.1 The need for synchronization

Let's look at three problems that demonstrate synchronization issues:

1. Two tasks trying to use the same data structure,
2. Two tasks attempting to share information, and
3. Two tasks attempting to access information from a third.

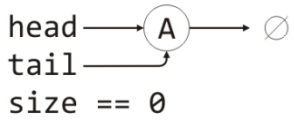
9.1.1 Sharing a data structure

First, suppose we have a shared linked list

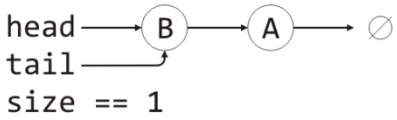
```
typedef struct single_node {
    void               *p_entry;
    struct single_node *p_next;
} single_node_t;

typedef struct {
    single_node_t *p_head;
    single_node_t *p_tail;
    size_t size;
} single_list_t;

bool single_list_push_front( single_list_t *const p_this, void *p_new_entry ) {
    bool success;
    single_node_t *p_new_node = (single_node_t *) malloc( sizeof( single_node_t ) );

    if ( p_new_node == NULL ) {
        success = false;
    } else {
        p_new_node->p_entry = p_new_entry;
        p_new_node->next = p_this->p_head;

        p_this->p_head = p_new_node;

        if ( p_this->size == 0 ) {
            p_this->p_tail = p_new_node;
        }

        ++( p_this->size );
        success = true;
    }

    return success;
}

single_list_t data_list;

void task_1( void ) {
    while ( 1 ) {
        Data *result;
        // Executing...

        single_list_push_front( &data_list, result );
    }
}
```

If only one task ever accesses the data structure, we are fine; however, what could happen if there were two copies of Task 1 executing, call these copies 1.a and 1.b. Now, suppose we started executing both copies at approximately the same time, and Task 1.a gets its data ready first, it calls `push_front`, but then a hardware interrupt occurs between the conditional statement and the auto-increment of the `size` field. At this point:

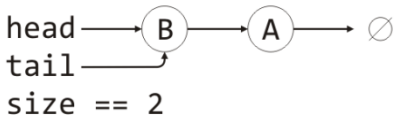
1. the new node was allocated and initialized,
2. the fields head and tail were updated, but
3. the size field is still zero.



Suppose the hardware interrupt is dealt with, and the scheduler decides that Task 1.b will continue executing. It gets to the conditional statement, which is therefore executed as size is still zero, so both head and tail are updated. It continues executing and the size field is incremented.



At some point later, the scheduler is called, and Task 1.a will continue running. At this point, all it does is increment the size field, and the state of the data structure is now



which is in an inconsistent state. If we now try to call `pop_front`, the first node will be removed from the linked list and the associated memory freed; however, the tail pointer will not be updated, so it will still contain the address of a freed location—it is a dangling pointer. Any attempt to access the object at the tail will result in either an invalid access (if we're unlucky) or a segmentation fault (if we're lucky).

Now, if this occurs in a program that is being run interactively, it is less of an issue: it will crash only once in a million times, or perhaps once in a billion times. If the program is, however, embedded in a system, this may cause the system to fail. If you're lucky, the system may simply reset itself and start again. In a real-time system, however, this may result in deadlines being missed.

> Imagine trying to find such a bug: your software package or application is being used by thousands of users, and approximately once a year, you get a spurious bug report from a user. Something goes terribly wrong, and it's clear your program failed, but you cannot recreate the symptoms of the bug in your own environment… Unfortunately, in the real world, this is a very serious problem—a bug that cannot be reproduced is one that will require a struggle to fix, at best.

9.1.2 Two tasks communicating information

Second, consider these two tasks trying to share information: Task 1 is preparing data which is to be sent to and used by Task 2. The result should not be overwritten by Task 1 until Task 2 has completed using it. Task 2 should not try to access the result until Task 1 has finished writing to it.

```
#include <stdbool.h>

bool result_is_produced = false;
Data *result;

// Producer
void task_1( void ) {
    while ( 1 ) {
        // Executing...

        // Prepare something for task 2 and assign to result
        while ( result_is_produced );
        write( result );
        result_is_produced = true;

        // Continue executing...
    }
}
```

```
// Consumer
void task_2( void ) {
    while ( 1 ) {
        // Executing...

        while ( !result_is_produced );
        read( result );
        result_is_produced = false;

        // Continue executing...
    }
}
```

Question: Is it possible that an error such as that shown in Example 1 will occur here?

No. Only after the data is set is the flag `result_is_produced` is set to true.

Question: Are there any problems here?

Yes. Suppose that we have only a single processor or core. In this case, while Task 2 is executing the while loop, Task 1 cannot spend any time actually preparing a result. Similarly, any time Task 1 is waiting for Task 2 to access the data is time that Task 2 is not spending processing that data.

One solution to this problem is to call a `yield()` command. This would make a system call which would then call the scheduler:

```
// Producer
void task_1( void ) {
    while ( 1 ) {
        // Executing...

        // Prepare something for task 2 and assign to result
        while ( result_is_produced ) {
            pthread_yield();
        }

        write( result );
        result_is_produced = true;

        // Continue executing...
    }
}

// Consumer
void task_2( void ) {
    while ( 1 ) {
        // Executing...

        while ( !result_is_produced ) {
            pthread_yield();
        }

        read( result );
        result_is_produced = false;

        // Continue executing...
    }
}
```

If we have a round-robin scheduler, this should not pose an issue; however, if we are in a real-time system where Task 1 has higher priority than Task 2 (or vice versa), then any call to `yield()` by Task 1 would always result in Task 1 being

scheduled again before Task 2—thus we arrive at an infinite loop; however, we cannot conclude this without also knowing the characteristics of the scheduler and the priorities of the tasks.

Note that this is an example of a producer-consumer problem. A single producer with a single consumer will allow a solution, even if it is processor intensive.

9.1.3 Multiple tasks communicating information

Suppose instead we have two copies of Task 2 executing (call them Tasks 2.a and 2.b). Now we have a different issue: we cannot have both consumers accessing the data structure simultaneously.

A first attempt at a solution: to try to prevent the other consumer from accessing the data, as soon as the while loop finishes, immediately set the flag `result_is_produced` state to `false`.

```
// Consumer
void task_2( void ) {
    while ( 1 ) {
        // Executing...

        while ( !result_is_produced ) {
            pthread_yield();
        }

        result_is_produced = false;

        read( result );

        // Continue executing...
    }
}
```

Question: Why does this not help us with respect to the data?

Now it may be possible that the producer, Task 1, accesses the data before Task 2.a finishes reading the previously stored value.

Let's introduce a second flag:

```
int consumer_is_reading = false;

// Consumer
void task_2( void ) {
    while ( 1 ) {
        // Executing...

        while ( !result_is_produced || consumer_is_reading ) {
            pthread_yield();
        }

        consumer_is_reading = true;

        read( result );
        result_is_produced = false;
        consumer_is_reading = false;

        // Continue executing...
    }
}
```

Question: Why does this not really solve the problem?

At this point, an interrupt could occur between the checking that reading is false, and setting it to true.

Now, if you consider that this may only happen once in a million iterations, suppose we are in a reasonably simple embedded system where this happens millions of cycles—in this case, such an error is certain, as opposed to sporadic.

9.1.4 Summary of problems

The problems of synchronization between tasks executing are:

1. Serialization: ensuring that one task is performed after another, and
2. Mutual exclusion: preventing more than one task from accessing data.

Flags are not a suitable solution to either problem as there are at least two operations that must occur when using flags:

1. checking a flag, and
2. setting that flag.

It is always possible that an interrupt can occur immediately between these two operations, and that a context switch may occur at that time.

Thus, this topic will first look at a graphical approach of displaying desired synchronizations. We will then look at four approaches to synchronization:

1. tokens,
2. a *test-and-reset* instruction,
3. semaphores, and
4. automatic equivalents to semaphores.

We will focus on semaphores and we will solve a number of straight-forward synchronization problems using semaphores; however, you will recall that C has manual memory allocation and deallocation while Java uses automatic garbage collection. In a similar way, semaphores will represent a manual technique, while other more complex (and class-like structures) will represent automatic techniques. We will conclude with a problem of priority inversion where a low priority task may block a high priority task.

9.2 Petri nets—describing synchronizations graphically

Petri Nets (PN) were developed 1939 by Carl Adam Petri, who started building an analog computer in 1941, at the age of 13. A PN is a mathematical description of the state of a computer system. In his doctoral thesis, Philip Meir Merlin added a timing extension to PNs, which he called time Petri nets (TPNs), that made it appropriate for modeling real-time systems. We will first discuss PNs and the next section will be on real-time extensions.

A PN is comprised of

1. places or conditions, and
2. transitions between conditions.

In many cases, "places" are conditions that must be satisfied, but at a higher level, it may simply indicate that a task is doing something. Consequently, we use the terms "place" and "condition" interchangeably; "condition" where it is appropriate, and "place" when we are discussing a more abstract state.

For example, consider the PN shown in Figure 9-1. There are two conditions: memory is available and memory is required. If both conditions are met, the memory is allocated, at which point, memory is no longer available.

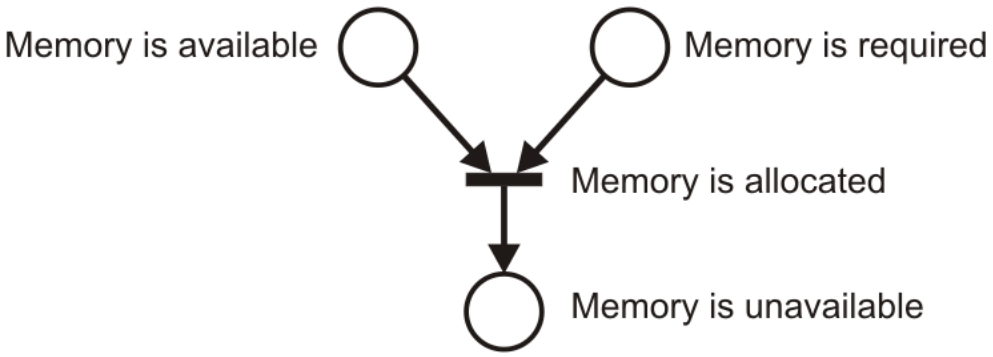


Figure 9-1. A simple PN.

Consider parsing an identifier in C, as shown in Figure 9-2. While parsing, if we are not parsing an identifier and the next letter is either an underscore or a letter, we start determining the identifier. Then, based on the value of the next character, we either stop parsing (saving the identifier), continue parsing the identifier, or issue a parse error. For example, `id``, `id@`, `id#` and `id$` are all invalid in C (you may wish to ask yourself how might you come across, for example, `id!` or `id}` in C, and `id~` in C++ but not in C)?

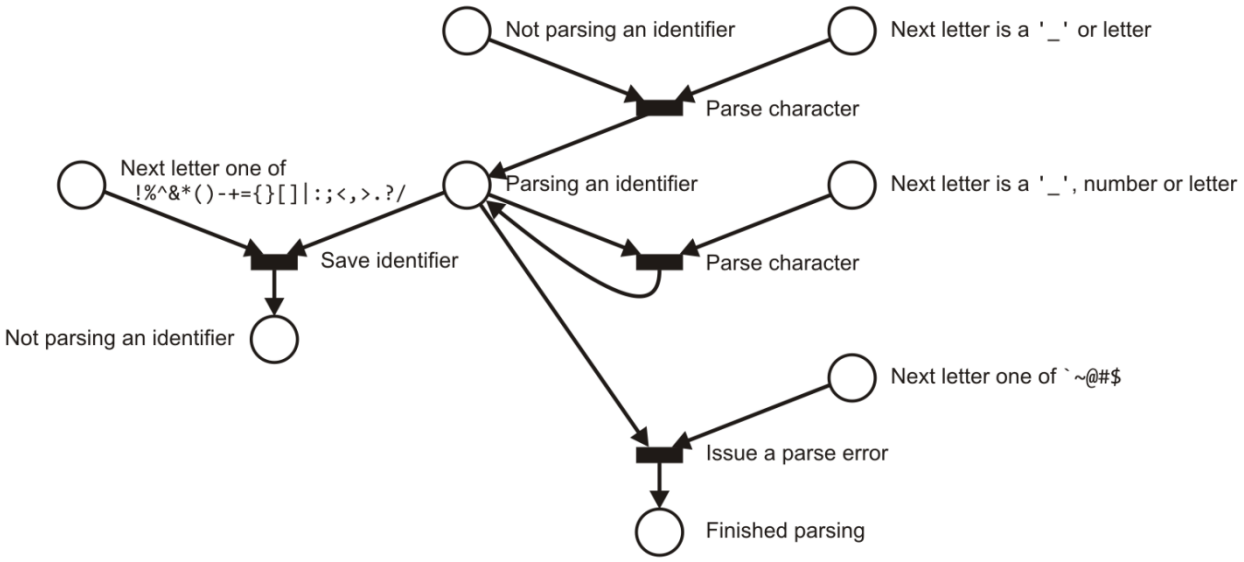


Figure 9-2. Parsing an identifier.

As a more applicable example, consider the PN in Figure 9-3.

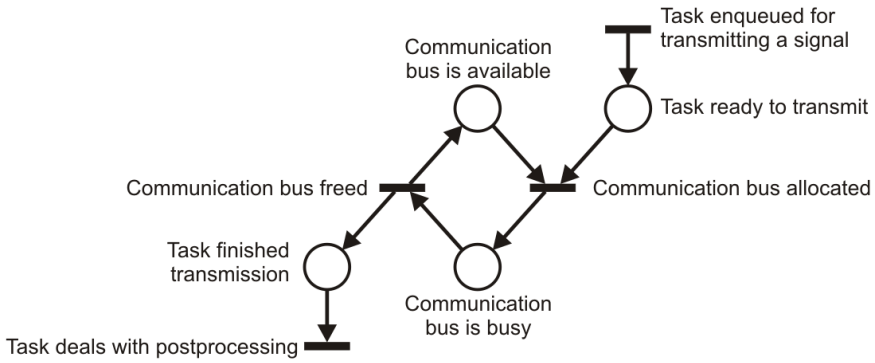


Figure 9-3. A PN surrounding the allocation of a communication bus.

Here, if a task is enqueued for transmitting a signal, and when the communication bus is available, the bus can be allocated. While the transmission occurs, the bus is busy. At some point, the transmission is finished, at which point the task has completed its goal and the bus is now available again. With the communication finished, the task cleans up (frees memory, etc.). Note that the node "Communication bus is busy" is more correctly referred to as a *place* than a condition, as while the communication bus being busy is a condition for freeing it, whenever it is in the state of being busy does not imply that it will immediately be freed. It will only be freed once the process of transmission is finished.

Here is another example in Figure 9-4. If the resource server is ready, a task requests a resource, and the resource is ready, the resource is allocated, used, and released, at which point the task continues processing and the resource is ready again. If the resource is not available, the task must be enqueued. If a task is enqueued waiting on a resource, the server is ready, and the resource is ready, again, it can be allocated.

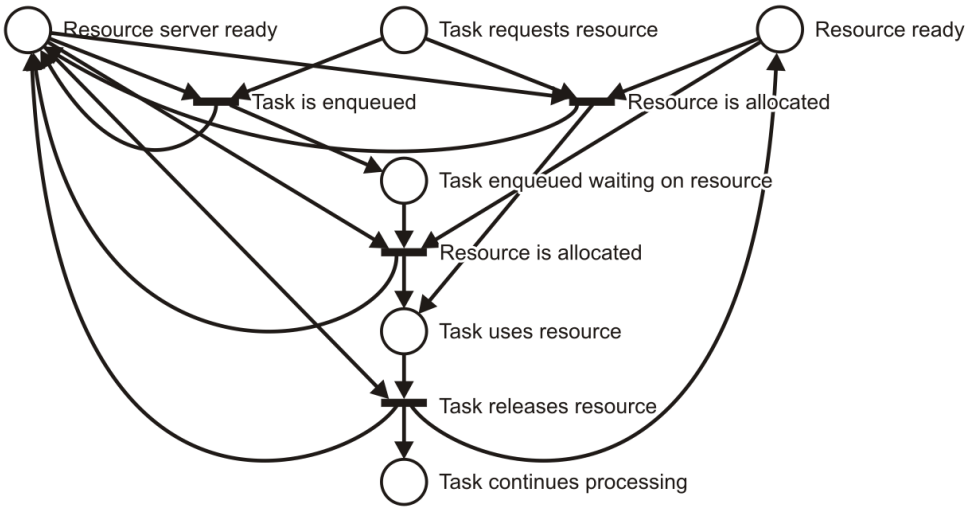


Figure 9-4. The allocation of resources.

All of these indicate the paths that tasks must take, but how do we indicate the state? Is a resource ready or not? Is the server ready or not? To represent the current state, we will use tokens. For example, in Figure 9-5, the server being ready is indicated by a token. This indicates that the condition is satisfied.

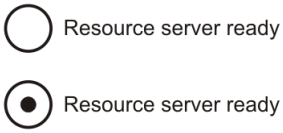


Figure 9-5. The resource server is not ready and a token indicating the server is ready.

It is possible for a server to have, for example, four tasks ready to provide the service. Consequently, this could be represented by four tokens, as is shown in Figure 9-6.

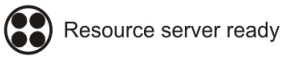


Figure 9-6. A server with four tasks ready.

A transition can fire whenever every condition is satisfied. For example, in Figure 9-7, memory may be available, but it is not required, so nothing happens. However, when memory becomes required, the transition fires, and now we are in the state that memory is unavailable.

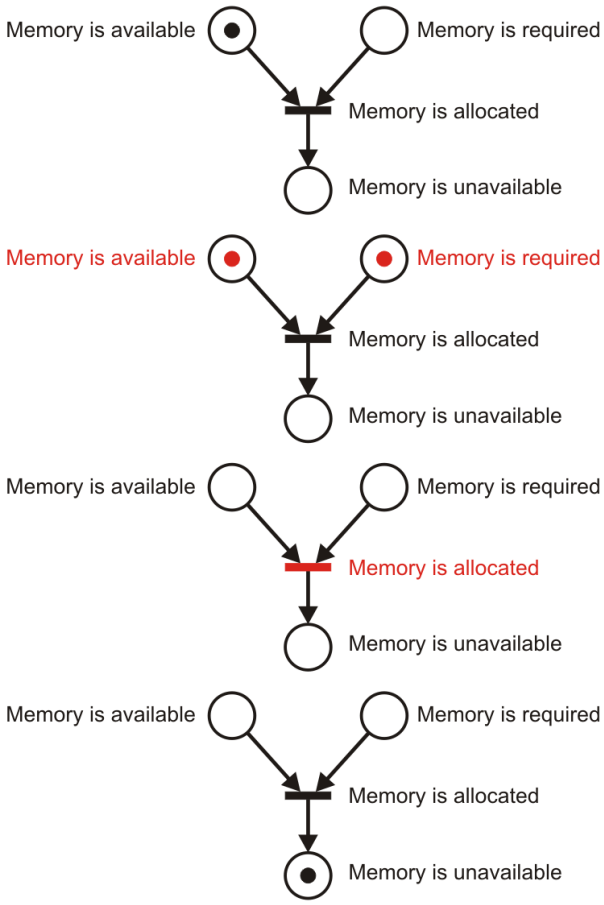


Figure 9-7. The firing of a transition.

Similarly, we may have the following situation where first the communication bus is available, and when a task is ready to transmit, the bus is allocated, used, and then freed, as shown in Figure 9-8.

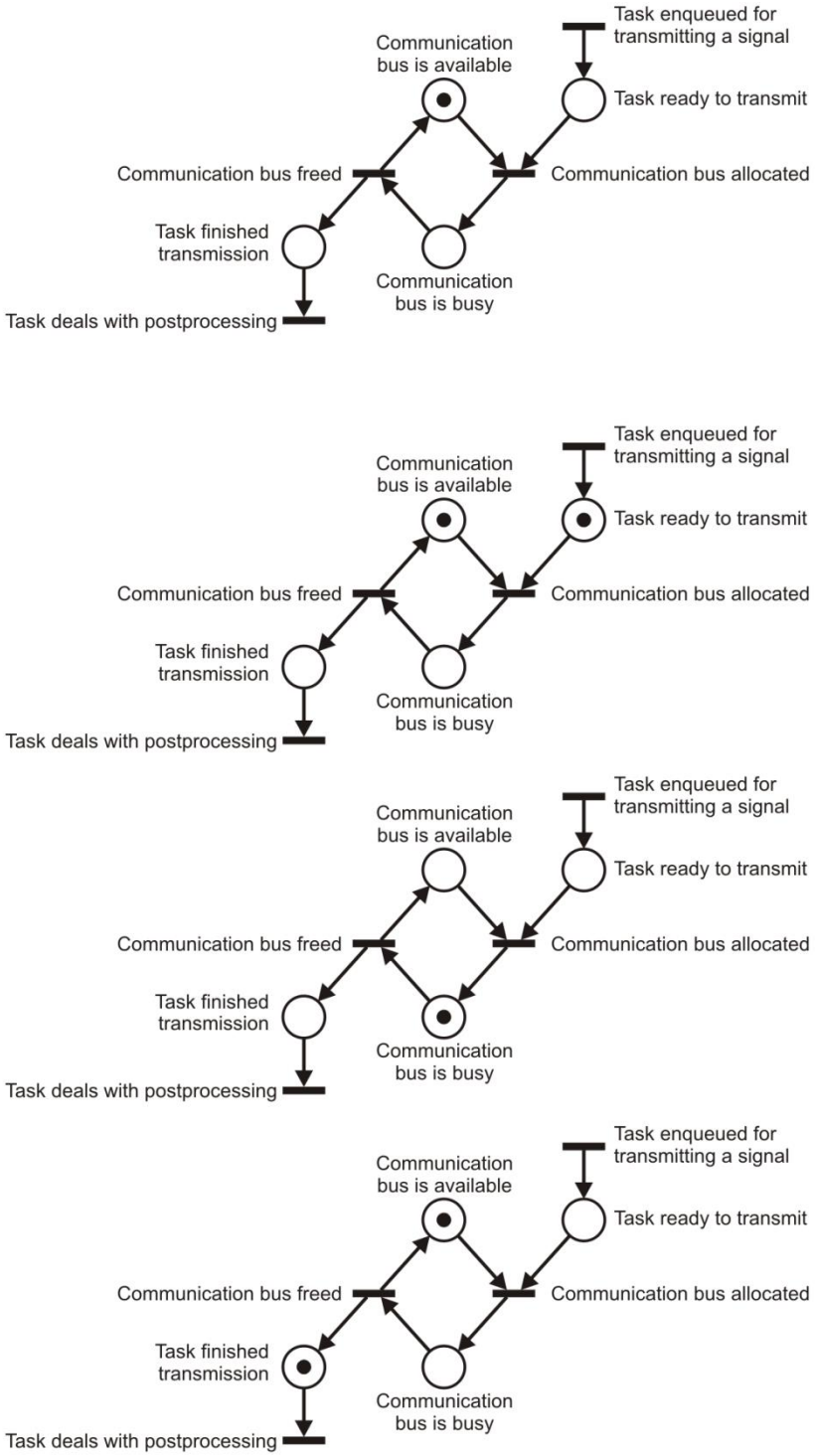


Figure 9-8. The use of a communication bus.

There are several common scenarios in synchronization:

1. concurrency,
2. conflict,
3. synchronization, and
4. merging.

The Petri Net structures for these scenarios are drawn in Figure 9-9.

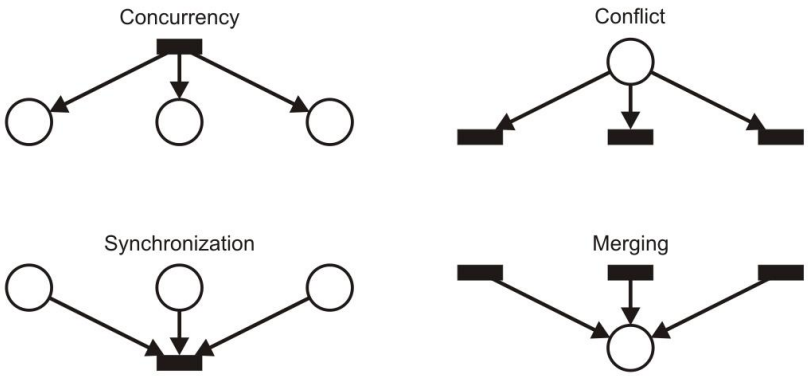


Figure 9-9. Possible synchronization scenarios using Petri nets.

Concurrency occurs when an event can spawn multiple places, while conflict occurs when one place could trigger other events without other triggers. When numerous places or conditions are required for a transition, this involves synchronization, and if multiple transitions result in the same place, the various tasks are merging. Any time that conflict occurs, there is the potential for a race condition.

9.3 Synchronization through token passing

Recall the producer-consumer problem where there are multiple consumers. It is potentially dangerous if two consumers attempt to simultaneously acquire the data produced. One solution for synchronizing the consumers is to essentially have a token that is passed between the consumers. The implementation has the properties that:

1. Each consumer has a unique identifier,
2. The token has the value of one of the identifiers, and
3. When a consumer has accessed the result, it updates the token to be the identifier of the next consumer.

We require a data structure storing the order in which the tokens are passed.

```
typedef struct {
        size_t this_id;
        size_t next_id;
} token_t;

token_t    pair[2] = {{1, 2}, {2, 1}};
token_t triplet[3] = {{1, 2}, {2, 3}, {3, 1}};
token_t quartet[4] = {{1, 2}, {2, 3}, {3, 4}, {4, 1}};
```

One benefit of tokens is that no additional support is required, but resources can only be accessed in the specified order. If one consumer has a higher priority than the others, this may result in issues in real-time systems.

```
void *consumer( void *p_arguments ) {
    token_t *p_identifier = (token_t *)p_arguments;

    while ( 1 ) {
        while ( ( reading != p_identifier->this_id ) || !result_is_produced ) {
                pthread_yield();
        }

        --result;

        printf( "%d ", result );

        result_is_produced = false;
        reading = p_identifier->next_id;
    }
}
```

We would have a global variable

```
size_t reading = 1;
```

and in `main( void )`, we have:

```
pthread_t thread_id[3];
token_t args[3] = {{1, 2}, {2, 3}, {3, 1}};

pthread_create( &thread_id[0], NULL, &consumer, &args[0] );
pthread_create( &thread_id[1], NULL, &consumer, &args[1] );
pthread_create( &thread_id[2], NULL, &consumer, &args[2] );
```

One application of tokens which you may have heard about is in Token rings: an IBM competitor to Ethernet in the 1980s and 1990s. Using tokens was a means of granting permission for one device to send a packet. Token rings were, at the time, superior to Ethernet in almost every way, except price: the royalties charged by IBM easily increased the cost of an Ethernet card by a factor of six.

9.4 Test-and-reset—a crude signal with polling

As hardware designers became aware of the issues with synchronization, they moved forward to solving such problems by providing machine instructions that can support synchronization. We will now look at a mechanism for achieving synchronization through a single variable and a single machine instruction. Recall our attempt to check and set a global variable to allow us to enter the critical region:

```
// Global variable
bool ready = true;

while ( !ready ) {
    scheduler();
}
// Achilles heel
ready = false;

// Access the data structure

ready = true;
```

The problem with this is that an interrupt can occur between the variable being tested and the variable being set to `false`. Instead, we must have some means of testing and setting the variable so that no interrupt can occur between the two operations. Thus, we require that if the variable is

1. `false`, it remains `false`, and
2. `true`, it is set to `false`, but its value must still appear to be `true`.[23]

The function `test_and_reset(…)` is a function that is translated to a single machine instruction which can be thought of as:

```
bool test_and_reset( bool *value ) {
    bool previous = *value;
    *value = false;
    return previous;
}
```

Now, the example of mutual exclusion may be performed as follows:

```
/* Global variable */
bool ready = true;

/* Inside task     */
while ( !test_and_reset( &ready ) ) {
    scheduler();
}

// Access the data structure

ready = true;
```

It is important to remember that all of these operations must be performed by a single machine instruction. If this is not the case, an interrupt can occur between any pair of instructions and this will result in loss of synchronization. Thus:

1. If `ready` is `false`, the value `false` is returned, the value is unchanged, and when the function returns, we yield and loop.

[23] All other references use a *test-and-set*, which switches the significance of `true` and `false`. This text uses a non-standard, but still valid, version as the behaviour of our *test-and-reset* will parallel the behavior of a binary semaphore, which we will see later.

2. If `ready` is `true`, the value `true` is returned, but `ready` is set to `false`, and when the function returns, we exit the loop.

In the second case, if `ready` is ever set to `true`, the first task that calls `test_and_reset` on it will immediately set it to `false`, so all other tasks will continue looping. If any other task calls `test_and_reset`, even if this is immediately after, the value is again `false`, and will go back into the loop.

One weakness of providing a test-and-reset mechanism is that we still have the issue of idle waiting—the loop could be called numerous times prior to the variable `ready` being set to `true`. In addition, in a real-time system, the busy waiting on the variable `ready` may have higher priority than the process that successfully set `ready` to `false` and now needs to finish executing the critical region before it sets it to `true`; consequently, the process in the critical region will never execute.

Therefore, while a test-and-reset instruction is necessary step to provide synchronization, it is only a first step. We will use this instruction to build a more robust data structure in the next section: semaphores.

9.5 Semaphores—a better signal *without* polling

As an alternate solution to synchronization, we will look at a more advanced data structure described by Dijkstra in 1965: the semaphore, a class of data structures for signalling. We will look at

1. binary semaphores,
2. the internal implementation of binary semaphores,
3. counting semaphores, and
4. the implementation of semaphores in various systems.

We will start by looking at binary semaphores.

9.5.1 Binary semaphores

In the last section, we discussed how a single *test-and-reset* command may be used as some form of flag that could be used by one task to signal another. Unfortunately, it is rather crude and requires busy waiting in order to achieve its objectives. We can expand on the concept to provide a more useful construct: the binary semaphore.

We will

1. give a description of binary semaphores,
2. look at applications of this data structure, and
3. describe a specialized binary semaphore for mutual exclusion.

We will begin with a description of this data structure.

9.5.1.1 Description

A *binary semaphore* is a variable that takes on one of two values, `0` or `1`, and it requires the support of, at the very least, a scheduler if not a full operating system. There are four operations on binary semaphores:

`binary_semaphore_init( binary_semaphore_t *const p_this, unsigned int n  )`
Initialize the semaphore to either `0` or `1`.

`binary_semaphore_wait( binary_semaphore_t *const p_this )`
If the semaphore is `1`, set it to `0` and continue executing.
Otherwise, flag this task as being blocked on this semaphore and prevent it from being scheduled.

`binary_semaphore_post( binary_semaphore_t *const p_this )`
If the semaphore is `1`, do nothing.
If the semaphore is `0` and there is at least one task blocked on this semaphore, unblock one of those tasks; otherwise, set the semaphore to `1`.

What differentiates a binary semaphore from a test-and-reset variable is that any task waiting on a binary semaphore that is `0` is blocked from even being scheduled. When a post is issued and there are tasks waiting on the semaphore, one of those waiting tasks is unblocked.

If nothing else, the semaphore interface must communicate with the scheduler, being able to flag a task as being blocked. The scheduler need not know why the task is blocked; only that it is not to be scheduled.

If you don't yet have a grasp of binary semaphores as a concept, think of the following scenario.

If you're at a highway coffee shop and you want to use the washroom, you may have to acquire the key from the staff. Only one individual can have the key at any one time, so if someone else comes asking for the key, they have to wait. When the key is returned, the staff will secure the key if no one is waiting; otherwise, they will pass the key to the next individual waiting for the washroom.

9.5.1.2 Applications of binary semaphores

We will now solve three simple problems using binary semaphores:

1. multiple tasks sharing a data structure (mutual exclusion),
2. a producer and consumer communicating via a common memory location, and
3. multiple producers and multiple consumers communicating via a common memory location.

9.5.1.2.1 Sharing a data structure

Recall the example of a linked list. Now, we could assign each linked list a semaphore. When a semaphore is used for mutual exclusion, the variable name usually contains the substring `mutex`. We will add such field to each single list data structure.

```
typedef struct single_node {
    void               *p_entry;
    struct single_node *p_next;     // Recall single_node_t is not yet defined
} single_node_t;

typedef struct {
    single_node_t *p_head;
    single_node_t *p_tail;
    size_t size;

    binary_semaphore_t mutex;
} single_list_t;

void single_list_init( single_list_t *const p_this ) {
    p_this->p_head = NULL;
    p_this->p_tail = NULL;
    p_this->size = 0;

    binary_semaphore_init( &( p_this->mutex ), 1 );
}

bool single_list_push_front( single_list_t *const p_this, void *p_new_entry ) {
    bool success;
    single_node_t *p_new_node = (single_node_t *) malloc( sizeof( single_node_t ) );

    if ( p_new_node == NULL ) {
        success = false;
    } else {
        p_new_node->p_entry = p_new_entry;
```

```
            binary_semaphore_wait( &( p_this->mutex ) ); {
                // Critical region
                p_new_node->p_next = p_this->p_head;

                p_this->p_head = p_new_node;

                if ( p_this->size == 0 ) {
                    p_this->p_tail = p_new_node;
                }

                ++p_this->size;
            } binary_semaphore_post( &( p_this->mutex ) );

            success = true;
        }

        return success;
    }
```

Any time we are modifying fields, we wait on the mutual exclusion semaphore `mutex`. If no other task is accessing this region, the semaphore will be set to 0 and the task enters the mutual exclusion block. If another task has already waited on this semaphore, it will be 0, so the current task will be blocked.

The code that is executed between waiting on a binary semaphore and posting to that binary semaphore in is called the *critical region*. Only one task can be in its critical region at a time.

Once the function finishes executing the mutual exclusion block, it will post to the same semaphore it waited on. If no task is blocked on the semaphore, the value of the semaphore is set to 1; otherwise, there is a task waiting and the semaphore chooses one of these tasks to be flagged as ready to execute.

A few notes, first on the semantics of a block to denote the mutual exclusion region. The block has nothing to do with the syntax of the C programming language. Instead, we are making use of C syntax to highlight the mutual exclusion block. What is being done above is no different from using indentation to mark the body of a looping statement. For example, consider the two examples:

```
binary_semaphore_wait( &( p_this->mutex ) );
p_new_node->p_next = p_this->p_head;

p_this->p_head = p_new_node;

if ( p_this->size == 0 ) {
    p_this->p_tail = p_new_node;
}

++p_this->size;

binary_semaphore_post( &( p_this->mutex ) );
```

```
binary_semaphore_wait( &( p_this->mutex ) ); {
    p_new_node->p_next = p_this->p_head;

    p_this->p_head = p_new_node;

    if ( p_this->size == 0 ) {
        p_this->p_tail = p_new_node;
    }

    ++p_this->size;

} binary_semaphore_post( &( p_this->mutex ) );
```

The code on the left is also syntactically correct, but is much more difficult to read, as without highlighting, it is difficult to easily see where the critical section begins and ends. It also makes it easier to accidently move the wait and post commands, or delete them altogether.[24]

[24] While the author came up with this notation on his own, after reading through the implementations of FreeRTOS, one may note a similar approach to blocking code for mutual exclusion.

If you are not aware of the word "semantics", the OED indicates that, in relation to computer science, semantics is "the meaning of the strings in a programming language." The syntax determines whether a sequence of characters is valid, while semantics determines what it actually means. First used in 1964 in *IEEE Trans. Electronic Computers* **13** 343/2: "A compiler and a description of the machine for which it compiles is a complete and formal description of the syntax (i.e., grammar) and semantics (i.e., meaning)."

The size of the critical region should be as small as possible. For example, in our code, we allocated memory outside the critical region. It would have been simpler to write each function as:

```
bool single_list_push_front( single_list_t *const p_this, void *p_new_entry ) {
    bool success;

    binary_semaphore_wait( &( p_this->mutex ) );
    single_node_t *p_new_node = (single_node_t *) malloc( sizeof( single_node_t ) );

    if ( p_new_node == NULL ) {
        binary_semaphore_post( &( p_this->mutex ) );
        success = false;
    } else {
        p_new_node->p_entry = p_new_entry;

        // Other instructions...
        binary_semaphore_post( &( p_this->mutex ) );
        success = true;
    }

    return success;
}
```

This, however, expands the period of time that any calling task will spend in this mutual exclusion block region. The first two instructions do not require mutual exclusion—all they require is a call to malloc. Another issue is the call to `malloc`: suppose `malloc` determines there is insufficient memory. If any other higher-priority process called malloc, it, too, would be blocked, only consider the following sequence:

Task 1 (low priority and ready)	**Task 2 (high priority but blocked)**
Call `push_front`	
Waits on `mutex` and continues	
Call `malloc` and is blocked	
	Becomes ready
	Calls `push_front`
	Waits on `mutex` and is blocked
Memory becomes available, and is allocated	
Posts on `mutex` and makes Task 2 ready	
	Becomes ready
	Calls `malloc` and is blocked

Here we have a high priority task blocked while the low priority process was able to place a node onto the linked list. Suppose now that the restricted mutual exclusion block is used:

Task 1 (low priority and ready)	**Task 2 (high priority but blocked)**
Call `push_front`	
Call `malloc` and is blocked	
	Becomes ready
	Call `push_front`
	Call `malloc` and is blocked
	Memory becomes available
	- it is allocated to the high-priority task
	Waits on `mutex`
	Posts on `mutex` and returns

In the second case, the higher priority process is the one to successfully place an object onto the linked list and therefore it may proceed executing. Once again, you may say, "This would be a rare event." True, but if performed often enough, rare events will become certainties: a one-in-a-*n* event has, in the limit, a $1-\frac{1}{e}\approx 63.21\%$ chance of occurring after *n* iterations.

This demonstrates one of the weaknesses of automatic synchronization in languages such as Java: a class is declared `synchronized` which essentially ensures that during one method calls, no other calls to any method from that class may be made.

Note that the term *mutex* is an abbreviation for mutual exclusion. Mutual exclusion is where you want to prevent more than one task from accessing a specific memory location or other resource. The use of semaphores is one means of achieving mutual exclusion, but it is not the only means of achieving mutual exclusion; however, when a semaphore is used to achieve mutual exclusion, it is often given the name `mutex`; however, you should simply think of it as a data structure on which tasks can wait on and post to. Later, we will see a more subtle implementation of a semaphore specifically designed for mutual exclusion.

9.5.1.2.2 Two tasks communicating information

Consider our second example, with a producer and a consumer. If there is only a single producer and a single consumer, we have already seen that this can be simply solved with a Boolean-valued flag; however, this results in busy waiting. Instead, we will use semaphores. Note, however, that unlike a Boolean flag, which can be checked for values of either "true" or "false", semaphores can only be waited upon. Therefore, we must consider any conditions under which a task may be required to wait:

1. The producer will be required to wait if the consumer has not yet processed the data, and
2. The consumer will be required to wait if the producer has not yet created the data.

Thus, we will require two separate semaphores:

1. `ready_to_write`: the data is ready to be written to by a producer, and
2. `ready_to_read`: the data is ready to be read by a consumer.

Initially, no data has been produced, so we set `ready_to_write` to `1`, while we cannot read the data, so `ready_to_read` is set to `0`.

```
data_t shared_result;
binary_semaphore_t ready_to_write;
binary_semaphore_t ready_to_read;

binary_semaphore_init( &ready_to_write, 1 );
binary_semaphore_init( &ready_to_read,  0 );

// Producer
void producing_task( void ) {
    while ( 1 ) {
        // Executing...

        // Prepare something for task 2 and assign to result
        binary_semaphore_wait( &ready_to_write );
        write( data..., &shared_result );
        binary_semaphore_post( &ready_to_read );

        // Continue executing...
    }
}
// Consumer
void consuming_task( void ) {
    while ( 1 ) {
        // Executing...

        binary_semaphore_wait( &ready_to_read );
        read( &data..., shared_result );
        binary_semaphore_post( &ready_to_write );

        // Continue executing...
    }
}
```

Now, if the consumer tries to access result, it will wait on `ready_to_read` and be blocked. At some point, the producer will come along, wait on `ready_to_write` (which is `1`, so it is set to `0`) and it produces and writes the result. When it is finished, it posts to `ready_to_read`, which flags the consumer as being ready to execute.

Note that in general, posting to a semaphore that you may not have waited on is usually acceptable. It is only when a semaphore is specifically designed to enforce mutual exclusion, in which case, the implementation may prevent any other task or thread other than the one that waited on it to post to it.

9.5.1.2.3 Multiple tasks communicating information

The above example with one producer and one consumer is not actually that specific. With multiple tasks acting as producers and/or multiple tasks acting as consumers, it is still possible to have all the tasks share and have exclusive access to the shared variable `result`. There is one weakness, however: suppose after one producer has a assigned a value to `shared_result`, then any other producer must wait until a consumer comes along. In Section 9.5.3, we will look at counting semaphores which can, at least in part, help with this issue.

9.5.1.2.4 Summary of applications

We have seen a few applications of binary semaphores, including allowing two tasks to share data and, more generally, allowing them to communicate. In either case, semaphores protect the data while it is accessed by the other task, and they can be used to signal the other task when the data is ready to be accessed.

9.5.1.3 Mutex: a more exclusive binary semaphore

The first application we looked at was using binary semaphores for mutual exclusion. Unfortunately, a programming error may have a task signal a post to a binary semaphore when it was successfully acquired by another task; as the second task is executing within the critical region, this would break mutual exclusion. One principle in software engineering is to take steps, where possible, to avoid common programming errors. One common remedy is to have a specialized binary semaphore (often given a name with allusion to mutual exclusion, e.g., `mutex_t`) where only the task that successfully acquired the mutual exclusion semaphore may post to it.

9.5.1.4 Summary of binary semaphores

We have introduced the abstract data type of a semaphore as a potential solution to many problems of synchronization, including serialization and mutual exclusion. While there are other solutions (some mentioned above, others which we will look at later), Downey notes that the benefits of semaphores include:

1. placing constraints that avoid programming errors,
2. allowing for solutions that are clean and organized, and
3. it is possible to implement semaphores on many platforms.

In the second case, clean and organized solutions have two benefits: it is less likely to result in logic errors during programming, and it is also easier to demonstrate the correctness of any solution.

Thus, the use of semaphores solves a number of problems over test-and-reset:

1. the code is much simpler to read and comprehend,
2. it solves the problem for both multiple producers and multiple consumers,
3. it avoids busy waiting by blocking any task from being scheduled if it is waiting on a semaphore already set to `0`, and
4. by blocking tasks, it avoids the additional issues that arise if the producers and consumers do not have the same priority.

These are two very simple examples of how semaphores can be used for both solving problems of mutual exclusion and serialization. The next section will look at numerous more complex problems.

9.5.2 Implementation of binary semaphores

In this section, we will consider both

1. the implementation of semaphores,
2. data structures for priority queues in semaphores, and
3. updating the priorities of or killing tasks within priority queues.

We will look at each here.

9.5.2.1 The binary semaphore algorithms

The most basic information we need for a binary semaphore is:

1. a value that is either `true` or `false` (`1` or `0`), and
2. a queue for those tasks waiting on the semaphore.

If we go back to the topic on multiple threads, we would therefore see that the data structure must be

```
typedef struct {
    bool is_acquirable;          // 1 if it can be acquired, 0 otherwise
    tcb_queue_t waiting_queue;
} binary_semaphore_t;
```

In order to initialize this, we must set the value and initialize the queue:

```
void binary_semaphore_init( binary_semaphore_t *const p_this, bool state ) {
    p_this->is_acquirable = state;
    TCB_QUEUE_INIT( p_this->waiting_queue );        // Initialize the queue
}
```

Now, we will implement a *wait* function to request the semaphore. This function will

1. call test-and-reset on the field `is_acquirable`,
2. if it returns `false`, we enter a loop that blocks the currently executing task and places it on the waiting queue associated with the tasks,
3. otherwise, we return: the task has been granted the semaphore.

The following may seem to be a reasonable implementation:

```
void binary_semaphore_wait( binary_semaphore_t *const p_this ) {
    // While the field is not acquirable, loop
    while ( !test_and_reset( p_this->is_acquirable ) ) {
        // Set the state of the currently executing task to blocked
        p_running_tcb->state = BLOCKED;

        // Place the task onto the ready queue of this semaphore
        TCB_ENQUEUE( p_this->waiting_queue, p_running_tcb );

        scheduler();
    }
}
```

In the loop, if the semaphore value is `true`, this will set the semaphore value to `false` and exit the loop. If the semaphore value is `false`, it will have to wait until another post is issued. One issue, however: what happens if an interrupt occurs after the task is put into a blocked state, but before it is put on the queue. Another task holding the semaphore could then release the semaphore, but not be aware that another will, very soon, be waiting on it. Thus, we require a second semaphore: one that protects access to the queue:

```
typedef struct {
    bool is_acquirable;          // 1 if it can be acquired, 0 otherwise
    bool is_safe_to_modify;
    tcb_queue_t waiting_queue;
} binary_semaphore_t;

void binary_semaphore_init( binary_semaphore_t *const p_this, bool state ) {
    p_this->is_acquirable = state;
    p_this->is_safe_to_modify = true;
    TCB_QUEUE_INIT( p_this->waiting_queue );        // Initialize the queue
}
```

Before we present the solution, you should determine why the following is not a solution:

```
void binary_semaphore_wait( binary_semaphore_t *const p_this ) {
    // While the field is not acquirable, loop
    while ( !test_and_reset( p_this->is_acquirable ) ) {
        // This is not a vaild solution
        while ( !test_and_reset( p_this-> is_safe_to_modify ) ) {
            // Busy waiting--acceptable as this will be quick
        }

        // Set the state of the currently executing task to blocked
        p_running_tcb->state = BLOCKED;

        // Place the task onto the ready queue of this semaphore
        TCB_ENQUEUE( p_this->waiting_queue, p_running_tcb );

        is_safe_to_modify = true;
        // This is not a vaild solution

        scheduler();
    }
}
```

A valid solution is as follows, where access to the entire data structure is protected.

```
void binary_semaphore_wait( binary_semaphore_t *const p_this ) {
    // While the field is not acquirable, loop
    while ( !test_and_reset( p_this->is_safe_to_modify ) ) {
        // Busy waiting--acceptable as this will be quick
    }

    while ( !test_and_reset( p_this->is_acquirable ) ) {
        // Set the state of the currently executing task to blocked
        p_running_tcb->state = BLOCKED;

        // Place the task onto the ready queue of this semaphore
        TCB_ENQUEUE( p_this->waiting_queue, p_running_tcb );

        p_this->is_safe_to_modify = true;

        scheduler();

        while ( !test_and_reset( p_this->is_safe_to_modify) ) {
            // Busy waiting--acceptable as this will be quick
        }
    }

    p_this->is_safe_to_modify = true;
}
```

The post will therefore have to ensure that if a task is waiting on the semaphore, then that task must be placed on the ready queue:

```
void binary_semaphore_post( binary_semaphore_t *const p_this ) {
    // If the semaphore is 'true', do nothing
    if ( p_this->is_acquirable ) {
        return;
    }

    while ( !test_and_reset( p_this-> is_safe_to_modify) ) {
        // Busy waiting--acceptable as this will be quick
    }

    p_this->is_acquirable = true;

    if ( !TCB_QUEUE_EMPTY( p_this->waiting_queue ) ) {
        tcb_t p_waiting_tcb = TCB_QUEUE_FRONT( p_this->waiting_queue );

        p_waiting_tcb->state = READY;
        TCB_DEQUEUE( p_this->waiting_queue );

        TCB_ENQUEUE( ready_queue, p_waiting_tcb );
    }

    p_this->is_safe_to_modify = true;
}
```

Now, we must ask a number of questions:

1. **Our macros for updating the ready queue do so without any mutual exclusion protection—is this okay?**
   Not really, as they are not currently, as written, protected by mutual exclusion. In this case, rather than using semaphores, we should turn off all maskable hardware interrupts.
2. **Could a task be blocked on waiting to put a task onto the ready queue?**
   In a real-time system, yes: for example, this semaphore post may attempt to put a task onto the ready queue, but an interrupt occurs that causes another task to become ready and running. It then posts to a semaphore that also may make a task ready. This could occur arbitrarily often, so the only way to avoid this is to turn off all maskable interrupts and not allowing a non-maskable interrupt to modify the ready queue.
3. **In a real-time system, can a wait on a semaphore result in a higher priority process executing?**
   No. The currently executing task is the task that is ready and is of highest priority. No task is being made ready by a call to `semaphore_wait(…)`.
4. **In a real-time system, can a post to a semaphore result in a higher priority process executing?**
   Yes. The currently executing task is the task that is ready and is of highest priority. No task is being made ready by a call to `semaphore_wait(…)`.
5. **Which task do we pick to dequeue?**
   In our example above, we only pop the task that has been waiting the longest. In a general-purpose system, this is probably considered *fair*, but does not consider priority. In a real-time system, we would pop the highest priority task that has been waiting the longest.
6. **What if that task has a higher priority than the currently executing task?**
   If the task made ready has a higher priority than the currently running task, the scheduler should be called and that task should be the one that is set to run.

9.5.2.2 Data structures for semaphores

In our example, we used simple linked lists for our queue. This allows for a FCFS approach, but is sub-optimal for a real-time system. The alternatives are using a

1. linked list queue with a searching pop,
2. binary min-heap,
3. leftist heap, and
4. skew heap.

We will consider all of these.

9.5.2.2.1 Using a linked-list queue with a searching pop

If we use a linked list, we must search through the linked list to find the task with highest priority. If there are multiple tasks of highest priority, the one closest to the front is the one that has been waiting the longest.

The run time of this scheme is linear in the number of tasks in the queue: $\Theta(n)$. If it is guaranteed that there are not many tasks waiting on a particular semaphore, this may be appropriate. For example, if only three tasks are known to access a particular data structure, this may be easily acceptable.

9.5.2.2.2 Using a binary min-heap

A binary min-heap is an array-based data structure. In this case, all required operations are $O(\ln(n))$ with the push operation having an average run-time of $\Theta(1)$ if the insertions are uniformly distributed among the priorities. Unfortunately, the queue must also be stored as an array, and therefore the maximum size of the array must be known.

For very small priority queue sizes, it is likely faster to use a linked-list queue as described in the previous section. If the queues are significantly bigger, there is a possibility of a lot of wasted space. Thus, we should consider a different approach.

9.5.2.2.3 A leftist heap

A leftist heap is similar to an AVL tree in that all operations are $O(\ln(n))$, but like an AVL tree, it is necessary to track the *null-path length* of any node: that is, the shortest path to a non-full descendant node. Tracking and maintaining this data may be expensive. It is also necessary to track a pointer to the parent, as a node may have to be removed. This overhead may make it undesirable to use a leftist heap. Therefore, let us consider another alternative: the skew heap.

9.5.2.2.4 A skew heap

A skew heap is similar to a leftist heap, but some operations are automatic as opposed to being based on null-path lengths: push into the right sub-heap, but then swap the left and right sub-heap. Consequently, the amortized run times are $O(\ln(n))$, but the worst case run times may be $O(n)$.

9.5.2.2.5 Analysis

In Rönngren and Ayani's paper, they point out that in hard real-time systems, the binary heap is the only acceptable data structure, but suggest skew heaps for soft real-time systems. Their conclusion is:

> In implementing a binary heap, this would require a fixed amount of memory. For real-time applications, the worst-case access time may be the most interesting measure. Of the tested priority queues, the implicit binary heap had the best worst-case performance for individual operations, which was better than $O(n)$. Thus, if guaranteed performance better than $O(n)$ is required, this is the only choice. The Splay Tree and the Skew Heap, however, showed better amortized worst cases and we did not observe any individual access to these queues with worse time complexity than $O(\log(n))$. Thus they are good alternatives for real-time systems without hard real-time requirements.

9.5.2.2.6 Summary of data structures for semaphores

We have considered a number of data structures that could be used to track tasks or threads waiting on a semaphore. A linked list is appropriate if there are only one or two tasks waiting on a given semaphore, binary min-heaps are most appropriate as long as there is an upper bound on the number of tasks that may have to wait on a semaphore, a leftist heap is a node-based data structure that maintains balance through operations similar to that of an AVL tree, and while a skew heap is simpler than a leftist heap, it only has an average run-time of $\Theta(\ln(n))$, making it less than desirable for real-time systems.

9.5.2.3 Priority inversion

One issue we haven't discussed up to this point is what happens if a high-priority task waits on a semaphore that is currently being held by a lower-priority task. There are two situations:

1. a high-priority task may be waiting for a lower-priority task for serialization; that is, for it to complete some action (perhaps reaching a rendezvous), or
2. a high-priority task may be waiting on a semaphore for mutual exclusion.

We will consider both of these issues, as the solutions differ.

In the first case, where serialization is the issue at hand, this should be dealt with at the design phase: if it is known that a lower-priority task is going to be required to reach some point to allow the higher-priority task to continue executing, this should be understood during design, and priorities should be appropriately updated at that time.

In the second, the problem becomes more difficult if the semaphore for mutual exclusion is used to restrict access to, for example, a resource. In this case, when the resource is required by a higher-priority process may not be obvious at design time—the high-priority task may simply be responding to unpredictable external events. In this case, it is quite likely that a high priority task may suddenly require access to a binary semaphore held by another task. The easy option is to kill a low-priority task and restart it, thereby freeing the semaphore for the higher-priority task; however, this will fail, for example, if the semaphore is protecting a data structure that is being updated. In the next chapter on resource management, we will consider a solution to where a high-priority task is waiting on a lower-priority task—a inversion in priorities.

9.5.2.4 Summary of the implementation of binary semaphores

We have now considered the implementation of binary semaphores. A test-and-reset—like instruction is used to check a variable and instead of polling that variable, the task is put to sleep (or blocked from executing). If a task is put to sleep, a reference to the task is placed into either a queue data structure, or—if priorities are relevant—a priority queue data structure. Finally, we introduced the likelihood of a priority inversion, a problem we will solve in the .

9.5.3 Counting semaphores

In some cases, we require more than just a binary semaphore. Suppose for example that a queue has exactly 10 entries. In this case, we could use a semaphore for mutual exclusion, and if the queue is full, we can block a task attempting to place something into the queue, at least, until another task pops an entry from the queue making a space available. We will

1. try to solve the problem with binary semaphores,
2. consider the design of a counting semaphore,
3. look at the interface of numerous semaphores in various operating systems, and
4. consider an application of counting semaphores.

9.5.3.1 Counting with binary semaphores

We have already seen how mutual exclusion can be implemented using semaphores. A semaphore `mutex` is set to 1, and around any code that should not be executed by more than one task at a time, we have

```
binary_semaphore_t mutex;
binary_semaphore_init( &mutex, 1 );

binary_semaphore_wait( &mutex ); {
    // Mutual exclusion block
} binary_semaphore_post( &mutex );
```

Now, suppose we want to restrict the number of tasks not to only one running, but perhaps to *n*. For example, in any client-server model, you may have an arbitrary number of clients requests that are being serviced, each by a separately executing task; however, to ensure the server is not overloaded, only a maximum of *n* tasks are allowed to run at any one time.

We will need some form of counter that can be observed by each executing task to determine whether or not the task can continue executing. If we have reached the limit, then we must wait on an event:

```
int count = 10; // We can run at most 10 tasks

binary_semaphore_t waiting_list;
binary_semaphore_init( &waiting_list, 0 );

void task( void ) {
    --count;

    if ( count < 0 ) {
        binary_semaphore_wait( &waiting_list );
    }

    // Execute the task

    ++count;

    if ( count <= 0 ) {
        binary_semaphore_post( &waiting_list );
    }
}
```

The global variable `count` tracks how many tasks are executing inside the mutual exclusion region. Now, if at most 9 tasks are executing, then when the next tasks begins executing, we decrement count and count is zero or positive. If, however, count becomes negative, that indicates that 10 tasks are already executing, so the task will wait on a semaphore.

When a task finishes, it increments the count. If the count is zero or negative, this means that at least one other task is waiting on the semaphore, so post to that semaphore. Note that in this case, the semaphore is never equal to 1.

What is the problem with this solution? Like before: we are modifying and then checking the global variable `count`.

```
int count = 10; // We can run at most 10 tasks

binary_semaphore_t waiting_list;
binary_semaphore_init( &waiting_list, 0 );
binary_semaphore_t mutex;
binary_semaphore_init( &mutex, 1 );

void task( void ) {
    binary_semaphore_wait( &mutex ); {
        --count;

        if ( count < 0 ) {
            binary_semaphore_wait( &waiting_list );
        }
    } binary_semaphore_post( &mutex );
```

```
    // Execute the task

    binary_semaphore_wait( &mutex ); {
        ++count;

        if ( count <= 0 ) {
            binary_semaphore_post( &waiting_list );
        }
    } binary_semaphore_post( &mutex );
}
```

This demonstrates a common the issue of deadlock that often occurs when synchronizing tasks. As soon as a task waits on the `waiting_list` semaphore, must also be holding the `mutex` semaphore; consequently, any other task wanting to post to the `waiting_list` semaphore will be blocked waiting for `mutex` to be released. The issue of deadlock will be covered in greater detail later in this class.

There is one further problem with this implementation: can any task post to the `waiting_list` semaphore if one task is waiting on it? We have to be very careful about where we post the mutual exclusion.

```
void task( void ) {
    binary_semaphore_wait( &mutex ); {
        --count;

        if ( count < 0 ) {
            binary_semaphore_post( &mutex );
            binary_semaphore_wait( &waiting_list );
        } else {
            binary_semaphore_post( &mutex );
        }
    }

    // Execute the task

    binary_semaphore_wait( &mutex ); {
        ++count;

        if ( count <= 0 ) {
            binary_semaphore_post( &waiting_list );
        }
    } binary_semaphore_post( &mutex );
}
```

9.5.3.2 The design of counting semaphores

This type of issue is so ubiquitous in real-time, embedded and operating systems that it is often easier to abstract such counting into a semaphore that facilitates this approach. Such a semaphore is a *counting semaphore* with the following behaviour:

1. the counting semaphore has an integer value (as opposed to just a binary true-or-false) where the initial value is positive (0 or greater),
2. a *wait* decrements the value of the counting semaphore and if the resulting number is strictly negative, the waiting task is blocked on the semaphore, and
3. a *post* increments the value of the counting semaphore, and if the initial value was strictly negative, this indicates at least one task is waiting on the semaphore, and therefore one task must be made ready and placed on the ready queue.

If the value n of the counting semaphore is strictly negative, then $-n$ is the number of tasks waiting on the semaphore.

The datatype for counting semaphores will be

```
counting_semaphore_t
```

and the interface functions will be similar to that of a binary semaphore:

```
void counting_semaphore_init( counting_semaphore_t *, size_t );
void counting_semaphore_wait( counting_semaphore_t * );
void counting_semaphore_post( counting_semaphore_t * );
```

9.5.3.3 An application of counting semaphores

Suppose a queue has *n* entries in it and multiple tasks are attempting to access this queue simultaneously, some pushing items into the queue, others popping them.

```
typedef struct {
    char array[PIPE_SIZE];
    size_t front, back, size;
    binary_semaphore_t mutex;
    counting_semaphore_t entries_available, entries_occupied;
} pipe_t;

void pipe_init( pipe_t *const p_this ) {
    p_this->front = 0;
    p_this->back = PIPE_SIZE - 1;
    binary_semaphore_init( &( p_this->mutex ), 1 );
    counting_semaphore_init( &( p_this->entries_available ), PIPE_SIZE );
    counting_semaphore_init( &( p_this->entries_occupied ),  0 );
}

void pipe_push( pipe_t *const p_this, char c ) {
    counting_semaphore_wait( &( p_this->entries_available ) );

    binary_semaphore_wait( &( p_this->mutex ) ); {
        p_this->back = (p_this->back == PIPE_SIZE - 1) ? 0 : p_this->back + 1;
        p_this->array[p_this->back] = c;
    } binary_semaphore_post( &( p_this->mutex ) );

    counting_semaphore_post( &( p_this->entries_occupied ) );
}

char pipe_pop( pipe_t *const p_this ) {
    char c;

    counting_semaphore_wait( &( p_this->entries_occupied ) );

    binary_semaphore_wait( &( p_this->mutex ) ); {
        c = p_this->array[p_this->front];
        p_this->front = (p_this->front == PIPE_SIZE - 1) ? 0 : p_this->front + 1;
    } binary_semaphore_post( &( p_this->mutex ) );

    counting_semaphore_post( &( p_this->entries_available ) );

    return c;
}
```

The benefits of using a counting semaphore is that if the pipe is ever empty or full, any tasks attempting to either pop or push, respectively, will be blocked waiting on another task either insert a new entry into or remove an entry from the pipe. It would be much more difficult to achieve this without a counting semaphore.

9.5.3.4 A summary of counting semaphores

While binary semaphores are very efficient at achieving mutual exclusion, counting semaphores allow the semaphores to control access to a finite number of resources. Once more than *n* requests are made to access a particular resource, subsequent tasks are put to sleep until a resource becomes available. The ability to block tasks making requests when none are available is integrated into the concept of the counting semaphore.

9.5.4 Implementation of counting semaphores

In this course, we distinguish between the two types of semaphores. In some systems, only counting semaphores are provided and a binary semaphore may be used by initializing the value of a counting semaphore to either 0 or 1 and then ensuring that under no conditions a post is performed on the semaphore when the value is 1. Thus, mutual exclusion can be achieved as follows:

```
counting_semaphore_t mutex;
counting_semaphore_init( &mutex, 1 );  // Initial value of 1

counting_semaphore_wait( &mutex ); {
    // Critical section...
} counting_semaphore_post( &mutex );
```

We will look at semaphores in POSIX, the Keil RTX RTOS and CMSIS-RTOS RTX. We will then consider how to implement counting semaphores if we only have binary semaphores.

9.5.4.1 POSIX semaphores

Multiple POSIX threads can share a counting semaphore that can be shared through a global variable.

```
int sem_init(    sem_t *p_this, int shared, unsigned int value );
int sem_destroy( sem_t *p_this );

int sem_wait( sem_t *p_this );
int sem_post( sem_t *p_this );

int sem_getvalue( sem_t *p_this, int *p_value );
```

The last function signature is interesting, as the value of semaphore could theoretically change even between the execution of the command and reading the value at the given memory location.

```
#include <semaphore.h>
sem_t mutex;
sem_init( &mutex, 0, 1 );


sem_wait( &mutex ); {
    // Critical section...
} sem_post( &mutex );
```

If you are not using shared memory (different processes do not share memory, but multiple threads in the same process do), then the semaphores must be set up in a shared memory location. This must be prepared separately. An excellent article describing such a set-up is

http://blog.superpat.com/2010/07/14/semaphores-on-linux-binary_semaphore_init-vs-sem_open/.

9.5.4.2 Keil RTX RTOS semaphores

The Keil RTX RTOS has both mutual exclusion and counting semaphores.[25] The binary semaphore is named after its primary application: achieving mutual exclusion:

```
#include <rtl.h>

// Declaration
OS_MUT mutex;

// During initialization of the system
os_mut_init( mutex );

// In the task seeking mutual exclusion
os_mut_wait( mutex, 0xffff ); {
    // Critical section...
} os_mut_release( mutex );
```

The `OS_RESULT os_mut_wait( OS_MUT, U16 )` can take a second argument, a 16-bit unsigned integer, which indicates the number of *system intervals* that it will wait before it times out. A system interval has a default value of 10 ms, but this is configurable. The argument `0xffff` indicates to wait forever. The return value indicates what occurred:

`OS_R_MUT`	The task was blocked before the binary semaphore was available.
`OS_R_TMO`	The timeout expired before the binary semaphore became available.
`OS_R_OK`	The binary semaphore was immediately available and the function returned immediately.

The counting semaphore is simply described as a semaphore:

```
#include <RTL.h>

// Declaration
OS_SEM entries_available, entries_occupied;

// During initialization of the system
os_sem_init( entries_available, tokens );
os_sem_init( entries_occupied,  0 );

// In the task requiring a counting semaphore
os_sem_wait( entries_available, 0xffff );
push( &pipe, obj );  // A pipe is a queue used for communication between tasks
os_sem_send( entries_occupied );

os_sem_wait( entries_occupied, 0xffff );
obj = pop( &pipe );  // A pipe is a queue used for communication between tasks
os_sem_send( entries_available );
```

The `OS_RESULT os_sem_wait( OS_SEM, U16 )` can take a second argument, a 16-bit unsigned integer, which indicates the number of *system intervals* that it will wait before it times out. A system interval has a default value of 10 ms, but this is configurable. The return value indicates what occurred:

`OS_R_SEM`	The task was blocked before a token was available.
`OS_R_TMO`	The timeout expired before a token became available.
`OS_R_OK`	A token was immediately available and the function returned immediately.

[25] These details are taken from the RL-ARM User's Guide (MDK v4) Library Reference, available at http://www.keil.com/support/man/docs/rlarm/rlarm_library.htm.

9.5.4.3 CMSIS-RTOS RTX semaphores

The CMSIS-RTOS RTX provides mutual exclusion and counting semaphores for all Cortex-M processors, but they also include a third serialization mechanism: event signaling.[26] Mutual exclusion is achieved through the mutex data structure:

```
#include "cmsis_os.h"

// Declaration
osMutexID mutex_id;
osMutexDef( mutex );  // Macro

// During initialization of the system
mutex_id = osMutexCreate( osMutex( mutex ) );
if ( mutex_id == NULL ) { /* failure in creating the mutex */ }

// In the task seeking mutual exclusion
osStatus status;

status = osMutexWait( mutex_id, uint32_t timeout );

if ( status == osOK ) {
    // Critical section

    status = osMutexRelease( mutex_id );

    if ( status != osOK ) { /* failure in releasing the mutex */ }
} else {
    // Failure in mutex acquisition
}

// the mutex is no longer required
status = osMutexDelete( mutex_id );
if ( status != osOK ) { /* failure in deleting the mutex */ }
```

In the case of the `osMutWait(…)` function, the second parameter is `uint32_t timeout` where a value of `0` indicates that it should return instantly (essentially, return the value of the mutex), the defined symbol `osWaitForever` indicates it should wait forever, and any other value is the number of milliseconds it should wait for a semaphore (returning 0 if mutual exclusion was acquired).

The return value of `osMutexWait` is one of an enumerated type `osStatus`, the relevant values of which are:

`osOK`	The mutex was acquired.
`osErrorTimeoutResource`	The wait timed out before a mutex could be acquired.
`osErrorResource`	It was not possible to obtain the mutex when `osWaitForever` was specified.
`osErrorParameter`	The argument `mutex_id` is invalid.
`osErrorISR`	The wait was incorrectly called from an interrupt service routine.

The return values of `osMutexRelease` and `osMutexDelete` includes these except for `osErrorTimeoutResource`.

[26] These details are taken from the CMSIS-RTOS RTX documentation, available at https://www.keil.com/pack/doc/CMSIS/RTX/html/.

The CMSIS-RTOS RTX also has counting semaphores:

```
#include "cmsis_os.h"

// Declaration
osSemaphoreID entries_available, entries_occupied;
osSemaphoreDef( entries_available );  // Macro
osSemaphoreDef( entries_occupied );   // Macro

// During initialization of the system
entries_available = osSemaphoreCreate( osSemaphore( semaphore ), tokens );
if ( entries_available == NULL ) { /* error in semaphore creation */ }

entries_occupied  = osSemaphoreCreate( osSemaphore( semaphore ), 0 );
if ( entries_occupied == NULL ) { /* error in semaphore creation */ }

// In the task requiring a counting semaphore
int32_t tokens;
osStatus status;

tokens = osSemaphoreWait( entries_available, uint32_t timeout );

if ( tokens > 0 ) {
    push( &pipe, obj );  // A pipe is a queue used for communication between tasks

    status = osSemaphoreRelease( entries_occupied );
    if ( status != osOK ) { /* failure in releasing the semaphore */ }
} else {
    // Failure to acquire the semaphore
}

tokens = osSemaphoreWait( entries_occupied, uint32_t timeout );

if ( tokens > 0 ) {
    obj = pop( &pipe );  // A pipe is a queue used for communication between tasks

    status = osSemaphoreRelease( entries_available );
    if ( status != osOK ) { /* failure in releasing the semaphore */ }
} else {
    // Failure to acquire the semaphore
}

// the semaphores are no longer required
status = osSemaphoreDelete( entries_available );
if ( status != osOK ) { /* failure in deleting the semaphore */ }

status = osSemaphoreDelete( entries_occupied );
if ( status != osOK ) { /* failure in deleting the semaphore */ }
```

In the case of the `osSemphoreWait(…)` function, the second parameter is `uint32_t millisec` where a value of `0` indicates that it should return instantly (essentially, return the value of the semaphore), the defined symbol `osWaitForever` indicates it should wait forever, and any other value is the number of milliseconds it should wait for a semaphore (returning 0 if no semaphore was acquired). The functions `osSemaphoreRelease` and `osSemaphoreDelete` have similar return values to those returned by the mutexes described previously.

The CMSIS-RTOS RTX has an additional mechanism for synchronization: signals. Each thread is associated with up to sixteen flags represented by the sixteen bits `0x0001`, `0x0002`, `0x0004`, `0x0008`, `0x0010`, …, `0x4000`, `0x8000` (labeled as flags 0 through 15, respectively). It is possible to set or clear the flags any combination of these associated with any other thread (provided you have its `osThreadID`) and the currently executing thread can wait on any combination of its flags until they have been set (most likely by other tasks).

Suppose that `thread_id` is a thread (task) that is waiting for signals from other tasks. Other tasks could then signal that thread:

```
int32_t  signals = osSignalSet( thread_id, 0x00000035 ); // Set flags 0, 2, 4, 5
```

The return value is the previous state of the flags for that thread or `0x80000000` (the largest negative 32-bit integer) if the arguments were invalid (the thread does not exist or more than 16 flags were set).

It is also possible to clear flags in the signals of another task:

```
int32_t  signals = osSignalClear( thread_id, 0x00001086 ); // Clear flags 1, 3, 7, 12
```

The return value is the previous state of the flags for that thread or `0x80000000` (the largest negative 32-bit integer) if either the arguments were invalid (the thread does not exist or more than 16 flags were cleared) or this function was called from an interrupt service routine (ISR).

A thread can now wait on its own flags by calling

```
osEvent event = osSignalWait( 0x00001004, unit32_t timeout ); // Wait on flags 2 and 12
```

The first argument can be `0`, indicating that it should wait until at least one flag is set, although it is not possible to determine which signals were set. The second argument, `timeout`, specifies the number of milliseconds that the function should wait before returning unsuccessfully. If the task is to wait forever, use `osWaitForever` to identify this. If this function returns successfully, the flags that were waited on are cleared.

The return value of `osSignalWait` is an `osEvent` structure containing three fields:

`osStatus status;`	The return status, similar to semaphores and mutexes.
`union { int32_t signals; … } value;`	The signal flags, other types are associated with messaging.
`union {…} def;`	Only associated with messaging.

The value of the status is one of:

`osOK`	Returned only if no signal was received when the timeout was 0 ms.
`osEventTimeout`	The signal was not sent by the end of the timeout.
`osEventSignal`	A signal occurred and the `value.signals` field contains the flags prior to clearing.
`osErrorValue`	The first argument flags fields outside of the allowable 16 LSBs.
`osErrorISR`	The wait was incorrectly called from an interrupt service routine.

9.5.4.4 Implementing counting semaphores with binary semaphores

We will now consider the problem of implementing counting semaphores using binary semaphores. We will require a count of tokens available, one semaphore for mutual exclusion and one semaphore for waiting.

```
typedef struct {
    size_t tokens;
    binary_semaphore_t mutex;
    binary_semaphore_t waiting_tasks;
} counting_semaphore_t;

counting_semaphore_init( counting_semaphore_t *const p_this, size_t init ) {
    p_this->tokens = init;
    binary_semaphore_init( &( p_this->mutex ), 1 );
    binary_semaphore_init( &( p_this->waiting_tasks ), 0 );
}
```

The following is a naïve implementation which you might think does the job:

```
void counting_semaphore_wait( counting_semaphore_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        --( p_this->tokens );

        if ( p_this->tokens < 0 ) {
            binary_semaphore_post( &( p_this->mutex ) );           // Hint
            binary_semaphore_wait( &( p_this->waiting_tasks ) );
        } else {
            binary_semaphore_post( &( p_this->mutex ) );
        }
    }
}

void counting_semaphore_post( counting_semaphore_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        ++( p_this->tokens );

        if ( p_this->tokens <= 0 ) {
            binary_semaphore_post( &( p_this->waiting_tasks ) );
        }
    } binary_semaphore_post( &( p_this->mutex ) );
}
```

Unfortunately, this implementation cannot work. Why? Recall that a binary semaphore does not have a memory greater than one, and suppose that one task is already waiting on this semaphore and now a second task issues a wait, but an interrupt occurs immediately after the executing of the line marked *hint*. A correct implementation of these two functions requires a different mechanisms for enforcing mutual exclusion. Because we generally regard `mutex` to represent a binary semaphore that can only be posted to by the task that waited on it, we will rename the field `restrict`:

```
typedef struct {
    size_t tokens;
    binary_semaphore_t restrict;
    binary_semaphore_t waiting_tasks;
} counting_semaphore_t;

counting_semaphore_init( counting_semaphore_t *const p_this, size_t init ) {
    p_this->tokens = init;
    binary_semaphore_init( &( p_this->restrict ), 1 );
    binary_semaphore_init( &( p_this->waiting_tasks ), 0 );
}

void counting_semaphore_wait( counting_semaphore_t *const p_this ) {
    binary_semaphore_wait( &( p_this->restrict ) ); {
        --( p_this->tokens );

        if ( p_this->tokens < 0 ) {
            binary_semaphore_post( &( p_this->access ) );
            binary_semaphore_wait( &( p_this->waiting_tasks ) );
        }
    } binary_semaphore_post( &( p_this->restrict ) );
}

void counting_semaphore_post( counting_semaphore_t *const p_this ) {
    binary_semaphore_wait( &( p_this->restrict ) );
    ++( p_this->tokens );

    if ( p_this->tokens <= 0 ) {
        counting_semaphore_post( &( p_this->waiting_tasks ) );
    } else {
        binary_semaphore_post( &( p_this->restrict ) );
    }
}
```

9.5.4.5 Summary of implementations of semaphores

In this topic, we looked at various implementations of semaphores, and there are variations and nuances to the implementations. We considered POSIX, Keil RTX RTOS and CMSIS-RTOS RTX semaphore implementations. As well, we looked at how counting semaphores can be created using binary semaphores.

9.5.5 A summary of semaphores

In this topic, we have described binary semaphores and counting semaphores. We will continue by looking at problems in synchronization and how they can be solved with semaphores.

9.6 Problems in synchronization

We will now look at three categories of problems in synchronization, including

1. basic,
2. intermediate, and
3. advanced.

problems in synchronization. We will discuss these over the next three sections.

9.6.1 Basic problems in synchronization

We have already discussed mutual exclusion as the first aspect of synchronization, but we will consider one further issue. The second is serialization. There are two basic serialization patterns that will form the basis of other solutions:

1. signalling, and
2. rendezvous.

We will look at solutions for this problem when there are two tasks.

9.6.1.1 Mutual exclusion

Binary semaphores can be used for mutual exclusion, but there is one weakness. Any task can issue a post to a binary semaphore; however, if the data structure tracks the task or thread that issues the wait on the binary semaphore and only allows that task or thread to post, such a data structure is often referred to as a *mutual exclusion data structure*, or *mutex*. and often it comes with an additional safeguard: the default value is always 1, and the only task or thread that can post is the one that waited on it. The Petri Net for two tasks sharing a semaphore for mutual exclusion is shown in Figure 9-10.

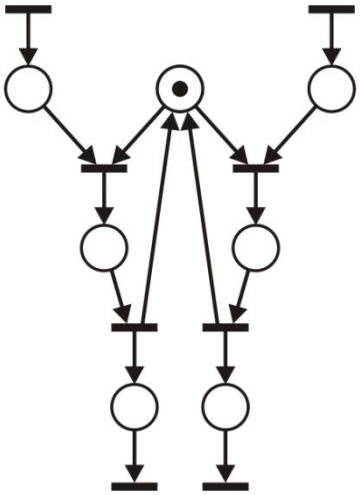


Figure 9-10. A Petri Net for mutual exclusion. The semaphore begins with a token.

As a quick review, mutual exclusion with semaphores may be achieved through a shared binary semaphore initialized to 1:

```
binary_semaphore_t mutex;
binary_semaphore_init( &mutex, 1 );
```

Then, any critical region need only be preceded by a wait and followed by a post:

```
binary_semaphore_wait( &mutex ); {
    // Critical region
```

```
binary_semaphore_post( &mutex );
```

9.6.1.2 Signalling

A semaphore can be used by one task to signal that an event has occurred. Any task waiting on that signal can then proceed in its execution. To achieve a signal, the semaphore is initialized to zero and the signalling semaphore will post to that semaphore to signal the event. Any task that waits on that signal before it is sent would be blocked for execution until the signal occurs. Any task that waits on that signal after it is sent would, likewise, continue in its execution. A Petri Net of a signal is shown in Figure 9-11.

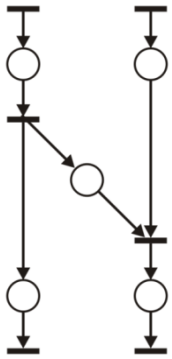


Figure 9-11. Petri Net of a signal.

The implementation is straight-forward:

```
binary_semaphore_t signal;
binary_semaphore_init( &signal, 1 );

void task_1( void ) {
    // ...

    // Prepare data
    binary_semaphore_post( &signal );

    // ...
}

void task_2( void ) {
    // ...

    binary_semaphore_wait( &signal );
    // Process data

    // ...
}
```

9.6.1.3 Rendezvous

Suppose we have two tasks executing, but neither should continue executing beyond a certain point until both have gotten to that point.

```
void task_0( void ) {
    // Prepare data...
    // Rendezvous
    // Process data...
}

void task_1( void ) {
    // Prepare data...
    // Rendezvous
    // Process data...
}
```

The Petri Net for this is actually simpler, as shown in Figure 9-12.

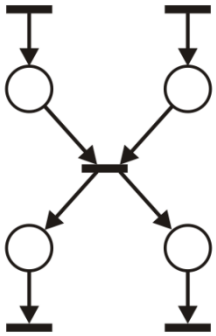


Figure 9-12. Petri Net for a two-task rendezvous.

In order to solve this problem, let us recall that there are two events here that require signalling:

1. The first event is that Task 0 is ready for the rendezvous, and
2. The second event is that Task 1 is ready for the rendezvous.

As soon as soon as each task reaches its rendezvous point, it must signal on a separate semaphore that it is completed. Each then waits for the other:

```
binary_semaphore_t signal[2];
binary_semaphore_init( &signal[0], 0 );
binary_semaphore_init( &signal[1], 0 );

void task_0( void ) {
    // Prepare data...
    binary_semaphore_post( &signal[0] );   // Signal that this task is ready
    binary_semaphore_wait( &signal[1] );   // Wait for the other task
    // Process data...
}

void task_1( void ) {
    // Prepare data ...
    binary_semaphore_post( &signal[1] );   // Signal that this task is ready
    binary_semaphore_wait( &signal[0] );   // Wait for the other task
    // Process data ...
}
```

Problem: Suppose you coded the above example in slightly a different way:

```
binary_semaphore_t signal[2];
binary_semaphore_init( &signal[0], 0 );
binary_semaphore_init( &signal[1], 0 );

void task_0( void ) {
    // Prepare data...
    binary_semaphore_wait( &signal[1] );   // Wait for the other task
    binary_semaphore_post( &signal[0] );   // Signal that this task is ready
    // Process data...
}

void task_1( void ) {
    // Prepare data...
    binary_semaphore_wait( &signal[0] );   // Wait for the other task
    binary_semaphore_post( &signal[1] );   // Signal that this task is ready
    // Process data...
}
```

This is a second example where poor programming practice will result in deadlock.

9.6.1.4 Waiting on interrupts: an application of serialization

In the previous topic, we asked how a task or thread could wait for an interrupt to occur. We suggested the problem of checking a global variable, but this required polling. Instead, a task or thread waiting for an event to occur could, instead, wait on a semaphore. Then, in the ISR, it would post to that semaphore.

If there was a possibility that multiple interrupts could occur prior to the thread processing the data, the ISR could, for example, store the data in a queue, in which case, the data would be available.

If the result of an interrupt is only relevant to waiting requests on an interrupt, an *event* could be used, instead.

9.6.1.5 Summary of basic serialization

We have discussed a refinement of binary semaphores that prevent other tasks or threads from issuing a post to a held semaphore, and two straight-forward serialization problems between two tasks. For the serialization problems, in both cases, a binary semaphore could be used. We will now proceed to look at more complex problems with multiple tasks interacting. In many cases, we will use counting semaphores to help solve our problems.

9.6.2 Intermediate problems in synchronization

Now we will look at

1. multiplexing,
2. group rendezvous,
3. light-switch, and
4. condition variables.

The first involves a generalization of mutual exclusion and the next two are two possible generalizations of rendezvous. In all three cases, we will consider multiple tasks.

These next few chapters heavily rely on an excellent source: *The Little Book of Semaphores* by Allen B. Downey, published by Green Tea Press and available for free at www.greenteapress.com/semaphores/. It is strongly recommended you reference this text book if you have any further issues with synchronization.

9.6.2.1 Multiplexing

A multiplex is mutual exclusion where at most *n* items can access the critical section. This is a generalization of the concept of mutual exclusions, except now up to *n* tasks can access the area. Justification for this may be processor usage: if more than *n* tasks are performing a processor-intensive operation, it degrades the quality of service for all tasks. Restaurants providing services do this all the time: the building has a maximum capacity based on the number of available seats. Allowing more people into the restaurant than the number of seats available will only reduce the quality of service.

The Petri Net for a multiplex is similar to that for mutual exclusion, only there are *n* tokens available, as is shown in Figure 9-13.

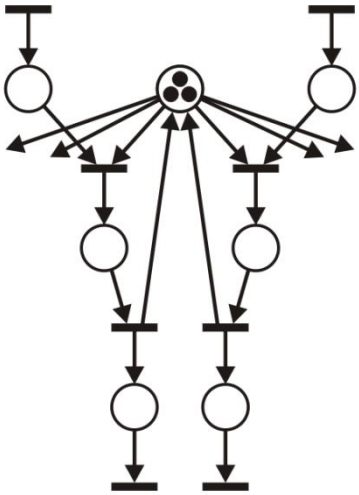


Figure 9-13. A Petri Net for a 3-plex with six possible tasks (two explicitly drawn).

To achieve this, we replace the binary semaphore with a counting semaphore:

```
counting_semaphore_t multiplex;
counting_semaphore_init( &multiplex, n );

counting_semaphore_wait( &multiplex ); {
    // Shared critical region
} counting_semaphore_post( &multiplex );
```

We use the same formatting convention to have the shared critical region stand out.

9.6.2.2 Group rendezvous and turnstile

The rendezvous we discussed works well for synchronizing two tasks, but it does not work in general for an arbitrary number of tasks. You could have one semaphore for each task, but this would be excessive. Instead, we would like a general mechanism to allow *n* tasks rendezvousing, and only after all *n* tasks have gotten to that point are they all allowed to continue. We will look at:

1. how to solve this problem,
2. the turnstile data structure,
3. the group rendezvous data structure, and
4. implementing the solution with these data structures.

The Petri Net for a group rendezvous is, like a two-task rendezvous, is also straight-forward, as is shown in Figure 9-14.

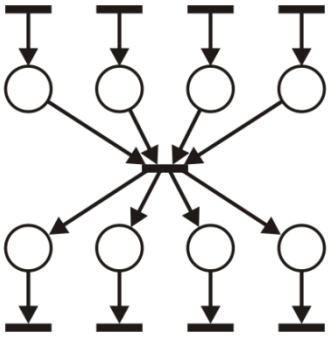


Figure 9-14. A Petri Net for a group rendezvous.

9.6.2.2.1 A sub-optimal solution

Your first thought might be to have $n - 1$ wait on a semaphore, and the last to reach the rendezvous would issue $n - 1$ posts to allow the waiting tasks to pass through the turnstile.

```
binary_semaphore_t mutex;
binary_semaphore_init( &mutex, 1 );             // For mutually exclusive access to
tasks_waiting

binary_semaphore_t rendezvous_point;
binary_semaphore_init( &rendezvous_point, 0 );  // A semaphore initially 0
                                                //  - anything waiting on
                                                //    it will be blocked

void task( void ) {
    // Executing...

    binary_semaphore_wait( &mutex ); {
        --tasks_not_at_rendezvous;

        if ( tasks_not_at_rendezvous == 0 ) {
            for ( i = 1; i < n; ++i ) {
                 binary_semaphore_post( &rendezvous_point );
            }

            binary_semaphore_post( &mutex );
        } else {
            binary_semaphore_post( &mutex );
            binary_semaphore_wait( &rendezvous_point );
        }
    }

    // Continue execution...
}
```

The problem with this is that the last task is likely also the lowest in priority. It may happen that the first post requires a context switch to the highest priority task. Then, when it is finished, you must return to the posting task, which then issues its second post. This is unnecessarily expensive, as it requires, in the worst case, $2n$ context switches. Instead, we'd like to reduce this to n context switches.

9.6.2.2.2 A better solution

To achieve this, there must be a global variable storing the number of tasks that are to rendezvous:

```
size_t tasks_not_at_rendezvous = ALL_TASKS;
                    // the number of tasks executing the rendezvous
```

One solution may be to have all the tasks wait until the last one gets there, and it will release all other tasks. This, however, causes more problems. Instead, we will simply have the last task arriving let a task through, and then each subsequent task that is let through will signal the next, and so on. Such an approach is said to be a *turnstile*.

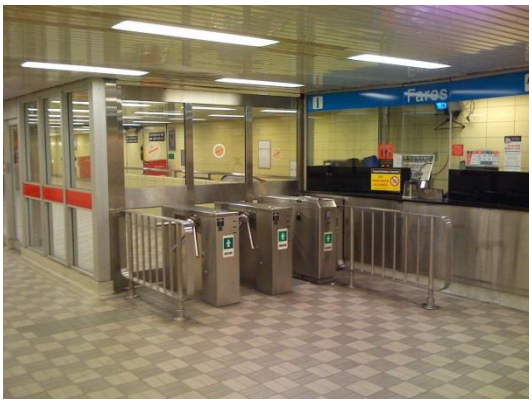


Figure 9-15. Turnstiles to enter the subway (TTC, Toronto, photograph by Sweet One / Neal Jennings).

Here is an implementation for a single rendezvous using a turnstile:

```
binary_semaphore_t mutex;
binary_semaphore_init( &mutex, 1 );             // For mutually exclusive access to
tasks_waiting

binary_semaphore_t rendezvous_point;
binary_semaphore_init( &rendezvous_point, 0 );  // A semaphore initially 0
                                        //  - anything waiting on it will be blocked

void task( void ) {
    // Executing...

    binary_semaphore_wait( &mutex ); {
        --tasks_not_at_rendezvous;

        if ( tasks_not_at_rendezvous == 0 ) {
            binary_semaphore_post( &rendezvous_point );
        }
    } binary_semaphore_post( &mutex );

    binary_semaphore_wait( &rendezvous_point );
    binary_semaphore_post( &rendezvous_point );

    // Continue execution...
}
```

Question: what if we put the statements

```
    binary_semaphore_wait( &rendezvous_point );
    binary_semaphore_post( &rendezvous_point );
```

inside the critical region?

9.6.2.2.3 A reusable group rendezvous (the current optimal design pattern)

This solution will work for both binary semaphores and for counting semaphores. One issue with this solution, however, is that this turnstile can only be used once. One possible solution described by Allan B. Downey is to have each task wait twice:

1. all wait at the first turnstile decrementing the count until the last gets there, who lets them through, and then
2. all wait at a second turnstile incrementing the count until the last gets there, who lets them through again.

Thus, only these tasks can get through regardless of any other circumstances. Does the following solution work?

```
size_t tasks_not_at_rendezvous = ALL_TASKS;

binary_semaphore_t mutex;
binary_semaphore_init( &mutex, 1 );

binary_semaphore_t rendezvous_point;
binary_semaphore_init( &rendezvous_point, 0 );


binary_semaphore_wait( &mutex ); {
    --tasks_not_at_rendezvous;

    if ( tasks_not_at_rendezvous == 0 ) {
        binary_semaphore_post( &rendezvous_point );
    }
} binary_semaphore_post( &mutex );

binary_semaphore_wait( &rendezvous_point );
binary_semaphore_post( &rendezvous_point );

    // Rendezvous point

binary_semaphore_wait( &mutex ); {
    ++tasks_not_at_rendezvous;

    if ( tasks_not_at_rendezvous == ALL_TASKS ) {
        binary_semaphore_wait( &rendezvous_point );
                                        // Use the last signal on the turnstile
    }
} binary_semaphore_post( &mutex );
```

We, however, still have a problem: what happens if one of the threads has a very high priority, so any chance it gets to run, it keeps running while all other tasks wait—at least, until it blocks on a semaphore.

```
size_t tasks_not_at_rendezvous = ALL_TASKS;

binary_semaphore_t mutex;
binary_semaphore_init( &mutex, 1 );

binary_semaphore_t rendezvous_point_enter;
binary_semaphore_init( &rendezvous_point_enter, 0 );

binary_semaphore_t rendezvous_point_exit;
binary_semaphore_init( &rendezvous_point_exit, 1 ); // Initially, the exit is open


binary_semaphore_wait( &mutex ); {
    --tasks_not_at_rendezvous;
```

```
    if ( tasks_not_at_rendezvous == 0 ) {
        binary_semaphore_wait( &rendezvous_point_exit );  //   Lock the 2nd turnstile
        binary_semaphore_post( &rendezvous_point_enter ); // Unlock the 1st turnstile
    }
} binary_semaphore_post( &mutex );

binary_semaphore_wait( &rendezvous_point_enter );
binary_semaphore_post( &rendezvous_point_enter );

    // Rendezvous point

binary_semaphore_wait( &mutex ); {
    ++tasks_not_at_rendezvous;

    if ( tasks_not_at_rendezvous == ALL_TASKS ) {
        binary_semaphore_wait( &rendezvous_point_enter );  //   Lock the 1st turnstile
        binary_semaphore_post( &rendezvous_point_exit );   // Unlock the 2nd turnstile
    }
} binary_semaphore_post( &mutex );

binary_semaphore_wait( &rendezvous_point_exit );
binary_semaphore_post( &rendezvous_point_exit );
```

This is called a *two-phase barrier*. As you may suspect, there are significant opportunities to make errors in the coding of such a barrier. Consequently, it is probably a good idea, as this forms a single block of code, to write a separate barrier structure. Note, however, that we may want both: a turnstile and a rendezvous point. We will implement both.

9.6.2.2.4 The turnstile data structure

The turnstile structure has three interfaces:

1. lock the turnstile,
2. unlock the turnstile, and
3. pass through the turnstile.

A single semaphore controls all of these:

```
#define TURNSTILE_LOCKED   true
#define TURNSTILE_UNLOCKED false

typedef struct {
    binary_semaphore_t turnstile;
} turnstile_t;

void turnstile_init( turnstile_t *const p_this, bool locked ) {
    binary_semaphore_init( &( p_this->turnstile ), locked ? 0 : 1 );
}

void turnstile_unlock( turnstile_t *const p_this ) {
    binary_semaphore_post( &( p_this->turnstile ) );
}

void turnstile_lock( turnstile_t *const p_this ) {
    binary_semaphore_wait( &( p_this->turnstile ) );
}

void turnstile_pass( turnstile_t *const p_this ) {
    binary_semaphore_wait( &( p_this->turnstile ) );
    binary_semaphore_post( &( p_this->turnstile ) );
}
```

We can now use this in our implementation of the group rendezvous.

Aside: Which of these two is `&data->field` equivalent to?

1. `(&data)->field`
2. `&( data->field )`

It happens that in the above example we don't necessarily have to use the parentheses, but the average programmer may not know this. It is better to be very clear as to what your intentions are, rather than relying on an implicit knowledge of operator precedence. As it turns out, the first one would be equivalent to saying `data.field`, so it may seem obvious, but on the other hand, which of the following is `*p_data.field` equivalent to?

1. `(*p_data).field`
2. `*( p_data.field )`

In this case, the desirable one is the first, but it is parsed as the second (`.` and `->` have precedence level 2, just above `*` and `&` that have precedence level 3). This is one of the reasons for having the `data->field` operator.

9.6.2.2.5 The reusable group rendezvous data structure

The group rendezvous uses two turnstiles.

```
typedef struct {
    size_t capacity;
    size_t waiting;
    binary_semaphore_t mutex;
    turnstile_t enter;
    turnstile_t exit;
} rendezvous_t;

void rendezvous_init( rendezvous_t *const p_this, size_t n ) {
    p_this->capacity = n;
    p_this->waiting = 0;

    binary_semaphore_init( &( p_this->mutex ), 1 );
    turnstile_init( &( p_this->enter ), TURNSTILE_LOCKED );
    turnstile_init( &( p_this->exit ),  TURNSTILE_UNLOCKED );
}

void rendezvous_wait( rendezvous_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        ++( p_this->waiting );

        if ( p_this->waiting == p_this->capacity ) {
            turnstile_lock(   &( p_this->exit  ) );
            turnstile_unlock( &( p_this->enter ) );
        }
    } binary_semaphore_post( &( p_this->mutex ) );

    turnstile_pass( &( p_this->enter ) );

    binary_semaphore_wait( &( p_this->mutex ) ); {
        --( p_this->waiting );

        if ( p_this->waiting == 0 ) {
            turnstile_lock(   &( p_this->enter ) );
            turnstile_unlock( &( p_this->exit  ) );
        }
    } binary_semaphore_post( &( p_this->mutex ) );
```

```
    turnstile_pass( &( p_this->exit ) );
}
```

Now, each task need only wait on the corresponding rendezvous, leaving code that is relatively straight-forward to understand and maintain.

```
rendezvous_t waiting_point;             // Global variable

// task initialization
void init( void ) {
    // ...
    rendezvous_init( &waiting_point, 12 );  // In some initialization routine
    // ...
}

void task( void ) {
    // Initialization

    while ( 1 ) {
        // Executing ...
        rendezvous_wait( &waiting_point );
        // Continue execution...
    }
}
```

Graphically, we can view this as in Figure 9-16 where the following occurs:

1. The tasks approach the rendezvous.
2. At some point, the last task approaches the rendezvous.
3. That last task unlocks the entrance and locks the exit of the next waiting area.
4. The tasks proceed to pass through the turnstile.
5. At some point, the last task passes through the turnstile.
6. That task locks the entrance and unlocks the exit of the waiting area.
7. The tasks now leave the rendezvous.

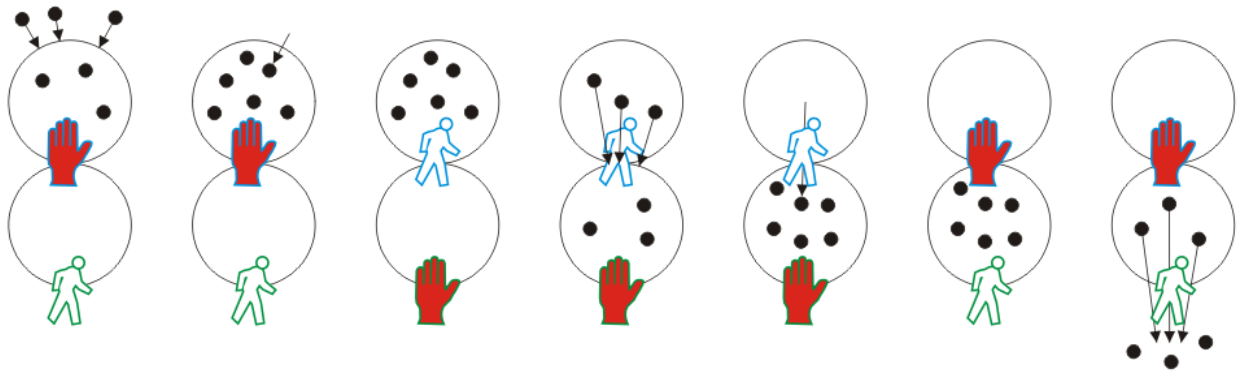


Figure 9-16. The operations of a group rendezvous with two turnstiles.

Once the tasks leave the rendezvous, they have the option of using it again.

9.6.2.2.6 Summary of the group rendezvous

In this section, we looked at the group rendezvous. We considered a sub-optimal solution and then considered the concept of a turnstile. We created data structures for both turnstiles and the group rendezvous, and then we converted the simplified solution into one using the data structures. We will now look at the next design pattern: the light switch.

9.6.2.3 Light-switches

If a room is being used for an event, the first person who enters the room marks the room as used. Others there for the event may also enter the room, but the room is blocked from others not associated with the event. This can be thought of as *light-switch*: the first user entering turns on the light, the last user leaving turns off the light.

```
size_t population = 0;

binary_semaphore_t mutex;
binary_semaphore_init( & mutex, 1 );

binary_semaphore_t light_switch;
binary_semaphore_init( &light_switch, 1 );

void task() {
    // Executing...

    binary_semaphore_wait( &mutex ) {
        ++population;

        if ( population == 1 ) {
            binary_semaphore_wait( &light_switch );
        }
    } binary_semaphore_post( &mutex );

    // Critical area...

    binary_semaphore_wait( &mutex ); {
        --population;

        if ( population == 0 ) {
            binary_semaphore_post( &light_switch );
        }
    } binary_semaphore_post( &mutex );

    // Continue executing...
}
```

Let's wrap all of this up in a structure with associated functions.

```
typedef struct {
    size_t population;
    binary_semaphore_t mutex;
    binary_semaphore_t *p_access_control;
} lightswitch_t;

void lightswitch_init( lightswitch_t *const p_this, binary_semaphore_t *p_bs ) {
    p_this->population = 0;
    binary_semaphore_init( &( p_this->mutex ), 1 );
    p_this->p_access_control = p_bs;
}
```

```
void lightswitch_wait( lightswitch_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        ++( p_this->population );

        if ( p_this->population == 1 ) {
            binary_semaphore_wait( p_this->p_access_control );
                                                    // Why is this inside the mutex?
        }
    } binary_semaphore_post( &( p_this->mutex ) );
}

void lightswitch_post( lightswitch_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        --( p_this->population );

        if ( p_this->population == 0 ) {
            binary_semaphore_post( p_this->p_access_control );
        }
    } binary_semaphore_post( &( p_this->mutex ) );
}
```

Now, all tasks accessing the semaphore `room_available` through the light-switch `group_access` may do so together, instead of individually.

```
binary_semaphore_t room_available;
binary_semaphore_init( &room_available, 1 );

lightswitch_t group_access;
lightswitch_init( &group_access, &room_available );

void task( void ) {
    // Executing...

    lightswitch_wait( &group_access );

    // Critical area...

    lightswitch_post( &group_access );

    // Continue executing...
}
```

9.6.2.4 Events

A synchronization tool similar to a binary semaphore is an *event* (waiting for a situation, or condition, to occur), only an event does not have a memory—it never temporarily stores tokens. If a task posts to an event and there is no task waiting on that event, the occurrence of that event is lost.

The interface for an event is similar, but slightly different from that of a counting semaphore. While the wait function is similar, we are not *posting* tokens; instead, we are simply *signalling* any waiting tasks. In this case, we have two options:

1. signalling a single task waiting for the event to occur, or
2. signal all (*broadcast* to all) tasks waiting for the event to occur.

We can try to implement events using two binary semaphores.

```
typedef struct {
    size_t size;
    binary_semaphore_t mutex;
    counting_semaphore_t waiting;
} event_t;

void event_init( event_t *const p_this ) {
    p_this->size = 0;
    binary_semaphore_init( &( p_this->mutex ), 1 );
    counting_semaphore_init( &( p_this->waiting ), 0 );
}

void event_wait( event_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        ++( p_this->size );
    } binary_semaphore_post( &( p_this->mutex ) );

    counting_semaphore_wait( &( p_this->waiting ) );
}

void event_signal( event_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        if ( p_this->size > 0 ) {
            --( p_this->size );
            counting_semaphore_post( &( p_this->waiting ) );
        }
    } binary_semaphore_post( &( p_this->mutex ) );
}

void event_broadcast( event_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        while ( p_this->size > 0 ) {
            --( p_this->size );
            counting_semaphore_post( &( p_this->waiting ) );
        }
    } binary_semaphore_post( &( p_this->mutex ) );
}
```

What are the problems with this implementation? Recall that any task that waits on the event after a signal or broadcast should not be released.

As an aside: In C, it is good practice to initialize the variables in the order in which they appear defined in the corresponding structure. This helps with readability. Where this becomes important, however, is in C++ where automatic initialization of member variables is in the order in which they are defined, and not in the order in which they appear in the initialization list.

```
class Class_name {
    private:  int x;
              int y;
    public:   Class_name():y(1), x(y + 1) {
                  // 'x' is assigned first before 'y' has been assigned 1
              }
};
```

Question: Suppose that the following situation occurs:

1. a low priority task waits on a counting variable, but after incrementing count and posting the mutual exclusion,
2. an event occurs, and therefore the event is posted,
3. a high priority task now waits on the event, it increments the count and waits on the counting semaphore, which currently is set to 1, so it immediately continues—even though the event occurred before the high priority task waited on the counting variable.

In this case, the wait should have only woken up low priority task—the high priority task should have waited for the next event. How would you propose fixing this?

```
typedef struct {
    size_t size, freed;
    binary_semaphore_t mutex, waiting;
    turnstile_t entrance;
} event_t;

void event_init( event_t *const p_this ) {
    p_this->size = 0;
    binary_semaphore_init( &( p_this->mutex ), 1 );
    binary_semaphore_init( &( p_this->waiting ), 0 );
    turnstile_init( &( p_this->entrance ), TURNSTILE_UNLOCKED );
}

void event_wait( event_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        turnstile_pass( &( p_this->entrance ) );
        ++( p_this->size );
    } binary_semaphore_post( &( p_this->mutex ) );

    binary_semaphore_wait( &( p_this->waiting ) );

    --( p_this->freed );

    if ( p_this->freed == 0 ) {
        turnstile_unlock( &( p_this->enter ) );
    } else {
        binary_semaphore_post( &( p_this->waiting ) );
    }
}

void event_signal( event_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        if ( p_this->size > 0 ) {
            turnstile_lock( &( p_this->entrance ) );
            p_this->freed = 1;
            --( p_this->size );
            binary_semaphore_post( &( p_this->waiting ) );
        }
    } binary_semaphore_post( &( p_this->mutex ) );
}

void event_broadcast( event_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        if ( p_this->size > 0 ) {
            turnstile_lock( &( p_this->entrance ) );
            p_this->freed = p_this->size;
            p_this->size = 0;
            binary_semaphore_post( &( p_this->waiting ) );
        }
    } binary_semaphore_post( &( p_this->mutex ) );
}
```

This solution, the first author claims, is novel yet reasonably efficient. It uses the turnstile approach for exiting the event, where each task signals the next to be released from the event. This reduces the need for context switches. The use of a turnstile entrance prevents other tasks from waiting on this event while others are being signaled. Finally, the *waiting* semaphore acts as a mutual exclusion semaphore for the record fields.

9.6.2.5 Summary of intermediate problems in synchronization

In this section, we looked at some intermediate problems in synchronization, including the

1. multiplex, allowing up to *n* tasks to access a critical region;
2. group rendezvous, requiring *n* tasks to reach a rendezvous point before any can proceed;
3. light switch, turning off a switch when the first enters and turning it back on when the last leaves; and
4. events.

We will continue by looking at some more advanced problems in synchronization.

9.6.3 Advanced problems in synchronization

Now we will look at a number of advanced problems in synchronization. These are scenarios that occur in real life, but the initial instance in real life was a complex circumstance that was studied in depth, and the situation was distilled and analyzed to expose the general form of the problem. In addition, to aid communication of the issue, the situation is recast in a humorous, memorable scenario. Thus, while we may talk about the *dining philosopher's problem* in jest, the underlying problem is a reason for concern. We will look at:

1. the dining philosophers' problem, and
2. the readers-writers problem.

9.6.3.1 Dining philosophers' problem

Five philosophers are in a room with a round table with five plates of rice with a chopstick between each of the plates. The philosophers walk around the room, talk and think, and when they get hungry, they go to sit down, grab one of the chopsticks on either side of the plate, then grab the other chopstick on the other side of the plate, then eat, and then place both chopsticks back down again. This works well until all philosophers sit down nearly simultaneously and each grabs the left chopstick. As all chopsticks are currently in someone's hand, nobody can pick up the second chopstick they would need to begin eating. None of the philosophers can eat—that is, they will starve.

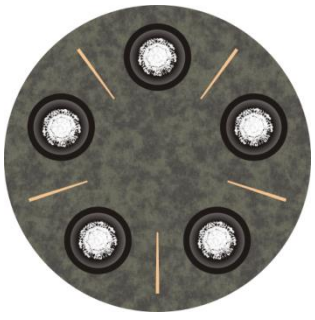


Figure 9-17. The philosophers' table.

What strategies could the philosophers use to ensure that:

1. they do not end up in a deadlock where none can eat, and
2. none of the philosophers starve.

Humour: Some text books have this problem posed with five forks and spaghetti instead of chopsticks and rice.

While it is rather painfully obvious that it would be difficult to eat anything other than perhaps sticky rick with only one chopstick, one wonders how one could reason that it is not possible to eat spaghetti with only one fork.

The Petri Net for some of the philosophers is as shown in Figure 9-18.

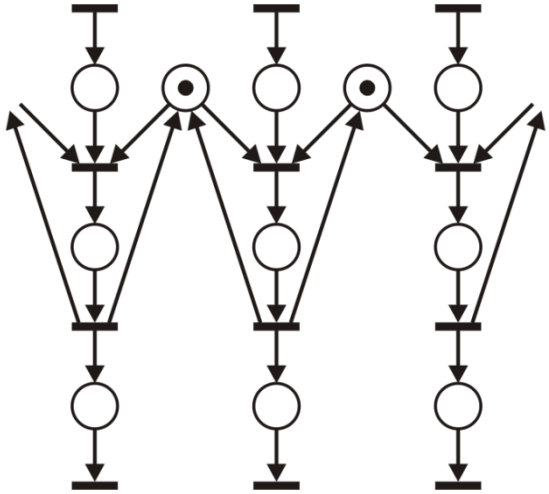


Figure 9-18. A component of the Petri net for the dining philosophers' problem.

9.6.3.1.1 First left, then right—deadlock

We could start by just having each philosopher lock the left chopstick, and then the right. If one is locked, that philosopher will wait until it is freed.

```
binary_semaphore_t chopstick_is_free[5];

int i;

for ( i = 0; i < 5; ++i ) {
    binary_semaphore_init( &( chopstick_is_free[i] ), 1 );
}

// Lock the left chopstick
size_t first( size_t i ) {
    return i;
}

// Lock the right chopstick
size_t second( size_t i ) {
    return (i + 1) % 5;
}

void philosopher( size_t n ) {
    while ( 1 ) {
        // Think...
        binary_semaphore_wait( &( chopstick_is_free[ first( n )] ) );
        binary_semaphore_wait( &( chopstick_is_free[second( n )] ) );
        // Eat...
        binary_semaphore_post( &( chopstick_is_free[ first( n )] ) );
        binary_semaphore_post( &( chopstick_is_free[second( n )] ) );
       // Keep thinking…
    }
}
```

We immediately run into a problem: suppose that every philosopher attempts to grab a chopstick at approximately the same time, so that each manages to lock the left chopstick. Now, all philosophers are locked on the other chopstick, so none will eat and—even worse—none will have a chance to think.

9.6.3.1.2 Accessing both chopsticks simultaneously

One possible solution is to try to grab both chopsticks at the same time, using the rules:

1. Try to grab both chopsticks at the same time,
2. If you cannot, flag yourself as hungry and wait for someone to nudge you,
3. If you can, lock both chopsticks, eat, unlock both, and nudge any neighbor who may be hungry.

Thus, we cannot end up with the situation that each philosopher has exactly one chopstick, and thus we avoid deadlock.

9.6.3.1.3 Ordering the resources

Another possible solution is to observe that there is a cycle that can develop as a result of the various locks. Each philosopher is holding on to a resource which the previous philosopher requires to proceed. By ordering the resources and requiring that each philosopher picks up the resources in order, it is not possible to generate a cycle and therefore it is possible to avoid deadlock:

```
// Lock the lower-ordered chopstick
size_t first( size_t i ) {
    return MIN( i, (i + 1) % 5 );
}

// Lock the higher-ordered chopstick
size_t second( size_t i ) {
    return MAX( i, (i + 1) % 5 );
}
```

Now, only one of Philosopher 0 and Philosopher 4 will be able to access Chopstick 0—the other will be blocked. Consequently, there are now only four philosophers trying to lock five chopsticks, so by the pigeonhole principle[27], at least one philosopher will be able to lock two chopsticks, eat, and release the chopsticks.

9.6.3.1.4 Restricting access to the table

Using multiplexing, another solution is to restrict the number of philosophers that have access to the table to four. In this case, again, the pigeonhole principle tells us that at least one of the four philosophers at the table will have access to two chopsticks.

9.6.3.1.5 Starvation

One question we must ask is, is it possible that a hungry philosopher may not be fed? This is always possible if there is a priority among the philosophers; however, these philosophers are egalitarian. In that case, is it possible for a philosopher to never get a chance to eat?

1. Suppose Philosopher 1 and 3 are eating, and Philosopher 0 comes to the table and promptly goes to sleep.
2. Philosopher 4 comes to the table and goes to sleep, after which Philosopher 3 leaves, waking up Philosopher 4 and he starts eating.
3. Philosopher 2 comes to the table and goes to sleep, after which Philosopher 1 leaves. Philosopher 1 may try to wake up Philosopher 0, but Philosopher 4 has Philosopher 0's other chopstick, so Philosopher 0 goes back to sleep. Philosopher 2 is woken up and starts eating.
4. Philosopher 1 comes to the table and goes to sleep, after which Philosopher 2 leaves, waking up Philosopher 1.

[27] The pigeonhole principle says that if there are *n* pigeonholes and there are more than *n* pigeons, then at least one pigeonhole must have at least two pigeons.

5. Philosopher 3 comes to the table and goes to sleep, after which Philosopher 4 leaves. Philosopher 4 may try to wake up Philosopher 0, but Philosopher 1 has Philosopher 0's other chopstick, so Philosopher 0 goes back to sleep. Philosopher 3 is woken up and starts eating.
6. Go back to Step 2.

In this case, Philosopher 0 starves to death. Solutions 9.6.3.1.3 and 9.6.3.1.4 do not suffer from starvation. Why?

9.6.3.1.6 Summary of the dining philosophers' problem

The dining philosophers' problem is a fanciful means of describing a situation where multiple tasks may require more than one of shared resources. The problem is simplified to one where each task only requires two resources, but this can be generalized to more tasks requiring more than two shared resources. As long as a cycle exists, it is possible that one each task holds a resource that the next requires.

9.6.3.2 Readers-writers problem

When accessing any resource that may be both read and written to, it is often possible that any number of readers can have access to the resource simultaneously, but only one writer at a time can ever change the resource. This may apply to something as simple as a data structure, or something more complex such as a file or database. Consider the C++ `const` modifier for member functions: there is no reason that two or more separate tasks could not call such functions simultaneously; however, as we have previously seen, it would be potentially disastrous if two different tasks tried to modify the data structure simultaneously.

We will consider a number of variations on this scenario:

1. readers may access the resource so long as there is no writer accessing the resource, and writers may access the resource so long as there are no readers or writers accessing the resource;
2. FCFS manner, where any number of readers may appear and are granted access to the critical section, but as soon as a writer appears, it waits until all readers have finished accessing the resource and any subsequent readers or writers must wait until this writer completes;
3. reader-priority where all writers that are not currently accessing the resource must wait so long as even one reader is waiting to access the resource; and
4. writer-priority where all readers that are not currently accessing the resource must wait so long as even one writer is waiting to access the resource.

9.6.3.2.1 The default problem

We have two tasks, one reading a resource, the other writing to the resource.

```
void reader( void ) {
    // Initialization

    while ( true ) {
        // Executing...
        // Get lock on resource
        // Read the resource
        // Release the resource
        // Continue execution...
    }
}
```

```
void writer( void ) {
    // Initialization

    while ( true ) {
        // Executing...
        // Get lock on resource
        // Write the resource
        // Release the resource
        // Continue execution...
    }
}
```

What do we require?

1. When one writer is accessing the resource, nothing else can access the resource.
2. When one or more readers are accessing the resource, additional readers can access the resource, but no writers can access the resource.

One semaphore can regulate access to the resource, and a light switch can be used to regulate access for readers. Most published implementations attempt to recreate the light switch; however, why reinvent the wheel?

```
binary_semaphore_t resource_mutex;
binary_semaphore_init( &resource_mutex, 1 );

lightswitch_t reader_access;
lightswitch_init( &reader_access, &resource_mutex );

void writer( void ) {
    // Initialization

    while ( true ) {
        // Executing...

        binary_semaphore_wait( &resource_mutex ); {
            // Critical region:  write to the resource
        } binary_semaphore_post( &resource_mutex );
    }
}

void reader( void ) {
    // Initialization

    while ( true ) {
        // Executing...

        lightswitch_wait( &reader_access ); {
            // Critical region:  read the resource
        } lightswitch_post( &reader_access );
    }
}
```

Now,

1. if a second writer comes along and waits on this semaphore while another writer is currently in the critical section, it will wait until the semaphore is posted, but
2. as soon as one reader waits on the light switch, there is a wait on the semaphore resource mutex, and this will block subsequent writers. However, if subsequent readers appear, they will be permitted into the critical region.

This solution, however, does have an issue in that it may starve the writers: suppose readers and writers come in at approximately intervals of once a minute, but the readers take 0.98 s to read their data while writers require only 0.1 s to modify their component of the resource. One may argue that this system is still functional; however, in the short term, it may happen that a sequence of *n* readers may come alone at slightly less than one-second intervals, in which case, there may be approximately *n* writers waiting on those readers to complete. If the system is overloaded; that is, the readers are coming in more frequently than they are completing their access to the resource, the writers may starve arbitrarily long. Let's try to fix this so that tasks wanting to access the resource are ordered FCFS.

9.6.3.2.2 Readers wait for one writer

Suppose *n* readers are currently accessing the resource and one writer appears. If the writer does not block new readers, the writer may have to wait forever. Consequently, we will force the readers to pass through a turnstile. If a writer ever appears, the writer will lock the turnstile

One semaphore can regulate access to the resource, and a light switch can be used to regulate access for readers.

```
binary_semaphore_t resource_mutex;
binary_semaphore_init( &resource_mutex, 1 );

turnstile_t reader_turnstile;
```

```
turnstile_init( &reader_turnstile, TURNSTILE_UNLOCKED );


lightswitch_t reader_access;
lightswitch_init( &reader_access, &resource_mutex );

void writer( void ) {
    // Initialization

    while ( true ) {
        // Executing...

        turnstile_lock( &reader_turnstile );
        binary_semaphore_wait( &resource_mutex );

        // Critical region:  write to the resource

        binary_semaphore_post( &resource_mutex );
        turnstile_unlock( &reader_turnstile );
    }
}

void reader( void ) {
    // Initialization

    while ( true ) {
        // Executing...

        turnstile_pass( &reader_turnstile );
        lightswitch_wait( &reader_access );

        // Critical region:  read the resource

        lightswitch_post( &reader_access );
    }
}
```

Now, if there are multiple writers, and those writers have a higher priority than the readers, then writers will always be able to access the document as soon as the last reader gives up that resource. If another writer appears while a writer has the resource, if its priority is higher than the priority of any readers that may have shown up between the two readers trying to access the document, the next writer—never-the-less—has priority.

Some other questions will appear on the homework problems.

9.6.3.2.3 Summary of the reader-writer problem

In this section, we have discussed the reader-writer problem where multiple readers may access a document, but any writer must have mutually exclusive access to the document. The problem with using just a light switch is that the readers may starve the writers; however, if updating the document is a priority, we can use a turnstile to prevent readers from accessing the resource if there are writers waiting.

9.6.3.3 Summary of advanced problems in synchronization

We have looked at two problems in synchronization:

1. the dining philosophers' problem, and
2. the reader-writer problem.

9.6.4 Summary of problems in synchronization

We have looked at three progressively more difficult classes of problems in synchronization, starting with some basic problems in serialization including signaling and the rendezvous; then considering multiplexing, group rendezvous and

multiplexing; and concluding with two advanced problems in synchronization including the dining philosophers' problem, and the reader-writer problem. These are not the only problems to be examined, and if you run into an issue of synchronization, you should consider referencing a book such as the *Little Book of Semaphores* to see if someone else has already proposed a reasonably optimal solution. Other solutions are also given names, including:

1. the no-starve mutex problem,
2. the cigarette smoker's problem,
3. the dining savages problem,
4. the barbershop problem,
5. Hilzer's barbershop problem,
6. the Santa Claus problem,
7. building H_2O,
8. the river crossing problem,
9. the roller coaster problem,
10. the search-insert-delete problem,
11. the unisex bathroom problem,
12. the baboon crossing problem,
13. the Modus Hall problem,
14. the sushi bar problem,
15. the child care problem,
16. the room party problem,
17. the Senate bus problem,
18. the Faneuil Hall problem,
19. the dining hall problem,

and so on. The purpose of listing these is not to make you think that you must understand all of these. Instead, if you run into an issue in synchronization, read the literature: someone may have already come across a similar problem and proposed a solution. This would require you to analyze the problem at hand and determine the core synchronization problem.

9.7 Automatic synchronization

Recall that with memory allocation, the optimal solution is to require explicit allocations and deallocations (manual memory management). This, however, significantly increases development costs, as it is much more difficult to ensure functional code; therefore, there are automatic alternatives to memory management such as garbage collection, but they come at an additional cost. We will now consider

1. the weaknesses of semaphores, and
2. automatic alternatives.

Specifically, we will describe the synchronization tools in Java and Ada, but we will also see a result that indicates that for purposes of synchronization, semaphores are the standard.

9.7.1 Weaknesses of semaphores

The strongest issue with semaphores is that, like memory allocation in C, they are manual. It is up to the programmer to ensure that every wait and post to a semaphore is correctly in place and failure to have even one statement in the right place could cause a system failure: either simultaneous access to a critical section or deadlock. We will consider automatic synchronization provided by Java. Languages such as Ada that target embedded and real-time applications have an even richer collection of automatic synchronization tools.

9.7.2 Automatic alternatives to semaphores for mutual exclusion

There are numerous solutions that are equivalent to the use of semaphores—that is, any synchronization achieved through semaphores may be achieved using *protected objects* in Ada, *monitors* in numerous other programming languages, and Java's concept of synchronized blocks and methods, and vice versa. That is, neither is more powerful than the other, only it may be easier to achieve synchronization using one paradigm over another. We will describe Java's implementation, as it provides a concrete version.

The Java `synchronized` keyword may be used either on a method or a block of code. For example, in a singly linked list class that is intended to be shared by multiple tasks, the `pushFront` method may be declared to be synchronized, indicating that if another thread attempts to call the same function, it will have to wait its turn.

```
class SingleList {
    private SingleNode listHead;
    private SingleNode listTail;
    private int listSize;

    public void synchronized pushFront( Object obj ) {
        SingleNode newNode = new SingleNode( obj, listHead );

        if ( newNode == null ) {
            return false;
        }

        listHead = newNode;

        if ( listSize == 0 ) {
             listTail = listHead;
        }

        ++listSize;

        return true;
    }

     // ...
}
```

Now, compare and contrast the two implementations:

```
bool single_list_push_front(
    single_list_t *const p_this, void *p_new_entry
) {
    bool success;

    single_node_t *p_new_node =
        (single_node_t *) malloc(
            sizeof( single_node_t )
        );

    if ( p_new_node == NULL ) {
        success = false;
    } else {
        p_new_node->p_entry = p_new_entry;

        binary_semaphore_wait( &( p_this->mutex ) ); {
            p_new_node->p_next = p_this->p_head;

            p_this->p_head = p_new_node;

            if ( p_this->size == 0 ) {
                p_this->p_tail = p_new_node;
            }

            ++( p_this->size );
        } binary_semaphore_post( &( p_this->mutex ) );

        success = true;
    }

    return success;
}
```

```
public bool synchronized pushFront(
    Object obj
) {
    bool success;

    SingleNode newNode =
        new SingleNode( obj, listHead );

    if ( newNode == null ) {
        success = false;
    } else {

        listHead = newNode;

        if ( listSize == 0 ) {
            listTail = listHead;
        }

        ++listSize;

        success = true;
    }

    return success;
}
```

Visibly, it appears that the period of mutual exclusion is only a few instructions less; however, consider that most of the instructions execute in a small number of cycles, with the major exception being the memory allocation. Memory allocation, however, usually requires significantly more time to execute by at least an order of magnitude (perhaps two orders of magnitude) than all other instructions—including the mutual exclusion.

Now, the `synchronized` keyword in Java allows for mutual exclusion, but not for serialization. There is, however, a feature equivalent to semaphores; only the wait is performed on the current instance of the class in place of the semaphore—in other words, the instance of the class becomes the resource, not the semaphore:

`wait()`
: Wait on the current instance of this class. In order to avoid deadlock, other tasks may now execute synchronized methods.

For example, consider the following program with a producer-consumer problem: There are two classes that implement the `Runnable` interface. One is a producer, and the other is a consumer. This means that these classes implement a function `void run()` that can be executed as a new thread. They will communicate through a synchronized coordination class that is passed to both as an argument in the constructor.

```
class Producer implements Runnable {
    Coordination coord;

    Producer( Coordination c ) {
        coord = c;
        new Thread( this, "Producer" ).start();
    }

    public void run() {
        for ( int i = 0; true; ++i ) {
            coord.produce( i );
        }
    }
}

class Consumer implements Runnable {
    Coordination coord;

    Consumer( Coordination c ) {
        coord = c;
        new Thread( this, "Consumer" ).start();
    }

    public void run() {
        while( true ) {
            coord.consume();
        }
    }
}
```

The coordination class has two methods that have access to an instance variable *n*. Now, the producer cannot produce another object unless the consumer has consumed the previously created object.

```
class Coordination {
    int n;
    boolean produced = false;

    public synchronized void produce( int value ) {
        if ( produced ) {
            try {
                wait();
            } catch ( InterruptedException e ) {
                e.printStackTrace();
            }
        }

        n = value;
        System.out.println( "Producing: " + n );
        produced = true;

        notify();
    }
```

```
    public synchronized int consume() {
        if ( !produced ) {
            try {
                wait();
            } catch ( InterruptedException e ) {
                e.printStackTrace();
            }
        }

        System.out.println( "Consuming: " + n );
        produced = false;
        notify();

        return n;
    }
}

class Executable {
    public static void main( String args[] ) {
        Coordination c = new Coordination();

        new Producer( c );
        new Consumer( c );

        System.out.println( "Ctrl-C to exit." );
    }
}
```

9.7.3 Rendezvous in Ada

To be completed.

9.7.4 Mutual exclusion in Ada

The Ada programming language offers a `protected` keyword that allows one to define a class similar to the `synchronized` keyword in Java. Here is an example of how this keyword can be used to enforce mutual exclusion.

```
-- Example from Jan Jonsson
protected type Mutual_exclusion is
  entry Wait;
  procedure Post;
private
  Acquired : Boolean := false;
end Mutual_exclusion;

protected body Wait is
  entry Wait when not Acquired is
  begin
    Acquired := true;
  end Wait;

  procedure Post is
  begin
    Acquired := false;
  end Post;
end Mutual_exclusion;

Mutex : Mutual_exclusion;

task Task1;

task body Task1 is
begin
  -- Initialization
  Infinite_loop:
    loop
      -- Preprocessing
      Mutex.Wait;
      -- Critical region
      Mutex.Post;
      -- Postprocessing
    end loop Infinite_loop;
end Task1;
```

The protected object implementation of Ada will automatically wake up if it is waiting on a critical region and the previous task executing that region has left it. For comparison of other synchronization tools in Java and Ada, see Benjamin M. Brosgol's paper *A Comparison of the Concurrency Features of Ada 95 and Java*, available at:

http://www.sigada.org/conf/sa98/papers/brosgol.pdf

9.7.5 The equivalence of synchronization tools

You may recall that previously, we saw that

1. any processor that can simulate a Turing machine can compute any possible algorithm, and
2. an algorithm using just blocks of instructions, condition statements, and condition-controlled loops (structured programming) is equivalent to the most general form of algorithm with arbitrary jumps (or *goto* statements), etc.

In the first case, it is not proven that any algorithm that can be written can be written to execute on a Turing machine; it is only a hypothesis (the Turing-Church hypothesis) that has not been disproven in in the past century. In the second

case, however, it is a theorem that structured programming is as powerful as more general programming without the additional restrictions required by structured programming. The next question we may ask is, are different approaches to synchronization equivalent?

This appears to be a much more difficult question to answer, as the concept of "synchronization" is much more difficult to describe than that of "computability". The consensus in the literature is that semaphores are the standard for synchronization, and any synchronization tool that allows one to produce something equivalent to a semaphore is capable of providing whatever synchronization serves that may be required. Java's approach is equivalent to semaphores, as are Ada's automatic synchronization tools, and it is believed that any other synchronization tools ever developed will fundamentally be no stronger than that provided by semaphores. If you wish to read an interesting paper, see *A pragmatic, historically oriented survey on the universality of synchronization primitives* by Jouni Leppäjärvi from the University of Oulu.

9.7.6 Summary of automatic synchronization

We have looked at two alternatives for automatic synchronization. In each case, there is less reliance placed on the programmer and the programming language and compiler deal with the implementation issues. As noted, however, they always provide weaker cases.

9.8 Summary of synchronization

We have described why synchronization is required, how we can represent synchronization graphically with Petri nets, and the achieving synchronization first through passing tokens, the test-and-reset command, and semaphores. After this, we considered problems in synchronization followed by a discussion of automatic synchronization found in other languages.

Problem set

9.1 If one thread is only ever reading a variable in main memory and another is only ever writing to it, can there be a problem with synchronization? If so, please elaborate on a situation where an issue could occur.

9.2 If one thread is only ever making a query about a data structure and another is only ever modifying the data structure, can there be a problem with synchronization? If so, please elaborate on a situation where an issue could occur.

9.3 Which instruction is vulnerable to significant issues with synchronization, and why?

```
a = b + c;
a = a + b;
```

9.4 In kindergarten, it is often the practice to have a *talking stick* (or some other object) where that stick is passed from one student to another around a circle. A student can only talk if he or she is holding the stick. Is this a valid implementation of a token ring? If not, why? If so, what is the shared resource?

9.5 How do token rings differ from simply passing the token to the next task (or in the case of the previous example, the next individual) requesting the token?

9.6 A simple test-and-reset function requires polling. How do our more complex implementations of binary semaphores (as described in class) reduce this unnecessary overhead?

9.7 Instead of using test-and-reset instruction, some processors have a test-and-increment instruction that is passed a variable, has a return value equal to the variable as it was passed in, but then increments it. Could you implement a binary semaphore using such an instruction? [difficult]

9.8 In a simple system where there is only one task responsible for accessing a sensor and another task communicates those values to on a communication channel, is there any significant need for anything else other than polling?

9.9 Why does the following not work?

```
/* Global variable */
bool mutex = false;

/* Inside task     */
if ( test_and_reset( &mutex ) ) {
    scheduler();
}

// Access the data structure

mutex = false;
```

9.10 In the example in the text where the binary semaphore for the singly linked list data structure, the critical zone only includes a subset of the statements. Can we expand the critical zone? If so, would this be good or bad? Can we shrink the critical region (removing either, for example, the first or last statements from the region and moving them outside the critical region)? Why or why not?

9.11 It was suggested that a binary min-heap (assuming low numbers have the lowest priority) is the fastest data structure for implementing priority queues for storing tasks waiting for a particular semaphore. Why must that binary min-heap use lexicographical ordering as opposed to simply using the priority, where each task has as its priority a pair (p, k) where p is the priority and k is monotonically increasing number.

9.12 Is it possible to have a semaphore shared between two different tasks if those two tasks do not have access to the same memory location (that is, they do not share memory)?

9.13 Implement a swap function that swaps the contents of two integer memory locations.

9.14 Implement a counting semaphore data structure using binary semaphores.

9.15 You are asked to implement a data structure with the following properties:

1. if a task asks to be put to sleep, it is blocked until another task issues a wake-up call, but
2. if a task issues a wake-up call and nothing is blocked, nothing happens, and this does not unblock future tasks that request to be put to sleep.

Why can you not use counting semaphores for this? Could you design a more complex data structure that uses either binary or counting semaphores (or both) to implement the desired functionality?

9.16 Describe a situation where a priority-based binary semaphore may result in a task being perpetually blocked on a wait issued on that semaphore.

9.17 What happens if a counting semaphore is used for mutual exclusion but it is initialized to 0? Will anything using that semaphore ever execute?

9.18 How does multiplexing differ from a light switch?

9.19 Implement a light switch that allows at most *n* objects into the room.

9.20 Suppose we have a single lane bridge and once people start passing over in one direction, others can continue crossing in that same direction until the last person travelling in that direction gets off the bridge. Then, if there is someone waiting to go in the opposite direction, they would then start travelling across the bridge.

9.21 The previous scenario works well as long as the gaps between arrivals (on either side) is larger than the time it takes to cross the bridge. What can happen if the time between arrivals is slightly less than the time it takes to cross the bridge?

9.22 Implement two tasks that use the data structures used in class to cross the bridge.

9.23 Modify the functionality of the previous question so that you can use a different data structure to require individuals to stop following those crossing the bridge in the same direction as soon as someone starts waiting on the opposite side.

9.24 Describe how you would go about producing a reader-writer solution where new readers are only allowed to begin reading a file if there are no higher priority tasks waiting to write to the file, and similarly, a new writer is only allowed to begin writing to a file if there are no higher priority tasks waiting to read the file. If tasks have the same priority, they are serviced on a FCFS basis. Thus, a reader-writer-reader sequence of arrivals (all at the same priority) would see the first reader granted access, then when finished, the writer would have access, and then when it finishes, the second reader would have access. Note: you may have to use English to describe the information you need at various steps.

9.25 How does **Question 9.24** relate to our previous problem on using a lexicographical order for binary min-heaps?

9.26 Describe why automatic synchronization is almost certainly better than semaphores in larger scale projects?

9.27 Describe in your own words what happened with the Pathfinder mission.

9.28 A Twix® factory requires that each outgoing Twix candy consists of two bars: one from the *left Twix factory* and the other from the *right Twix factory*. The two factories cannot access the same candy simultaneously and a factory that has added a bar to a candy cannot proceed to the next bar until the other factory has added its bar. Use semaphores to ensure these restrictions, and assume that each factory is represented by a task, and each task can issue a command `void add_bar( void )`.

9.29 In the light-switch data structure, the wait on the corresponding semaphore is located inside the mutual exclusion of mutex. Why do we use this instead of placing it outside the critical region for modifying the population variable? Hint: consider two individuals entering the room almost simultaneously.

```
void lightswitch_wait( lightswitch_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        ++p_this->population;
    } binary_semaphore_post( &( p_this->mutex ) );

    if ( p_this->population == 1 ) {
        binary_semaphore_wait( p_this->p_access_control );  // Why does this break?
    }
}
```

10 Resource management

To this point, we have discussed the idea of resource management, but we haven't gone into the details as to how to manage such resources, except perhaps in the most obvious approach: use semaphores. In this topic, we give a more in-depth look at resource management by considering

1. semaphores as an example of a resource,
2. the classification of resources,
3. device management,
4. resource management, and
5. the problem of priority and deadline inversion.

10.1 Semaphores

We have had a significant discussion on the management of semaphores; however, a semaphore is nothing more than a virtual resource: it is a token that can be acquired by a task or thread, and once that token is no longer required, it can be returned. A binary semaphore allows only a single token to be shared, while a counting semaphores allows the possibility that numerous tasks and threads can simultaneously acquire such a token. The characteristics of semaphores make them ideal for very straight-forward resource management: each resource is associated with a semaphore, and if a task or thread wants to use that resource, it must first acquire the corresponding semaphore. If another task or thread is using the resource, the task or thread attempting to acquire the semaphore will be blocked.

10.2 Classification of resources

While semaphores are an obvious resource management tool, it is useful to consider different types of resources:

1. reusable resources, and
2. consumable resources.

The first consists of resources that can be used by one task or thread and then be released and made available to others. A consumable resource is associated with a message or signal. For example, any interrupt is a consumable resource, including keystrokes on a keyboard, the movement of a mouse, or a message being received.

Some resources cannot be shared between tasks and threads while others can: we classify these as

1. exclusive resources, and
2. shared resources.

An exclusive resource is one that can be used by only one task at a time. A printer is the most obvious exclusive resource. The call stack of a task or thread (including any parameters and local variables) is almost certainly an exclusive resource, while global variables are meant to be shared. In some cases, the resource may be exclusive for one operation while potentially shared for others. The most obvious we have seen of this is a file: only one task or thread can modify a file, but multiple tasks or threads can read a file simultaneously. We used a semaphore for an exclusive resource, while a resource that can be shared can use a light switch data structure.

Reusable resources may either have to be dedicated to a single task or thread until it is finished with the resource, or it may be possible to temporarily *borrow* the resource while putting the task to sleep. We call these two types of resources

1. non-pre-emptible and
2. pre-emptible.

A pre-emptible resource is one that can temporarily be taken away from a task or thread and then returned back once it is used by another task. The most obvious pre-emptible resource is the processor itself. A task or thread can be interrupted and other tasks, threads or interrupt service routines can execute. Then, the state of the processor can be restored to the state it was in prior to the preemption.

In a sense, memory is also pre-emptible: it is possible to temporarily store the contents of a block of memory elsewhere (likely in secondary memory, either a solid-state drive (SSD) or a hard disk drive (HDD)), use that memory for another purpose, and then restore it to its original state. This, however, is not practical for embedded systems, as the time required to copy to secondary memory is often prohibitive.

Essentially, any pre-emptible resource must have a state that can be temporarily stored and then restored once the use of the resource is finished. The task or thread that originally had the resource will almost certainly have to be blocked.

From this discussion, the management of certain resources is sufficiently distinct from general resource management that they are given special consideration, including:

1. task or thread management,
2. memory management,
3. device management, and
4. file management.

We have already considered the intricacies of creating and scheduling tasks and threads, and we have already considered the difficulties of memory management when we considered numerous dynamic memory allocation schemes. Later in the course, we will discuss file management (Topic 17), but now we will focus on device management.

10.3 Device management

A device is any peripheral that is attached to the computer. We have already determined that the two means of communicating with a device are polling and interrupt. The first allows for the most rapid response assuming the task has a sufficiently high priority, but interrupts are the most convenient means of mediating such communication. We will assume interrupts for the balance of this section.

In a small-scale embedded system, most devices will be communicated with directly; however, as the complexity of a project increases, this will require a uniform interface to the various devices.

Each device is connected to a bus communicating with the processor through a *device controller*. This controller will have a number of registers that can be accessed through instructions on the processor.

If a device controller has just a single register for data, it will signal an interrupt and at some point a task will access that register and copy the value. The device, however, may produce new data, in which case it has a choice:

1. overwrite the existing register, or
2. discard the new value.

An alternative is to create a hardware buffer: the device controller has a number of registers forming a circular buffer. Each time the register is read, it sends the value stored in the longest occupied register. To allow simultaneous access to the buffer by both the device and the processor, the controller may implement double buffering.

A different approach to device communication is memory-mapped input/output. You will see this in the labs, but essentially, this requires hardware support, where the devices have direct memory access (DMA) and when a register is changed, it changes a corresponding location in main memory. Similarly, writing to that location in memory will write the result to the device. For example, to initialize memory protection on the LPC1768, numerous parameters must be written to specific locations in memory, and then writing to a last location signals that the memory protection unit can now use the balance of values to create a region of protected memory.

Now, you have two devices sharing the same memory locations for information. In this case, you no longer have the ability to use semaphores to mediate access between the two parties: one is a device, and the other is a task or thread executing on the processor. Thus, the usual solution is for separate memory locations dedicated to communicating information from the device, and another memory location for communicating information to the device. An additional memory location can indicate whether the information is in transition ("busy") or ready.

10.4 Resource managers

Finally, another approach is to require all access to resources through a uniform interface. Such an application programming interface (API) would prevent accidental access to the same resource simultaneously. In an embedded system without an operating system, there is always the issue that the resource will still be accidently used when it should not; however, the protections that are necessary require hardware support. Later, we will see that an operating system is essentially a resource manager.

One issue a resource manager will have to deal with is priority inversion: a lower priority task or thread may hold an exclusive resource that is required by a higher priority task or thread. Consequently, it is necessary to increase the priority of the lower priority task or thread to that of the higher priority one until the resource is released.

Additionally, a resource manager as opposed to simply being an interface that is executed when a task or thread requires or releases a resource, it could also itself be a thread that occasionally executes and checks to ensure that allocated resources are still associated with live tasks.

One benefit of a resource manager is that when a task or thread is terminated, the resource manager can immediately reclaim all resources associated with that task, making them once again available to other tasks and threads.

10.5 Priority and deadline inversion

An excellent reference for this section is Ragunathan Rajkumar's *Synchronization in Real-time Systems: A Priority Inheritance Approach*, published by Kluwer Academic Publisher in 1991.

Quick review: Recall that we have already seen that priorities are a very efficient mechanism for scheduling tasks in such a way that we can designate the order in which tasks should be executing at any time, and if it becomes impossible for all tasks to meet their deadlines, which tasks will fail. Recall also that we use 0 to represent the highest priority, while successively larger natural numbers represent successively lower priorities.

Consider the following scenario: three tasks, t_0, t_1 and t_2 are executing, where the subscript denotes the priority. Tasks t_0 and t_1 are, for some reason, blocked so the lowest priority thread t_2 is executing. It acquires a binary semaphore and begins entering the critical section. While t_2 is executing, an interrupt occurs and process t_0 becomes ready, and it too waits on the same binary semaphore; however, the higher priority process is now blocked, thus the processor is returned to the lower priority task. Then, prior to posting the binary semaphore, another interrupt occurs and now the intermediate task t_1 is ready to execute. As it has the highest priority of all ready processes, it will continue executing for an arbitrary length of time.

Under some circumstances, the lower-priority process may be starved for processer time, so it may never finish executing the critical section and releasing its binary semaphore. Under others, the intermediate task (or tasks) may be periodic, thus compounding the effect. Thus, we have a situation where a higher priority process is blocked on a lower priority process that will not be able to execute for an indeterminate amount of time—a condition not acceptable to any real-time system.

Such a situation is described as a priority inversion. There are multiple solutions:

1. block interrupts,
2. priority inheritance, and
3. priority ceilings.

We will look at each of these, but first a case study.

Similar to priority inversion, we can also have deadline inversion (assuming we are using earliest-deadline first (EDF) scheduling. A task with a later deadline may hold a semaphore or resource required by a task with an earlier deadline. As you read about solving the problem of priority inversion, you should be able to consider parallel solutions to the problem of deadline inversion.

10.5.1 Mars Pathfinder

In 1997, the Mars Pathfinder mission landed and the Sojourner rover, a precursor to Spirit and Opportunity, went off discovering this new world. Once on Mars, Pathfinder collected information and communicated it back to the Earth.

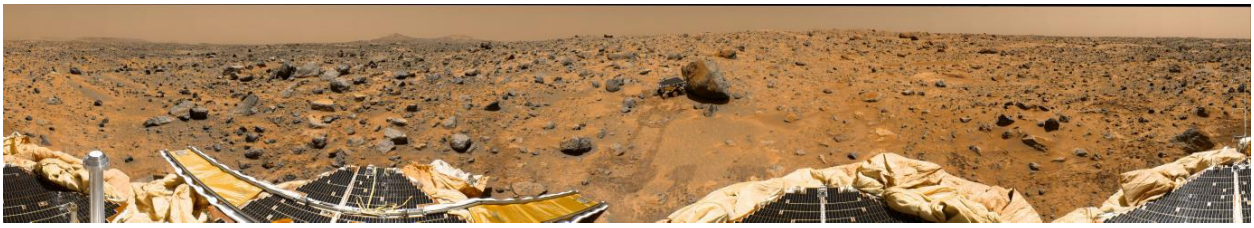


Figure 10-1. Panorama from the Carl Sagan Memorial Station with the Sojourner rover visible in the center (NASA).

A few days into the mission, the meteorological station was activated, and at some point after this, Pathfinder experienced a total system reset. As the station had no secondary storage, all collected data was lost. This happened again and again. Fortunately, the real-time operating system (VxWorks—the same as on Spirit and Opportunity, and also Curiosity), had logging capabilities and they were able to determine the following scenario involving access to an information bus and three tasks:

1. A high-priority bus-management task ran frequently in order to move data along the information bus as necessary for the operation of Pathfinder. It would acquire a semaphore for access to the information bus.
2. A low-priority meteorological-data-gathering task ran infrequently and required the information bus to transfer that data. It, too, would acquire a semaphore for access to that bus.
3. A medium-priority communication task ran very infrequently, but it had a long computation time.

On occasion, the low-priority meteorological-data-gathering task would acquire the information bus semaphore. While accessing the bus, it would be pre-empted by the communication tasks, which would execute for a long period of time. During this time, there was a higher probability that the bus-management task would be scheduled to run. The bus-management task would attempt to acquire the same semaphore and would be blocked. Rather than returning the processor to the meteorological-data-gathering task, the communication task would continue executing, and after a period of time, the bus-management task would miss its deadline, a watchdog timer was set to detect when the bus-management task fails to run, and the automatic response was that there was a serious problem that required a complete system reset. This is shown in Figure 10-2.

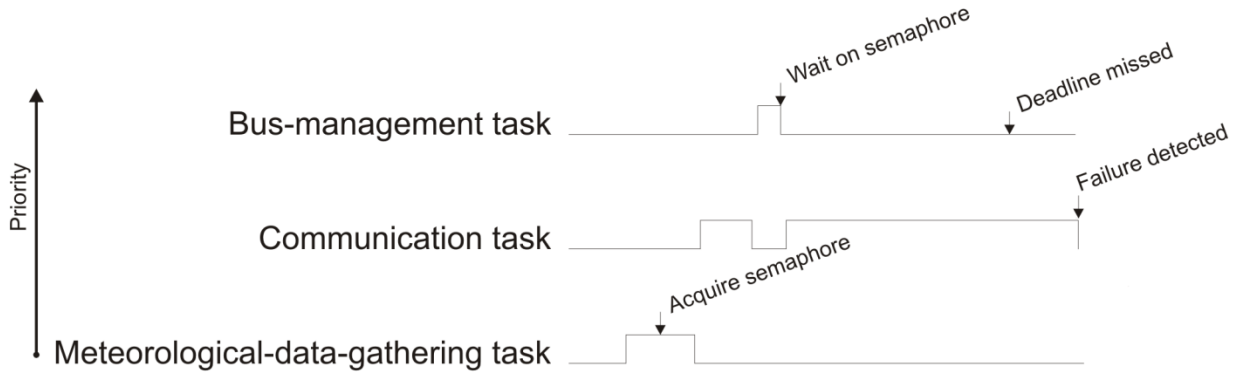


Figure 10-2. Priority inversion on the Mars Pathfinder mission.

We will continue to look at various solutions to this problem.[28]

10.5.2 Blocking interrupts

The easiest solution is to block interrupts for as long as the task is in the critical section. This may be appropriate if the size of the critical section is very short; say, a brief $\Theta(1)$ operation on a data structure. However, anything longer may block a higher priority task from executing, even if that task has no requirement to access the data structure being operated upon.

One might be able to solve such problems by attempting to block only lower priority interrupts; however, this does not solve the above scenario. Instead, any time a binary semaphore is waited upon, the interrupts must be blocked to the highest priority of any tasks waiting on that interrupt. In a static priority scenario, this may be possible, but it may be more difficult if we have dynamic priorities.

10.5.3 Priority inheritance

Another solution is to dynamically increase the priority of any task executing a critical section whenever a higher priority task waits on that same interrupt. Once the promoted task releases the semaphore, it is returned to its original priority. If multiple tasks were waiting for a lower-priority task to release a semaphore, then the executing task would receive the highest priority of all waiting tasks.

When the semaphore is released, it would be given to the highest priority waiting task, so the only requirements are:

1. When a binary semaphore is waited upon, if the task holding the semaphore has a lower priority than the task waiting on it, the priority of the holding task is dynamically set equal to the priority of the waiting task.
2. When a binary semaphore is released, if the priority of the task releasing the semaphore was dynamically increased as a result of other tasks waiting on that semaphore, the priority is returned to its original priority and control of the processor is given to the highest priority task waiting on the semaphore in a FCFS basis.

A beneficial feature is that any task waiting on a semaphore will only ever be required to wait on one lower-priority task completing its critical section before the lower-priority task releases the semaphore and the scheduler is called. Thus, the maximum delay for a high-priority task is the execution time of the critical section.

Note: The argument we have been discussing here with a binary semaphore. Suppose instead, we have a counting semaphore with an initial value of n and there are n tasks that have acquired that semaphore. Now, a higher priority task comes along and waits on that semaphore and is blocked. What do we do to all of the priorities of the waiting tasks that can be done in $o(n)$ time?

Back to Pathfinder: this was the solution used for the Mars Pathfinder mission. The semaphore actually had an option to do this, but the value of the parameter controlling this option was set to `false`. The value of this parameter was stored as a global variable, so the programmers were able to switch this value to `true` for this semaphore and thus solve two other potential problems. The feature had been turned off in the first place for an unspecified reason.

During the testing phase, the engineers did detect two deadlocks; however, they could never reproduce it, so they essentially ignored the problem. With Pathfinder, the mission critical event was the landing, and this held the lion's share of the attention for the engineers—for everything else, a reset was deemed to be an acceptable solution.

L. Sha, R. Rajkumar, and J. P. Lehoczky. *Priority Inheritance Protocols: An Approach to Real-Time Synchronization.* In IEEE Transactions on Computers, vol. 39, pp. 1175-1185, Sep. 1990.

[28] See From: Mike Jones, *What really happened on Mars?*
http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html

10.5.4 Priority ceiling

A less appealing solution, though perhaps easier to implement, is priority ceiling. The semaphore is assigned a priority and any task that waits on that task is assigned that priority. Necessarily, the priority assigned to the semaphore is the highest priority of any task that may acquire that semaphore and no lower or higher.

10.5.5 Priority demotion

Under either of these schemes, a task may lock multiple semaphores; consequently, as it releases those semaphores—and not necessarily in the same order they were locked—there are two options:

1. the task maintains the highest of the priorities until all semaphores have been released, or
2. the task is always running at the highest priority resulting from the semaphores that it does hold.

The first solution is $\Theta(1)$, while the second is at least logarithmic ($\Omega(\ln(n))$) in the number of semaphores locked. However, in the first case, the task could temporarily block a higher priority task.

10.5.6 Priority inheritance in mutual exclusion locks

Most implementations of mutual exclusion locks (mutexes) also include priority inheritance. This is true with the Keil RTX RTOS and **QNX** (*cue-niks*).

10.5.7 Summary of priority and deadline inversion

The problem of priority inversion (and its related problem of deadline inversion) has as its simplest solution promoting the priority of the task to that of the task requesting the same resource. This makes sense in that although the priority of that task or thread is low, if the semaphore or resource being held is also required by a high priority task or thread, the lower priority task or thread should finish its job with the same urgency of the higher priority task or thread.

10.6 Summary of resource management

In summary, there are a number of peripherals that tasks and threads will require access to in order to accomplish their goals; however, sharing these among tasks that are competing for those resources is an issue that must be addressed. Essentially, all the lessons we have learned from previous topics can be automatically applied to the issue of general resource management.

Problem set

10.1 Why can the management of resources be reduced to a question of the management of semaphores?

10.2 What are the benefits of using semaphores for resource management for smaller projects, and what are the costs of using semaphores for resource management for larger projects?

10.3 What are the costs of using a resource manager for smaller projects, and what are the benefits of using a resource manager for larger projects?

11 Deadlock

Most resources can only be used by one task at a time. Exceptions exist, such as files and databases that can be read simultaneously by any number of tasks, but is more of an exception. If tasks require multiple resources, it may occur that at some point, no task is able to continue even though sufficiently many resources exist for at least one task to continue execution.

As examples, suppose we have toy drum and one child grabs the drum while another grabs the sticks. At this point, each needs what the other child has, but neither will give up the component they have.[29] Suppose we have a quill pen and an ink well in a disorganized office. If one person acquires the quill pen and begins searching for the ink well, while another acquires the ink well and begins searching for the pen, neither can finish writing their letter. More practical, suppose data from two sensors connected to a bus must be copied to a single file. If one task locks the file for writing while another acquires the bus for copying the information from the sensor, neither can acquire the other component necessary to complete the transaction. While the two searching for the quill pen or the ink well may finally ask the other what they are looking for, tasks generally don't do that.

In previous topics, we have seen how deadlock can occur, and we've considered some solutions. In this topic, we will examine deadlock in more detail; specifically we will

1. describe the requirements for deadlock to occur,
2. introduce a model for describing deadlock,
3. discuss techniques for preventing deadlock, and
4. describe deadlock detection and recovery.

Dealing with deadlines is a question of the non-functional requirement of safety: if a critical task cannot execute at a point in time because it does not have the required resources, this can in some cases be dealt with by simply killing all other tasks that are currently using the required resources. If, however, a number of tasks are mutually deadlocked on execution, it is likely that the system will quickly move to an unsafe state: critical operations may fail to be performed, or critical communications with other devices will fail to occur. Checking for deadlock can, however, be an expensive operation, and we will discuss efficient strategies to ensure reasonable performance.

11.1 Requirements for deadlock

Any time a task attempts to lock two or more resources, deadlock can occur. There are, however, a number of requirements in addition to this that must be satisfied (Coffman et al., 1971):

1. mutual exclusion: the resource is either available (not currently assigned to a task) or it is assigned to exactly one task,
2. hold-and-wait condition: a process that is currently holding one or more resources may request but is forced to wait for an additional resource,
3. no-preemption condition: a resource that has been granted to a task can be released only by that task—it is not possible to reassign the resource while it is held, and
4. circular-wait condition: it must be possible for a cycle of tasks and resources to exist.

Let's consider the initial solution to the Dining Philosopher's problem: when philosophers sat down, they attempt to pick up the chopstick to the left, and after acquiring it, they attempt to pick up the chopstick to the right. Are the four requirements above satisfied?

1. each chopstick may be used by only one philosopher at a time,
2. each philosopher holds onto the first chopstick while attempting to grab the second,

[29] From the post "Deadlock: the Problem and a Solution" by Anthony Williams, September 18th, 2008.

3. it is not possible to take a chopstick away from a philosopher who has one, and
4. if each philosopher has acquired the chopstick to their left, we have circular waiting.

All conditions are satisfied: we have a deadlock. We will now discuss how we can avoid such deadlocks.

11.2 Deadlock modeling

Deadlock concerns the acquisition and locking of resources by tasks. To model such a system[30], we will describe tasks by using a circle:

We will discuss two types of resources:

1. reusable resources: resources that can be held, released, and subsequently reused, and
2. consumable resources: resources that can be granted but are then no longer available for other tasks to use—consumable resources must be replenished through the actions of another task.

A reusable resource will be represented by a cyan square, and the dots indicate the number of resources that are available.

A consumable resource will be represented by a truncated magenta square, and the dots indicate the number of resources that are currently available.

The justification for requiring consumable resources be replenishable is that any consumable resource that is not replenishable must be correctly allocated at design time. A system which has only two instances of a fixed consumable resource but requires three to avoid deadlock is simply badly designed.

A task can request a resource, and a resource can be held exclusively by a task. The first is shown by a dotted line indicating the request, and the latter is shown by that request being assigned to the task.

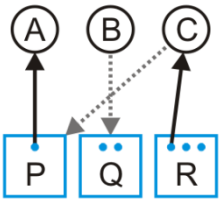


Figure 11-1. Three tasks with specified ownerships and requests.

In Figure 11-1, Task A has acquired a lock on Resource P, Task B is requesting Resource Q, Task C has a lock on one of the three available instances of Resource R and has requested Resource P. In this case, we can see that the request of Task B can be satisfied, but the request of Task C cannot be satisfied until Resource P is released by Task A. Tasks that are waiting on resources are considered to be blocked.

[30] This model is described in Gary Nutt's Operating Systems textbook, §10.2.

Now, you may be asking yourself: how can we get a situation where Task B has requested an instance of Resource Q, but that resource has not yet been allocated. Can this actually happen in real life?

Consider the following scenario: Tasks A has a higher priority than Task B and Task A held two instances of Resource Q. While Task A is blocked, Task B executes and makes a request for Resource Q. In the meantime, Task A is ready again, it releases two instances of Resource Q but continues executing. Task B is therefore in a state where it is requesting an instance of Resource Q, but that request has not yet been satisfied.

If we now look at Figure 11-2, we note that deadlock has occurred:

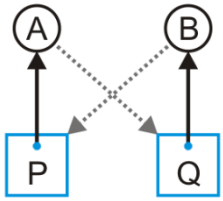


Figure 11-2. Example of deadlock.

Here, Task B is requesting a resource held by Task A, and vice versa.

With consumable resources, we have a sample case in Figure 11-3.

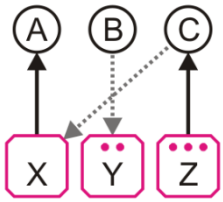


Figure 11-3. An example with processes and consumable resources.

In this example, Task A produces instances of Resource X, Task B is requesting an instance of Resource Y, and while Task C is producing instances of Resource Z (of which there are three), it currently is requesting an instance of Resource X which cannot at this time be satisfied. An example of a deadlock with a consumable resource is shown below in Figure 11-4.

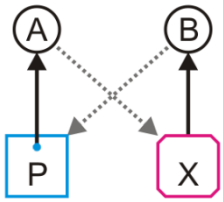


Figure 11-4. Deadlock with a reusable and consumable resource.

Here, Task A holds Resource P but cannot proceed because it is requesting an instance of Resource X. Resource X could be replenished by Task B, but Task B is currently blocked on a request for Resource P. If, however, there was already an instance of Resource X available, as is shown in Figure 11-5, the request of Task A could be satisfied.

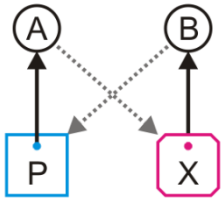


Figure 11-5. A live state with a consumable and a reusable resource.

In this case, we now have the situation in Figure 11-6 which is no longer deadlocked: at some point, Task A can give up Resource P and it can then be allocated to Task B.

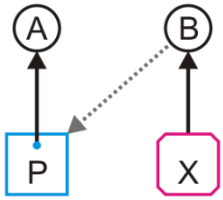


Figure 11-6. The state of Figure 11-5 after the consumable resource has been allocated.

If a task is not deadlocked and it holds a replenishable resource, we will assume that any number of requests for that resource will ultimately be satisfied and that, therefore, other tasks cannot be deadlocked on that resource. After all, any design that includes a task responsible for replenishing a resource that it does not do so is, again, bad design.

Note that three features are used to distinguish tasks from reusable resources from consumable resources: shape, color, and letter designation. We will continue using this throughout this topic.

Deadlock will occur whenever there is a cycle after all consumable resources have been allocated.

11.3 Techniques for preventing deadlock during the design

In order to prevent deadlock, it is simply necessary to keep one of the four conditions from being satisfied. We will investigate each of the conditions.

11.3.1 Mutual-exclusion condition

In general, resources must be mutually exclusive. Exceptions do occur, such as any resource that is read-only: such a resource can be accessed by multiple readers as we have previously discussed. For resources that cannot be shared, one option is to provide a *spooling daemon* for a resource. For example, a task allocates sufficient memory for the document to be printed. The information necessary to print the document is then transferred to the printer daemon, which prints documents as necessary and when possible. The task requesting the print job can then continue executing (it does not need to wait for the printer daemon to complete its work). As for the memory, either:

1. the memory is considered transferred to the daemon, which then frees that memory, or
2. the daemon must signal the requesting task when the print job is completed (possibly using semaphores). Then the requesting task will free the memory.

As the printer daemon cannot be deadlocked, no other tasks requesting print jobs will be deadlocked, either.

11.3.2 Hold-and-wait condition and two-phase locking

Preventing hold and wait requires that either that

1. each task being able to only ever hold one resource at a time,
2. a task is not executed unless all the resources it requires are available and can be assigned to it, or

3. unique resources (of which there is only one or perhaps a few) must be allocated in groups.

The resource that could, under such constraints, impose the most significant problem is memory. This is avoided through Rule 5 of the JPL coding standard, which specifies that

> there shall be no use of dynamic memory allocation after task initialization.

While at the same time preventing memory allocation from being a trigger for deadlock, the standard also comments that

> "A notable class of coding errors stems from mishandling memory allocation and free routines: forgetting to free memory or continuing to use memory after it was freed, attempting to allocate more memory than physically available, overstepping boundaries on allocated memory, using stray pointers into dynamically allocated memory, etc. Forcing all applications to live within a fixed, pre-allocated, area of memory can eliminate many of these problems and make it simpler to verify safe memory use."

One solution to break the hold-and-wait condition is *two-phase locking*: Two-phase locking is when a task attempts to lock all necessary resources at once, and if it is unable to do so, all the resources are released and, at some point in the future, the task tries again. Once the resources are no longer required, they are all released, and no task is ever allowed to acquire additional resources until all previously necessary resources have been released. These two phases are referred to as

1. the *expansion phase* when a task is acquiring resources, and
2. the *shrinking phase* when resources are being released and no additional resources are being acquired.

These phases can either involve the simultaneous acquisition or release of resources, or an incremental acquisition or release of resources. Semaphores cannot be used for acquiring the resources, as this would not solve the hold-and-wait condition. Instead, some form of mutual exclusion would be necessary to ensure only one task at a time has access to the process of either acquiring or releasing the resources.

If the acquisition of resources must be performed simultaneously, a task will attempt to acquire the necessary resources, and if it fails, it will not acquire any and it will wait to try again, signaling in some manner that it is waiting for resources. If the acquisition is allowed to be incremental, if at any time the acquisition of one of the resources fails, it is subsequently necessary to release all held resources and then try to acquire all the resources again. The incremental option, while more flexible, is likely detrimental to any real-time system, and thus we will focus on the implementation of the expansion phase when all resources must be acquired simultaneously.

The issue with the simultaneous acquisition of resources is what is to be done if some of the resources are currently allocated to other tasks? The ideal tool for this is the signalling mechanism described in Section 9.5.4.3 on page 283. For example, suppose there are eight resources, labelled 0 through 7. If a task was waiting on Resources 2, 4 and 6, then it would be placed on list of tasks waiting for resources and then wait to be signaled on `00101010`. Now, any time resources are released, any task waiting for resources would be signaled. For example, suppose that a task released Resources 2, 3 and 4. It would then set `00111000` to all tasks waiting for resources, in which case, the waiting task would now only be required to wait on Resource 6. When that resource is finally set when it is released by another task, the waiting task would be scheduled and could try again to acquire Resources 2, 4 and 6.

Allowing such a mechanism to be used in general may be problematic—there is no guarantee that the resources will be available once the task is woken up, and so it may be required to wait, again, on a subset of the required resources. Thus, two phase locking cannot be used as a general solution to deadlock in an arbitrary system, but in a design where the tasks have specified

1. resources they will use to accomplish indicated goals,
2. known priority levels, and
3. known time intervals on which resources will be used,

it will be possible to verify whether or not the design satisfies the real-time constraints of the system. Two-phase locking is the most common real-time solution to break the possibility of deadlock, but it is also used in other systems such as databases.

One aspect the reader may wish to consider is whether or not it is reasonable to consider strategies similar to priority inheritance to solve some of the issues associated with the use of two-phase locking. While priority inheritance may be appropriate for semaphores, where there is only a single resource, if a task is holding multiple resources which are then requested by various higher priority tasks, tracking the appropriate priority of any one task holding resources becomes more problematic. As indicated above, unfortunately, while two-phase locking can avoid deadlock, it is also subject to *starvation* and *livelock*, which we will look at now.

11.3.2.1 Starvation

Suppose our dining philosophers acquire both chopsticks simultaneously, and once they are finished, they put both chopsticks down until they are ready to eat again. This is an example of two-phase locking, and while deadlock will not occur, but starvation was another issue for this solution: it may occur that one philosopher is never able to eat; that is, they may never have an opportunity to grab both chopsticks. For example, Socrates may be sitting next to Plato and Aristotle. Socrates attempts to acquire the chopsticks, but Plato is eating, and before Plato finishes, Aristotle decides that he will eat, and he manages to acquire both chopsticks, so that when Plato does finish, Socrates still cannot acquire both chopsticks. If Plato and Aristotle alternate as such, Socrates will never be able to acquire the chopsticks and thus will starve to death.

11.3.2.2 Two-phase locking and livelock

If the requests for resources during the expansion phase are allowed to be incremental, if it ever occurs that a request is denied (as the resource is unavailable), then all held resources must be released and the task must attempt again to acquire all resources. This scenario is likely to be more problematic, as it could lead to a situation referred to as *livelock*: Each philosopher attempts to and succeeds in acquiring the left chopstick. Each philosopher then notes that the right chopstick is held by another philosopher, so each philosopher puts down the left chopstick and tries again. No philosopher is ever blocked, but rather is in a continuous cycle of acquiring and then releasing one of the two resources they require. If all resources are acquired using mutual exclusion—that is, only one philosopher may, at any one time, attempt to acquire chopsticks—at least one philosopher will always be able to be eating.

While this solution may prevent deadlock, if caution is not taken, it may It may, however, be a reasonable solution for a low-priority task that has an opportunity to execute, but does not want to get caught in a circular-wait situation described in Section 11.3.4.

11.3.2.3 Summary of the hold-and-wait condition

The practice of requesting additional resources while already holding other resources is one of four conditions required for deadlock to occur. The most common solution to breaking this condition is to adopt two-phase locking, where resources are acquired during the expansion phase and released during the shrinking phase. In a real-time system, such a strategy can only be used as a specific strategy for known tasks requiring known resources during the design phase; two-phase locking cannot be used as a general strategy as it is subject to starvation and livelock.

11.3.3 No-preemption condition

In general, resources cannot be pre-empted. A resource that is being held but not being used could be pre-empted under the following conditions:

1. the state of the resource can be saved,
2. the task currently assigned to the resource can be blocked (low priority), and
3. after the resource is used, it can be returned to its original state, returned to the original task which is then set back to ready.

A drone on a reconnaissance mission is a pre-emptible resource, as having started a mission, it can always be recalled and reallocated to a higher priority task. A robotic arm that is currently in a safe state (not responding to an event or exerting a force) or could be placed into a safe state is also a candidate for being pre-empted. We will see later that this may be possible with *virtual memory*, but virtual memory causes its own issues with respect to real-time systems, as we will discuss in a later topic.

11.3.4 Circular-wait condition

The circular-wait condition requires that it is possible for a cycle where each task holds a resource required by the next task in the cycle. We have seen that this can occur with two tasks each attempting to acquire two semaphores. Figure 11-7 shows a cycle of three tasks, and the Dining Philosopher's problem had a cycle of five.

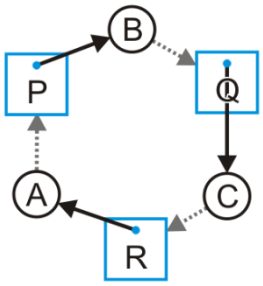


Figure 11-7. Example of a circular wait.

Both of the good solutions for the Dining philosopher's problem involved breaking this cycle.

1. The first solution prevented more than four philosophers from having access to the resources, consequently, preventing the cycle from being completed. This is, however, unlikely to be a situation that can be implemented in most real-time systems.
2. The second solution ordered the resources, requiring the philosophers to acquire the resources in order. In this case, Philosopher 4 could not acquire Chopstick 4 first and then acquire Chopstick 0. Instead, only one of Philosophers 0 and 4 will acquire Chopstick 0, and the other will be blocked, leaving Chopstick 4 available.

Ordering resources may not be completely achievable (for example, it would be difficult to enforce that all other resources are required prior to any request for additional memory), but it could be imposed on a subset of the more significant resources and thereby reducing the likelihood of deadlock occurring.

Theorem

Requiring that tasks acquire resources in a specific order prevents the circular wait condition.

Proof

Suppose that tasks must acquire resources in the order R_1, R_2, …, R_n. Suppose now that the circular wait condition has occurred between m tasks. In this case, let the order of the tasks in this circular wait be j_0, j_1, …, j_m where $j_0 = j_m$ and the resources requested by the tasks are k_0, k_1, …, k_m where $k_0 = k_m$. We may construct such lists by our assumption that there is a circular waiting condition. Now, for example, T_{j_1} is holding R_{k_0} and is requesting R_{k_1}, and so on for each of the tasks. Suppose that each of the tasks acquired their resources in order; that is, first $k_0 < k_1$. But this must be true for each task, and therefore it must also be true that $k_0 < k_1$, $k_1 < k_2$, …, $k_{m-1} < k_m$. By transitivity, it follows that therefore that $k_0 < k_m$. However, by assumption, these tasks were involved in a cycle of length m, and therefore $k_0 = k_m$. This is a contradiction. Therefore, at least one task must have acquired the resources out of order. ■

Let us consider a situation where this may help us. Recall the previous example where, in a large database, it is inefficient to lock the entire database when making a change. Instead, only those records that are being modified need be locked (recall that this is something of a simplification, but *c'est la vie*). In this case, if there was to be a transfer of information (funds, etc.) between two records, one may have the following:

```
void transfer( record *p_a, record *p_b, … ) {
    sem_wait( p_a->p_mutex );
    sem_wait( p_b->p_mutex );

    // Make the transfer

    sem_post( p_a->p_mutex );
    sem_post( p_b->p_mutex );
}
```

Suppose now the two commands

```
transfer( datafile[3952], datafile[471], … );
```

and

```
transfer( datafile[471], datafile[3952], … );
```

are executed almost simultaneously. In this case, it is possible that an interrupt occurs between the first two locks on the semaphore, during which time, the other transfer is initiated, thus locking the other semaphore. In this case, the correct solution is to use some sort of linear ordered key of the record and to always lock the two files in order—unlocking doesn't matter.

```
void transfer( record *p_a, record *p_b, … ) {
    assert( p_a->id != p_b->id );

    if ( p_a->id < p_b->id ) {
        sem_wait( p_a->p_mutex );
        sem_wait( p_b->p_mutex );
    } else {
        sem_wait( p_b->p_mutex );
        sem_wait( p_a->p_mutex );
    }

    // Make the transfer

    sem_post( p_a->p_mutex );
    sem_post( p_b->p_mutex );
}
```

Thus, all calls to this function will always lock the two in the same order. This can, of course, be generalized: for example, this transfer may involve other access to other data structures. In this case, it would be prudent to order rules in general:

1. Acquire locks to database records first, in order;
2. Then acquire locks to additional data structures necessary to make the appropriate transactions.

Otherwise, you may end up with the following:

```
void transfer( record *p_a, record *p_b, data_structure *p_data ) {
    assert( p_a->id != p_b->id );

    if ( p_a->id < p_b->id ) {
        sem_wait( p_a->p_mutex );
        sem_wait( p_b->p_mutex );
    } else {
        sem_wait( p_b->p_mutex );
        sem_wait( p_a->p_mutex );
    }

    sem_wait( p_data->p_mutex );

    // Make the transfer

    sem_post( p_data->p_mutex );

    sem_post( p_a->p_mutex );
    sem_post( p_b->p_mutex );
}

void update( record *p_a, data_structure *p_data ) {
    sem_wait( p_data->p_mutex );
    sem_wait( p_a->p_mutex );

    // Make the update

    sem_post( p_a->p_mutex );
    sem_post( p_data->p_mutex );
}
```

Here we have two different functions, likely written by two different authors, and yet, the following two calls could cause a deadlock:

```
transfer( datafile[3952], datafile[471], resource_info );
```

and

```
update( datafile[471], resource_info );
```

Such a bug would be subtle and difficult to catch, indeed. Consequently, good practice at design time will definitely help prevent such failures.

If you are ever requesting two resources, and there is an opportunity to request one before the other, this indicates that there is an opportunity to avoid deadlock by prescribing the order for all other requests that follow. This may involve investigating other situations where both resources are required, as the order in which the resources are acquired in a different situation may be more restricted. To illustrate, one task may require Resource P for a significant period of time but only temporarily requires Resource Q. Consequently, if another task requires both resources, it should adopt the order P and then Q, as well.

For example, memory is likely the most common resource required. Consequently, all memory should be allocated before requests for subsequent resources are made. To illustrate, if data is to be copied over a communication system, acquire read access for the data stored in memory prior to locking the communication system.

This is echoed in Rule 9 of the JPL coding standard:

> "The use of semaphores or locks to access shared data should be avoided (cf. Rules 6 and 8). If used, nested use of semaphores or locks should be avoided. **If such use is unavoidable, calls shall always occur in a single predetermined, and documented, order.** Unlock operations shall always appear within the body of the same function that performs the matching lock operation."

11.3.5 Other recommendations for deadlock prevention

In addition to avoid the four conditions necessary for deadlock, Laplante and Ovaska recommend:

1. minimizing the number of critical regions and their length,
2. all tasks must release their semaphores and resources as soon as possible,
3. do not suspend tasks when they are executing a critical region,
4. all critical regions must be error free,
5. do not allocate resources in interrupt handlers, and
6. always perform validity checks on pointers used in critical regions.

11.3.6 Summary of techniques for preventing deadlock

Avoiding deadlock requires that one of the four conditions for deadlock be prevented from occurring, a result shown by Havender in 1968. Not all of these measures can always be implemented in a sufficiently complex real-time systems, but they can be considered and given at least partial implementations, as suggested, in which case the likelihood of deadlock can be significantly reduced. We must accept that deadlock will occur, so we will proceed by seeing how we can detect deadlock and possibly recover from it.

11.4 Deadlock detection and recovery

In any system (real-time or otherwise), the management of resources is absolutely necessary, as most resources simply cannot be shared (consider two tasks sending instructions to a line printer simultaneously). There are two general means of dealing with such a situation:

1. requiring tasks to hold semaphores prior to using a specific resource (where a counting semaphore can be used if there is more than one instance of a resource available), or
2. having the allocation of resources (including semaphores) be the responsibility of an operating system.

We will describe operating systems and how such resource management can be executed a *supervisory* (or *kernel*) *mode* later; however, at this point, it is straight-forward enough to think of a resource as being managed by locking and unlocking semaphores. When semaphores are locked or tasks are blocked on semaphores, it is possible for the software to track these states. Alternatively, it is also possible to simply iterate through all semaphores and determine which tasks are holding or blocked on those semaphores.

Of course, requiring tasks to acquire semaphores before accessing a specific resource is something of a *gentleman's agreement*; that is, a non-binding arrangement that cannot be enforced. In smaller systems, such arrangements can

be very beneficial, as they do not require significant overhead. Such a non-binding arrangement works only as long as everyone follows the rules, and the more participants in the system, the more likely it is that one of them will, intentionally or otherwise, fail in this respect. Thus, in more complex systems, the allocation of resources will be delegated to data structures and functions protected within an infrastructure generally termed an *operating system* and access will be exclusively through this interface with an additional overhead cost.

If there is an operating system in place, the operating system can track which resources (including semaphores) are being held, which resources are being requested, and which tasks are currently blocked on requests.

We will describe how to:

1. probabilistically detect deadlock with watchdog timers,
2. algorithmically detect deadlock, and
3. recover from deadlock if we have detected it.

These will be described in the next two sections.

11.4.1 1 Probabilistic deadlock detection with watchdog timers

In our introductory topic, we already discussed the use of *watchdog timers*. If the dog has not been "kicked" in a specified period of time, it may be assumed that deadlock has occurred and that some corrective action needs to be taken. People do this all the time: if a running program does not respond after a while, they will either kill it or, if necessary, reboot the computer. There is no guarantee that the program was deadlocked or in an infinite loop, but resetting the system was preferable to further, and uncertain, waiting. Unfortunately, there are at least two issues with this approach that must be considered:

1. something other than deadlock that is causing the problem, in which case, the corrective action may not actually solve the problem—for example, the problem may be starvation, and
2. if deadlock is occurring, without further analysis (discussed in the next section), it is not possible to correct the situation apart from more dramatic responses such as rebooting the system.

Consequently, we will continue by discussing an algorithmic approach to deadlock detection.

11.4.2 A brief introduction to graph theory

A directed graph (sometimes called a *digraph*) is a set of points called *vertices* $V = \{v_1, \ldots, v_n\}$ where we denote the number of vertices by $|V| = n$. Together with these vertices, a directed graph also has a set of edges *E*, where each edge is an ordered pair of vertices (v_j, v_k) where $j \neq k$ indicates that that there is a connection from v_j to v_k. The number of edges is denoted by $|E| = n$.

For example, consider the directed graph

$$V = \{a, b, c, d, e, f, g, h, i, j, k, l, m\}$$

with

$$E = \{(a, b), (b, c), (c, d), (d, e), (e, f), (f, g), (g, h), (h, i), (i, f), (i, h), (i, j), (j, k), (k, l), (l, d), (l, h), (l, j), (l, m), (m, b)\}.$$

This is the graph of the block diagram of a position servo with multi-loop feedback, and is shown in Figure 11-8. You can read more about this at the Wikipedia article on single-flow graphs.

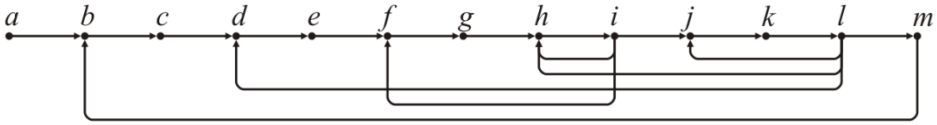


Figure 11-8. The directed graph of a block diagram of a position servo with multi-loop feedback.

The *in-degree* of a vertex *v*, denoted $\deg^-(v)$, is the number of edges coming into the vertex; that is, the number of edges of the form $(u,v) \in E$ where $u \in V$. The *out-degree* of a vertex *v*, denoted $\deg^+(v)$, is the number edges leaving the vertex to another; that is, the number of edges of the form $(v,w) \in E$ where $w \in V$. For example, $\deg^-(a) = 0$ and $\deg^+(a) = 1$ while $\deg^-(i) = 1$ and $\deg^+(i) = 3$.

A *source* is a vertex with in-degree zero, and a sink is a vertex with out-degree zero. In the above graph, there is one source (vertex *a*) and no sinks—the out-degree of each vertex is at least one.

A graph is said to be *bipartite* (from Latin, *bipartīre*, to divide into two) if the vertices can be divided into two mutually exclusive sets $V = V_1 \cup V_2$ with $V_1 \cap V_2 = \varnothing$ where for each edge (v_j, v_k), either $v_j \in V_1$ and $v_k \in V_2$ or $v_j \in V_2$ and $v_k \in V_1$. The graph in Figure 11-8 is not bipartite (why?). A bipartite graph with nine vertices is shown in Figure 11-9.

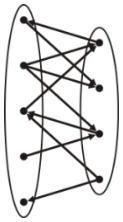


Figure 11-9. A bipartite graph.

A *path of length n* is an ordered sequence (v_0, v_1, …, v_n) of $n + 1$ vertices such that each pair of vertices forms an edge within the graph (there are *n* edges in a path with $n + 1$ vertices). For example, {*i*, *j*, *k*, *l*, *h*} forms a path of length four.

A *simple path* is a path where at most only the first and last vertices are identical, while a *simple loop* is a simple path of at least length 1 where the first and last vertices are identical, for example, {*j*, *k*, *l*, *j*}.

We will use bipartite directed graphs in our deadlock detection algorithms.

11.4.3 Algorithmic deadlock detection

To detect deadlock, it is necessary to construct a task-resource graph. Such a graph could either be

1. maintained globally and updated each time a semaphore or resource is requested or allocated, or
2. constructed locally and dynamically built based on the current states of the tasks when attempting to detect deadlock.

This graph would be a directed bipartite graph; that is, a graph where the vertices can be divided into two sets where each edges originates in one set and ends in the other. An example of such a possible scenario and the associated directed bipartite graph is shown in Figure 11-10.

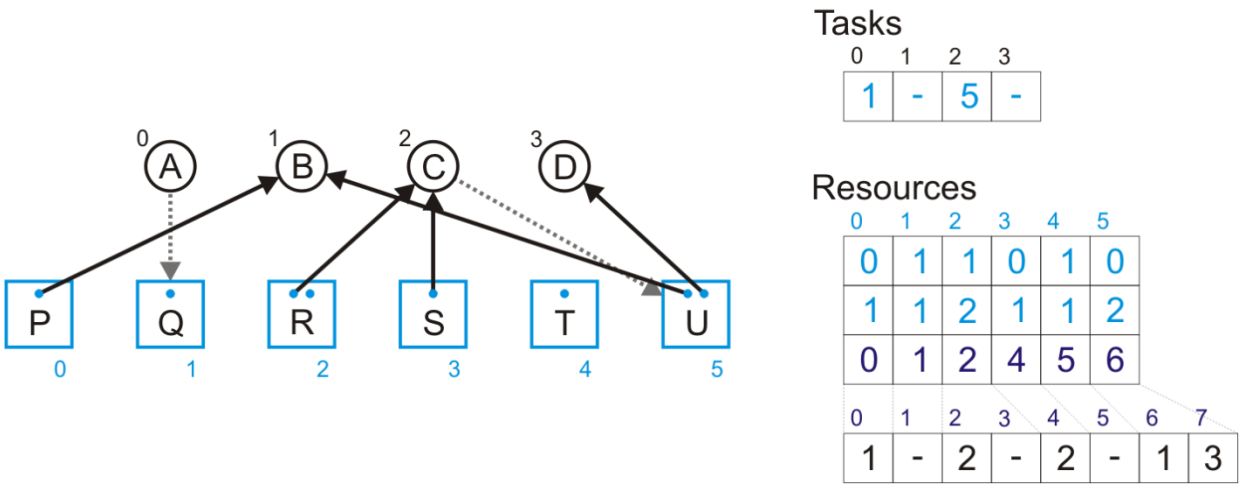


Figure 11-10. A scenario with four tasks and six resources, two of which have two instances.

In this graph, a dash for a task indicates that it is not currently waiting on a resource, whereas a number indicates that it is waiting on the corresponding resource; for each resource, we must know how many are free, the total number of instances of each resource; and for each instance of a resource, a dash indicates the resource is not locked by any task, whereas a number indicates that it is locked by that task. The last number in the resource columns indicates the initial position of that resource in the instance allocation array.

We have already seen in Section 11.3.4 that circular waiting is one requirement of deadlock, and it is this that we will search for in our graph; however, this isn't just an issue of finding cycles. For example, in Figure 11-11, the first situation is deadlocked, but in the second situation, Task A could still have its request satisfied, at which point, we will simply assume that it will release Resource P, which can then be given to Task B which may then continue executing.

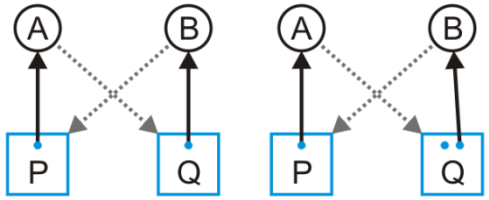


Figure 11-11. Two situations, one deadlocked and the other not.

Thus, we will require an algorithm that will allow us to strip away edges that are not associated with deadlock, at the end of which, any remaining edges should indicate deadlock. We will look at three variations on this algorithm:

1. all resources are reusable and a task can only be blocked on request for a single resource,
2. resources are either reusable or consumable and a task can only be blocked on request for a single resource, and
3. resources are either reusable or consumable, and a task can be blocked on a request for multiple resources simultaneously.

Each is more difficult than the previous.

11.4.3.1 Reusable resources with blocking on requests for individual resources

This easiest case requires us to simply remove edges that are not associated with deadlock. As a deadlock requires a cycle, we can make two observations:

1. any task not waiting on a resource cannot be deadlocked, and
2. any resource for which there is at least one instance available cannot be involved in a deadlocked.

The first situation is obvious—the task can execute as it is not waiting for any resources. Therefore, we will assume that it will continue executing and that at some point it will release its resources so that they can be used by other tasks.

The correctness of the second statement is more subtle: any task that is requesting that resource can have that request satisfied, so it could continue executing. Consequently, that task will continue executing and at some point it will release the resources it holds—including this resource—so that this resource can be assigned to the next task requesting it.

Thus, our algorithm consists of two steps:

1. Iterate through all tasks, and any task
    a. not waiting on any resources may be flagged as having given up all resources (that is, all incoming edges are removed), and
    b. not holding any resources may be flagged as no longer requesting those resources (that is, all outgoing edges are removed)—that task cannot be involved in deadlock, though it may wait on resources involved in deadlock.
2. Next, iterate through all reusable resources, and any resource for which the number of available resources (those not currently flagged as being assigned) equals or exceeds the number of unsatisfied requests for the resource may be flagged as having satisfied all requests (that is, all incoming and outgoing edges are removed).
3. If at least one edge remains and if at least one edge is removed, go back to Step 1; otherwise, we are finished.

On completion, there will be one of two situations:

1. all edges are removed—there is no deadlock; or
2. the remaining edges exist in cycles that indicate deadlock.

Why are we allowed to assume that a task will ultimately give up a given resource? This is again a design issue: if a resource is assigned in perpetuity to a given task, we may as well not consider it as a resource, as it is bound to one and only one task (it's not technically a *resource* under that condition). Thus, we are concerned primarily with those resources (buses, peripherals, communication channels, etc.) that are used and then released by the various tasks. Failure of a task to give up a resource is therefore a fault in the software.

We will illustrate this by looking at five examples.

11.4.3.1.1 Example 1

Consider the scenario in Figure 11-12. Tasks B and D have no outstanding requests, so we may assume that they will complete and they will subsequently give up their resources. At this point, Resources R and S have no outstanding requests, so they cannot be involved in deadlock. Resources Q and U have outstanding requests, but they can be satisfied. Consequently, they are also not involved in a deadlock. This removes all edges, and the system is not in deadlock.

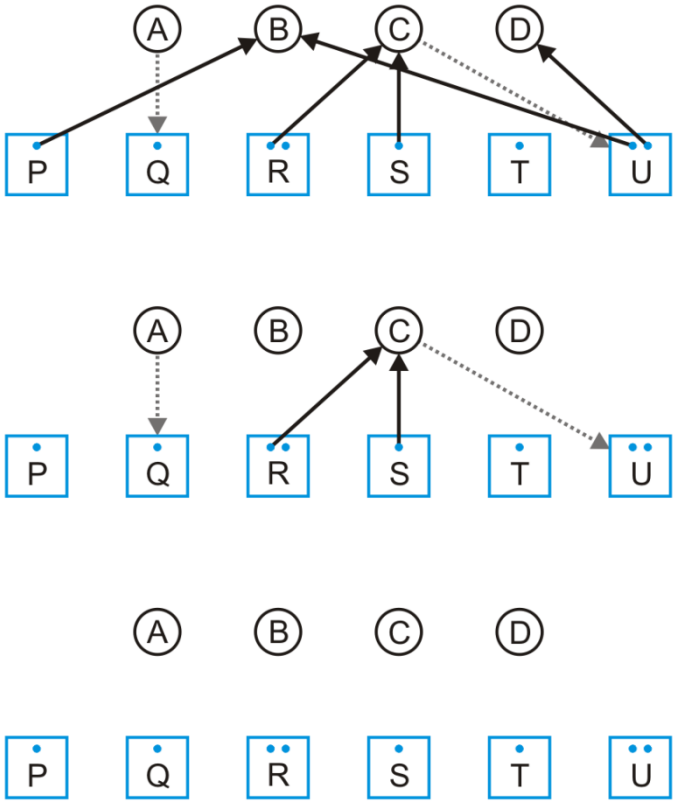


Figure 11-12. Deadlock detection algorithm, Example 1.

11.4.3.1.2 Example 2

Consider the scenario in Figure 11-13. Task C has no outstanding requests, so it may complete. Next, Resource U has no outstanding requests, so it cannot be involved in deadlock, so remove those edges, and Resource Q can satisfy its one request, so it cannot be in deadlock, so remove that edge. At this point, we cannot remove any further edges from either the tasks or the resources, so deadlock exists with the cycle B→T→D→P→B.

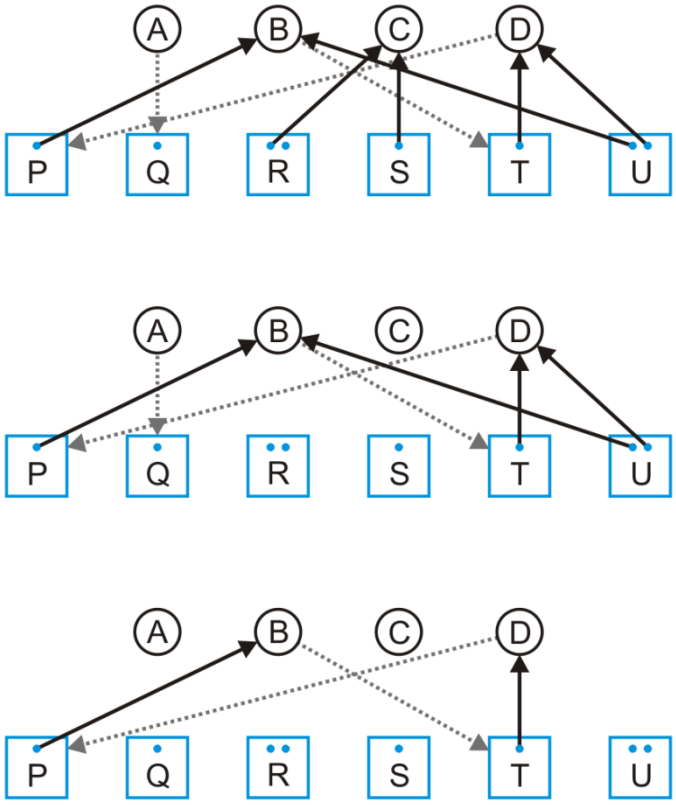


Figure 11-13. Deadlock detection algorithm, Example 2.

11.4.3.1.3 Example 3

Consider the scenario in Figure 11-13. None of the tasks can continue executing, so we examine the resources. Resources P and R have no outstanding requests and Resource Q can satisfy its one request, consequently, we can remove those edges. The edges that are left are, however, in deadlock with two cycles: B→S→C→U→B and D→S→C→U→D.

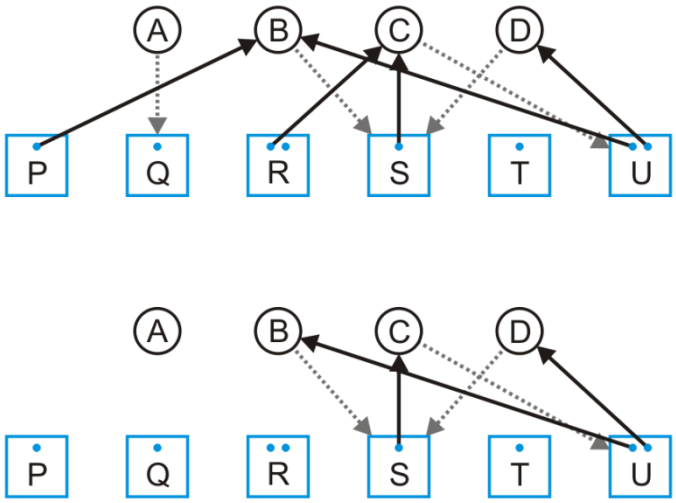


Figure 11-14. Deadlock detection algorithm, Example 3.

11.4.3.1.4 Example 4

Consider the scenario shown in Figure 11-15. Here, if we implement the algorithm as described above, the run time could be prohibitively expensive. The runtime, as described, would be $O(|T|^2 + |R|^2)$ where $|T|$ and $|R|$ are the number of tasks and reusable resources, respectively.

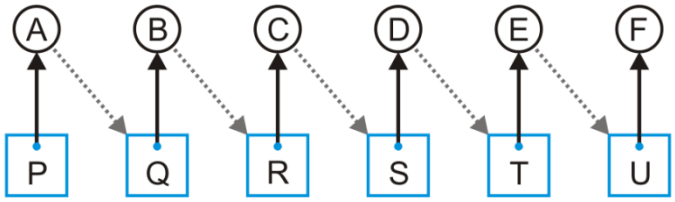


Figure 11-15. Worst-case scenario for the deadlock detection algorithm.

11.4.3.1.5 Example 5

As an example of how a task could be deadlocked but not within the cycle, consider Figure 11-16. Here, Task C is deadlocked, but because the resource it is requesting appears in the deadlock cycle A→Q→B→P→A.

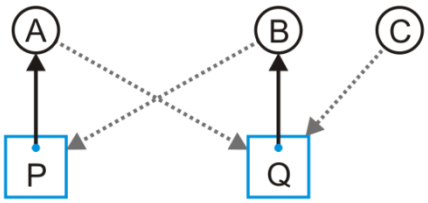


Figure 11-16. Non-cyclic possibility for deadlock.

11.4.3.2 Reusable and consumable resources with blocking on requests for individual resources

When a resource is consumable, the algorithm is only slightly more complex. The change is highlighted in red.

1. Iterate through all tasks, and any task
    a. not waiting on any reusable resources may be flagged as:
        i. having given up all resources (that is, all incoming edges from reusable resources are removed), and
        ii. able to produce arbitrary amounts of any consumable resource it is designated to produce (that is, all incoming edges from consumable resources are removed, and that resource is flagged as being able to produce arbitrary amounts of that consumable resource),
    b. not waiting on a consumable resource where there is sufficient number available for all requests (the in-degree is less than or equal to the number of consumable resources currently available), and
    c. not holding any reusable resources may be flagged as no longer requesting those resources (that is, all outgoing edges are removed)—that task cannot be involved in deadlock, though it may wait on resources involved in deadlock;
2. Next, iterate through all reusable resources, and any resource for which the number of available resources (those not currently flagged as being assigned) equals or exceeds the number of unsatisfied requests for the resource may be flagged as having satisfied all requests (that is, all incoming and outgoing edges are removed).
3. Third, iterate through all consumable resources, where
    a. if that resource is flagged as being able to produce arbitrary amounts, then all requests may be filled (that is, all incoming edges are removed, and consequently, all outgoing edges can also be removed), and
    b. if that resource has an available number of consumable resources that equals or exceeds the number of requests (incoming edges), then all those requests can be satisfied, so again, we remove all incoming and outgoing edges; and
4. If at least one edge remains and if at least one edge was removed, go back to Step 1; otherwise, we are finished.

When finished, we have one of three situations:

1. all edges are removed—there is no deadlock; or
2. the remaining edges exist are either in:
    a. cycles that indicate
        i. deadlock if there are no consumable resources available in the loop, or
        ii. potential deadlock if a consumable resource is available, but the requests for that resource exceed the availability, or
    b. tasks not in cycles but requesting consumable resources that are involved in cycles (but now, only possibly indicating deadlock).

If there are some consumable resources left, this indicates it may still be possible to prevent deadlock by a judicious allocation of those available resources. We will look at a few examples.

11.4.3.2.1 Example 1

Consider the example in Figure 11-17. The only task that can get its resources is Task D. We assume it can now produce sufficient amounts of consumable resources Y and Z. At this point, Task B's request for consumable resource Y is satisfied, so it executes and it is now assumed to produce sufficient amounts of Consumable Resource X. At this point, both Tasks A and C can acquire their resources and they are assumed to finish.

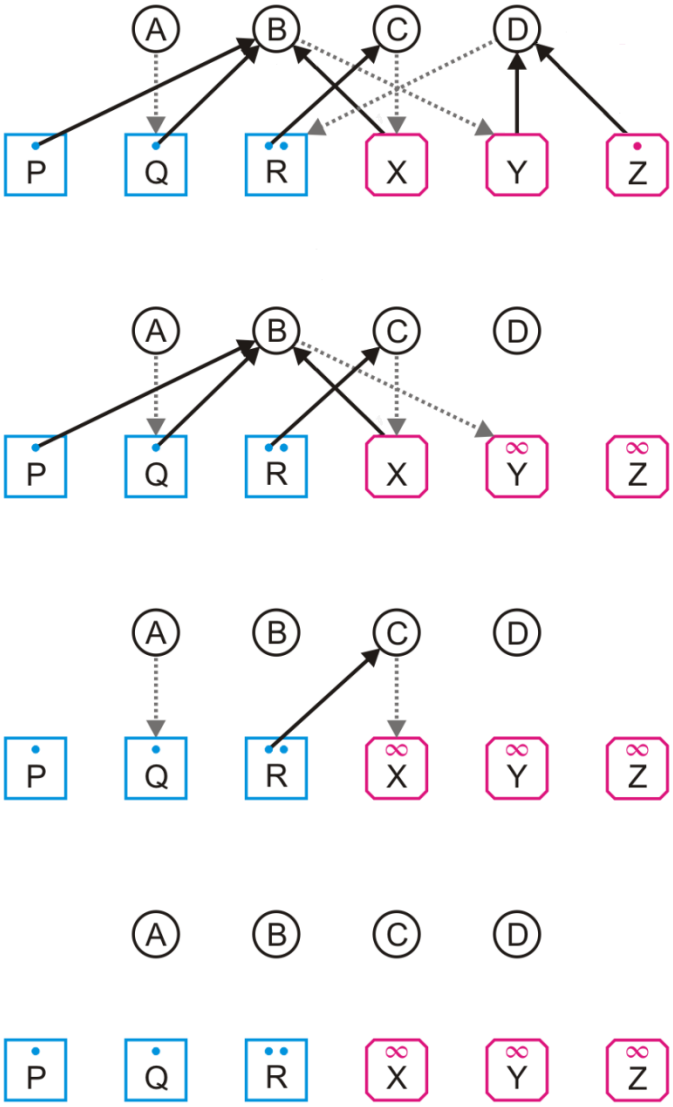


Figure 11-17. Deadlock detection algorithm with consumable resources, Example 1.

11.4.3.2.2 Example 2

In the example shown in Figure 11-18, Tasks F can run so it is assumed it will produce a sufficient number of Consumable Resource Z, Task A can satisfy its request, so it is assumed to give up its one resource, and Resource R has no outstanding request, so its ownership can be removed. Next, Task B and D requests can be satisfied, so they are assumed to finish executing, and now sufficient amounts of Consumable Resource X can be produced. Unfortunately, at this point, Tasks C and E cannot continue executing, and so the two are deadlocked.

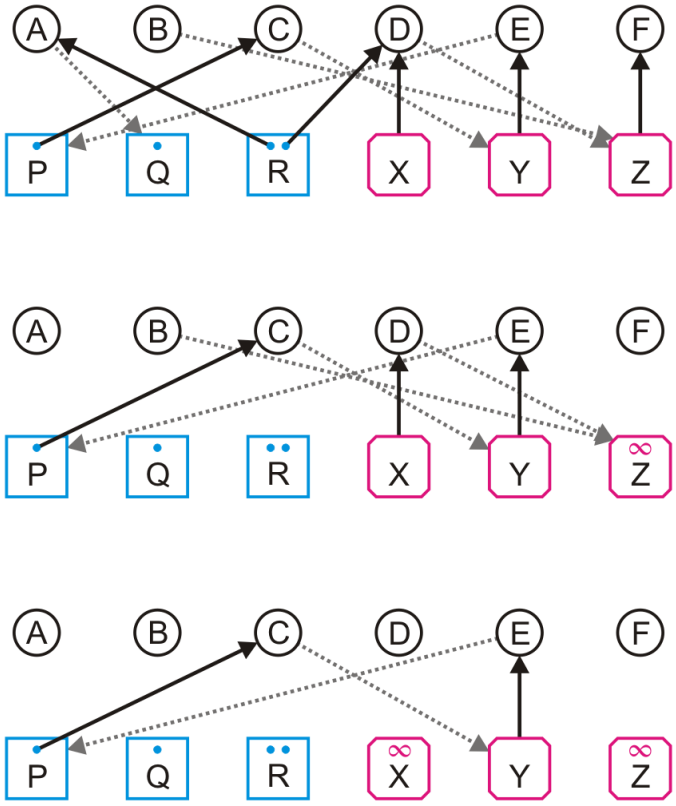


Figure 11-18. Deadlock detection algorithm with consumable resources, Example 2.

11.4.3.2.3 Example 3

Finally, consider the setup in Figure 11-19. If the Consumable Resource X is allocated to Task E, then Task E can continue executing and produce a sufficient amount of Consumable Resource Y. Further investigation also shows that Resource R is not in demand, thus Task A, while currently blocked, cannot be in deadlock, and thus Resource P being held by Task B is not in deadlock, either. However, Tasks B, C and D are still in deadlock.

If, however, the resource is allocated to Task C, all the tasks can complete and sufficient amounts of Consumable Resources X and Z are now produced, allowing Task E to acquire its requested resource, and thus we are finished. Consequently, deadlock detection with consumable resources may also determine whether or not an incorrect allocation may result in deadlock. The only question now is: does Task E have higher precedence than Task C, and if so, are we willing to tolerate the deadlock in order to allow Task E to continue?

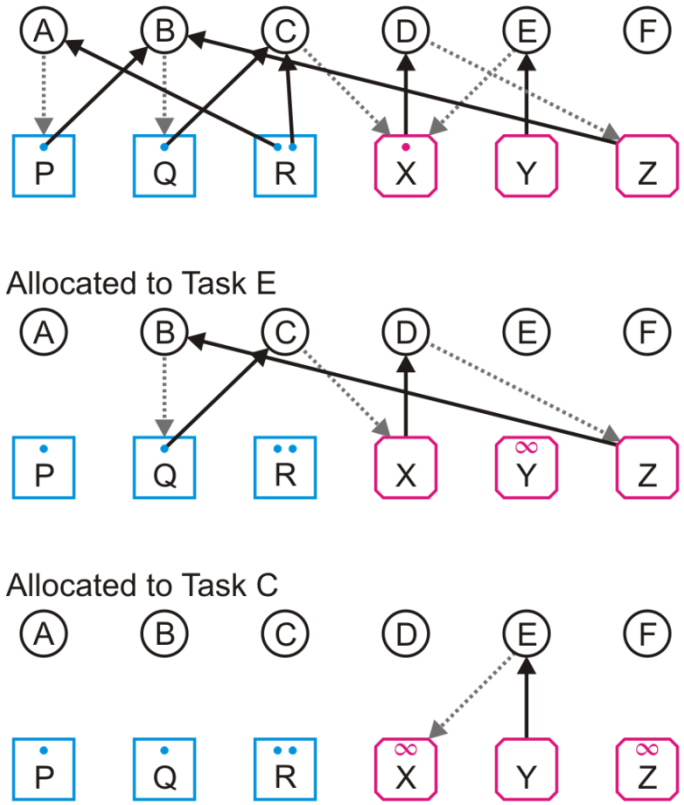


Figure 11-19. Deadlock detection algorithm with consumable resources, Example 3.

11.4.3.3 Cycle detection algorithms

Before delving into any algorithm, it is always important to consider the character of your problem. First, we are looking for cycles, and by our algorithm, the only edges left must be in cycles indicating deadlock.

11.4.3.3.1 Cycle detection with reusable resources with blocking on individual requests

As long as tasks cannot be blocked on more than one resource, and if all resources are reusable, all edges must be within a cycle, and we may make the additional observation that either:

1. the cycles are disjoint, or
2. the cycles intersect at resources.

Thus, it is only necessary to select any vertex and perform a traversal of the graph. Once that traversal is complete, all vertices in that traversal are involved in deadlock. By removing any one edge, deadlock is ended. If two or more cycles intersect at a reusable resource, then breaking either cycle will also break the other cycle: removing any link will ultimately make that resource available for the other cycle, as well.

Note that by observing the characteristics of the problem, and noting that tasks can have an out-degree of at most one, we can simplify the problem significantly, thereby allowing us to use significantly simpler algorithms. One could implement, for example, Tarjan's strongly connected components algorithm; however, that would be not necessary in this case.

11.4.3.3.2 Cycle detection with consumable resources with blocking on individual requests

If the system contains consumable resources, there may be individual tasks requesting a consumable resource where one is available; however, from the algorithm, that consumable resource must also be trapped in a deadlocked loop. Thus, any task that has in-degree zero and out-degree one is making a request for a consumable resource that is currently in another cycle.

If a consumable resource is not available within a cycle, the system is in deadlock in a manner similar to that described in the previous section.

If a consumable resource is available, technically, the system is not yet in deadlock; however, if that consumable resource is allocated to a task that is not in a loop, deadlock will occur. If, however, the consumable resource is allocated to a task within the cycle, that cycle will ultimately finish and the last entry in that cycle will become an active producer of that resource again. This is a case where deadlock detection may actually prevent deadlock from occurring.

11.4.3.3.3 Cycle detection with blocking on multiple requests

If a task can be blocked on multiple requests, the algorithms become much more complex, and recovering from deadlock will necessarily be more complex. Thus, allowing a task to make multiple requests simultaneously will solve the issue of 11.3.2 *hold-and-wait*; however, it makes recovery more difficult.

11.4.3.3.4 Summary of cycle detection algorithms

As long as we allow tasks to be blocked only on individual requests, once we have stripped out all other edges, the remaining edges are either within cycles or adjacent to a cycle. In this case, we can use very simple traversals to find those cycles. In the case of reusable resources, this would indicate a deadlock. In the case of consumable resources, there is the possibility that a judicious allocation of that resource will end the deadlock.

11.4.3.4 The algorithm when tasks may be blocked on requests for multiple resources

When this algorithm is presented in most textbooks, the locking on a single resource is so simple that they often jump to the case where a task can be blocked on multiple simultaneous requests. Now it becomes much more difficult to

determine if deadlock exists, as the algorithm is no longer as trivial. Tasks can now be involved in multiple cycles, and breaking one cycle may not break the other.

11.4.3.5 Summary of deadlock detection

We have described algorithms for detecting deadlock with both reusable and consumable resources. While we could focus on the general case where a task can be deadlocked on multiple requests, most systems will see tasks blocking on individual requests; consequently, we focused on this case. In situations where a task may be blocked on multiple requests, the reader is invited to consult other textbooks on the matter. In the case where tasks are blocked on a single resource, we will see that recovery is quite straightforward. In general, however, there are no straightforward algorithms for cleaning up deadlock in such cases. Before we discuss recovery, we will however quickly discuss cycle detection algorithms. Following that, we will look at recovering from deadlock and then asking when we should run deadlock detection algorithms.

11.4.4 Recovery

When a deadlock is detected, it is necessary to break the deadlock. If there are multiple cycles where deadlock is occurring, deadlock recovery should begin by breaking the cycle in which the highest priority task is involved.

There are two possible solutions:

1. preemption,
2. restarting the system,
3. kill a task and recover all of its resources, or
4. roll back a task to a point where it does not have a resource that is required.

We will consider all of these. It should be noted, however, that in situations where tasks may make multiple simultaneous requests and to be blocked on those requests, some of these solutions may become more difficult to implement.

11.4.4.1 Preemption

As discussed in Section 11.3.3, if it is possible to pre-empt a resource that is involved in a deadlock, this would be the simplest solution. One problem in a real-time system, however, may be that it is a resource that is being pre-empted from a high-priority task to a low-priority task (not all resources are pre-emptible, so it may be the case that within a cycle, we have this situation). In this case, it may be advisable to execute all other tasks in the cycle at the priority of the higher-priority task; otherwise, we may become involved in a priority inversion, as described in Section 8.13.

11.4.4.2 Rebooting the system

As has been demonstrated by our examples with the NASA Pathfinder mission and the Sprit rover, one solution is to simply reboot the system and hope that the conditions that caused the deadlock are no longer valid at some point in the future. While both of these were running real-time systems, starting again was considered an acceptable solution and no doubt sporadic reboots of Spirit still occur, as this requires the least effort. One issue, however, as demonstrated in the case of both systems, this is very much a case of hoping that the conditions that caused the deadlock do not reoccur.

11.4.4.3 Killing a task

When deadlock is detected, it may be possible to choose any task within a cycle and to kill it, thereby freeing up its resources for another task. In general, it would be appropriate to kill the lowest-priority task within the cycle and, if necessary, start that task up again. The resources it previously had would be allocated to the higher priority tasks and the restarted lower-priority task will have to start requesting resources to begin executing.

As an example, suppose a low priority backup task was executing and had locked a device, the information in which is to be backed up. However, when it requests a communication bus that is already in use, it is blocked. If the task using the communication bus then requests the device held by the backup task, we have deadlock. The low priority backup task is

killed, the backup is not completed, and another device is given the previously locked device. The backup task would be started again, and it would again, when the opportunity occurred, begin requesting resources to make the appropriate backups. Now, if backups were considered more important, then the priorities may be different and another task might be killed instead.

11.4.4.4 Rolling back to a check-point

A check-point is a snap-shot of the state of a task so that it is possible to begin re-executing that task at that point at some time in the future. This is, in general difficult, but may be possible with lower priority tasks that are not affecting the system. For example, when a resource is being requested, because it is a low priority process, this means there is likely time to, at the very least, make a copy of the state of the processor and perhaps a copy of at least some portion of the call stack. Then, if it becomes necessary to roll back to the check point, the state of the processor would be restored, the stack would be recovered, and execution would continue, only now, the task will be blocked on that particular resource.

Figure 11-20. Parody.[31]

We will look at two examples where a roll back is possible.

11.4.4.4.1 Example with memory allocation

Suppose that a task has requested some memory to begin copying data from another device that it has currently locked. Just prior to the memory being allocated to the task, a snap-shot of the task is made, and the memory is duly allocated. If at some point in the future, the task becomes involved in a deadlock and the block of memory is required by another task, this task could be rolled back to the check point just prior to the memory request. The memory is therefore allocated to the other task involved in the deadlock, and the deadlock is broken. This task will wait for its memory to be allocated and will start the process again.

Note that this is not necessarily always possible. Suppose, for example, in another block of memory (as opposed to a register), the task is tracking how many bytes have been copied. If the task relies on this value, it may be necessary to roll back the memory location, as well—something which may be significantly more complicated.

11.4.4.4.2 Example with a communication bus

Suppose a low priority task is involved in a deadlock where that task has locked a communication bus. That task could be rolled back to a point just prior to the request for the communication bus. The communication bus would be duly allocated to the higher priority task, thus breaking the deadlock. The lower priority task that was rolled back would be blocked on requesting the communication bus and when it is finally allocated that resource, the task will, again, begin transmitting. Any task receiving the communication will, of course, have to be prepared to have the communication restarted.

11.4.4.5 Examples 11.4.1.1.2 through 5

What must be done in Examples 11.4.1.1.2, 11.4.1.1.3 and 11.4.1.1.5 to break the deadlock?

In 11.4.1.1.2, deadlock could be broken by either:

1. pre-empting Resource P from Task B and giving it to Task D,

[31] Parodied from Wal-mart Stores, Inc.

2. pre-empting Resource T from Task D and giving it to Task B,
3. rolling back or killing either Task B or D.

In 11.4.1.1.3, in order to break this deadlock, it is necessary to either:

1. pre-empt Resource S to either Task B or D,
2. pre-empt Resource U and give it to Task C, or
3. kill or roll back any one of Tasks B, C or D.

Note that it is not necessary to break both cycles: because the cycles overlap, breaking one deadlock will break the other.

In 11.4.1.1.5, deadlock cannot be broken if Task C is killed, as it is not contained in the deadlock cycle. The other four tasks are deadlocked in a manner similar to 11.4.1.1.2.

11.4.4.6 Summary of deadlock recovery

In this topic, we looked at four possible solutions to recovering from deadlock. Under certain circumstances, a reboot of the entire system may work. This is no different than rebooting your computer when it locks. This is however, not the best possible solution: preemption may be possible, lower priority tasks could be killed and restarted, and tasks can be rolled back to checkpoints. Some of these solutions, specifically rebooting or killing tasks, operate under the hope that the next time the tasks run, the circumstances that caused the deadlock do not repeat themselves. Preemption or rollback, together with an appropriate use of priorities, is more likely to reduce the likelihood of subsequent reoccurrences of the same deadlock.

11.4.5 When to perform deadlock detection

When in a real-time system might you consider performing deadlock detection?

1. This is certainly one of the jobs that could be performed by the idle task. After all, if nothing else is ready to execute, there may be a problem.
2. Such a check could be sporadically executed if no real-time critical tasks are currently executing. Such a task may have higher priority than all non-real-time tasks, but lower priority than all real-time tasks. If this task executed once per second (or however often is deemed necessary), it would make sure that there are no deadlocks within the real-time tasks.
3. Such a check could also be executed if tasks begin to miss their deadlines.

Fortunately, if tasks lock resources serially, it is only ever possible for a task to deadlock within a single cycle. Therefore, the algorithm is much faster than if tasks are allowed to request and become locked on multiple resources simultaneously.

11.4.6 Summary of deadlock detection and recovery

This topic covered deadlock detection and recovery. If resources are being allocated by an operating system and tasks may request and lock those resources, it is possible for the operating system to determine whether or not deadlock has occurred by examining the directed graph of tasks with locks and requests for resources. If a cycle is found, it is possible to recover from the deadlock by either taking a resource away from one of the tasks (difficult at best and it would be desirable that the task has a low priority), rebooting the entire system and hoping the conditions that caused the deadlock do not occur again, killing a task within a cycle (preferably the one with lowest priority) and restarting it, or setting checkpoints whenever tasks request a resource and rolling tasks back to those checkpoints if necessary. Running deadlock detection is, however, time consuming and consequently, it should only be performed as a background task—if high priority tasks are executing, chances are deadlock has not occurred. If tasks are starting to miss their deadlines, deadlock may have occurred.

11.5 Deadlock avoidance

Most textbooks on operating systems consider *deadlock avoidance*: techniques for not allocating resources that may lead to a deadlocked situation; for example, the *banker's algorithm*. In general, such techniques are not reasonable for real-time systems. If you're interested in such topics, please see Section 6.6 of Tanenbaum, *Modern Operating Systems*, 3rd edition, pp.448-54. To quote that source, however, "[u]nfortunately, few authors have had the audacity to point out that although in theory the [Banker's] algorithm is wonderful, in practice it is essentially useless…"

11.6 Summary

This topic has covered the issue of deadlock where separate tasks become involved in situations where some resources are held and others are requested, and if such a sequence of requests by blocked tasks forms a cycle, this will lead to a situation where none of the tasks will ever be able to continue executing. This situation can be modeled most easily through a bipartite graph of tasks and resources, and techniques for preventing deadlock include breaking any of the four required conditions for deadlock to occur. If deadlock does, never-the-less, occur, it is possible to detect and then recover from deadlock through numerous techniques, some more subtle than others, including rolling back a task to a prior check-point, killing the task, pre-empting the resource, and rebooting the system.

Problem set

11.1 What are the four requirements for deadlock?

11.2 Provide an argument or proof that requiring tasks to obtain resources in a particular order will prevent deadlock.

11.3 Suppose you had two acquire and lock two objects. Suggest two techniques that could be used to order the objects to ensure that acquiring those objects will not cause deadlock.

11.4 How are critical regions for mutual exclusion similar to interrupt service routines?

11.5 When would you run deadlock detection?

11.6 Why would resetting a system potentially solve a deadlocked situation? Why is this not a satisfactory solution in a hard real-time system? Why is this appear to be acceptable in, for example, space missions?

11.7 When would you run a deadlock-detection algorithm?

11.8 Why is it essential that a deadlock-detection algorithm not request any memory in order to run?

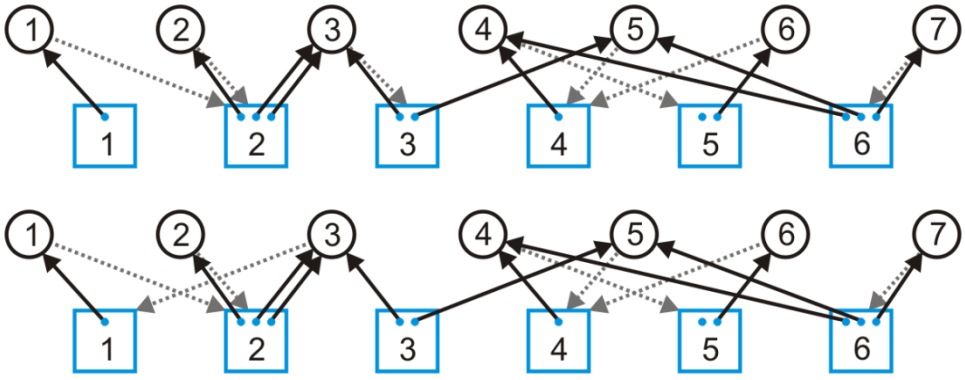
11.9 Why is it essential that if a deadlocked situation is found that it is not a high priority task that is killed in order to end the deadlock? What could potentially happen?

11.10 Why is rollback not an appropriate solution for all tasks?

11.11 At which point should a snapshot be taken of a task if it is to potentially be rolled back as a result of being detected in a deadlock?

11.12 Is there any point in being able to roll back a high priority task?

11.13 Which of the following situations is in deadlock?



11.14 If either of the situations in Question 11.13 is not deadlocked, this means that one of the tasks can acquire a resource. Assume that resource is acquired. What would happen if that task attempted to acquire any other resource?

12 Communication and distributed systems

To date, we have focused on threads and tasks running on a single processor that share main memory, and through the use of semaphores, it is possible for different tasks to communicate and signal each other. We will refer to the task or thread sending the message as the *sender*, and we will say that the sender *posts* the message when it is ready to be sent. The task or thread receiving the message will be the *receiver*, and we will say that the receiver is *waiting* for the message to be sent. We will now begin by considering the various classifications of messages.

12.1 Classification of communications

We will classify communications on

1. what is shared,
2. the behavior of the sender after posting the message and the receiver when waiting on a message,
3. the purpose of the communication,
4. the size of the message, and
5. whether messages are allowed to accumulate.

In larger systems, where separate processes may not share all their resources, or where the processes may be running on completely separate systems, for those processes to communicate, it is never-the-less necessary that they must share some resource, be it

1. main memory,
2. secondary memory, or
3. a communications network.

Access to these may be mediated through a memory manager, file system manager, or resource manager, possibly as part of an operating system. For example, in GNU/Linux, if two threads belong to separate processes, they do not share main memory; it is necessary for the threads to make a request to the operating system to designate a block of main memory as being shared. As systems become larger, such managers will inevitably be incorporated into an operating system that maintains exclusive control over the resources available to the tasks.

When the sender posts the message, there are two possible outcomes:

1. the sender continues executing as soon as the message is posted, or
2. the sender is blocked from executing until the message is received by the receiver.

Similarly, when a receiver waits on a message, if no message is waiting, there are also two possible outcomes:

1. the receiver is notified that no message is available and the receiver continues executing, or
2. the receiver is blocked from executing until a message from the sender is received.

Thus, we have four possible scenarios, three of which are common:

1. *synchronous send-and-receive*, where the sender is blocked until the receiver waits on and receives the message and the receiver is blocked when it waits for a message but no message has yet been received,
2. *asynchronous-send and synchronous-receive*, where the sender continues executing as soon as the message is sent, but the receiver is still blocked when it waits for a message but no message has yet been received, and
3. *asynchronous send-and-receive*, where the sender continues executing once the message is sent, and the receiver queries whether a message is waiting and continues executing if it is not.

The last scenario, *synchronous-send and asynchronous-receive* is so rare as to not merit any further discussion. The most common scenario is the asynchronous-send and synchronous-receive. In this case, the receiver may, never-the-less, send an acknowledgment that the message was indeed received.

The type of message being sent and received may be described as either for

1. synchronization, or
2. communication.

The information of a message can be either *implicit* or *explicit*:

1. An *implicit* message in where receiving the message itself is sufficient to convey the information, which is usually the case of semaphores but it is also possible to post a blank message.
2. An *explicit* message is where the message itself must contain additional information to convey the significance of the communication. Usually it is not enough just to know that a message has been received, the content of the message must be analyzed.

In the case of communication, messages tend to follow a specific format, with a header containing meta-data about the message, followed by the message itself. This meta-data may include information such as:

1. the sender,
2. the receiver,
3. the message type,
4. the length, and
5. additional control bits

all followed by the actual data.

In general, in any real-time system, messages will be continually passed between threads and tasks, and therefore we will not consider the scenario where only one message is being sent or received.

When multiple explicit messages are being sent, their size will be either

1. fixed, or
2. variable.

A fixed-sized message places additional restrictions on the sender, but at the same time this greatly simplifies the mechanisms of sending and receiving the messages. Often, variable-sized messages will be treated as fixed-sized messages by intermediate mechanisms that deal with sending the message.

Finally, we will classify messages as to whether:

1. posted messages must be received before subsequent messages can be posted, or
2. multiple messages can be stored in a queue as receivers wait on them.

As a real-world example, consider the process of sending mail through Canada Post, the only shared service is the postal service. Both sending and receiving are asynchronous, and it requires polling to determine whether mail is present. Mail is delivered periodically on weekdays with a period of one day. The purpose of mail is communication and the size is variable. Finally, messages can accumulate until they are retrieved.

12.2 Solutions for communication

In addition to the example of Canada post, we will consider a few other applications of inter-process communication under various combinations of the above classifications, including

1. binary and counting semaphores,
2. shared memory,
3. pipes,
4. message queues,
5. pipes and message queues in secondary memory,
6. sockets,
7. ports, and
8. mailboxes.

12.2.1 Binary and counting semaphores

A binary semaphore shares main memory, uses asynchronous-send and synchronous-receive, is used for synchronization with an implicit message, and messages must be received (waited on) prior to additional messages being posted.

A counting semaphore shares main memory, uses asynchronous-send and synchronous-receive, is used for synchronization with an implicit message, but messages can be stored in a queue as receivers wait on them.

Note that the terminology of *wait* and *post* becomes clearer when we start discussing the token of the semaphore as a message? We *post* a letter, and we *wait* for a letter to arrive. The binary semaphore is a data structure that holds that letter while it is not being held by other tasks or threads.

12.2.2 Shared memory

Another means of communicating messages using shared main memory is for the sender to allocate a block of memory for the message, and when it is ready to send that message, it uses a semaphore to signal the receiver that the message is ready. The receiver would wait on the semaphore and then access the shared memory location.

In this situation, it is normal for the ownership of the allocated memory to transfer from the sender to the receiver: the sender allocated the memory, but it is up to the receiver to deallocate it. This will almost certainly be the case if the message size varies from one message to the next.

Alternatively, the receiver could signal the sender after it has processed the message, at which point, the sender could reuse or deallocate the memory block.

12.2.3 Pipes

Another data structure that can be used to facilitate communication of explicit messages using shared main memory is a pipe. A pipe is a block of main memory that is interpreted as a circular queue, where each entry of the queue is fixed in size (say, one character). The sender divides the message into word-sized chunks and places each word into the circular queue one at a time. The receiver receives data from the pipe one character at a time.

Because the sender and receiver only post or wait on one character at a time, it is necessary for both of them to allocate sufficient memory for the message, and each will maintain the memory The sender and receiver will have to agree upon a specific character to specify that the full message has been sent—perhaps the null character `'\0'` or `0x00`. Alternatively, the first four bytes could be used to specify the size of the message. In this case, the receiver would simply pop that number of characters from the pipe.

One issue with a pipe is that the sender may attempt to push more data into the pipe when it is currently full—perhaps the receiver is of lower priority or it is still processing the data it has extracted up to this point. At this point, the sender would be blocked until the receiver has removed at least one character from the pipe. At the same time, if the receiver attempts to pop data from a pipe that is empty, it would be blocked until the sender finally pushes at least one character into the pipe.

Now we have an interesting situation when it comes to priority imbalance: suppose that the sender and receiver have different priorities. In a real-time system, that task or thread with the highest priority is the one that is running and

therefore if the sender has higher priority, it will fill the pipe, and as soon as the receiver takes out a single character, the sender will be woken and immediately fill the pipe again. As you may suspect this would require a context switch for each byte transferred. It would be the mirror image if the receiver had a higher priority. Consequently, it is likely that the sender and receiver should have, at least for the duration of the transfer, have equal priority.

In a shell such as tcsh in GNU/Linux, the following command automatically creates a pipe

```
% find / | more
```

taking the output of `find` to the pagination program `more`.

As we have described them, pipes are unidirectional; however, it is possible to design pipes so that the direction of the flow of information can be changed; such a pipe is said to be half-duplex. Pipes can be designed to be half-duplex or full duplex, the latter requiring essentially two circular queues.

12.2.4 Message queues

A message queue is similar to a pipe, except that it stores entire messages in fixed-size packets, as opposed to having the sender and receiver push and pop the bytes one at a time. As messages are fixed in size, the sender and receiver may have internal fragmentation.

12.2.5 Pipes and message queues in secondary memory

A pipe or message queue could be implemented using secondary memory as opposed to main memory. In such a case, the information could be persistent beyond the life of the tasks and it would even survive a reset of the system. A pipe stored in secondary memory is often referred to as a *named pipe*, as it is essentially a file with a name in the file system.

Another approach is a memory-mapped file. This requires the support of both the memory manager and the file system manager, where a file is copied into main memory, and any changes to the data in main memory is automatically copied to secondary memory.

12.2.6 Mailboxes

To be completed.

12.2.7 Ports

A port is a maritime facility that allows ships to load or unload passengers or cargo. A port will contain one or more wharfs where ships can dock and ships and their clients will coordinate with the port authority as to which wharf will be used for a particular transaction. The various tasks running on a processor may need to communicate with one or more tasks located remotely on other processors. How this communication takes place may depend on the scenario, but how the information is being sent out, and how it is being received needs to be regulated. One such approach is one very similar to wharfs in a port; however, the designers choose to use the word *port* instead of *wharf*.

A system that uses ports for communication allows for $2^{16} = 65536$ different port identifiers, from 0 to 65535, although port 0 is reserved and not used (similar to a null pointer). Two tasks must agree apriori that they will communicate via specific ports. Thus, Task A on System 1 will use Port 73 and Task B on System 2 will use Port 98. Task A must arrange with System 1 that it will use Port 73, and each time it sends a *datagram* to Task B, it will indicate that the datagram must be delivered to Port 98. Task B must arrange with System 2 that it will use Port 98, and once that arrangement is made, Task B can query if any datagram has been received. Numerous port numbers have already been assigned for specific tasks; for example, Port 80 is for HTTP for transferring web pages while Port 443 is for secure HTTPS. All ports up to Port 1023 are described as *well-known ports*. Ports 1024 through 49151 may be registered by software developers for a specific task (for example, Port 3101 is used by BlackBerry Enterprise Server communications, while Ports 49152 through 65535 are publicly available.

One common means of communicating through ports is through *sockets*. Sockets use a client-server model to allow tasks to communicate over a network using ports. We will discuss this next.

12.2.8 Sockets

When information is to passed across a communications network, the sender and receiver are on separate systems and need to communicate through the network ports. The term *socket* describes the Berkeley Software Distribution (BSD) of Unix approach of creating a connection between two tasks generally on separate systems wishing to communicate through specific ports. In Linux, all communications are treated as if they are files, and communications through a network are no different, thus network communications via sockets can be treated the same as reading and writing to a file. A socket is identified by an IP address together with a port number. The following are examples of source code that initiates a socket server and a client that communicates with that server. This is adapted from www.linuxhowtos.org/C_C++/socket.htm.

```c
/* socket_server.c */
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>


int main() {
    /* 1. Open socket retrieving a socket file descriptor */
    int sockfd = socket( AF_INET6, SOCK_STREAM, 0 );

    if ( sockfd < 0 ) {
        fprintf( stderr, "Error opening socket\n" );
        return EXIT_FAILURE;
    }

    /* 2. Bind socket */
    struct sockaddr_in server_addr = {AF_INET6, htons(1101), 0};
    inet_pton( "127.0.0.1", &server_addr.sin_addr );

    if( bind( sockfd,
              (struct sockaddr *)&server_addr,
              sizeof( server_addr ) ) < 0 ) {
        fprintf( stderr, "Error binding to port\n" );
        return EXIT_FAILURE;
    }

    /* 3. Listen for a connection */
    listen( sockfd, 1 );
    struct sockaddr_in client_addr;
    socklen_t client_len = sizeof( client_addr );

    /* 4. Accept a connection */
    int new_sockfd = accept( sockfd,
                             (struct sockaddr *)&client_addr,
                              &client_len );

    if ( new_sockfd < 0 ) {
        fprintf( stderr, "Error accepting connection\n" );
        return EXIT_FAILURE;
    }

    /* 5. Receive and send messages */
    uint8_t request[256];
    int n = read( new_sockfd, request, sizeof( request ) );

    if ( n < 0 ) {
        fprintf( stderr, "Error reading message\n" );
        return EXIT_FAILURE;
    }

    printf( "server received: %s\n", request );
    char response[] = "response";
    n = write( new_sockfd, response, sizeof( response ) );

    if ( n < 0 ) {
        fprintf( stderr, "Error writing message\n" );
        return EXIT_FAILURE;
    }


    return EXIT_SUCCESS;
}
```

```c
/* socket_client.c */
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

int main() {
    /* 1. Open socket retrieving a socket file descriptor */
    int sockfd = socket( AF_INET6, SOCK_STREAM, 0 );

    if ( sockfd < 0 ) {
        fprintf( stderr, "Error opening socket\n" );
        return EXIT_FAILURE;
    }

    /* 2. Request connection */
    struct hostent *he = gethostbyname( "localhost" );

    if ( he == NULL ) {
        printf( "Hostname not found\n" );
        return EXIT_FAILURE;
    }

    struct sockaddr_in server_addr = {AF_INET6, htons(1101), 0};
    memcpy( &server_addr.sin_addr, he->h_addr_list[0],
            he->h_length );

    if ( connect( sockfd,
                  (struct sockaddr *)&server_addr,
                   sizeof( server_addr ) ) < 0 ) {
        fprintf( stderr, "Error requesting connection\n" );
        return EXIT_FAILURE;
    }

    /* 3. Send and receive messages */
    char request[] = "request";
    int n = write( sockfd, request, sizeof( request ) );

    if ( n < 0 ) {
        printf( "Error writing message\n" );
        return EXIT_FAILURE;
    }

    uint8_t response[256];
    n = read( sockfd, response, sizeof( response ) );


    if ( n < 0 ) {
        fprintf( stderr, "Error reading message\n" );
        return EXIT_FAILURE;
    }

    printf( "client received: %s\n", response );
    return EXIT_SUCCESS;
}
```

Figure 12-1 is a screen shot from Wireshark, a free and open-source packet analyzer. Here you can see it recording packets sent to port 1101, as the above code does. The highlighted line is the actual message shown below with the null-terminated string `"request"`. The first three establish the connection, while the balance are acknowledgements and closing the connection.

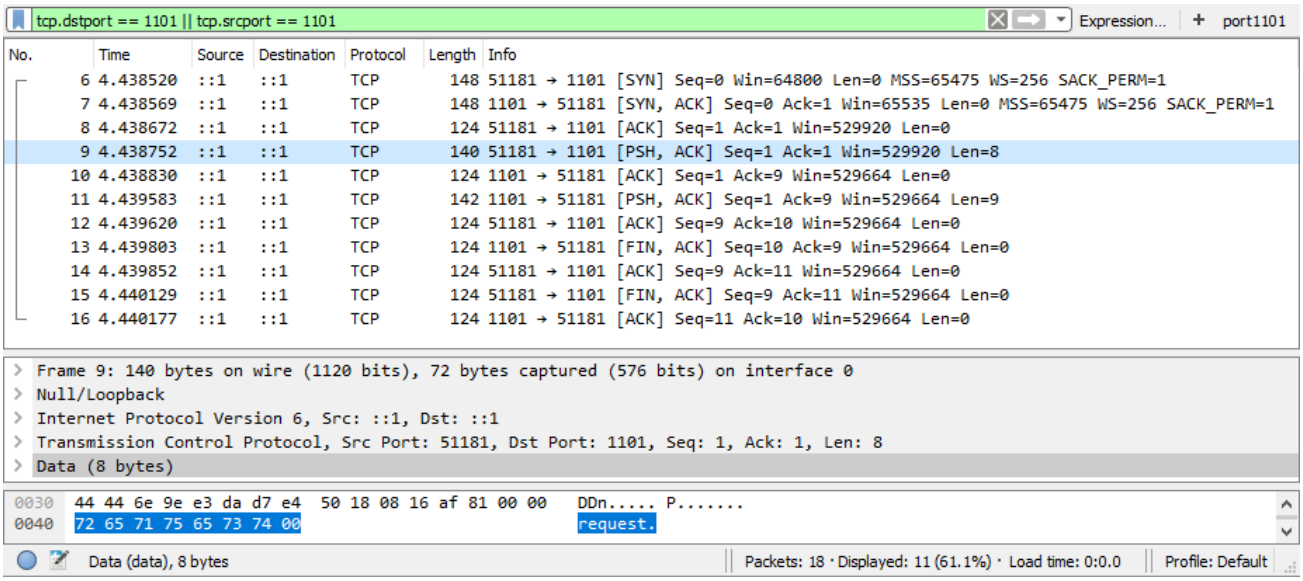


Figure 12-1. A screenshot from Wireshark packet analyzer
following the packets sent between the socket server and client.

12.2.9 Internet communications

In hard real-time systems, a communication system such as the Internet will never be used; however, in firm or soft real-time systems, it is still acceptable. Consequently, we will briefly describe the handling of messages through a network.

With Canada Post, there are two protocols[32] for sending letters:

1. regular mail (unregistered), and
2. registered mail.

In each case, the message is wrapped an appropriate envelope with the necessary information printed in the appropriate locations. The latter protocol provides a proof of delivery by securing a signature upon delivery, while the former does not. The information provided is also listed in reverse order:

Receiver's name
Address
Postal Code

In sending this letter, the Post Office is only interested in the Postal Code: this can be used to get the letter to the correct distribution point, usually within 100 m of the actual address at least within cities. Only once it reaches this distribution point is the address used to deliver the letter to the correct house or apartment, and once it is received, the name ensures it is received by the correct recipient.

[32] From the OED: A (usually standardized) set of rules governing the exchange of data between given devices, or the transmission of data via a given communications channel:

[1c] Each of the official formulas used at the beginning and end of a charter, papal bull, or other similar document.

[6d] A (usually standardized) set of rules governing the exchange of data between given devices, or the transmission of data via a given communications channel.

We will see that network communications uses a similar approach:

12.2.9.1 Application data

Communications may be between two applications, or perhaps between two users using applications. In either case, the sending application generates a binary message, and then is usually required to divide that message into blocks of an appropriate sizes based on the protocol that will be used to send that message. If the message is divided into two or more message blocks, the receiving application will have to piece those blocks back together again at the end to recreate the sent message. We will call these message blocks as *application data*. We will look now at how such data can be sent.

12.2.9.2 Message protocols

Similarly, network communication has three primary approaches (or *protocols*) to sending messages:

1. Transport Control Protocol (TCP),
2. User Datagram Protocol (UDP),
3. Real-time Transport Protocol (RTP).

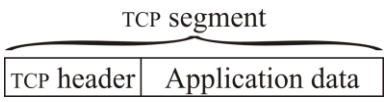
There are others, such as Internet Control Message Protocol (ICMP) which is used by network devices to send error messages and other information; however, we will only look at the three we've listed.

12.2.9.2.1 Transport Control Protocol (TCP)

The first protocol, TCP, is similar to registered mail. The application data is prefixed by a header to create a *TCP segment*. The header includes

1. the source port (2 bytes),
2. the destination port (2 bytes),
3. a sequence number (4 bytes) used to specify the relative position of this message block relative to others,
4. an acknowledgement number (4 bytes) to be sent back to verify that this message block was received,
5. a number of flags specifying various properties,
6. a checksum that is simply a 1s complement addition of the message plus a pseudo-header divided into 2-byte blocks,
7. the length of the message, and
8. a few other fields.

All this is used to allow receiver to correctly check, extract and order the application data to recreate the original message. An acknowledgement is returned to the sender, and the sender will resend a message if no acknowledgement is received.



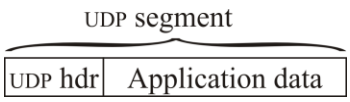
When sockets communicate using TCP, the socket software ensures that the data is presented to both the client and server in the order they were sent. Thus, any applications using sockets are unaware that their messages may be divided and reassembled as they are sent. In all cases, TCP uses acknowledgments when a segment is sent, so if for example a client uses TCP to send data and a server has not been established at the destination port, an error will occur. When acknowledging a TCP segment, the source and destination ports will be reversed.

12.2.9.2.2 User Datagram Protocol (UDP)

The second protocol, UDP, is more similar to regular mail. Like TCP, UDP appends an 8-byte header to the application data to create a UDP segment. This header contains only

1. the source port (2 bytes),
2. the destination port (2 bytes),
3. the total length (2 bytes allowing application data of length up to 65507), and
4. a checksum as used in TCP.

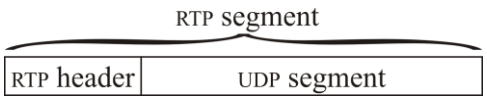
There is no acknowledgement or order, so the sender cannot confirm that the segment was received.



If an application uses UDP to send a message, and a server has not yet been established on the destination port, the message will be discarded and lost, although no error will occur. Because there is no confirmation, the source port is optional and may simply be designated as `0x0000`.

12.2.9.2.3 Real-time Transport Protocol (RTP)

The third protocol, RTP, is actually built on top of UDP. Without getting into the details of the header, it allows for additional information not available in UDP such as sequencing data without requiring the acknowledgements associated with the TCP transport protocol.



The receiver can use the sequencing information to re-establish the order of the information, but some mechanism must be in place to deal with lost segments. Thus, this protocol is used for such real-time systems that include streaming media. Thus, if a segment is lost, the receiver can extrapolate previously received data to fill in the gap. As more segments are lost, the quality of service begins to decrease. Streaming media could be described as a firm real-time system: if a segment is lost or is late, it is of no value, and the loss of that segment will result in a decrease in the quality of service.

12.2.9.3 Network protocols

You will note that both segments only include the source and destination ports—they do not contain the addresses of the destination of the segment. This is because the segments contain information necessary to direct the message once the message is received at the destination system. A different protocol will be used to send the segment to its appropriate destination, and the most common protocol is the Internet Protocol (IP). We will only describe IPv4, however, the reader is welcome to look at others including IPv6.

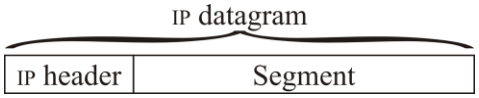
When received, a TCP or UDP segment is prefixed by a IPv4 header. This header includes information such as

1. four bits for the version, in this case `0100`,
2. the length of the header,
3. six bits to allow for differentiated services so that soft or firm real-time systems can get priority over other messages (e.g., Voice over IP or VoIP),
4. total length of the entire packet,
5. identification, flags, offsets, time-to-live,
6. a transport protocol identifier (`0x06` for TCP and `0x11` for UDP),
7. a checksum,
8. the source IPv4 address (4 bytes),

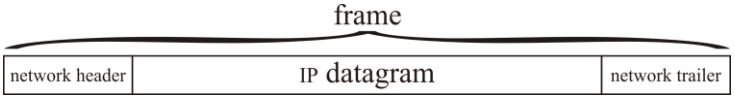
9. the destination IPv4 address (4 bytes), and
10. other optional fields, the length of which is defined by Point 2.

Domain names such as `www.uwaterloo.ca` must be translated to a 4-byte address through a Domain Name System (DNS) server. The Internet Corporation for Assigned Names and Numbers (ICANN) oversees global IP address allocation and manages the top-level DNS servers.

This creates an IP datagram.



This datagram is then sent through whichever hardware or virtual network the computer is connected to via a router. The router itself will include additional headers and trailers used in the transmission of the data through the network creating a *frame*:



If one network is connected to another via a router, the router will strip off the previous network header and trailer, replacing them with new headers and trailers creating an appropriate frame for the connected network.

12.2.9.4 Hardware

Depending on the hardware sending IP datagram, the datagram will itself be wrapped with additional data.

12.2.10 Summary of solutions for communication

In this section, we looked at various solutions for the many classifications of communication.

12.3 Priorities of messages

Any time that a message is stored in a mailbox or other messaging queue, one could use a first-in—first-out approach, but it is also possible to assign each message a priority, and messages are delivered in order of priority.

12.4 Synchronization

While the passage of information is often associated with communications, it has an important alternate use: synchronization. If two tasks do not share the same operating system, it is much more difficult to synchronize events, as semaphores are not necessarily available.

Recall that previously, we discussed how the wait and post commands issued to semaphores have the same behaviour as asynchronous-send and synchronous-receive messaging. Could we not, therefore, use such message passing in place of semaphores? The contents of the messages need not contain anything, beyond, perhaps, the addressee.

12.4.1 Serialization between two tasks or threads

In order to achieve this, some precautions must be taken: if a task or thread is waiting on a mailbox, then any message sent to that mailbox will unblock that task or thread. Consequently, we would really require that a separate mailbox be used for synchronization.

Now, in order to achieve serialization, Task A would send a message to Task B. Task B could not continue executing until it has received the message from Task A.

12.4.2 Mutual exclusion

In order to achieve mutual exclusion, however, is much more difficult: the two tasks are remote, and therefore neither has access to the same memory location, and thus, we cannot create a binary semaphore that both tasks can decrement. Additionally, one task cannot send a message to the other remote task expecting permission to continue executing, as the other task may not be waiting for such messages.

In order to achieve mutual exclusion, we will require a more subtle approach: we will require a third task that acts as a semaphore. This *semaphore* task will receive two types of messages:

1. a request to wait, and
2. a directive to post.

When any other task posts a message to request a wait, that task immediately issues a wait on the mailbox, thus being blocked until a reply is received.

We will describe a task acting as a binary semaphore: the task will wait on a mailbox, and its internal state (stored in a local variable—no protection is required) is either 0 or 1.

If the state is 0, and

1. it receives a request to wait, the semaphore task records that fact and does nothing, and
2. if it receives a post,
    a. if there is a record of a task waiting for a message, it sends that task a response (thereby allowing it to continue), otherwise,
    b. it increments the state to 1.

If the state is 1, and

1. it receives a request to wait, the state is decremented to 0 and a reply is sent to the requesting task, and
2. if it receives a post, nothing is done.

In either case, as soon as the semaphore task has responded, it issues another wait on its the mailbox so that it is ready to respond to the next request.

This semaphore task must run on a single processor, and all other tasks wanting to use this semaphore are required to send messages to this task. In order to ensure mutual exclusion, the messages could also contain identifiers of the tasks that are issuing the waits and posts, in which case, it would only allow the task that issued a request to wait to issue the corresponding post.

12.4.3 Counting semaphores

The generalization of a binary semaphore to a counting semaphore is quite straight-forward and is left as an exercise to the reader.

12.4.4 Priority inversion

As before, we may also have a situation where a low priority task is holding a semaphore as described above, but this is followed by another high priority task also sending a request to wait on that same semaphore task. Even though the tasks may be remote, they may still share, for example, a communication network. Consequently, we could have the same situation that arose in the Mars Pathfinder mission: an intermediate priority task prevents a low priority task from releasing a resource required by a high priority task.

Unfortunately, this is not that simple: the semaphore task, having received a message to request to wait from a higher priority task cannot simply send the lower priority task a message: it isn't waiting for a message to arrive.

Instead, we will require semaphore clients executing on each remote system, and those semaphore clients will communicate with one that has been designated as the server. Now, any task sending a message to either request a wait or to post to the semaphore would therefore have to communicate with the local semaphore client, and if that client is not the designated server, it would in turn forward the message to the semaphore server. If the semaphore server determines that the priority of a task must change, the semaphore server can make that change if it is local or send a message to corresponding semaphore client local to the task that must have its priority changed.

Thus, our situation would look as shown in Figure 12-2.

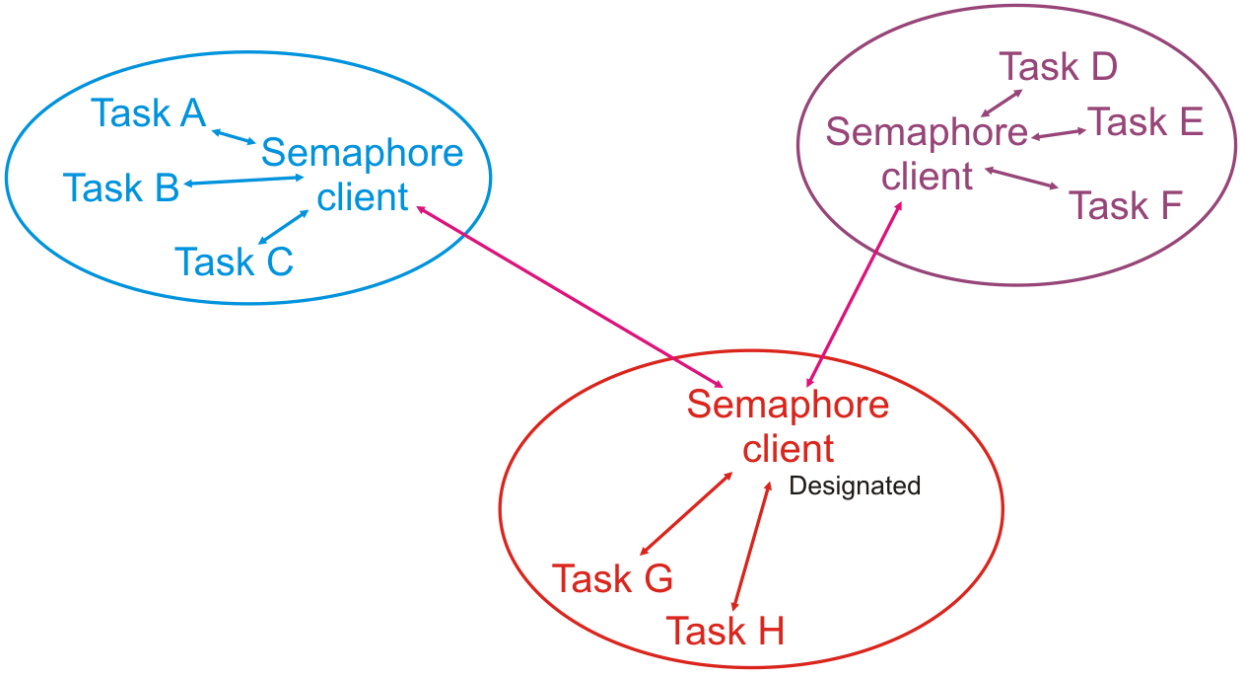


Figure 12-2. Local semaphore clients communicating with a designated semaphore server.

12.4.5 Summary of synchronization

In this topic, we looked at how it is possible to use messaging in place of semaphores for synchronization. While serialization is simple enough to achieve, to achieve equivalent synchronization power of semaphores, it is essentially necessary to create a single semaphore task with which all other tasks or threads must communicate. This task mediates the access to the semaphore. If additional control is required, so as to solve the problem of priority inversion, each local system must have its own semaphore client, and all communication with the semaphore server is through these local clients.

12.5 Coordination through election algorithms

Suppose you have *n* tasks, of which at any time *m* of those tasks may have to coordinate to complete a specific goal. Under certain circumstances, it is often better when one of the *m* tasks is designated the coordinator of the other $m - 1$ tasks attempting to accomplish the goal. This is usually done through some form of *election*, whereby tasks eligible to participate communicate amongst each other and elect or designate a coordinator.

12.5.1 The bully algorithm

In the bully algorithm, each task is given a unique and linearly ordered priority. Determining a coordinator is through a series of *elections* where the highest priority task that is available is always the coordinator.

1. If, at any time, Task A determines it is necessary to elect a coordinator for a goal, be it either that
    a. progress towards the goal has not yet begun, or
    b. it has become available to work towards that goal, perhaps having recovered from a failure, or
    c. it determines that the current coordinator is unavailable,

    then Task A proceeds with *calling an election* where it broadcasts an *election message* to all other tasks with higher priority where all available higher tasks are expected to respond with an *acknowledgment*, where
    d. if no other higher priority task acknowledges, Task A determines it *won* the election and broadcasts a *victory message*; and
    e. otherwise, a higher priority task does acknowledge, so Task A waits for a victory message and
        i. if, after an appropriate waiting period, another task issues a victory message, that other task is recognized as the coordinator,
        ii. otherwise, Task A calls another election.
2. If Task A ever receives an election message from a lower priority task, it responds with an acknowledgment and, if it is not already in the process of calling an election itself, it proceeds by calling an election.
3. If Task A ever receives a victory message from a higher priority task, it recognizes that task as the coordinator for the goal.
4. If Task A ever receives a victory message from a lower priority task,
    a. if it is working towards the same goal, it initiates a new election,
    b. otherwise, it records that that lower-priority task is the current coordinator of the goal in question.

This algorithm assumes that most of the messages are successfully communicated to all possible tasks. This algorithm could become expensive if turnaround of coordinators is frequent.

For example, suppose we have eight tasks, Task A (with the highest priority) through Task H (with the lowest priority). Initially, Task D becomes ready to accomplish a specific goal, so it broadcasts an election. Tasks A though C are not available to work toward this goal, so they do not acknowledge, and thus D broadcasts a victory message, as shown in Figure 12-3.

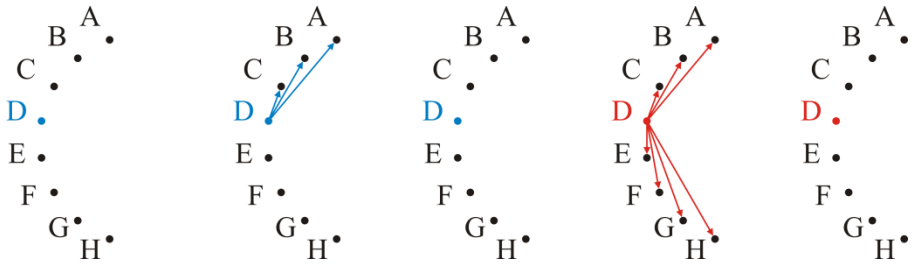


Figure 12-3. Task D calling an election, receiving no acknowledgment and thus sending a victory message.

Next, suppose Task F becomes ready to accomplish the same goal. It calls an election, but Task D acknowledges, which puts Task F into a waiting state. Task D then calls an election, and with no response, it sends out a victory message, and thus Task F is aware that Task D is coordinating it toward the goal. This is shown in Figure 12-4.

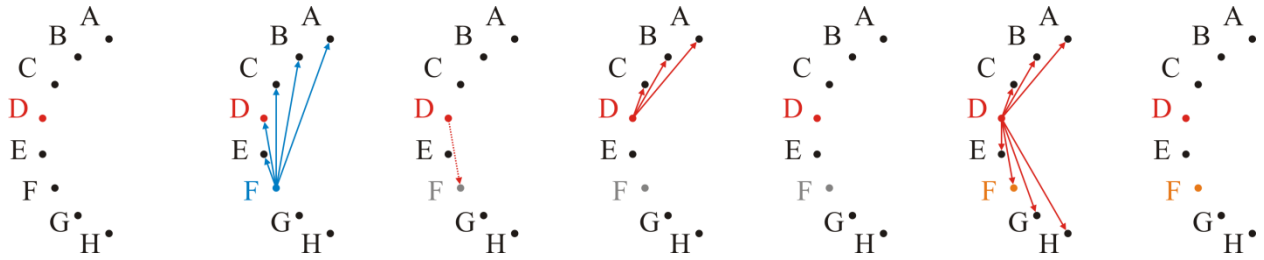


Figure 12-4. Task F calls an election, forcing Task D to call an election while acknowledging Task F. Task D receives no acknowledgment from its election, so it broadcasts a victory notice.

Next, if Task G wishes to work toward the goal, it calls an election, and is acknowledged by both Tasks D and F. Both Tasks D and F call an election, and Task D acknowledges Task F while waiting for its election results. Upon not receiving a response, Task D again broadcasts a victory message, as shown in Figure 12-5.

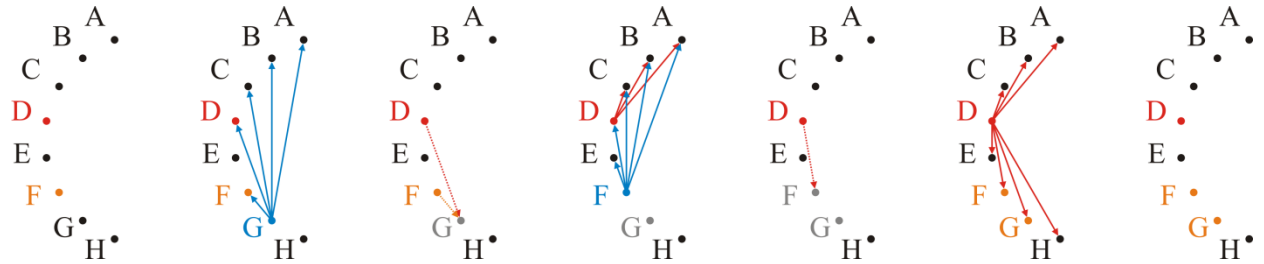


Figure 12-5. Task G calls an election, forcing both Tasks D and F to call elections while acknowledging Task G. Subsequently, Task F receives an acknowledgment from Task D, but Task D receives no acknowledgement from its election, so it broadcasts a victory notice.

Finally, Task B becomes ready to work toward the same goal. It calls an election, gets no acknowledgement and thus broadcasts a victory message, as shown in Figure 12-6.

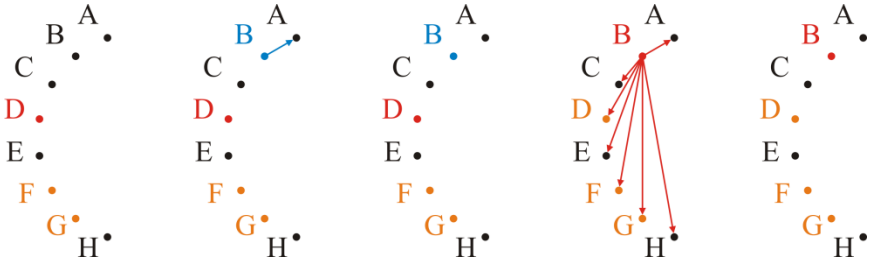


Figure 12-6. Task B calls an election and receives no acknowledgment, so it broadcasts a victory message.

This is the most basic algorithm, but it is possible to simplify the algorithm: for example, if a task is already designated the coordinator and it receives an election call (necessarily from a lower priority task), it could simply reply with an acknowledgment and broadcast a victory message.

The number of messages sent during an election (including election messages and acknowledgements) is $O(n^2)$, the worst-case scenario occurring when $\Theta(n)$ tasks are participating in achieving the goal and a low-priority task initiates the election. The least number of election messages sent is $\Theta(1)$ when a high priority task initiates an election. In all cases, $\Theta(n)$ victory messages are sent.

12.5.2 Chang-and-Roberts ring algorithm

One issue with the bully algorithm is the significant number of messages that may be sent if a large number of tasks are currently working towards a goal. The Chang-and-Roberts ring algorithm reduces this by giving each task a priority, but also ordering the tasks into a ring. Now, when an election is called under the same circumstances as listed in the bully algorithm, the task sends an election message along the ring. The election message contains an identifier for the task initiating the election and that task proposes that it is the coordinator. Each time a message is received by a task participating in achieving the goal, if its priority his higher than the task currently proposed as the coordinator, it replaces it with its own identifier. The message is then modified to the next task in the cycle. If a task within the ring is off line, it is skipped in this cycle. When the election message has cycled around the ring, the task calling the election sends a victory message with the coordinator identified along the ring.

For example, given the same eight tasks as in the example in Section 12.5.1, they may be arranged in a circle as shown in Figure 12-7. Note that the ordering of the cycle need not match the order of the priority. We will assume that Task H is not available, as opposed to not participating.

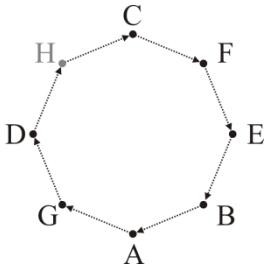


Figure 12-7. The ordering of Tasks A through H in a ring.

We will proceed in the same manner as our previous example: Task D first begins to participate in achieving a particular goal, followed by Tasks F, G and then Task B. When Task D begins participation, it sends out an election message that is not modified by any other tasks. When the message is initially sent to Task H, a time-out may indicate Task H is unavailable, and thus the message will be forwarded to Task C, the next task in the cycle. This is shown in Figure 12-8. When the message is returned to Task D, it sends a victory message through the cycle identifying itself as the coordinator.

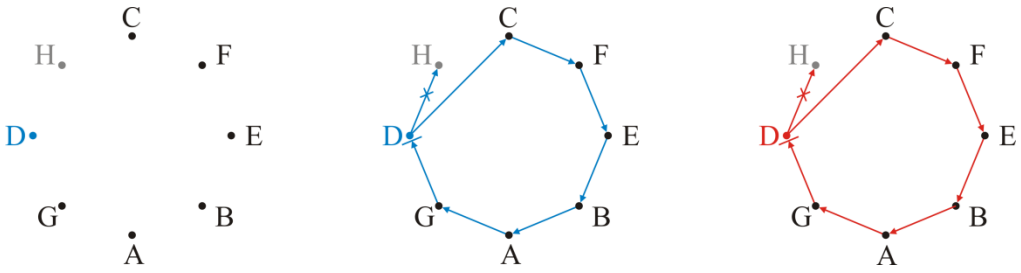


Figure 12-8. Task D initiates an election and sends an election message that is not modified—there are no other participants. Once the election message cycles through, Task D sends a victory message identifying itself as the coordinator.

Suppose Task F now begins participation towards the same goal. It sends out an election message identifying itself as a candidate for the coordinator, and when this message is received by Task D, it replaces the candidate with itself. The message continues through the cycle until it is, once again, received by Task F, which then sends out a victory message identifying Task D as the coordinator. This is shown in Figure 12-9.

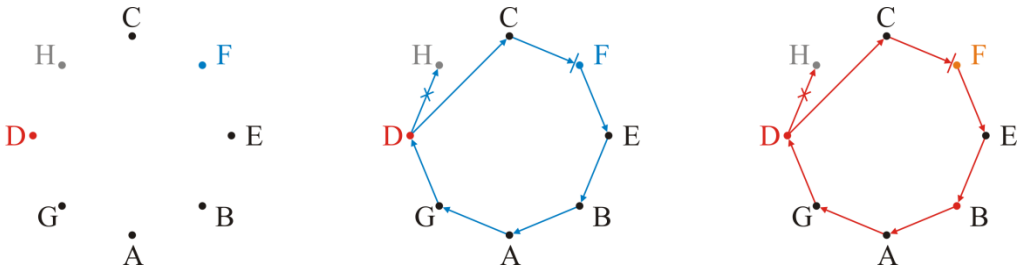


Figure 12-9. Task F begins participating, so it sends an election message identifying itself as a candidate for the coordinator. When the message reaches Task D, it updates the message to identify itself as the coordinator. The message is ultimately received by Task F which then sends a victory message identifying Task D as the coordinator.

Now suppose Task G begins participation: it sends an election message to Task D identifying itself as a candidate for the coordinator, a field that is immediately updated by the higher priority Task D. The message continues through the cycle, and Task F leaves the candidate unchanged. Task G then sends a victory message, again, identifying Task D as the coordinator. This is shown in Figure 12-10.

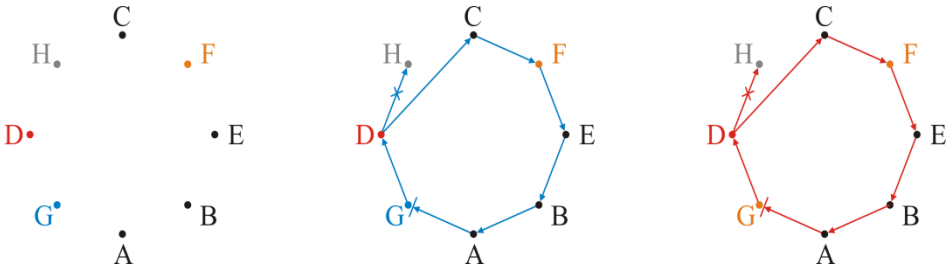


Figure 12-10. Task G begins participation, and thus sends an election message. Task D modifies the message to identify itself as the coordinator, and Task F leaves it unchanged. Task G then sends a victory message identifying Task D as the coordinator.

Finally, if Task B begins participation, it sends an election message placing itself as a candidate for the coordinator. The message is passed through the cycle unchanged, as Tasks G, D and F—the only ones participating—all have lower priority. Task B then sends out a victory message identifying itself as the coordinator. This is shown in Figure 12-11.

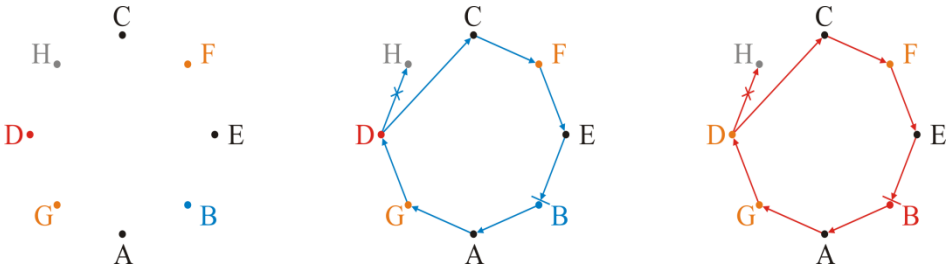


Figure 12-11. Task B begins participation, so it sends an election message identifying itself as a candidate for the coordinator. The message is passed around the cycle and neither Tasks D, F or G modify it, so when it is received again by Task B, Task B sends a victory message identifying itself as the coordinator.

The number of election messages sent is always $\Theta(n)$, which makes it better than the bully algorithm in the worst case, and equivalent to it in the best case.

12.5.3 Summary of coordination through election algorithms

In a situation where a subset of tasks may participate in achieving a particular goal, but where one task is required to coordinate the participation, it is common to initiate an election to determine which tasks is available to coordinate. Both the bully algorithm and the Chang-and-Roberts ring algorithm hold elections through message passing, and each time a new task becomes available, a new election is held. With the bully algorithm, the best case number of election messages sent is $\Theta(1)$ while the worst case is $\Theta(n^2)$; however, this is mitigated by the requirement that $\Theta(n)$ victory messages must be sent. In the case of the Chang-and-Roberts ring algorithm, each time an election is held, $\Theta(n)$ election and victory messages must be sent.

12.6 When a message is sent over Network communications

When a message is sent over a network—that is, an intermediate communication channel—one expects a delay in the transmission.[33] That delay is due to numerous factors, including

1. formatting the messaging,
2. waiting for access to the communication channel (queuing delay),
3. transmitting the message,
4. notifying the receiver if we are using an asynchronous-receive, and
5. deformatting the message.

The queuing delay depends on the protocol used to administer the communication channel, which include approaches such as waiting for

1. a time slot during which to send a message (TTP/C and FlexRay),
2. a transmission token (Token Ring),
3. a contention-free transmission (Ethernet),
4. network priority negotiation (CAN), and
5. removal from a priority queue (Switched Ethernet).

These issues arise from the specific protocol used, and each protocol has its own advantages and disadvantages. These protocols can be classified into three general categories based on the communications being

1. token based,
2. collision based, and
3. contention free.

We will consider each of these and look at examples

12.6.1 Token-based communications

We have already discussed the use of tokens in synchronization: a token is passed between those tasks requiring the network, and each task can use the network only while it holds the token.

12.6.2 Collision-based communications

The most common protocols (Ethernet and CAN) are collision-based, that is, it is entirely possible that two tasks will attempt to use the communication channel simultaneously and therefore some steps must be taken to correct this situation. With Ethernet, each sender steps back and waits a random amount of time before trying again. Unfortunately, therefore, there is no upper bound as to how long the queuing delay will be. CAN (Controller Area Network) uses a redundant bus topology where each task that is attempting to send messages is also listening to the bus.

[33] This section is based on a presentation of the material by Jan Jonsson of the Chalmers University of Technology for his course EDA222, *Real-Time Systems*.

Prior to CAN, an electronic control unit (ECU) would have point-to-point connections with the various components (transmission, ABS, fuel injection, tire pressure sensors, power windows, power steering, lights, air-condition and heating, locks, etc.) it needs to communicate with. Additionally, various components had to communicate with each other, introducing further point-to-point connections. This is shown in Figure 12-12.

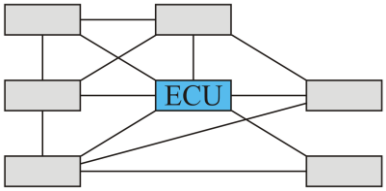


Figure 12-12. An ECU with point-to-point connections.

In 1985, Bosch introduced a single bus to which each component could connect, and each component could communicate with each other component, as is shown in Figure 12-13. While it was originally developed for the automotive industry, it has now been extended to use in industrial automation, aerospace, medical, and most other communication- and information-based industries.

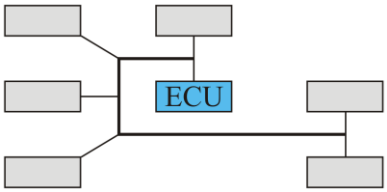


Figure 12-13. An ECU and all other components connected to a CAN bus.

Up to 112 nodes can be connected to this bus. This network is a high-integrity serial-broadcast bus that has speeds up to 1 Mbit/s. With messages being approximately 100 bits in length, this allows on the order of 10 thousand messages per second, a rate that is generally suitable for real-time systems that involve interactions on the macroscopic level. The bus is composed of two dedicated wires: CAN high and CAN low, as is shown in Figure 12-14.

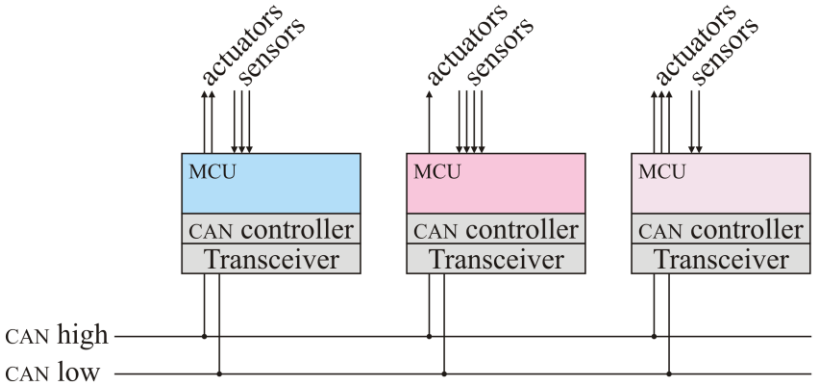


Figure 12-14. Connections of each component to two wires: CAN high and CAN low.

In idle each wire carries 2.5 V which signals a “1”. When any device wants to transmit a “0”, it sets the CAN high wire to 3.75 V and the CAN low wire to 1.25 V, there now being a difference of 2.5 V, as shown in Figure 12-15.

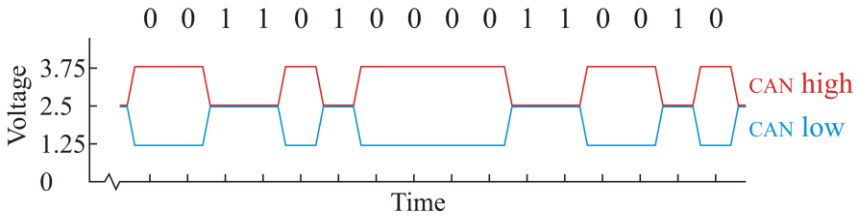


Figure 12-15. Transmission of 0011010000110010 on a CAN bus.

However, if one device attempts to signal a “1” and another signals a “0” simultaneously, the wires carry a “1”. Consequently, a device signalling a “0” while detecting a “1” can become aware that another device is simultaneously attempting to use the bus.

With CAN, each sender starts by sending a message header beginning with a start-of-message bit followed by an identifier that allows collision-detection mechanism to determine which sender will be allowed to proceed. All other devices connected to the bus will listen for a start-of-frame (SOF) bit (a `0`) and will not ever try to transmit until after it has detected an end-of-frame (EOF) sequence (`1111111`). There must be at least three inter-frame bits following any message when no device is sending (`111`). There are four types of frames:

1. message,
2. remote,
3. error, and
4. overload.

The format of a short CAN message frame allows the transmission of 0 to 8 bytes, as shown in Figure 12-16.

Figure 12-16. Format of a short CAN message frame.

The 11-bit identifier is now often used to specify the priority of the message, where lower numbers indicate higher priorities. This is used to quickly determine access to the bus in case of a collision: each task is continually monitoring the communication channel, and each is sending a signal of `1`. The value seen by all tasks is the logical OR of all signals that are being transmitted. As soon as any one task wants to send a message, it signals a `1`, and therefore all tasks see `1`: this indicates to all tasks that a message is being sent.

If two or more tasks begin sending a message simultaneously (in the same cycle), they will all see that the channel now holds a value of `0`. They now begin to transmit their identifier. Because lower values indicating a higher priority, the first bit that differs between the two priorities can be used to differentiate the two: the lower priority task will have a `1` while the higher priority task will have a `0`. The five pairs of priorities shown in Figure 12-17 demonstrate this where, in each case, the first row contains the higher priority value and the second row contains a lower priority value. You will note that the first bit that differs is all that is necessary to determine which task has higher priority.

```
00000000011   00000101010   00001000110   00110010011   00010011100 0
00000010100   00001011011   00001101011   00110010111   00010011100 1
```

Figure 12-17. Five pairs of priorities.

As each task is sending its priority, each task is also tracking the value on the communication channel and because it is a logical AND of all values, the first time the bits differ, the higher priority task will see the bit it signaled—a `0`—whereas the lower priority task will see a `0` when it signaled a `1`. Consequently, the lower priority task will see a different value from what it is transmitting and it will therefore cease the transmission.

To demonstrate this, consider Figure 12-18. Suppose Senders A and B both signal they are going to send a message. Each signals a 0, and now the channel has a value of 0, indicating to all other potential senders that a message is being sent. The next 11 bits of both Sender A and B are their priorities. Sender A has priority 42 (= 00000101010_2) which is higher than that of Sender B with priority 91 (= 00001011011_2). The first four bits sent are both `0`, and each reads `0` on the channel. With the 5th bit, however, Sender B sends a `1` but Sender A sends a `0`. The AND of all signals is `0`, and Sender B recognizes that 0 differs from the signal it sent, so it must have lower priority—it immediately stops sending. Sender A continues sending `1`, `0`, `1`, `0`, `1`, `0`, and the channel always contains the value it sent, so it knows it is okay to send, so it continues sending the control bits and then the message.

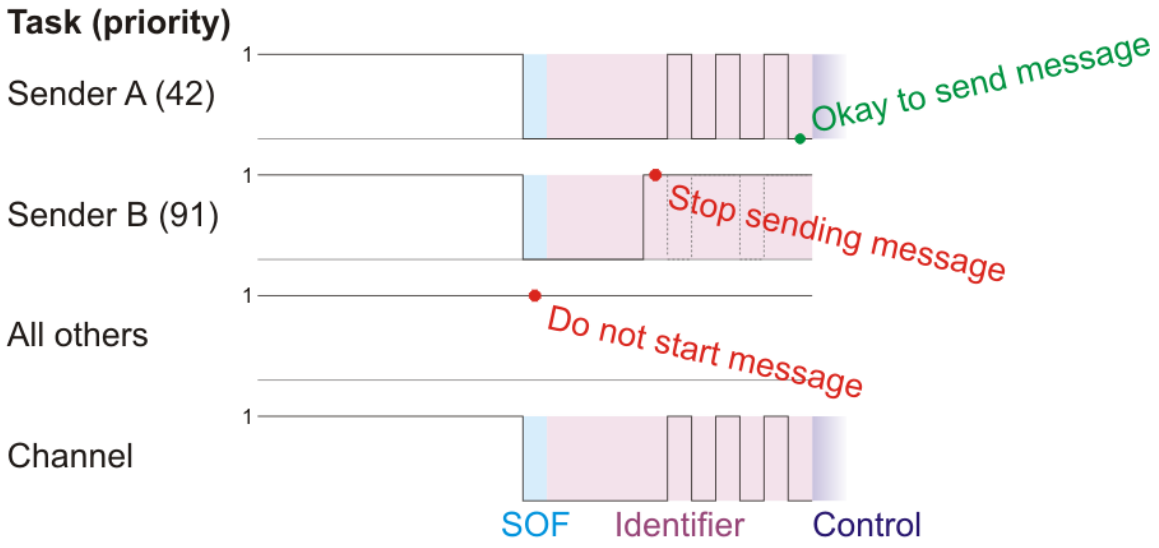


Figure 12-18. Priorities with CAN message frames.

If three or more senders attempted to send a message simultaneously, then one by one, all the lower priority messages would be rejected. Note that this requires a unique identifier for each sender. A high priority message can now be sent after a queuing delay limited to having to wait for any current message that is being sent.

The error control uses a 15-bit CRC using the polynomial $x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$, as described previously in Section 13.2.3, which is followed by a delimiter (`1`). The next bit is not signaled by the sender, but rather the receiver, which acknowledges receiving the message by signaling a (`0`).

The end-of-frame sequence is seven ones (`1111111`). To avoid any other sequence of seven bits being falsely recognized as an end-of-frame sequence, CAN uses a technique called *bit stuffing*. If five bits are the same, either `00000` or `11111`, the next bit is immediately set to the opposite value. This next bit is ignored by both the sender and receiver.

The Keil data structure for a CAN message frame is defined in `RTX_can.h`:

```
/* CAN message object structure                                            */
typedef struct  {
    U32 id;                 /* 29 bit identifier                           */
    U8  data[8];            /* Data field                                  */
    U8  len;                /* Length of data field in bytes               */
    U8  ch;                 /* Object channel                              */
    U8  format;             /* 0 - STANDARD, 1- EXTENDED IDENTIFIER        */
    U8  type;               /* 0 - DATA FRAME, 1 - REMOTE FRAME            */
} CAN_msg;
```

To initialize one of the CAN controller,

```
#include <RTL.h>
#include "CAN_Cfg.h"

CAN_ERROR CAN_init     ( U32 ctrl, U32 baudrate );
CAN_ERROR CAN_start    ( U32 ctrl);
CAN_ERROR CAN_send     ( U32 ctrl, CAN_msg *msg, U16 timeout );
CAN_ERROR CAN_request  ( U32 ctrl, CAN_msg *msg, U16 timeout );
CAN_ERROR CAN_set      ( U32 ctrl, CAN_msg *msg, U16 timeout );
CAN_ERROR CAN_receive  ( U32 ctrl, CAN_msg *msg, U16 timeout );
CAN_ERROR CAN_rx_object( U32 ctrl, U32 ch, U32 id, CAN_FORMAT filter_type );
CAN_ERROR CAN_tx_object( U32 ctrl, U32 ch, CAN_FORMAT filter_type );
```

The `ctrl` parameter is either 1 or 2, indicating which CAN controller is being signaled. The baud rate is up to 2^{20}.

12.6.3 Contention-free communications

There are two approaches to contention-free communications, including

1. dedicated lines, and
2. time slots.

The first is obvious and significantly more expensive. We will therefore consider the second. Each task is allocated a number of time slots to communicate, whether or not it needs to. This will guarantee bounded messaging queuing delays, but it may also reduce network usage. Examples of protocols that use this include TTCAN, TTP/C and FlexRay. We will briefly describe TTCAN.

Time-triggered CAN is based on CAN and also uses a redundant bus topology. Synchronization is achieved through the transmission of a *time master's* reference message. The period between two reference messages is the *basic cycle*. The basic cycle is broken into a number of windows during which messages can be sent, including

1. exclusive,
2. arbitrating, and
3. free

windows. The first type of window is dedicated to specific tasks, the second allow for spontaneous messages using normal CAN arbitration, and the last are for planned future expansions. An example is shown in Figure 12-19.

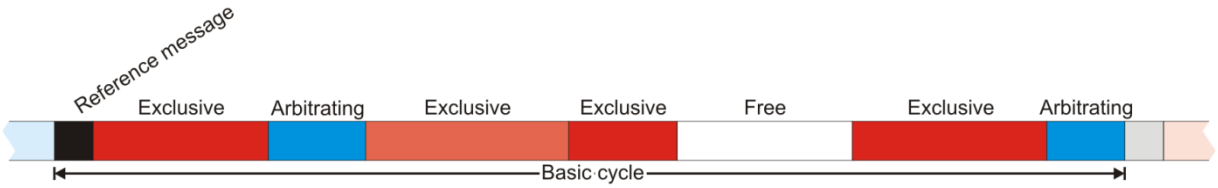


Figure 12-19. A basic cycle of TTCAN broken into seven windows.

TTCAN is used in automotive systems and allows a maximum transmission of 1 Mbit/s. For further details to TTCAN, see *Time Triggered Communication on CAN* by Thomas Führer et al.

12.6.4 Summary of network communications

Processes communicating over a network will inevitably result in some form of queuing delay as the messages are waiting to be sent. Contention-free communications require that the time be divided into windows that are allocated to specific processes. There are windows available for spontaneous messages, but these are only at specific times. Token-based communications are more rare today, but collision-based communications are more popular with Ethernet and CAN. The latter is specifically designed to allow two senders to determine while they are sending which conflicting message is to be sent. Ethernet does not have this option, and therefore there is no upper bound on the potential queuing delay.

12.7 Summary of inter-process communication

In this topic, we have considered different mechanisms for inter-process communication, where threads and tasks may not share memory due to either protections enforced by an operating system locally or because the threads and tasks are executing on remote systems. We classified the various means of passing information, and then considered implementations that solved the message passing problem under some of the conditions we considered. We discussed the problem of priorities and messages, and then we saw how message passing can be used to achieve synchronization. In one sense, it is as straight-forward as semaphores for simple problems such as serialization, but if we require the full power of semaphores, together with solving the problem of priority inversion through priority promotion, this requires a more complex system of semaphore clients with a single semaphore server. We concluded by looking at network communications. The appendix contains an implementation of a buffer.

Problem set

12.1 Explain why semaphores are an example of an asynchronous-send—synchronous-receive message transmission protocol.

…

12.6 Explain how semaphores are similar to asynchronous-send—synchronous-receive message passing.

12.7 How could two tasks synchronize their events through message passing; that is, how could you create a simple rendezvous?

12.8 Consider the following attempt to synchronize three events through message passing. By sending messages to each other, each ensures that the other tasks have reached that point in execution. Comment on the implementation.

```
// Task A                        // Task B                        // Task C
// Rendezvous with B and C       // Rendezvous with C and A       // Rendezvous with A and B
send( B );                       send( C );                       send( A );
receive( B );                    receive( C );                    receive( A );
send( C );                       send( A );                       send( B );
receive( C );                    receive( A );                    receive( B );
```

12.9 Suppose that one task is acting as a semaphore by receiving and sending messages. Explain how a task could behave as a counting semaphore. What data structure would you use in order to store a list of tasks that are waiting to access a semaphore?

12.10 We could allow local semaphore clients to negotiate between a semaphore server in order to allow semaphores that allow issues such as priority inversion to be solved. One problem with this is, however, suppose that a task requests and receives a semaphore, a higher priority task also requests that semaphore, so a message is sent to the specific client associated with the task holding the semaphore to increase its priority. Unfortunately, prior to the semaphore client receiving the message, the task released the semaphore and the semaphore client has set a message releasing that semaphore. What mechanism could you use to ensure that the subsequent message to increase that task's priority is ignored?

12.11 Suppose we had a semaphore client acting as a semaphore server. Suppose that there is a possibility the host going out of range of the other semaphore clients. How could such a situation be resolved? We will consider this later again when it comes to building fault tolerant systems.

12.12 Why would CAN allow a zero-length message inside a message frame?

13 Fault tolerance

No system will be perfect, and within the design, it will be understood that there are various defects, or faults, within the design and components. A fault is any software or hardware defect that may cause an error, and again if that error is not detected and corrected, the error may lead to a failure. A system that is designed to accept and deal with such faults is described as being *fault tolerant*; that is, the system is able to continue operation despite the existence of perturbations and faults within the system. Fault tolerance is another approach to ensuring non-functional requirements of safety within the system.

If a fault causes an error which then causes the system to no longer meet the a requirement of the system, that fault is said to have caused a *failure*. A failure in a hard requirement may be catastrophic to the system, while a failure in a firm requirement may result in the degradation of the quality of the service provided. If steps can be taken to subsequently minimize or eliminate harm resulting from an error, those steps are described as being a *failsafe*. If steps can be taken to ensure that the failure to meet one requirement does not prevent the system from meeting other requirements, such a system is described as being able to demonstrate *graceful degradation*.

As one example, an fault in programming that causes a periodically executing task to terminate by generating an uncaught exception (for example, an integer division-by-zero operation or following a wild or dangling pointer). Ideally, Such an error could be mitigated by having a second periodically executing task ensuring that all tasks that should be executing are executing, and if it is determined that a task has stopped execution, that task is restarted.

As another example, of these is a quad- versus a hexacopter drone. Each motor has a likelihood of failing, and if one motor failing causes the drone to crash, such a drone would not be fault tolerant. A quadcopter drone could adjust the rotation on the opposite rotor to the one that failed in order to keep the drone airborne. As it could still accomplish some of its objectives in this state, this demonstrates a form of graceful degradation; however, it would be impossible to control if a second motor failed. A quadcopter that attempted to land safely as soon as one of its motors failed would be failsafe. A hexacoptor drone would be more fault tolerant as it would be able to continue operation with up to two motors failing. In this case, it could exhibit a period of graceful degradation after one engine failed, and if a second engine failed, it could choose to either go into a failsafe mode (and land) or continue under even more averse conditions. In order to continue operation, however, the motors would have to be significantly more powerful than what would be normally required affecting either cost and weight.

Figure 20. Quad- and hexacoptor drones. Image source: http://smashingdrones.com/

As a brief introduction to fault tolerance, we will describe in greater detail some of the aspects of these, after which we will look at some look at solutions to achieve fault tolerance through

1. error detection and correction;
2. redundancy; and
3. communication schemes, specifically
    a. the problem of synchronizing clocks, and
    b. the (more general) Byzantine general's problem.

As a motivating example, many of the failures we have discussed thus far have been associated with more publicized space exploration missions, and for good reason: failures in industry tend not to be publicized, as they will in almost all cases adversely affect the brand reputation, both of the producer ("Why did your product fail?") and the user ("Why did you use that product?"). Thus, one situation from many years ago that was relayed in confidence to the first author was a situation in a manufacturing plant where a robot using a rail to allow movement, as shown in Figure 13-21, experienced a fault that resulted in the robot moving at high speeds down the rail and not stopping before it reached the end of the rail, thus careening off the end of the rail and destroying both itself, a significant amount of the product, and subsequently disrupting production. Had a person been standing there, he or she could have suffered either serious or fatal injuries as a result.

Figure 13-21. A robot on rails from the FANUC America Corporation, reproduced here for educational purposes.

The robots in Figure 13-21 are prevented from *falling off the rails* through both hardware and software mechanisms. Failures in industry will usually only receive attention if critical or fatal injuries are inflicted on humans, such was the case on July 1st, 2015, when a Volkswagen assembly-line robot struck the chest of a 21-year-old maintenance worker who was inside a cage meant to prevent contact between the robot and people.

We will now continue through a more thorough description of failures and fault tolerance.

13.1 Errors and failure in real-time systems

To describe tolerance and fault tolerance in real-time systems, we will begin with two definitions:

1. a *failure* is when the response of a system deviates from a specification as a result of an error,
2. an *error* is the manifestation of a fault within the system, and
3. a *fault* is any cause of an error, be it physical (mechanical or electrical) or algorithmic.

Sources of error may be described as

1. environmental effects,
2. specification or design faults,
3. implementation faults, and
4. component effects.

The ability to deal with the first is generally described as being robust, while the ability to deal with the balance is generally described as being fault tolerant; however, the line between the two is not concrete, and one party may suggest an effect is internal while others may suggest it is external. There are different strategies for dealing with errors caused by external factors, and we will investigate this with error detection and correction, but fault

A fault within the system may be described as being any one of:

1. *Transient faults* begin at a point in time, remain for indeterminate period of time, and then disappear; for example, the interference of solar events may cause electrical faults.
2. *Intermittent faults* are transient faults that reoccur; for example, a component may overheat, cease functioning, cool down and then continue functioning.
3. *Permanent faults* remain within the system until external intervention to repair it; for example, the flipped bit in Voyager 2 was a permanent fault.

One may argue, however, whether or not the third example is one of an internal fault (the bit was wrong) or the system being intolerant of external electromagnetic interference. We will, however, defer such discussions

There are two main approaches to dealing with faults:

1. *Fault prevention*, where steps are taken to prevent faults from occurring (providing sufficient cooling, duplicating sources of power, *etc*.).
2. *Fault tolerance*, where faults are detected and steps are taken within the system to continue operation within specifications.

We will discuss each of these next.

13.1.1 Fault prevention

The process of fault prevention is where external steps are taken by developers to prevent faults from occurring within a system. There are two aspects to fault prevention:

1. Prior to the deployment of a system, *fault avoidance* are those steps taken to minimize the likelihood of faults occurring in the system; while
2. Once deployment has occurred, *fault removal* involves those steps taken detect and correct faults when failures occur.

We will discuss both of these here.

13.1.1.1 Fault avoidance

Steps that can be taken to avoid fault include:

1. an adequately accurate model of the physical environment,
2. adequately reliable hardware with sufficient shielding for any anticipated interference, and
3. within software development, we require appropriate:
    a. specifications (sufficiently rigorous),
    b. design methodologies; for example, the spiral model is a sequence of
        i. determining objectives,
        ii. identifying and resolving risks,
        iii. development and testing, and
        iv. planning of the next iteration

        through a sequence of prototypes prior to the final development and release;
    c. programming languages, and
    d. computer-based tools.

Most of these are covered in software engineering courses; however, we will discuss the role of programming languages in real-time systems.

13.1.1.2 Fault removal

Once a system is deployed, either in the testing facility or following its release, failures will occur. While it would be ideal to remove all faults during testing, this is often impossible as it is almost never possible to test systems thoroughly under realistic conditions. This process involves detecting failures and analyzing their symptoms in order to determine the underlying fault. A software fault may be removed through an update, while a hardware fault may be dealt with through hardware replacement or steps taken to compensate for the fault; for example, overriding the effect of a driver stepping on the accelerator if that driver is simultaneously also stepping on the brake.

13.1.1.3 Summary of fault prevention

The aspects of fault prevention include steps taken to avoid faults prior to deployment to minimize the likelihood of faults and steps taken to remove faults after deployment has occurred.

13.1.2 Fault tolerance

The other approach to dealing with faults is to design the system to compensate for faults once they occur. The response of a system to faults may include:

1. full fault tolerance where specifications are still met for a given period of time,
2. graceful degradation where there is an increasing degradation in quality of service, and
3. failsafe where the system temporarily halts in a safe state.

As an example, the failure of the solid-state drives on Opportunity starting in December 2014 resulted in the system being switched to RAM-only mode on Sol 4027, 3775% beyond its intended lifespan. As Opportunity was designed to operate in RAM-only mode, this suggests the design was fault tolerant with graceful degradation built in: the rover is still capable of performing its tasks, although with a degraded quality of service: the amount of available memory is now restricted to 120 MiB of volatile DRAM without the 256 MiB of solid-state memory, and in the case of a reset, any photographs take or other date recorded is lost.

The ability of a system to tolerate hardware faults is the subject matter in your other courses. Means of handling faults during deployment include:

1. error detection, including
    a. watchdog timers,
    b. exceptions, and
    c. error detecting codes;
2. error correction including
    a. recovery blocks and

    b. error correcting codes; and
3. fault masking including
    a. redundancy, and
    b. error masking.

We will discuss two means of achieving fault tolerance in software, including

1. recovery blocks, and
2. exceptions.

The first reverts back to a prior state, while the second takes alternate actions as a result of a detected fault.

13.1.2.1 Recovery blocks

Previously, in Section 11.4.4.4, we have already seen that it is possible to take a snapshot of the state of the processor. If such states are periodically stored, these would establish recovery points. If a fault is detected, it may be possible to return to a previous recovery point and either proceed again, under the assumption that the fault is transient, or choose an alternate path of execution. The programming language Ada has support for recovery blocks where a sequence of possible alternative approaches is used so long as an acceptance test fails. If none of the alternate approaches is deemed acceptable, an error occurs.

13.1.2.2 Exceptions and exception handling

An exception is any condition other than that which is expected. In its simplest form, exception handling is performed by means of a return value. For example, if `malloc(…)` is unable to find a block of memory for the size requested, it returns `NULL`. It is up to the calling function to determine what is to be done if memory is not available; however, how often do you see something like:

```
size_t request = 3254;
int *p_data = (int *) malloc( request * sizeof( int ) );;

while ( p_data == NULL ) {
    request /= 2;
    p_data = (int *) malloc( request * sizeof( int ) );
}

size_t array_capacity = request;
```

A more comprehensive approach is to identify specific exceptions, and then when one type of exception is thrown, this is returned through the call stack until a function capable of handling that exception is identified, in which case, the appropriate block of code is executed to deal with that exception. For example, in C++, an exception is an instance of a type (primitive or aggregate):

```
#include <iostream>

int main() {
        try {
                throw 3;
        } catch( int n ) {
                std::cout << "Got " << n << std::endl;
        }

        return 0;
}
```

In this case, the output printed to the screen is “Got 3”.

Now, the statement throw 3 could have been executed by a function called inside the try block. For example:

```
#include <iostream>

void g() {
        throw 3;
        std::cout << "Finished 'g'" << std::endl;
}

void f() {
        g();
        std::cout << "Finished 'f'" << std::endl;
}

int main() {
        try {
                f();
        } catch( int n ) {
                std::cout << "Got " << n << std::endl;
        }

        return 0;
}
```

As before, the only output printed to the screen is "Got 3". The try-catch mechanism allows the exception to go back down the call stack until one of the calling functions is ready to deal with it.

Now, throwing an integer is not so useful; however, you can throw an instance of a specific class, and if a calling function catches such a class, it is now possible to pass information back as to what went wrong. For example, an out-of-bounds error could throw an instance of a class that contains information about the lower bound, the upper bound, and the value (outside that limit) that was accessed.

Such a mechanism is *forward recovering*—the issue is being dealt with. It is not possible to go back to a previous state.

13.1.2.3 Summary of fault tolerance

We discussed two approaches to software fault tolerance: recovery blocks, which return to a previous point in execution and make an alternate attempt, and exception handling, which allows faults to be signaled and future execution to deal with the faults.

13.1.3 Summary of failures

We have discussed failures in real-time systems, and focused on mechanisms in software to deal with faults. We have discussed fault prevention and fault tolerance.

13.2 Error detection and correction in signals

When communications between tasks occurs over ideal communication channels, such as using shared memory or usually or some form of cabled communication channel such as a bus (be it internal or peripheral), the introduction of errors into messages may be so remote that tasks may assume that any messages sent will be unchanged when received. When using a noisy communication channel, where there is a much higher probability of errors being introduced to the message, we must design the system to detect and either correct or at least mitigate such errors. We will model a noisy communication channel as a the received digital signal $r[k]$ being the sum of the sent signal $s[k]$ and noise added to the signal $n[k]$, as shown in Figure 13-22.

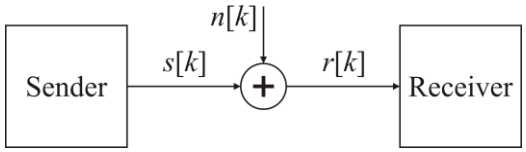


Figure 13-22. Communication under noisy conditions.

First, we will consider algorithms for detecting errors, and then we will look at algorithms for detecting and correcting errors; that is, reconstructing the original (error-free) transmission. We will look at a number of error detection and correction schemes including

1. repetition,
2. check-sums (parity),
3. cyclic-redundancy check, and
4. error-correcting codes.

These have the purpose of maintaining the *integrity* of the message.

13.2.1 Repetition

The simplest, but also most expensive, means of error detection is to send the message $n > 1$ times. This could be used if the channel is very noisy or if the sender and receiver are humans. Assuming that the probability of a bit being flipped is p, that the noise for different bits is independent and $p^2 \ll p$, we have an asymptotic approximation of the probability of receiving the wrong message shown in .

n	Receiving the correct message or detecting an error	Receiving the wrong message	Detecting an error
1	$1 - p$	p	-
2	$1 - p^2$	p^2	$2p$
3	$1 - 3p^2$	$3p^2$	-
4	$1 - 4p^3$	$4p^3$	$6p^2$
5	$1 - 10p^3$	$10p^3$	

What may appear surprising is that sending the message three times instead of twice, or five times instead of four is more likely to result in the wrong message being received as correct. To make sense of this apparent paradox, suppose you send a message twice, and the probability of an error is 1:10 (one in ten). In this case, the probability of receiving the wrong message is approximately 1:100. Suppose you now send a third copy. If either zero or two errors occur in the first two, the third does not affect the outcome, specifically, if there already were two errors (a probability of 0.01), we are no better off; however, if there is one error in either of the first two but not both, a probability of 0.2, an error in the third will reinforce this initial error and thus increase the probability of an error by 0.02, or a total probability of 0.03. Now, if we do detect an error, we would use the same method to resend, in which case, we may again detect an error, ad nauseam. Consequently, the number of incorrect messages that may potentially be received by sending two or four

$$n\sum_{k=0}^{\infty} p^2(2p)^k = \frac{np^2}{1-2p} \text{ and } n\sum_{k=0}^{\infty} 4p^3\left(6p^2\right)^k = \frac{4np^3}{1-6p^2},$$

respectively, requiring a total of

$$\sum_{k=0}^{\infty} 2n(2p)^k = \frac{2n}{1-2p} \text{ and } \sum_{k=0}^{\infty} 4n\left(6p^2\right)^k = \frac{4n}{1-6p^2},$$

messages, respectively.

Suppose you have 1000 messages to send and the probability of an error is 0.1.

Copies sent	Messages required	Incorrect messages accepted
1	1000	100
2	2500	12.5
3	3000	30
4	4255	4.3
5	5000	10

Replication is appropriate in only one situation: in real-time systems with firm (and not hard) real-time requirements and where noise is very high. If noise is significantly lower, check sums or cyclic redundancy checks are appropriate, and error correcting codes can be used to minimize the number of necessary transmissions.

13.2.2 Check sums

A *checksum* is a generic term for any function that, given a sequence of bits (or other data), provides a value such that if there is an error introduced to the data, the resulting calculation will return a different value, consequently, notifying the receiver that there was an error in transmission.

The simplest checksum is a parity bit, which calculates the *exclusive or* (XOR) of the bits in questions. This value is then appended to the end of the bit sequence. The receiving party can then make the same calculation and check whether or not they get the same value. This is identical to *even parity* which includes an extra bit that is

1. 0 if the sequence of bits has an even number of 1s, and
2. 1 if the sequence has an odd number of 1s.

This will detect an error in one bit (or any odd number of bits), but not two (or any even number of bits). Under the assumption that the probability of a bit being incorrect is $0 < p < 1$ and that these events are independent, the probability of two errors becomes $p^2 < p$. The 128 characters in ASCII use seven bits where the 8th bit was optionally used as a parity bit. Assuming that the probability of an error is less than 1:100 and that the errors are independent, the probability a message is successfully sent without any errors is approximately $1 - 8p + 28p^2$, the probability of there being only one error is approximately $8p - 56p^2$, and the likelihood of there being two or more errors is approximately $28p^2$. If the errors are not independent, meaning that errors occur in clusters so that if bit b_k is affected, there is a higher likelihood of either b_{k-1} or b_{k+1} being affect, too, parity bits perform significantly worse, for if there is an error in one bit, there is a higher likelihood that there will be two errors that will cancel each other out.

Another possibility is a parity word, where the text is divided into fixed-sized words which are then bit-wise XORed with each other. An *n*-bit parity word can be used to detect up to *n* faults, though two faults occurring *n* bits apart would cancel each other. Unfortunately, once again, there is a high probability that two errors cancel each other; consequently, we must use a different approach and we will next investigate cyclic redundancy checks.

13.2.3 Cyclic redundancy check (CRC)

A cyclic redundancy check is similar to a checksum, but the algorithm used is to treat the input as the coefficients of a polynomial and to then calculate a polynomial remainder using modulo 2 arithmetic (1 + 1 = 0). For example, if $x^3 + x^2 + 1$ is our polynomial, this will result in a 3-bit value.

Note that you can do polynomial division in Maple:

```
> Rem( x^10 + x^5 + x^4 + x^2 + x, x^3 + x^2 + 1, x) mod 2;
```

$$x^2 + x + 1$$

Essentially, one does long division on binary numbers, but instead of subtraction, an exclusive-or is performed. This suggests that such an algorithm would be very simple to implement in hardware, and so it is. CRC are used in a host of applications to detect errors in transmissions, including Ethernet, USB, Bluetooth, etc. Unlike parity bits, CRCs are much more sensitive to changes. For example, a parity bit would not detect a transposition of two characters, nor would it detect an increase in the length of a transmission if the padded characters were zero. To give an example of how polynomial division works in base two arithmetic, consider this example:

$$\begin{array}{r} 10011 \\ 1000101\overline{)\ 10010101001} \\ \oplus\underline{1000101} \\ 0001111100 \\ \oplus\underline{1000101} \\ 01110011 \\ \oplus\underline{1000101} \\ 110110 \end{array}$$

In this example, the polynomial is $x^6 + x^2 + 1$. The remainder must be six bits. You can readily see that this is straightforward to implement in hardware. To demonstrate that this is the correct remainder, we multiply the quotient and the divisor and add the remainder, always remembering to use bit-wise XOR instead of addition:

$$\begin{array}{r} 1000101 \\ \times\underline{10011} \\ 1000101 \\ 10001010 \\ \oplus\underline{10001010000} \\ 10010011111 \\ \oplus\underline{110110} \\ 10010101001 \end{array}$$

The polynomial equivalent of this is

```
> Quo( x^10 + x^7 + x^5 + x^3 + 1, x^6 + x^2 + 1, x) mod 2;  # =  10011
```

$$x^4 + x + 1$$

```
> Rem( x^10 + x^7 + x^5 + x^3 + 1, x^6 + x^2 + 1, x) mod 2;  # = 110110
```

$$x^5 + x^4 + x^2 + x$$

This is by definition the idea behind polynomial division; however, when the algorithm is actually implemented in a standard, modifications may be made. The most common of which is that the number on which the CRC is being calculated is appended with *n* zeroes (where the divisor is a polynomial of degree *n*) before the calculation is performed, and the CRC replaces these *n* zeros. This has the added benefit that it is trivial to check an answer: perform the CRC on the result (following the same algorithm) and the result had better be all zeros.

```
                  10011110101
1000101 ) 10010101001000000
        ⊕1000101
          0001111100
          ⊕1000101
            01110011
            ⊕1000101
              1101100
              ⊕1000101
                1010010
                ⊕1000101
                 01011100
                 ⊕1000101
                   1100100
                   ⊕1000101
                     100001
```

This would yield the CRC string **10010101001100001**. If you were to repeat this again with this as the input string, the output would be **10010101001100001000000**, and if any bits were changed, the last six bits would likely no longer be zero.

In a later chapter, we will discuss the Controller Area Network (CAN) bus designed for real-time embedded systems. The CAN bus message formats use the polynomial $x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$ which is `1100010110011001` or `0xC599`, resulting in a 15-bit CRC. This polynomial was chosen as the minimum number of bits that must be flipped before there is ever a chance that the resulting CRCs will be the same is six. Therefore, using this polynomial guarantees that up to five bit flips will be detected as the resulting CRCs will differ.

Aside: The number of corresponding bits that differ between two binary numbers is said to be the *Hamming distance* between the two numbers. For example, the Hamming distance between 00101010 and 01101011, 10101000 and 00101100 are all two.

An implementation of an 8-bit CRC is shown here where `sn` is the number of characters (bytes) in the string `string` and `pn` is the number bytes in the polynomial `poly`. This is based loosely on code provided by Raj Abishek[34].

```
#include<stdio.h>
#include<string.h>

void crc( char *string, size_t sn, char *poly, size_t pn ) {
        char remainder[pn + 1];

        size_t i, j, k;

        // The length of the string is 'sn + pn' with characters in the first 'sn' locations
        // Set the last 'pn' bytes of the string to '\0' and initialize
        // the remainder with the first 'pn' bytes from the string
        //   Note:  We set the characters to '\0' first, as the
        //   length of the string may be zero (sn == 0).

        for ( i = 0; i < pn; ++i ) {
                string[sn + i] = '\0';
        }

        for ( i = 0; i < pn; ++i ) {
                remainder[i] = string[i];
        }

        // The next bit that will be added is the first bit
        // of the pn-th character in the string.

        for ( j = pn; j < sn + pn; ++j ) {
                remainder[pn] = string[j];

                for ( k = 0; k < 8; ++k ) {
                        // Find the leading bit
                        char leading_bit = remainder[0] & 128;

                        // Shift the first 'pn' bytes to the left by 1
                        // shifting leading bits and copying the leading
                        // bit of the next byte

                        for ( i = 0; i < pn; ++i ) {
                                remainder[i] <<= 1;

                                if ( remainder[i + 1] & 128 ) {
                                        remainder[i] |= 1;
                                }
                        }

                        // Shift the last byte to the left by 1.
                        remainder[pn] <<= 1;

                        if ( leading_bit ) {
                                for ( i = 0; i < pn; ++i ) {
                                        remainder[i] ^= poly[i];
                                }
                        }
                }
        }

        // Copy the crc to the end of the string

        for ( i = 0; i < pn; ++i ) {
                string[sn + i] = remainder[i];
        }
}
```

[34] See https://gist.github.com/rajabishek

13.2.4 Error correcting codes

Check sums and CRCs allow for errors to be detected, but if it is absolutely necessary that the correct message to be sent, both these schemes require that the message be resent. Hamming introduced a means of encoding messages to allow for single bit errors to not only be detected but also be corrected by sending more than one parity bits, a process described as *single error correction* (SEC). The simplest example is triplication, with each bit being sent three times. If any one bit is flipped, the transmitted bit can still be deduced by the majority. This is referred to as a Hamming(3, 1) code, meaning that one bit can be sent as and corrected using three bits. To introduce a more practical approach, we will look at Hamming(7, 4) code. According to this scheme, if you want to send 4 bits $b_3b_2b_1b_0$ while correcting for a one bit error, you must include three parity bits $p_6p_5p_4$, each of which is the parity of only three of the four bits, according to this table:

	b_3	b_2	b_1	b_0	**Formula**
p_4	✓	✓	✓		$p_4 \leftarrow b_3 \oplus b_2 \oplus b_1$
p_5	✓	✓		✓	$p_5 \leftarrow b_3 \oplus b_2 \oplus b_0$
p_6	✓		✓	✓	$p_6 \leftarrow b_3 \oplus b_1 \oplus b_0$

You would then transmit $p_6p_5p_4b_3b_2b_1b_0$. For example, the parity bits of **1010** would have us consider **101**0, **1**0**1**0, **1**0**10** producing the three parity bits **010**, so **0101010** would be transmitted. Once the message is retransmitted, you simply recalculate the parity bits p_0', p_1' and p_2' and check the parities again and make sure they match up. If they do not, then you proceed based on what has changed. There are two possible cases with one-bit errors: where one of the

1. parity bits was flipped, which would affect only that parity bit, or
2. four bits was flipped, which would affect two or three parity bits.

If only one parity bit was flipped, ignore that and accept $b_3b_2b_1b_0$. If two or three parity bits differ, when you calculate the parity bits on the transmitted b_0 through b_3, you will find that

1. if p_4 and p_5 differ, then b_2 must have changed,
2. if p_4 and p_6 differ, then b_1 must have changed,
3. if p_5 and p_6 differ, then b_0 must have changed, and
4. If p_4, p_5 and p_6 differ, then b_3 must have changed.

In each of these cases, flip that bit back. Thus, eight 7-bit words map to each 4-bit original word, requiring exactly 8 $2^4 = 2^7$ words, so this accounts for all possible 7-bit words. Suppose the probability of an error in each bit was q, then the probability of no errors is $(1 - q)^7$, and the probability of one bit being flipped is $7q(1 - q)^6$, the probability that either no errors or only one error occurs is approximately $1 - 21q^2$ for small values of q. If the probability of an error in transmission is $q = 0.01$, this indicates that 99.79 % of all transmissions will be successful with an error of only one in 500, and when $q = 0.001$, 99.9979 % transmissions with an error in only approximately 1 in 50,000.

With one additional overall parity bit defined as $p_7 = p_6 \oplus p_5 \oplus p_4 \oplus b_3 \oplus b_2 \oplus b_1 \oplus b_0$, Hamming codes can be devised to correct one bit and detect two bit errors, described as *single error correction and double error detection* (SECDED). When one additional parity bit is added to Hamming(7, 4), it is often described as Hamming(8, 4) code. The following is an implementation of this code whereby an array of n characters is encoded within an array of $2n$ characters. Any single bit errors are corrected and if a two bit error is detected, the return value is set to false and the decoder returns.

```
#include <stdbool.h>

unsigned char encode_tab[16] = {   0, 225, 210,  51, 180,  85, 102, 135,
                                 120, 153, 170,  75, 204,  45,  30, 255};

// Correct decodes are highlighted in blue and 16 is used to indicate a two bit errors
unsigned char decode_tab[256] = {
     0,  0,  0, 16,  0, 16, 16,  7,  0, 16, 16, 11, 16, 13, 14, 16,
     0, 16, 16,  3, 16,  5, 14, 16, 16,  9, 14, 16, 14, 16, 14, 14,
     0, 16, 16,  3, 16, 13,  6, 16, 16, 13, 10, 16, 13, 13, 16, 13,
    16,  3,  3,  3,  4, 16, 16,  3,  8, 16, 16,  3, 16, 13, 14, 16,
     0, 16, 16, 11, 16,  5,  6, 16, 16, 11, 11, 11, 12, 16, 16, 11,
    16,  5,  2, 16,  5,  5, 16,  5,  8, 16, 16, 11, 16,  5, 14, 16,
    16,  1,  6, 16,  6, 16,  6,  6,  8, 16, 16, 11, 16, 13,  6, 16,
     8, 16, 16,  3, 16,  5,  6, 16,  8,  8,  8, 16,  8, 16, 16, 15,
     0, 16, 16,  7, 16,  7,  7,  7, 16,  9, 10, 16, 12, 16, 16,  7,
    16,  9,  2, 16,  4, 16, 16,  7,  9,  9, 16,  9, 16,  9, 14, 16,
    16,  1, 10, 16,  4, 16, 16,  7, 10, 16, 10, 10, 16, 13, 10, 16,
     4, 16, 16,  3,  4,  4,  4, 16, 16,  9, 10, 16,  4, 16, 16, 15,
    16,  1,  2, 16, 12, 16, 16,  7, 12, 16, 16, 11, 12, 12, 12, 16,
     2, 16,  2,  2, 16,  5,  2, 16, 16,  9,  2, 16, 12, 16, 16, 15,
     1,  1, 16,  1, 16,  1,  6, 16, 16,  1, 10, 16, 12, 16, 16, 15,
    16,  1,  2, 16,  4, 16, 16, 15,  8, 16, 16, 15, 16, 15, 15, 15
};

void hamming_encode( unsigned char *plain_text, unsigned char *coded_text, size_t plain_len ) {
    size_t i, j;

    for ( i = j = 0; i < plain_len; ++i ) {
        coded_text[j++] = encode_tab[plain_text[i] >> 4];
        coded_text[j++] = encode_tab[plain_text[i] & 15];
    }
}

bool hamming_decode( unsigned char *plain_text, unsigned char *coded_text, size_t plain_len ) {
    bool is_valid = true;
    unsigned char chr;
    size_t i, j;

    for ( i = j = 0; i < plain_len; ++i ) {
        chr = decode_tab[coded_text[j++]];

        if ( chr == 16 ) {
            is_valid = false;
            break;
        } else {
            plain_text[i] = chr << 4;
        }

        chr = decode_tab[coded_text[j++]];

        if ( chr == 16 ) {
            is_valid = false;
            break;
        } else {
            plain_text[i] ^= chr;
        }
    }

    return is_valid;
}
```

Hamming codes in themselves are not necessarily ideal, either, as three or more errors may result in an invalid message being accepted as correct. If it is absolute necessary that a message must be received intact, the following is a reasonable strategy:

1. Calculate the CRC and append this to the message.
2. Encode both the message and the appended CRC using the Hamming(8, 4) code.

When it comes time to decode the message:

1. Decode the message and reject the message if any of the blocks are detected as having at least two bits flipped.
2. If the decoding process works correctly, use the CRC to check the result:
    a. if the CRC matches, it is almost certain that the message is correct,
    b. otherwise, reject the message.

The overhead of Hamming(8, 4) is 100 % additional memory being required for any message being sent. In general, m parity bits can be used to generate a Hamming($2^m - 1$, $2^m - m - 1$) code that allows for SEC, and including an additional parity bit allows for SECDED. For larger codes, however, the increase in both size and run-time of software solutions becomes prohibitive. One application of a hardware implementation of Hamming codes is to ensure the correctness of RAM. On a 64-bit system, Hamming(127, 120) is used to encode 8 bytes using the additional seven parity bits and then adding an additional 8^{th} parity bit for error detection.

13.2.5 Summary of error detection and correction

The purpose of this section is to make you aware of means of error detection and correction in the transmission of messages over noisy channels. Implementations of these are widely available in libraries, and it does not require significant additional effort to implement these algorithms on top of any communications.

13.3 Redundancy

First suggested by John von Neumann in the 1950s, one solution to ensure fault tolerance is for a system to be redundant. The more obvious version is spatial redundancy where hardware components are repeated. Normally, this is prohibitively expensive; however, it can useful in extreme environments. In some space missions, we have four times redundancy in hardware, and the same program is executed on each of the systems. The answer is then found through a majority vote. If one of the four systems fails in some way, the system will still function correctly. If two systems fail, but fail in different ways, it is still possible to proceed. If two systems fail and in the same way, this becomes an issue as it is no longer possible to determine the correct response; however, this ensures that the fault is detected and can be dealt with by other means. The Space Shuttle had a fifth computer that would be loaded only if there was a 2-2 tie.[35] Another possibility is time redundancy, where calculations are performed repeatedly and the results are compared against each other.

With duplication in either time or spatial redundancy, it is possible to detect a single error, though one cannot proceed because it is not clear which of the two values was correct. With triplication, it is possible to detect either one or two different failures, although if two failures are the result of the same issue, this may cause an incorrect outcome.

If we can estimate the probability of a failure as p, and we know the frequency f of such an operation, it is possible to calculate the mean time between failures as $1/pf$. If the probability of a failure is p, then the probability that m out of n tasks performing that operation will experience a failure is

$$\tilde{p} = \binom{n}{m} p^m (1-p)^{n-m}.$$

You will note that

$$\sum_{m=0}^{n} \binom{n}{m} p^m (1-p)^{n-m} = (1-p)^n \left(\frac{1}{1-p}\right)^n = 1,$$

so it is guaranteed that somewhere between $m = 0$ and $m = n$ failures are occurring.

For example, if there is a probability $p = 0.02$ of a failure, and this computation is performed 10 times per second (a frequency of $f = 10$ Hz), the estimated wait time between failures would be $\frac{1}{p \cdot f} = \frac{1}{0.02 \cdot 10} = 5\,\text{s}$. If there are $n = 4$ processors performing this computation, the probability of there being 0 or 1 failures is

$$\binom{4}{0} 0.02^0 (0.98)^4 + \binom{4}{1} 0.02^1 (0.98)^{4-1} = 0.99766352.$$

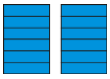
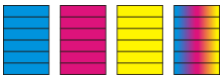
Thus, the probability of there being more than one failure is 0.00233648.

Redundancy does not however, always require duplication or triplication, as this example will show. As you are already aware, persistent data-storage drives (e.g., hard-disk drives and solid-state drives) are known to fail over time. In any embedded or real-time system where persistent storage is critical, this may be possible solution. The measure of failure of such drives is usually described by the *mean-time between failures* or MTBF, although the standard deviation may be quite significant and the distribution describing the likelihood of failure is more likely to follow a *bathtub distribution* than the more familiar normal distribution. A bathtub distribution has a higher probability that the drive will initially fail due to an undetected manufacturing defect, and if the disk survives this *infancy period*, there tends to be a lower rate of failure until the likelihood of components wearing out causes the probability to once again increase. Consequently, it is

[35] See https://history.nasa.gov/computers/Ch4-4.html.

often useful to arrange drives in parallel; such a scheme is described as being a *redundant array of independent disks* (RAID). In each case, given *n* drives, all containing different information, there are one or two additional drives that contain redundant information making it possible to reconstruct the previous drives if one fails. To increase access time, files are usually spread across all *n* drives, thus making it possible to simultaneously load up to *n* blocks into main memory. For each scheme, reading a block requires access to only one drive but writing a block requires that two or more drives are accessed. The various schemes are labeled RAID 0, 1, 2, *etc*., although Table 4 only describes the more common implementations. RAID 1, 4 and 5 each allow one disk to fail, while RAID 6 allows two disks to fail and is therefore more fault tolerant. One significant observation has been made is that when a failed drive is being reconstructed, this is a period during which there is a significant amount of disk activity for all drives, thereby making it even more likely that a second disk will fail, suggesting that a scheme such as RAID 6 is more reasonable if fault tolerance is critical. The parity blocks may be achieved using a scheme as simple as exclusive or, but other schemes have been used. As an example, Innodisk, Inc. produces hardware that implements RAID 1 suitable for embedded systems.

Table 4. Descriptions of RAID 1, 4, 5 and 6.

	Scheme	Drives	Description	Representation
RAID 1	duplication	2	Two drives have the same information on each. Writing requires access to both drives.	
RAID 4	parity	$n + 1$	An additional drive contains parity information of the first n. Failure in any drive can still allow all information to be accessed. Writing requires access to the drive containing the block and the parity drive. If $n = 1$, this is equivalent to RAID 1.	
RAID 5	parity	$n + 1$	Like RAID 4, but where the parity block is distributed throughout the $n + 1$ drives, thus distributing the required number of writes to the parity block. Writing requires access to the drive containing the block and the drive containing the corresponding parity block.	
RAID 6	double parity	$n + 2$	Like RAID 5, but where two parity blocks are distributed throughout the $n + 1$ drives, thereby allowing two drives to fail but still distributing the load. Writing requires access to the drive containing the block and the drives containing the two corresponding parity blocks.	

13.4 Clocks

Another issue when you have multiple independent systems interacting is the synchronization of clocks. Independent devices will each have separate clocks, and in time, each of these clocks will experience drift from a standard clock time. A fault tolerant system will have a mechanism for synchronizing these clocks despite the possibility of one or more of the clocks failing. Additionally, even if the clocks are not failing, an error may be introduced in the communication of the time during synchronization. We will describe a clock C as giving a time $C(t)$ at some actual time t. An ideal clock is one where $C(t) = t$ at all times; however, this is impossible to achieve. Instead, we need to list the desirable properties of functioning clocks and these will determine how we can maintain time synchronization.

In any system, we may have one of two possibilities:

1. there is a standard clock $C_s(t)$ against which all other clocks are synchronized against, or
2. there is no standard clock, but all other clocks must synchronize with each other.

We will discuss these two classifications of systems where timing is required, the requirements for timing, how to achieve synchronization, and how to achieve synchronization in terms of the two classifications. We will start, however, with an example of where unsynchronized clocks led to numerous deaths.

13.4.1 Example: the Patriot missile system

This summary is based on the report by the Government Accountability Office of the United States and a note from Robert Skeel sent to SIAM News, July 1992, Volume 25, Number 4, page 11.

The Patriot missile system was developed during the early 1970s to intercept Soviet aircraft and cruise missiles travelling at approximately Mach 2 or 2400 km/h. It was designed to simultaneously track numerous targets in real time and it was built on a 24-bit computer; the system allowed it to process one target at a time. Consequently, each target would move between registrations and a tracking algorithm would determine where the target should be at the next registration. If the system was expecting, for example, cruise missiles and with the next registration, the target did not move according to the predicted trajectory of a cruise missile, it was assumed that either the initial object was not a cruise missile or it was a ghost. In either case, the registration would be dropped.

The tracking algorithm also used the system clock, which stored time as an integral number of tenths of a second since the system had been last turned on or reset. The tracking algorithm required that this time be converted into a floating-point number, and this was done by multiplying by

$$0.00011001100110011001100_2,$$

a truncated 24-bit fixed-point approximation of 0.1 with relative error of 0.000095 %. The overall system was also designed to be highly mobile and was meant to operate for only a few hours before it would be redeployed to avoid Soviet countermeasures; consequently, it had never been tested for long periods of time.

When the system was sent to the Middle East to protect American assets and interests from Iraqi Scud tactical ballistic missiles travelling at 4400 km/h, the system had to be upgraded. In addition, the system would be static and remain operational for days on end. With faster targets, numerous changes were made to the software system and six upgrades were issued between August 1990 and February 1991. One aspect of one of these upgrades was to introduce a more accurate integer–to–floating-point conversion algorithm; unfortunately, it was not introduced at each instance where the clock time was converted to a floating-point number. Thus, different components of the algorithms would be using different times. If the system was up for only 2 h, this difference would be negligible: only 69 ms and thus different components would differ in their calculations of the expected position of the Scud by approximately 8 m—well within the margin of error.

Up until January 18th, 1991, no Patriot interceptors had been fired (and the one on that date had been fired on a phantom target). On February 11th, 1991, the Israelis had reported a problem to their American counterparts: after 8 h of

operation, the system was already significantly off, as the tracking software was now off by 34 m. They also found that resetting the system (requiring 60 to 90 seconds) solved this problem. The problem was tracked down and a software upgrade was released on February 16th, but distribution was still through physical channels. On February 21st, operators throughout the Middle East were warned that "very long run times" would result in degraded, but "very long" was not quantified. Consider the burden this placed on the operators: the system appears to be up and running and any reset to the system would be entirely the responsibility of the operator; a restart would place others at risk while the system is down. Had they been told to reset the system every eight hours, that responsibility would have been shifted from the operators to those who issued the order to reset on that schedule.

Consequently, on February 25th, the system at Dhahran, Saudi Arabia, had not yet received the software patch and it had been running for approximately 100 h without a reset: the different components placed the expected locations of the Scuds over half a kilometre apart. That day, a Scud missile launched at Dhahran, Saudi Arabia, was not intercepted and the missile struck an army barrack killing 28 Americans; the software upgrade arrived the following day.

You can read the report, *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*, at http://www.gao.gov/assets/220/215614.pdf.

Note that in storing the approximation of 0.1, they did not even use reasonable rounding:

0.00011001100110011001100
0.0001100110011001100110011001100110011001100⋯
0.00011001100110011001101

The approximation $0.000110011001100110011001\mathbf{1}_2$, found using IEEE-754 rounding rules, has a relative error of 0.000024 % as opposed to the 0.000095 % relative error of the approximation that was used. After 100 h of operation, the clocks would have had only a 0.086 difference.

A benefit of procedural programming and software factorization: had the conversion from integer to floating-point been uniformly implemented as a single function call or macro, changing it in one location would have solved the problem uniformly; however, without code factorization, each instance had to be hunted down and change individually.

13.4.2 Requirements

Assuming we have a standard clock, we require:

1. Correctness: the difference between the clock and standard time is bounded, or $\left|C(t)-C_s(t)\right|<\varepsilon$.
2. Bounded drift: the rate at which the clock deviates from the standard clock is bounded, or $\left|\frac{\mathrm{d}}{\mathrm{d}t}\left(C(t)-C_s(t)\right)\right|=\left|\frac{\mathrm{d}}{\mathrm{d}t}C(t)-\frac{\mathrm{d}}{\mathrm{d}t}C_s(t)\right|=\left|\frac{\mathrm{d}}{\mathrm{d}t}C(t)-1\right|<\rho$. This says that the clock stays within a linear envelope of the standard clock.
3. Monotonicity: $C(t_1) \geq C(t_0)$ whenever $t_1 > t_0$.
4. Chronoscopicity: the measurement of two time intervals of equal length should be approximately equal. This says that if $t_2-t_1=t_4-t_3$, then $C(t_2)-C(t_1)\approx C(t_4)-C(t_3)$. This can be achieved locally if the concavity is bounded: $\left|\frac{\mathrm{d}^2}{\mathrm{d}^2t}C(t)\right|<\gamma$.

To demonstrate each of these, consider Figures 2 to 5 where the cyan envelope (lines parallel to the diagonal) is ε from the correct time $C_s(t) = t$ and the red envelope (forming a cone) is a deviation from that at a rate of ρ per unit time. In each case, only one of the four requirements is broken.

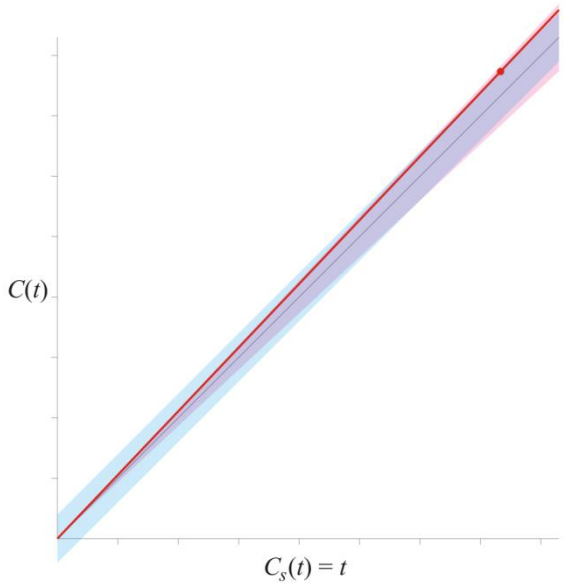


Figure 13-23. Breaking the first requirement: the clock is too fast.

In Figure 13-23, after being synchronized at time $t = 0$, the drift is so large that it differs from the standard time by more than ε. Synchronizing clocks will solve this problem.

7

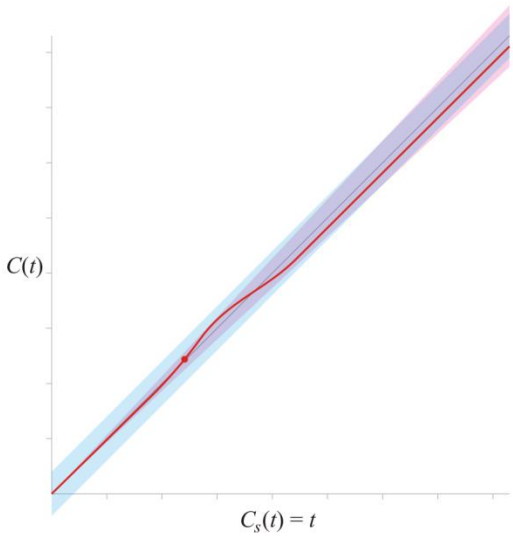


Figure 13-24. Breaking the second requirement: the rate of change is too fast.

In Figure 13-24, the slope at the indicated point is greater than $1 + \rho$. If possible, we should consider counting more ticks per unit time.

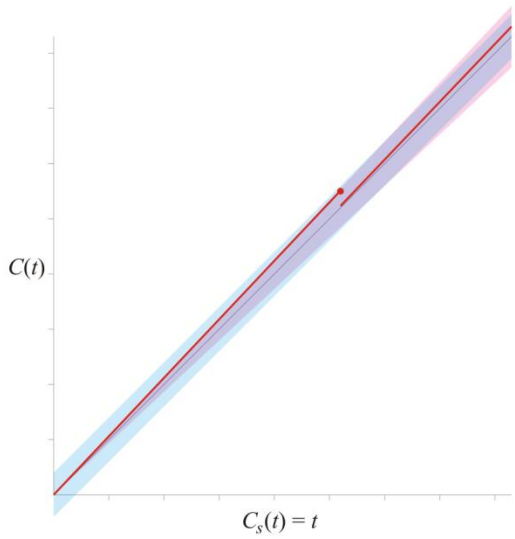


Figure 13-25. Breaking the third requirement: time is no longer monotonic.

In Figure 13-25, by synchronizing the clocks, it is as if time went backward for one split second. We will have to adopt a more gradual means of synchronizing clocks.

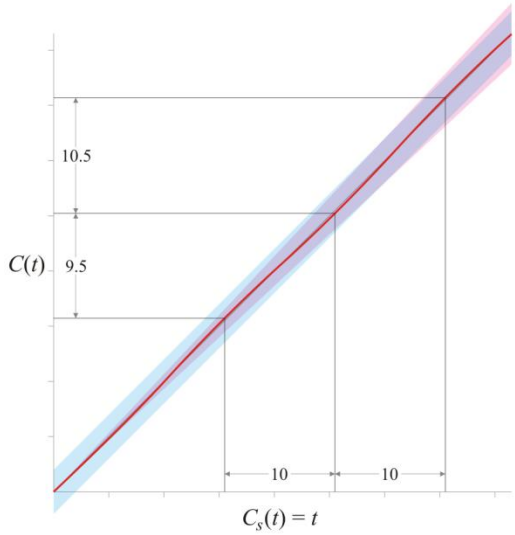


Figure 13-26. Breaking the fourth requirement: the same interval appears to have significantly different lengths when measured at different times.

Finally, in Figure 13-26, the clock is oscillating around the standard time, but never enough to break any of the other requirements; however, two consecutive 10 ms intervals are measured as being 9.5 ms in the first case and 10.5 ms in the second.

Suppose we have a quartz clock and it is designed to vibrate at $2^{15} = 32768$ Hz.[36] Each clock, however, will have an error associated with it, and therefore we will have to synchronize it from time to time. How often, and how we synchronize the clock will depend on the requirements on the behavior of the clock.

[36] This is chosen so that each clock tick increments a 15-bit counter and a counter overflows indicates one second.

13.4.3 Restoring synchronization

Suppose we determine that our clock is 100 ms fast (most quartz clocks, under normal conditions, will drift by half a second per day), we cannot simply update the time to match the actual time: this would break the last three of our requirements. For example, if the clock jumps back by 200 ms, this may result in alarms being triggered twice. Instead, we could, for example,

1. count one second as 32769 Hz for 3277 s or approximately 55 minutes,
2. count one second as 32770 Hz for half that time (half an hour), or
3. count one second as 32800 Hz for 100 s.

By playing a graduated game of catch-up or slow-down, the requirements for clocks are maintained. Sudden changes could destroy monotonicity and chronoscopicity.

13.4.4 Achieving synchronization

We will discuss how often it is necessary to synchronize clocks when there is a standard clock against which all others are synchronized, and some of the problems with distributed synchronizations with one solution.

13.4.4.1 Synchronization with a standard clock

If we are synchronizing with a standard clock, we need only use the previous algorithm to adjust our clock. The more significant question is, however, how often must we synchronize?

First, there is an error due to synchronization; for example, the transmission time of the time. Thus, let us assume at time t_0, the clocks are synchronized and

$$\left|C\left(t_0\right)-t_0\right|<\delta .$$

Over time, the clock will drift so that

$$\left|C\left(t\right)-t\right|<\delta+\rho\left(t-t_0\right).$$

To ensure that $\left|C\left(t\right)-t\right|<\varepsilon$, it is therefore necessary to ensure that

$$\delta+\rho\left(t-t_0\right)<\varepsilon ,$$

so solving this we have that $t-t_0<\dfrac{\varepsilon-\delta}{\rho}$, and therefore the clocks should be synchronized at least once every $\dfrac{\varepsilon-\delta}{\rho}$ time units.

For example, if ε = 1 s, δ = 50 ms, and the clock drifts by half a second per day, that is

$$\rho=\frac{0.5\,\mathrm{s}}{1\,\mathrm{d}}=\frac{0.5\,\mathrm{s}}{86400\,\mathrm{s}}\approx 5.79\times10^{-6},$$

then we must synchronize clocks at least every 164160 s or every 1 d 21 h 36 min.

13.4.4.2 Distributed synchronization

In a distributed system, the goal is to synchronize all the clocks. Unfortunately, with *n* clocks, it is entirely possible that some clocks, say *d*, could be defective (or faulty); that is, arbitrarily wrong due to local issues (a clock could, for example, stop, become damaged, or the power source is weakened, or there is noise in the communication channel). One may think that the best solution would be to simply average the times of the current clock and all surrounding clocks; however, in Lamport and Melliar-Smith 1985 paper (*Synchronizing clocks in the presence of faults*), they demonstrate that one faulty clock, can prevent two functioning clocks from properly synchronizing. The faulty clock can send incorrect information to each functioning clock in such a way to prevent synchronization. Note that it may not be the clock itself that is faulty, errors could be introduced in the transmission, as well. Consider the situation shown in Figure 13-27 from Lamport and Melliar-Smith's paper.

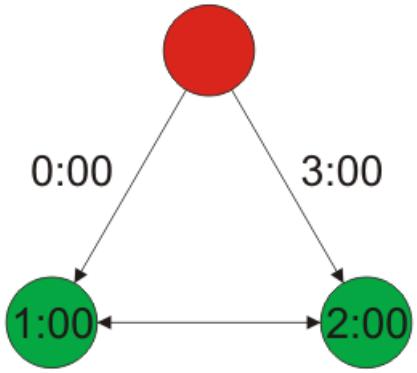


Figure 13-27. A malfunctioning clock sends two separate times to two different clocks.

Here we have two clocks that are functioning normally. They could synchronize their times and agree upon 1:30 as a synchronized time. The third clock, however, is faulty in that it sends different times to each clock. Now, the first clock averages 0:00, 1:00 and 2:00 to determine that it need not change its time, and the second clock averages 1:00, 2:00 and 3:00 to also determine that it, too, need not change its time.

In general, if the number of accurate clocks is not greater than twice the number of faulty clocks, no algorithm exists to the synchronization of the accurate clocks, for arguments that are similar to that provided in the Byzantine general's problem which we will see next.

Synchronizing clocks between systems can be an issue if there is a delay in the transmission. One may say "we are at 5848324 s", but it takes 3 seconds to transmit, in which case, the reported time is three seconds off. Usually, however, we will assume that the transmission time is significantly below the unit time (say, on the order of milliseconds when the clock counts in seconds). If more precision is necessary, messages could be bounced back and forth, and the time could be adjusted by half the time between received messages.

13.4.4.2.1 Synchronization algorithm

The algorithm for synchronizing *n* clocks of which *d* are defective was proposed by Leslie Lamport et al. We will first consider the basic algorithm and then discuss a perhaps unintuitive design choice in the algorithm.

At time *t*, for the k^{th} clock, let C_j be the time reported by each of the clocks in the system. If one of the clocks does not report a time, let $C_j = \infty$. Now, for each $j \neq k$, define

$$\tilde{C}_j = \begin{cases} C_j & |C_j - C_k| < \varepsilon \\ C_k & \text{otherwise} \end{cases}.$$

That is, the k^{th} clock will assume that it is correct and if any other clock has a reported time that is more than ε from the k^{th} clock, it will assumed to be broken and its value will be replaced by the value C_k. We then then update[37] the k^{th} clock to have the value

$$C_k \leftarrow \frac{1}{n}\sum_{j=1}^{n}\tilde{C}_j \,.$$

Under these conditions, all good clocks will differ by at most

$$\left|C_i - C_j\right| < \frac{3d}{n}\varepsilon$$

and if we have at least $n = 3d + 1$ clocks, they will differ by less than ε. This algorithm is named CNV (from *convergence*) and is classified as a convergence-averaging algorithm.

See Lamport and Mellior-Smith, *Synchronizing clocks in the presence of faults*, *J. ACM*, Vol. 32, No. 2 (Feb 1980), pp.105-17.

13.4.4.2.2 Duplication over exclusion

In CNV, if another clock indicates a time greater than ε away, it is assumed that clock is faulty and its value is replaced with the value from this clock. It would be easier to exclude such values and to only average those that, at this point, are not considered faulty—the spread of these values would be tighter. Unfortunately, it would also be further away from the actual time.

For example, if four clocks, all initially synchronized at time $t = 0$ and $\varepsilon = 1$ and after 1000 seconds, the clocks read

999.600 999.800 999.900 1000.300

After synchronization, all clocks read 999.9. Suppose, however, with the next synchronization, one clock malfunctions; it begins to slow down:

1998.600 1999.500 2000.000 2000.200

Now, if each clock only averaged the acceptable values, the times would be

1999.050 1999.575 1999.900 1999.900

The average of the three functioning clocks has slowed to 1999.792. Instead, using CNV, the latter two would have greater weight away from the faulty value:

1999.350 1999.575 1999.925 1999.975

Consequently, the impact of the malfunctioning clock will be reduced on the average of the functioning clocks. In this example, the average of the three functioning clocks is 1999.825, and thus the impact of the malfunctioning clock is 30 % less the previous average of 1999.792.

If the malfunction is permanent or intermittent, and the clock continues to slow down at the same rate, with the next synchronization, it will be approximately 2998.35, in which case it will have no further impact on the functioning clocks

[37] Note, again, we cannot just replace the time, but we can speed the clock up or slow it down for a specific period of time until the times match.

which will be approximately 2999.825. If the malfunction was transient, the four clocks will still converge again to approximately 2999.7.

13.4.4.2.3 Summary of distributed synchronization

We have discussed the problem of distributed synchronization and how to synchronize clocks in the presence of faults using the algorithm proposed by Lamport and Mellior-Smith.

13.4.4.3 Summary for achieving synchronization

We have considered how often synchronization must occur between clocks to ensure that they do not differ by more than a prescribed upper bound of ε units of time, and how to achieve synchronization in a distributed system (where synchronization with a standard clock is a trivial problem).

13.4.5 Summary of clocks

In this topic, we considered a slight variation on the synchronization problem: this one being a problem in synchronizing clocks. If multiple independent clocks are to attempt to synchronize, where one or more of those clocks may be faulty, determining the correct time becomes more difficult if there is no centralized server.

13.5 Byzantine generals' problem

As another algorithm to allow fault tolerance, suppose we have three independent tasks that must communicate in order to achieve a common goal, and at least two tasks are required to successfully coordinate in order complete the goal. However, the independent tasks can only communicate asynchronously with each other through messages. Suppose one independent tasks is malfunctioning and sending out erroneous error messages or that one of the messages sent has an error. Is it possible for the two functioning independent tasks to successfully detect the malfunctioning task and take appropriate steps to solve the problem?

There are two types of malfunctions that could affect any such communication:

1. omission failures, and
2. commission failures.

An omission failure is one where the system fails to respond; this could be the result of a hardware failure (e.g., a reset or faulty hardware) or a software failure in failing to either send or receive a message. A commission failure is one where a is sent and received, but the message is invalid due to faulty or inappropriate generation or processing of messages or corrupt data.

Like our problems with synchronization, the problem of communication has been reduced to a convenient-to-remember story about the Byzantine Empire.

Figure 28. Constantinople (modern-day Istanbul) at the time of the Byzantine Empire flanked by Justinian I and Theodora. Photographs by Petar Milošević.

As the empire began to decay—especially in the face of a growing Arab caliphate—generals would often vie for power, even sabotaging others. This has been formalized into the following scenario:

Suppose we have *n* lieutenants commanding *n* divisions of troops. These lieutenants can only communicate with one another through messages. Now, suppose there is a general who sends out a single message to all the lieutenants. This general could be a designated or elected task sending instructions, or it may be a scenario that is being simultaneously observed by all of the tasks. In either case, the lieutenants need to follow an appropriate agreed-upon course of action. There is only one problem:

The general himself may be disloyal.

The characteristics of a general are:

1. a loyal general will send the same message to all lieutenants, while
2. a disloyal general will send different messages to different lieutenants.

Remember to interpret "*loyal*" as "*functioning*" and "*disloyal*" as "*faulty*". Also note that a faulty unit may only be malfunctioning for one interaction: a noisy communication channel may have obfuscated one transmission.

For any course of action to work, all loyal lieutenants must follow the same course; however, suppose we have two loyal lieutenants and a disloyal general. We may have the scenario shown in Figure 13-29.

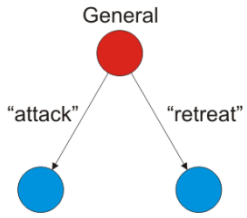


Figure 13-29. A disloyal general confounding his lieutenants.

Note that we use "attack" and "retreat" to represent a binary message. We could include a more complex message if we wanted.

The only way for the loyal lieutenants to realize there is a problem is to communicate with each other. You could have a rule-of-thumb such as "if in doubt, retreat". Unfortunately, the leads to the situation in

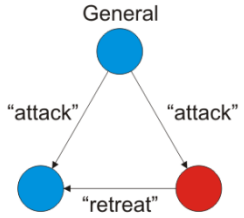


Figure 13-30. A disloyal lieutenant.

In this scenario, the loyal lieutenant is told to attack, and the attack will succeed if he follows this course of action; however, the disloyal lieutenant passes on the message "retreat". Consequently, the rule-of-thumb "if in doubt, retreat" is followed. You could impose a different rule-of-thumb, but in any case, some sequence of orders will result in the loyal lieutenants from executing the successful course of action.

In general, it can be shown that, given d disloyal participants, there must be more than $3d$ participants for the loyal lieutenants to come to an agreed upon course of action where:

1. a loyal general requires $2d$ loyal lieutenants for them to follow his course of action, while
2. a disloyal general requires $2d + 1$ loyal lieutenants for the lieutenants to come to up with a course of action on their own.

We will assume that any message passing will be correct in that any corrupted message can be reinterpreted as a disloyal lieutenant or general. We will first look at a non-solution, and then we will look at a solution.

13.5.1 A non-solution

Suppose that each lieutenant sends a message to each other lieutenant reiterating the received order. In this case, the lieutenants could then follow the majority course of action. Unfortunately, this can fail if there is a faulty general and a collaborating lieutenant regardless of how many loyal lieutenants there are, as is shown in Figure 13-31, where the general sends half the loyal lieutenants the command to attack, while the other half are told to retreat. All loyal lieutenants could send each other the command that they were issued, and the collaborator could simply force each half to follow the order given by reinforcing it: the left half will count four votes for "attack" and three for "retreat"; while the other half will count three votes for "attack" and four for "retreat".

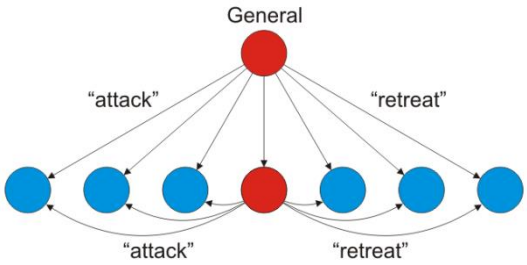


Figure 13-31. A disloyal general with a collaborating lieutenant.

13.5.2 A solution not using signatures

Suppose that each lieutenant can send messages, and those messages can be forwarded, but disloyal lieutenants can modify those messages. In this case, if there are no more than d disloyal participants, with at least $3d + 1$ participants, then we can apply a recursive algorithm.

13.5.2.1 No disloyal participants

If there are no disloyal participants ($d = 0$), lieutenants may simply follow the directions of the general.

13.5.2.2 At most one disloyal participant

If there is at most one disloyal participant ($d = 1$), each lieutenant must query other lieutenants as to the course of action indicated by the general. Each loyal lieutenant will forward, unchanged, any message sent to it by the general. Each lieutenant will then follow a majority decision.

For example, if there were $n = 10$ participants, each loyal lieutenant would fill in the following table:

General	1	2	3	4	5	6	7	8	9
v (?)	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9

Once all messages have been received, then v = majority($v_1, \ldots, v_{n-1}$). If there is a tie, there is a pre-selected default course of action.

In a nutshell, if you are a lieutenant:

1. First, you tell everyone what the general told you, and all the other lieutenants tell you what the general told them.
2. Then, you forget what the general told you, and instead, you make your decision entirely based on the majority of what other lieutenants told that the general told them.

In essence, you say what you were told and then assume the general may be untrustworthy so instead you follow what the majority of your colleagues told you they were told.

13.5.2.3 At most two disloyal participants

If there are at most two disloyal participants ($d = 2$), each lieutenant must query every other lieutenant as to the course of action indicated by the general, but in addition, each lieutenant must then query every other lieutenant as to what the lieutenants indicated the general's course of action would be.

In a sense, each lieutenant would construct the above vector, and then each lieutenant would forward that vector to every other lieutenant. For demonstration purposes, let us assume that this table is generated by the third lieutenant (assuming she is loyal). Lieutenant 3 would create this vector, where $\boldsymbol{v_3}$ was received from the general, and all other entries were received from other lieutenants.

General	1	2	3	4	5	6	7	8	9
$v = ?$	v_1	v_2	$\boldsymbol{v_3}$	v_4	v_5	v_6	v_7	v_8	v_9

Once this vector was generated, she is unfortunately not sure about these values, so she forwards this vector to all other lieutenants, and each lieutenant forwards her their vector. It is essential that all loyal lieutenants forward their information unchanged. Thus, each lieutenant would create the following table, where $v_{i,j}$ is read as "Lieutenant i says that Lieutenant j said that the general said v". We do not include the entries on the diagonal, as $v_{i,i}$ is identical to saying "Lieutenant i says that Lieutenant i said that the general said v". This information is already in the above table, which forms the third row of this table:

General	1	2	3	4	5	6	7	8	9
$v = ?$	$v_1 = ?$	$v_2 = ?$	$\boldsymbol{v_3}$	$v_4 = ?$	$v_5 = ?$	$v_6 = ?$	$v_7 = ?$	$v_8 = ?$	$v_9 = ?$
		$v_{1,2}$	$v_{1,3}$	$v_{1,4}$	$v_{1,5}$	$v_{1,6}$	$v_{1,7}$	$v_{1,8}$	$v_{1,9}$
	$v_{2,1}$		$v_{2,3}$	$v_{2,4}$	$v_{2,5}$	$v_{2,6}$	$v_{2,7}$	$v_{2,8}$	$v_{2,9}$
	$v_{3,1} = v_1$	$v_{3,2} = v_2$	-	$v_{3,4} = v_4$	$v_{3,5} = v_5$	$v_{3,6} = v_6$	$v_{3,7} = v_7$	$v_{3,8} = v_8$	$v_{3,9} = v_9$
	$v_{4,1}$	$v_{4,2}$	$v_{4,3}$		$v_{4,5}$	$v_{4,6}$	$v_{4,7}$	$v_{4,8}$	$v_{4,9}$
	$v_{5,1}$	$v_{5,2}$	$v_{5,3}$	$v_{5,4}$		$v_{5,6}$	$v_{5,7}$	$v_{5,8}$	$v_{5,9}$
	$v_{6,1}$	$v_{6,2}$	$v_{6,3}$	$v_{6,4}$	$v_{6,5}$		$v_{6,7}$	$v_{6,8}$	$v_{6,9}$
	$v_{7,1}$	$v_{7,2}$	$v_{7,3}$	$v_{7,4}$	$v_{7,5}$	$v_{7,6}$		$v_{7,8}$	$v_{7,9}$
	$v_{8,1}$	$v_{8,2}$	$v_{8,3}$	$v_{8,4}$	$v_{8,5}$	$v_{8,6}$	$v_{8,7}$		$v_{8,9}$
	$v_{9,1}$	$v_{9,2}$	$v_{9,3}$	$v_{9,4}$	$v_{9,5}$	$v_{9,6}$	$v_{9,7}$	$v_{9,8}$	

Now, one point is that Lieutenant 3 doesn't care what anyone else claims she said the general said, so she will ignore that column (essentially, replacing all values in the column with the value she actually received). Thus, we replace $v_{i,3} \leftarrow \boldsymbol{v_3}$.

General	1	2	3	4	5	6	7	8	9
$v = ?$	$v_1 = ?$	$v_2 = ?$	$\boldsymbol{v_3}$	$v_4 = ?$	$v_5 = ?$	$v_6 = ?$	$v_7 = ?$	$v_8 = ?$	$v_9 = ?$
		$v_{1,2}$	$\boldsymbol{v_3}$	$v_{1,4}$	$v_{1,5}$	$v_{1,6}$	$v_{1,7}$	$v_{1,8}$	$v_{1,9}$
	$v_{2,1}$		$\boldsymbol{v_3}$	$v_{2,4}$	$v_{2,5}$	$v_{2,6}$	$v_{2,7}$	$v_{2,8}$	$v_{2,9}$
	$\boldsymbol{v_1}$	$\boldsymbol{v_2}$		$\boldsymbol{v_4}$	$\boldsymbol{v_5}$	$\boldsymbol{v_6}$	$\boldsymbol{v_7}$	$\boldsymbol{v_8}$	$\boldsymbol{v_9}$
	$v_{4,1}$	$v_{4,2}$	$\boldsymbol{v_3}$		$v_{4,5}$	$v_{4,6}$	$v_{4,7}$	$v_{4,8}$	$v_{4,9}$
	$v_{5,1}$	$v_{5,2}$	$\boldsymbol{v_3}$	$v_{5,4}$		$v_{5,6}$	$v_{5,7}$	$v_{5,8}$	$v_{5,9}$
	$v_{6,1}$	$v_{6,2}$	$\boldsymbol{v_3}$	$v_{6,4}$	$v_{6,5}$		$v_{6,7}$	$v_{6,8}$	$v_{6,9}$
	$v_{7,1}$	$v_{7,2}$	$\boldsymbol{v_3}$	$v_{7,4}$	$v_{7,5}$	$v_{7,6}$		$v_{7,8}$	$v_{7,9}$
	$v_{8,1}$	$v_{8,2}$	$\boldsymbol{v_3}$	$v_{8,4}$	$v_{8,5}$	$v_{8,6}$	$v_{8,7}$		$v_{8,9}$
	$v_{9,1}$	$v_{9,2}$	$\boldsymbol{v_3}$	$v_{9,4}$	$v_{9,5}$	$v_{9,6}$	$v_{9,7}$	$v_{9,8}$	

Now, Lieutenant 3 will calculate for each *j* the value:

$$v_j = \underset{1 \le i < n,\, i \ne j}{\text{majority}} \{ v_{i,j} \};$$

that is, "I will take v_j to be the majority of what I and others claim to have heard from the i^{th} lieutenant." Then, having calculated the majority opinion of what each lieutenant said, now calculate

$$v = \text{majority}\{v_1, \ldots, v_{n-1}\}$$

using the values we just calculated.

In a nutshell, if you are a lieutenant:

1. First, you tell everyone what the general told you, and all the other lieutenants tell you what the general told them.
2. Second, you then tell every other lieutenant what every lieutenant told you the general said.
3. Then, you forget what the every lieutenant told you, and instead, you decide what every lieutenant said the general said by taking the majority of what the other lieuenants said each lieutenant said.
4. Finally, you forget what the general told you, and you base your decision based on the majority of what you decided each lieutenant said they were told.

This can subsequently be extended more and more often.

13.5.2.3.1 Example with seven participants and two disloyal lieutenants

If the general is loyal, he will send the same order to all lieutenants. Consequently, if there were seven participants (six lieutenants), two disloyal lieutenants could not produce a majority opposing the general for any of the loyal lieutenants. For example, suppose the general issues the order *v*. Lieutenant 3 would create the following table based on information sent from other lieutenants. Regardless of what other lieutenants claim 3 said, 3 is aware that it received *v*, so it populates that column. Question marks indicate any possible order issued by the disloyal Lieutenants 5 and 6.

General	1	2	3	4	5	6
v = ?	v_1 = ?	v_2 = ?	***v***	v_4 = ?	v_5 = ?	v_6 = ?
		v	***v***	*v*	?	?
	v		***v***	*v*	?	?
	v	*v*		*v*	?	?
	v	*v*	***v***		?	?
	?	?	***v***	?		?
	?	?	***v***	?	?	

Regardless of what Lieutenants 5 and 6 claim the general said, in columns 1, 2, 3 and 4, the majority will vote for *v*, producing the vector

General	1	2	3	4	5	6
?	*v*	*v*	***v***	*v*	?	?

and thus, Lieutenant 3 (and all other loyal lieutenants) would deduce that the general issued *v*.

Note that if there was one fewer loyal lieutenant, the disloyal lieutenants could affect the vote, as the majority of a tie goes to a default value, so if the general did not issue the default value, the two disloyal lieutenants could force any number of lieutenants to follow the default.

13.5.2.3.2 Example with seven participants with a disloyal general and a collaborating lieutenant

With seven participants where a tie goes to retreat, if the collaborator reinforced the order that the general sent, we would have the two scenarios. Suppose Lieutenants 1 and 2 were issued the order to retreat. Then, for example, Lieutenant 1 could create the table:

General	1	2	3	4	5	6
?	**R**	?	?	?	?	?
		R	**A**	**A**	**A**	R
	R		A	A	A	R
	R	R		A	A	A
	R	R	**A**		A	A
	R	R	**A**	A		A
	?	?	**A**	?	?	

Tallying up the columns, Lieutenant 1 never-the-less determines that the course of action is to attack.

General	1	2	3	4	5	6
?	**R**	R	A	A	A	A

Lieutenant 3 was ordered to attack, and thus would create the table

General	1	2	3	4	5	6
?	?	?	**A**	?	?	?
		R	**A**	A	A	R
	R		**A**	A	A	R
	R	**R**		**A**	**A**	**A**
	R	R	**A**		A	A
	R	R	**A**	A		A
	?	?	**A**	?	?	

Again, tallying up the columns, we end up with the same result: attack.

If the disloyal lieutenant issued more retreat orders than attack orders, the last entry would become retreat, and thus, the majority of a split decision would go to retreat.

Note: a *collaborating participant* may be representative of two automata that fail in the same way.

13.5.2.3.3 Example with six participants and two disloyal lieutenants

If the general is loyal, he will send the same order to all lieutenants. Consequently, with six participants (five lieutenants), two disloyal lieutenants can now cause the other two lieutenants to not follow the order of the loyal general. For example, suppose the general issues the order A. Lieutenant 3 would create the following table based on information sent from other lieutenants. If both Lieutenants 4 and 5 issue the signal to retreat, now with a tie goes to the default value: retreat.

General	1	2	3	4	5
v = ?	v_1 = ?	v_2 = ?	A	v_5 = ?	v_6 = ?
		A	A	R	R
	A		A	R	R
	A	A		?	?
	R	R	A		R
	R	R	A	R	

Thus, it appears four out five lieutenants received a signal to retreat:

General	1	2	3	4	5
?	R	R	**A**	R	R

and thus, Lieutenant 3 (and all other loyal lieutenants) would deduce that the general issued retreat. The general, however, is attacking and the two disloyal lieutenants are doing nothing. Thus, those loyal did not agree upon a common strategy.

13.5.2.3.4 Example with six participants with a disloyal general and a collaborating lieutenant

With six participants where a tie goes to retreat, if the collaborator reinforced the order that the general sent, we would have the two scenarios. Suppose Lieutenants 1 and 2 were issued the order to retreat. Then, for example, Lieutenant 1 could create the table:

General	1	2	3	4	5
?	**R**	R	A	A	?
		R	**A**	**A**	R
	R		A	A	R
	R	R		A	A
	R	R	**A**		A
	?	?	**A**	?	

Tallying up the columns, Lieutenant 1 never-the-less determines that the course of action is to retreat:

General	1	2	3	4	5
?	**R**	R	A	A	R

Lieutenant 3 was ordered to attack, and thus would create the table:

General	1	2	3	4	5
?	?	?	**A**	?	?
		R	**A**	A	R
	R		**A**	A	R
	R	**R**		**A**	**A**
	R	R	**A**		A
	?	?	**A**	?	

Again, tallying up the columns, we end up with the same result: retreat.

In this case, a disloyal general and a collaborating lieutenant is insufficient to prevent the lieutenants from deciding on a course of action; however, there is only a 1/3 chance that the general will be one of the disloyal participants.

13.5.2.4 At most three disloyal participants

If there are at most three disloyal participants ($d = 3$), we require one further round of consultation. Once Lieutenant 3 finishes the above table, she will forward it to every other lieutenant, and every other lieutenant will forward that table to her. Once again, any entry in the table $v_{i,j,3}$ says "Lieutenant i said that Lieutenant j said that Lieutenant 3 said that the general said v"; however, she already knows what she heard, so she will substitute all of these with the value she received; that is, $v_{i,3,k} \leftarrow \boldsymbol{v_3}$. Similarly, $v_{i,3,k}$ says that "Lieutenant i says that Lieutenant 3 says that Lieutenant k says the general said v"; however, Lieutenant 3 already knows what she heard from Lieutenant k, so once again, these values will be replaced by what Lieutenant 3 actually heard from them; that is, $v_{i,3,k} \leftarrow \boldsymbol{v_{3,k}}$. Any entries that have duplicate indices would not be counted—this is where Lieutenant 3 will reaffirm what he or she has already said.

Then, once again, we calculate majorities:

$$v_{j,k} = \underset{1 \le i < n,\, i \ne j,\, i \ne k}{\text{majority}} \left\{ v_{i,j,k} \right\}$$

followed by

$$v_k = \underset{1 \le j < n,\, j \ne k}{\text{majority}} \left\{ v_{j,k} \right\}$$

which are finally followed by

$$v = \text{majority}\{v_1 \ldots, v_{n-1}\}.$$

13.5.2.5 Generalization and analysis

We can generalize this to a larger number of disloyal participants; however, the cost grows quickly. The number of messages that are passed is $d(n-1)(n-2) = \Theta(dn^2)$ in addition to the $n-1$ initial messages sent by the general. At each step, however, the amount of information that must be passed also increases: initially only 1, then $n-1$ with $d = 1$, and $(n-1)^2$ with $d = 2$, and so on. Thus, this scheme, while guaranteed to ensure that all loyal participants follow the same course of action, is not practical for problems with significant cases of unreliability. In addition, these schemes will only work if there are at least $n = 3d + 1$ participants. If there are only $n = 3d$ participants, the disloyal participants could gather together in a block and behave in a manner similar to that described in the case above where it is impossible to tell if there is one disloyal participant in three.

For example, trying to coordinate 10 robots where each is making measurements of the environment to determine what to do next (here, the environment is the general). As no two robots will take the exact same measurements, the conclusion as to what the course of action that should be followed will differ between the robots. Consequently, here the

environment acts as a *malfunctioning general* that passes possibly different information to different robots. Also, assuming that we must tolerate failure in one robot, we would therefore have to assume that $d = 2$. Therefore, 90 messages must be passed indicating the course of action, followed by another 90 messages of vectors of 10 courses of action.

Alternatively, if only one robot was making measurements of the environment to determine the course of action (that is, the *general* makes the measurements), and that robot sends out its course of action to all other robots, if we must tolerate failure in one robot, then 9 messages must be sent out, and then an additional 72 messages must be sent out between the robots as to which course of action must be followed. If the *general* was malfunctioning, the 9 robots will never-the-less agree upon a single course of action. If the general is functioning, it will follow its course of action and at least 8 of the remaining 9 robots will follow that course of action.

To solve the problem for situations where there is significantly more unreliability, we will look at adding unalterable signatures to each command; that is, a disloyal participant cannot claim that the general said v' when in fact the general said v.

Note: If you read any other description of the Byzantine general's problem, you will note that the description is very different. In those descriptions, it provides a general algorithm, but it is opaque and not clear as to why it works. By placing the values into tables, and describing the purpose of the tables, and by focusing on these tables being sent (as opposed to individual entries), this sheds light on why and how this algorithm works.

13.5.3 A solution using signatures

Suppose that an order can be signed using an unforgeable signature. In our Byzantine example, this could be a wax seal; however, today we could use either

1. digital signatures using public keys (where each participant will, of course, be aware of the public keys of other participants), or
2. specialized functions where each participant has a unique set-up that can generate messages with its signature and validate signatures from other participants.

An order with a signature can itself be signed, and both signatures can be verified. Consequently, we can proceed as follows:

1. We will assume that there are at most m disloyal participants.
2. We no longer will assume that the general can contact all lieutenants; instead, we will only assume that the general can contact at least m other participants.

The general signs an order and sends it out to m other participants. If the general is disloyal, he may send different orders to different participants. Each participant will accept incoming messages, verify the signatures, and if all is good, will track the orders it receives. If the message has $k < m$ signatures, it sign the message and forward it to those tasks that have not yet appeared on the list.

Note, there are means (*m-regular graphs*) that can be used to reduce the number of messages that need be sent.

13.5.4 Application of the Byzantine generals' problem

As an example of where the Byzantine generals' problem is applied is with block chains and Merkle trees and applications include cryptocurrency such as Bitcoin and large distributed databases. These data structures are used in conjuction with cryptographic hashes, and we will examine how the

13.5.5 Summary of Byzantine generals' problem

The Byzantine generals' problem describes a scenario where independent automata must communicate with each other in order to come to a uniform goal in the presence of malfunctioning automata. The situation is reinterpreted in a late-Roman imperial setting during the decay of the eastern Byzantine Empire.

13.6 Summary of fault tolerance

In this topic, we've looked at creating fault tolerant systems. We have discussed redundancy, error detection and correction methods, the Byzantine generals' problem, and the problem of synchronizing clocks. All of these often appear to assume the worst-case scenarios; however, in any actual situation, problems like this may occur once in blue moon; however, even that is quite common if you wait long enough. After all, the Mars rover *Opportunity* was expected to survive for 92½ days—it's been going now for over 4229 days or 46 times its original life expectancy.

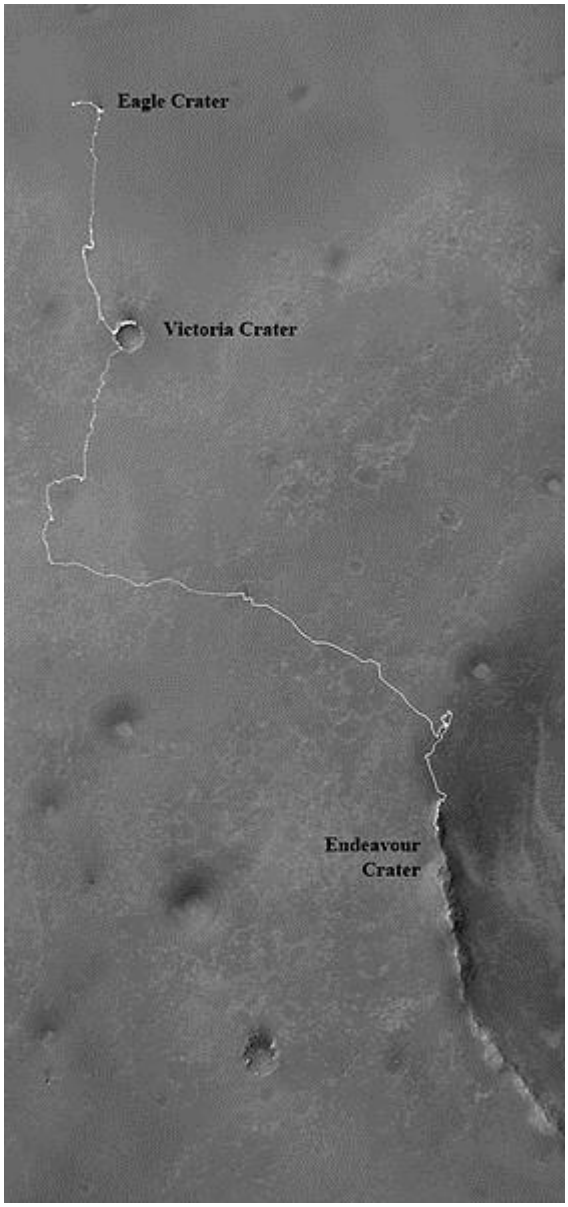


Figure 13-32. A NASA image updated by the first author.

Problem set

13.1 Suppose that we have four processors redundantly calculating the same result. Suppose that each processor has a $p = 0.000001$ probability of calculating an incorrect value. Explain why the formula

$$\binom{4}{2} p^2 (1-p)^2$$

gives the probability that two processors fail. Suppose ten computations are performed each second. What is the estimated wait time between two or more failures? What is the estimated wait time between three or more failures?

13.2 Suppose that if two processors fail, then they have a 0.01 % probability of failing in the same way. How would you calculate the mean wait time between two processors failing in the same way?

13.3 Suppose that the probability of a transmission failure is $p = 0.01$ for each bit that is sent and you need to ensure that there is no greater than a 0.01 % probability of not recognizing that a failure occurred. Do you replicate each bit three times or five times?

13.4 Suppose that the individual probability of a bit being flipped is $p = 0.000000001$ (one in a billion). What is the probability that with a 7-bit message and one parity bit, that an error will not detected?

13.5 Now, if you answered the previous question correctly, you would have determined the probability of there being 2, 4, 6 or 8 errors. Is it really necessary to consider the possibility of 4 or more errors?

13.6 What is the run time of a CRC assuming that the divisor is of fixed size?

13.7 Using the algorithm of the cyclic redundancy check, calculate the remainder of 101010 (equal to 42 in binary) when divided by 1011, and then calculate the remainder of 101010000 when dividing by the same divisor.

13.8 In the previous question, you should have 110 and 001, respectively. Show that dividing 101010001000 when divided by 1011 has a remainder of 0.

13.9 Find the Hamming(7, 4) code for 0001, 0010, 0011 and 0100.

13.10 Suppose you receive the four garbled Hamming(7, 4) codes

```
0100100 1010011 0110111 0001100
```

What was the original message (as two ASCII characters)?

13.11 If you add to the Hamming(7, 4) code one additional bit that ignores the 4th bit, you now have an 8-bit code that can correct an error in one bit, and detect (but not correct) if there are errors in two bits where:

1. if the eighth parity bit is correct, then first seven bits can be either accepted as correct or corrected as described above; however,
2. if the eight parity bit is incorrect, the first seven bits must match a Hamming code exactly.

In the second case, if the bits to not match exactly, flag that an error has been detected but not corrected.

13.12 In the previous example, we see that we can send $4n$ bits using $8n$ bits to correct at most one error per four bits, and detect at most two errors per four bits. Suppose that the probability of an error in any one bit is 0.000001 (one in one million). What is the probability that n messages will get through without two errors appearing in any byte? Your answer will be some number raised to the power n. What is that probability if $n = 32$?

13.13 Suppose you have four distributed clocks and their times are designed to have an error no greater than 0.1 s between synchronizations and they pass between each other the four times

10.26 10.32 10.35 10.41

What are the four times after the synchronization? Assuming only one clock is defective, can you tell what it is?

13.14 Suppose you have the same question as before, but you now have the four times

10.24 10.32 10.35 10.41

What are the four times after the synchronization? Assuming only one clock is defective, can you tell what it is?

13.15 Suppose that a clock receives the following three pairs of messages (t, C_k) where t is the time according to this clock when the message is received, and C_k is the time sent by one of three clocks. Suppose also that $\tau = 0.2$ and $\varepsilon = 0.5$. After the third message is received, by how much does the current clock have to be modified?

(1232.42, 1232.47), (1401.22, 1400.91), (1543.92, 1544.25)

13.16 If you were new to a project, and you saw the time stamps used in the above synchronization, what question might you want to ask?

13.17 Why do you not simply want to update the time of the current clock when you calculate the synchronized time?

13.18 Advanced question: Why not just ignore clocks that are too out of synch? Why do we replace the times of clocks that are further away than ε with the time of this clock when making our calculation?

13.19 Show how two tasks receiving instructions from a third task can never be guaranteed to agree on an interpretation of that message if one of the three is defective.

13.20 Suppose that you have four tasks and one task is required to send a message to the other tasks. Under the assumption that no more than one task is malfunctioning, if it is necessary that all functioning tasks receive the same message, show how this can be solved strictly through message passing.

13.21 Suppose that you have seven tasks and one task is required to send a message to the other tasks. Under the assumption that no more than two tasks are malfunctioning, if it is necessary that all functioning tasks receive the same message, show how this can be solved strictly through message passing.

13.22 Suppose that you have a high certainty that individual tasks will not fail; however, the communications channel is noisy and therefore communications are subject to corruption. Which scheme described in this topic would you choose for your system, and why?

14 Operating systems

Throughout this course, we have discussed a number of issues relating to resources:

1. the processor is a resource,
2. semaphores or other synchronization tools are virtual resources that can be *shared* by tasks, and
3. other peripherals are resources

all of which may be used by tasks or threads to achieve their goals. To date, we have simply considered these resources as being available to all tasks and threads, and one could restrict access to those resources through semaphores and mutual exclusion.

Unfortunately, as systems become more complex, it becomes more difficult to ensure that all code is bug-free, and one task may accidently attempt to access a resource being used by another. For small projects, this is unlikely, as a small group of developers can often keep the entire system in their mind's eye. However, as projects become larger, different development groups will be tasked to work on separate components, and there will be the inevitable conflicts. Thus, these are usually grouped together in a piece of software known as an *operating system*.

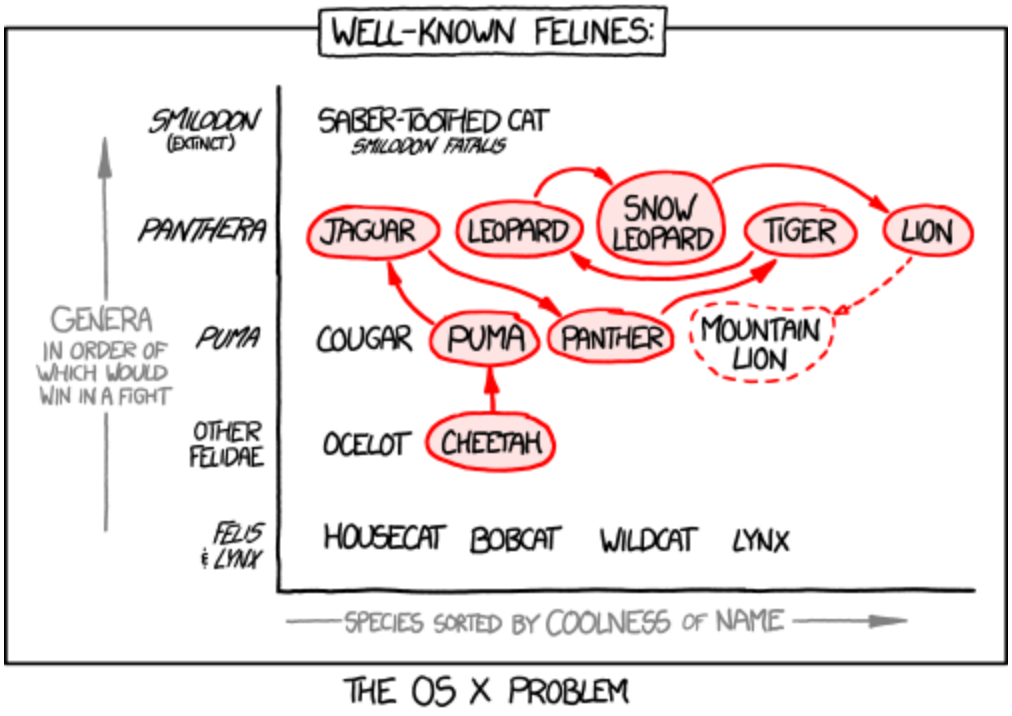


Figure 14-1. A humorous cartoon from XKCD (http://xkcd.com/1056/), reproduced for academic purposes.

14.1 Operating systems as resource managers

One means of solving this problem is to factor out all resource management, hence all tasks and threads can only access the resources though a common interface. That interface can then ensure that a resource is dedicated to one task or thread, only.

The benefit is clear: each individual task or thread is not dealing with resource management directly—all of this is dealt with through the common interface. The drawback of this approach is that there is inevitably more overhead—it is no longer possible to immediately access any resource and consequently, the system will be slower.

Unfortunately, even this is insufficient in any large project: recall how with half-fit and other algorithms, the linking data structures were stored in the memory location immediately prior to memory location allocated? What happens if a user accidently assigned to memory location `array[k]` where `k` happened to have been a negative number or `k` pointed to a memory location beyond the allocated memory?[38] In this case, the user would corrupt the corresponding data structure, not only for the memory that was allocated to it, but also for memory allocated to other tasks.

There is always the other issue that even if all tasks nominally access all resources through a common resource-management interface, it may happen that a programmer will, never-the-less, make a mistake. Perhaps an exception will not be caught, or there is always the temptation to bypass the interface for critical systems. This temptation will always be there, and sooner or later, a developer short on time and resources, will yield to it.

14.2 Processor modes

In order to prevent such cheating, this ultimately requires processor support: it must be possible for the processor to prevent users from issuing instructions that access resources. This can only be done with hardware support. The standard mechanism these days is to allow the processor to have two modes:

1. kernel mode, and
2. user mode.

In kernel mode, the processor can execute all instructions, while in user mode, the processor can only execute instructions associated with processor operations and not with other interfaces such as any of the various buses the microcontroller may be attached to. Thus, each task is run in *user mode*, and all resource management is executed in *kernel mode*. In addition, in order to protect the integrity of the resource management tools (now running in kernel mode), we will block off a section of memory that can only ever be accessed in kernel mode: we will call this section of memory *kernel space*. Any data structures related to resource management will be described as *kernel data structures* and these will be allocated in kernel space.

The question is: how do you switch from user mode to kernel mode? If this is nothing more than an instruction, could not a tired and overworked programmer simply execute such an instruction and then give him or herself access to all instructions and all memory?

To solve this problem, again, it requires hardware support. Recall that with an interrupt, the interrupt handler determines which interrupt service routine (ISR) needs to be called? The peripheral issuing the interrupt does not decide this; this is entirely decided by the programmer of the microcontroller.

Suppose we could similarly restrict the commands that a user can run. In fact, the approach is identical, and the mechanism is even called a *software interrupt*: a request to execute specific code.

Like the handling of hardware interrupts, software interrupts are implemented as follows:

1. The programmer of the resource manager writes code that is assumed to execute in kernel mode.

[38] You may have run into this if you have ever implemented a circular array for a queue data structure. It is quite easy to accidently find yourself assigning to memory locations `array[-1]` or `array[ARRAY_SIZE]`.

2. Each interface function is given a unique number, in 0 to $n - 1$.
3. The kernel tracks a software interrupt vector (or *trap table*) storing the addresses of these n functions.
4. When the user wants to call one of these functions, instead of calling a function as per normal, instead, the arguments are placed in the correct location, and then a request is made for a software interrupt with the corresponding number.

When this software interrupt, or *trap*, is called, the processor:

1. switches to kernel mode,
2. finds the address in the corresponding entry of the software interrupt vector, and
3. calls that function.

When that function exits, the processor mode is switched back to user mode. Thus, regardless of how disgruntled or tired our programmer is, he or she cannot execute their own commands in kernel mode: they can only call specific instructions that have been predefined by the developers of the operating system.[39]

Another name for issuing a software interrupt is to make a *system call*. An example of how a call to `fork()` is translated into a system call through a software interrupt, with `sys_fork()` executing in supervisor mode.

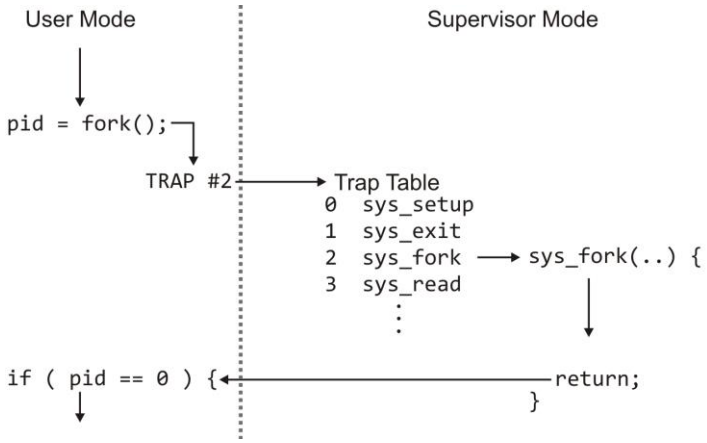


Figure 14-2. Execution of a software interrupt in Unix.

In Linux, the trap table is large and growing. The first seventeen entries are:

```
00 sys_setup
01 sys_exit
02 sys_fork
03 sys_read
04 sys_write
05 sys_open
06 sys_close
07 sys_waitpid
08 sys_creat
09 sys_link
10 sys_unlink
11 sys_execve
12 sys_chdir
13 sys_time
```

[39] That is, until he or she gets access to the kernel itself. ☺

```
14 sys_mknod
15 sys_chmod
16 sys_lchown
```

The *kernel* of an operating system are all functions that are persistent in memory.

To make a system call in Linux, there are actually multiple ways:

```
#include <syscall.h>
#include <stdio.h>

int main( void ) {
        long pid_1, pid_2, pid_3;

        pid_1 = syscall( SYS_getpid );
        printf( "syscall( SYS_getpid ) = %ld\n", pid_1 );

        pid_2 = syscall( 39 );
        printf( "syscall( 39 ) = %ld\n", pid_2 );

        pid_3 = getpid();
        printf( "getpid() = %ld\n", pid_3 );

        return 0;
}
```

The output of this piece of code is

```
syscall( SYS_getpid ) = 30530
syscall( 39 ) = 30530
getpid() = 30530
```

14.2.1 Multiple processor modes

Some processors will have three modes:

1. user mode,
2. device driver mode, and
3. kernel mode.

The intermediate mode is for user-authored device drivers that allow access to a slightly larger range of instructions than user mode. The assumption is that this will be used only for those functions meant to interface with peripheral devices.

14.3 Memory management

One issue we have not yet discussed is the problem with memory. In very large systems, the operating system could issue separate processes different segments of memory. Rule 10 of the JPL coding standard states that

> "Where available, i.e., when supported by the operating system, memory protection shall be used to the maximum extent possible. When not available, safety margins and barrier patterns shall be used to allow detection of access violations."

We will look at memory protection by seeing an example in the Cortex-M3 and we will then discuss safety margins and barrier patterns.

14.3.1 Memory protection unit

As an example of memory protection, we will look at the Cortex-M3 Memory Protection Unit (MPU). This is an optional package that allows the programmer to

1. allocate memory accessible only in kernel mode (for use only by the operating system),
2. allocate memory for tasks that is not accessible by other tasks,
3. designating regions of memory as read-only, and thereby protecting critical data, and
4. detecting invalid accesses to memory, such as when a stack exceeds its limit.

The MPU will signal an exception that can then potentially be handled. Each region is set up by writing the appropriate values to memory-mapped registers at `0xe000ed90`. The significance of these twenty bytes is shown in Figure 14-3.

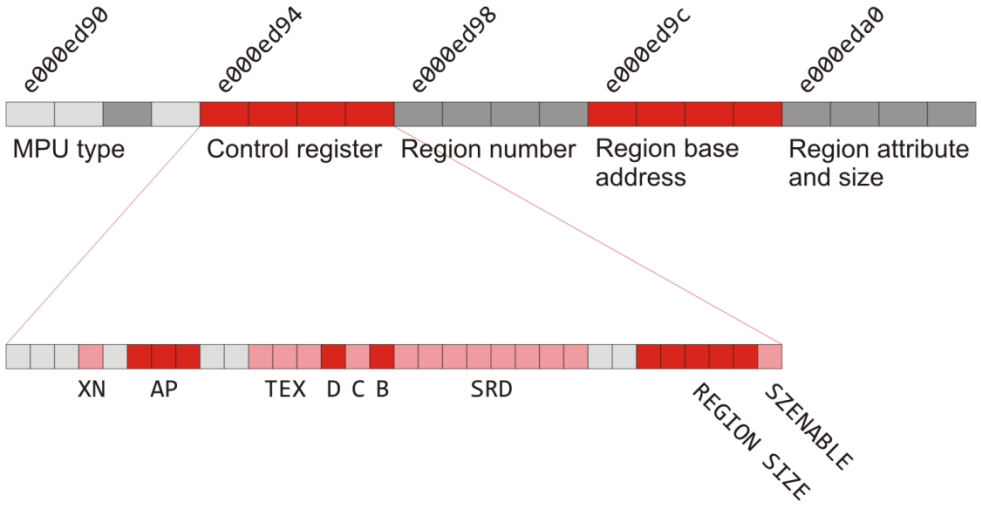


Figure 14-3. Memory-mapped registers of the MPU.

14.3.2 Safety margins and barrier patterns

If memory protection cannot be used, it may be necessary to have explicit spaces between allocated memory, and those spaces will be filled with a fixed pattern. A reasonable pattern to use would be one that is neither all 0s nor all 1s, but which has an easily recognizable visual pattern for a human reader when reading a core dump. Perhaps the one most recognizable to programmers (who tend to consume a significant amount of caffeine) is the 16-bit number 51966, which is `cafe` as a hexadecimal number, although a palindromic sequences of hexadecimal characters would also work; for example, `abba` and `bdbdbdbd`. During testing, and even in deployment, the idle task or thread could be tasked with continually verifying that all safety margins continue to hold these patterns, and to take corrective action if any of the barrier patterns have been modified.

14.4 Microkernels

The size of kernels grew over time as more and more tasks were included in it. For Unix, this began with the Berkeley Software Distribution that included device, file and network management in the kernel. To date, Linux keeps this model. More recently, there has been a move to minimize the size of the kernel by having most of the additional functionality run in user mode as servers. Thus, the kernel itself has a significantly smaller footprint and faults in any of the servers cannot affect the functioning of the microkernel. To contrast microkernels with traditional larger kernels, the latter are referred to as *monolithic* kernels. With the exception of **QNX**, most real-time operating systems are monolithic.

14.5 Real-time operating systems

A real-time operating system is one that guarantees maximum response times for functionality such as responding to interrupts and scheduling. Traditional general-purpose operating systems have average response times, but they do not have guaranteed response times, consequently making them inappropriate for hard real-time systems.

Now, in some cases, the guarantees may be relative to the size of your system. Many of the dynamic memory allocation schemes we looked at run in linear time with respect to, for instance, the number of blocks of memory that are deallocated. If this size is guaranteed to be small, then dynamic memory allocation is guaranteed to be fast; however, this requires that the engineer designing the system know these restrictions and know how to design an appropriate system.

14.6 Examples of real-time operating systems

We will look at a number of real-time operating systems, including:

1. the Keil RTX RTOS,
2. the CMSIS-RTOS RTX,
3. FreeRTOS, and
4. Wind River's VxWorks, and
5. BlackBerry **QNX**.

The term *RTX* is an abbreviation for *R*eal-*t*ime e*x*ecutive. There is an RTLinux (real-time Linux) which implements a microkernel which then runs the Linux kernel as a pre-emptible process and RTSJ (the real-time specification for Java). We will only do a brief overview to allow you view the similarities and differences between the various real-time operating systems.

14.6.1 The Keil RTX RTOS

The Keil RTX RTOS is a real-time operating system that is reasonably stripped down. It does not have a large footprint and it is summarized on the Keil web site with the image shown in Figure 14-4.

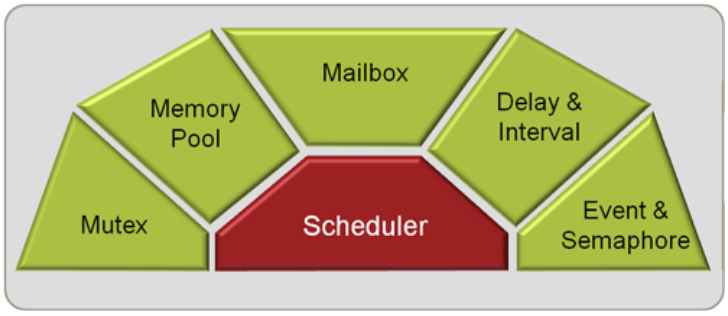


Figure 14-4. The Keil high-level representation of their RTOS(http://www.keil.com/rl-arm/kernel.asp), reproduced for academic purposes.

The executable kernel is guaranteed to be less than 4 KiB. As for the footprint in main memory, we have the following:

Feature	Footprint in main memory
kernel space	Less than 300 bytes with 128 bytes for a user stack
overhead for each task	52 bytes plus the stack size
overhead for a mailbox	16 bytes plus 4 bytes for the maximum number of messages
semaphore	8 bytes
mutex	12 bytes
user timer	8 bytes

The only hardware requirement for the RTX real-time operating system, in addition to the microprocessor, is the `SysTick` timer. The `RTX_Conf_CM.c` file allows the programmer to configure parts of the RTX, including:

1. the maximum number of concurrently running tasks,
2. the size of each task's call stack (for those that are to be allocated by the RTX),
3. whether to check for stack overflow,
4. the number of tasks that will have user-provided call stacks, and
5. whether the tasks should be run in kernel (*privileged*) mode or not.

and if round-robin task switching is to be used, the number of ticks a task will execute before the scheduler is called (the system allows between 1 and 100 ticks). The libraries available deal with

1. synchronization and message passing,
2. memory management,
3. task management, and
4. time management.

14.6.1.1 Synchronization and message passing

There are five packages related to synchronization and message passing:

1. event flags,
2. mailbox,
3. mutex, and
4. semaphores.

The counting semaphores have type `OS_SEM` and the semaphore package includes four functions:

Function signature	Comments
`void os_sem_init( OS_ID, U16 tokens )`	Initialize the semaphore with the given number of tokens.
`OS_RESULT os_sem_send( OS_ID )`	Posts to a semaphore with the return value always being `OS_R_OK`.
`OS_RESULT os_sem_wait( OS_ID, U16 timeout )`	Wait on a semaphore, blocking the task on that semaphore if no token is available. It will wait a maximum of timeout system intervals with `0xFFFF` indicating an indefinite timeout period. The return value indicates what happened: 1. `OS_R_SEM` indicates the task waited until a semaphore became available, 2. `OS_R_TMO` indicates that a timeout occurred prior to a token becoming available, and 3. `OS_R_OK` indicates that function returned immediately.
`void isr_sem_send( OS_ID )`	For posting to a semaphore from within an interrupt service routine (ISR).

As an example of the use of a semaphore:

```
#include <rtl.h>

OS_SEM mtx;

int main( void ) {
    os_sem_init( &mtx, 0 );

    while ( 1 ) {
        // ...
    }
}

__task void tsk1( void )  {
    // ...
    os_sem_wait( &mtx, 0xffff );
    // ...
}

__task void tsk2( void )  {
    // ...
    os_sem_send( &mtx );
    // ...
}

// SysTick interrupt service routine
void SysTick_Handler( void )  {
        isr_sem_send( &mtx );
}
```

The mutual exclusion data type is `OS_MUT`, but has only three functions, as one would not acquire or release mutual exclusion while servicing an interrupt. The task that acquired mutual exclusion must also be the task that releases it, and priority inheritance is used to solve the priority inversion problem.

Function signature	Comments
`void os_mut_init( OS_ID )`	Initialize the mutual exclusion.
`OS_RESULT os_mut_release( OS_ID )`	`Releases a mutex with a return value` 1. `OS_R_OK` if the release was successful, and 2. `OS_R_NOK` if either the mutex was not acquired or if the task is not the one that acquired the mutex.
`OS_RESULT os_mut_wait( OS_ID, U16 timeout )`	Wait on a mutex, blocking the task on that mutex if it is unavailable. It will wait a maximum of timeout system intervals with `0xFFFF` indicating an indefinite timeout period. The return value indicates what happened: `OS_R_MUT` indicates the task waited until the mutex became available, `OS_R_TMO` indicates that a timeout occurred prior to the mutex becoming available, and `OS_R_OK` indicates that function returned immediately.

As an example of the use of a mutual exclusion:

```
#include <rtl.h>

OS_MUT mtx;

int main( void ) {
    os_mut_init( &mtx );

    while ( 1 ) {
        // ...
    }
}

__task void tsk( void )  {
    // ...
    os_mut_wait( &mtx, 0xffff ); {
        // Critical region...
    } os_mut_release( &mtx );
    // ...
}
```

A mail box is declared through a specific macro:

```
// Declare a mailbox for n messages
os_mbx_declare( mailbox_id, n );
```

Like semaphores, the mailbox has functions that are to be used in interrupt service routines, but the behavior is appropriately modified for that purpose.

Function signature	Comments
`OS_RESULT os_mbx_check( OS_ID )`	This returns the number of messages that can still be added into the mailbox. `OS_RESULT` is an appropriately sized unsigned integer.
`void os_mbx_init( OS_ID, U16 capacity );`	Initialize a mailbox with a capacity of `capacity` bytes.
`OS_RESULT os_mbx_send( OS_ID, void *msg, U16 timeout );`	Send a message, blocking the task if the mailbox is full. It will wait a maximum of timeout system intervals with `0xFFFF` indicating an indefinite timeout period. The return value indicates what happened: `OS_R_TMO` indicates that a timeout occurred prior to the message being sent, and `OS_R_OK` indicates that the message has been put in the mailbox.
`OS_RESULT os_mbx_wait( OS_ID, void **msg, U16 timeout );`	Receive a message from the mailbox if it is not full. It will wait a maximum of timeout system intervals with `0xFFFF` indicating an indefinite timeout period. The return value indicates what happened: `OS_R_MBX` indicates that a message was received, but the task was first put to sleep, `OS_R_TMO` indicates that a timeout occurred prior to a message being received, and `OS_R_OK` indicates that a message was available when requested and the task was not put to sleep.

To be discussed further: event flag management, memory allocation management, system functions, task management, time management and user timer management.

14.6.2 The CMSIS-RTOS RTX

The CMSIS-RTOS RTX is a lightweight real-time operating system that is implemented on all Cortex-M3 processors.

```
#define osFeature_MainThread      [01]         Does main run as a thread?
#define osFEATURE_SysTick         [01]         Is the kernel system timer
                                               available?
#define osCMSIS                   0x10002      Version of CMSIS-RTOS RTX
#define osKERNELSYSTEMId          KERNEL       RTOS identification string
                                  V1.00
#define                           100000000    RTOS system timer frequency in Hz
osKernelSysTickFrequency
#define                           100          Previous value divided by 106
osKernelSysTickMicroSecond

osStatus osKernelInitialize( void )    Initialize the kernel
osStatus osKernelStart( void )         Start the kernel
int32_t osKernelRunning( void )        Check if the kernel has been started
uint32_t osKernelSysTick( void )       Returns the kernel system counter.
```

The priorities of threads is much more restricted than in the Keil RTX RTOS, with seven priorities. To be continued with thread management, generic wait functions, timer management, signal management, mutex management, semaphore management, memory pool management, message queue management, mail queue management, generic data types and definitions, and status and error codes.

14.6.3 FreeRTOS

FreeRTOS is an open-source freely available real-time operating system. There are two approaches to multitasking, regular tasks and light-weight co-routines:

1. tasks that have four states: READY, RUNNING, BLOCKED and SUPSENDED, while
2. co-routines have only the first three states.

Both tasks and co-routines can be assigned priorities, but co-routines have lower priority than any task. All co-routines share the same stack—consequently, if a co-routine is blocked, only static local variables maintain their values.

For inter-process communication, there are five data structures:

1. queues for message passing,
2. binary semaphores,
3. counting semaphores,
4. mutexes (a binary semaphore where the task holding the mutex must be the one to post to it), and
5. recursive mutexes.

The latter is a mutex where the same task can recursively wait on it, incrementing a counter. That task must then issue a post for each wait.

As well, we have already discussed the five dynamic memory allocation implementations that come with FreeRTOS. FreeRTOS has a site describing Running the RTOS on an ARM Cortex-M Core.

14.6.4 Wind River's VxWorks

Wind River's VxWorks is the most used real-time operating system today. It is running on Earth, in space and on Mars including the Sojourner, Spirit, Opportunity and Curiosity Mars rovers, the SpaceX Dragon, the Boeing 787 Dreamliner,

the Boeing AH-64 Apache, the James Webb space telescope, as well as many more applications in automotive, consumer electronics, industrial robots, transportation, controllers, and other applications.

14.6.5 Blackberry's QNX

QNX is a mature real-time operating system using a microkernel approach where most operating system tasks run as separate servers, the kernel itself contains scheduling, inter-process communication, interrupt servicing and timers. The benefit is that if any of these servers fail, the failure cannot affect the running kernel and the server need only be restarted. It was developed by Gordon Bell and Dan Dodge, former students of the University of Waterloo.

We have previously discussed various states that tasks can be in, but grouped numerous related states as simply being `BLOCKED`. **QNX** has twenty states; these descriptions are taken from the *Get Programming with the QNX Neutrino RTOS* guide, reproduced here for academic purposes:

	`STATE_READY`	Not running on a processor but is ready to run
	`STATE_RUNNING`	Actively running on a processor
	`STATE_NANOSLEEP`	Sleeping for a period of time
	`STATE_DEAD`	Terminated, but the kernel is waiting to release the resources
BLOCKED	`STATE_CONDVAR`	Waiting for a condition variable to be signaled
	`STATE_INTR`	Waiting for an interrupt
	`STATE_JOIN`	Waiting for another thread to complete
	`STATE_MUTEX`	Waiting to acquire a mutex
	`STATE_NET_REPLY`	Waiting for a reply to be delivered across the network
	`STATE_NET_SEND`	Waiting for a pulse or message to be delivered across the network
	`STATE_RECEIVE`	Waiting for a client to send a message
	`STATE_REPLY`	Waiting for a server to reply to a message
	`STATE_SEM`	Waiting to acquire a semaphore
	`STATE_SEND`	Waiting for a server to receive a message
	`STATE_SIGSUSPEND`	Waiting for a signal
	`STATE_ISGWAITINFO`	Waiting for a signal
	`STATE_STACK`	Waiting for more stack to be allocated
	`STATE_WAITCTX`	On SMP systems, waiting for a register context to become available
	`STATE_WAITPAGE`	Waiting for the process manager to resolve a fault on a page
	`STATE_WAITTHREAD`	Waiting for a thread to be created

The services available in **QNX** include

1. threads and processes,
2. scheduling,
3. synchronization,
4. clock and timer services, and
5. interrupt servicing.

QNX has numerous synchronization tools available:

1. mutual exclusion locks (`mutex`) allowing only the thread acquiring the lock to unlock it and providing priority inheritance to avoid the priority inversion problem,
2. condition variables (`cond`) that can be waited on inside a critical region protected by mutual exclusion,
3. barriers (`barrier`) that prevent threads from continuing until all threads reached the barrier,
4. sleep-on locks (`sleepon`), a variation on condition variables,
5. reader-writer locks (`rwlock`) that implement a solution to the multiple-reader–single-writer problem, and
6. semaphores.

Their documentation also points out that mutual exclusion can be achieved through using a FIFO scheduling policy where tasks of the same priority continue executing until they voluntarily release the processor. If an interrupt or higher priority thread becomes ready and the current thread is placed back into the ready queue, it is placed at the start of the ready queue for its priority. This is described as FIFO scheduling in **QNX**. **QNX** also provides the programmer with a library of atomic operations, including

1. adding or subtracting a value, and
2. clearing, setting or toggling bits.

The clock services include the following functions:

`int ClockTime( clockid_t, … )`	Get or set a clock.
`int ClockAdjust( clockid_t, … )`	Adjust the time of a clock.
`uint64_t ClockCycles( void )`	Get the number of clock cycles.
`int ClockPeriod( clockid_t, … )`	Get or set a clock period.
`int ClockID( pid_t, int )`	Get the CPI-time clock ID for a given process and thread.

14.7 Summary of operating systems

Resource management in a small system can be dealt with by the individual tasks and threads; however, as systems grow in size, the management of resources becomes much more difficult; consequently, it is often appropriate to combine all aspects of resource management (including scheduling) into a single package termed an operating system. In order to protect this operating system, it is often run in a protected mode on a processor which gives it exclusive access to certain memory and instructions, thereby preventing any user tasks or threads from inadvertently or purposely affecting the integrity and stability of the system. Memory management can also be implemented to protect regions of memory from other tasks. Now, over time, the size of those instructions sets have grown, allowing every more functionality and control by the programmer.

Problem set

14.1 How is it guaranteed that the introduction of a kernel mode together with a software interrupt mechanism allows any data structures and operations executed under kernel mode protect them from even malicious users.

14.2 What differentiates a real-time operating system from a general-purpose operating system?

14.3 Does every component of a real-time operating system have to run in $\Theta(1)$ time? What is required if an algorithm cannot be implemented so that it runs in $\Theta(1)$ time (and therefore runs in $\omega(1)$ time)?

14.4 Suppose we developed an operating system that used:

1. half fit for dynamic memory allocation, and
2. a priority scheduler using a fixed sized priority queue.

How would you explain the limitations to a customer?

14.5 Suppose we developed an operating system that used:

1. best-fit for dynamic memory allocation, and
2. an earliest deadline first scheduler using a node-based leftist heap.

How would you explain the limitations to a customer?

15 Software simulation

We have discussed how real-time systems are meant to interact with the physical world, but one means of validating and informally verifying that our system functions correctly is to test the software under simulated conditions. The question is, what is the best way to simulate reality? We model reality mathematically. In some cases, it might be quite straight-forward: the borders of a room may be solid, and a robot *bumps* into a wall (signals a sensor) whenever the coordinates, velocity and orientation of the robot indicate that it has made contact with the wall and the force of the contact will be relative to the velocity (speed and angle of contact).

The purpose of software simulation is associated with the non-functional requirement of safety: can we validate the system that was designed actually solves the engineering problem at hand. While the requirements are being authored, certain invalid assumptions may be made about the system that make the final product invalid, but without simulation, these can only be tested under actual conditions. The Soviet early warning satellites were designed to detect the launch of American Intercontinental Ballistic Missiles (ICBMs) from silos in the midwestern United States. On September 26, 1983, the sytem failed: a rare alignment of sunlight on high-altitude clouds above North Dakota with the orbit of the satellites was interpreted as a launch, and it was only the human intervention of the duty officer, Stanislav Petrov, that prevented a nuclear retaliation by the Soviety Union.

However, how do you model, for example, a slippery floor? How do you model random processes? We will discuss some of these here.

15.1 Physics engines

A *physics engine* is a popular term to describe a system for simulating physical realities. These will approximate reality using differential equations and then simulate reality by solving these systems of equations using initial-value problem solvers and partial-differential equation solvers.

ENIAC, one of the first general-purpose computers, was used at times as a physics engine when it would, for example, simulate artillery shells to create ballistics tables. See W.T. Moye, "ENIAC: The Army-Sponsored Revolution", US Army Research Laboratory, 1996.

15.2 Modelling client-server systems

Suppose you are modelling a server and your clients are acting independent of each other—that is, the time at which one client requests a service does not affect the times at which other clients request a service. Suppose you're a bank manager and you want to determine how many tellers should be working on a Saturday afternoon. You know from past experience that you get about 80 clients per hour and you know that clients get frustrated if the queue is longer than four people. Now, if each server takes an average of two minutes to service each request, then two tellers will not be able to keep up with the demand (they will have only serviced 60 clients after one hour and another 20 will be left waiting). If you have three tellers, they will be able to provide an appropriate level of service *on average*, but what about worst-case situations? Now, we must make some assumptions.

Let us start by assuming that requests (clients) are dealt with on a FCFS basis. Suppose, for example, three clients come within a minute of each other and each has a request that will take 7 minutes to complete. In this case, you still expect, on average, 8 clients to arrive in the next six minutes, and they will all be waiting in a reasonably long queue. What is the probability of something like this occurring? Should you have five tellers on hand to guarantee this will never happen? Is it worth it to have so many tellers on hand at all times on a Saturday afternoon if most of their time is spent twiddling their thumbs?[40]

[40] There is a saying among managers that "work expands to fill the time available"—it might be difficult to tell when you are overstaffed.

To analyze and/or make predictions about this scenario, we must have a model. Very often, you have a situation where one group of tasks is requesting a service, while other tasks are servicing those requests. Such a scenario is modelled by a *client-server model*. We will look at such a model and describe a mathematical model of the scenario.

A client-server model has clients arriving and waiting on requests, and servers satisfying those requests. While this model describes conceptually what is occurring, we must now find a mathematical model of the clients and servers. The purpose of the mathematical model is to be able to

1. construct simulations that are reasonably true to reality (assuming our model is correct), and
2. determine what we can say statistically about the expected behavior.

We will assume that the long-term behavior of both the arrival of the clients and the number of clients serviced is constant:

1. the average arrival rate (clients arriving per unit time) will be represented by λ, while
2. the average service rate (clients served per unit time) will be represented by μ..

Both must have the same units for the following analyses. Usually such data will be obtained from long-term observation and their value may depend on other factors such as the time of day. We will first begin by defining the load factor of the system, followed by Little's theorem that relates the number of individuals being serviced (including those waiting), the mean arrival rate and the mean time waiting to be serviced (including any time spent waiting). Finally, we will see how we can describe systems based on whether arrival rates or processing rates are deterministc or independent and thus random.

15.2.1 *Load factor*

Given a single server, the load factor is the arrival rate λ over the service rate μ; that is,

$$\rho = \frac{\lambda}{\mu} \ .$$

If we have more than one server, say n, the load factor is previous ratio divided by n: the work can be distributed between the servers:

$$\rho = \frac{\lambda}{n\mu} \ .$$

In either case, if the load factor $\rho > 1$, as you may suspect, there are more arrivals per unit time than there are clients being serviced in that time; consequently, you will always get a backlog of requests and the queue storing these requests will grow indefinitely over time. On the other hand, if the load factor does not exceed 1 but arrivals are independent and thus seemingly random, it is still likely that when a new client arrives, an existing client is still being served. Consequently, if there is any random component to either the arrival rates or the service rates, it will always be necessary to have to temporarility place some requests in a queue until a server becomes available.

15.2.2 Little's theorem

Given an average arrival rate of λ, the number of the clients N either being serviced or waiting to be served equals the arrival rate multiplied by the time T the clients wait being served together with any time waiting to be served. That is,

$$N = \lambda T.$$

If there are n servers, then there will be $N - n$ clients waiting to be served. If the arrival rate increases and nothing is done to reduce the time spent waiting, the total number of individuals waiting will increase proportionally to the increase in the arrival rate.

15.2.3 Modelling real-world arrival and service rates

In some cases, the arrival rates may be periodic and well defined: two vehicles pass a particular point on the assembly line every minute. In others, such as clients visiting a web page, the arrival rate may appear random; however, just as we can describe the randomness of, for example, the roll of a die, we can also describe the behavior of randomly arriving clients. Similarly, the processing rate may be fixed: 10323 simple web requests can be serviced every minute, versus variable processing rates where larger web pages take proportionately longer to service.

We will look at two common models that can possibly describe the arrival rates and service times:

1. deterministic, and
2. Markovian.

We will describe these here and use these in the next section.

15.2.3.1 Deterministic rates

If the arrival rate is periodic, that is, new requests for services arrive with an exact interval between each arrival, we will call that behavior deterministic. We will represent a deterministic arrival rate by the letter "D". If the time between arrivals is ℓ, the arrival rate is $\lambda = 1/\ell$.

If each service requires the exact same amount of time, this translates to a fixed service rates. If each service requires m units of time, the service rate is $\mu = 1/m$. We will also represent this by "D".

15.2.3.2 Markov rates

Given aperiodic requests for service where, on average, there are λ arrivals per unit time, but each arrival is independent of the other, we will say that the arrival rate is *Markovian*.

For example, a population of M robots may on average be observed to have a failure rate of 3 failures per day. Consequently, the repair shop will expect to see $\lambda = 3$ robots per day. However, there may be some days where only one or two robots fail (or possibly even none), while on other days, four or five or more may fail. It is only over the long term that it has been observed that approximately three robots fail per day. If the failures are independent of each other (for example, due to parts wearing out, as opposed to two robots crashing into each other), we will represent such Markovian arrival times by the letter "M".

If, on average, it takes $1/\mu$ units of time to service a request, but the probability of finishing a service in the next Δt units of time is approximately proportional to Δt (I'm twice as likely to be finished with this service task in the 12 hours as I am to be finished within the next 6 hours), we will also describe such a service rate as Markovian and also represent it by the letter "M".

For example, in repairing a robot, suppose I have been working on it for one hour. Each repair has a fixed number of steps, and in general, I will be twice as likely to be finished in two hours as I am after another hour. Now, this is only an approximation, and may not always apply; however, it is usually a reasonable mathematical model.

15.2.4 Describing client-server setups

We may now describe the scenario as having either deterministic or Markovian arrival rates and deterministic or Markovian processing time with *c* servers. We use the notation

$$P/Q/n$$

where *P* describes the arrival rate (deterministic or Markovian), *Q* describes the service rate (again, deterministic or Markovian), and *n* is the number of servers; however, we will only look at three of these cases: D/D/*n*, M/D/1 and M/M/1. The letter *c* is used by convention.

15.2.4.1 D/D/n

If the load factor does not exceed 1, no tasks will ever be queued. Each request will be serviced as it arrives.

15.2.4.2 M/D/1

This is more usual when interacting with independent clients: arrivals are Markovian but service time is deterministic. We will only consider the case with one server. In this case, even when the load factor is less than one, it may be possible that the server is busy when a client arrives. Statisticians have determined various characteristics of such a system. For example, the mean length of the queue is $\frac{\rho^2}{2(1-\rho)}$, and a plot of this value is shown in Figure 15-1.

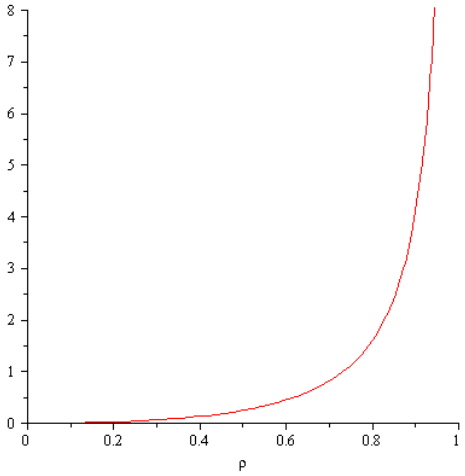


Figure 15-1. A plot of the average queue size given the load factor ρ.

The average size of the queue exceeds 1 when the load factor exceeds $\sqrt{3}-1 \approx 0.732$. The standard deviation of the average queue size is

$$\frac{1}{1-\rho}\sqrt{\rho-\frac{3\rho^2}{2}+\frac{5\rho^3}{6}-\frac{\rho^4}{12}},$$

so if the queue size is not to be exceeded with a confidence of 99 %, we must ensure the queue size is the expected queue size plus 2.326 times the standard deviation (this is because the integral of a standard normal from –2.326 to 2.326 is approximately 0.99). A plot of this is shown in Figure 15-2, and we see that if the queue size is 8, we cannot have a load factor greater than 0.817.

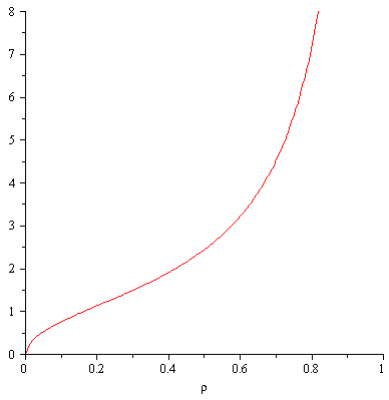


Figure 15-2. An upper bound on the queue size for M/D/1 with 99 % confidence.

In a real-time system, suppose the clients are sensors that must relay information to a central server. If a sensor is queued, the server can signal the sensor for information; however, if the sensor request is ignored because the queue is full, we have two options:

1. signal the sensor that it must attempt to repeat the request at a future point, or
2. that data is lost.

The appropriate course of action will be situation dependent.

We can determine other parameters: the average time spent in the queue is the average length of the queue times the inter-arrival time; that is,

$$\frac{1}{\lambda}\cdot\frac{\rho^2}{2(1-\rho)}=\frac{1}{\lambda}\cdot\frac{\rho\frac{\lambda}{\mu}}{2(1-\rho)}=\frac{\rho}{2\mu(1-\rho)}.$$

The average time spent in the system is the average time spent in the queue plus the time required to be serviced; that is,

$$\frac{\rho}{2\mu(1-\rho)}+\frac{1}{\mu}=\frac{\rho+2(1-\rho)}{2\mu(1-\rho)}=\frac{2-\rho}{2\mu(1-\rho)}.$$

The actual queue size should, however, be larger in order to accommodate greater than average queue lengths.

15.2.4.3 M/M/1

If both the arrival rate and processing rate is Markovian, there is an added uncertainty in the processing time. Consequently, the average queue length is larger than that of M/D/1 by a factor of two: $\frac{\rho^2}{1-\rho}$. This equals unity when the load factor is the inverse of the golden ratio: $\frac{\sqrt{5}-1}{2}\approx 0.618$. The standard deviation of the average queue size is

$$\frac{\sqrt{\rho}}{1-\rho},$$

so if the queue size is not to be exceeded with a confidence of 99 %, we must ensure the queue size is the expected queue size plus 2.326 times the standard deviation. A plot of this is shown in Figure 15-3 and we see that if the queue size is 8, we cannot have a load factor greater than 0.721.

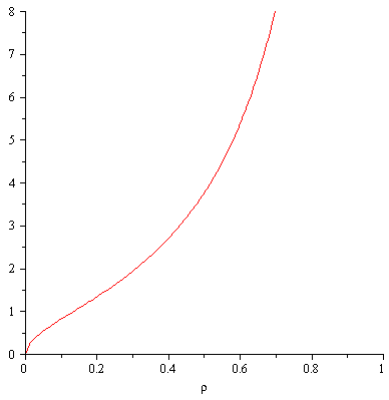


Figure 15-3. An upper bound on the queue size for M/M/1 with 99 % confidence.

As before the average time spent in the queue is

$$\frac{1}{\lambda}\cdot\frac{\rho^2}{(1-\rho)}=\frac{1}{\lambda}\cdot\frac{\rho\frac{\lambda}{\mu}}{2(1-\rho)}=\frac{\rho}{\mu(1-\rho)},$$

The average time spent in the system is the average time spent in the queue plus the time required to be serviced; that is,

$$\frac{\rho}{\mu(1-\rho)}+\frac{1}{\mu}=\frac{\rho+(1-\rho)}{\mu(1-\rho)}=\frac{1}{\mu(1-\rho)}.$$

15.2.4.4 Summary

This section described some of the characteristics of the most common client-server models.

15.2.5 Multiple queues or one queue

As a quick observation: if you have multiple servers, there are two options. You can have one queue and each time a server becomes ready, the next waiting request is sent to that server. Alternatively, you can have a queue for each server and when a new request comes in, the request can be placed into the queue of one of the servers.

A grocery store is an example of a queue for each server, while banks have *multiserver* queues—queues that are used for multiple servers.

It turns out that having a single queue is better than having multiple queues on the expected waiting times, as it may occur that one server may become idle while there are other servers with non-empty queues. You have likely experienced this in a grocery store when you think you're in a short line, but then a hold-up in front of you forces you to wait significantly longer than all other lines around you.

15.2.6 Simulating Markov arrivals and processing times

It's easy enough to simulate deterministic arrival times and processing times; however, how does one simulate Markovian arrival times—after all, the arrival rate is only given as an expected value. We will look at

1. a statistical distribution that tells us the distribution of likely arrivals,
2. a means of simulating it, and
3. an example.

15.2.6.1 The distribution of arrivals

Suppose, for example, even if the expected arrival rate is λ = 5/h, in one hour it could be 3 and in another it could be 7. Fortunately, a rather straight-forward statistical distribution describes the likelihood of each of these appearing: the Poisson (pronounced *pwa-son*, as he was French) distribution is defined as follows: the probability of *k* events occurring is given by

$$e^{-\lambda}\frac{\lambda^k}{k!}.$$

For example, if we substitute $k = 0, 1, 2, \ldots, 10$ into the above formula, this gives us the sequence

k	$e^{-\lambda}\frac{\lambda^k}{k!}$ with $\lambda = 5$
0	0.00674
1	0.03369
2	0.08422
3	0.14037
4	0.17547
5	0.17547
6	0.14622
7	0.10444
8	0.06528
9	0.03627
10	0.01813

Thus, if you are expecting five customers an hour, you will actually only get exactly five 17.55 % of the time, and the likelihood of getting 10 customers (around 2 %) is better than the likelihood of getting no customers (about half a percent). From calculus, you will note that

$$\sum_{k=0}^{\infty} e^{-\lambda}\frac{\lambda^k}{k!} = e^{-\lambda}\sum_{k=0}^{\infty}\frac{\lambda^k}{k!} = e^{-\lambda}e^{\lambda} = e^{-\lambda+\lambda} = e^0 = 1,$$

as by definition, $e^{\lambda} \stackrel{\text{def}}{=} \sum_{k=0}^{\infty}\frac{\lambda^k}{k!}$, which is what we would expect. Thus, the likelihood of there being between 2 and 8 customers in one hour when we expect five per hour is $\sum_{k=2}^{8} e^{-5}\frac{5^k}{k!} \approx 0.8915$, or 90 %.

15.2.6.2 Simulating such arrivals

Unfortunately, this doesn't tell you how to approximate a Poisson distribution. Fortunately, the distribution of the intervals between random events can be described by an exponential distribution: recall that in an exponential distribution, the likelihood of an event occurring in the next two units of time is twice the likelihood of an event occurring in the next unit of time, and if we wait twice as long, we expect twice the probability of a customer showing up in the longer period of time.

Fortunately, it's really easy to calculate an exponential distribution: the probability density function (pdf) itself is defined as

$$f(x) \overset{\text{def}}{=} u(x)\lambda e^{-\lambda x} = \begin{cases} 0 & x < 0 \\ \lambda e^{-\lambda x} & x \geq 0 \end{cases}$$

where u is the unit-step function. Recall that the likelihood of something happening between $x = a$ and $x = b$ is given by the integral $\int_a^b f(\xi)\,d\xi$. Thus, the probability of something happening for a value less than or equal to x is given by the cumulative distribution function (CDF) and is the integral from negative infinity to x of the probability density function (pdf):

$$F(x) = \int_{-\infty}^{x} f(\xi)\,d\xi = \begin{cases} 0 & x < 0 \\ \int_0^x \lambda e^{-\lambda\xi}\,d\xi & x \geq 0 \end{cases} == \begin{cases} 0 & x < 0 \\ 1 - e^{-\lambda x} & x \geq 0 \end{cases}.$$

In order to pick a number in the distribution, we pick a random number X from a uniform distribution on (0, 1) and calculate $F^{-1}(X)$. In this case, the inverse of the CDF is

$$F^{-1}(X) = \begin{cases} -\dfrac{\ln(1-X)}{\lambda} & 0 < X < 1 \\ \text{undefined} & \text{otherwise} \end{cases}.$$

Thus, if an event in our simulation occurred at time t, the next event will occur at time $t - \dfrac{\ln(1-X)}{\lambda}$.

Code for calculating the times of a sequence of events is provided here:

```
for ( k = 0; k < N; ++k ) {
    // drand48() returns a value in [0, 1),
    // but we need (0, 1] so use 1 - drand48()

    arrival_time[k] = arrival_time[k - 1] - log(1.0 - drand48())/LAMBDA;
}
```

Note that while you have always used ln(x) to describe the natural logarithm and, perhaps, $\log_{10}(x)$ as the common logarithm, in almost all mathematical libraries (including C, C++, Java, C#), the natural logarithm is represented by a function `double log( double )`.

Similarly, the processing time was also assumed to be exponentially distributed. Consequently, when an arrival occurs, we may approximate the processing time by, again, sampling from an exponential distribution, only now with the parameter μ.

```
processing_time[k] = -log(1.0 - drand48())/MU;
```

15.2.6.3 Example

Suppose we have a system where $\lambda = 4$ and $\mu = 5$. The load factor should be $\rho = 0.8$. Using the above formula, the following events were created together with their processing time. The Δt_k and c_k were generated using exponential distributions with the appropriate factors. We stopped after 22 events as the next event went beyond $t = 5$.

Event	Δt_k	t_k	c_k
1	0.23176	0.23186	0.05149

2	**0.41109**	0.64285	0.06277
3	**0.94940**	1.59235	0.44791
4	0.05573	1.64808	0.18506
5	0.21242	1.86049	0.01133
6	0.04281	1.90321	0.31124
7	0.22151	2.12472	0.05752
8	0.00090	2.12561	0.15760
9	0.23772	2.36333	0.06114
10	0.09110	2.45443	0.22183
11	0.12124	2.57577	0.05498
12	0.04151	2.61728	0.16873
13	0.18818	2.80546	0.02842
14	0.23377	3.03913	0.02858
15	0.04466	3.08389	0.00673
16	0.18650	3.27039	0.13975
17	0.37393	3.64422	0.11734
18	**0.63583**	4.28005	0.20233
19	0.08189	4.36194	0.00234
20	0.14509	4.50702	0.06148
21	0.11659	4.62361	0.21173
22	0.12356	4.74727	0.07307

There are three inter-arrival times that are perhaps slightly larger than expected, and these are highlighted in bold cyan numbers. Figure 15-4 shows the events, the queue size and the server usage.

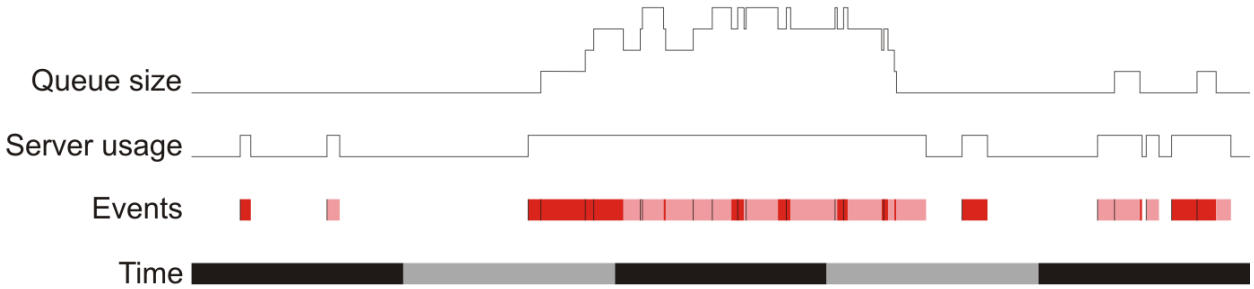


Figure 15-4. The queue size and server usage under simulated events.

The events timeline shows arrivals as black horizontal lines and processing of the events alternating in red and pink. When an arrival occurs during a processing period, that arrival is placed on a queue. The queue size is shown on the top. The queue is mostly empty, but between 1.5 s and 3.5 s, it varies between 1 and 4.

We can make the following observations:

1. The total processing time (sum of the third column) is 2.66337, which indicates a load factor of 0.53. This is below our expected load factor of 0.8.
2. The number of arrivals in the five units of time is 22, which is close to our expected number of arrivals, which is $5 \times 4 = 20$.
3. The average queue length expected to be $\frac{\rho^2}{1-\rho} = 3.2$.

The random numbers were generated in Maple.

15.2.6.4 Simulating sporadic arrivals

One issue with the previous case is that arrivals are aperiodic: there is no lower limit to the time between arrivals. Consequently, such a set-up cannot be used to describe sporadic events. Recall that sporadic events cannot occur more closely than some minimum time τ. We could use the above technique and simply ignore any processing time less than this minimum period; however, this would have the effect of artificially decreasing the arrival rate: this can be more clearly demonstrated with a case where the mean arrival rate is 2/s, but the minimum inter-arrival time is 0.1 s. Consequently, the only processing times that will be accepted are those greater than or equal to 0.1 and thus the average will be greater than expected, so it is unlikely that you will still have a mean arrival rate of 2/s.

For example, a simulation was run generating a sequence of 1000 events. In the first case, the above formula was used to determine the inter-event time. The time for those 1000 events was 486 s, yielding approximately 2.057/s. When run again, but this time excluding those intervals less than 0.1 s, the total time was 588 s, resulting in an arrival rate of 1.701/s.

Instead, if your arrival rate is λ, then use the arrival rate of $\lambda/(1 - \lambda\tau)$ and then add τ to each calculated inter-arrival time. In the example just worked out, this works out to $\lambda_{\text{sporadic}} = 2.5$. Rerunning the simulation, we get the 1000 events occurring over 505 s, resulting in an average arrival time rate of 1.980/s—sufficiently close to the expected arrival rate.

15.2.6.5 Summary

In this section, we have discussed what we expect to see with independent arrivals, how we can model it, and worked with an example. We also considered how to deal with sporadic events.

15.2.7 Summary of modeling client-server systems

In this topic, we have discussed the client-server model, described it mathematically, estimated the size of queues required, and shown how to simulate such a situation.

15.3 Simulating variation

When any physical system is built, it is designed using parts with various specifications: most easily, a resistor may be 120 Ω. However, the actual component seldom has that precision—there will always be variation. You can pay more money, but that will only reduce the variation. Thus, when simulating a system, it is necessary to take this variation into account.

To determine if a system is robust, it is not necessary to assume each component is at one extreme or the other: with n components, this would require 2^n simulations. Instead, it is often sufficient to assign random values to each component within their required specifications and simulate the system with these random values. This would be performed a number of times, but it is sufficient to validate that the system does satisfy the needs of the client.

Thus, suppose each parameter of the various components is known to have characteristics such as having

1. a uniform distribution of $\mu \pm w/2$ (where w is the width of the interval), or
2. a normal distribution with mean μ and standard deviation σ.

To approximate each of these, we are reduced to using the tools provided to us by our software systems, usually restricted to a random number generator that produces a uniform random variable on the range [0, 1) . We will look at simulating each of these.

15.3.1 Simulating uniform distributions

Given a random variable x that has a value on [0, 1), to generate a uniform distribution as described, simply calculate

$$\mu - w/2 + xw = \mu + (x - ½)w.$$

15.3.2 Simulating normal distributions

We have already seen that we can simulate a random variable from an exponential distribution by taking a uniform random normal and calculating the inverse of the commutative distribution function. In the case of the exponential distribution, the pdf, the CDF and its inverse are

$$f(x) \stackrel{\text{def}}{=} \begin{cases} 0 & x < 0 \\ \lambda e^{-\lambda x} & x \geq 0 \end{cases}$$

$$F(x) = \int_{-\infty}^{x} f(\xi)\, d\xi = \begin{cases} 0 & x < 0 \\ 1 - e^{-\lambda x} & x \geq 0 \end{cases}$$

$$F^{-1}(X) = \begin{cases} -\dfrac{\ln(1-X)}{\lambda} & 0 < X < 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

For a normal distribution with mean μ and standard deviation σ, it is much easier to calculate a standard normal random variable, one that has a mean of 0 and a standard deviation equal to 1. Suppose that X is such a random variable. In this case, we would simply calculate $\mu + \sigma X$. This still leaves us with the problem of approximating a standard normal distribution. Unfortunately, this isn't so easy: the probability density function is $\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$ and the cumulative distribution function is $\frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$ where erf is the Gaussian error function. One could simply solve for this, $\sqrt{2}\operatorname{erf}^{-1}(2X - 1)$ where $0 < X < 1$. Unfortunately, there is no simple formula for either the error function or its inverse.

There are two common solutions to approximating a standard normal:

1. twelve random samples on (0, 1),
2. Marsaglia's polar method, and
3. the Box-Muller method.

We will consider each.

15.3.2.1 Twelve random samples

If you calculate twelve uniform random variables on (0, 1), add them together and subtract 6, you get a good approximation to a standard normal.

```
X = drand48() + drand48() + drand48() + drand48() + drand48() + drand48()
  + drand48() + drand48() + drand48() + drand48() + drand48() + drand48() - 6;
```

If we calculation 2000 such numbers, we get a distribution of values shown in Figure 15-5.

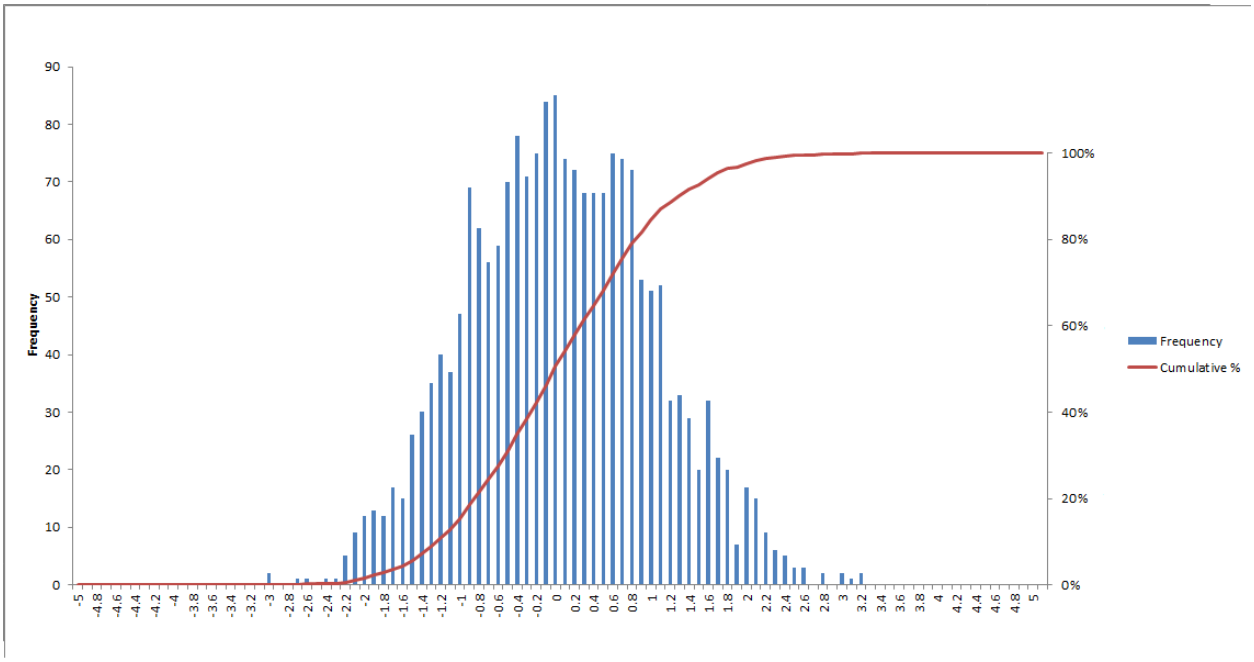


Figure 15-5. 2000 approximations to random standard normals by adding 12 uniform random numbers in [0, 1] and subtracting 6.

This is useful if you have a uniform random number generator, but no math library. More superior to this, however, is the next approximation.

15.3.2.2 Marsaglia's polar method

In this technique, we take two random variables on [–1, 1] and call them U_1 and U_2. If $S = U_1^2 + U_2^2$ and $S < 1$, then

$$X_1 = U_1\sqrt{\frac{-2\ln(S)}{S}}$$

$$X_2 = U_2\sqrt{\frac{-2\ln(S)}{S}}$$

are two standard normal distributions. This code generates two such random variables:

```
double X1, X2;

while ( 1 ) {
    X1 = 2.0*drand48() - 1.0;   // A value in [0, 1) is mapped to [-1, 1)
    X2 = 2.0*drand48() - 1.0;

    double S = X1*X1 + X2*X2;

    if ( S < 1 && S != 0 ) {
        S = sqrt( -2.0*log(S)/S );
        X1 = X1*S;
        X2 = X2*S;
        break;
    }
}
```

To convert these to a random normal with mean μ and standard deviation σ, use `MU + X1*SIGMA` and `MU + X2*SIGMA`.

Running this algorithm 1000 times gives 2000 random standard variables with a distribution gives the distribution of values shown in Figure 15-6.

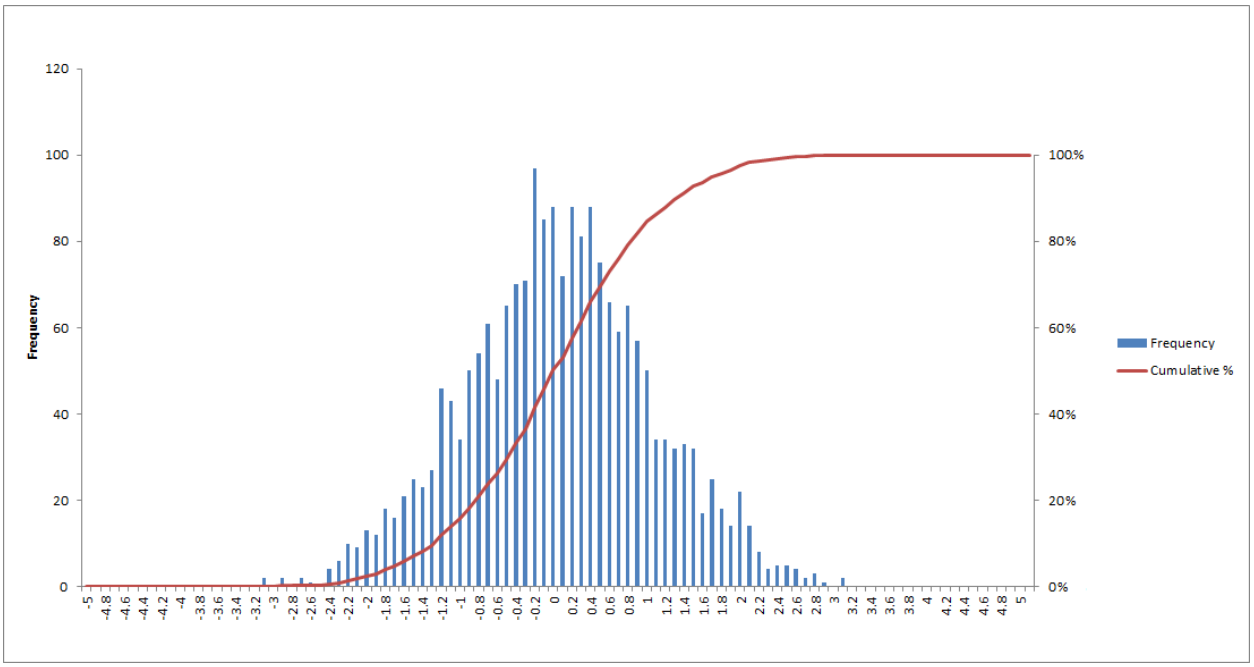


Figure 15-6. 2000 approximations to standard normal using Marsaglia's polar method.

15.3.2.3 Box-Muller transform

While Marsaglia's polar method is efficient for simulations, one issue for real-time systems is the potential uncertainty with respect to the run time. The probability that pair is outside the unit circle is $1-\frac{\pi}{4}\approx 21.46\,\%$. Consequently, the probability that the loop will iterate six or more times is still a significant 0.01 % or one in ten thousand, even if the average number of iterations of the loop is only $\sum_{k=1}^{\infty} k\left(1-\frac{\pi}{4}\right)\frac{\pi}{4}=\frac{4}{\pi}\approx 1.273$. If the algorithm is to be to be strictly for simulations, this is more than appropriate; however, if the algorithm is to be used in a hard real-time system, this may be unacceptable. The Box-Muller transform takes Marsaglia's method and transforms it into Cartesian coordinates.

In this technique, again, we take two random variables on (–1, 1) and call them U_1 and U_2 and define

$$X_1 = \sqrt{-2\ln(U_1)}\cos(2\pi U_2)$$
$$X_2 = \sqrt{-2\ln(U_1)}\sin(2\pi U_2)$$

giving two independent random variables with standard normal distributions. The following gives a structure that can be used to generate standard normal variables:

```
#define PI acos( -1.0 )

typedef struct {
    double theta;
```

```
    double R;
    bool stored;
} box_muller_t;

void init_box_muller( box_muller_t *const p_this ) {
    p_this->stored = false;
}

double next_box_muller( box_muller_t *const p_this ) {
    double normal_deviate;

    if ( !p_this->stored ) {
        // Recall that drand48() gives a value in [0, 1)
        p_this->R      = sqrt( -2.0*log( drand48() - 1.0 ) );
        p_this->theta  = 2.0*PI*drand48();
        p_this->stored = true;

        normal_deviate = (p_this->R)*sin( p_this->theta );
    } else {
        p_this->stored = false;
        normal_deviate = (p_this->R)*cos( p_this->theta );
    }

    return normal_deviate;
}
```

The cost of this algorithm for a fixed execution time is the two additional calls to trigonometric functions. To more closely balance the work between calls to `next_box_muller()`, the second trigonometric function is not evaluated immediately, but rather evaluation is delayed until that variable is needed. For general purpose computing, the Marsaglia's polar method is more efficient. To convert these the output to a random normal with mean μ and standard deviation σ, use

```
MU + next_box_muller( p_bm )*SIGMA;
```

15.3.2.4 Summary of simulating normal distributions

In approximating a normal random variable, it is usual to approximate a standard normal random variable and then perform the appropriate affine transformation. The Marsaglia polar method is the most efficient that gives an excellent approximation, but if a mathematics library is not available, one can still approximate a standard normal by adding 12 uniform random variables on [0, 1] and subtracting 6.

15.3.3 Summary of simulating variation

In simulating a real-world situation, there will always be variation both in the components we have control over and in natural phenomena. We can, however, often describe such variation using either uniform or normal distributions.

15.4 Summary of simulating physical systems

In this topic, we've briefly described how we can simulate physical systems. When dealing with simple laws of physics with respect to the movement of objects, a physics engine is the appropriate tool. When we are trying to simulate a situation where a server is satisfying requests for clients, we use the inverse of the cumulative distribution function. Finally, when we are simulating either uniform or normal distributions, these can be done by using samples from a uniform distribution on [0, 1].

Problem set

15.1 Suppose that we have a single server and we are expecting λ = 7.4 periodic requests per second. Under the assumption that no other tasks are executing, suppose that each service requires a call to an interrupt service routine (ISR) that takes a fixed amount of time to execute. What is the maximum possible runtime of that ISR in order to ensure that the load factor does not exceed 1?

15.2 In Question 15.1, suppose that the maximum processor utilization for other tasks is U = 0.83. What is the maximum possible runtime of the ISR in order to ensure that the load factor does not exceed 1?

15.3 If the requests were arriving independent of each other, why do we not want to simply ensure that the load factor is less than or equal to 1?

15.4 Would you use a binary semaphore or a counting semaphore for the task handling the requests? Why?

15.5 In Question 15.2, suppose we did not want to exceed a queue size of four 99 % of the time. What is the maximum amount of time that the servicing task can execute?

15.6 In our statistical analysis of queuing theory, what assumption is made about the priority of any task that services the routine? Can we place any reliance on the average queue size given a particular load factor if the task servicing the requests has a lower priority? Why or why not?

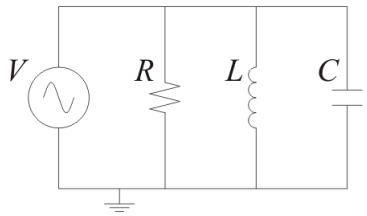
15.7 In servicing a request, the ISR notes that the queue is full; it now has two options, to ignore the most recent request or to throw out (overwrite) the oldest request. Which would you consider if:

1. the requests are coming from separate entities where there are currently 10 entities waiting to have their requests serviced, and
2. the requests are sequences of data to be processed about the current state of a particular device that requires monitoring as the temperatures tend to occasionally go beyond specified limits and require that fans either be turned on or off.

15.8 In a hard real-time system, why must a sporadic task be serviced in less time than its minimum period τ even if this value is much smaller than the average inter-arrival time?

15.9 Suppose you want to simulate a sporadic event as follows: After each time τ, calculate a uniform random variable $0 < X < 1$, and if $X < p$, assume that sporadic event occurred. How would you calculate p if τ = 0.01 and λ = 5?

15.10 Consider the following circuit



As you will recall from your circuits course, with a frequency *f* of the voltage source, the angular frequency is $\omega = 2\pi f$, we have inductive reactance $X_L = \omega L$, capacitive reactance $X_C = \dfrac{1}{\omega C}$ and impedance

$$Z = \frac{1}{\sqrt{\left(\frac{1}{R}\right)^2 + \left(\frac{1}{X_C} - \frac{1}{X_L}\right)^2}}.$$

The phase angle is satisfies the relationship $\cos(\phi) = \dfrac{Z}{R}$. Suppose that V = 110 V, f = 100 Hz, R = 120 Ω, L = 12 mH and C = 10 μF. Suppose that each component has a maximum error of 10 %. How would you find the maximum possible range of the phase angle?

15.11 Suppose in the previous question, you determined that the range of phase angles is between 84.8° and 87.3°. How likely is it that you will experience either of these extremes? How would you simulate a more likely scenario to determine the phase angle?

15.12 Suppose you were aware that the errors were normally distributed with a standard deviation of 5 % of the value. How would you simulate a more likely scenario to determine the phase angle?

15.13 You are given the uniform random numbers

0.40, 0.19, 0.02, 0.80, 0.43, 0.84, 0.41, 1.00, 0.39, 0.69, 0.77, 0.73

and you are asked to generate an approximation of a normally distributed value with mean 75 and standard deviation 10.

15.14 You are given given two uniform random numbers 0.10651 and 0.39641. Find two approximations of a normal distribution with mean 75 and standard deviation 10. You may wish to recall these two formulas:

$$X_1 = U_1\sqrt{\frac{-2\ln(S)}{S}}$$

$$X_2 = U_2\sqrt{\frac{-2\ln(S)}{S}}$$

16 Software verification

Suppose you have software that is meant to satisfy a set of requirements for functioning in a particular system. Previously, the most common means of testing such a system was to generate tests and to run simulations on the system. The inputs would be a subset of expected values from the system, and the response of the system would be compared to the expected response. Simulations can be used to determine two objectives:

1. does the design satisfy the needs of the client (design validation), and
2. does the software satisfy the specifications of the design (software verification)?

Validating a design is necessary early on in the process, as one can have software that satisfies the requirements of a design, but if that design does not satisfy the needs of the user, the software is useless. However, once the design is validated, it is necessary that the software be checked to ensure that it satisfies the specifications of the design. In this case, using a large number of inputs may initially find failures in the system, but as failures are found and corrected, the remaining failures will be likely more and more difficult to find—often requiring subtle relationships between inputs. Using simulation therefore does not necessarily ensure comprehensive testing coverage: not having detected any failures in the last thousand simulations does not mean that there will be no failure in the 1001^{st} simulation.

Therefore, testing and simulation can be used to demonstrate the existence of failures, but are incapable of demonstrating the absence of failures. One cannot even estimate the expected number of remaining failures. Consequently, testing and simulation cannot guarantee exceptionally high reliability in any reasonable amount of time. In many real-time systems, this may be acceptable so long as the number of failures is sufficiently low. For example, the system may simply periodically reset. This may lead to a degradation of performance, but it may not be catastrophic. If, however, safety is critical, it is absolutely necessary to examine all possible states of the system. For example, a robotic arm traversing along a rail can never fail to stop prior to running off the rail at either end.

Currently, all known software verification tools require that time be broken up into fixed intervals with a period τ. Rules are based on events occurring in the current time intervals, and how the system responds during the next or subsequent periods. The size of this period depends on how quickly the system is required to respond; if the aforementioned robotic arm has a task that monitors to ensure that the arm will not drop off the end of the rail, the period must be sufficiently small to ensure that the individual steps involved in responding to the situation can occur in separate subsequent intervals. If the size of the period is too large, a verified system may need to respond unnecessarily early to situations, as subsequent responses must occur in separate time intervals; however, if the period is too small, this may lead to a system where creating the rules becomes too onerous to maintain and too computationally expensive to verify in a reasonable amount of time. Fortunately, recall from Section 7.4.5.2.4 that if the periods of the periodic and sporadic tasks are mutually harmonious, this implies that RM scheduling is effective even if the utilization is 100 %. Therefore, the minimum period is an excellent candidate for the period used for verification, and the state of the system will be determined by what can be performed during the execution of this smallest time interval.

Thus, we will represent our system $\mathcal{M}$ as a state-transition graph within a set of states S together with an initial state $s_0 \in S$. It is then necessary to determine whether or not the system satisfies the specification. To achieve this, however, we must provide our specifications through an appropriate mathematical description. The approach we will use in this class is to provide the specification in terms of the *linear temporal logic* (LTL) first described by Amir Pnueli in 1977. The mathematical representation of our specification will be given by ϕ. Without going into details in this course, the complement specification $\neg\phi$ is then translated into a *Büchi* state-transition graph. This state-transition graph is the composed with the state-transition graph of the system $\mathcal{M}$ producing a system where the only accepted paths are those that violate the specifications. This is then fed to a model checker that attempts to find such a path and if a path is found, this path may then be printed and it provides a counterexample to the claim that the system satisfies the specifications. If no counterexample exists, we will say that $\mathcal{M} \models \phi$.

As you may suspect, the number of possible states can be exceptionally large. For example, something as simple as an n-bit register can have 2^n states. Research into simplifying the search on such large spaces can, under some circumstances, significantly reduce the run-time of algorithms searching for such counterexamples.

We have already described the process of *verification*, namely, ensuring that the design and implementation satisfies the requirements; and *validation*, ensuring that the requirements satisfy the needs of the user or client. We will now present a scenario where we will

1. begin with a statement of the needs of the user;
2. derive a collection of requirements
    a. first discussing functional requirements,
    b. describing constraints, and
    c. adding real-time performance requirements.

Each of the three types of constraints will require an introduction to

1. propositional logic,
2. predicate logic, and
3. linear temporal logic.

We will then proceed by discussing verification.

16.1 The scenario and user or client needs

Suppose we want to write down the requirements for driving a car down a highway under normal conditions (that is, there are no accidents or other special circumstances). Let us therefore consider the following situation: the speed of this car is v km/h on a road with a speed limit of ℓ km/h and the distance to the next vehicle is d metres away (infinite if no vehicle is in sight ahead of this car). These rules are written for a passenger vehicle, so we will call the vehicle being driven a "car". Another vehicle on the road will be referred to as "vehicle". We will use the Ontario Ministry of Transportation requirement that you are at least two seconds from the vehicle in front of you.

The needs of the user include:

1. This car should move at a reasonable speed at or above the speed limit, but never such that a speeding ticket will result in acquiring demerit points (over 15 km/h over the speed limit),
2. If there is a vehicle in front of this car, the distance between this car and the vehicle should never be less than two seconds away.

There are three actions we can perform: depress the accelerator, maintain the state of the accelerator, and release the accelerator. In order to satisfy these two needs, we will

1. attempt to maintain a speed between 5 km/h and 10 km/h over the speed limit, and
2. attempt to ensure the minimum distance to the next vehicle never goes below 2.4 s of the vehicle in front of us.

We can, however, also describe the specifications mathematically, in which case, the mathematical expression of the specifications is represented by ϕ, and the prototype is represented by a model $\mathcal{M}$. If the model satisfies our specifications, we say that

$$\mathcal{M} \models \phi \ .$$

16.2 Propositional logic

We will begin by considering propositional logic. For the most part, you have covered propositional logic in your MTE 262 *Introduction to Microprocessors and Digital Logic* course; however, you will have missed one important operator: implication. We will quickly review those operators and tautologies you have learned and consider further tautologies that are relevant to this course.

16.2.1 Review of propositional logic

We will review three Boolean logic operations: AND, OR and NOT. We will use different notation in this course to express these, as is shown in Table 5.

Table 5. Mathematical operations.

Operator	Engineering	C/C++	Mathematical	Uppaal
logical AND	AB	`A && B`	$A \wedge B$	`A and B`
logical OR	$A + B$	`A || B`	$A \vee B$	`A or B`
logical NOT	$\overline{A}$	`!A`	$\neg A$	`not A`

Another operator you may remember from high school is *if-and-only-if* or IFF. This says that two states are equivalent. For this, we use the operator

$$f \leftrightarrow g$$

and this is the logical analogy of the mathematical equals operator "=" and it has the truth table

f	g	$f \leftrightarrow g$
F	F	T
F	T	F
T	T	T
T	F	F

Here are a few aide-de-memoires:

The symbol $\wedge$ looks like A (as in "AND") and it is reminiscent of the *intersection* of sets:

$(x \in S \cap T) \leftrightarrow \left[(x \in S) \wedge (x \in T)\right]$ (something that is in set S **AND** in set T).

The symbol $\vee$ looks like a script "r" in *o*r and is reminiscent of the *union* of sets:

$(x \in S \cup T) \leftrightarrow \left[(x \in S) \vee (x \in T)\right]$ (something that is in set S **OR** in set T).

You were also introduced to a number of *tautologies*, that is, statements that are true no matter what. For example, de Morgan's laws:

$$\neg(f \wedge g) \leftrightarrow (\neg f \vee \neg g)$$
$$\neg(f \vee g) \leftrightarrow (\neg f \wedge \neg g)$$

Note that the precedence of logical not is highest: we will never interpret $\neg f \leftrightarrow g$ as meaning $\neg(f \leftrightarrow g)$. In general, we will use brackets everywhere else. Beyond this, we will introduce a new operator holding a meaning you implicitly understand, but have never seen in writing.

16.2.2 The implication operator

In general, we want the system to respond to certain events, and therefore we will write our requirements in the form

if *f* occurs, *g* must happen,

or more succinctly, "if *f*, *g*" or "*f* implies *g*". Due to the ambiguities of natural languages, we will rewrite our requirements in terms of propositional and predicate logic. Given two statements *f* and *g*, we say

$$f \to g$$

if *g* is true whenever *f* is true. The statement that $f \to g$ is false if it ever occurs that *g* is false while *f* is true. The easiest way to see this is to consider a few examples:

1. "if it is raining, there are clouds in the sky" is a true implication, while
2. "if there are clouds in the sky, it is raining" is a false implication.

The truth table for implication is

f	g	$f \to g$
F	F	T
F	T	T
T	T	T
T	F	F

We may now introduce our first new tautology,

$$\left[(f \to g) \wedge (g \to f)\right] \leftrightarrow (f \leftrightarrow g).$$

That is, both *f* implies *g* and *g* implies *f* is equivalent to *f* if-and-only-if *g*.

Now, there are some constructions in logic that are always true, and these are called *tautologies*. We will only look at a few, but to introduce some:

1. $f \vee (\neg f)$; that is, either *f* is true or false.
2. $\left[\neg(f \vee g)\right] \leftrightarrow \left[(\neg f) \wedge (\neg g)\right]$; that is, *f* or *g* is false if and only if both *f* and *g* are false.
3. $\left[f \to (\neg f)\right] \to (\neg f)$; that is, if the truth of *f* implies that *f* is false, then *f* is false (*reductio ad absurdum*; for example, if you assume there are a finite number of prime numbers, from this we deduce there are not a finite number of prime numbers; therefore, there are not a finite number of prime numbers).
4. $\left\{f \to \left[g \wedge (\neg g)\right]\right\} \to (\neg f)$; that is, if *f* being true implies that *g* is both true and false, the *f* is false (the more traditional *proof by contradiction*).
5. $(f \to g) \leftrightarrow \left[(\neg f) \vee g\right]$; that is, *f* implies *g* is the same as saying either *f* is false or *g* is true (compare "if it is raining, there are clouds in the sky" and "either it is not raining or there are clouds in the sky").

We will now look at our driving example.

16.2.3 Our example

Suppose we want to write down the requirements for driving a car down a highway under normal conditions (that is, there are no accidents or other special circumstances) and that other vehicles on the road are sufficiently sparse that we can react in time to each event. Let us therefore consider the following situation: the speed of this car is *v* km/h on a road with a speed limit of ℓ km/h and the distance to the next vehicle is *d* metres away (infinite if no vehicle is in sight ahead of this car). These rules are written for a passenger vehicle, so we will call the vehicle being driven a "car". Another vehicle on the road will be referred to as "vehicle". We will use the Ontario Ministry of Transportation requirement that you are at least two seconds from the vehicle in front of you.

We can transform these into a set of functional requirements in the form of propositions:

1. If this car is moving at an appropriate speed, keep the accelerator in the current state.
2. If this car is moving too slow, depress the accelerator to increase acceleration.
3. If this car is moving too fast, release the accelerator to decrease acceleration (decelerate).
4. If this car is moving faster than 10 km/h over the speed limit ($v > \ell + 10$), this car is moving too fast.
5. If there is a vehicle within two-and-two-fifths seconds of this car, this car is too close.
6. If this car is too close, this car is moving too fast.
7. If this car is moving slower than 10 km/h over the speed limit and there is a vehicle between two and three seconds away, this car is moving at an appropriate speed.
8. If there is no vehicle within three seconds and the speed is between 5 km/h and 10 km/h over the speed limit ($\ell + 5 \leq v \leq \ell + 10$), this car is moving at an appropriate speed.
9. If there is no vehicle within three seconds and the speed is less than 5 km/h over the speed limit, this car is moving too slow.

First, while measuring distance travelled in two seconds is the easiest measure of distance for humans, some other means of measuring distance is likely more reasonable for a computerized system, so we convert the timing into distance (v km/h ⋅ 2 s = cv m):

1. a vehicle two seconds away is $\frac{5}{9}v$ m ahead,
2. a vehicle 2.4 s away is $\frac{2}{3}v$ m ahead, and
3. a vehicle three seconds away is $\frac{5}{6}v$ m ahead.

At any time, the state of this car can be described by its distance to the next vehicle, either driving

1. d_{close}, too closely to,
2. d_{safe}, at a safe distance from, or
3. d_{far}, at a significant distance from

the next vehicle; and the speed of this car as being

1. v_{slow}, too slow,
2. v_{good}, at an appropriate level, or
3. v_{fast}, too fast.

The reactions may be defined as

1. P, press down the accelerator,
2. M, maintain the accelerator in the current position, or
3. R, release the accelerator.

We may now write the nine English requirements in terms of propositions:

1. $v_{\text{slow}} \rightarrow \text{P}$
2. $v_{\text{good}} \rightarrow \text{M}$
3. $v_{\text{fast}} \rightarrow \text{R}$
4. $(v > \ell + 10) \rightarrow v_{\text{fast}}$
5. $(d < \frac{2}{3}v \text{ m}) \rightarrow d_{\text{close}}$
6. $d_{\text{close}} \rightarrow v_{\text{fast}}$
7. $[(v \leq \ell + 10) \wedge (\frac{2}{3}v \text{ m} \leq d \leq \frac{5}{6}v \text{ m})] \rightarrow v_{\text{good}}$
8. $[(d > \frac{5}{6}v \text{ m}) \wedge (\ell + 5 \leq v \leq \ell + 10)] \rightarrow v_{\text{good}}$

9. $[(d > \frac{5}{6}v\ \mathrm{m}) \wedge (v < \ell + 5)] \rightarrow v_{\mathrm{slow}}$

Using rules of logic (technically, they are called "tautologies"), we can make simplifications. For example, one tautology is called a *syllogism* and it says that $[(a \rightarrow b) \wedge (b \rightarrow c)] \rightarrow (a \rightarrow c)$. Applying this, we can make the following simplifications:

1. $[(v > \ell + 10) \rightarrow v_{\mathrm{fast}}] \wedge [v_{\mathrm{fast}} \rightarrow \mathrm{R}] \rightarrow [(v > \ell + 10) \rightarrow \mathrm{R}]$
2. $[(d < \frac{2}{3}v\ \mathrm{m}) \rightarrow d_{\mathrm{close}}] \wedge [d_{\mathrm{close}} \rightarrow v_{\mathrm{fast}}] \wedge [v_{\mathrm{fast}} \rightarrow \mathrm{R}] \rightarrow [(d < \frac{2}{3}v\ \mathrm{m}) \rightarrow \mathrm{R}]$
3. $\{[(v \leq \ell + 10) \wedge (\frac{2}{3}v\ \mathrm{m} \leq d \leq \frac{5}{6}v\ \mathrm{m})] \rightarrow v_{\mathrm{good}}\} \wedge [v_{\mathrm{good}} \rightarrow \mathrm{M}]\}$
   $\rightarrow \{[(v \leq \ell + 10) \wedge (\frac{2}{3}v\ \mathrm{m} \leq d \leq \frac{5}{6}v\ \mathrm{m})] \rightarrow \mathrm{M}\}$
4. $\{[(d > \frac{5}{6}v\ \mathrm{m}) \wedge (\ell + 5 \leq v \leq \ell + 10)] \rightarrow v_{\mathrm{good}}\} \wedge [v_{\mathrm{good}} \rightarrow \mathrm{M}]\}$
   $\rightarrow \{[(d > \frac{5}{6}v\ \mathrm{m}) \wedge (\ell + 5 \leq v \leq \ell + 10)] \rightarrow \mathrm{M}\}$
5. $\{[(d > \frac{5}{6}v\ \mathrm{m}) \wedge (v < \ell + 5)] \rightarrow v_{\mathrm{slow}}\} \wedge [v_{\mathrm{slow}} \rightarrow \mathrm{P}]\} \rightarrow \{[(d > \frac{5}{6}v\ \mathrm{m}) \wedge (v < \ell + 5)] \rightarrow \mathrm{P}\}$

Note that the syllogism says "if *a* implies *b* and *b* implies *c*, then *a* implies *c*, as well." For example, one may make an argument that these two implications are valid:

1. If it is raining, there are clouds in the sky.
2. If there are clouds in the sky, significant amounts of thermal radiation are reflected into the atmosphere.

Alternatively we could write these as

1. It is raining implies that there are clouds in the sky.
2. There being clouds in the sky implies that significant amounts of thermal radiation are reflected into the atmosphere.

However these are worded, they are both of the form $a \rightarrow b$. Thus, because the consequence of the first is the condition of the second, we may automatically apply the syllogism to get

3. If it is raining, significant amounts of thermal radiation are reflected into the atmosphere.

There is no need to prove this as a separate proposition.

We can also use the logical rule that $[(a \rightarrow c) \vee (b \rightarrow c)] \leftrightarrow [(a \vee b) \rightarrow c]$ to join the first and second and the third and fourth of these propositions:

1. $\{[(v > \ell + 10) \rightarrow \mathrm{R}] \vee [(d < \frac{2}{3}v\ \mathrm{m}) \rightarrow \mathrm{R}]\}$
   $\leftrightarrow \{[(v > \ell + 10) \vee (d < \frac{2}{3}v\ \mathrm{m})] \rightarrow \mathrm{R}\}$
2. $\{[(v \leq \ell + 10) \wedge (\frac{2}{3}v\ \mathrm{m} \leq d \leq \frac{5}{6}v\ \mathrm{m})] \rightarrow \mathrm{M}\} \vee \{[(d > \frac{5}{6}v\ \mathrm{m}) \wedge (\ell \leq v \leq \ell + 10)] \rightarrow \mathrm{M}\}$
   $\leftrightarrow \{[(v \leq \ell + 10) \wedge (\frac{2}{3}v\ \mathrm{m} \leq d \leq \frac{5}{6}v\ \mathrm{m})] \vee [(d > \frac{5}{6}v\ \mathrm{m}) \wedge (\ell + 5 \leq v \leq \ell + 10)]\} \rightarrow \mathrm{M}$
3. $[(d > \frac{5}{6}v\ \mathrm{m}) \wedge (v < \ell + 5)] \rightarrow \mathrm{P}$

This tautology says "a implies c or b implies c if and only if either a or b will imply c."

Another tautology you may be aware of is called the *contrapositive*: The statement $a \to b$ is true if and only if the statement $\neg b \to \neg a$ is true. You only have to try it out: the contrapositive of "if it is raining, there are clouds in the sky" is "if there are no clouds in the sky, it is not raining." Mathematically, we write this as $(a \to b) \leftrightarrow (\neg b \to \neg a)$.

Note that $(a \to b) \nleftrightarrow (\neg a \to \neg b)$, after all, "if there are clouds in the sky, it is raining" is obviously false.
One could, however, say $[(a \to b) \leftrightarrow (\neg a \to \neg b)] \leftrightarrow (a \leftrightarrow b)$.

Another way is to use a technique similar to Karnaugh maps. First, we consider each category of speed and each category of distance, and determine which applies. If we follow through the logic on each of these, we get the following table:

Distance	**Speed**		
	slow	good	fast
close	R	R	R
safe	M	M	R
far	P	M	R

As you may remember from Karnaugh maps, you may suddenly realize that these rules may be simplified in a similar way. For example, looking at the first row and third column, we see that if this car is too close or this car is too fast, decelerate.

In either case, we will get the same set of rules, and rendering our simplified requirements back into English, we get:

1. If this car is moving faster than 10 km/h over the speed limit ($v + 10 > \ell$) or there is a vehicle less than $\frac{2}{3}v$ m away, decelerate.
2. If both this car is moving no more than 10 km/h over the speed limit ($v \leq \ell + 10$) and there is a vehicle between $\frac{2}{3}v$ m and $\frac{5}{6}v$ m ahead, or both the closest vehicle is over $\frac{2}{3}v$ m ahead and the speed is between 5 km/h and 10 km/h over the speed limit ($\ell + 5 \leq v \leq \ell + 10$), maintain the current speed.
3. If this car is moving slower than 5 km/h over the speed limit ($v < \ell + 5$) and the closest vehicle is over $\frac{5}{6}v$ m away, accelerate.

You will notice a weakness in the English language: note the awkward wording If both a and b, or both c and d, e. It is much clearer to write $[(a \wedge b) \vee (c \wedge d)] \to e$.

One observation from the table is that we note we could simplify these requirements if we write them as an if—else-if—else statement; that is,

1. If this car is moving faster than 10 km/h over the speed limit ($v + 10 > \ell$) or there is a vehicle within two seconds of this car (less than $\frac{2}{3}v$ m away), decelerate;
2. else if this car is moving slower than 5 km/h over the speed limit ($v < \ell + 5$) and there is no vehicle within three seconds (the closest vehicle is over $\frac{5}{6}v$ m away), accelerate;
3. else, this car is moving at an appropriate speed.

Here, if Requirement 1 holds, we do not consider Requirements 2 or 3 and if Requirement 2 holds, we do not consider Requirement 3. This is a formulation that can be rendered easily in both English and most programming languages; however, it cannot easily be written as a propositional statement.

Question: Is it ethical to design a real-time system to explicitly break a regulation (going at least 5 km/h over the speed limit)?

Why are we using propositional logic?

Converting requirements into logical propositions allows us to discover contradictions in the requirements. For example, another tautology is that $[(a \rightarrow b) \wedge (a \rightarrow c)] \rightarrow [a \rightarrow (b \wedge c)]$; however, if we determine that $a \rightarrow b$ and $a \rightarrow \neg b$, then $a \rightarrow (b \wedge \neg b)$, and $(b \wedge \neg b) \leftrightarrow \mathbf{F}$. This implies a contradiction. Suppose now we had many other rules, taking into account other situations:

1. Is someone tailgating us?
2. Is there traffic ahead?
3. Does it look like someone is going to change into this lane?
4. Is there fog? If there is fog, how dense is it?
5. How dark is it? If it is dark, are there street lights?

With all of these additional considerations, it might be quite easy to come up with a small set of rules such that we ultimately deduce that under a certain set of conditions, we should both accelerate and decelerate:

1. $(a \wedge b \wedge c \wedge d \wedge e \wedge f) \rightarrow \mathrm{P}$
2. $(a \wedge b \wedge c \wedge d \wedge e \wedge f) \rightarrow \mathrm{R}$

It is not possible to do both, and thus our requirements have a contradiction. This would require us to re-examine our conditions.

16.2.4 A fun example

For any real-time system, we expect this system to be used in a real-world environment and it must perform according to specifications that are set out for the system. For simple real-time systems, these specifications can be written down using English, with all of its ambiguities; for example,

1. After a child squeezes Tickle Me Elmo[41], it should laugh and make a statement.
2. After a child squeezes Tickle Me Elmo three times within five seconds, it should begin to shake and laugh for many seconds.
3. Tickle Me Elmo should be child safe.
4. Tickle Me Elmo should be cheap.
5. Tickle Me Elmo should be battery powered.
6. Tickle Me Elmo should not fail before the end of the Christmas holiday.

Tickle Me Elmo is, of course, a soft real-time system. If Elmo responds outside of the specifications, there is a gradual decline in quality of service (that is, keeping the attention of the child). The goal is to develop a set of actuators, speakers, motors, and a power supply (battery), and result of this endeavour should satisfy the above specifications. Some of these, of course, don't apply to the software system.

Given "if Elmo is squeezed, Elmo should laugh and make a statement". Thus, we have

$$x \rightarrow (y \wedge z).$$

Afterward, you quickly realize Elmo should not:

1. make the same statement twice in a row, or
2. say the statements in the same order.

[41] Tickle Me Elmo is a trademark of Tyco Toys and is used here as an example for educational purposes.

You may, however, require a number of conditions to hold prior to a consequence following:

$$(w \wedge x \wedge y) \rightarrow z$$

Alternatively, you may require that any number of conditions may trigger a consequence, in which case, we write

$$(w \vee x \vee y) \rightarrow z$$

There may be multiple actions that trigger the same response: squeezing his hand or touching his belly or a rapid deceleration (after having been thrown).

16.3 Predicate logic

In addition to requirements, we would also like to specify constraints. A constraint is a condition that must always hold. We may specify constraints through predicate logic. We will define predicate logic and then continue our example.

16.3.1 Introduction to predicate logic

To describe these, we will introduce quantifiers:

Next, we consider the quantifiers of predicate logic: let *f*(*x*) be a statement about a value *x* that is selected from a universal set *U*. Now, we write

1. $\forall x : f(x)$ if *f* is true for all values of $x \in U$, and
2. $\exists x : f(x)$ if *f* is true for at least one value of $x \in U$.

There are some nice relationships between these two:

1. $\neg[\forall x : f(x)] \leftrightarrow \exists x : \neg f(x)$, or "it is false that *f*(*x*) is true for all values of *x* if-and-only-if there exists an *x* such that *f*(*x*) is false; and
2. $\neg[\exists x : f(x)] \leftrightarrow \forall x : \neg f(x)$, or "it is false that there is an *x* such that *f*(*x*) is true if-and-only-if for all *x*, *f*(*x*) is false.

Here are a few more aide-de-memoires:

The upside-down A stands for the '*a*' in "for *a*ll". If $\forall x : f(x)$ is true, we say "for all *x*, *f*(*x*) is true.

The backwards E stands for the '*e*' in "there *e*xists". If $\exists x : f(x)$ is true, we say "there exists an *x* such that *f*(*x*) is true.

16.3.2 Our driving example

We want to say that at all times, the speed of this car cannot exceed 15 km/h over the speed limit and the distance to the next vehicle cannot be less than two seconds. In this case, both speed and distance are functions that change over time. Therefore, to say that something is true for all time, we write

1. $\forall t : v(t) < \ell + 15 \text{ km/h}$, and
2. $\forall t : d(t) > \frac{5}{9} v(t) \text{ m}$.

In modeling, however, we will often us discrete time, where t_0 is our initial time and the time between t_k and t_{k+1} is some fixed amount. Therefore, we will write the speed and distance at time t_k as v_k and d_k, respectively. Thus, our constraints would be

1. $\forall k : v_k < \ell + 15 \text{ km/h}$, and
2. $\forall k : d_k > \frac{5}{9} v_k \text{ m}$.

If you wanted to make a requirement such as, there is a time such that the vehicle stops, one could write

$$\exists t : v(t) = 0 \text{ km/h or } \exists k : v_k = 0 \text{ km/h}.$$

16.4 Linear temporal logic

In the above example, we did not include any timing information, nor did we indicate which rules were *hard* and which were *firm* or *soft*. To be able to define hard requirements, it is necessary to introduce further operators, and one collection of these operators is referred to collectively as *linear temporal logic* (LTL). We will look at the four related operators, and then look at how specific requirements can be formulated using these operators.

16.4.1 LTL operators

LTL is one means of describing temporal events. We will slowly build up the tools necessary to describe real-time systems. This first tool will only be used to describe relative events and it is a variation of the quantifiers we have used in our previous example.

First, we will divide time into discrete moments at which we will evaluate whether or not a statement is true. Thus, our first approximation is to take continuous time and break it into intervals. This is directly relevant to processors where events occur according to the clock cycles of the processor; however, we will usually use a much larger time quantum to specify our real-time system.

16.4.1.1 During the next time interval

First, we may want to say that if *f* is true at a given time, then *g* will be true during the next time interval. Use $\circ g$ to indicate that *g* is true during the next time interval. Thus,

$$f \rightarrow \circ g$$

says that *f* is true implies that *g* will be true during the next time interval. We may express that *g* will be true during the n^{th} time interval into the future with

$$f \rightarrow \circ^n g .$$

This says that if *f* is true, then *g* will be true after *n* time steps.

16.4.1.2 Now and always in the future (for all time from now on)

Next, we may wish to indicate that a statement is true for all time forward from the current moment in time. For this, we use $\Box g$. Thus,

$$f \rightarrow \Box g$$

says that if *f* is true, then *g* will be true for all future moments in time. This is equivalent to saying

$$\begin{aligned}\Box g &\leftrightarrow g \wedge \left(\circ g\right) \wedge \left(\circ^2 g\right) \wedge \left(\circ^3 g\right) \wedge \left(\circ^4 g\right) \wedge \cdots \\ &\leftrightarrow \bigwedge_{k=0}^{\infty} \circ^k g \\ &\leftrightarrow \forall k : \left(k \in \mathbf{Z}_0^+ \rightarrow \circ^k g\right)\end{aligned}$$

To remember that the square means *for all time from now on*, you may recall that the universal quantifier uses an up-side-down "A" and the next two letters are a pair of the letter el, which is reminiscent of a square:

16.4.1.3 Now or at some point in the future (there exists a time now or in the future)

Alternatively, we may wish to say that *g* will be true either now or at some point in the future. For this we use $\Diamond g$. For example,

$$f \rightarrow \Diamond g$$

says that if *f* is true, *g* will be true either now or at some point in the future. This is equivalent to saying

$$\begin{aligned}\Diamond g &\leftrightarrow g \vee \left(\circ g\right) \vee \left(\circ^2 g\right) \vee \left(\circ^3 g\right) \vee \left(\circ^4 g\right) \vee \cdots \\ &\leftrightarrow \bigvee_{k=0}^{\infty} \circ^k g \\ &\leftrightarrow \exists k \in \mathbf{Z} : \left(k \geq 0 \rightarrow \circ^k g\right)\end{aligned}$$

Again, differentiating the square and diamond may be frustrating, and thus, we provide the following image to remind you that this operator indicates that *there exists a time either now or in the future*, and whereas the existential quantifier is a backwards "E", the diamond can, at least, be superimposed over the "x":

Note that we may combine these to say, for example,

$\Box\Diamond f$ It will always be true that *f* will become true either now or at some point in the future.

$\Diamond\Box f$ Eventually *f* will always be true.

Thus, we may write *f* as a sequence of truth values: *f* = F, F, F, T, F, T, T, F, F, F, T, F, T, T, ….

16.4.1.4 Until

The final operator we will look at is *until*: *f* must be true until that point in time when *g* is true:

$$f \, \mathcal{U} \, g \, .$$

Note that it is required that *g* will be true either now or at some point in the future and it says nothing about the state of *f* after *g* is true. Thus, if *g* is true now, *f* need not ever be true.

16.4.1.5 Equivalencies

There are equivalent statements, for example,

$$(\Box f) \leftrightarrow \neg\left[\Diamond(\neg f)\right]$$

says that *f* is always true in the future is equivalent to saying that it is false that at some point in the future *f* will be false.

There are duality laws:

$$\neg(\circ f) \leftrightarrow \circ(\neg f)$$
$$\neg(\Box f) \leftrightarrow \Diamond\neg f$$
$$\neg(\Diamond f) \leftrightarrow \Box(\neg f)$$

For example, it is false that *f* is true now and always in the future is equivalent to saying that *f* is false now or at some point in the future. Similarly, if *f* is not true either now or at some point in the future is equivalent to saying that *f* is false now and always in the future.

Similarly, we may expand these:

$$\Box f \leftrightarrow f \wedge \circ\Box f$$
$$\Diamond f \leftrightarrow f \vee \circ\Diamond f$$
$$f \,\mathcal{U}\, g \leftrightarrow g \vee \left(f \wedge \circ(f \,\mathcal{U}\, g)\right)$$

These say that *f* is true now and in the future is equivalent to *f* is true now and at the at the next instance, *f* is true then and at all time in the future.

For example, we may want to discuss how requests are received:

1. it is true from now on that when a message is sent, it will be received either now or at some point in the future,
2. it is true from now on that once a message is received, it will be processed in the next time interval,
3. it is true from now on that once a message is processed, it will either now or at some point in the future be done.

We may write this by defining

1. MS as the message is sent,
2. MR as the message is received,
3. MP as the message is processed, and
4. MC as the message processing is completed.

Thus, we may rewrite our requirements as

1. $\Box(MS \rightarrow \Diamond MR)$,
2. $\Box(MR \rightarrow \circ MP)$, and
3. $\Box(MP \rightarrow \Box\Diamond MC)$.

Thus, it should be false that at some point, a message is sent, but it is never completed: $(\Box MS) \wedge \left[\Box(\neg MC)\right]$.

16.4.2 Examples of requirements

The following requirements on a system can be described in LTL:

1. liveness,
2. bounded response,
3. duration,
4. invariance,
5. safety,
6. fairness,
7. oscillation,
8. mutual exclusion,
9. signalling, and
10. rendezvous.

We have already seen many of these in previous chapters, but we will look at how these requirements can be described using LTL.

16.4.2.1 Liveness

The characteristic of liveness specifies what must occur. While it its simplest form, we could describe liveness as $\Diamond g$; however, such a requirement may be associated with another action. For example, the requirement that if *f* happens, then *g* must subsequently occur may be written as $\Box(f \to \Diamond g)$. For example, each request for memory must be eventually satisfied.

16.4.2.2 Bounded response

If an action must occur within a specified period of time, this could be written as

$$\bigvee_{k=1}^{n}\left(\circ^k g\right) = (\circ g) \vee \left(\circ^2 g\right) \vee \cdots \vee \left(\circ^n g\right),$$

where the use of V is similar to that of, for example, summation notation. As before, it may be written as having a bounded response to a particular event:

$$\Box\left(f \to \bigvee_{k=1}^{n}\left(\circ^k g\right)\right)$$

For example, if each memory request must be satisfied within the next time interval, we would have $\Box(f \to \circ g)$.

16.4.2.3 Duration

If an event must be true for a specific period of time, it could be written as

$$\bigwedge_{k=0}^{n}\left(\circ^k g\right) = g \wedge (\circ g) \wedge \left(\circ^2 g\right) \wedge \cdots \wedge \left(\circ^n g\right).$$

16.4.2.4 Invariance

If it is required that at some point that *g* must hold forever, this could be written as $\Diamond(\Box g)$. For example, suppose that each request for a service, f_k, must at some point be flagged as closed, g_k, this could be written as

$$f_k \to \Diamond(\Box g_k).$$

16.4.2.5 Safety

From this point on, the event *f* never occurs, or $\Box(\neg f)$. For example, the system may never be overloaded.

16.4.2.6 Fairness

An even occurs infinitely often, or $\square(\Diamond f)$. For example, the status checker will always have an opportunity to run.

16.4.2.7 Oscillations

The truth of *f* oscillates, or $\square\left[\left(f \wedge \circ(\neg f)\right) \vee \left(\neg f \wedge \circ f\right)\right]$. If at any time, *f* holds the same state between two time periods, both operands of the OR operator would be false, thus negating the requirement that one of the two is true for all time. For example, detection is turned on every second cycle.

16.4.2.8 Mutual exclusion

Two states cannot be true simultaneously, or $\square\neg(f \wedge g)$. For example, two tasks may not be executing their critical regions simultaneously.

16.4.2.9 Signaling

An event causes another to occur in the future, or $f \rightarrow \circ(\Diamond g)$. More specifically, an event causes another to occur within the next two time intervals, or $f \rightarrow \left((\circ g) \vee \left(\circ^2 g\right)\right)$

16.4.2.10 Rendezvous

A number of events signal subsequent events to occur, or $(f \wedge g \wedge h) \rightarrow (\circ\Diamond f' \wedge \circ\Diamond g' \wedge \circ\Diamond h')$.

16.4.3 Case study: automated aircraft control architecture

In an excellent paper by Kristin Y. Rozier of the NASA Ames Research Center in Moffett Field, California, the author describes an automated aircraft control architecture.[42] Her paper summarizes research from 1977 to 2009 and provides one example of many where software verification has been successfully used; other applications involving human safety considerations include airplane separation assurance, autopilot, CPU designs, life-support systems and medical equipment.

For this architecture, there are seven states, and each in state each of five atomic propositions is assigned a value of either *true* or *false*, including

1.	aircraft request	`ac_req`
2.	auto-resolver (AR) command	`AR_cmd`
3.	controller request	`ctrl_req`
4.	Tactical Separation Assisted Flight Environment (TSAFE) command	`TSAFE_cmd`
5.	TSAFE clear	`TSAFE_clear`

Using aircraft request as an example, when a state has this atomic proposition flagged as true, `ac_req` is used; while when it is false, `~ac_req` is used. Figure 16-1 is recreated from Rozier's paper.

[42] K.Y. Rozier, *Linear Temporal Logic Symbolic Model Checking*, Computer Science Review (2010), doi:10.1016/j.cosrev.2010.06.002

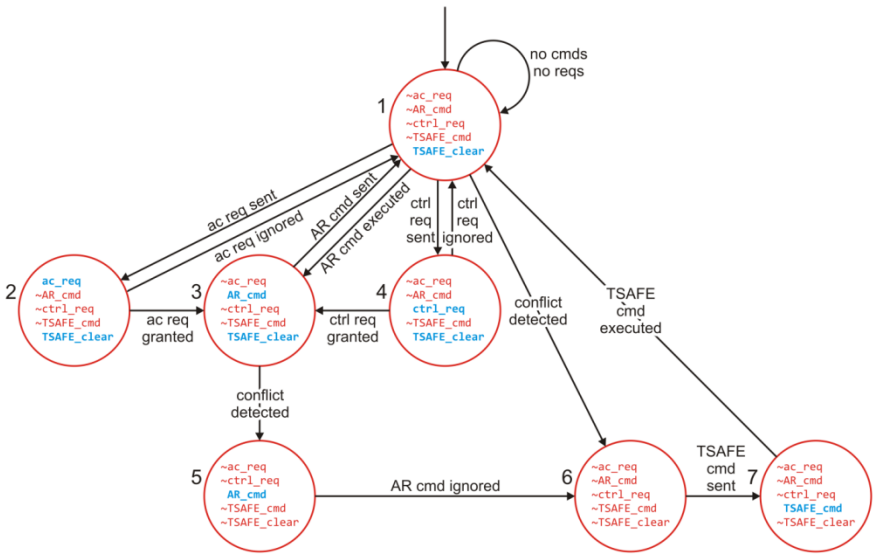


Figure 16-1. State-transition graph for an aircraft control architecture (see K.Y. Rozier).

There are four liveness specifications for this architecture:

1. every conflict is addressed,

$$\square\left(\neg \text{TSAFE\_clear} \rightarrow \lozenge \text{TSAFE\_cmd}\right)$$

2. all conflicts are eventually resolved,

$$\square\left(\left(\neg \text{TSAFE\_clear} \rightarrow \lozenge \text{TSAFE\_clear}\right)\right)$$

3. all controller requests are eventually addressed, and

$$\square\left(\text{ctrl\_req} \rightarrow \lozenge\left(\neg \text{ctrl\_req}\right)\right)$$

4. all aircraft requests are eventually addressed.

$$\square\left(\text{ac\_req} \rightarrow \lozenge\left(\neg \text{ac\_req}\right)\right)$$

There are two safety specifications for this architecture:

5. every conflict is addressed in one time interval, and

$$\square\left(\neg \text{TSAFE\_clear} \rightarrow \circ \text{TSAFE\_cmd}\right)$$

6. the system will never issue conflicting commands.

$$\square\left(\neg\left(\text{AR\_cmd} \wedge \text{TSAFE\_cmd}\right)\right)$$

16.5 Computation tree logic (CTL)

In addition to LTL, there is a second temporal logic termed *computation tree logic* (CTL) that considers branching time. Here, every temporal statements prefixed by either the requirement that

1. every future path has the temporal statement being true, or
2. there exists a path such that the temporal statement is true.

For example, the following would be read as

1. $\exists \Diamond g$ means that there exists a path where g will eventually be true,
2. $\forall \Diamond g$ means that for all future paths, g will eventually be true,
3. $\exists \Box g$ means that there exists a future path along which g will always be true,
4. $\forall \Box g$ means that for all future paths, g will always be true,
5. $\exists \circ g$ means that there exists a future path such that g will true at the next instance, and
6. $\forall \circ g$ means that for all future paths, g will be true at the next instance.

16.6 Model checkers

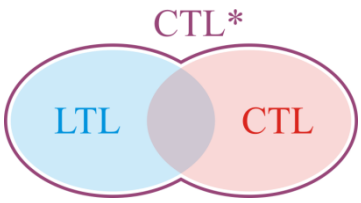
The two temporal logics LTL and CTL are not equivalent: there are requirements that can be stated in one that cannot be stated in the other. For example, using an example from Benoît Fraikin, a requirement that can be made in CTL that cannot be made in LTL is:

> "for all future states along possible paths, there exist a path to a state such that the system can be safely reset, even if that state is never reached."

or symbolically $\forall \Box (\exists \Diamond R)$. LTL can only state that at some point the system will be reset, or $\Diamond R$. A similar statement in LTL that cannot be made in CTL is:

> "at some point, the system will always be safe thereafter to shut down"

or symbolically $\Diamond (\Box S)$. For a complete paper, see *Branching vs. Linear Time: Final Showdown* by Moshe Y. Vardi. Thus, you can consider LTL and CTL to be two separate approaches to describing temporal requirements. If we allow both types of statements, the system is called CTL*, as is shown in .



The problem is we want to be able check whether or not our model satisfies our requirements. Currently, there are logic checkers for LTL and ones for CTL; however, there is no logic checker for both. If you wish to perform software verification, you must choose one. Later, we will look at Uppaal, which uses CTL. Unfortunately, CTL is less intuitive than LTL, which is more *linear* in its conception. From Vardi's paper, consider the two statements $\Diamond (\circ f)$ and $\circ (\Diamond f)$; both of these state that f will occur at some point in the strict future. On the other hand, $\forall \circ (\forall \Diamond f)$ and $\forall \Diamond (\forall \circ f)$ mean two different things. The first says "for all next instants, for all paths from that next instance, there is a point where f is true", which is equivalent to $\circ (\Diamond f)$. The second says that for all future paths, there is a point such that at all next instances, f is true—this is a much more restrictive definition. It is generally more difficult to work with CTL.

16.7 Modelling software

There are numerous products out there that are available. One product, Uppaal, the result of collaboration between

1. the Design and Analysis of Real-Time Systems group at the Uppsala University, Sweden and
2. Basic Research in Computer Science at Aalborg University, Denmark.

This is a free environment that is developed in Java. This allows more significant control over variables and cause-and-effect. For example:

1. an edge being crossed may modify the value of a shared variable, and
2. other edges can be triggered on those changes to shared variables.

On the other hand, signalling is also possible:

1. an edge being crossed may signal a variable, and
2. other edges can be triggered on those signals.

Variables can be synchronized with clocks. As a very simple example that is shown in a tutorial by Behrmann et al. is a light switch that can be activated by a user:

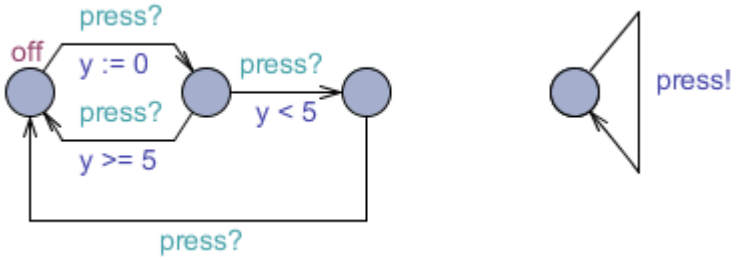


Figure 16-2. A light switch.

16.8 Summary of software verification

In this topic, we have introduced software verification. We have discussed user or client needs, and then reviewed propositional logic to show how some needs and requirements can be formulated using such logic; however, this is often insufficient. Predicate logic allows us to make statements of either universal existence or non-existence of specific states, but this too is insufficient, so we introduce both linear temporal logical and computation tree logic as possibilities for quantifying user needs and requirements. We described model checkers and modelling software.

Problem set

Some examples are taken from Lewis Carroll (yes, *that* Lewis Carroll—the author of such great books as Alice in Wonderland, Through the Looking Glass, Symbolic Logic, and Games in Symbolic Logic).

16.1 Implication is the last logical operator you have seen. Using mathematics, we would write $a \rightarrow b$; however, in expressing an implication in English, we may use phrases such as these:

1. If *a*, *b*.
2. *b* if *a*.
3. Whenever *a*, *b*.
4. Either not *b* or *a*.
5. When *a* then *b*.

However, expressions in English can be much more obscure; for example,

> The only animals that belong to me are in that field.

is the implication, but we should write it as:

> If an animal belongs to me, it is in that field.

Why is it wrong to interpret the statement as the following?

> If an animal is in the field, it belongs to me.

16.2 What implications are in the following statements?

1. The actuator is only triggered if the sensor signals an interrupt.
2. The sensor sending an interrupt will trigger the sensor.
3. The actuator is triggered when the sensor signals an interrupt.
4. If the actuator is triggered, the sensor must have signaled an interrupt.
5. The actuator is always triggered immediately after the sensor sends an interrupt.
6. The actuator must always be triggered immediately after the sensor sends an interrupt.

These will be one of:

1. the sensor signals an interrupt $\rightarrow$ the actuator is triggered
2. the actuator is triggered $\rightarrow$ the sensor signals an interrupt
3. the sensor signals an interrupt $\leftrightarrow$ the actuator is triggered

The third is simply $(a \rightarrow b) \wedge (b \rightarrow a)$.

16.3 Restate the following as implications.

1. All the circuit elements in this project have a tolerance of less than 5 %.
2. None of the circuit elements are manufactured by NXP, except those involved in the sensors.
3. I have not tested any that were purchased last week.
4. All that are not verified are manufactured by NXP.
5. All made by Phillips are tagged for inspection.
6. Those with a tolerance of less than 5 % are tested.
7. No circuit element used for the past year is not verified.
8. No circuit element that is tagged for inspection is involved in the sensors.

16.4 Which of the following are equivalent to the contrapositive, that is $(a \to b) \leftrightarrow (\neg b \to \neg a)$?

$$(a \to \neg b) \leftrightarrow (\neg b \to \neg a)$$
$$(a \to \neg b) \leftrightarrow (b \to \neg a)$$
$$(\neg a \to b) \leftrightarrow (b \to \neg a)$$
$$(\neg a \to b) \leftrightarrow (\neg b \to a)$$
$$(a \to b) \leftrightarrow (b \to \neg a)$$
$$(\neg a \to \neg b) \leftrightarrow (b \to a)$$
$$(a \to \neg b) \leftrightarrow (\neg b \to a)$$

16.5 What is the most general deduction you can make from Question 16.3? For example, consider the two statements:

1. All that are not verified are manufactured by NXP.
2. No circuit element used for the past year is not verified.

From this, we may deduce that "No circuit element used in the past year is manufactured by NXP."

16.6 Suppose that:

1. It takes 1 ms to wake up a microcontroller that is sleeping.
2. The microcontroller will be sleeping 99.9 % of its lifetime.
3. A functioning microcontroller can respond to an event in 200 μs, after which it can be put back to sleep.
4. If an event occurs, the next event will be within the next 350 μs or the next event will not occur for at least another 10 s.

Deduce a set of rules that can govern when the microcontroller can be put to sleep.

16.7 Suppose you know that a user will mostly likely give the correct password after five tries and it takes one second per try. Suppose also you never want to lock out the system, but at the same time, you want to make it progressively more difficult for someone trying to guess the password. Finally, the user should be able to make the five tries within 20 seconds. Finally, if a reasonable amount of time has passed since the last attempt at a password, the time required to wait between attempts should go down. Write down a set of rules to control such an interface.

16.8 A sensor with a mass of 13.5 g and made mostly of carbon steel cannot exceed a temperature of 120 °C. In the worst case, the temperature will increase at 2 °C/s. Carbon steel has a specific heat of 0.49 J/(g·K). To wake up the fan takes 0.5 s and once it starts (say at time t_0), it dissipates $p\frac{t-t_0}{t-t_0+1}$ J/s but uses p^2 J of energy. Recommend a strategy for maintaining the temperature of the sensor below the given temperature and argue why yours is a reasonable strategy.

Recall that $\forall x : f(x)$ says that *f*(*x*) is true for all values of *x* in our *universe of discourse* (the set of things we are discussing), a set that is assumed to be non-empty. Similarly, $\exists x : f(x)$ says that there is a value of *x* in our *universe of discourse* such that *f*(*x*) is true.

16.9 Argue in English that $\neg\left[\forall x : f(x)\right] \leftrightarrow \exists x : \neg f(x)$ and $\neg\left[\exists x : f(x)\right] \leftrightarrow \forall x : \neg f(x)$ are always true.

16.10 What can you deduce, if anything, from each of the following:

1. $\left[\forall x : f(x) \rightarrow g(x)\right] \wedge \left[\forall x : g(x) \rightarrow h(x)\right]$
2. $\left[\forall x : f(x) \rightarrow g(x)\right] \wedge \left[\exists x : g(x) \rightarrow h(x)\right]$
3. $\left[\exists x : f(x) \rightarrow g(x)\right] \wedge \left[\forall x : g(x) \rightarrow h(x)\right]$
4. $\left[\exists x : f(x) \rightarrow g(x)\right] \wedge \left[\exists x : g(x) \rightarrow h(x)\right]$

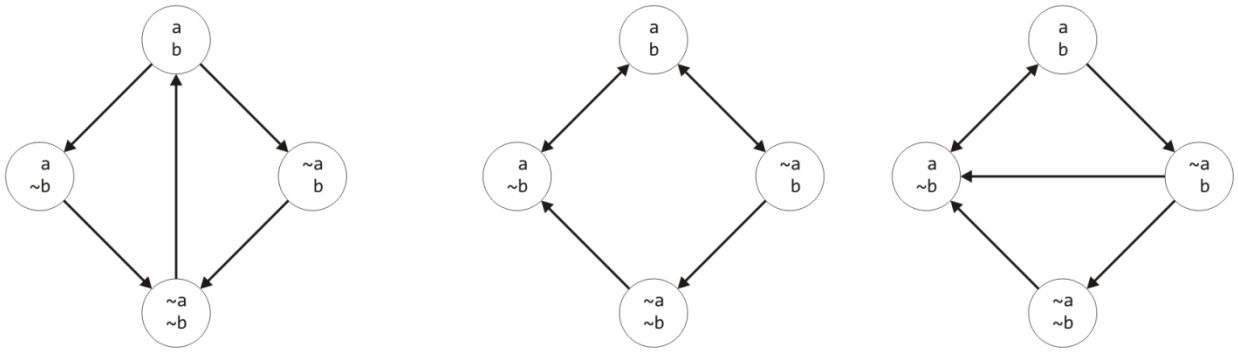
16.11 What does the following about how many elements in our collection satisfy the condition *f*?

$$\exists x : \forall y : f(x) \wedge \left[(x = y) \vee \neg f(y)\right]$$

16.12 Using temporal logic, how would you indicate that the microcontroller is sleeping until an event occurs?

16.13 Using temporal logic, how would you indicate that the microcontroller is sleeping until the moment in time after an event occurs?

16.14 Consider the following three systems.



For each of the following statements, describe in English what it says.

For each system, which of the following statements are true, and if a statement is false, recommend a change that makes it true. Note that you may assume that if an edge is indicated, it will be triggered at some point in the future.

1. $\Box\left(\neg\left(\neg a \rightarrow \neg\left(\circ a\right)\right)\right)$
2. $\Box\left(\neg\left(\neg a \rightarrow \left[\neg\left(\circ a\right) \wedge \neg\left(\circ \circ a\right)\right]\right)\right)$
3. $\Box\left(\left(a \wedge b\right) \rightarrow \circ\left(a \vee b\right)\right)$
4. $\Box\left(\neg\left(a \vee b\right) \rightarrow \circ\left(a \wedge b\right)\right)$
5. $\Box\left(a \vee b\right)$
6. $\Box\left(\neg a \rightarrow \circ\left(a \vee \neg b\right)\right)$

For each system, which of the above statements are true, and if a statement is false, recommend a change by adding or removing edges that makes it *true*. Note that you may assume that if an edge is indicated, it will be triggered at some point in the future.

17 File management

A file is a named resource organized as a one-dimensional array of bytes that is available for the persistent storage and retrieval of information. Unlike data structures stored in main memory, a file outlasts the task that created or modified it. A file system is an abstract data type (ADT) for managing long-term storage of data in much the same way that a memory allocator is an ADT for managing access to main memory.

In general, for each file, a file system will store:

1. the bytes associated with the data,
2. a name by which the file can be accessed, and
3. other associated meta-data.

The organization of files usually follows a hierarchical structure where each device contains a *directory* or *folder* where a directory can hold files and other directories. In environments where there are multiple persistent storage devices, usually one of two strategies is employed:

1. each storage device is given its own identifier (a *drive* in Windows), or
2. one storage device is given precedence, and other storage devices becomes *folders* within a folder of the main storage device (the Linux approach).

In embedded systems, this is seldom a cause for concern, as there will seldom be more than one storage device. For example, the Spirit and Opportunity rovers have a flash drive available for temporary storage.

For a file system to function correctly, it is necessary at least some information be kept in main memory as part of an appropriate data structure that can be accessed by appropriate functions. Any information in this data structure must also be stored on the device, or at least, it must be possible to reconstruct the information. It was the size of this data structure that caused the reboot cycle on the Spirit rover.

We will start by discussing block addressability of drives, we will then describe files and the organization of files into directories briefly, we will then look at file systems, data formats, and the file abstraction of Unix. We will finish with a look at the file systems available with the Keil RTX RTOS.

17.1 Block addressable

Recall that main memory tends to be byte addressable. If you want to access an individual bit, you must never-the-less load (at least) the byte containing the bit into a register. Similarly, hard-disk drives (HDD) and flash drives (SSD) are block addressable. A block tends to be approximately 4 KiB in size, and while this is variable, there is a trend toward using this size—this will be emphasized again in the next topic. If you want to access a byte from a block on the drive, you must load the entire block into main memory. Similarly, if you want to save a modified bit to the drive, you must write the entire block to the drive.

A file will also occupy an integral number of blocks. A consequence of this is that if blocks are 4 KiB in size and a file is 12542 bytes in size, as three blocks are 12 KiB = 12288 B, such a file must therefore occupy four blocks and therefore 3842 bytes are wasted. Thus, when deciding the block size, smaller blocks imply less memory can be accessed (with 32-bit hard drive addresses, 4 KiB blocks means that 16 TiB can be addressed, but with 16-bit hard drive addresses, only 256 MiB can be addressed).

17.2 Files

A file is a contiguous block of characters. If data is not contiguous and it is to be stored as a file, it must be converted to a contiguous block; for example, if you wanted to store an AVL tree, for example, as represented in Figure 17-1, in a file, you must convert it into a sequence of characters.

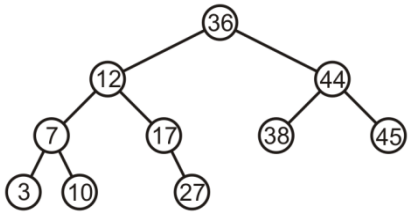


Figure 17-1. An AVL tree.

For example, you may end up converting the tree structure into an ordered list,

```
3,7,10,12,17,27,36,38,44,45
```

and then when the data is read from the file, you would reconstruct the tree.

A file will be stored in an integral number of blocks, and therefore a consequences of having 4 KiB blocks is that there will be internal fragmentation: most files will not occupy an integral number of blocks, and therefore the last block will contain unused drive space. For example, a file is 12542 bytes in size is larger than three blocks, or 12 KiB = 12288 B. Therefore it is necessary to allocate four blocks capable of storing 16384 B with 3842 B unused. The additional drive space is wasted cannot be used for another file—even if that file perfectly fits into that location.

Note that the block size can be set when the drive is formatted. Consequently, you could have very large blocks, but this results in more wasted space. Smaller blocks, however, require separate accesses to the drive to load them into memory, requiring more overhead. In addition, file systems tend to have a fixed upper bound on the number of blocks, so if the block size is too small there may be portions of the device that are inaccessible.

File names and directory are generally identified through human-readable names and thus they use either ASCII, or more recently, UNICODE characters. Thus, each file requires at least three pieces of information to be stored:

1. the name,
2. the location on the drive (the addresses of the blocks), and
3. the size of the file.

This additional information is called *metadata* and will need to be stored elsewhere.

17.3 Organization

By default, a file system could simply store a number of files; however, this makes organization very difficult, and consequently, a tree structure is used to store data: each node of a tree is termed a directory, and each directory can store a fixed (although often large) number of files as well as containing sub-directories. The base of the tree is called the *root directory*. The possible actions in a file system are:

1. access the root directory,
2. move to a sub-directory of the current directory,
3. move up to the parent directory,
4. move to a directory relative to the root directory, and
5. move to a directory relative to the current directory.

Operations in a directory include:

1. listing,
2. creating,
3. moving, and
4. deleting

files or directories.

17.4 File systems

A *file system* is a collection of data structures storing metadata describing the directory structure and files stored within a drive. It must be possible to store the entire file system on the drive itself, as the drive may be transported from one system to another; however, when a file system is *mounted* on a particular system, a portion of the data structures must be loaded into main memory. At the very least, the data structure describing the root directory must be loaded into main memory. In addition to these data structures, there must be an interface that the user can access.

We will look at data structures for storing both files and directories. In both cases, because drives are block addressable, the data structures will often be tailored this situation. We will then discuss issues such as fragmentation of hard disk drives (HDD) and how to ensure fault tolerance using journaling. Finally, we will review the issue with the Spirit Mars Rover.

17.4.1 Sample file structures

We will look at two file structures:

1. file allocation tables (FAT), and
2. inodes.

17.4.1.1 File allocation tables (FAT)

An older file system is known as File Allocation Tables (FAT), a system that was first introduced at Microsoft. For example, FAT 12 allows up to 2^{12} blocks and the blocks are numbered 0, 1, 2, …, 4095. Now, we will look at a simplified version of FAT; however, the actual implementation uses these ideas at the core. The hard drive is divided into three sections:

1. a section associating the file names with the number of the first block containing them,
2. a section of size $2^{12} \times 12$ bits = 6144 B used to record all the blocks associated with the various files, and
3. a section of size at most 2^{12} blocks storing the actual files.

For example, suppose we had three files as follows:

```
README.TXT           7       2058
```

```
SETUP.EXE              27      1039259
REMOVE.EXE             48      759285
```

Suppose that the block size is 1 KiB. Looking at the sizes, we note that $\left\lceil \frac{2058}{2^{10}} \right\rceil = 3$, $\left\lceil \frac{1039259}{2^{10}} \right\rceil = 1015$ and $\left\lceil \frac{759285}{2^{10}} \right\rceil = 742$, so together we are using 1760 blocks out of the 4096 possible blocks available (note that a drive may not have all possible blocks). The memory for the files is 1800602 bytes, but the memory used on the drive will be 1802240 (not including the overhead for the meta-data).

The next section is nothing more than a linked list of entries. Every block is associated with a 12-bit entry in this table, and

1. each entry stores the address of the next block associated with the given file, and
2. the entry associated with the last block of a given file is its own address.

In this case, it is only necessary to walk through the linked list to determine all the blocks associated with a file.

When you deleted a file, it only removed the name from the look-up table. The linked list of next pointers was not changed, so it was reasonably easy to detect deleted files that could be recovered.

Suppose we had a simpler system, say FAT 4, capable of storing up to 2^4 = 16 blocks.

```
README.TXT     0       1258
SETUP.EXE      3       6835
REMOVE.EXE     7       2783
```

Thus, a little math shows that we require 2 + 7 + 3 blocks. Suppose that the allocation of blocks is as follows:

```
README.TXT     0 1
SETUP.EXE      3 9 c 6 4 5 8
REMOVE.EXE     7 a d
```

The linked list component would then look as follows:

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
1	1		9	5	8	4	a	8	c	d		6	d		

Of course, it would be useful to have a linked list of unused blocks:

```
README.TXT     0       1258
SETUP.EXE      3       6835
REMOVE.EXE     7       2783
UNUSED         2       -
```

Thus, our table would look like:

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
1	1	b	9	5	8	4	a	8	c	d	e	6	d	f	f

Note that we cannot use 0 to indicate the end of a file, as 0 is a valid address. Similarly, because we are using only 4 bits to store the linked list of points, we cannot use, for example, -1, to store the end of the linked list, either. By duplicating the next pointer, we can still recognize the end of the linked list without allocating additional memory.

While file allocation tables (FAT) have an historical association with MS-DOS, and it is occasionally derided

because of that, it is a reasonable approach in many cases where only simple file systems are required. FAT is simple and widely supported system for managing files and accordingly many devices designed for high interoperability are formatted using the FAT file system.

17.4.1.2 Unix index nodes or inodes

The initial Unix inode is a hybrid index-based and node-based data structure used for storing the locations of the sequential blocks of a file. Unlike FAT, inodes allow for Θ(1) access to any block within the data structure. The structure contains meta-data about the file, including

1. the size of the file in bytes,
2. identifier for the device storing the file,
3. identifier of the file's owner,
4. identifier of the file's group,
5. the file mode for read, write and execute for the owner, group and global access,
6. additional flags,
7. various time stamps (last time the inode was modified, last time the file was modified, and the last time it was accessed), and
8. the number of hard links to the inode.

The first 12 addresses store the addresses of the first twelve blocks of memory. Thus, files no greater than 4 KiB require only one entry in the inode, as shown in Figure 17-2.

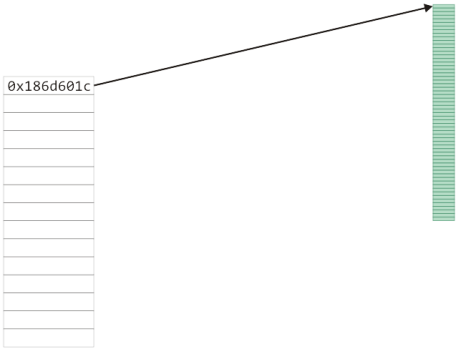


Figure 17-2. An inode storing the address of one 4 KiB block.

Files up to 48 KiB would have the first 12 blocks stored in the first twelve indices, as shown in Figure 17-3.

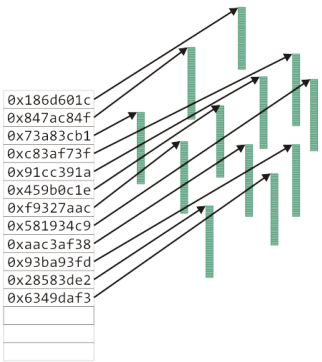


Figure 17-3. An inode storing twelve addresses for a file greater than 44 KiB but no greater than 48 KiB.

The next pointer in the inode stores the address of an *indirect block* that will be used to store the addresses of the next 1024 blocks of the file. The file in Figure 17-4 requires 19 blocks and therefore the inode itself stores the address of the first 12, and the first indirect block stores the addresses of the next 7.

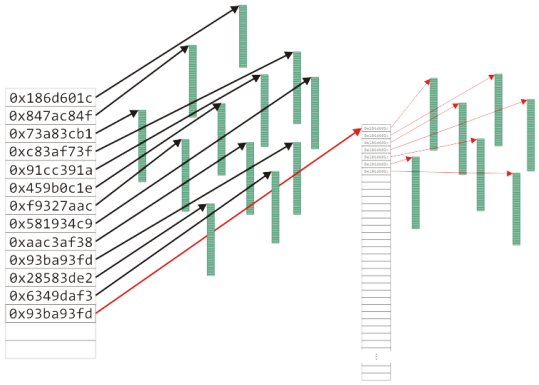


Figure 17-4. An inode storing a file requiring 19 blocks.

Suppose now that the file requires more than 12 + 1024 = 1036 blocks (that is, greater than 4144 KiB (or 4 $\frac{3}{64}$ MiB)). The next address of the inode stores the address of a *double indirect block*. This in turn stores the address of 1024 indirect blocks, each of which stores the address of 1024 blocks.

For example, the inode in shows Figure 17-5 how 12 + 1024 + 9 = 1045 blocks could be used to store a file requiring more than 4176 up to 4180 KiB.

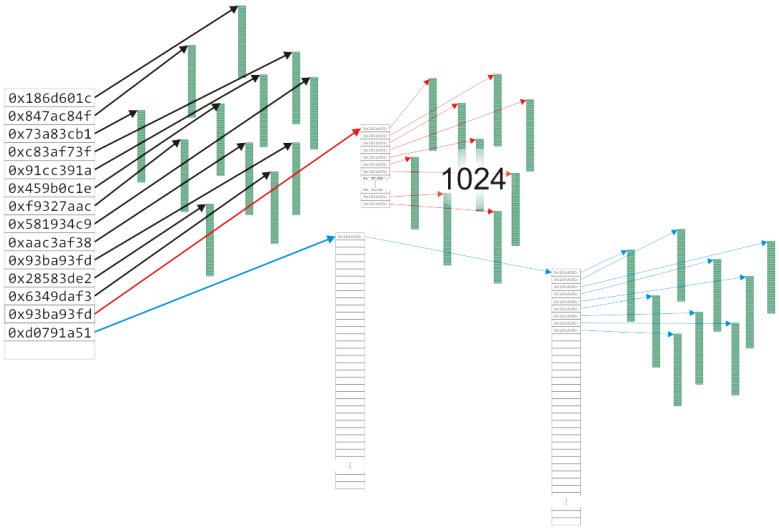


Figure 17-5. The inode of a file occupying 1045 blocks.

If another 1024 blocks are required, we could use the next entry of the double indirect block to store another indirect block, as shown in Figure 17-6.

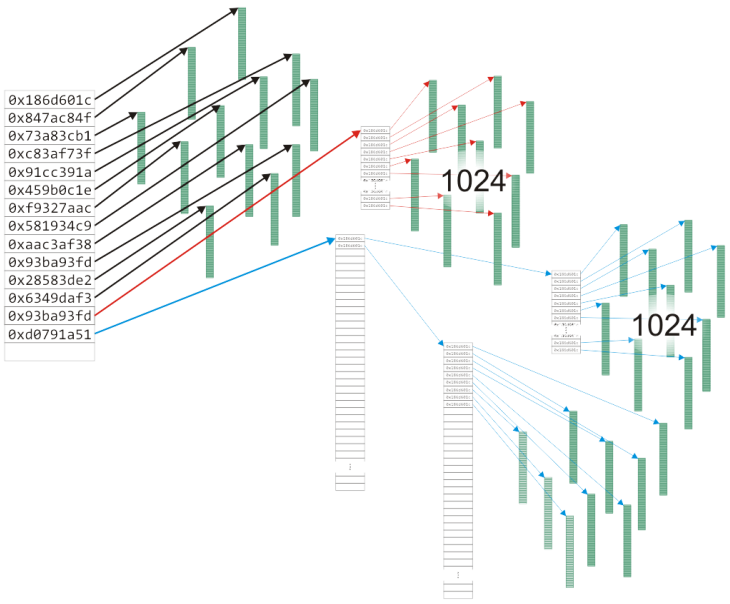


Figure 17-6. The inode of a file occupying 12 + 2048 + 9 = 2069 blocks.

Finally, suppose we fill the entire double indirect block. This would require a file greater than

$$12 + 1024 + 1024^2 = 1049612$$

4 KiB blocks, or $4100\frac{3}{64}$ MiB. The next pointer in the inode stores the address of a triple indirect block, which stores the addresses of 1024 double indirect blocks, each of which storing the address of 1024 indirect blocks, each of which stores the address of 1024 blocks, allowing a maximum file size of

$$12 + 1024 + 1024^2 + 1024^3 = 1074791436$$

blocks or approximately 4.0039 TiB.

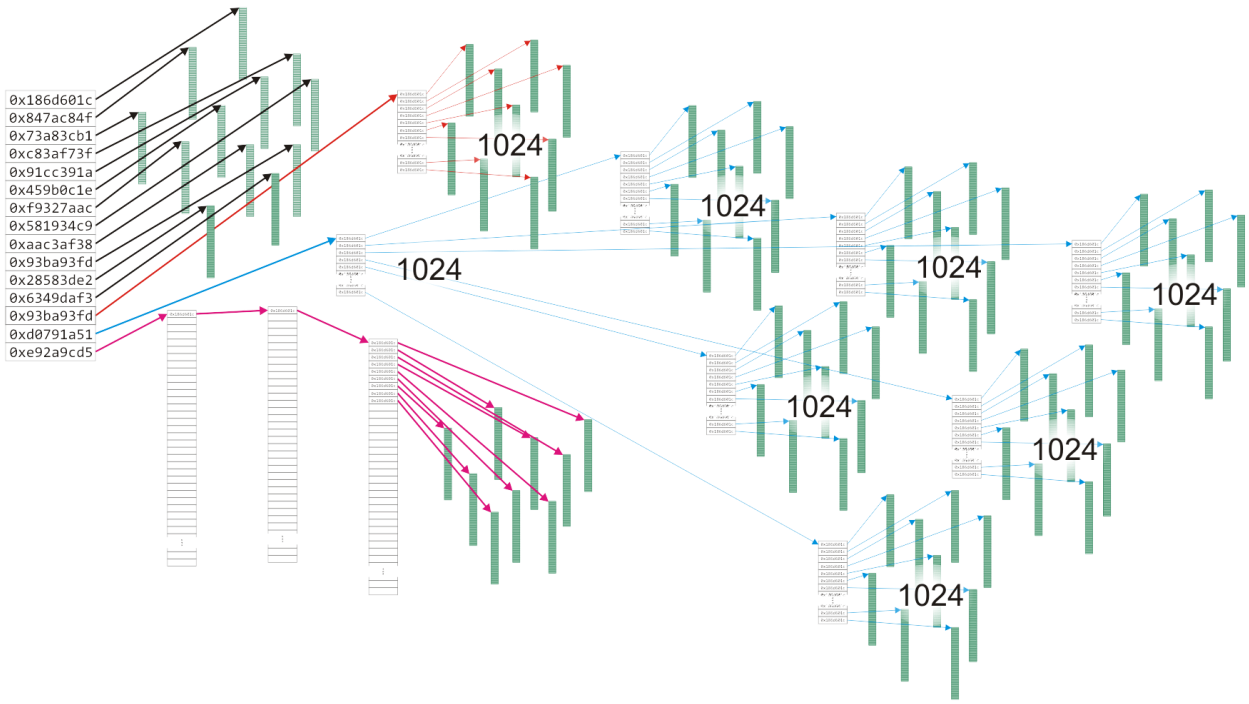


Figure 17-7. The inode of a file occupying 1049621 blocks.

The following code would access the block containing the n^{th} byte.

```
#define BLOCK_SIZE   4096
#define ADDRESS_SIZE    4
#define REDIRECT_ENTRIES (BLOCK_SIZE/ADDRESS_SIZE)
#define REDIRECT_MASK    (REDIRECT_ENTRIES - 1)
int index1 = n/BLOCK_SIZE;

if ( index1 < 12 ) {
    load( inode[index1] );
} else {
    index1 -= 12;

    if ( index1 < REDIRECT_ENTRIES ) {
        load ( inode[12][index1] );
    } else {
        index1 -= REDIRECT_ENTRIES;

        if ( index1 < REDIRECT_ENTRIES*REDIRECT_ENTRIES ) {
            int index2 = index1/REDIRECT_ENTRIES;
            index1 &= REDIRECT_MASK;
            load( inode[13][index2][index1] );
        } else {
            int index3 = index1/REDIRECT_ENTRIES/REDIRECT_ENTRIES;
            int index2 = (index1/REDIRECT_ENTRIES) & REDIRECT_MASK;
            index1 &= REDIRECT_MASK;
            load( inode[14][index3][index2][index1] );
        }
    }
}
```

17.4.1.3 Summary of file structures

We have looked at two file structures: the file allocation table (FAT), an almost ubiquitous file descriptor recognized universally for data transfer and access, and the Unix inode, a data structure still in use today in modern Linux file systems.

17.4.2 Sample directory structure (B+-trees)

In general, a directory stores an ordered list of the files and subdirectories stored within the directory, together with any metadata about them. A subdirectory may be recorded by a name, permissions, creation date, as well as other information together with the location of the data structure describing the directory. A file may be recorded as an inode.

To store ordered data that can easily be modified, one may consider using an AVL tree or some other balanced search-tree structure. Unfortunately, this doesn't work well with the blocks of a hard drive. Instead, we will look at a variation on a binary search tree. We will consider three changes:

1. allow each leaf to occupy a block on the drive and store most of the information in the leaves,
2. allow each internal node to occupy a block on the drive, storing only information directing the search to the appropriate leaf, and
3. impose rules to keep this structure balanced.

The resulting data structure is called a *B+-tree* and is ubiquitous in both file systems and databases for these reasons.

17.4.2.1 Information in leaves

A leaf node should occupy a 4 KiB block, and therefore each leaf node can contain information about a lot of different files and directories. The files and directories would be linearly ordered (as in an array) in such a block, and while this is slightly sub-optimal, the time it would take to rearrange a block of file and directory data structures in 4 KiB of memory is negligible compared to the amount of time it takes to either load or save that block from or to a drive. We will also not require that the leaf nodes are full. This has the added benefit that if by adding an additional file fills up a leaf block, we can just split it into two half-full blocks.

For example, suppose that the author has his collection of U2 albums and singles stored in a single directory, but where all singles are stored as files, all songs from albums are stored in subdirectories. In this case, suppose each leaf node can store 24 records associated with either files or subdirectories. In this case, the directory may initially be stored as a single leaf node, as shown in Figure 17-8.

Figure 17-8. A single leaf node containing 18 files and directories.

Suppose now that the author goes out and purchase all albums after "The Joshua Tree". This would increase the number of files and directories to 25, so this requires a second leaf node, and the files and subdirectories would be split between the two, as shown in Figure 17-9.

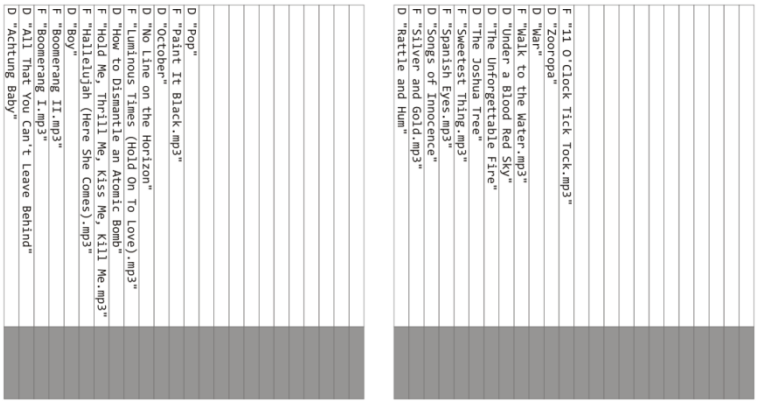


Figure 17-9. Two leaf nodes storing 25 files and subdirectories.

Assuming each leaf node occupies one block on the drive, these two nodes now occupy two blocks on that drive. The question now is: how do we associate the two?

17.4.2.2 Internal nodes

If all the files and subdirectories fill only a single leaf node, that leaf node can represent the entire directory; otherwise, we to expand our tree structure. In a binary search tree, a node could contain the smallest file name in the right child, in this case, "Rattle and Hum". Now, if you try to access the metadata for the album "October", you would have to look in the left child, while if you were looking for the album "War", you would look in the right child, as shown in Figure 17-10.

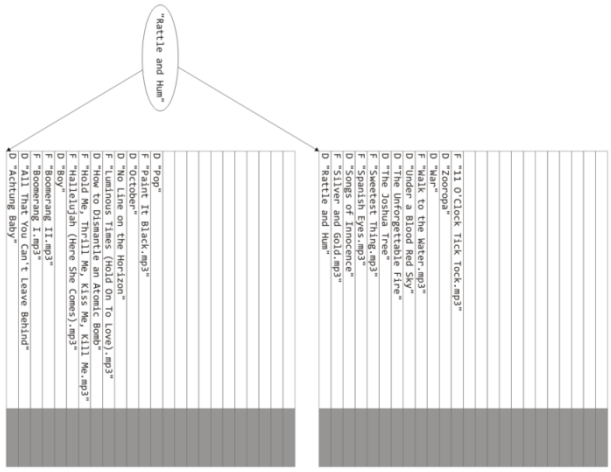


Figure 17-10. A binary search node pointing to two directories.

The problem with this, however, is that if there are now three or more leaf nodes, you essentially have an AVL tree where the leaf nodes must now be stored somewhere on the drive. You could make an attempt to ensure that all these nodes are stored in the same block, but this may become difficult as the directory or database grows in size. Instead, we will use a different approach. Suppose we are storing all homophones[43] in English, and each word is linked with its homophones in the leaf nodes. Rather than having a binary node, let's have a 4-way node, which stores pointers to four children and it stores the smallest value in the second, third and fourth children, as shown in Figure 17-11.

[43] The list of homophones comes from Ian Miller's page
http://www.singularis.ltd.uk/bifroest/misc/homophones-list.html

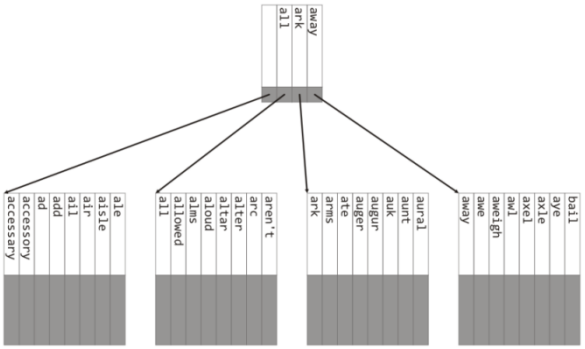


Figure 17-11. One level of a 4-way node.

Now,

1. any word that has a homophone that appears alphabetically before “all” must be in the first child,
2. any word having a homophone after and including “all” but before “ark” is in the second child,
3. words after and including “ark” but before “away” are in the third child, and
4. any word after and including “away” is in the last child.

Once we reach the child, we can search for the word and find its homophones (if any). Now, this only allows $4 \times 8 = 32$ words to appear in the leaf nodes. Thus, we can add one more level. In Figure 17-12, you will see that new top-level node stores four pointers to children, and three values: the smallest word in the second, third and fourth sub-trees.

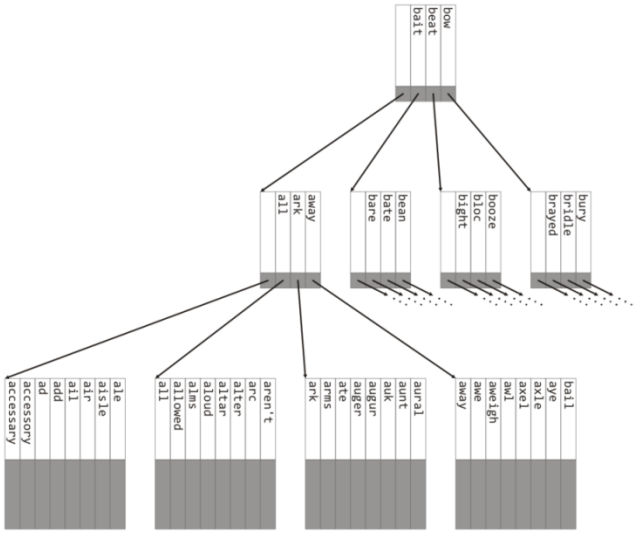
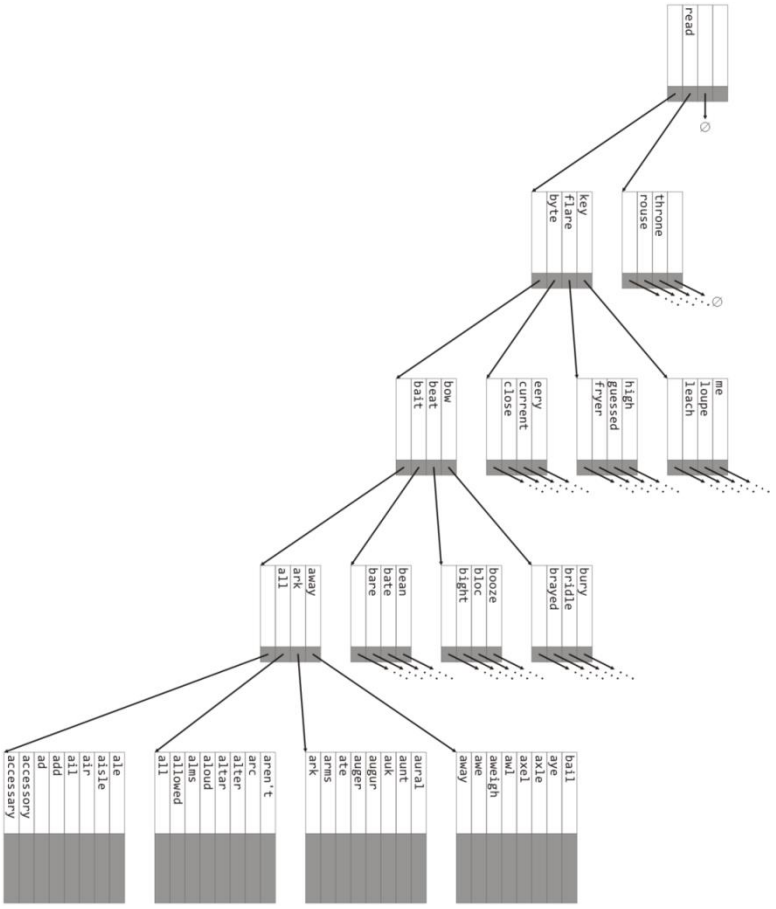


Figure 17-12. Two levels of 4-way trees.

Again, searching for a word simply requires us to determine which sub-tree the word is in, and then we can again search forward until we find the correct leaf node. For example, searching for “bazaar”, we would search the second subtree of the root: “bazaar” appears alphabetically after “bait” but before “beat”. Then, looking at the next node, we follow the third subtree, as “bazaar” appears alphabetically after “bate” but before “bean”. We would then search the leaf node for the word and its homophone (“bizarre”). This now allows us to store $4^2 \times 8 = 128$ words. To store all 934 homophones, we must have four levels of 4-way trees; however, the root node need point to only two children.



The height of this tree (including the leaf nodes) is $h = 4$, and a 4-way tree with L entries in the leaf nodes can store $4^h \times L$ entries. If we tried to create a binary search tree where each node simply stored the homophones of the word corresponding to that node, we would require a tree of height at least 10.

In reality, the trees stored on hard drives are even more extreme than 4-way trees. If we are using blocks of memory anyway, why not use the full block for the internal nodes. Suppose, for example, that our block is 4 KiB in size and that the addresses of a block is 4 bytes while the identifier is 28 bytes. Thus, we could store a 128-way tree in each node. Thus, if each leaf node stored $L = 64$ inodes, a 128-way tree of height $h = 3$ could store inodes for up to $128^3 \times 64 = 134217728$ or 134 million entries. Most directories will be only a leaf node or a leaf node together with one internal node (up to 8192 files or subdirectories).

The beauty of this system is that only a very small number of internal nodes must be copied from the drive to main memory at any one time. The only problem is that the tree we set up cannot be changed: what happens if we determine that, "caret" and "carrot" should be labeled as homophones. In the above scheme, we would essentially load all the leaf nodes into main memory and reshuffle all of them to insert the new entry. There are better ways of keeping balance.

17.4.2.3 Maintaining balance: B+-trees

Like AVL trees and red-black trees, it is possible to impose a set of rules on the above tree structure that ensures balance (n objects are guaranteed to be stored in a tree of height $\Theta(\ln(n))$). Suppose that the leaf nodes contain up to L entries each, and the internal nodes are M-way nodes. The rules are:

1. If there are L or fewer entries, the root node is a leaf node, otherwise
2. all three of the following must hold:
    a. the leaf nodes are at least half full and are at the same depth,
    b. the root node is an M-way tree with at least two children, and

c. all other internal nodes are *M*-way trees with at least $\left\lceil \frac{M}{2} \right\rceil$ children.

You may think that it is exceptionally difficult to ensure that all children are at the same depth, but this can be achieved as follows. If we are trying to insert a new entry, we proceed as follows:

1. Attempt to insert the entry into the leaf node where the tree indicates the entry should be, but if it is already full, got to Step 2, otherwise we are finished.
2. Split the node into two nodes and distribute the entries evenly between the existing node and the new one, then either
    a. if there is a parent, insert a new entry into the parent of the existing node, but if the parent node is already full, return to Step 2, otherwise we are finished, or
    b. if there is no parent, create a new root node, and have it point to both the existing node and the newly created node.

Similarly, if we are trying to remove an existing entry, we proceed as follows:

1. If the root node is a leaf node, just remove the entry, otherwise
2. Remove the entry from the node, and if the node is less than half full, proceed to either
    a. redistribute the entries between it and an adjacent node, or
    b. if an adjacent node is just at half empty, merge the two blocks and remove an entry from the parent. Then
        i. if the parent was the root node and it only had two pointers, remove the root node and this node becomes the new root node, otherwise,
        ii. remove an entry the corresponding entry for the removed node from the parent and go to Step 2.

The maximum number of entries in a B+-tree of height h is $M^h \times L$, while the minimum number of entries is $2\left\lceil \frac{M}{2} \right\rceil^{h-1} \times \left\lceil \frac{L}{2} \right\rceil$; however, in both cases, the height is logarithmic in the number of entries. For any practical applications as we have just described, the maximum difference in height will be 1.

17.4.2.4 Summary of a sample directory structure

In this topic we looked at how directories can be maintained. The most common is to use a B+-tree tailored to the block size on the hard drive. To minimize the height of the B+-tree, all information is kept at the leaf nodes, and the internal nodes are only pointers to the appropriate leaf node.

17.4.3 Fragmentation

Fragmentation is where blocks associated with the same file are widely scattered throughout the drive. Consequently, for a HDD, this can result in slower access times, as the head must travel significantly further for each block. Of course, this is not a problem for flash memory. HDDs would, occasionally require *defragmentation*, so the above blocks could be reorganized as:

```
README.TXT    0    1258
SETUP.EXE     2    6835
REMOVE.EXE    9    2783
UNUSED        d    -
```

Thus, our table would look like:

0 1 2 3 4 5 6 7 8 9 a b c d e f

If a HDD is associated with a real-time system, and access to the drive can affect whether deadlines are met or not, this may require periodic defragmentation.

17.4.4 Fault tolerance and journaling

One significant problem with a file system is that the structure may be left in an inconsistent state if there is an improper reset or shutdown. For example, a file may be only partially copied or only partially saved, or information may be only partially updated in the associated file structures. In such a situation, it is necessary to check each file data structure and it may require additional information to correct any errors.

To deal with this more efficiently, many modern file systems provide fault tolerance using a concept known as *journaling*. This uses a circular buffer in secondary memory where it stores instructions that it will execute (that is, what changes are being made to which data structures) to perform the requested operation. Once this has been *recorded*, the set of instructions are flagged as having been correctly written to the journal and the file system begins to execute those instructions. Once the instructions have been successfully executed, the task is flagged as being *completed*. Now, if a reset or shutdown occurs prior to the executing being finished, when the file system is started up again, it goes through the journal and examines any transactions that were not successfully completed:

1. if the journal entry is marked as completed, there is nothing to do;
2. if the journal entry is marked as recorded but not completed, the instructions are re-executed so as to complete the requested operation; and
3. if the journal entry has not marked as recorded, the requested operation is discarded.

For example, suppose that there is a request to delete a file. This requires three operations:

1. remove the file from the corresponding directory,
2. mark the space for the file as available, and
3. mark the space for the inode as free.

First, these three operations would be recorded in the journal and once they are written there, they would be flagged as recorded. If only two operations were recorded and a reset or shutdown occurred, the instructions would not be flagged as having been recorded, in which case, the instructions are ignored.

Next, the file system first removes the file from the directory structure, then it walks through the inode and flags the blocks of the file as free, and finally it walks through the inode flagging those blocks of the inode as free. If a reset or shutdown occurred at this point, these instructions would be re-executed. Each would be checked to determine whether or not the instruction had completed, and incomplete instructions would be re-executed.

There are two types of journals:

1. those that store only changes to the meta-data or the data-structures used by the file system to record the existence and location of the file, and
2. those that store every block that is to be written to secondary memory.

In the latter case, this can require a significant amount of overhead, but is necessary if absolute fault tolerance is required.

17.4.5 Memory and file systems: Spirit Mars rover

You may recall that one issue with the Spirit rover was that the size of the file manager was too large for the memory available. Consequently, when the file manager asked for more memory, an error was generated and this error resulted in the system being reset. When the reset loaded the file manager, it asked for too much memory, so they system was—

again—reset, in perpetuity, or at least, until they managed to put Spirit to sleep and determine what happened. Fortunately, Opportunity landed on the same day that they managed to put Spirit to sleep, and they were able to resolve the issue with Opportunity prior to it causing the same infinite reset cycle seen in Spirit.

17.4.6 Summary of file systems

A file system is a collection of data structures that describe both directories and the files stored on a drive (be that drive physical or virtual). The system is usually stored on the drive containing the files, and therefore the data structures used often take advantage of this to create efficient data structures. We have also discussed the problem of fragmentation in HDD, a problem not related to SSDs, as the latter has random access.

17.5 Data formats

Previously, we discussed the issue of storing nonlinear data structures such as tress as files. In the case of an AVL tree, because it stores linearly ordered data, we need only traverse the tree and store the data as a sequence of values. Other data structures, however, are not so trivial. First, even representing a string can be challenging:

> That kind of skeptical, questioning, "don't accept what authority tells you"…

would be represented in ASCII as the string:

```
"That kind of skeptical, questioning, \"don\'t accept what authority tells you\"..."
```
[44]

The quotes must be *escaped* using a backslash, and backslashes themselves are represented by a pair of backslashes:

```
"These characters must be escaped:  \\ \" \' and \?"
```

How can we represent mathematics or data structures? Donald Knuth spent a decade developing and perfecting TeX so that a string such as

```
The integral $\int_0^\infty sin\left(x^2\right) dx$ is $\frac{\sqrt{2\pi}}{4}$.
```

is displayed approximately as

$$\text{The integral } \int_0^\infty \sin\left(x^2\right)dx \text{ is } \frac{\sqrt{2\pi}}{4}.$$

requires even more interpretation of many symbols. We will look at three common means of storing data:

1. the eXtensible Mark-up Language (XML),
2. JavaScript Object Notation (JSON), and
3. the Java serializable interface.

The first is for storing general data, the first two are human readable, and the second two are specifically for storing instances of classes.

17.5.1 The extensible mark-up language (XML)

XML is a flat representation of a tree structure. Each node is represented by an opening tag `<identifier>`, followed by contents, and a closing tag `</identifier>`. Consequently, the angled brackets must be escaped, and this is done with entities: `&entity;` (such as `&lt;` and `&gt;` and `&amp;`) where these are used not only for angled brackets but also non-ASCII characters (`&ge;`).

[44] Carl Sagan, *Talk of the Nation*, 3 May 1996.

Like parentheses, brackets and braces in C, each closing tag must match the most recent unmatched opening tag. This means that XML is actually the coding of a tree structure:

The most obvious use of XML is XHTML, used for describing web pages.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
               "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<xhtml><head><title>A simple web page</title></head><body bgcolor="#FFFFFF">
<h3>17.6.1 The extensible mark-up language (XML)</h3>
<p>XML is a flat representation of a tree structure. Each node is represented by an opening tag
<tt>&lt;identifier&gt;</tt>, followed by contents, and then a closing tag
<tt>&lt;/identifier&gt;</tt>. Consequently, the angled brackets must be escaped, and this is done
with entities:  <tt>&amp;entity;</tt> (such as <tt>&amp;lt;</tt>...
</body></xhtml>
```

The significance of the tags described by an *XML Schema*, and a schema together with an XML document can be fed to a parser that produces a tree data structure that can then be accessed. There are parsers that have reasonably small footprints: the binary for Mini-XML is on the order of 35 KiB.

17.5.2 The JavaScript object notation (JSON)

While originally used in JavaScript, this format is designed for transmitting information about objects. It has become more popular than XML in many circumstances as it is more compact, easily human readable, and designed specifically for storing objects.

```
{
    "givenName": "Douglas",
    "surname": "Harder",
    "age": 42,
    "address": {
      "careOf": "ECE Department",
      "streetAddress": "200 University Ave. W",
      "apartment": null
      "city": "Waterloo",
      "province": "ON",
      "postalCode": "N2L3G1"
    },
    "phoneNumbers": [
        {
            "type": "home",
            "number": "5198884567"
            "extension": "37023"
        },
        {
            "type": "fax",
            "number": "5197463077"
        }
    ],
    "position": "Continuing Lecturer"
}
```

17.5.3 Java serializable

Java has a built-in mechanism for serialization, and any class that implements Serializable will be written as a character stream to an output file, and can be read back in. For a class to be serializable, all instance variables must also be serializable. Those instance variables that should not be saved should be flagged as `transient`, in which case, when the instance of the class is restored, this variable is set to its default. Here, the `Employee` class contains a number of strings, integers, an address (which is also serializable) and an array of serializable phone numbers. The instance variable `hasMail` has been flagged as being transient—there is no requirement to save it.

```
import java.io.*;
```

```
public class Address implements Serializable {
    private String        careOf;
    private String        streetAddress;
    private String        apartment;
    private String        city;
    private String        province;
    private String        postalCode;
    // ...
}

public class PhoneNumber implements Serializable {
    private String        type;
    private String        number;
    private String        extension;
    // ...
}

public class Employee implements Serializable {
    private String        givenName;
    private String        surname;
    private int           age;
    private Address       mailingAddress;
    private PhoneNumber[] phoneNumbers;
    private String        position;

    private transient boolean hasMail;

    // ...
}
```

We can thus save and restore an object as follows where the name of the file storing serializable objects is, by convention, given the extension `.ser`.

```
import java.io.*;

public class EmploymentManagementSystem {
    // ...

    protected void createAndSaveEmployee( ... ) {
        // Initialize the newly hired employee
        Employee new_hire = new Employee( ... );

        // ...

        // Save the newly hired employee
        try {
            FileOutputStream fileOut = new FileOutputStream( "filename.ser" ); {
                ObjectOutputStream objOut = new ObjectOutputStream( fileOut ); {
                    objOut.writeObject( new_hire );
                } objOut.close();
            } fileOut.close();
        } catch ( IOException ioExcept ) {
             ioExcept.printStackTrace();
        }

        // ...
    }

    protected Employee restoreCurrentEmployee( ... ) {
        // ...

        Employee current_employee = null;
```

```
        try {
            FileInputStream fileIn = new FileInputStream( "filename.ser" ); {
                ObjectInputStream objIn = new ObjectInputStream( fileIn ); {
                    current_employee = (Employee) objIn.readObject();
                } objIn.close();
            } fileIn.close();
        } catch ( IOException ioExcept ) {
            ioExcept.printStackTrace();
            return null;
        } catch ( ClassNotFoundException cNFExcept ) {
            cNFExcept.printStackTrace();
            return null;
        }

        // ...

        return current_employee;
    }
}
```

17.5.4 Summary of data formats

We have briefly reviewed three file formats for storing organized data: XML is general and human readable, while JSON is specific to storing instances of classes, and the Java serializable interface allows instances of classes to be stored in a non-human readable (and more compact) file.

17.6 The file abstraction

The Keil RTOS supports the C standard input/output library (`stdio.h`). Files are treated as an ordered sequence of characters (bytes). Thus, it is useful to look at interface of the file abstraction.

17.6.1 open

To open a file, use

```
int open( char *filename, int flags )
```

Here, the filename is a null-terminated character array, and the value returned will be a unique integer that you will use later to refer to this file. This integer is referred to as the *filehandle*. There are numerous flags, but at the very least, one of `O_RDONLY`, `O_WRONLY`, or `O_RDWR` is required to indicate whether the file is being opened for reading only, writing only, or reading-and-writing, respectively.

17.6.2 close

To close a file, use

```
int close( int filehandle )
```

At this point, the operating system may choose to reuse the filehandle for a subsequent call to open.

17.6.3 read

To read characters from the file

```
ssize_t read( int filehandle, char *p_buffer, size_t length )
```

This will copy up to `len` characters from the current position in the file and copy them to the character array `buffer`. The position in the file is updated to the next character. The returned value will be the number of characters copied and a negative number if there was an error. (Here, `ssize_t` indicates the type is a signed `size_t`.) The returned value may be less than `len` if, for example, you have reached the end of a file or, if you are accessing another device, there may not be `len` characters currently ready.

17.6.4 write

To write characters to the file

```
ssize_t write( int filehandle, char *p_buffer, size_t length )
```

Again, this function will copy up to `len` characters from character array `buffer` to the current position in the file. The position in the file is updated to the next character. The returned value will be the number of characters copied and a negative number if there was an error. The returned value may be less than `len` if, for example, you have reached the end of a file or, if you are writing to another device and it currently cannot accept more than `len` characters.

17.6.5 lseek

To change the current position within the file, use

```
off_t write( int filehandle, off_t offset, int flags )
```

This repositions the current position to an offset (positive or negative) relative to either

1. the current position (`SEEK_CUR`),
2. the start of the file (`SEEK_SET`), or
3. the end of the file (`SEEK_END`)

which is passed as the third argument.

17.6.6 Buffered I/O

The standard input/output library supports higher-level functions such as `fopen`, `fread`, `fwrite`, `fprintf`, `fscanf`, etc. These use buffering strategies to reduce the number of required reads and writes to the actual devices. If you want to force the buffers to be written, you can use `fflush`.

Incidentally, `printf( … )` is equivalent to `fprintf( stdout, … )`.

17.6.7 Summary

To summarize, Unix treats files as a stream of characters and you can move around in that stream. In general, Unix abstracts provides a file-like interface to essentially every device, providing an interface that treats it as if it were a stream of characters; so whether you want to access a drive, a CD player, or any other device (including mice, keyboards, audio, network sockets, etc.), the interface is the same: a stream of characters. Of course, in some cases, it may not be possible to move back in that stream (such as with mice).

17.7 Keil RTX RTOS

The Keil RTX RTOS allows you to use either FAT12, FAT16 or FAT32 for memory including

1. SD (Secure Digital) cards,
2. NAND flash, and
3. USB (Universal Serial Bus) drives.

It also has a proprietary embedded file system for

1. NOR flash,
2. SPI flash, and
3. RAM devices.

One issue with NAND flash is that it has limited write (program-erase or P/E) cycles (although with new flash drives allowing 100 million P/E cycles, this may no longer be an issue) and to maximize the lifespan of the flash, writes are redirected toward different physical blocks.

17.8 Summary

In this topic, we have very quickly looked at file systems. As a mechatronics engineer, you will likely use file systems already in place. Consequently, we have described the concept of the tree directory structure, block addressability, FAT as an example of a file system, the file abstraction, file I/O in the Keil RTOS, and a review of the issues with the Spirit Mars rover.

Problem set

17.1 What does it mean for a drive to be block addressable?

17.2 Suppose we have a block size of 4 KiB and each block has a 32 bit address. What is the maximum drive size? What is the maximum drive size if the addresses are 48 bites?

17.3 Give a rational for not using a block for multiple files. Specifically, suppose a block is shared by two files and two separate tasks want to lock the two files to write data to those files.

17.4 In Question 15.1, suppose that the maximum processor utilization for other tasks is $U = 0.83$. What is the maximum possible runtime of the ISR in order to ensure that the load factor does not exceed 1?

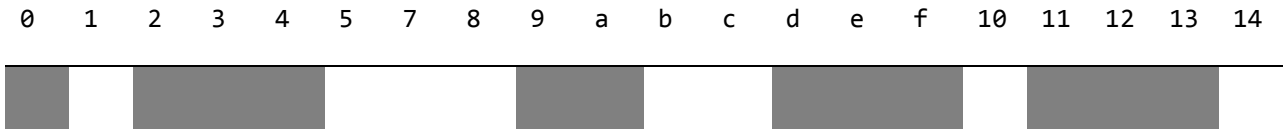
17.5 Explain why a B+ tree is designed to function with blocks. Asymptotically speaking, is there any difference between B+ trees and AVL trees?

17.6 Explain why Unix inodes are designed to function with blocks.

17.7 Practically speaking, is it fair to say that accessing any byte within a file stored using a Unix inode is $\Theta(1)$? Why or why not?

17.8 What is the run time to access the n^{th} byte of file stored using FAT?

17.9 Suppose you were required to map out the blocks for a 37 MiB file in a file system using FAT and the allocation table currently is as follows where the gray cells denote occupied blocks.



17.10 How would you store unallocated blocks using FAT?

17.11 Seek time on average hard disk drives (HDDs) is approximately 12 ms. Assuming that the time to transfer data from the drive to memory is insignificant in comparison to seek time once the head is in place, comment on the time it would take to read a fragmented file versus one contiguous on the drive platters.

17.12 A solid state drive (SSD) will begin transferring a block in memory after only 10 μs and a typical HDD has a data transfer rate of approximately 1 Gbit/s versus 1-4 Gbit/s for SSD. What are the drawbacks of SSDs?

17.13 Assuming that the root directory is one node in size, that any inodes is also only one block in size, and that only the block associated with the root directory is currently in main memory, how many blocks must be loaded into main memory to load the file `stdio.h`?

```
$ ls -al /
drwxr-xr-x 110 root root  4096 May 12  2011 usr
$ ls -al /usr/
drwxr-xr-x 110 root root 12288 Nov 24 07:18 include
$ ls -al /usr/include/stdio.h
-rw-r--r-- 1 root root 28341 Sep 16 02:05 /usr/include/stdio.h
```

18 Data management

We will now review data structures that are relevant to real-time systems, including those that store

1. linearly ordered data,
2. unordered data using hash tables, and
3. graphs.

This will be followed by a discussion of databases appropriate for real-time systems. With respect to the non-functional requirements for real-time systems, the appropriate use of data structures makes it possible to ensure acceptable performance and therefore enhance safety.

18.1 Linear data structures

Stacks designed with one-ended arrays, and queues and deques designed with circular arrays provide Θ(1) access to the front or top, Θ(1) pop and provided that sufficient memory has been allocated for the array, Θ(1) push. Array-based stacks, queues and deques

18.1.1 Array-based stacks, queues and deques

An array-based stack, queue or deque can only be real-time if either

1. sufficient memory is allocated apriori for the maximum size of the data structure, or
2. there is an appropriate mechanism to deal with full data structures.

A stack would be implemented simple one-ended array, while a queue or a deque could be implemented using a circular array.

18.1.2 Node-based stacks, queues and deques

In a node-based stack, queue or deque, each entry is stored in a data structure that also contains a reference (an address or offset) to the next node. Such data structures are only appropriate for real-time systems if:

1. the nodes are created with the appropriate reference pointer already in place, or
2. there is a readily available pool of unused nodes that can be allocated efficiently.

We have already seen how a task control block (TCB) stores one or more pointers used for various purposes such as placing the task on either the ready queue or on a queue waiting for a particular semaphore or other resource.

```
typedef struct single_node {
    void               *p_entry;
    struct single_node *p_next;
} single_node_t;

single_node_t  a_node_pool[NODE_POOL_CAPACITY];
single_node_t *p_node_pool_next;
```

```
void node_pool_init() {
    int i;
    node_t *p_node_itr   = a_nood_pool;
    p_node_pool_next     = a_nood_pool;

    for ( i = 0; i < NODE_POOL_CAPACITY - 1; ++i ) {
        p_node_itr->p_next = node + 1; // Pointer arithmetic, the address of the next node
        ++p_node_itr;
    }

    p_node_itr->next = NULL;
}

node_t *node_pool_alloc() {
    node_t *p_allocated_node;

    if ( p_node_pool_next == NULL ) {
        p_allocated_node = NULL;
    } else {
        p_allocated_node = p_node_pool_next;
        p_node_pool_next = p_node_pool_next->p_next;
    }

    return p_allocated_node;
}

void node_pool_free( node_t *p_freed_node ) {
    p_freed_node->p_next = p_node_pool_next;
    p_node_pool_next = p_freed_node;
```

18.1.3 Sorted list data structures

Storing sorted data where entries can either be inserted or removed requires a sorted tree structure in order to possibly run in o(n) time. The use of B+-trees, as described in Section 17.4.2 is appropriate if there is a large amount of data or data that is meant to be persistent and therefore will be stored in secondary memory. If the sorted data is meant to be stored in main memory only, either AVL or red-black trees are appropriate. AVL trees are, on average shallower than red-black trees, but require more effort for insertions and erases, and therefore

1. AVL trees are more appropriate if query operations are more common, while
2. red-black trees are more appropriate if insertions and erases are more common.

In either case, the necessary pointers and additional parameters can already be integrated into a larger data structure or, as suggested in the previous section, a pool of such nodes could be prepared at initialization time. Such nodes would have the format

```
typedef struct tree_node {
    void *p_entry;
    struct tree_node *p_left;
    struct tree_node *p_right;
    uint8_t height; // For an AVL tree
                          //  - if the height is greater than 255, this is not real
    // bool is_black;     // For a red-black tree
} tree_node_t;
```

We have already discussed the implementation of priority queues in Section 9.5.2.2. In general, the ideal choices are

1. binary heaps assuming the memory can be allocated at once, or
2. leftist heaps if a node-based structure is available.

For a leftist heap, the parameter `height` would be changed to `null_path_length`.

18.2 Hash tables

A hash table is a data structure that has an average run time of accessing, inserting or removing is $\Theta(1)$ but which may have a worst-case run time of $\Theta(n)$. Consequently, most real-time developers will prefer balanced search trees, so while the average case behavior is worse, $\Theta(\ln(n))$, the worst-case behavior is at most a constant multiple of the average-case behavior (a factor of two for red-black trees and a factor of $\frac{1}{\lg\left(1+\sqrt{5}\right)-1} \approx 1.44042$ for AVL trees).

We will discuss hash tables that are, under certain circumstances, appropriate for real-time systems, including:

1. quadratic probing, and
2. cuckoo hashing.

The reader may also wish to investigate hopscotch hasing. Hash tables should only ever be used internally. If there is any opportunity for an external source to feed the keys for the hash table, this could be used to corrupt the system by deliberately feeding keys that produce the worst-case scenario.

18.2.1 Quadratic probing

Given a hash table of size $M = 2^m$, an object x is placed into the bin corresponding with the hash value $h(x)$. If that bin is occupied, the algorithm begins searching forward according to

```
unsigned int i, b = h(x);

for ( i = 1; i <= M; ++i ) {
    // Examine bin 'b'
    //  - if appropriate, break
    b = (b + i) & (M - 1);
}
```

This algorithm is referred to as *quadratic probing* because the bins are visited in the order $\left(h(x)+\frac{i(i+1)}{2}\right) \bmod M$ for $i = 0, \ldots, M - 1$. Such an algorithm is reasonably efficient if the load factor λ—the ratio of occupied bins over the total number of bins—is less than 0.5. The average number of bins searched to find an object already stored in the table is $\frac{1}{\lambda}\ln\left(\frac{1}{1-\lambda}\right)$ and the average number of bins searched to find an empty bin is $\frac{1}{1-\lambda}$. Consequently, if $\lambda \leq 0.5$, then the number of searches required, on average, is 1.39 and 2, respectively—significantly faster than any linearly ordered data structure described previously.

Unfortunately, the worst-case number of searches required is $O(n)$, a performance significantly worse than either AVL or red-black trees. Consequently, such an approach cannot be used unless the data being stored in the hash table has already been tested beforehand. Such an approach cannot be used if the data originates from a point external to the real-time system, as a malicious agent could intentionally generate input that would produce the worst-case scenario. If fast access is required where the data is inserted at non-critical times and where access must be $\Theta(1)$, cuckoo hashing may be appropriate; however, if all operations must be efficient ($O(\ln(n))$, then balanced trees are likely the only feasible solution.

For more details regarding this algorithm, see Cormen et al. analysis, see Donald Knuth's book on seminumerical algorithms. There is also a *Robin Hood* modification of quadratic probing which the reader is invited to look up.

18.2.2 Cuckoo hashing

A hash table using cuckoo hashing has guaranteed $\Theta(1)$ access time, but inserting new entries may require $\Theta(n)$ time. The algorithm is named after the European cuckoo bird that lays its eggs in other bird's nests and where the cuckoo chicks *kick out* the chicks of the resident. The cowbird, shown in Figure 18-1, is an equivalent species in North America.

Figure 18-1. Two cowbird chicks that have been fed by the mother sparrow, while the smaller sparrow chick goes hungry.

The idea is quite simple:

1. Create two arrays $\mathbf{a}_1$ and $\mathbf{a}_2$, each of size $M = 2^m$.
2. Define two has functions h_1 and h_2.

In inserting a new object, call it x, into the hash table,

1. if the entry $a_1[h_1(x)]$ is unoccupied, fill it with x;
2. otherwise remove what is currently in that location, y, and replace it with x, and
    a. if the entry $a_2[h_2(x)]$ is unoccupied, fill it with y;
    b. otherwise, remove what is currently in that location, call it x, replace it with y, and go back to Step 1.

At this point, one of two things will happen: either it will be possible to place all entries, or we will go into an infinite loop. In the latter case, there are two possibilities:

1. choose two new hash functions and try again, or
2. double the capacity of the two arrays and tray again with two new hash functions.

In either case, the run time is $O(n)$; however, now accessing and erasing entries from the hash table may both be done in $\Theta(1)$ time:

1. to access an entry x, it is only necessary to check the locations $a_1[h_1(x)]$ and $a_2[h_2(x)]$, and
2. to erase an entry, it is only necessary to mark that entry as unoccupied.

The following are implementations of the insert, access and erase functions:

```
void insert( type key, type value ) {
    unsigned int h, parity;
    parity = 0;

    while ( true ) {

        unsigned int h = hash[parity]( key );

        if ( !array[parity][h].occupied ) {
            array[parity][h].occupied = true;
            array[parity][h].key = key;
            array[parity][h].value = value;
            break;
        }

        // Detect if in infinite loop, and if so, resize arrays and try again

        swap( &key,   &array[parity][h].key   );
        swap( &value, &array[parity][h].value );
        parity = 1 - parity;
    }
}

bool access( type key, type *p_value ) {
    bool success = false;
    unsigned int h, parity;

    for ( parity = 0; parity < 2; ++ parity ) {
        h = hash[parity]( key );

        if ( array[parity][h].occupied && array[parity][h].key == key ) {
            *p_value = array[parity][h].value;
            success = true;
            break;
        }
    }

    return success;
}

bool erase( type key ) {
    bool success = false;
    unsigned int h, parity;

    for (parity = 0; parity < 2; ++ parity ) {
        h = hash[parity]( key );

        if ( array[parity][h].key == key ) {
            array[parity][h].occupied = false;
            success = true;
            break;
        }
    }

    return success;
}
```

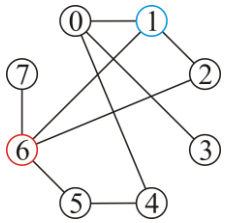
18.3 Graphs

The efficient implementation of graphs can be critical for real-time systems that must traverse graphs for finding, for example, optimal paths. If the maximum number of neighbors of any vertex is $\Theta(1)$ with respect to the number of nodes, say four, it would be reasonable to represent the graph using an adjacency list.

A graph that is fixed in size and with respect to the number of connections can always be programmed to have a minimal foot print. For example, a graph with *m* being 256 or fewer verticies with *n* vertices could be stored as:

1. a $\lceil \lg(m) \rceil$-bit unsigned integer array of size *m* storing the number of neighbors of each vertex,
2. a $\lceil \lg(m) \rceil$-bit unsigned integer array of size 2*n* indices storing the neighbors, and
3. a $\lceil \lg(n) \rceil$-bit unsigned integer array of size *m* storing the index in the previous array of the first neighbor of a vertex.

where the lg function is the base-2 logarithm. For example, consider the following graph with eight vertices and nine edges:



This graph can be stored compactly as the three arrays

```
uint8_t num_neighbors[ 8] = {3, 3, 2, 1, 2, 2, 4, 1};
uint8_t neighbor_list[18] = {1, 3, 4, 0, 2, 6, 1, 6, 0, 0, 5, 4, 6, 1, 2, 5, 7, 6};
uint8_t neighbor_index[8] = {0, 3, 6, 8, 9, 11, 13, 17};
```

requiring 34 bytes. At the expense of run-time, this could be reduced to a footprint of only

$$8 \times 3 + 2 \times 9 \times 3 + 8 \times 5 = 118 \text{ bits}$$

or 15 bytes, although this would require significant bit manipulations.

A graph with a fixed number of vertices, but possibly changing edges where there is a reasonable upper bound *N* on the number of edges could be implemented as an an adjacency list with a node pool of size *N*. As a new edge is included, a node could be taken from the pool, and as edges are removed, the nodes would be returned to the pool. This would require:

1. a $\lceil \lg(N+1) \rceil$-bit unsigned integer array of size *m* indices storing the index of the first node, and
2. an array of structures of size *N* representing the node pool, where each structure stores
    a. a $\lceil \lg(m) \rceil$-bit unsigned integer storing the index of the next neighbor, and
    b. a $\lceil \lg(N+1) \rceil$-bit unsigned integer storing the next node in the linked list.

The *N* + 1 is necessary to ensure that there is one index used to represent `NULL`. Consequently, an graph with 256 verticies and where each vertex has a maximum of six neighbors could be stored using

$$256 \times 11 + 1536 \times (8 + 11) = 32000 \text{ bits}$$

or 4000 bytes, although as before this would require a significant amount of bit manipulation. Reducing the need for bit manipulations would see a requirement for 256 × 2 + 1536 × (1 + 2) = 5120 bytes, an almost insignificant amount of additional memory.

If the above graph with eight vertices did not have an upper limit on the number edges, meaning it could possibly be used to store a complete graph, then the memory required would be 8 × 7 + (8 × 7) × (3 + 7) = 616 bits or 77 bytes, although if we restricted ourselves to using `uint8_t`, this would require 8 × 1 + (8 × 7) × (1 + 1) = 120 bytes.

18.4 Non-relational databases

A simple database is one that persistently stores key-value pairs. As an example, the BerkeleyDB is a database that allows the user to store, access and delete key-value pairs in a database. Features include:

1. Locking mechanisms to allow multiple readers or single writers.
2. Transactions and logging are used to ensure that the database is recoverable if there is a failure during an operation. Once the system is reset, the interrupted operation may either be redone (and completed) or undone (leaving the database in the state it was prior to the start of the interrupted operation).
3. Caching of pages within the database to allow faster access but also writing modified pages to the file system.
4. Encryption algorithms to protect the database from being read externally.

The underlying data structure of the database may be selected by the user, including

1. B-trees,
2. hash tables, or
3. queues.

The structure for databases is `DB` and a stand-alone database is created as follows:

```
DB *p_db;
int return_value = db_create( &p_db, NULL, 0 );
```

The pointer `p_db` is now assigned the address of the database structure and this structure has a number of fields that are assigned function pointers that can now be used. Five of these are described here, with the second argument set to `NULL`, as will be the case with many stand-alone uses.

Operation	Signature
Open	`int p_db->open( DB *p_db, NULL, const char *file_name, NULL, DBTYPE type, u_int32_t flags, int mode )`
Close	`int p_db->close( DB *p_db, u_int32_t flags )`
Put	`int p_db->put( DB *p_db, NULL, DBT *p_key, DBT *p_value, u_int32_t flags )`
Get	`int p_db->get( DB *p_db, NULL, DBT *p_key, DBT *p_value, u_int32_t flags )`
Delete	`int p_db->del( DB *p_db, NULL, DBT *p_key, u_int32_t flags )`

They keys and values are passed through a `DBT` structure:

```
typedef struct {
    void *p_data;
    u_int32_t size;
    u_int32_t ulen;
    u_int32_t dlen;
    u_int32_t doff;
    u_int32_t flags;
} DBT;
```

The two relevant fields are the first two (highlighted in red). This database simply stores a byte array of `size` bytes. Thus, the first field is assigned the addresses of the object to be stored in the database, and the second field stores the size; the database is not aware of what is being stored. The following is example of how this database is used, this is

modified from a tutorial created by Tobias Oetiker[45]. In this example the database is used to associate an integer with a coordinate structure.

```
#include <sys/types.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "db.h"

typedef struct {
    double latitude;
    double longitude;
    bool occupied;
} coordinate_t;

#define DATABASE "access.db"

int main() {
    DB *p_db;
    DBT db_key, db_data;
    int return_value, close_return_value;

    u_int32_t identifier = 42;
    coordinate_t location;
    location.latitude = 43.47;
    location.longitude = 80.54;
    location.occupied = true;

    // Create the database
    return_value = db_create( &p_db, NULL, 0 );

    if ( return_value != 0 ) {
        fprintf( stderr, "error in db_create: %s\n", db_strerror( return_value ) );
        exit( 1 );
    }

    // Open the database using the underlying data structure:
    //    DB_TREE, DB_HASH and DB_QUEUE
    return_value = p_db->open( p_db, NULL, DATABASE, NULL, DB_BTREE, DB_CREATE, 0664 );

    if ( return_value != 0 ) {
        p_db->err( p_db, return_value, "%s", DATABASE );
        goto close_db;
    }

    // Initialize the data structures for the key-values pairs
    memset( &db_key,   0, sizeof( db_key )   );
    memset( &db_value, 0, sizeof( db_value ) );

    db_key.data = &identifier;
    db_key.size = sizeof( u_int32_t );

    db_value.data = &location;
    db_value.size = sizeof( coordinate_t );

    // Store the key-value pair in the database
    //  - this will store a shallow copy of the key and value
    return_value = p_db->put( p_db, NULL, &db_key, &db_value, 0 );
```

[45] See http://sepp.oetiker.ch/db-4.2.52-mo/ref/simple_tut/intro.html

```
    if ( return_value != 0 ) {
        p_db->err( p_db, ret, "error message" );
        goto close_db;
    }

    // Retrieve the value associated with the given key from the database
    return_value = p_db->get( p_db, NULL, &db_key, &db_value, 0 );

    if ( return_value == 0 ) {
        coordinate_t *p_loc = (coordinate_t *) db_value.data;

        printf( "The identifier '%d' is %s located at (%f, %f)\n",
                *((int *) db_key.data),
                (p_loc->occupied ? "" : "not "),
                 p_loc->latitude,
                 p_loc->longitude );
    } else {
        p_db->err( p_db, ret, "error message" );
        goto close_db;
    }

    // Delete the value associated with the given key from the database
    return_value = p_db->del( p_db, NULL, &db_key, 0 );

    if ( return_value != 0 ) {
        p_db->err( p_db, return_value, "error message" );
        goto close_db;
    }

    // Try to retrieve the value associated with the key we have just deleted
    return_value = p_db->get( p_db, NULL, &db_key, &db_value, 0 );

    // This should fail
    if ( return_value != 0 ) {
        p_db->err( p_db, return_value, "error message" );
    }

    // Close the database
    close_db: {
        close_return_value = p_db->close( p_db, 0 );

        if ( close_return_value != 0 && ret == 0 ) {
            return_value = close_return_value;
        }
    }

    return 0;
}
```

18.5 Relational databases

In general, relational databases are likely not useful for real-time systems. The runtime of a query of a relational database can potentially be very large; however, inserting data can often be done in constant time. Consequently, a real-time system may store information in a relational database when it will then be queried either off-line or by tasks with priorities lower than those of any real-time tasks.

18.6 Summary of data management

This chapter has looked at data structures and data storage management schemes appropriate for real-time systems.

19 Virtual memory and caching

One rule-of-thumb in software engineering is that 10 % of source code is executed 90 % of the time. From this and other observations, we may deduce two principles. This is often relevant in optimizing code: if you speed up the 10 % of the code that is executing 90 % of the time by a factor of two, this will reduce the run time by 45 %; however, speeding up the other 90 % of the code by a factor of two will only see a 5 % decrease in the overall run time. Fortunately, tools such as profilers can be used to determine which statements and which functions execute most in any system. There are two other principles that have been observed in computer science:

1. *The principle of temporal locality*, which states that a memory location that has been recently accessed is likely to be accessed in the near future; and
2. *The principle of spatial locality*, which states that a memory location close to one that has been recently accessed is likely to be accessed in the near future.

Using these principles, computer scientists and engineers were able to speed up the execution of code on processors by introducing caches and virtual memory. The first is done entirely in hardware, while the second has software support. Caches were so successful that today, many microprocessors have numerous levels, each one faster than the previous. We will discuss the use of hard-disk drives and solids-state drives as virtual memory, discuss the issues with virtual memory and real-time systems, and describe the phenomenon of thrashing.

19.1 Caches and virtual memory

Two issues with main memory today are

1. transfer rates from main memory to registers are slower than the clock speeds of processors (although this is being reduced), and
2. there is insufficient main memory for many shared multi-user systems.

The first issue was more significant from the late 1990s up to around 2010 when processor speeds were increasing significantly faster than main memory transfer rates. Both of these can, however, be solved using the same solution.

1. First, we introduce faster memory, which we will call *caches*. The transfer rates of a cache are much higher than the transfer rates of main memory.
2. Second, we designate a portion of a hard-disk drive (HDD) to be *virtual memory*. Now, the memory used by a program (text, the heap and the stack) is stored on the hard-disk drive.

The memory stored on the HDD, main memory, and the cache are divided into 4 KiB *frames*. The actual memory required by an executing program is divided into *pages*. Each page is fit into one of the frames on the HDD. Recall that the compiler will only decide whether or not a value is in a register or whether it is stored in main memory at an address. If it requires a value that is currently located in main memory, it will load that value into a register. When it tries to load that value, it will follow the process:

1. if the page containing the address is in a cache frame, fetch that value, otherwise
2. the page is not currently stored in the frame so we issue a *cache miss* that signals the hardware to fetch the page in question from main memory; in which case
    a. if the page being fetched is in main memory, copy the page into the cache, otherwise
    b. the page is not currently stored in a frame of main memory, so we issue a page miss that signals the virtual-memory system to fetch the page in question from the hard-disk drive.

Thus, we may have a program where the text, heap and stack occupy a number of pages, but only a subset of those have recently been accessed, so only those are currently residing in main memory. Even more recently, only a few pages have had their values loaded to registers or had register values saved to them. This is demonstrated in Figure 19-1.

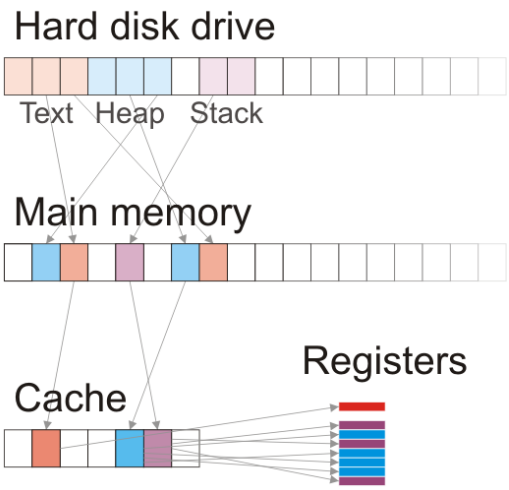


Figure 19-1. The function of virtual memory and caching in a computer system.

When a new page is loaded into either main memory or the cache into a frame, that frame may already contain a different page. We could write that page back out into either main memory or the hard drive, as appropriate; however, if nothing has changed in that page since it was loaded (for example, instructions from the text segment will only ever be loaded into the instruction register, and therefore such a page need never be written back into either main memory or the hard drive. For other pages, however, it is possible to associate a *dirty bit*; that is, a binary value that is initially zero, but if a change is ever made to that page, the bit is set to 1. When it comes time to replace the frame containing the page, then if the dirty bit is not set, it is only necessary to copy the new page into the frame, but if the bit is set, it is also necessary to copy the page back out.

Note that caching and virtual memory are only possible because of the principles of spacial and temporal locality. If every subsequent fetch occurs in a different location, both the caching and virtual-memory systems would continually be responding to cache misses and page faults, respectively.

19.2 Multiple levels of cache

Today you may have also heard of Level-2 and Level-3 caches. This is nothing more than more-and-more levels being placed into our hierarchy, where Level-1 caches are the fastest, but with the least capacity, Level-2 caches being slower than Level-1 caches, but with a greater capacity, and so on. There is even the introduction of *miss caches* and *victim caches*, caches where old pages are copied rather than being moved immediately back to a lower level.

19.3 Using solid-state drives as caches

One non-solution to reduce the transfer time between a hard-disk drive and main memory is to use faster memory such as solid-state drives (SSD or flash memory). Unfortunately, SSD has a limited number of write operations, after which it tends to fail. Consequently, it cannot be used for virtual memory.

As a demonstration of the failure of flash memory, in December of 2014, NASA had detected the apparent failure of the Opportunity rover to write telemetry information to one of seven flash memory banks. This failure led to the system entering a never-ending reset cycle (c.f. Spirit's initial issues described in Section 1.3.3.2) and thus interrupting communications around Christmas of that year. Attempts to correct this by restricting which flash memory banks were written to failed, hence on May 23, 2015, Opportunity was reprogrammed to only use volatile main memory.

For further information, read "Mars Rover Opportunity Suffers Worrying Bouts of 'Amnesia'", dated December 29, 2014, by Ian O'Neill at

http://news.discovery.com/space/mars-rover-opportunity-suffers-worrying-bouts-of-amnesia-141229.htm.

19.4 Virtual memory and real-time systems

Hard drives are slow: accessing a hard drive requires a seek time on the order of 10 ms. Consider a task that periodically restarts the watchdog timer. If that task is swapped out of main memory onto the hard drive, if there are only 9 ms left before the watchdog timer resets the system, there is a high probability that the system will be reset despite there being no issues. Thus, recall our discussion on the Linux `watchdog` daemon. This task runs every 10 s writing to the `/dev/watchdog`. If it does not write to this device for over a minute, the system will reset. If there is a high load, the daemon may be *swapped out*, in which case, it may not be loaded into main memory in time to signal. To solve this problem, it is possible to signal to the virtual-memory system to never swap this particular daemon into virtual memory, and this is done by setting a `realtime` variable in the `watchdog.conf` configuration file.

Alternatively, you can use the `mlock` and `mlockall` functions available in memory management library `mman.h`. The functionality is as follows:

```
#include <sys/mman.h>

int mlock  ( const void *p_address, size_t length );
int munlock( const void *p_address, size_t length );

int mlockall( int flags );
int munlockall();
```

`mlock` and `munlock` deal with specific memory that should not be swapped into virtual memory, while `mlockall` and `munlockall` locks into memory everything associated with the process, including shared memory and libraries, kernel data related to the process, and memory-mapped files.

Consequently, while caching will, on average, speed up the system and virtual memory will, on average, make more memory available for various tasks, together they make the analysis for determining whether or not deadlines will be met much more difficult.

19.5 Thrashing

One issue with virtual memory is that if the number of page faults becomes too large, the system will spend all of its time fetching pages from the hard-disk drive, a situation known as thrashing. Additionally, if each task is simultaneously using so many pages, it may be possible that most tasks cannot effectively continue executing without loading other pages, in which case every task switch results in a huge number of page faults.

In one situation, the NeXT Step OS was being loaded onto a NeXT computer with only 2 MiB of RAM instead of the required 4 MiB. In this case, the steady state of the operating system (where it was doing nothing) was to be in a constant state of swapping.

19.6 Page replacement algorithms

There are a number of algorithms available to decide which pages should be swapped out of the cache into main memory or out of main memory into virtual memory.[46] Additionally, is it ever useful to pre-fetch pages before there is an explicit request? There are some interesting observations: pages that have not changed need not be copied back out—this can save some time, as it is only necessary to copy a new page into main memory from the hard drive or copy a new page into a cache from main memory. It is even possible to describe desirable characteristics, including *Belady's optimal algorithm*—that is, swap that page out that will be accessed most distantly into the future. This is, of course, unachievable, but some algorithms are better than others at approximating this ideal. Other observations are that if you allocate a task more frames, in general, it is desirable that the number of page faults goes down. This is, unfortunately, not the case for the most trivial algorithms: just replace the frames in the order in which they appear in memory.

19.6.1 Two page-replacement algorithms

Two algorithms for deciding which page to replace are

1. first-in—first-out (FIFO), and
2. least-recently used (LRU).

We will quickly discuss each here.

19.6.1.1 First-in—first-out

The easiest algorithm to implement is first-in—first-out (FIFO). Under the assumption that the page loaded into a frame most distantly in the past is also least likely to be accessed (an approximation of temporal locality). The implementation is the simplest of all algorithms: for N frames, store an index k that cycles from 0 to $N - 1$ and replace the page in Frame k. Unfortunately, the assumption that simply because a page was loaded most distantly in the past does not mean it has not been referenced since, and if it has been referenced again recently, it is likely to be referenced again in the future.

19.6.1.2 Least-recently used

The least-recently used (LRU) algorithm indicates that the page to be replaced is that which has been accessed most distantly in the past. At first glance, you may consider a priority queue with time stamps; however, this is overkill for this purpose, and the run times are $\Theta(\ln(N))$. Instead, we note that there are only two operations necessary:

1. when a page in a frame is accessed, the frame is moved to the back of the list, and
2. when a frame is required, the page in the frame at the front of the list is replaced, and that frame is moved to the back of the list.

This requires nothing more than a cyclic doubly linked list (the cyclic component reduces some operations): a frame at an arbitrary location is moved to the back, or the frame at the front is moved to the back. With a cyclic doubly linked list, both these operations can be performed in $\Theta(1)$ time. For example, suppose we are reading the entries of a large array.

[46] This section is based on and expands on §12.4 of Gary Nutt's textbook *Operating Systems*.

As we scan the entries, each successive page is loaded into the next available frame, so we end up with a situation as we have in Figure 19-2. The head points to frame with the least-recently used page.

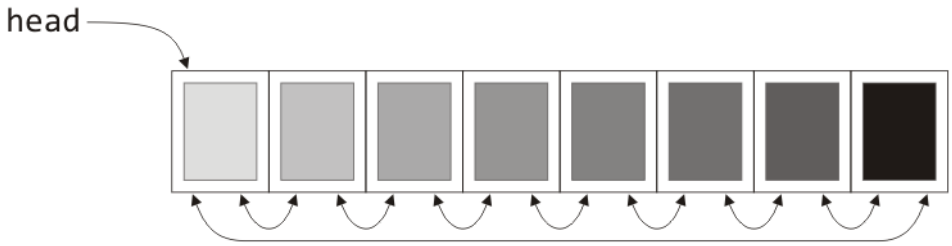


Figure 19-2. A cyclic doubly linked list of frames after a sequential access of pages.

Now, suppose that the next page accessed is already in Frame 4. Now, Frame 4 is moved to just prior to the head of the list, and the previous and next frames are linked to each other, as is shown in Figure 19-3. This would involve swapping three pairs of values in five frames.

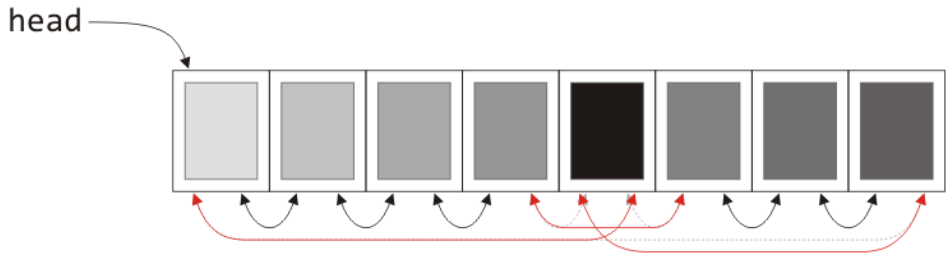


Figure 19-3. Frame 4 moved to the tail (or back) of the doubly linked list.

Now, given the state in Figure 19-3, a new page is accessed, so a cache miss results. Now, the frame at the head (or front) of the linked list is inspected. If the contents have been modified, the contents would be copied back to main memory; otherwise, we would just overwrite it. In either case, the new page is copied into that frame, and head pointer is advanced by one, as is shown in Figure 19-4.

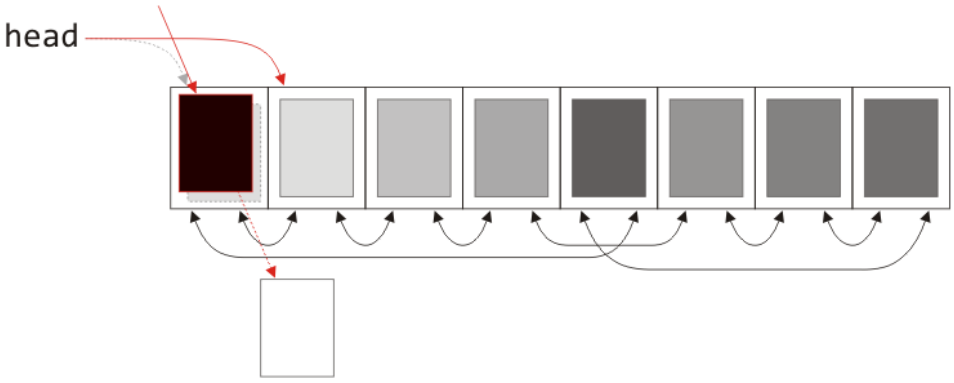


Figure 19-4. The state after the page in Frame 0 (at the head of the linked list) is replaced.

19.6.1.3 Summary of two page-replacement algorithms

We have quickly introduced two very straight-forward page replacement algorithms. We will now look at some of the issues with these based on observations made by László Bélády.

19.6.2 Bélády's optimal replacement algorithm

The first of two contributions made by László Bélády (Lăs-lō Bā-lă-dĭ) is his optimal replacement algorithm, which says to

> replace that page that will be accessed most distantly into the future.

It can be shown that if, at each step, the page that will be accessed most distantly into the future (or never again), is the page that is replaced, this must minimize the number of page replacements—no algorithm can do better. Unfortunately, to know a priori which page will be accessed most distantly into the future requires an *oracle* or *genie*, something that we have currently not yet developed; however, the performance of an implementable algorithm can be compared to Bélády's optimal algorithm, and if an algorithm has only 1 % more page replacements, such an algorithm may be considered to be reasonably efficient. Thus, when we develop page replacement algorithms, we want to develop them so that they most closely approximate this optimal algorithm.

FIFO assumes that if a page was loaded a long time ago, it is not likely to continue to be used; however, a little thought can suggest situations where this may not occur; for example, the instructions for accessing a data structure that is accessed numerous times will likely be accessed quite early on, and yet continue to be accessed periodically over time. Thus, once every N page replacements, these pages will be copied out only to be almost immediately copied back in.

LRU assumes that a page that has not been accessed for a long time has therefore gone into disuse. This is a more reasonable algorithm than FIFO, but it still has its weaknesses: consider looping over a very large array. For example, suppose an array was $N + 1$ pages in size but the cache or main memory has only frames available for N pages. In this case, after the first N page are loaded into main memory, each subsequent page accessed will lead to a page miss. There are modifications to both of these algorithms that try to improve or combine their characteristics; however, this is beyond the scope of this course.

19.6.3 Bélády's anomaly

In addition to proposing his optimal replacement algorithm, Bélády also noted an anomaly that

> not all page replacement algorithms will necessarily have fewer page misses if more frames are made available.

This may seem counter-intuitive—if more frames are available, there should be fewer page misses. Unfortunately, in his paper introducing this observation, Bélády demonstrates that FIFO suffers from this anomaly and he provided an example demonstrating this. More recently, Gary Nutt developed a theorem that demonstrated that LRU does not. suffer from this anomaly.

19.6.4 Summary of page-replacement algorithms

In this topic, we considered two algorithms that could be used for page replacement. Both LRU and FIFO have weaknesses, but the former is generally better in that it usually better approximates Bélády optimal algorithm and does not suffer from Bélády's anomaly.

19.7 Summary

This topic summarizes the concepts of caching and virtual memory. You need to be aware of what these technical solutions are and how they affect real-time systems.

Problem set

18.1 Why must all blocks associated with real-time tasks be locked in main memory in any system that uses virtual memory?

18.2 In a very heavily used system, argue that the clock algorithm reduces to FIFO.

18.3 Under reasonable page accesses, argue that another appropriate name for the clock method is *not recently used*.

20 Digital signal processing

The on-line version contains a chapter introducing digital signal processing, including many useful filters.

21 Digital control theory

The on-line version has a chapter introducing digital control theory.

22 Security and cryptography[47]

The on-line version has an appendix discussing the use of fixed-point algorithms for rate monotonic scheduling.

[47] Acknowledgment: most of the information in this section is taken from Wikipedia pages.

Appendices, references and index

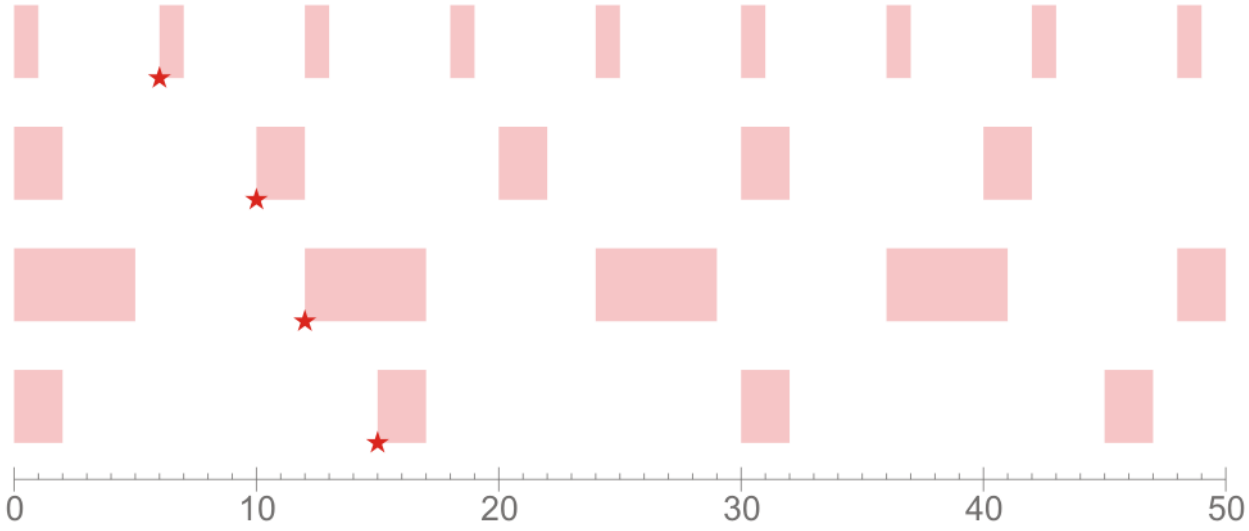
Appendix A Scheduling examples

We will look at techniques you can do to perform earliest-deadline first and rate monotonic scheduling by hand. This section is not included to make you proficient at such techniques, but rather, in understanding how to perform these optimally, it may give you insights into the algorithm. In both cases, we will schedule the tasks in this table:

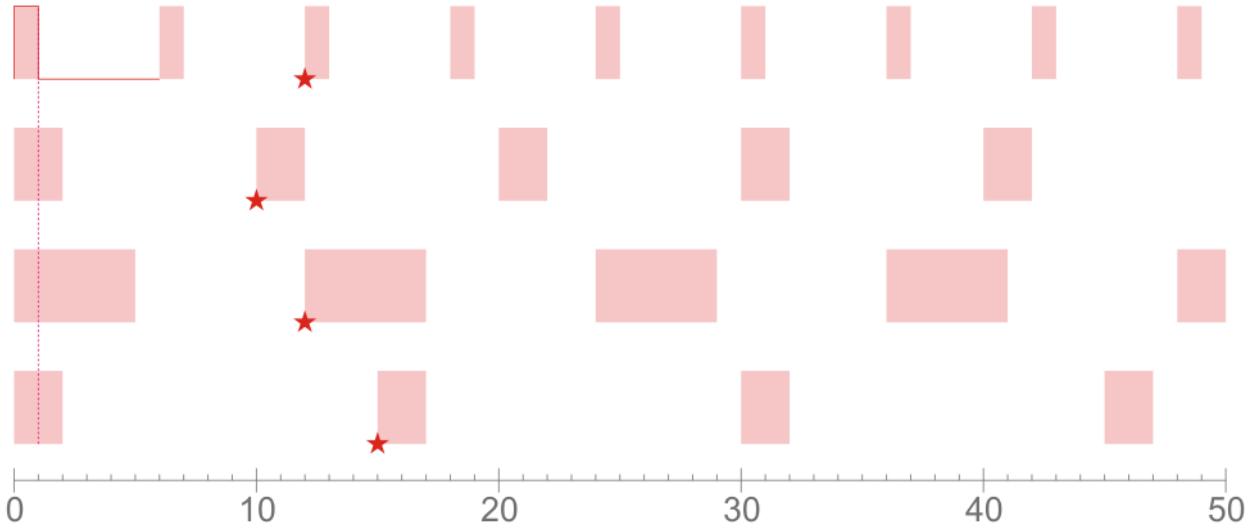
Task	c_k (ms)	τ_k (ms)
A	1	6
B	2	10
C	5	12
D	2	15

A.1 Earliest deadline first scheduling

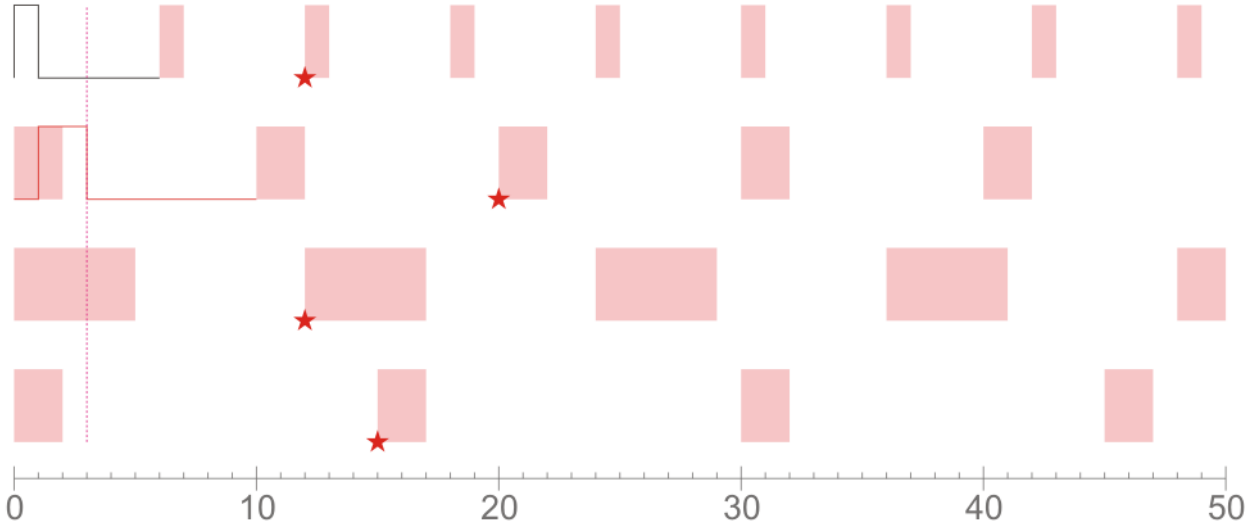
Calculating an earliest-deadline first schedule is actually much more complex than rate monotonic. For each task, indicate the duration and start time of each period.



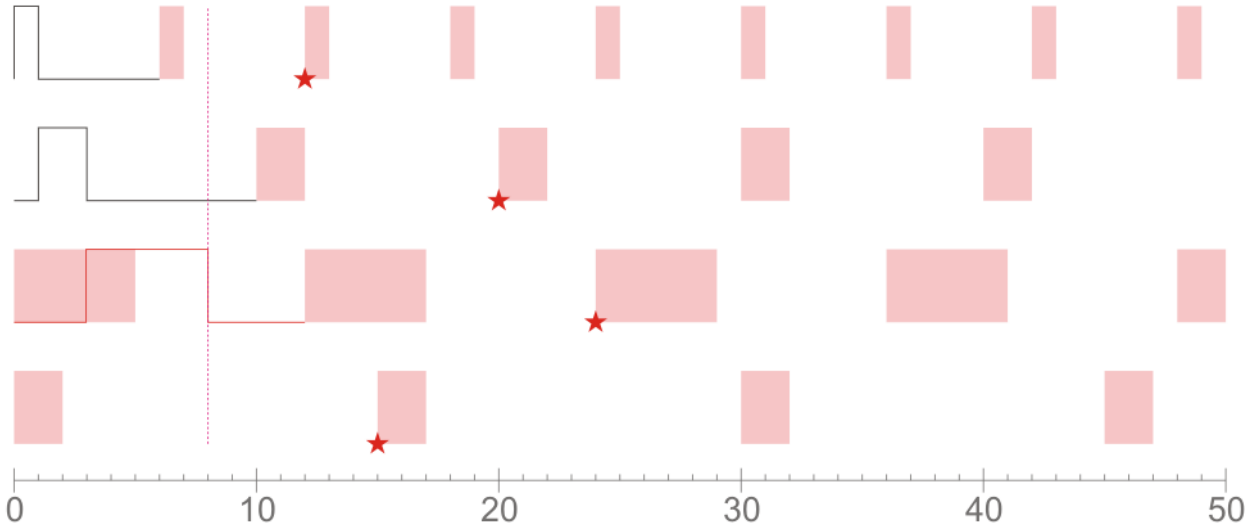
Of the deadlines of unscheduled tasks, the first task has the earliest deadline, so schedule it to run and update its deadline.



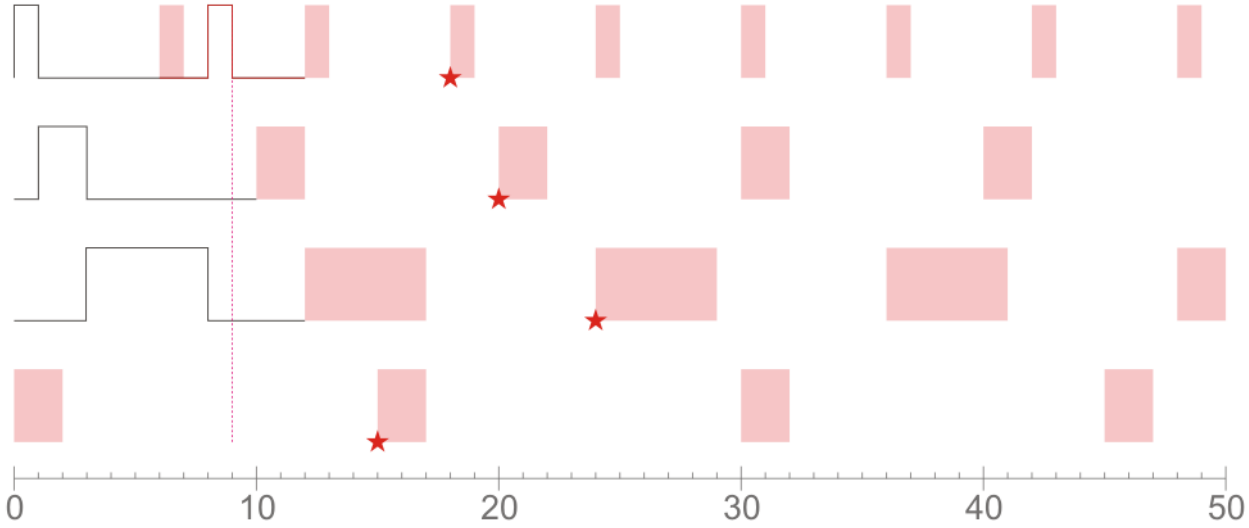
Next, Task B has the earliest deadline, so schedule it as soon as it can run.



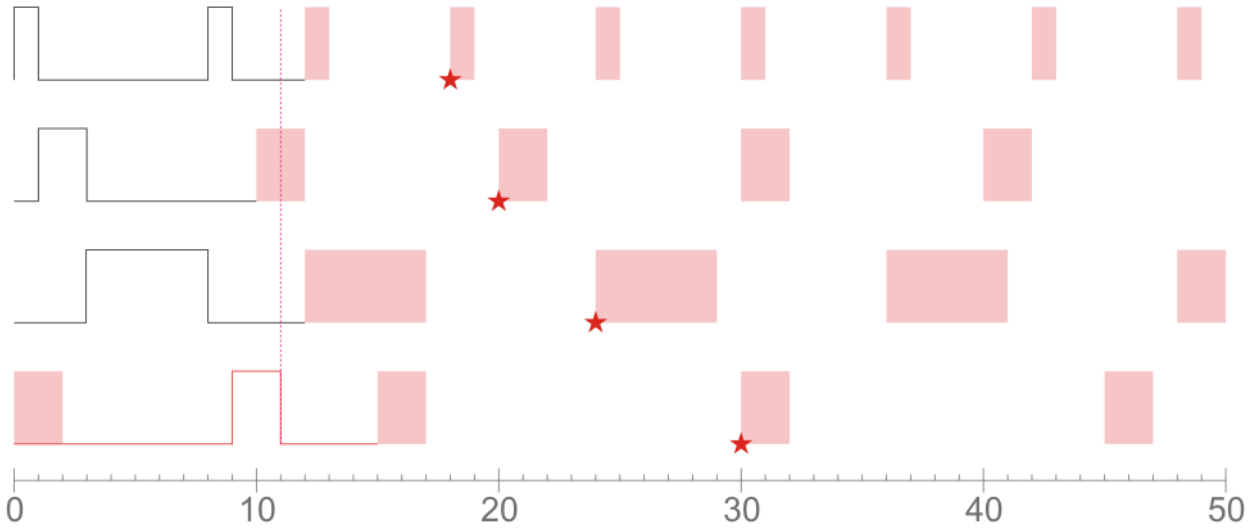
Next, there are two tasks we could execute: Tasks A and C, as both have the same deadline, but the next period of Task A does not start until time $t = 6$, so we can only schedule Task C.



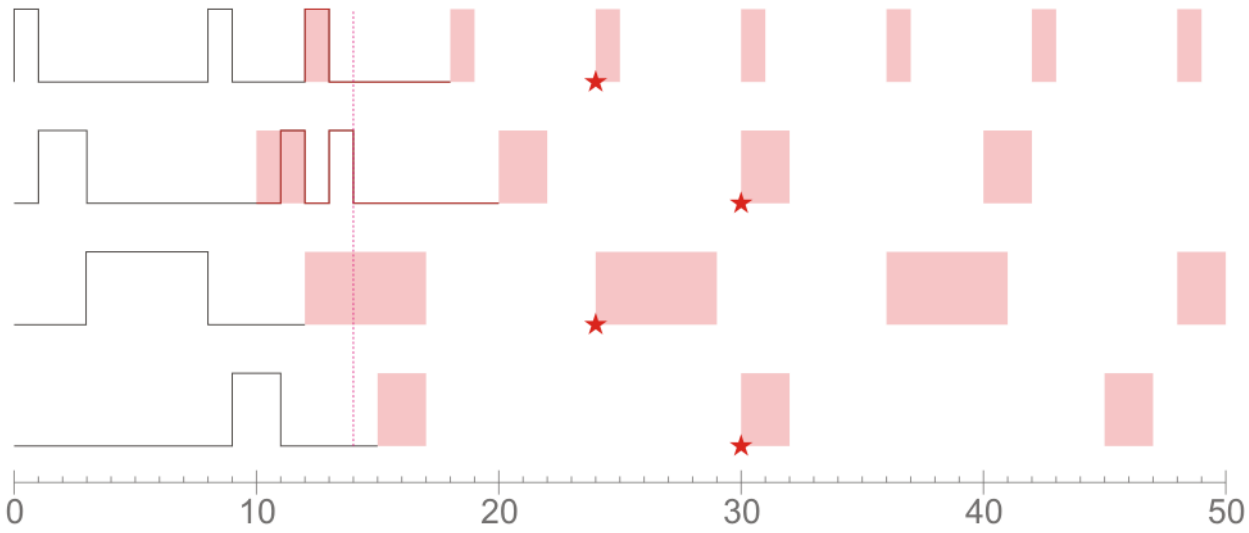
At this point, Task A is ready to execute, and as it has the earliest deadline, we schedule it.



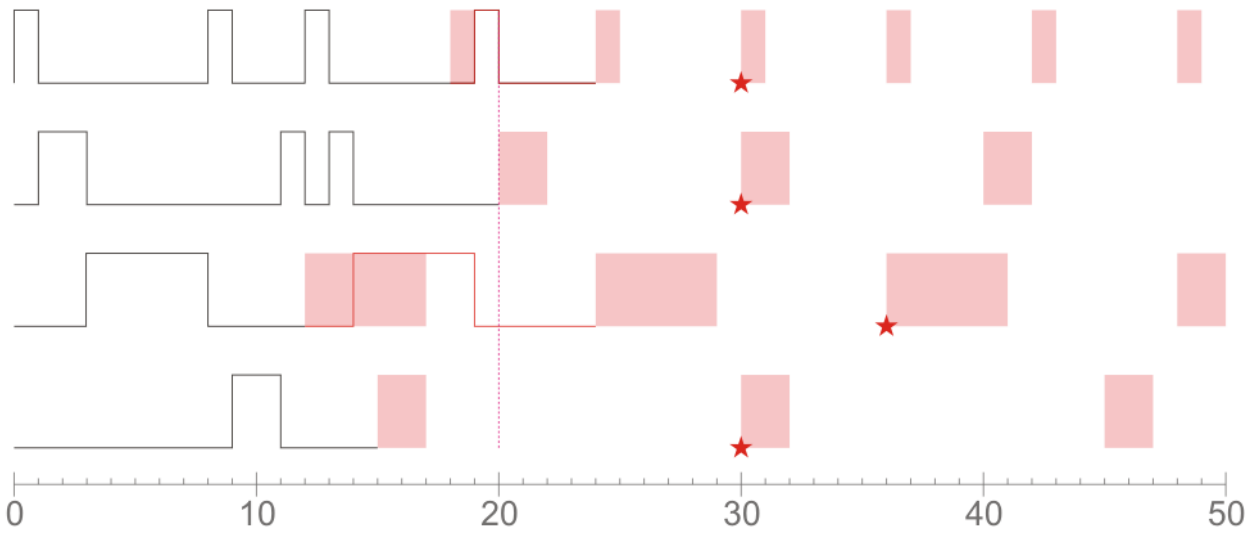
At this point, only Task D is ready to execute, so we schedule it.



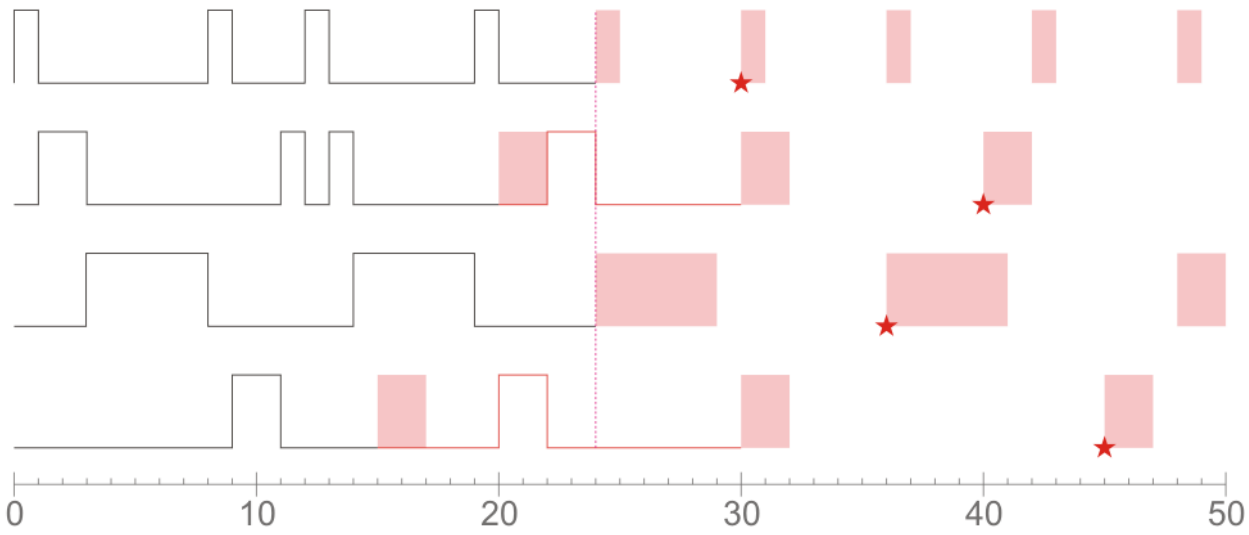
At time $t = 11$, there is only one task ready to execute: Task B. However, after it executes for one second, both Task A and C are also ready to execute. The issue is that the now-ready Task A has an earlier deadline than the executing Task B, so Task B gets pre-empted and Task A runs. Once Task A completes, between Tasks B and C, Task B has the earliest deadline, so we re-schedule Task B.



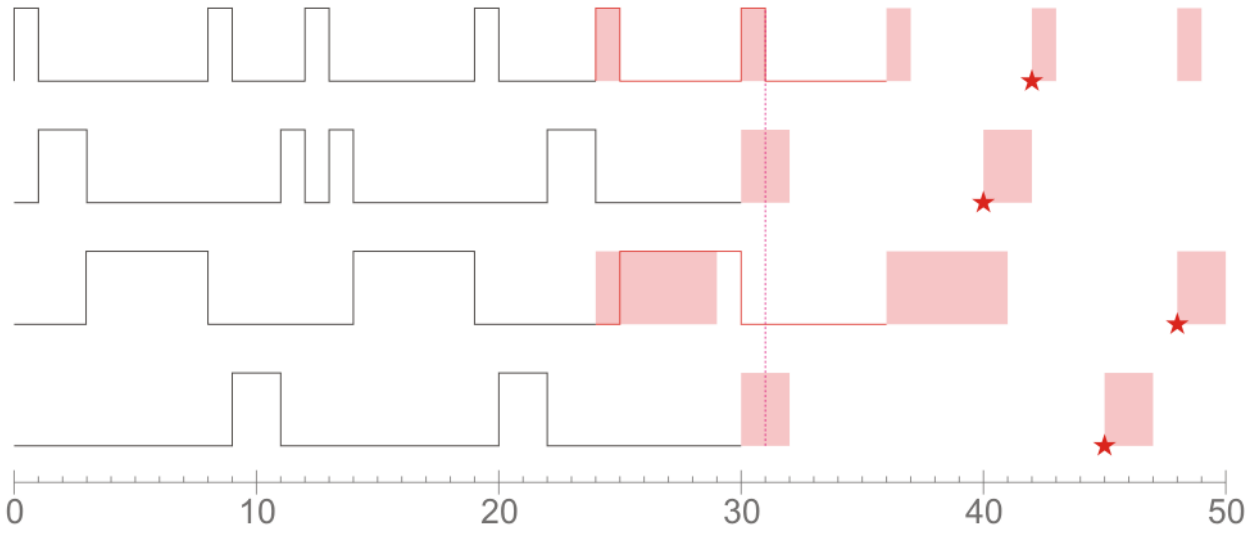
At this point in time, only Task C is ready to execute, so we start executing it. After it runs for 1 ms, Task D is also ready to execute, but it has a later deadline. After Task C executes for 4 ms, Task A is also ready to execute. As Task A and C both have the same deadline, we can proceed with either Task C or A, so as to save a context switch, keep Task A executing, but as soon as Task C finishes, we can schedule Task A.



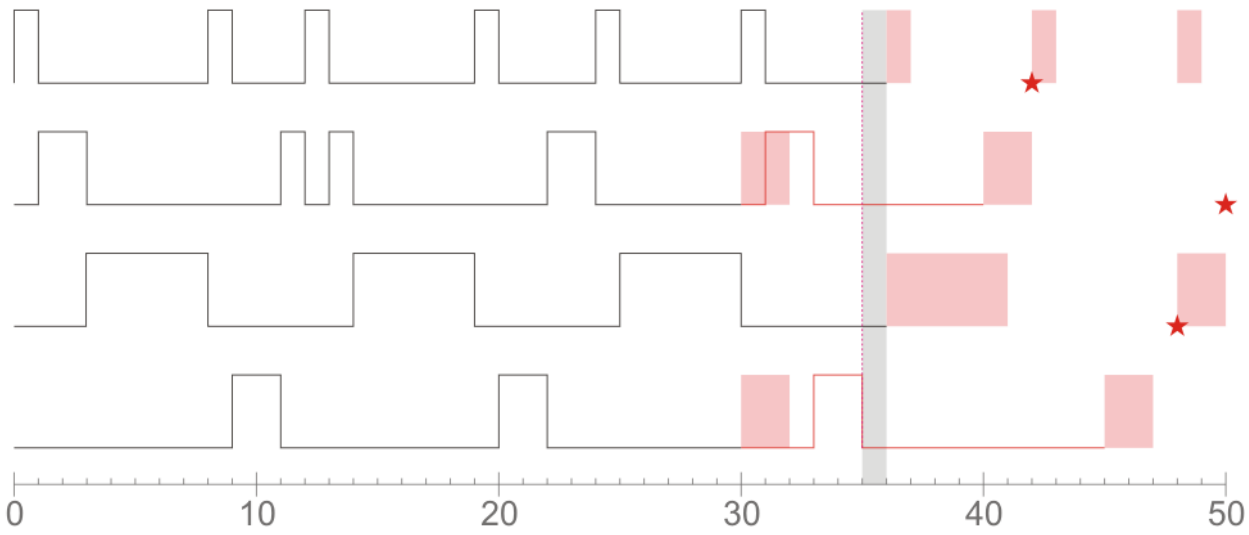
At this moment in time, Tasks B and D are ready to execute, and both have the same deadline, so we can pick either. Internally, we are likely to use a data structure that would schedule the task waiting the longest, so let's schedule Task D and then Task B.



At this point, between Tasks A and C, Task A has the earliest deadline, so schedule it, and then schedule Task C. Looking ahead, we note that Task A becomes ready again once Task C completes, so we can immediately schedule Task A again.

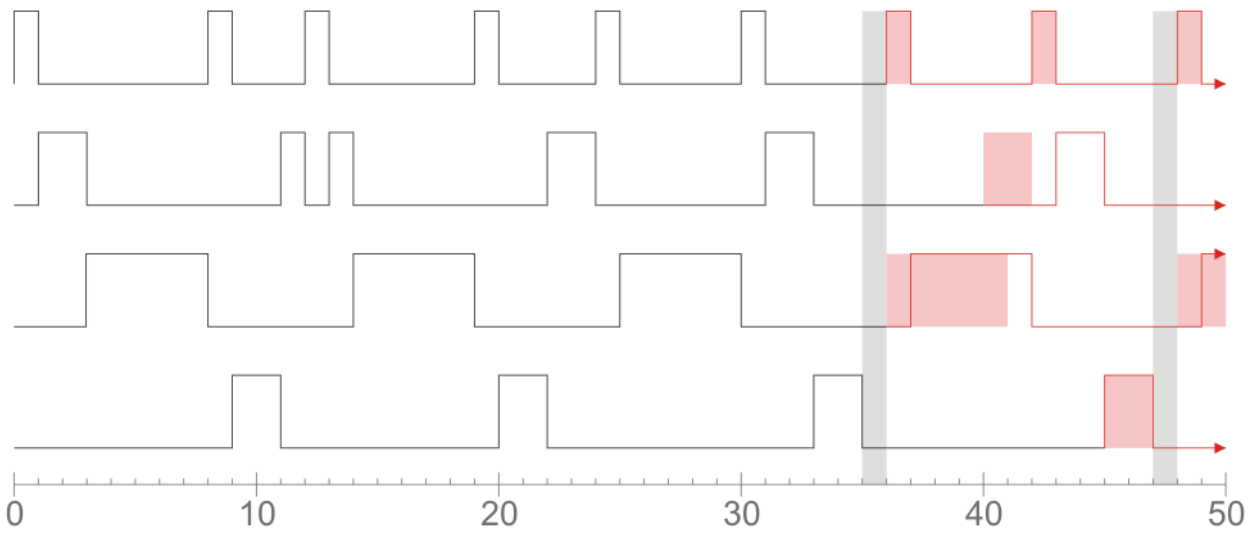


At this point, Tasks B and D are ready, and Task B has the earliest deadline, so schedule it. Following its execution, Task D is the only task ready to execute, so schedule it.



At this point in time, no tasks are ready to execute, so the processor is idle for 1 ms.

1. After this, Tasks A and C are ready to execute, and the first has the earliest deadline, so schedule it.
2. Then Task C is the only task ready to execute, so it is scheduled. While Task C is executing, Task B becomes ready, but it has a later deadline than Task C, so Task C continues.
3. When Task C completes, Task A is also ready again, and it has a deadline earlier than Task B, so it is scheduled next.
4. When Task A completes, Task B is run (it has a deadline earlier than Task D).
5. Next we run Task D, after which the processor is idle for 1 ms.
6. We then run Task A again.
7. Finally, we schedule Task C and it runs for 1 ms.



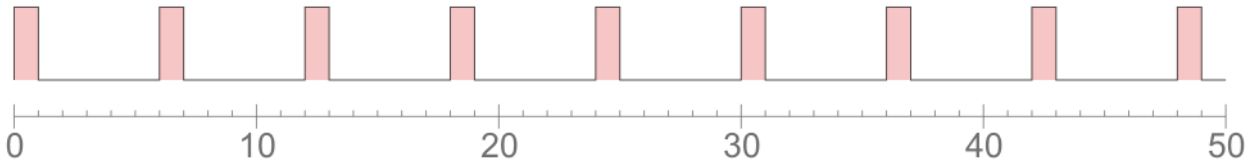
We note that

$$\frac{1}{6}+\frac{2}{10}+\frac{5}{12}+\frac{2}{15}=\frac{11}{12}=0.91\overline{6}$$

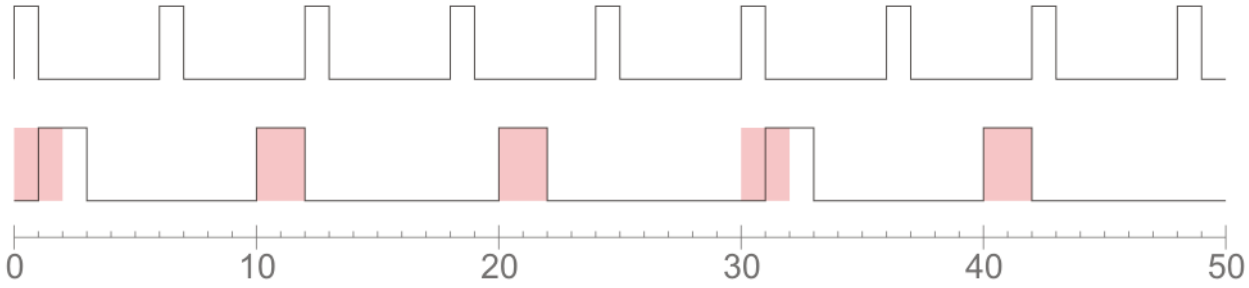
so we expect approximately a 92 % processor utilization. In the first 50 ms, we have a 96 % processor utilization, however, if we were to schedule the next 10 ms, we would find another 3 ms where the processor is idle.

A.2 Rate monotonic scheduling

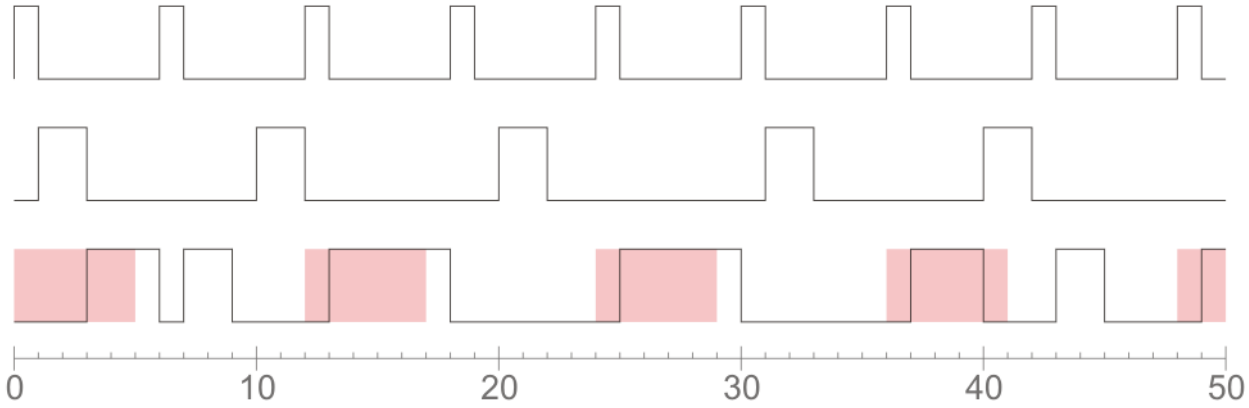
Choose the task with shortest period and schedule it to run at the start of each of its periods.



Next, indicate the worst-case computation time starting at the beginning of each of the periods of the next tasks, in this case, Task B. It may not always be possible to start Task B when it is ready, so start it as soon as possible when the processor is free. At time $t = 0$ and time $t = 30$, Task A is already executing.



We proceed to the next task with next longest period. Again, indicate a block corresponding to the worst-case execution time at the start of each period. Now try to schedule that block as soon as possible.



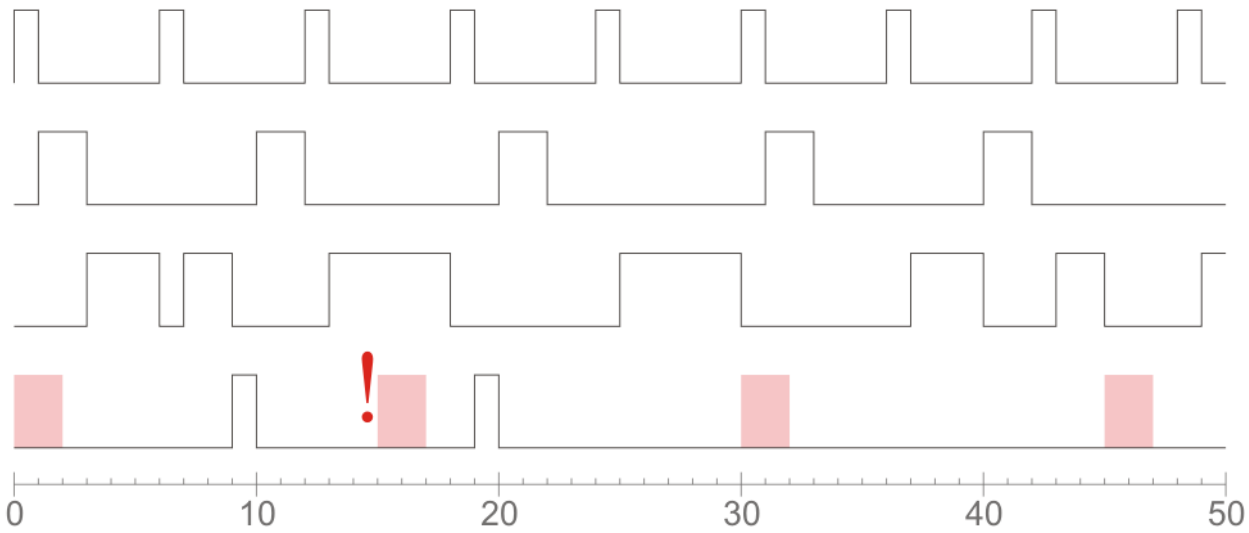
Note that as

$$\frac{1}{6}+\frac{2}{10}+\frac{5}{12}=0.78\overline{3}>3\left(2^{1/3}-1\right)\approx 0.780\ ,$$

we were not guaranteed that RM scheduling will work for these three tasks, let alone all four, but at least for the first 50 seconds, it appears that it is reasonable. We should really go out to

$$\text{lcm}(6\text{ ms}, 10\text{ ms}, 12\text{ ms}) = 60\text{ ms}$$

to ensure that it is schedulable. Next, we try to schedule the last task. As we are not guaranteed that RM will work for the three shortest-period tasks, it is even less likely to work when we consider all four. Indeed, we note that there is only a one millisecond interval in the first 15 ms that is not used, so immediately, Task D fails to meet its first deadline. The earliest that the first execution of Task D can complete is at $t = 19$ ms.



In this case, we have two options, if Task D is

1. if Task D is soft real-time, we may schedule it anyway; however,
2. if Task D is firm or hard real-time, it may be better to just not schedule it for its first execution.

If we do not schedule it, this could improve the chance that future executions will meet their deadlines.

Appendix B Representation of numbers

The on-line version has a section discussing the implementation of integers and real numbers.

Appendix C Fixed-point algorithms for RM schedulability tests

The on-line version has an appendix discussing the use of fixed-point algorithms for rate monotonic scheduling.

Appendix D Common data structures and algorithms

This appendix will simply summarize the some of the more common data structures used in this text, including a:

1. singly linked list with a pool of nodes,
2. stack using an array of entries,
3. queue using an array of entries,
4. priority queue using an array of entries,
5. disjoint set using a binary min-heap implemented using an array, and
6. sorted list using an AVL tree with a pool of nodes.

All functions implemented here attempt to use the best practices required for real-time systems, and therefore there is a significant amount of the code is not necessary for a general-purpose implementation of these data structures. Those aspects related to best practices are often color coded, to include the following:

1. All data structures will store pointers to entries. This can, of course, be changed; however, the sorted list data structure uses stacks of pointers for traversals.
2. While every container data structure (all but the disjoint set) has a full set of functions (implemented as macros) to determine if the data structure is full or empty and to return the capacity and size, all functions that may possibly change the contents of the container will always indicate when the operation was unsuccessful either by returning false or a null pointer.
3. All functions that have a return value almost universally use a single variable that is assigned and then returned, and any statement related to declaring, assigning to, or returning this variable are highlighted in magenta. For example:

```
void *p_front_entry = NULL;
// some code...
     p_front_entry = ...;
// some more code...
return p_front_entry;
```

   In each case, the return value is initialized with a default value, usually either `false` or `NULL`.
4. Memory should never be allocated dynamically at any time other than at initialization. Additionally, the programmer may want to specify a specific location in memory for an instance of a data structure. All data structures use a fixed amount of memory that is passed to the initialization routine. The term *capacity* refers to the number of entries that can be stored in the data structure, and the term *size* refers to the number of entries currently in the data structure. All aspects with respect to the management of memory, including the passing of that memory to the initialization function, any required initialization of that memory, and access to pools of nodes are highlighted in blue. For example,

```
void single_list_init( ..., single_node_t *p_single_list_entries,
                                                size_t single_list_capacity ) {
    // some code...
    p_this->p_node_pool = p_single_list_entries;
    p_this->capacity = single_list_capacity;

    // code to initialize the node pool
```

5. All code related to the mutually exclusive access of the data structure is given in dark red. For example, the semaphore may be declared, initialized, waited upon, or posted to:

```
binary_semaphore_t mutex;
binary_semaphore_init( &( p_this->mutex ), 1 );
binary_semaphore_wait( &( p_this->mutex ) );
binary_semaphore_post( &( p_this->mutex ) );
```

6. A version counter is associated with each data structure, and each time the data structure is changed, it is incremented. We use an unsigned 64-bit integer to ensure that overflow will not happen, but other implementations may take other steps to use less memory. In one data structure, the priority queue, this version counter is used as part of a lexicographical ordering that ensures first-in—first-out for entries with the same priority. Everything associated with this counter is highlighted in orange, for example:
```
uint64_t version;
p_this->version = 0;
++( p_this->version );
```
7. We will only describe the worst-case run time.
8. In order to highlight the encapsulation aspect of these data structures, the address of the data structure is always passed as a first argument to all functions, and it is given the argument name `p_this`, or *pointer-to-this-data-structure*.
9. Assertions are used to prevent invalid input. For example, we require that any pointers to be inserted into the container are not `NULL`.

Finally, in general, we try to use function names similar to the methods in the C++ standard template library (STL). Thus, familiarity with these data structures will also bring familiarity with the containers within the STL.

D.1 Singly linked lists

A singly linked list is a node-based linear data structure that allows Θ(1) insertions to both the front and back of the list, and Θ(1) insertions at the front. For an embedded system, memory allocation cannot be dynamic, and therefore there are two approaches:

1. the link list is initialized with a finite pool of nodes which it can use, or
2. the linked list acquires its nodes from a memory pool.

This implementation will use the first. Initialization therefore runs in Θ(*n*) time, but clearing can now run in Θ(1) time. Additionally, a binary semaphore will restrict access to the data structure while it is being updated. Finally, it may be necessary to determine whether a list has changed, and thus, a version field will be incremented with each change. Certain functions are defined as macros, where trivial operations ensure that the result cannot be on the left-hand side of an assignment.

An example of the use of this data structure is as follows:

```
single_list_t sl;
int array[11] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

single_list_init( &sl, (single_node_t *)malloc( 10*sizeof( single_node_t) ), 10 );

int i;

for ( i = 0; i < 5; ++i ) {
    printf( "%d ", single_list_push_front( &sl, array + 2*i ) );
    printf( "%d ", single_list_push_back(  &sl, array + 2*i + 1 ) );
}

printf( "%d\n", single_list_push_front( &sl, array + 10 ) );

for ( i = 0; i < 7; ++i ) {
    int *p_n = (int *)single_list_pop_front( &sl );
    printf( "%d ", *p_n );
}

printf( "\n" );

single_list_clear( &sl );
```

```c
// single_list.h
#ifndef CA_UWATERLOO_DWHARDER_SINGLE_LIST
#define CA_UWATERLOO_DWHARDER_SINGLE_LIST

#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>

typedef struct single_node {
    void               *p_entry;
    struct single_node *p_next;
} single_node_t;

void  single_node_init( single_node_t *const p_this,
                        void          *p_new_entry,
                        single_node   *p_new_next );

typedef struct {
    single_node_t *p_head;
    single_node_t *p_tail;
    size_t size;

    single_node_t *p_node_pool;
    size_t capacity;
    binary_semaphore_t mutex;
    uint64_t version;
} single_list_t;

void  single_list_init(       single_list_t *const p_this,
              single_node_t *p_new_single_list_entries, size_t new_single_list_capacity );

void *single_list_front(      single_list_t *const p_this );
void *single_list_back(       single_list_t *const p_this );
bool  single_list_push_front( single_list_t *const p_this, void *p_new_entry );
bool  single_list_push_back(  single_list_t *const p_this, void *p_new_entry );
void *single_list_pop_front(  single_list_t *const p_this );
void  single_list_clear(      single_list_t *const p_this );

#define single_list_empty( p_this )    ((p_this)->size == 0)
#define single_list_size( p_this )     ((p_this)->size + 0)
#define single_list_full( p_this )     ((p_this)->size == (p_this)->capacity)
#define single_list_capacity( p_this ) ((p_this)->capacity + 0)
#define single_list_version( p_this )  ((p_this)->version + 0)

#endif
```

```
// single_list.c
#include "single_list.h"
#include <assert.h>

void  single_node_init( single_node_t *const p_this,
                        void          *p_new_entry,
                        single_node   *p_new_next ) {
    p_this->p_entry = p_new_entry;
    p_this->p_next = p_new_next;
}

void single_list_init( single_list_t *const p_this,
                       single_node_t *p_new_single_list_entries,
                       size_t         new_single_list_capacity ) {
    p_this->p_head = NULL;
    p_this->p_tail = NULL;
    p_this->size = 0;

    p_this->p_node_pool = p_new_single_list_entries;
    p_this->capacity = new_single_list_capacity;

    // Place all the nodes in the array onto a 'free list' where each node
    // points to the next in the list and the last node points to NULL.
    size_t i;
    single_node_t *p_node_itr = p_this->p_node_pool;

    for ( i = 0; i < new_single_list_capacity - 1; ++i ) {
        p_node_itr->p_next = p_node_itr + 1;
        ++( p_node_itr );
    }

    p_node_itr->p_next = NULL;

    binary_semaphore_init( &( p_this->mutex ), 1 );
    p_this->version = 0;
}

void *single_list_front( single_list_t *const p_this ) {
    void *p_front_entry = NULL;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( single_list_empty( p_this ) ) {
            // p_front_entry = NULL;
        } else {
            p_front_entry = p_this->p_head->p_entry;
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return p_front_entry;
}

void *single_list_back( single_list_t *const p_this ) {
    void *p_back_entry = NULL;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( single_list_empty( p_this ) ) {
            // p_back_entry = NULL;
        } else {
            p_back_entry = p_this->p_tail->p_entry;
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return p_back_entry;
}
```

```
bool single_list_push_front( single_list_t *const p_this, void *p_new_entry ) {
    bool success = false;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( single_list_full( p_this ) ) {
            // success = false;
        } else {
            success = true;

            // Pop a node off of the free list
            single_node_t *p_new_node = p_this->p_node_pool;
            p_this->p_node_pool = p_this->p_node_pool->p_next;

            single_node_init( p_new_node, p_new_entry, p_this->p_head );
            p_this->p_head = p_new_node;

            if ( single_list_empty( p_this ) ) {
                p_this->p_tail = p_new_node;
            }

            ++( p_this->size );
            ++( p_this->version );
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return success;
}

bool single_list_push_back( single_list_t *const p_this, void *p_new_entry ) {
    assert( p_new_entry != NULL );
    bool success = false;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( single_list_full( p_this ) ) {
            // success = false;
        } else {
            success = true;

            single_node_t *p_new_node = p_this->p_node_pool;
            p_this->p_node_pool = p_this->p_node_pool->p_next;

            p_new_node->p_entry = p_new_entry;
            p_new_node->p_next = NULL;

            if ( single_list_empty( p_this ) ) {
                p_this->p_head = p_new_node;
            } else {
                p_this->p_tail->p_next = p_new_node;
            }

            p_this->p_tail = p_new_node;

            ++( p_this->size );
            ++( p_this->version );
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return success;
}
```

```
void *single_list_pop_front( single_list_t *const p_this ) {
    void *p_popped_entry = NULL;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( single_list_empty( p_this ) ) {
            // p_popped_entry = NULL;
        } else {
            p_popped_entry = p_this->p_head->p_entry;

            if ( single_list_size( p_this ) == 1 ) {
                p_this->p_node_pool = p_this->p_head;  // p_next is already NULL

                p_this->p_head = NULL;
                p_this->p_tail = NULL;
                p_this->size = 0;
            } else {
                single_node_t *p_popped_node = p_this->p_head;
                p_this->p_head = p_this->p_head->p_next;
                p_popped_node->p_next = p_this->p_node_pool;
                p_this->p_node_pool = p_popped_node;

                --( p_this->size );
            }

            ++( p_this->version );
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return p_popped_entry;
}

void single_list_clear( single_list_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) );
        if ( !single_list_empty( p_this ) ) {
            p_this->p_tail->p_next = p_this->p_node_pool;
            p_this->p_node_pool = p_this->p_head;

            p_this->p_head = NULL;
            p_this->p_tail = NULL;
            p_this->size = 0;
            ++( p_this->version );
        }
    binary_semaphore_post( &( p_this->mutex ) );
}
```

D.2 An iterator for a singly linked list

Suppose it is necessary to iterate through all of the entries of a linked list. It would be possible to simply access the `p_head` field and step through the entries, but what if the data structure is changed and, for example, the node you are currently referencing is no longer a component of the linked list? For this, we will introduce an iterator:

```
single_iterator_t itr;
single_iterator_begin( &itr, &some_single_list );
some_type *p_next_entry;
bool valid;

while ( p_next_entry = (some_type *)single_iterator_next( &itr, &valid ) ) {
        // Work with 'p_next_entry'...
}

if ( !valid ) {
    // We did not iterate through all of the entries of the list
    //  - something changed in the list prior to getting to the end
}
```

There are two possible ends to this loop, whereby either

1. all entries in the linked list have been iterated through, or
2. prior to iterating through all the entries, a change was made to the linked list either by this task or another.

Consequently, it is always necessary to pass an argument that registers whether or not the current result is even valid, or if a change has been made to the data structure after it was created.

In the second case, the version stored is no longer equal to the version of the linked list.

```
// single_iterator.h
#ifndef CA_UWATERLOO_DWHARDER_SINGLE_ITERATOR
#define CA_UWATERLOO_DWHARDER_SINGLE_ITERATOR

#include "single_list.h"

typedef struct {
    single_list_t *p_list;
    single_node_t *p_current_node;
    uint64_t version;
} single_iterator_t;

void single_iterator_begin( single_iterator_t *p_this,
                            single_list_t     *p_new_list );
void single_iterator_is_after_end( single_iterator_t *p_this, bool *valid );
void *single_iterator_next( single_iterator_t *p_this, bool *valid );

#endif
```

```
// single_iterator.c
#include "single_iterator.h"
void single_iterator_begin( single_iterator_t *p_this,
                            single_list_t     *p_new_list ) {
    binary_semaphore_wait( &( p_new_list->mutex ) );
        p_this->p_list         = p_new_list;
        p_this->p_current_node = p_new_list->p_head;
        p_this->version        = p_new_list->version;
    binary_semaphore_post( &( p_new_list->mutex ) );
}

bool single_iterator_is_after_end( single_iterator_t *p_this, bool *valid ) {
    bool is_after_end;

    binary_semaphore_wait( &( p_this->p_list->mutex ) );
        if ( p_this->version != p_this->p_list->version ) {
            *valid = false;
        } else {
            *valid = true;
        }

        is_after_end = ( p_this->p_current_node == NULL );
    binary_semaphore_post( &( p_this->p_list->mutex ) );

    return is_after_end;
}

void *single_iterator_next( single_iterator_t *p_this, bool *valid ) {
    void *p_current_entry = NULL;

    binary_semaphore_wait( &( p_this->p_list->mutex ) );
        if ( p_this->version != p_this->p_list->version ) {
            *valid = false;
            // p_current_entry = NULL;
        } else {
            *valid = true;

            if ( p_this->p_current_node != NULL ) {
                p_current_entry = p_current_node->p_entry;
                p_this->p_current_node = p_this->p_current_node->p_next;
            }
        }
    binary_semaphore_post( &( p_this->p_list->mutex ) );

    return p_current_entry;
}
```

D.3 Stacks

A stack is a last-in—first-out (LIFO) data structure. For an embedded system, the capacity must be fixed and access to the data structure must be restricted by a semaphore. All operations run in $\Theta(1)$ time. A version number is included to allow a task to determine whether or not a change has occurred to a stack.

```
// stack.h
#ifndef CA_UWATERLOO_DWHARDER_STACK
#define CA_UWATERLOO_DWHARDER_STACK

#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>

typedef struct {
    size_t size;

    void **pp_entries;
    size_t capacity;
    binary_semaphore_t mutex;
    uint64_t version;
} stack_t;

void  stack_init( stack_t  *const p_this,
                  void    **pp_new_stack_entries,
                  size_t    new_stack_capacity );
bool  stack_push( stack_t *const p_this, void *p_new_entry );
void *stack_top( stack_t *const p_this );
void *stack_pop( stack_t *const p_this );
void  stack_clear( stack_t *const p_this );

#define stack_empty( p_this )    ((p_this)->size == 0)
#define stack_size( p_this )     ((p_this)->size + 0)
#define stack_full( p_this )     ((p_this)->size == (p_this)->capacity)
#define stack_capacity( p_this ) ((p_this)->capacity + 0)
#define stack_version( p_this )  ((p_this)->version + 0)

#endif
```

```
// stack.c
#include "stack.h"
#include <assert.h>

void stack_init( stack_t  *const p_this,
                 void    **pp_new_stack_entries,
                 size_t    new_stack_capacity ) {
    p_this->size = 0;

    p_this->pp_entries = pp_new_stack_entries;
    p_this->capacity = new_stack_capacity;
    binary_semaphore_init( &( p_this->mutex ), 1 );
    p_this->version = 0;
}

bool stack_push( stack_t *const p_this, void *p_new_entry ) {
    assert( p_new_entry != NULL );
    bool success = false;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( stack_full( p_this ) ) {
            // success = false;
        } else {
            success = true;

            p_this->pp_entries[p_this->size] = p_new_entry;
            ++( p_this->size );
            ++( p_this->version );
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return success;
}

void *stack_top( stack_t *const p_this ) {
    void *p_top_entry = NULL;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( stack_empty( p_this ) ) {
            // p_top_entry = NULL;
        } else {
            p_top_entry = p_this->pp_entries[p_this->size - 1];
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return p_top_entry;
}
```

```
void *stack_pop( stack_t *const p_this ) {
    void *p_popped_entry = NULL;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( stack_empty( p_this ) ) {
            // p_popped_entry = NULL;
        } else {
            --( p_this->size );
            p_popped_entry = p_this->pp_entries[p_this->size];
            ++( p_this->version );
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return p_popped_entry;
}

void stack_clear( stack_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) );
        if ( !stack_empty( p_this ) ) {
            p_this->size = 0;
            ++( p_this->version );
        }
    binary_semaphore_post( &( p_this->mutex ) );
}
```

D.4 Queues

A queue is a first-in—first-out data structure. Like the stack, a queue for an embedded system must have a fixed capacity and access to the data structure must be restricted by a semaphore. All operations run in Θ(1) time. A version number is included to allow a task to determine whether or not a change has occurred to a stack.

```
// queue.h
#ifndef CA_UWATERLOO_DWHARDER_QUEUE
#define CA_UWATERLOO_DWHARDER_QUEUE

#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>

typedef struct {
    size_t size;
    size_t head;
    size_t tail;

    void **pp_entries;
    size_t capacity;
    binary_semaphore_t mutex;
    uint64_t version;
} queue_t;

void  queue_init( queue_t  *const p_this,
                  void    **pp_new_queue_entries,
                  size_t    new_queue_capacity );
bool  queue_push( queue_t *const p_this, void *p_new_entry );
void *queue_front( queue_t *const p_this );
void *queue_pop( queue_t *const p_this );
void  queue_clear( queue_t *const p_this );

#define queue_empty( p_this )    ((p_this)->size == 0)
#define queue_size( p_this )     ((p_this)->size + 0)
#define queue_full( p_this )     ((p_this)->size == (p_this)->capacity)
#define queue_capacity( p_this ) ((p_this)->capacity + 0)
#define queue_version( p_this )  ((p_this)->version + 0)

#endif
```

```
// queue.c
#include "queue.h"
#include <assert.h>

void queue_init( queue_t  *const p_this,
                 void    **pp_new_queue_entries,
                 size_t    new_queue_capacity ) {
    p_this->size = 0;
    p_this->head = 0;
    p_this->tail = queue_capacity - 1;

    p_this->pp_entries = pp_new_queue_entries;
    p_this->capacity = new_queue_capacity;
    binary_semaphore_init( &( p_this->mutex ), 1 );
    p_this->version = 0;
}

bool queue_push( queue_t *const p_this, void *p_new_entry ) {
    assert( p_new_entry != NULL );
    bool success = false;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( queue_full( p_this ) ) {
            // success = false;
        } else {
            success = true;

            p_this->tail = (p_this->tail + 1) % p_this->capacity;
            p_this->pp_entries[p_this->tail] = p_new_entry;
            ++( p_this->size );
            ++( p_this->version );
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return success;
}

void *queue_front( queue_t *const p_this ) {
    void *p_front_entry = NULL;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( queue_empty( p_this ) ) {
            // p_front_entry = NULL;
        } else {
            p_front_entry = p_this->pp_entries[p_this->head];
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return p_front_entry;
}
```

```
void *queue_pop( queue_t *const p_this ) {
    void *p_popped_entry = NULL;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( queue_empty( p_this ) ) {
            // p_popped_entry = NULL;
        } else {
            p_popped_entry = p_this->pp_entries[p_this->head];

            p_this->head = (p_this->head + 1) % p_this->capacity;
            --( p_this->size );
            ++( p_this->version );
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return p_popped_entry;
}

void queue_clear( queue_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) );
        if ( !queue_empty( p_this ) ) {
            p_this->size = 0;
            p_this->head = 0;
            p_this->tail = p_this->capacity - 1;
            ++( p_this->version );
        }
    binary_semaphore_post( &( p_this->mutex ) );
}
```

D.5 Priority queues (heap based)

A priority queue is a sorted queue based on the *priority* or *key* of each entry. The entry with the lowest priority is the entry to be popped next. In this implementation, the lower the value of the key, the higher the priority. The worst-case run times for pushing and popping are $\Theta(\ln(n))$ while the priority queue can be cleared and the front element can be accessed both in $\Theta(1)$ time. In order to ensure that this data structure provides a first-in—first-out behavior for entries with the same priority, it uses a lexicographical ordering based on the (*key*, *version*) where $(k_1, v_1) < (k_2, v_2)$ if and only if $k_1 < k_2$ or both $k_1 = k_2$ and $v_1 < v_2$.

```
// priority_queue_heap.h
#ifndef CA_UWATERLOO_DWHARDER_HEAP_BASED_PRIORITY_QUEUE
#define CA_UWATERLOO_DWHARDER_HEAP_BASED_PRIORITY_QUEUE

#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>

#ifndef CA_UWATERLOO_DWHARDER_PRIORITY_QUEUE_TRIPLET
#define CA_UWATERLOO_DWHARDER_PRIORITY_QUEUE_TRIPLET
typedef struct {
    void *p_entry;
    int key;
    uint64_t version;         // Also needed for the lexicographical order of the priority
} priority_triplet_t;
#endif

typedef struct {
    size_t size;

    priority_triplet_t *p_entries;
    size_t capacity;
    binary_semaphore_t mutex;
    uint64_t version;
} priority_queue_heap_t;

void  priority_queue_heap_init(  priority_queue_heap_t   *const p_this,
                                 priority_triplet_t *p_new_priority_queue_heap_entries,
                                 size_t              new_priority_queue_heap_capacity );
bool  priority_queue_heap_push(  priority_queue_heap_t *const p_this,
                                 void *p_new_entry, int new_key );
void *priority_queue_heap_front( priority_queue_heap_t *const p_this );
void *priority_queue_heap_pop(   priority_queue_heap_t *const p_this );
void  priority_queue_heap_clear( priority_queue_heap_t *const p_this );

#define priority_queue_heap_empty( p_this )    ((p_this)->size == 0)
#define priority_queue_heap_size( p_this )     ((p_this)->size + 0)
#define priority_queue_heap_full( p_this )     ((p_this)->size == (p_this)->capacity)
#define priority_queue_heap_capacity( p_this ) ((p_this)->capacity + 0)
#define priority_queue_heap_version( p_this )  ((p_this)->version + 0)

#endif
```

```c
// priority_queue_heap.c
#include "priority_queue_heap.h"
#include <assert.h>

void priority_queue_heap_init( priority_queue_heap_t   *const p_this,
                               priority_triplet_t *p_new_entries,
                               size_t              new_capacity ) {
    p_this->size = 0;

    p_this->p_entries = p_new_entries - 1;  // front is at p_entries[1]
    p_this->capacity =  new_capacity;
    binary_semaphore_init( &( p_this->mutex ), 1 );
    p_this->version = 0;
}

bool priority_queue_heap_push( priority_queue_heap_t *const p_this,
                               void *p_new_entry, int new_key ) {
    assert( p_new_entry != NULL );
    bool success = false;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( priority_queue_heap_full( p_this ) ) {
            // success = false;
        } else {
            success = true;

            ++( p_this->size );
            size_t position = p_this->size;
            size_t parent = position/2;

            // No need to compare with .version
            while ( ( position > 1 ) && ( new_key < p_this->p_entries[parent].key ) ) {
                p_this->p_entries[position] = p_this->p_entries[parent];
                position = parent;
                parent /= 2;
            }

            p_this->p_entries[position].p_entry = p_new_entry;
            p_this->p_entries[position].key = new_key;
            p_this->p_entries[position].version = p_this->version;

            ++( p_this->version );
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return success;
}

void *priority_queue_heap_front( priority_queue_heap_t *const p_this ) {
    void *p_front_entry = NULL;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( priority_queue_heap_empty( p_this ) ) {
            // p_front_entry = NULL;
        } else {
            p_front_entry = p_this->p_entries[1].p_entry;
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return p_front_entry;
}
```

```
#define priority_triplet_less( a, b )                                          \
   ( (p_this->p_entries[a].key < p_this->p_entries[b].key) || (                 \
             (p_this->p_entries[a].key == p_this->p_entries[b].key)             \
          && (p_this->p_entries[a].version < p_this->p_entries[b].version)      \
   ) )

void *priority_queue_heap_pop( priority_queue_heap_t *const p_this ) {
    void *p_popped_entry = NULL;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( priority_queue_heap_empty( p_this ) ) {
            // p_popped_entry = NULL;
        } else {
            p_popped_entry = p_this->p_entries[1].p_entry;

            size_t old_position = p_this->size;
            size_t new_position = 1;
            size_t left_child =   2;
            size_t right_child =  3;

            --( p_this->size );

            while ( right_child <= p_this->size ) {
                if ( priority_triplet_less( left_child, old_position ) ) {
                    if ( priority_triplet_less( left_child, right_child  ) ) {
                        p_this->p_entries[new_position] = p_this->p_entries[left_child];
                        new_position = left_child;
                    } else {
                        p_this->p_entries[new_position] = p_this->p_entries[right_child];
                        new_position = right_child;
                    }
                } else if ( priority_triplet_less( right_child, old_position ) ) {
                    p_this->p_entries[new_position] = p_this->p_entries[right_child];
                    new_position = right_child;
                } else {
                    break;
                }

                left_child = 2*new_position;
                right_child = left_child + 1;
            }

            if ( left_child == p_this->size ) {
                if ( priority_triplet_less( left_child, old_position ) ) {
                    p_this->p_entries[new_position] = p_this->p_entries[left_child];
                    new_position = left_child;
                }
            }

            p_this->p_entries[new_position] = p_this->p_entries[old_position];

            ++( p_this->version );
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return p_popped_entry;
}
```

```
void priority_queue_heap_clear( priority_queue_heap_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) );
        if ( !priority_queue_heap_empty( p_this ) ) {
            p_this->size = 0;
            ++( p_this->version );
        }
    binary_semaphore_post( &( p_this->mutex ) );
}
```

D.6 Priority queues (array-based)

The previous implementation of a priority queue is heap based to ensure that all operations are O(ln(*n*)), but the additional overhead and the size of the code base may be prohibitive in smaller embedded systems where the priority queue may never have more than a fixed number of entries. In this case, a simpler array-based priority queue may be more than appropriate, as all operations may be O(*n*), but if *n* is guaranteed to be sufficiently small, the additional effort required may be compensated by the simplicity and small footprint of the algorithm. This is similar to using insertion sort over quick sort if the number of items being sorted is sufficiently small.

```
// priority_queue_array.h
#ifndef CA_UWATERLOO_DWHARDER_ARRAY_BASED_PRIORITY_QUEUE
#define CA_UWATERLOO_DWHARDER_ARRAY_BASED_PRIORITY_QUEUE

#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>


#ifndef CA_UWATERLOO_DWHARDER_PRIORITY_QUEUE_TRIPLET
#define CA_UWATERLOO_DWHARDER_PRIORITY_QUEUE_TRIPLET
typedef struct {
    void *p_entry;
    int key;
    uint64_t version;         // Also needed for the lexicographical order of the priority
} priority_triplet_t;
#endif

typedef struct {
    size_t size;
    size_t head;
    size_t tail;

    priority_triplet_t *p_entries;
    size_t capacity;
    binary_semaphore_t mutex;
    uint64_t version;
} priority_queue_array_t;

void  priority_queue_array_init( priority_queue_array_t  *const p_this,
                                 priority_triplet_t      *p_new_entries,
                                 size_t                   new_capacity );
bool  priority_queue_array_push( priority_queue_array_t *const p_this,
                                 void *p_new_entry, int new_key );
void *priority_queue_array_front( priority_queue_array_t *const p_this );
void *priority_queue_array_pop( priority_queue_array_t *const p_this );
void  priority_queue_array_clear( priority_queue_array_t *const p_this );

#define priority_queue_array_empty( p_this )    ((p_this)->size == 0)
#define priority_queue_array_size( p_this )     ((p_this)->size + 0)
#define priority_queue_array_full( p_this )     ((p_this)->size == (p_this)->capacity)
#define priority_queue_array_capacity( p_this ) ((p_this)->capacity + 0)
#define priority_queue_array_version( p_this )  ((p_this)->version + 0)

#endif
```

```
// priority_queue_array.c
#include "queue.h"
#include <assert.h>

void priority_queue_array_init( priority_queue_array_t *const p_this,
                                priority_triplet_t     *p_new_entries,
                                size_t                  new_capacity ) {
    p_this->size = 0;
    p_this->head = 0;
    p_this->tail = new_capacity - 1;

    p_this->p_entries = p_new_entries;
    p_this->capacity = new_capacity;
    binary_semaphore_init( &( p_this->mutex ), 1 );
    p_this->version = 0;
}

bool priority_queue_array_push( priority_queue_array_t *const p_this,
                                void                   *p_new_entry,
                                int                     new_key ) {
    assert( p_new_entry != NULL );
    bool success = false;
    size_t i, current, empty;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( priority_queue_array_full( p_this ) ) {
            // success = false;
        } else {
            success = true;

            current  =  p_this->tail;
            p_this->tail = (p_this->tail + 1) % p_this->capacity;
            empty = p_this->tail;

            for ( i = 0; i < p_this->size; ++i ) {
                if ( p_this->p_entries[current].key > new_key ) {
                    p_this->p_entries[empty] = p_this->p_entries[current];
                    current  = ( current == 0) ? p_this->capacity - 1 : ( current - 1);
                    empty    =    (empty == 0) ? p_this->capacity - 1 :    (empty - 1);
                } else {
                    break;
                }
            }

            p_this->p_entries[empty].p_entry = p_new_entry;
            p_this->p_entries[empty].key = new_key;
            p_this->p_entries[empty].version = p_this->version;

            ++( p_this->size );
            ++( p_this->version );
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return success;
}
```

```
void *priority_queue_array_front( priority_queue_array_t *const p_this ) {
    void *p_front_entry = NULL;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( priority_queue_array_empty( p_this ) ) {
            // p_front_entry = NULL;
        } else {
            p_front_entry = p_this->p_entries[p_this->head].p_entry;
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return p_front_entry;
}

void *priority_queue_array_pop( priority_queue_array_t *const p_this ) {
    void *p_popped_entry = NULL;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( priority_queue_array_empty( p_this ) ) {
            // p_popped_entry = NULL;
        } else {
            p_popped_entry = p_this->p_entries[p_this->head].p_entry;

            p_this->head = (p_this->head + 1) % p_this->capacity;
            --( p_this->size );
            ++( p_this->version );
        }

    binary_semaphore_post( &( p_this->mutex ) );

    return p_popped_entry;
}

void priority_queue_array_clear( priority_queue_array_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) );
        if ( !priority_queue_array_empty( p_this ) ) {
            p_this->size = 0;
            p_this->head = 0;
            p_this->tail = p_this->capacity - 1;
            ++( p_this->version );
        }
    binary_semaphore_post( &( p_this->mutex ) );
}
```

D.7 Disjoint sets

The on-line version has an implementation of a disjoint set.

D.8 Sorted list

Storing a sorted list requires either:

1. an array with O(n) insertions, erases and finds, or
2. a balanced tree with Θ(ln(n)) insertions, erases and finds.

Trees usually require some form of dynamic memory allocation, but in this case, we will use a memory pool supplied by the programmer at initialization time. This implementation uses AVL balancing to maintain the balance whereby the height of the left and right subtrees differ by at most one. Rather than storing the heights explicitly, the balance is either left heavy (the height of the left subtree is greater than the right), balanced or right heavy (the reverse of left heavy). Unlike many implementations of AVL trees, this uses iteration and therefore requires an explicit stack; most implementations use recursion, which implicitly uses the call stack. Additionally, AVL trees nominally use height balancing, requiring each node to store the height of the subtree rooted at that node. This, however, requires at least one additional byte and arithmetic computation. This implementation reduces this to a ternary-valued flag: the node is either *left heavy* (the height of the left sub-tree is greater than that of the right), *balanced* (equal heights) or *right heavy*. As a new node is included in the tree, changes to the balance can be determined solely on changes to balance in the subtrees.

Note that Rule 4 of the JPL coding standard specifies that: There shall be no direct or indirect use of recursive function calls.

Access to the nodes is through pointers. This memory requirement could, however, be significantly reduced if the offset into the array was stored, instead. In this case, the memory required for each node would be reduced to:

1. the size of the key,
2. the size of the entry,
3. $4\lceil \log_2(n) \rceil$ bits where n is the maximum number of nodes in the structure, and
4. 2 bits to store AVL balance.

Thus, the memory required in addition to the memory required for actually storing the keys and entries is 3.75 bytes per entry with 128 entries, 7.75 bytes per entry with 32 768 entries, and 15.75 bytes per entry with 2 147 483 648 entries. All of these are significantly less than the current version which uses pointers as opposed to indices.

The allocated blocks of memory should, where possible, fit into a minimal number of memory blocks on systems that use caches.

In order to allow the iterator to operate in Θ(1) time per operation with Θ(1) memory, each node has previous and next pointers. All code associated with these is in **brown**. Additionally, the sorted list structure has front and back pointers, thereby essentially creating a doubly linked list within the AVL tree. All this code can safely be removed without adversely affecting the implementation. With the previous and next pointers, iterating through all the entries in a sorted list is Θ(n).

```
binary_semaphore_wait( &( p_some_list->mutex ), 0 );
    for ( sorted_node_t *itr = p_some_list->p_front; itr != NULL; itr = itr->p_next ) {
        // Do something with itr->p_entry
    }
binary_semaphore_post( &( p_some_list->mutex ), 0 );
```

It is still possible to create an iterator for a sorted list stored as an AVL tree if there are no previous and next pointers, but this requires two stacks requiring $\Theta(\ln(n))$ additional memory, and each operation to step forward or backward requires $\Theta(\ln(n))$ time.

```
// sorted_node.h
#ifndef CA_UWATERLOO_DWHARDER_SORTED_NODE
#define CA_UWATERLOO_DWHARDER_SORTED_NODE

#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>

#define AVL_LEFT_HEAVY  -1
#define AVL_BALANCED     0
#define AVL_RIGHT_HEAVY  1

typedef struct sorted_node {
    void               *p_entry;
    int                 key;              // The order will be decided by the key
    int                 balance;
    struct sorted_node *p_left;
    struct sorted_node *p_right;
    struct sorted_node *p_previous;
    struct sorted_node *p_next;
} sorted_node_t;

void sorted_node_init( sorted_node_t *const p_this,
                       void *p_new_entry,
                       int new_key,
                       int new_balance,
                       sorted_node_t *p_new_left,
                       sorted_node_t *p_new_right,
                       sorted_node_t *p_new_previous,
                       sorted_node_t *p_new_next );

#define sorted_node_entry( p_this )          ((p_this)->p_entry + 0)
#define sorted_node_key( p_this )            ((p_this)->key + 0)
#define sorted_node_balance( p_this )        ((p_this)->balance + 0)
#define sorted_node_is_left_heavy( p_this )  ((p_this)->balance == AVL_LEFT_HEAVY)
#define sorted_node_is_balanced( p_this )    ((p_this)->balance == AVL_BALANCED)
#define sorted_node_is_right_heavy( p_this ) ((p_this)->balance == AVL_RIGHT_HEAVY)
#define sorted_node_left( p_this )           ((p_this)->p_left + 0)
#define sorted_node_right( p_this )          ((p_this)->p_right + 0)
#define sorted_node_previous( p_this )       ((p_this)->p_previous + 0)
#define sorted_node_next( p_this )           ((p_this)->p_next + 0)
#endif

// sorted_node.c
#include "sorted_node.h"

void sorted_node_init( sorted_node_t *const p_this,
                       void *p_new_entry,
                       int new_key ) {
    p_this->p_entry    = p_new_entry;
    p_this->key        = new_key;
    p_this->balance    = AVL_BALANCED;
    p_this->p_left     = NULL;
    p_this->p_right    = NULL;
    p_this->p_previous = NULL;
    p_this->p_next     = NULL;
}
```

```
// sorted_list.h
#ifndef CA_UWATERLOO_DWHARDER_SORTED_LIST
#define CA_UWATERLOO_DWHARDER_SORTED_LIST

#include "sorted_node.h"
#include "stack.h"
#include <assert.h>

typedef struct {
    sorted_node_t *p_root;
    size_t size;
    stack_t traversal_stack;
    size_t maximum_depth;

    sorted_node_t *p_node_pool;
    size_t capacity;
    binary_semaphore_t mutex;
    uint64_t version;
    sorted_node_t *p_front;
    sorted_node_t *p_back;
} sorted_list_t;

size_t sorted_list_maximum_depth( size_t sorted_list_capacity );

void sorted_list_init( sorted_list_t *const p_this,
                       sorted_node_t *p_new_entries,
                       size_t         new_capacity,
                       void         **pp_new_traversal_stack_entries );

void *sorted_list_front ( sorted_list_t *const p_this );
void *sorted_list_back  ( sorted_list_t *const p_this );
void *sorted_list_find  ( sorted_list_t *const p_this, int sought_key );
bool  sorted_list_insert( sorted_list_t *const p_this, void *p_new_entry, int new_key );
void *sorted_list_erase ( sorted_list_t *const p_this, int old_key );
void  sorted_list_clear ( sorted_list_t *const p_this );

#define sorted_list_root( p_this )     ((p_this)->p_root + 0)

#define sorted_list_empty( p_this )    ((p_this)->size == 0)
#define sorted_list_size( p_this )     ((p_this)->size + 0)
#define sorted_list_full( p_this )     ((p_this)->size == (p_this)->capacity)
#define sorted_list_capacity( p_this ) ((p_this)->capacity + 0)
#define sorted_list_version( p_this )  ((p_this)->version + 0)

#endif
```

```
// sorted_list.c
#include "sorted_list.h"

// This calculates 1.5*floor( log[2]( capacity ) )
//  - this overestimates the maximum capacity of a traversal stack by
//    at most two, but usually zero or one.
size_t sorted_list_maximum_depth( size_t sorted_list_capacity ) {
    assert( sorted_list_capacity > 0 );
    size_t stack_capacity = 0;

    while ( sorted_list_capacity >>= 1 ) {
        ++stack_capacity;
    }

    return stack_capacity + (stack_capacity >> 1);
}

void sorted_list_init( sorted_list_t *const p_this,
                       sorted_node_t *p_new_entries,
                       size_t         new_capacity,
                       void         **pp_new_traversal_stack_entries ) {
    p_this->p_root = NULL;
    p_this->size = 0;

    p_this->maximum_depth = sorted_list_maximum_depth( new_capacity );
    stack_init( &( p_this->traversal_stack ), pp_new_traversal_stack_entries,
                p_this->maximum_depth );

    p_this->capacity = new_capacity;
    p_this->p_node_pool = p_new_entries;

    // Have each node point to the next in the pool through the left subtree
    size_t i;
    sorted_node_t *p_node_itr = p_this->p_node_pool;

    for ( i = 0; i < new_capacity - 1; ++i ) {
        p_node_itr->p_next = p_node_itr + 1;
        ++( p_node_itr );
    }

    p_node_itr->p_next = NULL;

    binary_semaphore_init( &( p_this->mutex ), 1 );
    p_this->version = 0;

    p_this->p_front = NULL;
    p_this->p_back  = NULL;
}
```

For both finding the first and last entries in the sorted list, we use a for loop instead of a while loop. This ensures that, even if there is an error, these loops will terminate and not go into infinite loops.

```
void *sorted_list_front( sorted_list_t *const p_this ) {
    void *p_front_entry = NULL;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( sorted_list_size( p_this ) == 0 ) {
            // p_front_entry = NULL;
        } else {
            size_t depth;
            sorted_node_t *p_front = sorted_list_root( p_this );

            for ( depth = 0; depth < p_this->maximum_depth; ++depth ) {
                if ( sorted_node_left( p_front ) == NULL ) {
                    p_front_entry = sorted_node_entry( p_front );
                    break;
                } else {
                    p_front = sorted_node_left( p_front );
                }
            }
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return p_front_entry;
}

void *sorted_list_back( sorted_list_t *const p_this ) {
    void *p_back_entry = NULL;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( sorted_list_size( p_this ) == 0 ) {
            // p_back_entry = NULL;
        } else {
            size_t depth;
            sorted_node_t *p_back = sorted_list_root( p_this );

            for ( depth = 0; depth < p_this->maximum_depth; ++depth ) {
                if ( sorted_node_right( p_back ) == NULL ) {
                    p_back_entry = sorted_node_entry( p_back );
                    break;
                } else {
                    p_back = sorted_node_right( p_back );
                }
            }
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return p_back_entry;
}
```

If the implementation includes the front and back pointers, both these implementations simplify greatly:

```
#define sorted_list_front( p_this )  ((p_this)->p_front + 0)
#define sorted_list_back( p_this )   ((p_this)->p_back + 0)
```

```
void *sorted_list_find( sorted_list_t *const p_this,
                        int            sought_key ) {
    void *p_sought_entry = NULL;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( sorted_list_empty( p_this ) ) {
            // p_sought_entry = NULL;
        } else {
            size_t depth;
            sorted_node_t *p_node = sorted_list_root( p_this );

            for ( depth = 0; depth < p_this->maximum_depth; ++depth ) {
                if ( sorted_node_key( p_node ) == sought_key ) {
                    p_sought_entry = sorted_node_entry( p_node );
                    break;
                } else if ( sought_key < sorted_node_key( p_node ) ) {
                    if ( sorted_node_left( p_node ) == NULL ) {
                        // p_sought_entry = NULL;
                        break;
                    } else {
                        p_node = sorted_node_left( p_node );
                    }
                } else {
                    assert( sorted_node_key( p_node ) < sought_key );

                    if ( sorted_node_right( p_node ) == NULL ) {
                        // p_sought_entry = NULL;
                        break;
                    } else {
                        p_node = sorted_node_right( p_node );
                    }
                }
            }
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return p_sought_entry;
}
```

```
static bool sorted_find_and_insert_only( sorted_list_t  *const p_this,
                                         stack_t        *p_stack,
                                         sorted_node_t  *p_node,
                                         sorted_node_t  *p_new_node,
                                         size_t          maximum_depth ) {
    bool success = false;
    size_t depth;

    stack_clear( p_stack );
    stack_push( p_stack, p_node );

    for ( depth = 0; depth < maximum_depth; ++depth ) {
        sorted_node_t *p_top = (sorted_node_t *)stack_top( p_stack );

        if ( sorted_node_key( p_new_node ) < sorted_node_key( p_top ) ) {
            if ( sorted_node_left( p_top ) == NULL ) {
                success = true;
                p_top->p_left = p_new_node;
                p_new_node->p_next = p_top;

                if ( p_top->p_previous == NULL ) {
                    p_this->p_front = p_new_node;
                } else {
                    p_new_node->p_previous = p_top->p_previous;
                    p_top->p_previous->p_next = p_new_node;
                }

                p_top->p_previous = p_new_node;

                break;
            } else {
                stack_push( p_stack, sorted_node_left( p_top ) );
            }
        } else if ( sorted_node_key( p_top ) < sorted_node_key( p_new_node ) ) {
            if ( sorted_node_right( p_top ) == NULL ) {
                success = true;
                p_top->p_right = p_new_node;
                p_new_node->p_previous = p_top;

                if ( p_top->p_next == NULL ) {
                    p_this->p_back = p_new_node;
                } else {
                    p_new_node->p_next = p_top->p_next;
                    p_top->p_next->p_previous = p_new_node;
                }

                p_top->p_next = p_new_node;
                break;
            } else {
                stack_push( p_stack, sorted_node_right( p_top ) );
            }
        } else {
            // success = false;
            break;
        }
    }

    return success;
}
```

These next five functions implement specific aspects of AVL rebalancing. There are four types of imbalances that need to be corrected, and these are implemented in the next four functions. The fifth function determines which type of imbalance has occurred and calls the appropriate function. This appropriately refactors the `sorted_list_insert(…)` function, in accordance with Rule 25 of the JPL coding standards:

> "Functions should be no longer than 60 lines of text and define no more than 6 parameters."

```
static void sorted_left_left_rebalance( stack_t        *p_stack,
                                        sorted_node_t **pp_root,
                                        sorted_node_t  *p_current,
                                        sorted_node_t  *p_child ) {
    p_current->p_left  = sorted_node_right( p_child );
    p_current->balance = AVL_BALANCED;

    p_child->p_right   = p_current;
    p_child->balance   = AVL_BALANCED;

    if ( stack_empty( p_stack ) ) {
        *pp_root = p_child;
    } else {
        sorted_node_t *p_top = (sorted_node_t *)stack_top( p_stack );

        if ( sorted_node_left( p_top ) == p_current ) {
            p_top->p_left  = p_child;
        } else {
            p_top->p_right = p_child;
        }
    }
}

static void sorted_left_right_rebalance( stack_t        *p_stack,
                                         sorted_node_t **pp_root,
                                         sorted_node_t  *p_current,
                                         sorted_node_t  *p_child ) {
    sorted_node_t *p_grandchild = p_child->p_right;
    p_child->p_right  = sorted_node_left(  p_grandchild );
    p_current->p_left = sorted_node_right( p_grandchild );
    p_grandchild->p_left  = p_child;
    p_grandchild->p_right = p_current;

    p_current->balance = sorted_node_is_left_heavy( p_grandchild ) ?
                                                AVL_RIGHT_HEAVY : AVL_BALANCED;
    p_child->balance   = sorted_node_is_right_heavy( p_grandchild ) ?
                                                AVL_LEFT_HEAVY  : AVL_BALANCED;
    p_grandchild->balance = AVL_BALANCED;

    if ( stack_empty( p_stack ) ) {
        *pp_root = p_grandchild;
    } else {
        sorted_node_t *p_top = (sorted_node_t *)stack_top( p_stack );

        if ( sorted_node_left( p_top ) == p_current ) {
            p_top->p_left  = p_grandchild;
        } else {
            p_top->p_right = p_grandchild;
        }
    }
}
```

```
static void sorted_right_right_rebalance( stack_t        *p_stack,
                                          sorted_node_t **pp_root,
                                          sorted_node_t  *p_current,
                                          sorted_node_t  *p_child ) {
    p_current->p_right = sorted_node_left( p_child );
    p_current->balance = AVL_BALANCED;

    p_child->p_left    = p_current;
    p_child->balance   = AVL_BALANCED;

    if ( stack_empty( p_stack ) ) {
        *pp_root = p_child;
    } else {
        sorted_node_t *p_top = (sorted_node_t *)stack_top( p_stack );

        if ( sorted_node_left( p_top ) == p_current ) {
            p_top->p_left  = p_child;
        } else {
            p_top->p_right = p_child;
        }
    }
}

static void sorted_right_left_rebalance( stack_t        *p_stack,
                                         sorted_node_t **pp_root,
                                         sorted_node_t  *p_current,
                                         sorted_node_t  *p_child ) {
    sorted_node_t *p_grandchild = sorted_node_left( p_child );
    p_child->p_left  =   sorted_node_right( p_grandchild );
    p_current->p_right = sorted_node_left(  p_grandchild );
    p_grandchild->p_right = p_child;
    p_grandchild->p_left  = p_current;

    p_current->balance = AVL_BALANCED;
    p_child->balance   = AVL_BALANCED;

    p_child->balance   = sorted_node_is_left_heavy( p_grandchild ) ?
                                                AVL_RIGHT_HEAVY : AVL_BALANCED;
    p_current->balance = sorted_node_is_right_heavy( p_grandchild ) ?
                                                AVL_LEFT_HEAVY  : AVL_BALANCED;
    p_grandchild->balance = AVL_BALANCED;

    if ( stack_empty( p_stack ) ) {
        *pp_root = p_grandchild;
    } else {
        sorted_node_t *p_top = (sorted_node_t *)stack_top( p_stack );

        if ( sorted_node_left( p_top ) == p_current ) {
            p_top->p_left  = p_grandchild;
        } else {
            p_top->p_right = p_grandchild;
        }
    }
}
```

```
static void sorted_list_insert_rebalance( stack_t        *p_stack,
                                          sorted_node_t **pp_root,
                                          sorted_node_t  *p_current,
                                          sorted_node_t  *p_child,
                                          int             new_key ) {
    if ( sorted_node_is_left_heavy( p_current ) ) {
        if ( sorted_node_right( p_current ) == p_child ) {
            p_current->balance = AVL_BALANCED;
        } else if ( new_key < sorted_node_key( p_child ) ) {
            sorted_left_left_rebalance( p_stack, pp_root, p_current, p_child );
        } else {
            sorted_left_right_rebalance( p_stack, pp_root, p_current, p_child );
        }
    } else {
        assert( sorted_node_is_right_heavy( p_current ) );

        if ( sorted_node_left( p_current ) == p_child ) {
            p_current->balance = AVL_BALANCED;
        } else if ( new_key < sorted_node_key( p_child ) ) {
            sorted_right_left_rebalance( p_stack, pp_root, p_current, p_child );
        } else {
            sorted_right_right_rebalance( p_stack, pp_root, p_current, p_child );
        }
    }
}
```

Inserting a new node into an AVL tree involves three cases:

1. the tree is full, in which case, reject the insertion,
2. the tree is empty, in which case the new entry is placed into a new node at the root, and
3. all other cases, where we only insert the node if there is not already another node in the tree.

In the third case, there are two steps:

1. placing the new node, and
2. performing any additional required corrections to the tree structure to ensure AVL balance.

In this last case, we need only traverse back until either

1. we have made a correction to restore AVL balance, or
2. the insertion has made an unbalanced node balanced.

In either case, it is guaranteed that all ancestors back to the root continue to be AVL balanced, and no further checks are necessary. There are four distinct corrections that can be made to bring the tree back into AVL balance.

```
bool sorted_list_insert( sorted_list_t *const p_this,
                         void          *p_new_entry,
                         int            new_key ) {
    assert( p_new_entry != NULL );
    bool success = false;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( sorted_list_full( p_this ) ) {
            // success = false;
        } else if ( sorted_list_empty( p_this ) ) {
            success = true;
            p_this->size = 1;

            p_this->p_root = p_this->p_node_pool;
            p_this->p_node_pool = sorted_node_next( p_this->p_node_pool );
            sorted_node_init( sorted_list_root( p_this ), p_new_entry, new_key );
            p_this->p_front = p_this->p_root;
            p_this->p_back  = p_this->p_root;
        } else {
            stack_t *p_ts = &( p_this->traversal_stack );

            sorted_node_t *p_new_node = p_this->p_node_pool;
            p_this->p_node_pool = sorted_node_next( p_this->p_node_pool );
            sorted_node_init( p_new_node, p_new_entry, new_key );

            success = sorted_find_and_insert_only( p_this, p_ts,
                                                   sorted_list_root( p_this ), p_new_node,
                                                   p_this->maximum_depth );

            if ( success ) {  // Traverse back to the root ensuring AVL balance
                ++( p_this->size );

                sorted_node_t *p_current = p_new_node;
                size_t depth;

                for ( depth = 0; !stack_empty( p_ts ) && depth < p_this->maximum_depth;
                                                         ++depth ) {
                    sorted_node_t *p_child = p_current;
                    p_current = (sorted_node_t *)stack_pop( p_ts );

                    if ( sorted_node_is_balanced( p_current ) ) {
                        if ( sorted_node_left( p_current ) == p_child ) {
                            p_current->balance = AVL_LEFT_HEAVY;
                        } else {
                            p_current->balance = AVL_RIGHT_HEAVY;
                        }
                    } else {
                        sorted_list_insert_rebalance( p_ts, &( p_this->p_root ),
                                                      p_current, p_child, new_key );
                        break;
                    }
                }
            } else {
                // Put the new node back onto the free list
                p_new_node->p_next = p_this->p_node_pool;
                p_this->p_node_pool = p_new_node;
            }
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return success;
}
void *sorted_list_erase( sorted_list_t *const p_this,
                         int            old_key ) {
```

```
    bool *p_erased_entry = NULL;

    binary_semaphore_wait( &( p_this->mutex ) );
        if ( sorted_list_empty( p_this ) ) {
            // p_erased_entry = NULL;
        } else {
            // How would you implement this?
        }
    binary_semaphore_post( &( p_this->mutex ) );

    return p_erased_entry;
}

void sorted_list_clear( sorted_list_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) );
        if ( !sorted_list_empty( p_this ) ) {
            sorted_node_t *p_last_popped = NULL;

            p_this->p_back->p_next = p_this->p_node_pool;
            p_this->p_node_pool = p_this->p_front;

            p_this->p_root = NULL;
            p_this->size = 0;
        }
    binary_semaphore_post( &( p_this->mutex ) );
}
```

D.9 An iterator for a sorted list

Suppose it is necessary to iterate through all of the entries of a linked list. It would be possible to simply access the `p_head` field and step through the entries, but what if the data structure is changed and, for example, the node you are currently referencing is no longer a component of the linked list? For this, we will introduce an iterator:

```
sorted_iterator_t itr;
sorted_iterator_init( &itr, &some_single_list );
some_type *p_next_entry;
bool valid;

while ( p_next_entry = (some_type *)single_iterator_next( &itr, &valid ) ) {
        // Work with 'p_next_entry'...
}

if ( !valid ) {
    // We did not iterate through all of the entries of the list
    //  - something changed in the list prior to getting to the end
}
```

There are two possible ends to this loop, whereby either

3. all entries in the linked list have been iterated through, or
4. prior to iterating through all the entries, a change was made to the linked list either by this task or another.

In the second case, the version stored is no longer equal to the version of the linked list.

```
// sorted_iterator.h
#ifndef CA_UWATERLOO_DWHARDER_SORTED_ITERATOR
#define CA_UWATERLOO_DWHARDER_SORTED_ITERATOR

#include "sorted_list.h"

typedef struct {
    sorted_list_t *p_list;
    sorted_node_t *p_current_node;
    bool before_begin;
    bool after_end;
    uint64_t version;
} sorted_iterator_t;

#define sorted_iterator_is_finished( p_this ) ((p_this)->finished && true)
#define sorted_iterator_is_valid( p_this )                                    \
                                        ((p_this)->version == (p_this)->p_list->version)

void sorted_iterator_begin( sorted_iterator_t *p_this,
                            sorted_list_t     *p_new_list );
void sorted_iterator_end( sorted_iterator_t *p_this,
                          sorted_list_t     *p_new_list );

bool sorted_iterator_is_after_end(    sorted_iterator_t *p_this, bool *valid );
bool sorted_iterator_is_before_begin( sorted_iterator_t *p_this, bool *valid );

void *sorted_iterator_previous( sorted_iterator_t *p_this, bool *valid );
void *sorted_iterator_next(     sorted_iterator_t *p_this, bool *valid );

#endif
```

```
// sorted_iterator.c
#include "sorted_iterator.h"
void sorted_iterator_begin( sorted_iterator_t *p_this,
                            sorted_list_t     *p_new_list ) {
    binary_semaphore_wait( &( p_new_list->mutex ) );
        p_this->p_list         = p_new_list;
        p_this->p_current_node = p_new_list->p_front;
        p_this->version        = p_new_list->version;
        p_this->before_begin   = false;
        p_this->after_end      = sorted_list_empty( p_new_list );
    binary_semaphore_post( &( p_new_list->mutex ) );
}

void sorted_iterator_end( sorted_iterator_t *p_this,
                          sorted_list_t     *p_new_list ) {
    binary_semaphore_wait( &( p_new_list->mutex ) );
        p_this->p_list         = p_new_list;
        p_this->p_current_node = p_new_list->p_back;
        p_this->version        = p_new_list->version;
        p_this->before_begin   = sorted_list_empty( p_new_list );
        p_this->after_end      = false;
    binary_semaphore_post( &( p_new_list->mutex ) );
}

bool sorted_iterator_after_end( sorted_iterator_t *p_this, bool *valid ) {
    bool is_after_end;

    binary_semaphore_wait( &( p_new_list->mutex ) );
        if ( p_this->version != p_this->p_list->version ) {
            *valid = false;
        } else {
            *valid = true;
        }

        is_after_end = p_this->after_end;
    binary_semaphore_post( &( p_new_list->mutex ) );

    return is_after_end;
}

bool sorted_iterator_before_begin( sorted_iterator_t *p_this, bool *valid ) {
    bool is_before_begin;

    binary_semaphore_wait( &( p_new_list->mutex ) );
        if ( p_this->version != p_this->p_list->version ) {
            *valid = false;
        } else {
            *valid = true;
        }

        is_before_begin = p_this->before_begin;
    binary_semaphore_post( &( p_new_list->mutex ) );

    return is_before_begin;
}
```

```
void *sorted_iterator_previous( sorted_iterator_t *p_this, bool *valid ) {
    void *p_current_entry = NULL;

    binary_semaphore_wait( &( p_this->p_list->mutex ) );
        if ( p_this->version != p_this->p_list->version ) {
            *valid = false;
            // p_current_entry = NULL;
        } else {
            *valid = true;

            if ( p_this->p_current_node == NULL ) {
                if ( p_this->after_end ) {
                    p_this->p_list->p_back;
                    p_this->after_end = false;
                    p_this->before_begin = sorted_list_empty( p_this->p_list );
                }
            } else {
                p_current_entry = p_this->p_current_node->p_entry;
                p_this->p_current_node = p_this->p_current_node->p_previous;

                if ( p_this->p_current_node == NULL ) {
                    p_this->before_begin = true;
                }
            }
        }
    binary_semaphore_post( &( p_this->p_list->mutex ) );

    return p_current_entry;
}

void *sorted_iterator_next( sorted_iterator_t *p_this, bool *valid ) {
    void *p_current_entry = NULL;

    binary_semaphore_wait( &( p_this->p_list->mutex ) );
        if ( p_this->version != p_this->p_list->version ) {
            *valid = false;
            // p_current_entry = NULL;
        } else {
            *valid = true;

            if ( p_this->p_current_node == NULL ) {
                if ( p_this->before_begin ) {
                    p_this->p_list->p_front;
                    p_this->before_begin = false;
                    p_this->after_end = sorted_list_empty( p_this->p_list );
                }
            } else {
                p_current_entry = p_this->p_current_node->p_entry;
                p_this->p_current_node = p_this->p_current_node->p_next;

                if ( p_this->p_current_node == NULL ) {
                    p_this->after_end = true;
                }
            }
        }
    binary_semaphore_post( &( p_this->p_list->mutex ) );

    return p_current_entry;
}
```

D.10 B†-tree

The on-line version has an implementation of a B tree.

D.11 Sorting algorithms

As recommended by Nigel Jones, we will look at implementations of insertion, shell and heap sort. Each of these algorithms operate in place with only $\Theta(1)$ memory required, and insertion sort is stable. The run time of these three algorithms, as implemented here, are $O(n^2)$, $O(n^{4/3})$ and $\Theta(n \ln(n))$, respectively; however, insertion sort is optimal for small lists or when the list is already almost sorted, shell sort is reasonably fast for lists of size up to one million, and heap sort is fast for large arrays. While an embedded system is unlikely to deal with arrays of size one million, it is conceivable that a real-time non-embedded system does.

To put the performance into perspective, we may understand that using insertion sort is only beneficial if either the array is nearly sorted or very small. After this, shell sort is comparable to heap sort for arrays of up to size $n = 1\,000\,000$, as is shown in Figure D-1 with graphs of n^2, $n^{4/3}$ and $n \ln(n)$. You may note that while $n \ln(n) = o(n^{4/3})$, it does not take a significant difference in coefficients to have, for example, $n^{4/3} < 8\, n \ln(n)$, on the given range. After we present all three algorithms, we will then discuss the results of performance tests on these algorithms.

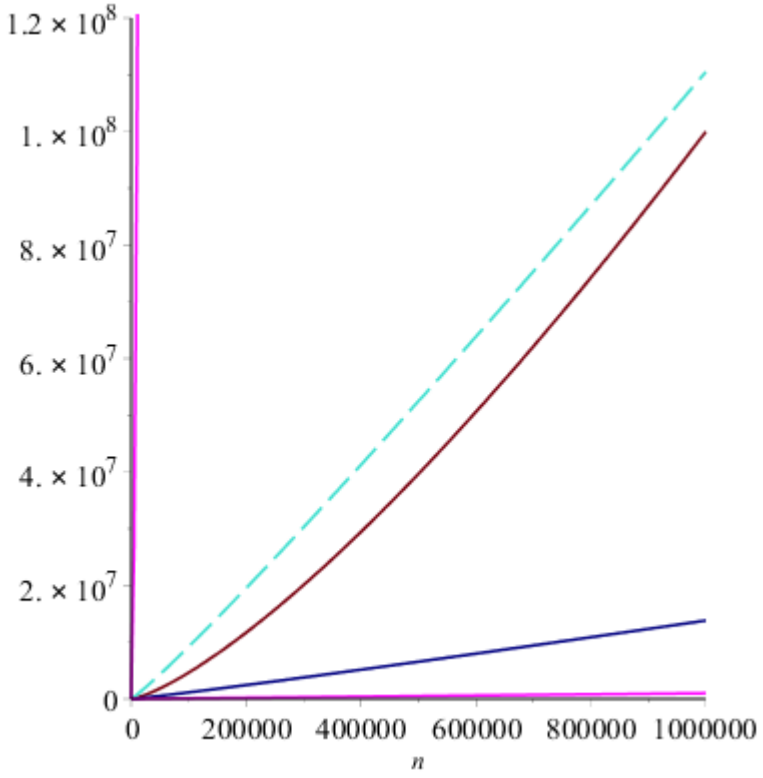


Figure D-1. A plot of $n^{4/3}$ in blue versus $n \ln(n)$ in red, with $8\, n \ln(n)$ shown as a dashed turquoise line and n and n^2 in magenta.

Now, in each case, the C programming language does not allow for generic sorting algorithms, and consequently it is necessary to use an appropriate mechanism to allow these sorting algorithms to be written once, after which they may be used to sort objects of arbitrary types. To do this, each algorithm is written in a macro which takes as its argument the desired type. For example, the macro

```
INSERTION_SORT_PRIMITIVE_DECLARE( int )
```

declares the function

```
void insertion_sort_int( int *array, size_t const n );
```

and we may define the function using the macro

```
INSERTION_SORT_PRIMITIVE_DEFINE( int )
```

The arguments are the array that is to be sorted and its size. With any of these algorithms, if you wish to sort a sub-range, say from m to $n - 1$, the appropriate call would be of the form

```
uint32_t array = (uint32_t *)malloc( N*sizeof( uint32_t ) );
// populate array with entries
insertion_sort_uint32_t( array + m, n - m );
```

In each case, however, it may be more desirable to implement these algorithms to be able to sort aggregate data structures, and therefore we must resort to have an algorithm that sorts an array of pointers to such objects. For example, it may be desirable to sort an array of thread control blocks based on their priority. For this, we introduce a second function that sorts an array of pointers to records. An arbitrary pointer is represented by `void *`, which has nothing to do with the `void` return type. Because the array will now be an array of pointers, the type must therefore be `void **`—the address of an array that stores addresses of other records or, in other words, a pointer to pointers:

```
void insertion_sort_p_tcb_t( void **array, size_t const n,
                             bool (*is_in_order)( void *, void * ) );
```

The fourth formal parameter will be a pointer to a function that returns true or false depending on whether or not the two arguments are in order, respectively. We will have to use casting for such a function, as the signature requires that the arguments are void pointers, but inside the function, we will have to indicate what we are expecting. For example, if we wished to write a function to compare two TCBs based on the field `priority`, we would have to implement a function

```
bool is_in_order_p_tcb_t( void *p_1, void *p_2 ) {
    tcb_t *p_tcb1 = (tcb_t *) p_1;
    tcb_t *p_tcb2 = (tcb_t *) p_2;

    return p_tcb1->priority <= p_tcb2->priority;
}
```

With insertion sort, this may be sufficient, but in order to create a stable shell or heap sort, it is also necessary to have a field reflecting the initial order of the entries. If such a field was `order`, a lexicographical ordering could be generated as follows:

```
bool is_in_order_p_tcb_t(void *p_1, void *p_2 ) {
    tcb_t *p_tcb1 = (tcb_t *) p_1;
    tcb_t *p_tcb2 = (tcb_t *) p_2;

    // A lexicographical ordering is useless if both fields are simultaneously equal
    assert( (p_tcb1->priority != p_tcb2->priority) || (p_tcb1->order != p_tcb2->order) );

    return p_tcb1->priority < p_tcb2->priority || (
        (p_tcb1->priority == p_tcb2->priority) && (p_tcb1->order < p_tcb2->order)
    );
}
```

A word of warning: Despite what is written by amateurs on the Internet, do not under any circumstances use bubble sort. Even in implementations that both bubble up and sink down and flagging intervals at either end that are demonstrably sorted, bubble sort still requires approximately 1.5 comparisons for each comparison of insertion sort.

D.11.1 Insertion sort

The insertion sort is the fastest $O(n^2)$ algorithm, running in $\Theta(v)$ time where v is the number of inversions. An inversion is the number of pairs of numbers in the list being sorted that are out of place relative to each other. In the worst case, when the array is reverse sorted, the number of inversions is $\binom{n}{2} = \frac{n(n-1)}{2}$ and for a randomly generated list, it will be approximately half this number, but for an *almost* sorted list, insertion sort can be very fast. It is also a *stable* algorithm, in that items that are equal remain in the same order they appeared in in the original list.

```
// insertion_sort.h
#ifndef CA_UWATERLOO_DWHARDER_INSERTION_SORT
#define CA_UWATERLOO_DWHARDER_INSERTION_SORT
#define INSERTION_SORT_PRIMITIVE_DECLARE( Type )              \
void insertion_sort_##Type( Type *p_array, size_t const n );

INSERTION_SORT_PRIMITIVE_DECLARE( uint32_t )
INSERTION_SORT_PRIMITIVE_DECLARE( float )

void insertion_sort_pointer( void **pp_array, size_t const n,
                             bool (*is_in_order)( void *, void * ) );
#endif

// insertion_sort.c
#include "insertion_sort.h"

#define INSERTION_SORT_PRIMITIVE_DEFINE( Type )               \
void insertion_sort_##Type( Type *p_array, size_t const n ) { \
    Type *const p_end = p_array + n;                          \
    Type *p_prev, *p_curr, *p_next, next_entry;               \
                                                              \
    for ( p_next = p_array + 1; p_next < p_end; ++p_next ) {  \
        next_entry = *p_next;                                 \
                                                              \
        for ( p_prev = p_next - 1, p_curr = p_next;           \
              p_prev >= p_array && (*p_prev > next_entry);    \
            --p_prev,            --p_curr                     \
        ) {                                                   \
            *p_curr = *p_prev;                                \
        }                                                     \
                                                              \
        *p_curr = next_entry;                                 \
    }                                                         \
}

INSERTION_SORT_PRIMITIVE_DEFINE( uint32_t )
INSERTION_SORT_PRIMITIVE_DEFINE( float )

void insertion_sort_pointer( void **pp_array, size_t const n,
                             bool (*is_in_order)( void *, void * ) ) {
    void **const pp_end = pp_array + n;
    void **pp_prev, **pp_curr, **pp_next, *p_next_entry;

    for ( pp_next = pp_array + 1; pp_next < pp_end; ++pp_next ) {
        p_next_entry = *pp_next;

        for ( pp_prev = pp_next - 1, pp_curr = pp_next;
              pp_prev >= pp_array && !is_in_order( *pp_prev, p_next_entry );
            --pp_prev,             --pp_curr
        ) {
            *pp_curr = *pp_prev;
        }

        *pp_curr = p_next_entry;
    }
}
```

D.11.2 Shell sort

The first sorting algorithm to run faster than $O(n^2)$ was shell sort, first introduced by Donald Shell. He proposed performing insertion sort not on every element initially, but rather on elements that are p steps apart. Having completed this, successively choose smaller step sizes until the step size is one, in which case, the algorithm reverts to insertion sort. The choice sequence of gaps is highly controversial and the user is welcome to implement this algorithm using a different sequence of gaps, but we use a list recommended by Sedgewick[48] defined by $4^k + 3 \cdot 2^{k-1} + 1$ for positive integral values of k forming a decreasing sequence of gap sizes ending in

…, 65921, 16577, 4193, 1073, 281, 77, 32, 8,

which results in a run time that is $O(n^{4/3})$. There are numerous other possible sequences for the gap size, but after repeated iterations, this sequence provided appears to be more than adequate. In the last step, we call insertion sort, as the insertion sort algorithm is significantly more clean than shell sort when the gap size is equal to one, and therefore it is more efficient to make one additional call rather than adding additional code to check for and deal with this special case.

[48] See Sedgewick, Robert, *Algorithms in C*, 3rd ed., Addison-Wesley, 1998, pp. 273–281.

```
// shell_sort.h
#ifndef CA_UWATERLOO_DWHARDER_SHELL_SORT
#define CA_UWATERLOO_DWHARDER_SHELL_SORT

#define SHELL_SORT_PRIMITIVE_DECLARE( Type )                                \
void shell_sort_##Type( Type *p_array, size_t const n );

INSERTION_SORT_PRIMITIVE_DECLARE( uint32_t )
INSERTION_SORT_PRIMITIVE_DECLARE( float )

void shell_sort_pointer( void **pp_array, size_t const n,
                         bool (*is_in_order)( void *, void * ) );
#endif

// shell_sort.c
#include "shell_sort.h"
#include "insertion_sort.h"

#define SHELL_SORT_PRIMITIVE_DEFINE( Type )                                 \
void shell_sort_##Type( Type *p_array, size_t const n ) {                   \
    size_t pow4, pow2, gap;                                                 \
    Type *const p_end = p_array + n;                                        \
    Type *p_start, *p_prev, *p_curr, *p_next, next_entry;                   \
                                                                            \
    if ( n > 8 ) {                                                          \
        for ( pow4 = 4, pow2 = 1;                                           \
              4*pow4 + 6*pow2 + 1 < n;                                      \
              pow4 *= 4, pow2 *= 2                                          \
        );                                                                  \
                                                                            \
        for ( ; pow2 > 0; pow2 >>= 1, pow4 >>= 2 ) {                        \
            gap = pow4 + 3*pow2 + 1;                                        \
                                                                            \
            for ( p_start = p_array; p_start < p_array + gap; ++p_start ) { \
                for ( p_next = p_start + gap;                               \
                      p_next < p_end; p_next += gap ) {                     \
                    next_entry = *p_next;                                   \
                                                                            \
                    for ( p_prev = p_next - gap, p_curr = p_next;           \
                          p_prev >= p_array && (*p_prev > next_entry);      \
                          p_prev -= gap,         p_curr -= gap              \
                    ) {                                                     \
                        *p_curr = *p_prev;                                  \
                    }                                                       \
                                                                            \
                    *p_curr = next_entry;                                   \
                }                                                           \
            }                                                               \
        }                                                                   \
    }                                                                       \
                                                                            \
    insertion_sort_##Type( p_array, n );                                    \
}

SHELL_SORT_PRIMITIVE_DEFINE( uint32_t )
SHELL_SORT_PRIMITIVE_DEFINE( float )
```

```
// An implementation for sorting an array of pointers
void shell_sort_pointer( void **pp_array, size_t const n,
                         bool (*is_in_order)( void *, void * ) ) {
    size_t pow4, pow2, gap;
    void **const pp_end = pp_array + n;
    void **pp_start, **pp_prev, **pp_curr, **pp_next, *p_next_entry;

    if ( n > 8 ) {
        //                                        k      k-1
        // Find the maximum gap in the sequence 4  + 3 2    + 1 < n,
        // with k = 1, 2, ...
        for ( pow4 = 4, pow2 = 1;
              4*pow4 + 6*pow2 + 1 < n;
              pow4 *= 4, pow2 *= 2
        );

        // For each successively smaller gap, apply the insertion sort algorithm
        // on entries 0, m, 2m, ...; 1, m+1, 2m+1, ..., 2, m+2, 2m+2, ..., up to
        // m-1, 2m-1, 3m-1, ... . Then repeat with the next smaller gap size.
        for ( ; pow2 > 0; pow2 >>= 1, pow4 >>= 2 ) {
            gap = pow4 + 3*pow2 + 1;

            for ( pp_start = pp_array; pp_start < pp_array + gap; ++pp_start ) {
                for ( pp_next = pp_start + gap; pp_next < pp_end; pp_next += gap ) {
                    p_next_entry = *pp_next;

                    for ( pp_prev = pp_next - gap, pp_curr = pp_next;
                          pp_prev >= pp_array && !is_in_order( *pp_prev, p_next_entry );
                          pp_prev -= gap,          pp_curr -= gap
                    ) {
                        *pp_curr = *pp_prev;
                    }

                    *pp_curr = p_next_entry;
                }
            }
        }
    }

    // At the end, apply insertion sort, as the maximum number of inversions is 8n
    insertion_sort_pointer( pp_array, n, is_in_order );
}
```

D.11.3 Heap sort

We will conclude with heap sort, an algorithm that converts the unsorted list into a max-heap in linear time, and repetitively pops the top element *n*, placing the popped element into the blank left at the end of the array. The footprint of this sorting routine is on the order of the shell sort routine once the inclusion of insertion sort is taken into account.

```
// heap_sort.h
#ifndef CA_UWATERLOO_DWHARDER_HEAP_SORT
#define CA_UWATERLOO_DWHARDER_HEAP_SORT

#define HEAP_SORT_PRIMITIVE_DECLARE( Type )                                 \
void heap_sort_##Type( Type *p_array, size_t const n );

HEAP_SORT_PRIMITIVE_DECLARE( uint32_t )
HEAP_SORT_PRIMITIVE_DECLARE( float )

void heap_sort_pointer( void **pp_array, size_t const n,
                        bool (*is_in_order)( void *, void * ) );
#endif
```

```
// heap_sort.c
#include "heap_sort.h"

#define HEAP_SORT_PRIMITIVE_DEFINE( Type )                                  \
static void percolate_down_##Type( Type  const entry,  size_t posn,         \
                                   Type *const p_heap, size_t const n );    \
                                                                            \
void heap_sort_##Type( Type *p_array, size_t const n ) {                    \
    size_t i;                                                               \
    Type *const p_heap = p_array - 1;                                       \
    Type entry;                                                             \
                                                                            \
    for ( i = n/2; i >= 1; --i ) {                                          \
        percolate_down_##Type( p_heap[i], i, p_heap, n );                   \
    }                                                                       \
                                                                            \
    for ( i = n; i >= 2; --i ) {                                            \
        entry = p_heap[i];                                                  \
        p_heap[i] = p_heap[1];                                              \
                                                                            \
        percolate_down_##Type( entry, 1, p_heap, i - 1 );                   \
    }                                                                       \
}                                                                           \
                                                                            \
static void percolate_down_##Type( Type  const entry,  size_t posn,         \
                                   Type *const p_heap, size_t const n ) { \
    bool found = false;                                                     \
    size_t left  = posn << 1;                                               \
    size_t right = left|1;                                                  \
                                                                            \
    while ( !found && (left < n) ) {                                        \
        if ( p_heap[left] > p_heap[right] && p_heap[left] > entry ) {       \
            p_heap[posn] = p_heap[left];                                    \
            posn = left;                                                    \
            left  = posn << 1;                                              \
            right = left|1;                                                 \
        } else if ( p_heap[right] > entry ) {                               \
            p_heap[posn] = p_heap[right];                                   \
            posn = right;                                                   \
            left  = posn << 1;                                              \
            right = left|1;                                                 \
        } else {                                                            \
            p_heap[posn] = entry;                                           \
            found = true;                                                   \
        }                                                                   \
    }                                                                       \
                                                                            \
    if ( !found ) {                                                         \
        if ( (left == n) && (p_heap[left] > entry) ) {                      \
            p_heap[posn] = p_heap[left];                                    \
            p_heap[left] = entry;                                           \
        } else {                                                            \
            p_heap[posn] = entry;                                           \
        }                                                                   \
    }                                                                       \
}

HEAP_SORT_PRIMITIVE_DEFINE( uint32_t )
HEAP_SORT_PRIMITIVE_DEFINE( float )
```

```
static void percolate_down_pointer( void  *const p_entry, size_t posn,
                                    void **const pp_heap, size_t const n,
                                    bool (*is_in_order)( void *, void * ) );

void heap_sort_pointer( void **pp_array, size_t const n,
                        bool (*is_in_order)( void *, void * ) ) {
    size_t i;
    void **const pp_heap = pp_array - 1;
    void *p_entry;

    for ( i = n/2; i >= 1; --i ) {
        percolate_down_pointer( pp_heap[i], i, pp_heap, n, is_in_order );
    }

    for ( i = n; i >= 2; --i ) {
        p_entry = pp_heap[i];
        pp_heap[i] = pp_heap[1];

        percolate_down_pointer( p_entry, 1, pp_heap, i - 1, is_in_order );
    }
}

static void percolate_down_pointer( void  *const p_entry, size_t posn,
                                    void **const pp_heap, size_t const n,
                                    bool (*is_in_order)( void *, void * ) ) {
    bool found = false;
    size_t left  = posn << 1;  // = 2*posn
    size_t right = left|1;     // = left + 1

    while ( !found && (left < n) ) {
        if ( !is_in_order( pp_heap[left], pp_heap[right] ) &&
             !is_in_order( pp_heap[left], p_entry )
        ) {
            pp_heap[posn] = pp_heap[left];
            posn = left;
            left  = posn << 1;   // = 2*posn
            right = left|1;      // = left + 1
        } else if ( !is_in_order( pp_heap[right], p_entry ) ) {
            pp_heap[posn] = pp_heap[right];
            posn = right;
            left  = posn << 1;   // = 2*posn
            right = left|1;      // = left + 1
        } else {
            pp_heap[posn] = p_entry;
            found = true;
        }
    }

    if ( !found ) {
        if ( (left == n) && !is_in_order( pp_heap[left], p_entry ) ) {
            pp_heap[posn] = pp_heap[left];
            pp_heap[left] = p_entry;
        } else {
            pp_heap[posn] = p_entry;
        }
    }
}
```

D.11.4 Analysis

Determining when to use the various sorting algorithms depends on the characteristics of the array being sorted. This depends primarily on the number of inversions in the array, and secondarily on the size of the array. The worst case number of inversions occurs when the list is reverse sorted, in which case, there are $\binom{n}{2} = \frac{n(n-1)}{2}$ inversions. A list that is nearly sorted, however, may contain significantly fewer inversions. Thus, we have the following observations:

1. If the number of inversions is O(n), insertion sort is the fastest.
2. If the number of inversions is ω(n),
    a. if the size of the array is small (n < 100), insertion sort may still be useful,
    b. otherwise, shell sort or heap sort are likely more appropriate.

The run time of insertion sort is Θ(d + n), so if the number of inversions is relatively low (O(n), meaning that the array is essentially sorted, the run-time of insertion sort is Θ(n). On the other hand, the run time of shell sort, given our sequence of gaps, is Ω(n $\log_4(n)$) and heap sort is Θ(n ln(n)), and while this implementation of shell sort is known to be O($n^{4/3}$), the coefficient is sufficiently small relative to heap sort that, together with its more compact footprint, shell sort is often more appropriate for an embedded system.

In order to demonstrate the relationship between the run time of these sorting algorithms, Figure D-2 shows the run times for the three algorithms for sorting a list of size n that has between n and 2n inversions. The growth of insertion sort is linear with a least squares best fitting exponent of O($n^{1.045}$) while the growth of the shell sort algorithm is O($n^{1.336}$). The n-ln-n heap sort algorithm improves on shell sort once the size of the array is beyond half a million, but this is not likely to be common in embedded systems.

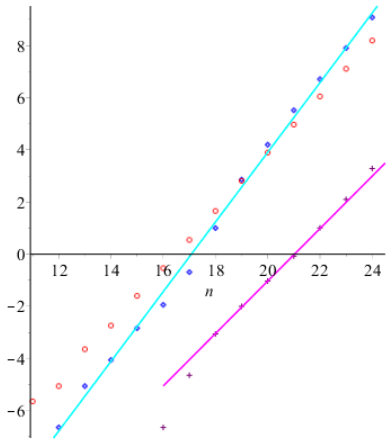


Figure D-2. The $\log_2$-$\log_2$ plot of array size versus the run time for insertion sort (purple crosses), shell sort (blue diamonds) and heap sort (red circles) for arrays with between n and 2n inversions.

For smaller arrays, the appropriate algorithm depends on the sortedness of the array, although for smaller arrays, the overhead of heap sort makes it prohibitively expensive. If the array is nearly sorted, as suggested above, the run time of insertion sort makes it the algorithm of choice. If the array is randomly generated, on average, it will contain approximately $\frac{1}{2}\binom{n}{2} \sim \frac{1}{4}n^2$ inversions, as is shown in the left graph of Figure D-3. In this case, insertion sort is still superior up to arrays of size 100; however, if the array is reverse sorted (with $\binom{n}{2}$ inversions), even for the smallest arrays, shell sort is superior as shown in the right graph Figure D-3.

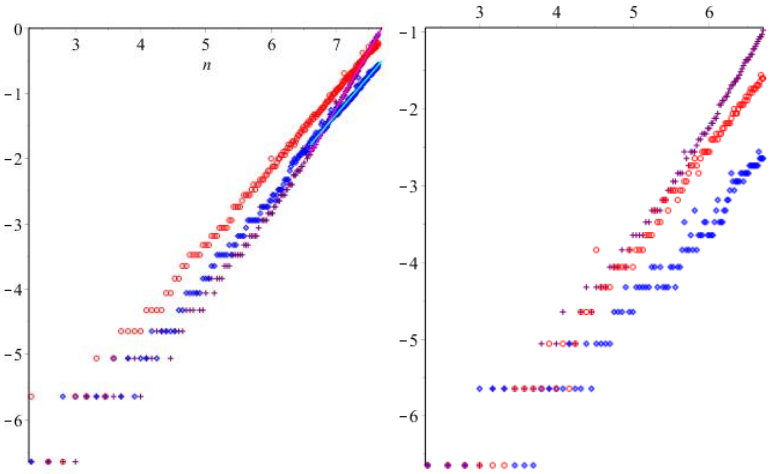


Figure D-3. Timing of numerous test runs for insertion sort (purple crosses), shell sort (blue diamonds) and heap sort (red circles) when the array is nearly sorted, randomly ordered and reverse sorted, respectively.

Consequently, from a real-time point-of-view, we may deduce that

1. if the array is guaranteed to be almost sorted (with O(n) inversions), use insertion sort,
2. otherwise, do not use insertion sort.

To determine the more appropriate sorting algorithm for larger arrays, we will consider sorting essentially sorted lists, random lists, and reverse sorted lists. These run times are plotted on log-log plots in Figure D-4 with best fitting lines passing through those algorithms with expected polynomial behavior.

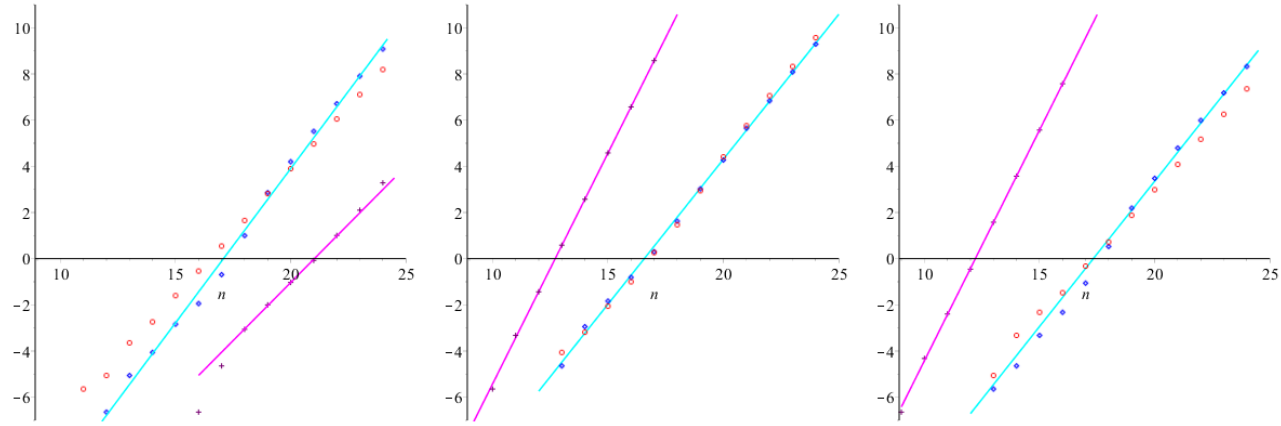


Figure D-4. Timing of numerous test runs for insertion sort (purple crosses), shell sort (blue diamonds) and heap sort (red circles) when the array is nearly sorted, random and reverse sorted, respectively.

We note that from the best-fitting lines of the points for insertion sort that the run times are O($n^{1.045}$), O($n^{2.000}$) and O($n^{1.998}$) , respectively. For shell sort, the slopes indicate that the run times are O($n^{1.336}$), O($n^{1.256}$) and O($n^{1.258}$), respectively, and again, all are in agreement with the theoretical worst-case run time of O($n^{4/3}$). Heap sort, however, while running in n-ln-n time, does not do as well as one may expect even with an array as large as 2^{24} or sixteen million, where heap sort is only twice as fast as shell sort. Thus, while heap sort appears to have many of the characteristics one would desire in a sorting algorithm (running in place with a $\Theta(n \ln(n))$ run time), the additional overhead of executing the algorithm make it unlikely to be a reasonable candidate for a real-time or embedded system where shell sort is almost certain to be more than sufficient.

Appendix E Synchronization data structures

p_this appendix will simply summarize the various synchronization data structures constructed from binary semaphores, including:

7. counting semaphores,
8. turnstiles,
9. group rendezvous,
10. light switches, and
11. events.

E.1 Counting semaphores

A counting semaphore has three interface functions:

```
void counting_semaphore_init( counting_semaphore_t *const p_this, unsigned int n );
```
Initialize the counting semaphore with *n* tokens.

```
void counting_semaphore_wait( counting_semaphore_t *const p_this );
```
Decrement the number of tokens available. If the number of available tokens is now strictly negative, block the calling task or thread, otherwise, let it the calling task or thread continue executing.

```
void counting_semaphore_post( counting_semaphore_t *const p_this );
```
Increment the number of tokens available. If the number of tokens available is 0 or less, then wake up one of the tasks currently waiting on the semaphore.

A counting semaphore can be implemented with two binary semaphores and a count.

```
typedef struct {
    size_t tokens;
    binary_semaphore_t mutex;
    binary_semaphore_t waiting_tasks;
} counting_semaphore_t;

void counting_semaphore_init( counting_semaphore_t *const p_this, size_t init ) {
    p_this->tokens = init;
    binary_semaphore_init( &( p_this->mutex ), 1 );
    binary_semaphore_init( &( p_this->waiting_tasks ), 0 );
}

void counting_semaphore_wait( counting_semaphore_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        --( p_this->tokens );

        if ( p_this->tokens < 0 ) {
            binary_semaphore_post( &( p_this->mutex ) );
            binary_semaphore_wait( &( p_this->waiting_tasks ) );
        }
    } binary_semaphore_post( &( p_this->mutex ) );
}

void counting_semaphore_post( counting_semaphore_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) );
    ++( p_this->tokens );

    if ( p_this->tokens <= 0 ) {
        counting_semaphore_post( &( p_this->waiting_tasks ) );
    } else {
        binary_semaphore_post( &( p_this->mutex ) );
    }
}
```

E.2 Turnstile

The interface of the `turnstile_t` data structures includes two defined constants:

```
#define TURNSTILE_LOCKED   true
#define TURNSTILE_UNLOCKED false
```

and four interface functions:

```
void turnstile_init( turnstile_t *const p_this, bool locked );
```
Initialize the turnstile.

```
void turnstile_unlock( turnstile_t *const p_this );
```
Unlock a locked turnstile—the behavior is undefined if the turnstile is unlocked.

```
void turnstile_lock( turnstile_t *const p_this );
```
Lock an unlocked turnstile—the behavior is undefined if the turnstile is locked.

```
void turnstile_pass( turnstile_t *const p_this );
```
If the turnstile is unlocked, p_this function call immediately returns,
otherwise, the calling task or thread is blocked until the turnstile is unlocked.

The turnstile can be entirely implemented with binary semaphores.

```
typedef struct {
    binary_semaphore_t turnstile;
} turnstile_t;

void turnstile_init( turnstile_t *const p_this, bool locked ) {
    binary_semaphore_init( &( p_this->turnstile ), locked ? 0 : 1 );
}

void turnstile_unlock( turnstile_t *const p_this ) {
    binary_semaphore_post( &( p_this->turnstile ) );
}

void turnstile_lock( turnstile_t *const p_this ) {
    binary_semaphore_wait( &( p_this->turnstile ) );
}

void turnstile_pass( turnstile_t *const p_this ) {
    binary_semaphore_wait( &( p_this->turnstile ) );
    binary_semaphore_post( &( p_this->turnstile ) );
}
```

E.3 Group rendezvous

The interface of the `rendezvous_t` data structures includes four interface functions:

```
void rendezvous_init( rendezvous_t *const p_this, size_t n )
```
Initialize the rendezvous for *n* tasks or threads.

```
void rendezvous_wait( rendezvous_t *const p_this );
```
Wait on the rendezvous point, and if p_this is the n^{th} task or thread, it and the previous $n - 1$ tasks or threads pass through.

The group rendezvous can be most easily implemented using a binary semaphore and two turnstiles:

```
typedef struct {
    size_t capacity;
    size_t waiting;
    binary_semaphore_t mutex;
    turnstile_t enter;
    turnstile_t exit;
} rendezvous_t;

void rendezvous_init( rendezvous_t *const p_this, size_t n ) {
    p_this->capacity = n;
    p_this->waiting = 0;

    binary_semaphore_init( &( p_this->mutex ), 1 );
    turnstile_init( &( p_this->enter ), TURNSTILE_LOCKED );
    turnstile_init( &( p_this->exit ),  TURNSTILE_UNLOCKED );
}

void rendezvous_wait( rendezvous_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        ++( p_this->waiting );

        if ( p_this->waiting == p_this->capacity ) {
            turnstile_lock(   &( p_this->exit  ) );
            turnstile_unlock( &( p_this->enter ) );
        }
    } binary_semaphore_post( &( p_this->mutex ) );

    turnstile_pass( &( p_this->enter ) );

    binary_semaphore_wait( &( p_this->mutex ) ); {
        --( p_this->waiting );

        if ( p_this->waiting == 0 ) {
            turnstile_lock(   &( p_this->enter ) );
            turnstile_unlock( &( p_this->exit  ) );
        }
    } binary_semaphore_post( &( p_this->mutex ) );

    turnstile_pass( &( p_this->exit ) );
}
```

E.4 Light switch

The interface of the `rendezvous_t` data structures includes four interface functions:

`void lightswitch_init( lightswitch_t *const p_this, binary_semaphore_t *s )`
: Initialize the light switch.

`void lightswitch_wait( lightswitch_t *const p_this );`
: Wait on the light switch, and if the light switch is *on*, continue; otherwise, try to acquire the binary semaphore *s* and, when it is acquired, turn the light switch on.

`void lightswitch_post( lightswitch_t *const p_this );`
: Post to the light switch, and if p_this is the last task or thread waiting on p_this light switch, turn off the light and release the binary semaphore *s*.

The light switch can be most easily implemented using a binary semaphore:

```
typedef struct {
    size_t population;
    binary_semaphore_t mutex;
    binary_semaphore_t *p_access_control;
} lightswitch_t;

void lightswitch_init( lightswitch_t *const p_this, binary_semaphore_t *s ) {
    p_this->population = 0;
    binary_semaphore_init( &( p_this->mutex ), 1 );
    p_this->p_access_control = s;
}

void lightswitch_wait( lightswitch_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        ++( p_this->population );

        if ( p_this->population == 1 ) {
            binary_semaphore_wait( p_this->p_access_control );
                                            // Why is p_this inside the mutex?
        }
    } binary_semaphore_post( &( p_this->mutex ) );
}

void lightswitch_post( lightswitch_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        --( p_this->population );

        if ( p_this->population == 0 ) {
            binary_semaphore_post( p_this->p_access_control );
        }
    } binary_semaphore_post( &( p_this->mutex ) );
}
```

E.5 Events

The interface of the `rendezvous_t` data structures includes four interface functions:

`void event_init( event_t *const p_this )`
: Initialize the event.

`void event_wait( event_t *const p_this );`
: Wait on the event.

`void event_signal( event_t *const p_this );`
: Signal the event and if there is at least one task waiting on p_this event, release that task.

`void event_broadcast( event_t *const p_this );`
: Signal the event and release all tasks that are currently waiting on the event.

The condition can be most easily implemented using two binary semaphores (one for mutual exclusion and the other for waiting), a turnstile, a count of the number waiting, and a count of the number to be released:

```
typedef struct {
    size_t size, freed;
    binary_semaphore_t mutex, waiting;
    turnstile_t entrance;
} event_t;

void event_init( event_t *const p_this ) {
    p_this->size = 0;
    binary_semaphore_init( &( p_this->mutex ), 1 );
    binary_semaphore_init( &( p_this->waiting ), 0 );
    turnstile_init( &( p_this->entrance ), TURNSTILE_UNLOCKED );
}

void event_wait( event_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        turnstile_pass( &( p_this->entrance ) );
        ++( p_this->size );
    } binary_semaphore_post( &( p_this->mutex ) );

    binary_semaphore_wait( &( p_this->waiting ) );

    --( p_this->freed );

    if ( p_this->freed == 0 ) {
        turnstile_unlock( &( p_this->enter ) );
    } else {
        binary_semaphore_post( &( p_this->waiting ) );
    }
}
```

```
void event_signal( event_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        if ( p_this->size > 0 ) {
            turnstile_lock( &( p_this->entrance ) );
            p_this->freed = 1;
            --( p_this->size );
            binary_semaphore_post( &( p_this->waiting ) );
        }
    } binary_semaphore_post( &( p_this->mutex ) );
}

void event_broadcast( event_t *const p_this ) {
    binary_semaphore_wait( &( p_this->mutex ) ); {
        if ( p_this->size > 0 ) {
            turnstile_lock( &( p_this->entrance ) );
            p_this->freed = p_this->size;
            p_this->size = 0;
            binary_semaphore_post( &( p_this->waiting ) );
        }
    } binary_semaphore_post( &( p_this->mutex ) );
}
```

Appendix F Implementation of a buffer

We will first look at abstract buffers. A buffer is an abstract data type that has two operations:

`void push( char c );`
If the buffer is full, put the calling function to sleep until the buffer is no longer full. Enter the character onto the buffer. If the buffer was empty and there are tasks waiting for the character, wake one of them up.

`char pop();`
If the buffer is empty, put the calling function to sleep until the buffer is no longer empty. Remove and return a character from the buffer. If the buffer was full and there are tasks waiting for the hole, wake one of them up.

Let's implement these two functions in C using POSIX semaphores.

```
#include <semaphore.h>

typedef struct {
    char *data;
    size_t size;
    size_t capacity;

    size_t head;
    size_t tail;

    sem_t on_empty;
    sem_t on_filled;
} buffer_t;

void buffer_init( buffer_t *const p_this, size_t n ) {
    p_this->size = 0;
    p_this->data = (char *) malloc( n );
    p_this->capacity = ( p_this->data == NULL ) ? 0 : n;

    sem_init( &on_empty,  0, 0 );
    sem_init( &on_filled, 0, p_this->capacity );
}
```

```
void buffer_push( buffer_t *const p_this, char c ) {
    sem_wait( &(p_this->on_filled ) );

    sem_wait( &(p_this->mutex) );

    if ( p_this->size == 0 ) {
        p_this->head =  p_this->tail = 0;
        p_this->data[0] = c;
        p_this->size = 1;
    } else {
        ++p_this->tail;

        if ( p_this->tail == p_this->capacity ) {
             p_this->tail = 0;
        }

        p_this->data[p_this->tail] = c;
        ++p_this->size;
    }

    sem_post( &( p_this->mutex ) );
    sem_post( &( p_this->on_empty ) );
}

char buffer_pop( buffer_t *const p_this ) {
    sem_wait( &( p_this->on_empty ) );

    sem_wait( &( p_this->mutex ) );

    if ( p_this->size == 0 ) {
        p_this->head =  p_this->tail = 0;
        p_this->data[0] = c;
        p_this->size = 1;
    } else {
        ++( p_this->tail );

        if ( p_this->tail == p_this->capacity ) {
             p_this->tail = 0;
        }

        p_this->data[p_this->tail] = c;
        ++( p_this->size );
    }

    sem_post( &( p_this->mutex ) );
    sem_post( &( p_this->on_full ) );
}
```

Appendix G An introduction to bitwise operations

For reasons such as scarcity of memory, parity bits, checksums, memory allocation and communications, embedded systems more likely to require bit manipulations than general-purpose applications. Most programming languages have some concept of bit-wise operations; rather than treating the bytes at a memory location (be it an `int`, a `double`, or `char`) as the datatype it represents, instead, the operations are simply performed on the bits themselves.

The C/C++ programming language has one unary bit-wise operator, NOT; three binary bit-wise operators AND, OR and XOR; and two bit-shifting operators, left and right shift. We will discuss all six operators and discuss various applications of these operators.

G.1 Bitwise unary not

The bitwise not operator `~` returns the data with each `0` converted to a `1` and each `1` converted to a `0`.

As an example,

```
short x = 42;
printf( "%d %d %d\n", x, ~x, (~x) + 1 );
```

has the output `42 -43 -42`. You will understand why this is true if you remember the 2's complement representation of negative numbers. If you were to look at the binary representation, you would see what is happening, recalling that 42 = 32 + 8 + 2 = 101010_2:

```
 x          0   0   0   0   0   0   0   0   0   0   1   0   1   0   1   0
~x          1   1   1   1   1   1   1   1   1   1   0   1   0   1   0   1
(~x) + 1    1   1   1   1   1   1   1   1   1   1   0   1   0   1   1   0
```

Note: Does unary `~` or binary `+` have higher precedence? That is, is `~x + 1` interpreted as `(~x) + 1` or `~(x + 1)` (which will result in two different answers)? Rather than looking it up, simply write your code to be as clear as possible with respect to your intentions. One recommendation from Steve Oualline's *Practical C Programming* is: `*` and `/` have higher precedence than `+` and `-`, and put parentheses around everything else.

G.2 Bitwise binary AND

The bitwise AND operator `&` performs a logical AND on each pair of corresponding bits. For example, given

```
short x = 29837, y = -15743;
printf( "%d %d %d\n", x, y, x & y );
```

the output is `29837 -15743 16513`, as can be seen here:

```
  x         0   1   1   1   0   1   0   0   1   0   0   0   1   1   0   1
  y         1   1   0   0   0   0   1   0   1   0   0   0   0   0   0   1
x & y       0   1   0   0   0   0   0   0   1   0   0   0   0   0   0   1
```

You may note that 16513 = $2^{14} + 2^7 + 2^0$ = 16384 + 128 + 1 = 100000010000001_2. Obviously, bitwise AND is not normally useful for integers except in one case: it is trivial to determine if a number is odd by doing a bitwise AND with `1` (recalling that in C and C++, if the condition of an

```
if ( x & 1 ) {
    printf( "x is odd\n" );
} else {
    printf( "x is even\n" );
}
```

Basically, the bitwise AND of a variable `x` with 1 sets all other bits to zero, and it keeps the last bit of `x`. If `x` is positive and odd, then the last bit is `1`, and if `x` is negative and odd, then last bit of `-x` is a `1`, the complement of that bit is therefore `0`, and after we add `1` to the result, it is again `1`.

Bitwise operations are also useful for extracting bits. For example, if we wanted to consider both the first byte and the second byte of a number, the following would do so:

```
short x = 29837;
short MASK_BYTE_0 = 255;     // = 2^8 - 1
short MASK_BYTE_1 = 65280;   // = (2^8 - 1)*2^8
printf( "%d %d %d\n", x, x & MASK_BYTE_0, x & MASK_BYTE_1 );
```

As the variable name suggests, when we use a value to extract a byte (or any number of bits), we often call that value a *mask*, as it *masks* off part of the number. The output of this is `29837 141 29696`, and we can see this is true as follows:

```
      x          0   1   1   1   0   1   0   0   1   0   0   0   1   1   0   1
  MASK_BYTE_0    0   0   0   0   0   0   0   0   1   1   1   1   1   1   1   1
x & MASK_BYTE_0  0   0   0   0   0   0   0   0   1   0   0   0   1   1   0   1
```

and 141 = 128 + 8 + 4 + 1. Similarly,

```
      x          0   1   1   1   0   1   0   0   1   0   0   0   1   1   0   1
  MASK_BYTE_1    1   1   1   1   1   1   1   1   0   0   0   0   0   0   0   0
x & MASK_BYTE_1  0   1   1   1   0   1   0   0   0   0   0   0   0   0   0   0
```

and 29696 = $2^{14} + 2^{13} + 2^{12} + 2^{10}$.

The bitwise and operator can also be used for a compact representation of Boolean values. For example, if we had five Boolean values, if we were to declare five Boolean variables

```
bool zero, sign, carry, overflow, parity;
```

this would occupy four bytes; but for five Boolean values, we really only need five bits, or less than one byte. Thus, we could use each bit in a one-byte flag variable, as follows:

	Blank	Blank	Blank	Parity	Overflow	Carry	Sign	Zero
Bit	7	6	5	4	3	2	1	0

Generally, to do this, we define a number of masks, as follows, and then test:

```
// In a header file or at the top of the file outside any function
#define ZERO      1
#define SIGN      2
#define CARRY     4
#define OVERFLOW  8
#define PARITY   16

// Inside a function
unsigned char status_flag = 20;

if ( status_flag & ZERO ) {
    printf( "the result was zero\n" );
} else {
    printf( "the result was non-zero\n" );
}

if ( status_flag & SIGN ) {
    printf( "the sign is negative\n" );
} else {
    printf( "the sign is positive\n" );
}

if ( status_flag & CARRY ) {
    printf( "a carry resulted during the last operation\n" );
}

if ( status_flag & OVERFLOW ) {
    printf( "an overflow resulted during the last operation\n" );
}

if ( status_flag & PARTY ) {
    printf( "the parity is odd (an odd number of 1s)\n" );
} else {
    printf( "the parity is even (an even number of 1s)\n" );
}
```

If the status flag is 20, or `00010100`, then the output is

```
the result was non-zero
the sign is positive
a carry resulted during the last operation
the parity is odd (an odd number of 1s)
```

It is also possible to clear a bit within a flag using bitwise AND and bitwise NOT:

```
// Set the carry flag to 0
status_flag = status_flag & ~CARRY;
          // C/C++ also allows you to write status_flag &= ~CARRY, similar to x += 15
```

If you now wanted to introduce another flag that had three states, there are three bits left over, so we could use two of them to store:

```
#define VOLTAGE_MASK    96      // 01100000
#define LOW              0      // 00000000
#define HIGH            32      // 00100000
#define HIGH_IMPEDANCE  64      // 01000000
```

Now, we could run additional tests:

```
switch ( status_flag & VOLTAGE_MASK ) {
    case LOW:             printf( "voltage was low\n" ); break;
    case HIGH:            printf( "voltage was high\n" ); break;
    case HIGH_IMPEDANCE:  printf( "voltage was set to high impedance\n" ); break;
    default:              printf( "an error occurred\n" );
}
```

The default would be triggered if both bits were set to `1`.

Note that `&` is very different from `&&`. In a conditional statement, any non-zero value is true, while only `0` is false. Thus, while `6 & 9` is `0` (and therefore false) `6 && 9` evaluates to true, which is usually `1`.

You will note that because C was originally designed to implement operating systems, because bitwise operations are much more common than logic operations, the bit-wise operation got the simpler notation.

Bitwise and can also be used as efficient means of calculating a number modulo a power of two. The modulus (or remainder) operation is very expensive in general: if you must calculate the modulus, try to ensure that it is a power of two, and then use this technique instead:

If $N = 2^n$, the expression

```
y[x % N];         // E.g., y[x % 256]
```

is equivalent to

```
y[x & (N - 1)];   // E.g., y[x & 255]
```

For even better suggestions, see:

http://embeddedgurus.com/stack-overflow/2011/02/efficient-c-tip-13-use-the-modulus-operator-with-caution/

Another interesting application of binary operators is determining if a number is a power of two:

```
value != 0 && (value & (value - 1)) == 0
```

If you are using signed integers, you must also check that `value != INT_MIN`.

Next, we move on to bitwise OR.

G.3 Bitwise binary OR

The bitwise OR operator | performs a logical OR on each pair of corresponding bits. For example, given

```
short x = 29837, y = -15743;
printf( "%d %d %d\n", x, y, x & y );
```

the output is `29837 -15743 -2419`, as can be seen here:

```
   x          0   1   1   1   0   1   0   0   1   0   0   0   1   1   0   1
   y          1   1   0   0   0   0   1   0   1   0   0   0   0   0   0   1
 x | y        1   1   1   1   0   1   1   0   1   0   0   0   1   1   0   1
```

You may note that $2419 = 2^{11} + 2^8 + 2^6 + 2^5 + 2^4 + 2^1 + 2^0 = 100101110011_2$. The result of the above computation is the 2's complement of this number. As with bitwise AND, bitwise OR is also not normally useful for integers, but exceptions exit. There are some cases where it is necessary that a number is odd (as with double hashing). In this case, a number can be made odd by switching the last bit to `1`:

```
// 'x' may be even or odd
x = x | 1;                  // C/C++ also allows you to write x |= 1, like x += 7
// 'x' is now odd
```

This is much more efficient than either of

```
if ( !( x & 1 ) ) {          if ( (x % 2) == 0 )
    x += 1;                      x += 1;
}
```

In other cases, when it is known that a number is even, `++x` can be replaced by `x |= 1`. This is often the case if you are calculating `x = 2*x + 1`, which can be replaced by two statements `x *= 2` and `x |= 1`.

The bitwise OR operator can be used to set a bit within a flag:

```
// Set the zero flag to 1
status_flag |= ZERO;
```

This sets the zero bit to `1` regardless of its previous value.

G.4 Bitwise binary XOR

The bitwise exclusive-OR or XOR operator ^ performs a logical exclusive OR on each pair of corresponding bits. For example, given

```
short x = 29837, y = -15743;
printf( "%d %d %d\n", x, y, x ^ y );
```

the output is `29837 -15743 -18932`, as can be seen here:

```
  x         0   1   1   1   0   1   0   0   1   0   0   0   1   1   0   1
  y         1   1   0   0   0   0   1   0   1   0   0   0   0   0   0   1
x & y       1   0   1   1   0   1   1   0   0   0   0   0   1   1   0   0
```

You may note that 18932 = $2^{15} + 2^{12} + 2^8 + 2^7 + 2^6 + 2^5 + 2^4 + 2^2 = 16384 + 128 + 1 = 100100111110100_2$. The result of the above computation is the 2's complement of this number. Obviously, bitwise AND is not normally useful for integers except in one case: it is trivial to determine if a number is odd by doing a bitwise AND with `1`:

The bitwise XOR operator can be used to flip a bit within a flag:

```
// Flip the parity bit
status_flag ^= PARITY;
```

This changes a parity bit of `1` into a `0` and vice versa.

The XOR operator can also be used to calculate the parity of a byte (counting how many 1s):

```
unsigned short x = 29837;
int parity = 0, bit;

//           16
// 65536 == 2
for ( bit = 1; bit <= 65536; bit *= 2 ) {
    if ( x & bit ) {
        parity ^= 1;
    }
}
```

You will also see how the XOR operator is used in the calculation of the cyclic-redundancy-check, a more complicated algorithm than calculating the parity, but used for the same purpose.

As a humorous aside, as it is not very practical, you can swap two variables without an intermediate variable. For example, you can write

```
tmp = x;
x = y;
y = x;
```

as

```
x ^= y;
y ^= x;
x ^= y;
```

G.5 Shifting operators

Another common operation used in operating systems are shifting operators: moving all the bits either to the left or to the right. The left- and right-shift operators are `<<` and `>>`, respectively, and the second operand is always an integer indicating how many bits the given direction.

```
x = y << n;    // Shift the bits of 'y' n bits to the left
```

```
x = y >> m;    // Shift the bits of 'y' m bits to the right
```

The first would include zeros in the *n* least-significant bits of the result, and the second would include zeros in the *m* most-significant bits of the result. For example, given

```
short x = 29837;
printf( "%d %d %d\n", x, (short)(x << 3), (short)(x >> 5) );
```

the output is `29837 -15743 -18932`, as can be seen here:

x	0	1	1	1	0	1	0	0	1	0	0	0	1	1	0	1
x << 3	1	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0
x >> 5	0	0	0	0	0	0	1	1	1	0	1	0	0	1	0	0

The reason for casting the `(short)(x << 3)` is that C/C++ will put the result into an `int`, in which case the answer is $11101001000110100\mathbf{0}\mathbf{0}\mathbf{0}_2$, and not $10100100011010\mathbf{0}\mathbf{0}\mathbf{0}_2$.

One of the most obvious applications of bit shifting is multiplication- and division-by-powers-of-2:

```
int i;
short x = 1;
printf( "%d", x );

for ( i = 1; i <= 16; ++i ) {
    x <<= 1;
    printf( ", %d", x );
}

printf( "\n" );
```

The output is

```
1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, -32768, 0
```

Note that 1000000000000000_2 is negative according to the 2's complement representation of integers, and the value is the negative of $0111111111111111_2 + 1 = 1000000000000000_2 = 32768$.

With bit shifting, it is very easy to create masks:

```
printf( "%u\n", (1 << 6) - 1 );
```

has the output $63 = 111111_2$, and

```
printf( "%u\n", ((1 << 6) - 1) << 5 );
```

has the output 11111100000_2. Of course, internally, all bits to the left of the leading 1 are 0.

Therefore, we may define:

```
#define MASK_BYTE_0  ((1 << 8) - 1)       // 00000000000000000000000011111111
#define MASK_BYTE_1  (MASK_BYTE_0 << 8)   // 00000000000000001111111100000000
#define MASK_BYTE_2  (MASK_BYTE_1 << 8)   // 00000000111111110000000000000000
#define MASK_BYTE_3  (MASK_BYTE_2 << 8)   // 11111111000000000000000000000000
```

At this point, we may mask a 4-byte datatype:

```
unsigned int c = 3026017933;
printf( "%u\n", c & MASK_BYTE_0 );    // %u indicates it is an unsigned integer type
printf( "%u\n", c & MASK_BYTE_1 );
printf( "%u\n", c & MASK_BYTE_2 );
printf( "%u\n", c & MASK_BYTE_3 );
```

The output is

```
141
24064
6094848
3019898880
```

so whereas $c = 10110100010111010101111010001101_2$, these four values are

$$00000000000000000000000010001101_2$$
$$00000000000000000101111000000000_2$$
$$00000000010111010000000000000000_2$$
$$10110100000000000000000000000000_2$$

In order to determine the bytes themselves, we must shift them to the right:

```
unsigned int c = 3026017933;
printf( "%u\n", c & MASK_BYTE_0 );
printf( "%u\n", (c & MASK_BYTE_1) >>  8 );
printf( "%u\n", (c & MASK_BYTE_2) >> 16 );
printf( "%u\n", (c & MASK_BYTE_3) >> 24 );
```

Now the output is

```
141
94
93
180
```

or 10001101_2, 01011110_2, 01011101_2 and 10110100_2, respectively.

Of course, masks need not respect the boundaries of bytes. One technique of converting an arbitrary 32-bit number into an *m*-bit hash value is to multiply the number by a large prime and then extract the middle *m* bits:

```
unsigned int hash( unsigned int x, unsigned int m ) {
    assert( 1 <= m && m <= 32 );     // Requires #include <assert.h> or <cassert>

    if ( m == 32 ) {
        return 3026017933 * x;
    }

    unsigned int offset = 32 - (m >> 1);   // 32 - m/2

    // MASK =   0000000000 111111111111 0000000000
    //          |<---  --->|<--m bits-->|<---  --->|
    //      32 - m - offset bits          offset bits
```

```
    unsigned int MASK = ((1 << m) - 1) << offset;

    unsigned int return_value = ((3026017933*x) & MASK) >> offset;
}
```

This could be made more efficient, but it brings across the point.

Similarly, we can use bit-wise operations for division by powers of two: the output of

```
unsigned int x = 525;
printf( "%d %d\n", x >> 2, x /   4 );
printf( "%d %d\n", x >> 5, x /  32 );
printf( "%d %d\n", x >> 8, x / 256 );
```

is

```
131 131
16 16
2 2
```

Note: As a question of clarity versus efficiency, you may be very tempted to replace

```
#define MASK_BYTE_0  ((1 << 8) - 1)       // 00000000000000000000000011111111
#define MASK_BYTE_1  (MASK_BYTE_0 << 8)   // 00000000000000001111111100000000
#define MASK_BYTE_2  (MASK_BYTE_1 << 8)   // 00000000111111110000000000000000
#define MASK_BYTE_3  (MASK_BYTE_2 << 8)   // 11111111000000000000000000000000
```

with

```
#define MASK_BYTE_0  255
#define MASK_BYTE_1  65280
#define MASK_BYTE_2  16711680
#define MASK_BYTE_3  4278190080
```

because you think that the second is more efficient than the first—after all, the first has to be computed, whereas in the second, the numbers are hard coded. In a short word: **DON'T**. For four good reasons:

1. The compiler will determine that `((1 << 8) - 1)` works out to a constant—the compiler will calculate this value and replace that with, in this case, `255`.
2. What if you made a mistake in your calculation? The compiler will calculate `4278190080` every time correctly.
3. More subtly, what happens if someone does a search and replace of 1900 with 2000 in the source file, assuming that 1900 only appears as a base year?
4. A reader can always understand the intention of the explicit code. This author knows that 255 is $2^8 - 1$ and that 25280 is sort-of-close to 2^{16}, but the significance is not obvious.

G.6 Summary

We have introduced bitwise and shifting operators in C and C++, and looked at various applications of these operators in various common applications within the practice of programming, mostly with respect to aspects related to embedded programming and the authoring of operating systems.

Appendix H Efficient mathematics

The on-line version of this text book has examples of efficient calculation of mathematical functions and operators.

Appendix I Trigonometric approximations

The on-line version of this textbook has an exceptionally accurate implementation of the sine and therefore other trigonmetric functions.

Appendix J Complex numbers and linear algebra

The on-line version has an appendix introducing complex numbers and linear algebra.

Glossary

ADT abstract data type
API application programming interface
AVL Adelson-Velsky and Landis
BIBO bounded-input—bounded-output
CAN controller area network
CMSIS Cortex Microcontroller Software Interface Standard
CPU central processing unit
CRC cyclic redundancy check
CTL computational tree logic
DFT discrete Fourier transform
digraph directed graph
DM deadline monotonic
ECU electronic control unit
EDF earliest-deadline first
FCFS first-come—first-served
FFT fast Fourier transform
FIFO first-in—first-out
FIR finite impulse response
GPU graphics processing unit
HDD hard-disk drive
IDE integrated development environment
IIR infinite impulse response
ISR interrupt service routine
JPL Jet Propulsion Laboratory
JSON JavaScript object notation
LLF least-laxity first (equivalent to LSF)
LR link register
LRU least recently used
LSB least-significant bit
LSF least-slack first
LTI linear and time independent
LTL linear temporal logic
`malloc` memory allocate
μC microcontroller
MCU microcontroller (micro-controlling unit)
MPU memory protection unit
MSB most-significant bit
MSP main stack pointer
NASA National Aeronautics and Space Administration
OO object oriented
PC program counter
PN Petri net
POSIX Portable Operating System Interface
PSP process (thread) stack pointer
RAM random-access memory

RM	rate monotonic
ROM	read-only memory
RTOS	real-time operating system
RTX	real-time executive
SHA	Secure Hash Algorithm
SP	stack pointer
SSD	solid-state drive
STN	shortest-task next
`tid`	thread identifier
TCB	thread/task control block
TTCAN	time-triggered controller area network
XML	extensible mark-up language

References

Books

Stan Augarten, *State of the Art: A Photographic History of the Integrated Circuit*, 1983.

Robert L. Benson, "The Venona Story", Fort George G. Meade, MD, National Security Agency, Center for Cryptologic History, 2001.

Alan Burns and Andy Wellings, *Real-time Systems and Programming Languages: Ada 95, Real-time Java and Real-time POSIX*, 3rd ed., Addison Wesley, 2001.

Albert M.K. Cheng, *Real-time Systems: Scheduling, Analysis and Verification*, John Wiley & Sons, Inc., 2002.

Thomas H Cormen et al., *Introduction to Algorithms*, 3rd ed., 2009.

Allen B. Downey, *The Little Book of Semaphores*, Green Tea Press, 2008.

Maurice Herlihy and Nir Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.

S.M. Kuo, B.H. Lee, and W. Tian, "Real-Time Digital Signal Processing: Implementations and Applications", Wiley, 2006.

Phillip A. Laplante and Seppo J. Ovaska, *Real-Time Systems Design and Analysis: Tools for the Practitioner*, 4th ed., IEEE Press, 2012.

Donald Knuth, *The Art of Computer Programming*: *Fundamental Algorithms*, 3rd ed., Addison-Wesley, 1997.

Donald Knuth, *The Art of Computer Programming*: *Seminumerical Algorithms*, 3rd ed., Addison-Wesley, 1997.

Donald Knuth, *The Art of Computer Programming*: *Sorting and Searchign*, 3rd ed., Addison-Wesley, 1998.

James Martin, *Programming Real-time Computer Systems*, Prentice-Hall, 1965.

Gary Nutt, *Operating Systems*, 3rd ed., Pearson, Addison Wesley, 2003.

William H. Press et al., *Numerical Recipes in Fortran 77: The Art of Scientific Computing*, 2nd ed., 1992.

Ragunathan Rajkumar, *Synchronization in Real-time Systems: A Priority Inheritance Approach*, Kluwer Academic Publisher, 1991.

Alan C. Shaw, *Real-time Systems and Software*, John Wiley & Sons, Inc., 2001.

Steven W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, self-published, 1997-8.

William Stallings. *Operating Systems: Internals and Design Principles.* Prentice Hall, 4th ed., 2000.

Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 2nd ed., 2001.

TimeSys Corporation, *The Concise Handbook of Real-Time Systems*, v. 1.3, 2002.

Elecia White, *Making Embedded Systems: Design patterns for great software*, O'Reilly and Associates, Inc., 2011.

Papers

Steven M. Bellovin, “Frank Miller: Inventor of the One-Time Pad”, Cryptologia 35 (3), 2011, pp.203–222.

http://www.tandfonline.com/doi/abs/10.1080/01611194.2011.583711

S. Bensalem and Klaus Havelund, *Reducing False Positives in Runtime Analysis of Deadlocks*

http://ti.arc.nasa.gov/m/pub-archive/442h/0442%20(Bensalem).pdf

Andreas Grabner, *Sync your Timeouts: When Load Balancers Cause Database Deadlocks*

http://apmblog.compuware.com/2014/04/09/
sync-your-timeouts-when-load-balancers-cause-database-deadlocks/

Jet Propulsion Laboratory, *JPL Institutional Coding Standard for the C Programming Language*, California Institute of Technology, 2009.

Lamport and Mellior-Smith, *Synchronizing clocks in the presence of faults*, *J. ACM*, Vol. 32, No. 2 (Feb 1980), pp.105-17.

Robert Rönngren and Rassul Ayani, *A comparative study of parallel and sequential priority queue algorithms*, ACM Transactions on Modeling and Computer Simulation (TOMACS), Vol 7 Issue 2, April 1997, pp 157-209.

http://dl.acm.org/citation.cfm?id=249205

K.Y. Rozier, *Linear Temporal Logic Symbolic Model Checking*, Computer Science Review (2010), doi:10.1016/j.cosrev.2010.06.002

William Stallings, *Queuing Analysis*.

Index

About the authors

Douglas Harder is a continuing lecturer at the University of Waterloo in the department of electrical and computer engineering and currently holds a limited engineering licence. He previously worked at Waterloo Maple, Inc. as an intermediate developer of mathematical software and as a master corporal in the Canadian Forces reserve in the infantry and military intelligence.

Jeff Zarnett is a computer engineering graduate of the University of Waterloo and is currently a licenced professional engineer in Ontario. He is the director of Zarnett Consulting, Inc. and is a part-time lecturer at the University of Waterloo.

Vajih Montaghami is a graduate student at the University of Waterloo.

Allyson Giannikouris is a computer engineering graduate of the University of Waterloo and is a lecturer at the University of Waterloo. She is currently a licenced professional engineer in Ontario.

Colophon

The cover image was taken by Douglas Wilhelm Harder who holds the copyright.

From Wikipedia: *Phyllomedusa sauvagii*, commonly known as the *waxy monkey leaf frog*, is a hylid frog belonging to the subfamily of South and Central American leaf frogs, *Phyllomedusinae*, that inhabits the Chaco (dry prairie) of Argentina, Brazil, Bolivia and Paraguay. The subfamily consists of around 50 species in three well-known genera, *Phyllomedusa*, *Agalychnis*, and *Pachymedusa*. The vast majority of known species, including *Phyllomedusa sauvagei*, belong to the genus *Phyllomedusa*.

Phyllomedusa sauvagii has adapted to meet the demands of life in the trees. It does not need to return to the ground during the mating season; rather, it lays its eggs down the middle of a leaf before folding the leaf, sandwiching the eggs inside. Its nest is attached to a branch suspended over a stream, so the hatching tadpoles drop into the water. In common with other *phyllomedusines*, it has physiological and behavioural adaptations to limit water loss, including reducing water loss through the skin by lipid secretions, excretion of uric acid (uricotelism), and diurnal torpor. Lipid secretions are produced in a special type of cutaneous gland, and are spread over the surface of the skin by the legs in a complex sequence of wiping movements.

Males and females range from about 2 to 3 inches in length, with the females usually about 25 % larger than males. They move by walking rather than hopping, which is the reason for the "monkey" in their name. They are very calm, careful creatures. During the day, they bask in the sun with their legs pulled underneath them, and hunt for various insects at night.