Electrical and Computer Engineering
University of Waterloo

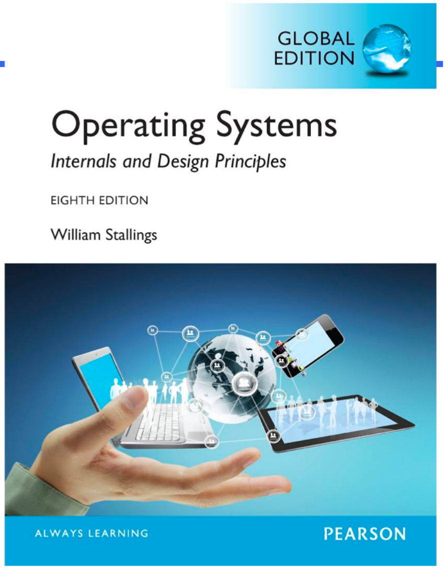
Operating Systems and Systems Programming

The Linux philosophy is to laugh in the face of danger. Oops. Wrong one. Do it yourself. That's it.

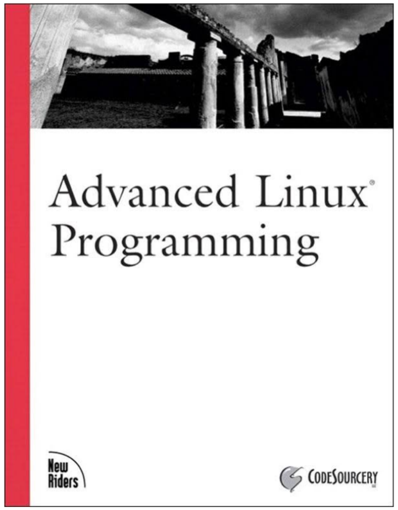
Linus Torvalds

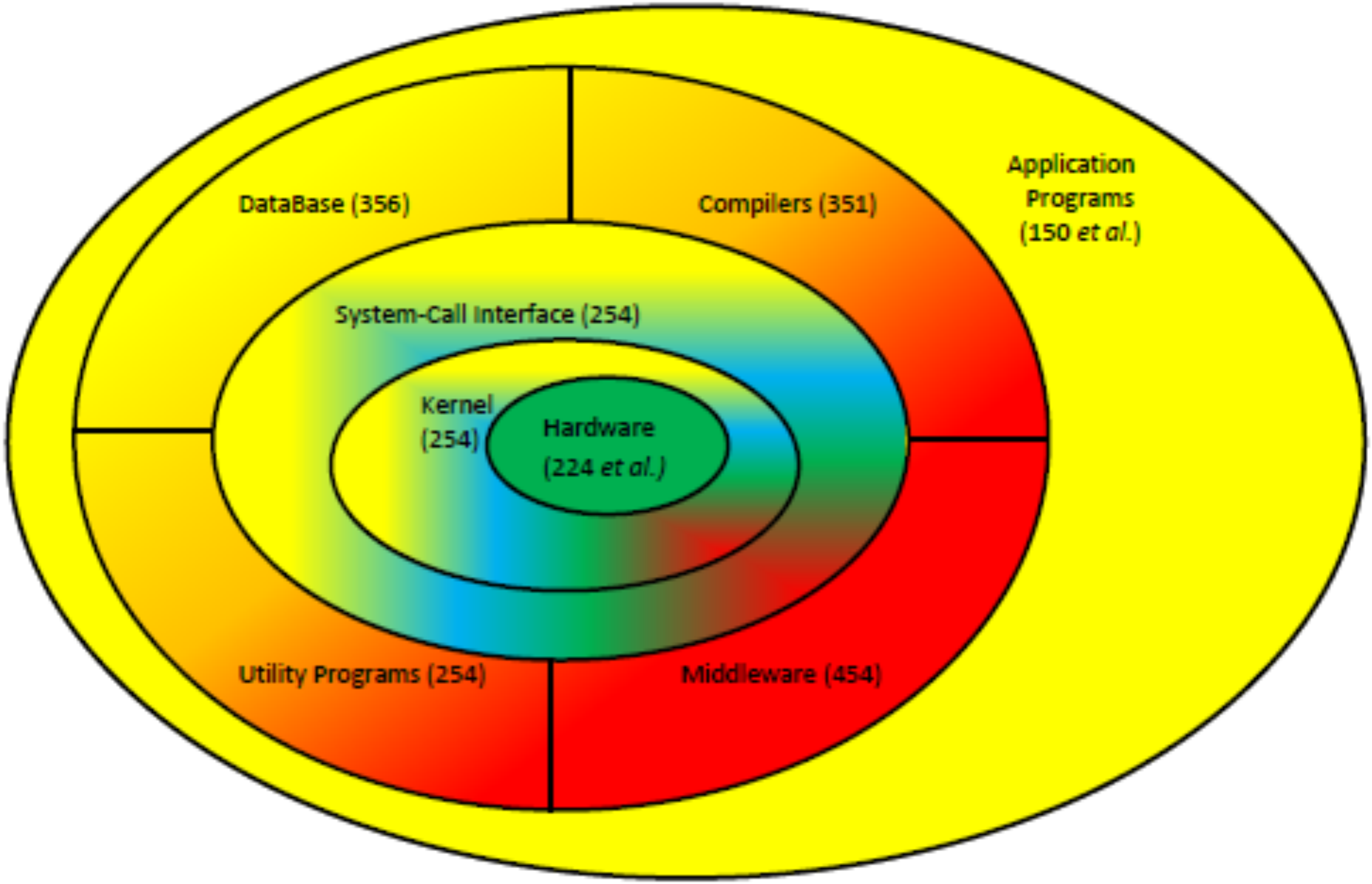
References

- William Stallings. Operating Systems: Internals and Design Principles



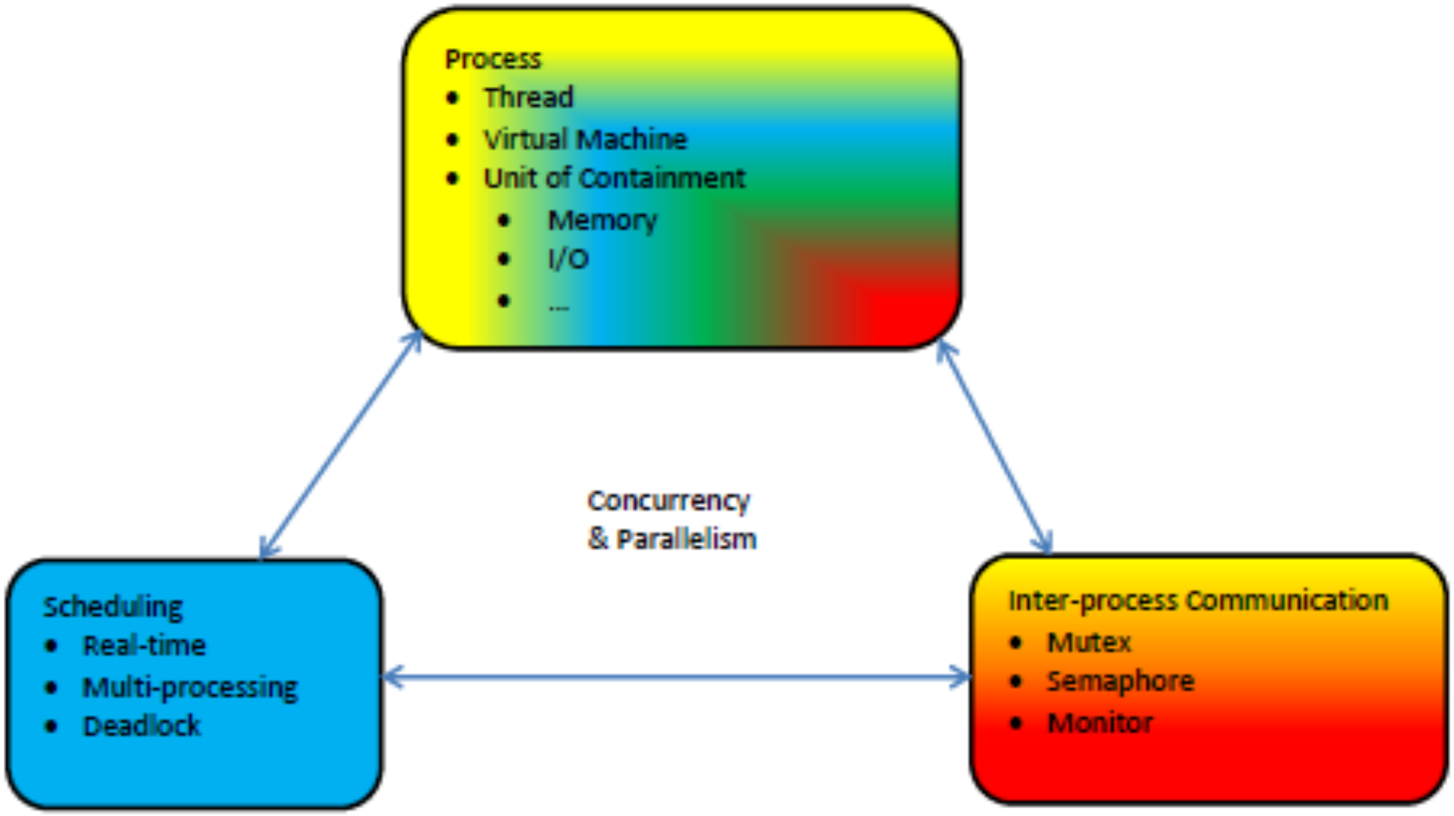
- Advanced Linux Programming http://www.advancedlinuxprogramming.com





The Modern Computer System

Operating System Concepts



What is an OS?

1. Hardware Abstraction

2. Resource Manager

Hardware Abstraction (1)

Hardware Abstraction (2)

This is beautiful?

```
fd = socket(AF_INET, SOCK_STREAM, 0);
getsockname(fd, (sockaddr *)&binder,
            &binderSockLen);
nready = select(maxfd+1, &rset, NULL, NULL, NULL);
FD_ISSET(listenfd, &rset);
connfd = accept(listenfd,
                (sockaddr *)&cliaddr, &clilen);
```

(Compared to the alternative, yes, it is)

Layers of Computer System

user views "system" as a set of applications

masks details of HW from Programmer

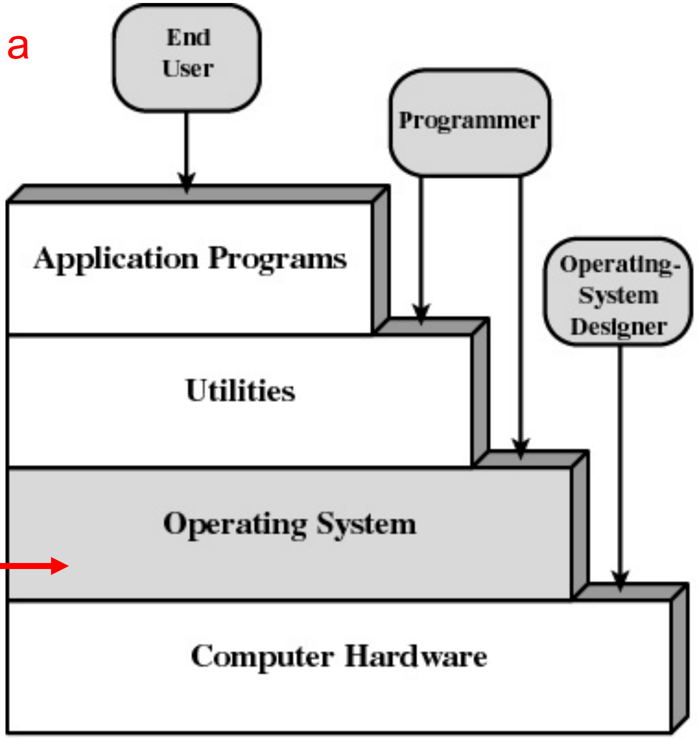


Figure 2.1 Layers and Views of a Computer System

William Stallings. Operating Systems: Internals and Design Principles.

Resource Manager

- More than one process is executing at a time
  - Need to facilitate sharing of
    - Processor
    - I/O
      - Keyboard
      - Mouse
      - Touchscreen
      - Disk
      - Network Interface Card (NIC)
      - ….
    - Memory

OS History

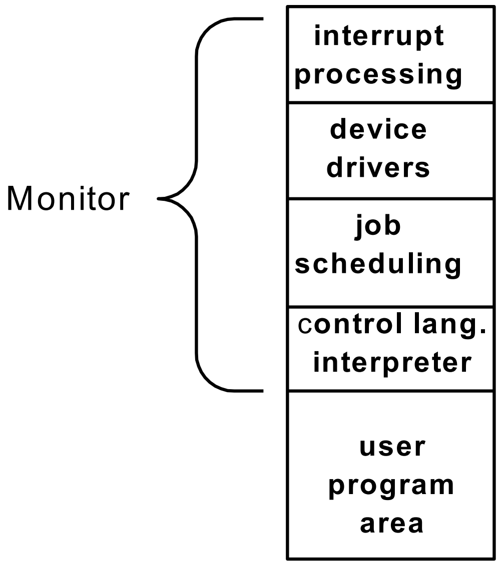
- Early days
  - no operating system
  - machine directly controlled from console [toggle switches, display lights], input device and printer
  - serial processing
  - need to schedule time to reserve machine
  - typical sequence:
    - load the compiler, load source program,
    - compile, load & link the compiled program
    - execute the compiled program

Early OS History [2]

- Simple batch systems with ‘monitors’
- Monitor
  - software that controls execution of programs
  - ‘jobs’ batched together
  - control returns back to monitor when program execution finished
  - monitor is resident in main memory, always available for execution

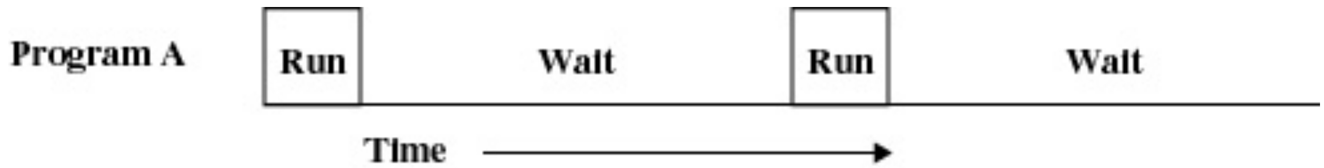
Job Control Language (JCL)

- special type of programming language
- contains instruction to the monitor
  - what compiler to use
  - what data to use

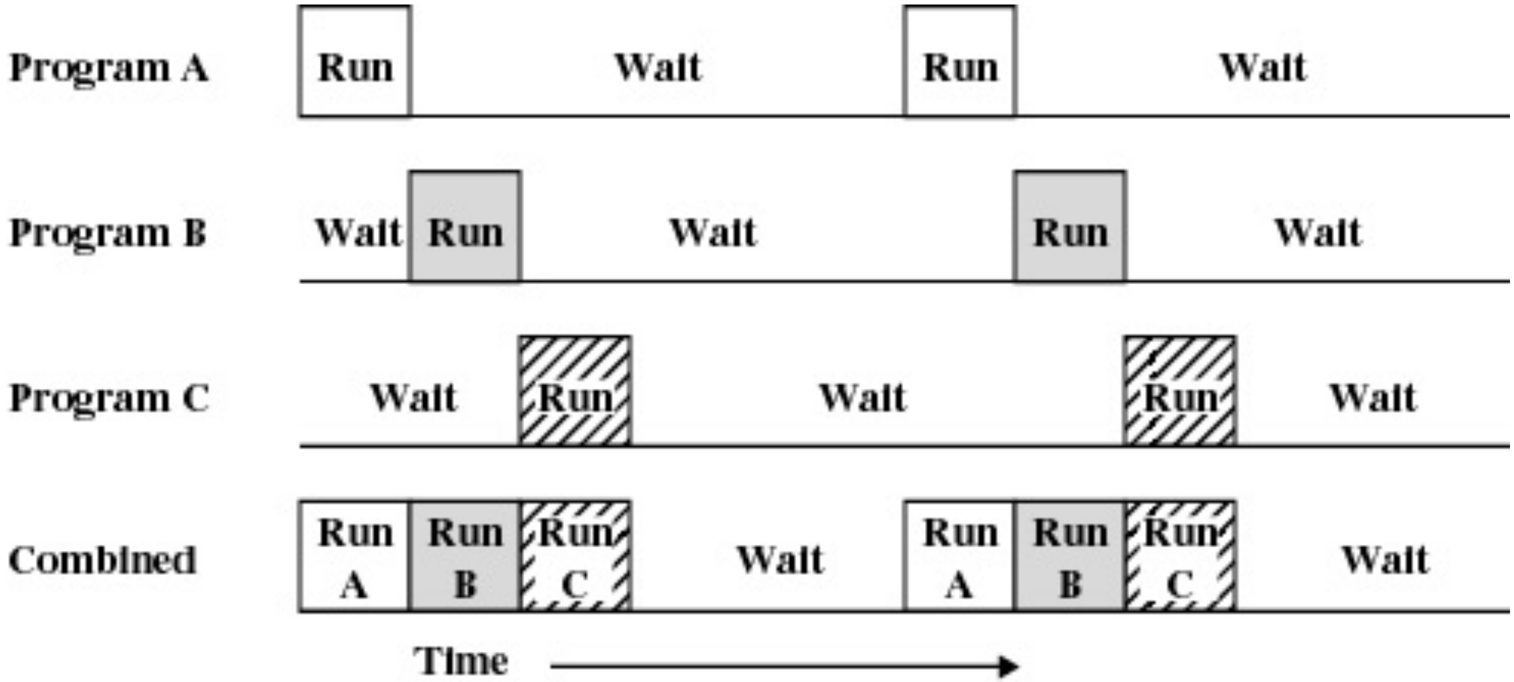


Memory layout for a resident monitor

Multiprogramming



(a) Uniprogramming



(c) Multiprogramming with three programs

OS History: Time Sharing OS

- Using multiprogramming to handle multiple interactive programs
- Processor time is shared among several users
- Each user has the illusion of having his/her own computer

Batch Multiprogramming vs Time Sharing

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job Control Language commands provided with the job	Commands entered at the terminal

(Brief) Hardware Review

- CPU
  - Fetch, decode, execute, fetch, decode, execute, ….
    - (with pipelining to improve performance, but we will ignore that)
  - General-purpose registers: R_1, R_2, … R_n
  - Special-purpose registers:
    - Program Counter (PC)
      - points to next instruction
    - Stack Pointer (SP)
      - points to current stack, which contains current frame
        - » input parameters, local variables, …
      - does not exist in every architecture
    - Program Status Word (aka, flags)
      - mode (user/kernel), condition code bits, *etc*.

OS Concepts

- Services
- Kernel
- Processes
- Memory management
  - Volatile memory
- File System
  - Long-term memory

OS Concepts

- Security/Protection
- Scheduling
- System Calls

OS Services

- Concurrency and Sharing
  - Concurrent execution of programs
  - Shared and controlled access to memory
  - Shared and controlled access to I/O devices
  - Shared and controlled access to files
  - Shared and controlled access to other system resources

load data + instructions,
schedule execution

-protect from unauthorized access, resolve conflicts

-knowledge of I/O devices + structure of data contained in file/storage system

OS Services [2]

- Detection of errors and recovery
  - internal and external hardware errors
    - memory error, device failure
  - software errors
    - *e.g.*, access to forbidden memory locations

-arithmetic overflow
-not able to grant request
-etc…..
-terminate program?
-retry?
-send error report?

OS Services [3]

- Monitoring and Accounting
  - monitor computer operation
  - monitor performance
  - collect execution time data
  - compute execution time statistics
  - maintain accounting information

maintain high processor utilization

-kernel (most frequently used) resident
-rest on storage, get as needed

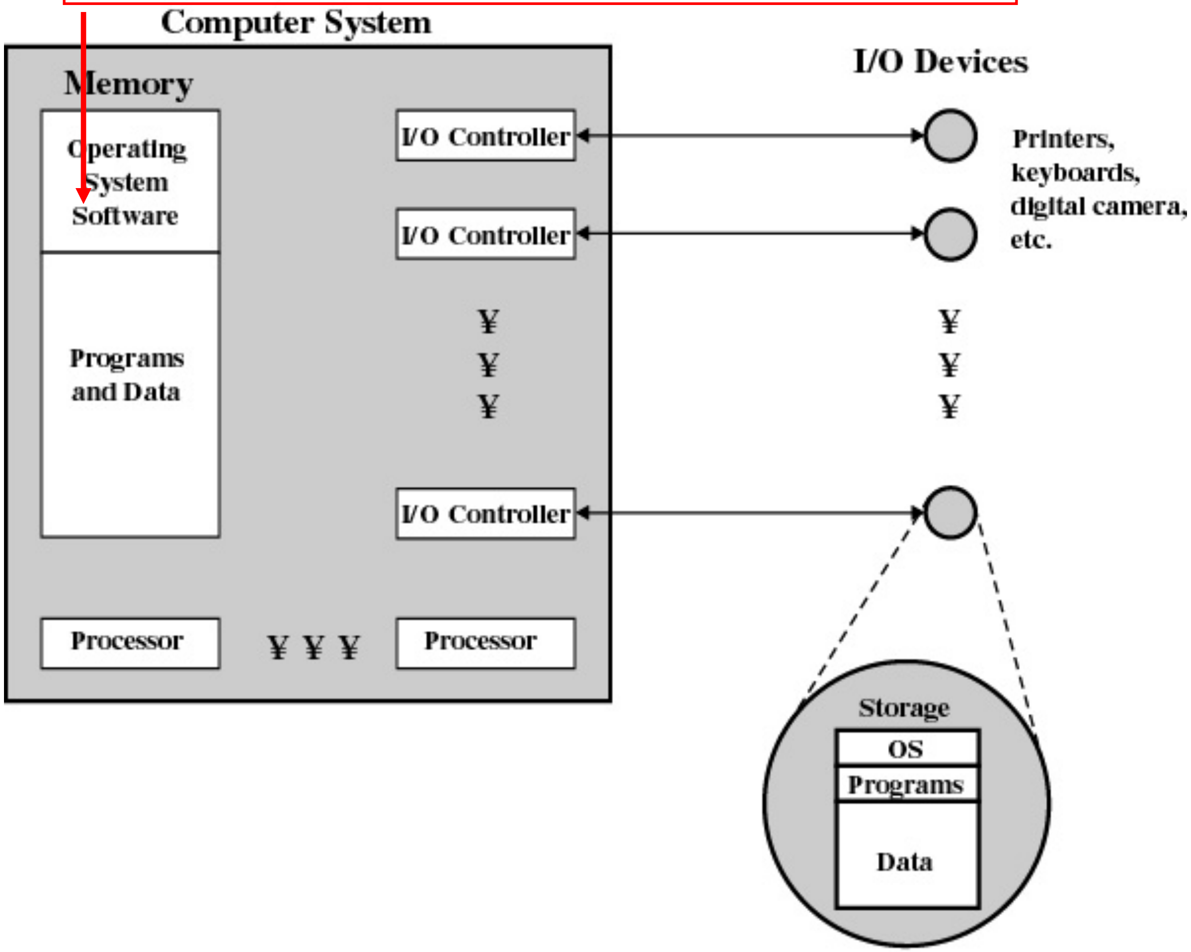


Figure 2.2 The Operating System as Resource Manager

Kernel

- Portion of operating system that provides the basic OS functionality
- Always resident in main memory

- interrupt/device drivers & handlers
- process/memory/resource management

Process

- consists of three components
  1. an executable program
  2. data needed/created by the program
  3. execution context of the program
     - all information OS needs to manage the process during and between execution [PC, CPU registers, stack pointers,
       files opened, resources owned...]

Memory Management

- Separation of process address spaces
- Protection and access control
- Automatic allocation and management
- Long-term storage management

typically met with Virtual Memory

Virtual Memory

- Physical memory [RAM]: limited, shared
- VM allows programmers to work with independent address spaces
- Parts of process address space may be in RAM, parts on disk
- If required, VM must also allow individual processes to share regions of memory

Paging

- Process view: main memory consists of a number of fixed-size blocks, called pages
- Main memory consists of correspondingly sized memory blocks called frames
- Virtual address has two components:
  a page number and an offset within the page
- A page may be located in any frame in main memory
- Underlying computer design issue:
  - does address bus carry virtual or real addresses?

Virtual Memory Addressing

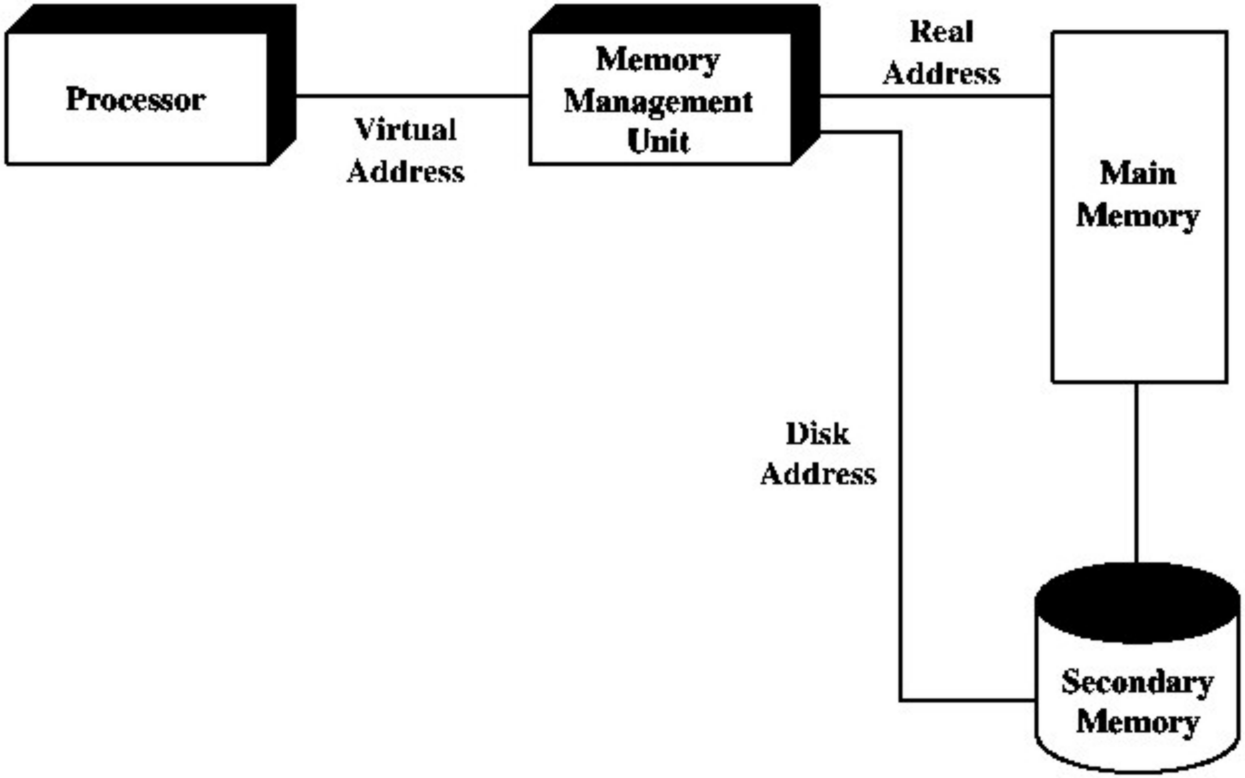


Figure 2.10 Virtual Memory Addressing

File System

- Implements long-term store
- Information stored in named objects called files
- Hierarchical structure: directories, subdirectories, files

Information Protection and Security

- Access control
  - regulate user access to the system
- Information flow control
  - regulate flow of data within the system and its delivery to programs/processes
- Certification
  - proving that access and flow control perform according to specifications

- enforce desired protection and security

Scheduling, Resource Management

- General situation: K resources, N processes wishing to use resources
- Need to decide who gets what and when
- Objectives (may conflict)
  - Fairness
  - Differential responsiveness
    - discriminate between different classes of processes
  - Efficiency
    - maximize throughput, minimize response time, and accommodate as many processes as possible

System Calls

- A call into the system!
  - Meaning:
    - Switching from user to kernel mode
  - Cost?
    - Need to preserve all state of calling entity

Example

- count = read(fd, &buf, n);

  1. User program
     a. Push n
     b. Push &buf
     c. Push fd
     d. Call “read” (go to 2)
     e. ….

  2. Library Set up trap
     a. Put code for “read” in relevant register
     b. Trap (go to 3)
     c. Return to 1.e

  3. In kernel:
     a. Dispatch
     b. Return to 2.c

System Calls - **Process Management**

- `pid = fork();`
- `pid = waitpid(pid, &statloc, options);`
- `rc = execve(name, argv, environp);`
- `exit(status);`

System Calls – **File Management**

- `fd = open(file, flags, mode);`
  - `fd = open(“./fubar”, O_RDWR);`
- `rc = close(fd);`
- `n = read(fd, buf, n);`
- `n = write(fd, buf, n);`
- `position = lseek(fd, offset, whence);`
- `rc = stat(name, &buf);`

System Calls

- Manual 2 contains information on all system calls
  - Thus: “man 2 <system call name>” will give you all you need to know about that system call
  - *e.g.*, “man 2 lseek” returns:

```
LSEEK(2)                    Linux Programmerâs Manual                  LSEEK(2)

NAME
       lseek - reposition read/write file offset

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       off_t lseek(int fd, off_t offset, int whence);

....
```

- (See “man man” for information on the un*x manual)
  - Use “apropos <whatever>” to hunt around for stuff

OS Software Structure

- Monolithic
- Layered
- Microkernel

Monolithic

- Single program in kernel mode
- All system calls are in a single address space
  - No protection once something is in the kernel
- *E.g.*, Unix

UNIX

- Hardware is surrounded by the operating-system
- Key part = the Unix kernel
- Comes with a number of user services and interfaces
  - shell
  - C compiler

UNIX Layered Structure

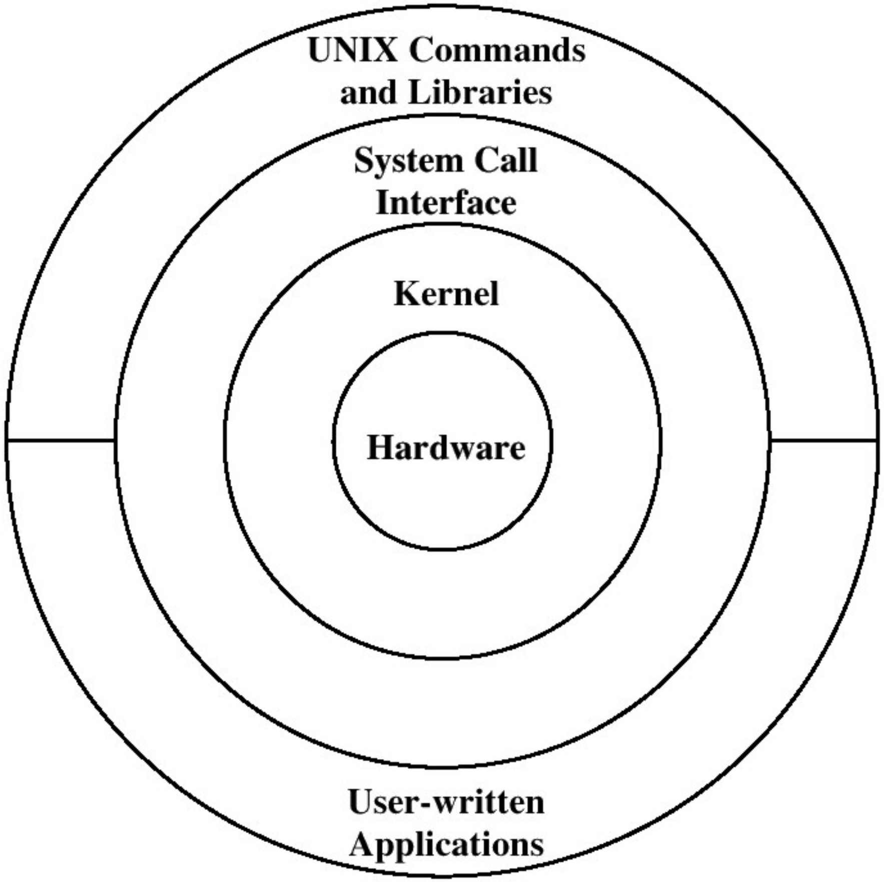


Figure 2.15 General UNIX Architecture

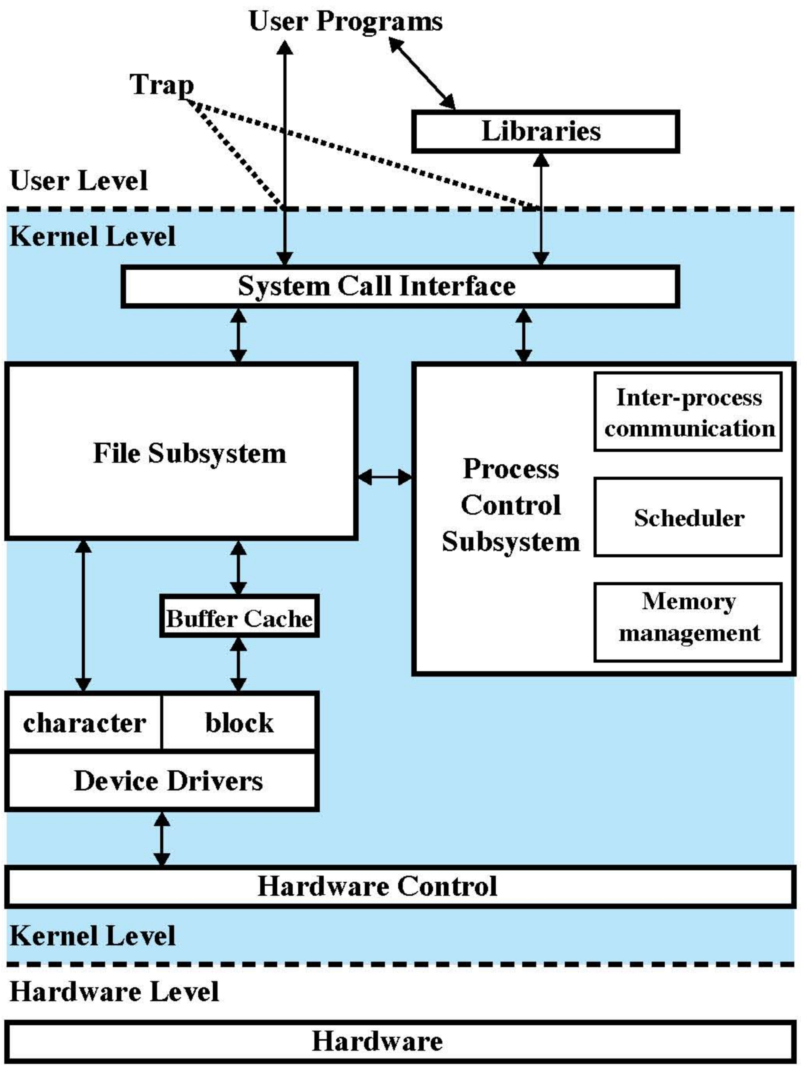


Figure 2.15 Traditional UNIX Kernel [BACH86]

Layered

- OS software structured as a collection of layers
- each layer performs a set of functions which are related
- each layer relies on the next lower level to perform more primitive functions
  - MULTICS
  - THE

Microkernel

- A few essential functions put in the kernel
  - basic process support
  - inter-process communication (IPC)
  - basic process scheduling
  - address space support
- Or even just a concurrency control/protection mechanism
  - Or even just an API for CC/P
  - Windows NT (as originally envisioned)
  - MINIX

Additional Characteristics of OSes

- Multithreading
  - process is further subdivided into threads that can run concurrently
- Thread
  - dispatchable unit of work
  - executes sequentially and is interruptible
- Process is a collection of one or more threads

Symmetric Multiprocessing

- OS capable of supporting multiple processors
  - or multiple cores
- these processors share same main memory and I/O facilities
- all processors can perform the same functions

OS Classification

- Different OSes depending on the nature of requirements
  - "General purpose"
    - Catchall name for OSes that will be put in front of a person
      - *e.g.*, Windows, MacOS, DOS, Un*x
    - Make general assumptions based on user-behaviour studies
  - Real-time
    - tailored to specific needs of real-time systems
      - Often associated with embedded, but most "general purpose" OSes have some real-time capability
        - » Why?
  - Embedded
    - Used in devices not often viewed as computers
      - *e.g.*, microwave ovens, TV Set, DVD player, anti-lock braking system, ….
    - Do not have the "typical" I/O facilities
      - *e*.g., no keyboard, mouse, monitor
      - Often no non-volatile storage

Processes and Threads

Process Model

- A program in execution
  - Which means what?
    - Sequential execution
      - ➔ Program Counter to keep track of next instruction
- Give the illusion (is it?) of a dedicated computer
  - CPU
  - Memory
  - Files
  - I/O

Major Requirements of an OS

- Interleave the execution of several processes
  - Concurrency (multiprogramming) is hard; why bother?
    - to maximize processor and resource utilization
      - Possibly while providing reasonable response time
- Allocate resources to processes
- Maintain separation of processes
- Support application program creation and termination of processes
- Support inter-process communication

So we have:

- One CPU
  - With one Program Counter
  - One memory subsystem
  - One file subsystem
  - One I/O subsystems
- N processes
  - Each with their own current view of
    - Where they are in execution (*i.e.,* their own PC)
    - What memory they have
    - What files they have open
    - What I/O they are doing

From the Machine Perspective

- Imagine a CPU with opcodes that are all one byte in size, and the programs are all simple sequential code
  - *e.g.*, a program could be
    - `int main (void) { int a; int b; a = 1; b = 2; a = a+b; return 0; }`
  - and Process A and B will both execute the same program
- PC = 0x1234 (Process A, instruction x)
- PC = 0x1235 (Process A, instruction x+1)
- PC = 0x1236 (Process A, instruction x+2)
- PC = 0x4962 (Switch to Process B, instruction y)
- PC = 0x4963 (Process B, instruction y+1)
- PC = 0x1237 (Switch to Process A, instruction x+3)

From the Process Perspective

- Process A does
  - …
  - instruction x
  - instruction x+1
  - instruction x+2
  - instruction x+3
  - …

- Process B does
  - …
  - instruction y
  - instruction y+1
  - instruction y+2
  - …

From the Time Perspective

B ----- -----

A ----- -----

Time →

- Which means you cannot assume you have time knowledge within your process; you need to use device timers if you want timing information
  - otherwise known as PacMan made of an 8088 computer running at 4.77 MHz doesn't work so well on a 80386 computer running at 25 MHz

So How Does an OS Manage This?

- It will need
  - Some data structure to maintain the state of any given process
  - Some data structure to maintain the state of all currently “executing” processes
    - Why is “executing” in quotation marks?
  - Process creation and termination mechanisms
  - Other control structures
  - Some mechanism(s) for deciding when to change the state of any given process

Maintaining Individual Process State

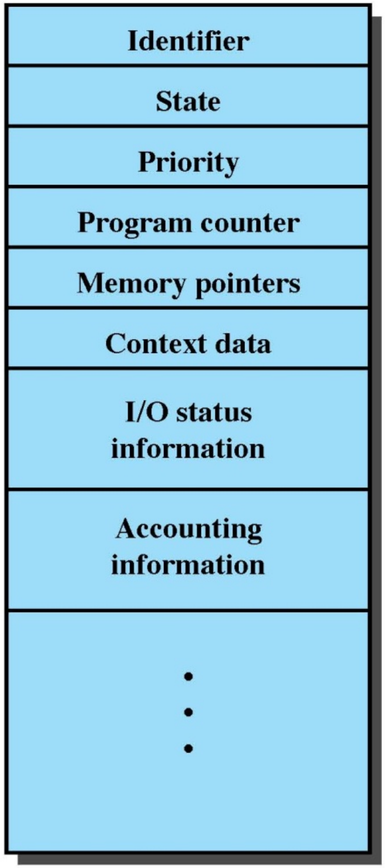


Figure 3.1 Simplified Process Control Block

Context data == registers

PCB: Process Identification

- Identifiers [often numeric]
  - Identifier of this process
  - Identifier of the process that created this process (parent process)
  - User identifier

PCB: Processor State Related

Application Programmer Visible Registers

- Control and Status Registers
  - A variety of processor registers that are employed to control the operation of the processor. These include
    - *Program counter*
    - *Condition codes*
    - *Status information*
    - *Stack Pointers*

PCB: Scheduling Info

- Scheduling and State Information

  This is information that is needed by the operating system to perform its scheduling function. Typical items:

  – *Process state:* defines the readiness of the process to be scheduled for execution

    - *e.g.*, running, ready, waiting, halted
    - see below

  – *Event:*

    - Identity of event the process is waiting for (if any)

PCB: Scheduling Info [2]

- *Scheduling-related information:*
  - dependent on the scheduling algorithm used.
  - examples: amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.
- *Priority:*
  - the scheduling priority of the process
  - In some systems, several values are required
    - *e.g.,* default, current, highest-allowable

PCB: OS Data Structure Support

- OS maintains a number of data structures to support its operation
- some data structures contain/refer to processes, *e.g.,*
  - all processes in a waiting state for a particular resource may be linked in a queue.
  - a process may be in a parent-child (creator-created) relationship with another process.
- the process control block may contain pointers/references to other processes to support these structures

PCB: Other Info

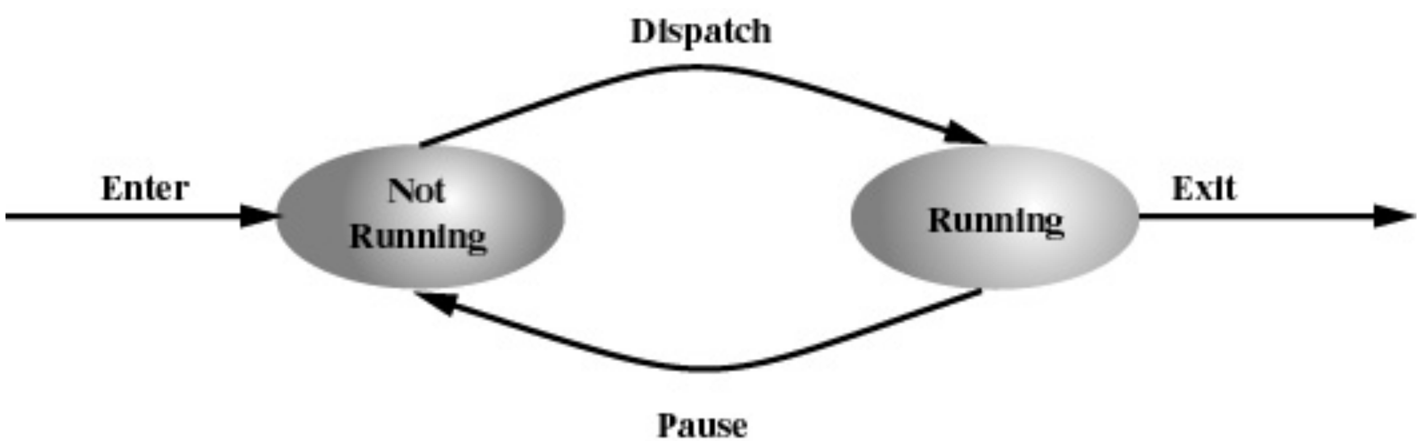
- Interprocess Communication Related
  - Various flags, signals, and messages may be associated with communication between two independent processes.
- Process Privilege Related
  - Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed.
  - Access to system utilities and services.

PCB: Other Info [2]

- Memory Management Related
  - pointers to tables that describe the virtual memory assigned to this process
- Resource Ownership and Utilization Related
  - resources given to the process may be indicated, such as opened files.
  - A history of utilization of the processor or other resources

Maintaining the State of all Processes A: 2-state

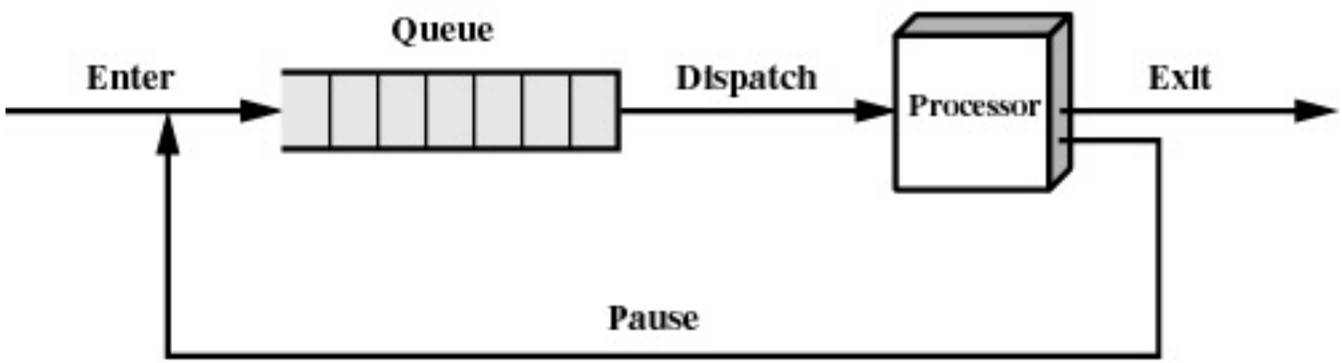
- Process may be in one of two states
  - Running
  - Not-running



(a) State transition diagram

Implementation: Queue

- not-running processes in a queue
- ‘dispatcher’ gives the processor to a process



(b) Queuing diagram

Maintaining the State of all Processes B: 5-state

- not-running state must be further subdivided into
  - ready to execute
  - blocked
    - *e.g.,* waiting for I/O
  - newly created
  - exiting
- reason: need to assist the dispatcher
  - dispatcher cannot just choose the process that has been in the queue the longest because it may be blocked

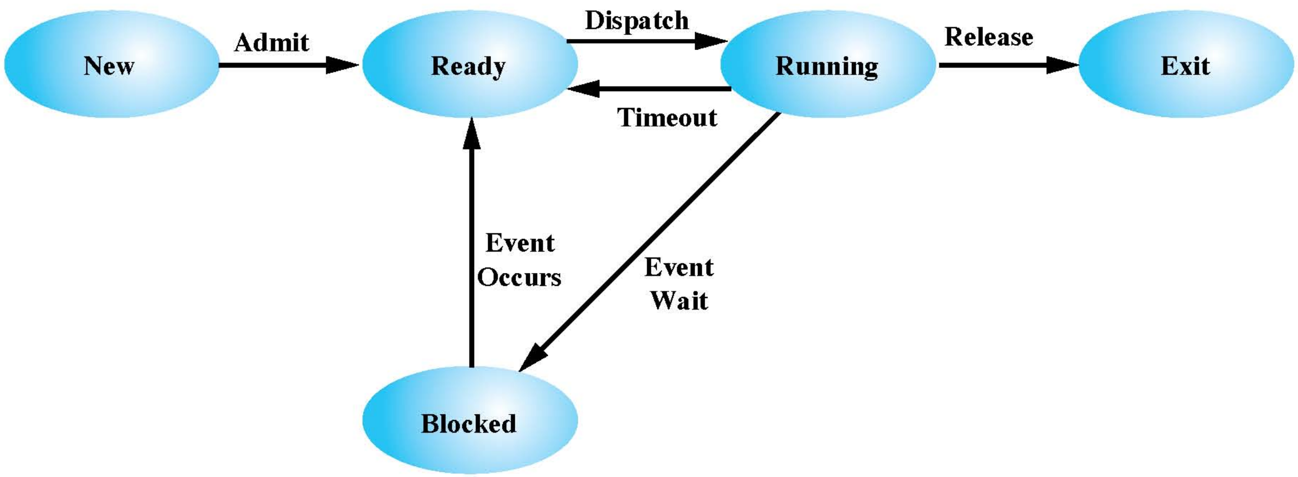
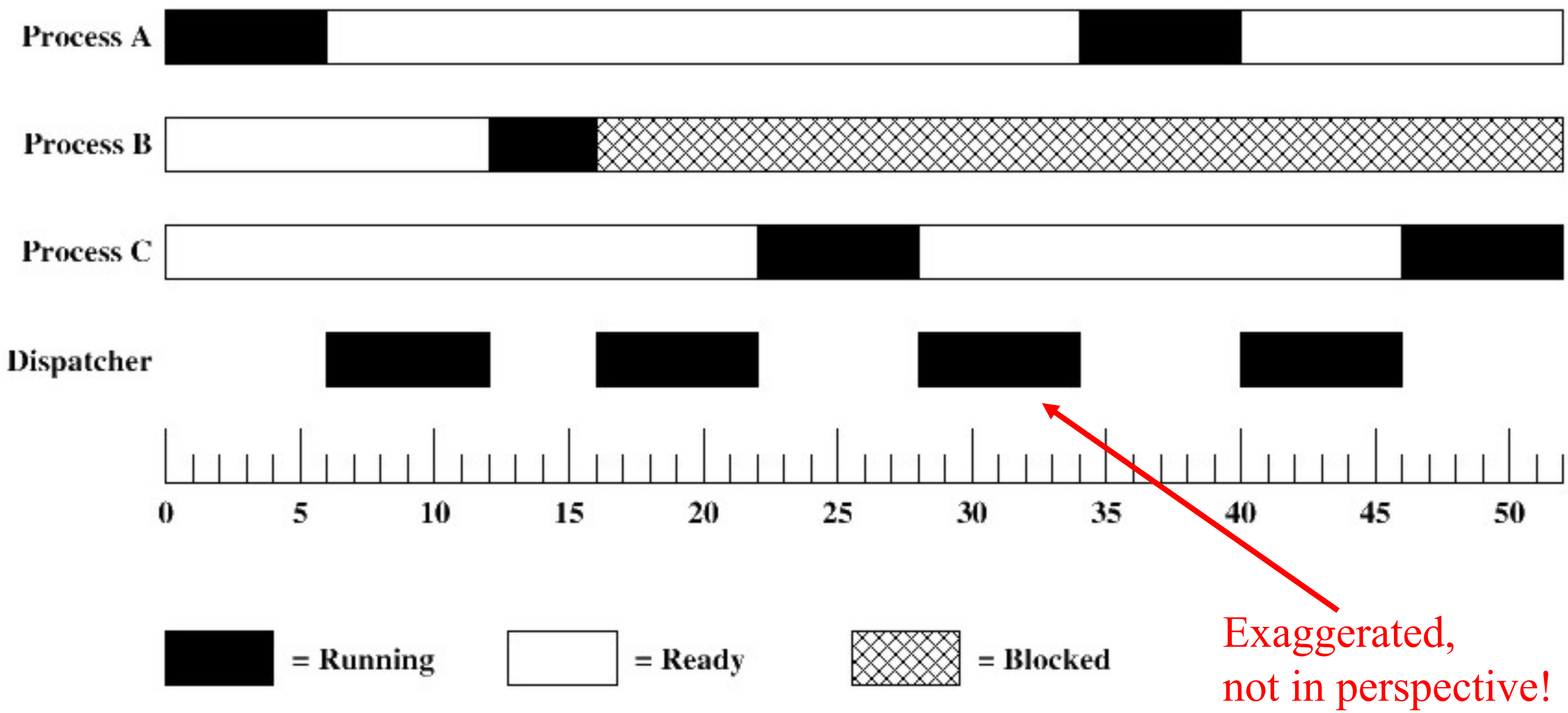
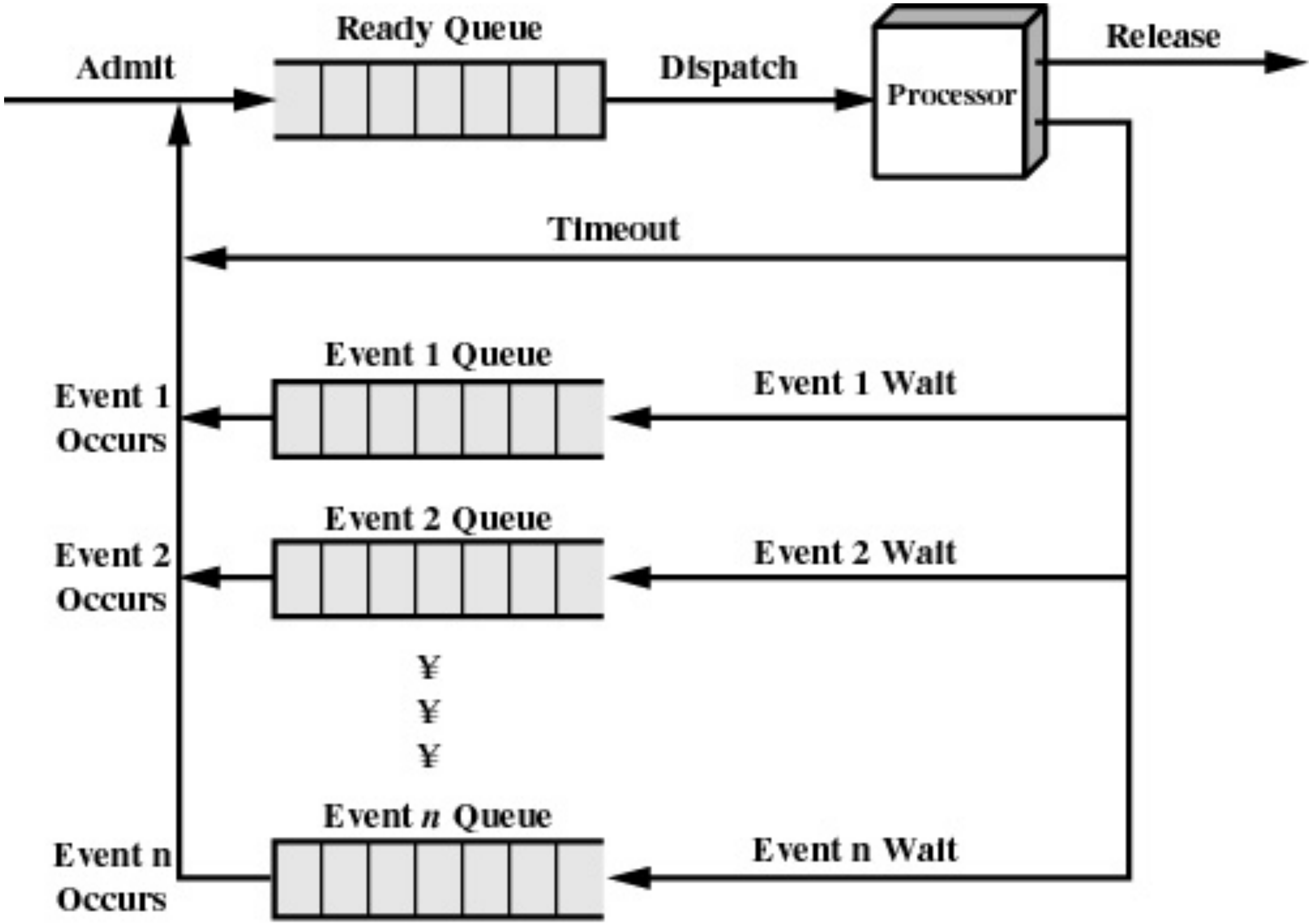


Figure 3.6 Five-State Process Model

(New, Ready, Blocked, Running, Exit)


Process A
Process B
Process C
Dispatcher
0
5
10
15
20
25
30
35
40
45
50
= Running
= Ready
= Blocked
Exaggerated,
not in perspective!



(b) Multiple blocked queues

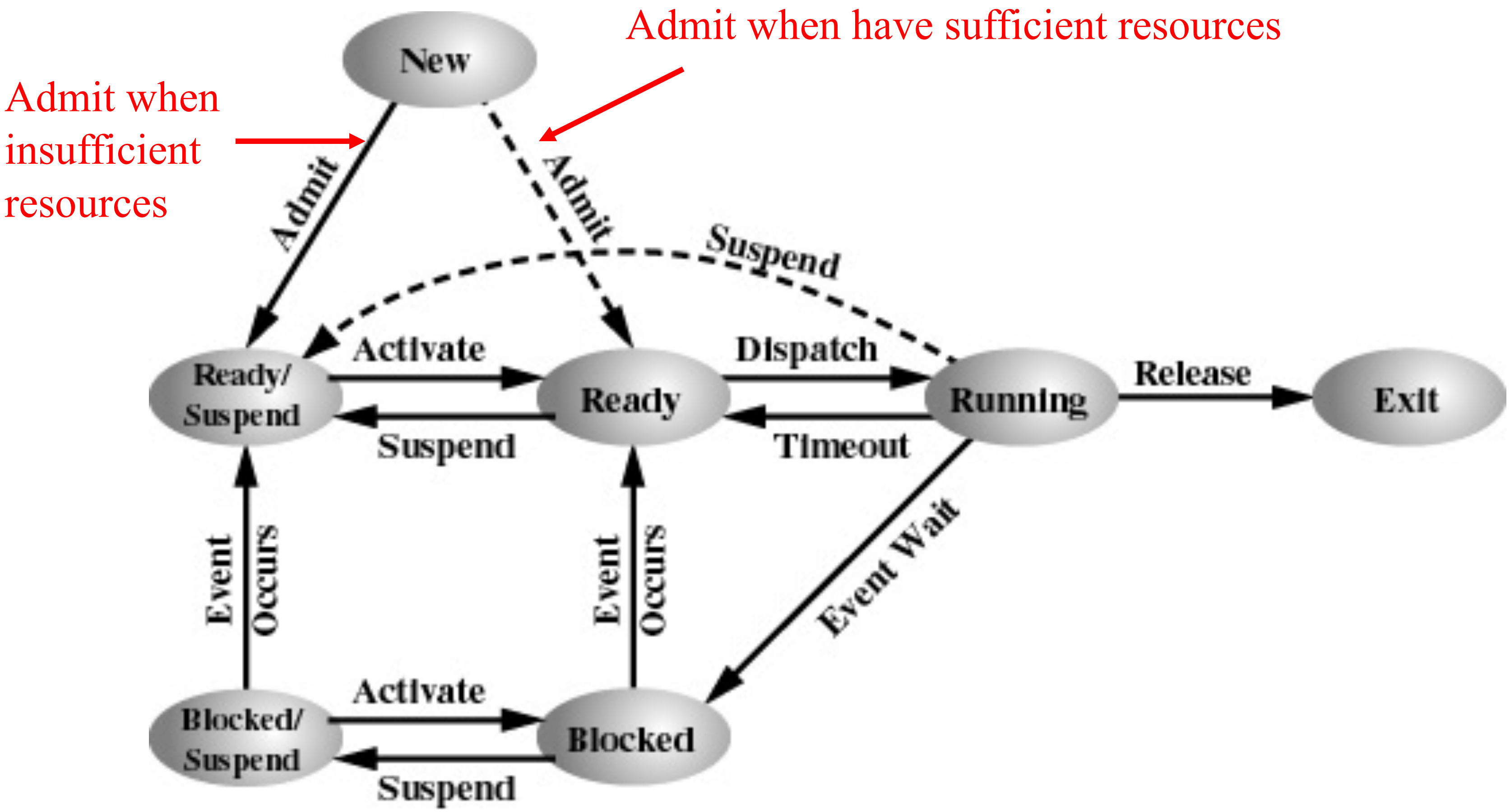
Process Suspension

- variety of scenarios under which it is desirable to temporarily ‘suspend’ a process
- two new states
  - ready, suspend
  - blocked, suspend

Reasons for Process Suspension

Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The OS may suspend a background or utility process or a process that is suspected of causing a problem.
Interactive user request	A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.
Timing	A process may be executed periodically and may be suspended while waiting for the next time interval
Parent process request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendents.

Maintaining the State of all Processes C: 7-state



(b) With Two Suspend States

Process Creation: When?

- Done at
  1. System initialization
     - Which will typically start some process at boot-up, and then use mechanism (2), below, to create any additional processes.
  2. System call by a running process
     - On un*x: `fork`() (often followed by `execve`())
     - On Windows: `CreateProcess`()
  3. User request
     - Which will be *via* either shell or GUI and will do (2) above.
  4. Batch-job initiation
     - *E.g.*, cron job on un*x
       – (see "`man cron`", "`man crontab`", and "`man 5 crontab`")
       – In this case the "cron daemon" must be running, and it starts the relevant cron job at the requisite time using mechanism (2)

Process Creation: What?

- Assign a unique process identifier
- Allocate space for the process
  - program, data, stack, PCB
- Initialize process control block
- Set up appropriate linkages
  - *e.g.*, add new process to linked list used for scheduling queue
- Create or expand other data structures
  - *e.g.*, maintain an accounting file

Process Termination

1. Process exits normally, by program choice
   – *e.g.*, "`exit(0);`"
   – See "`man 3 exit`"
2. Process exits with error
   – Per (1) above, with non-zero error code
   – `perror()` can be useful here; see "`man 3 perror`"
3. Fatal error
   – *e.g.*, uncaught signal
      • *e.g.*, `SEGV` without associated signal-handler code to catch the violation.
4. Killed by another process
   – e.g., "`kill -HUP <pid>`"
   – Note that "`kill`" simply sends a signal to the relevant process

Other Control Structures

- Information about the current state of each process and resource
- This information is often structured in tables
- Tables are constructed for most entities the operating system manages

Memory Tables

- Allocation of main memory to processes
- Allocation of secondary memory to processes
- Protection attributes for access to shared (or dedicated) memory regions
- Information needed to manage virtual memory

I/O Tables

- I/O device status: available or assigned
- I/O operation status
- Regions/locations in main memory being used as the source or destination of the I/O transfer

File Tables

- Files in existence
- For each file
  - location on secondary memory
  - current Status
  - attributes
- Sometimes this information is maintained by a file-management system

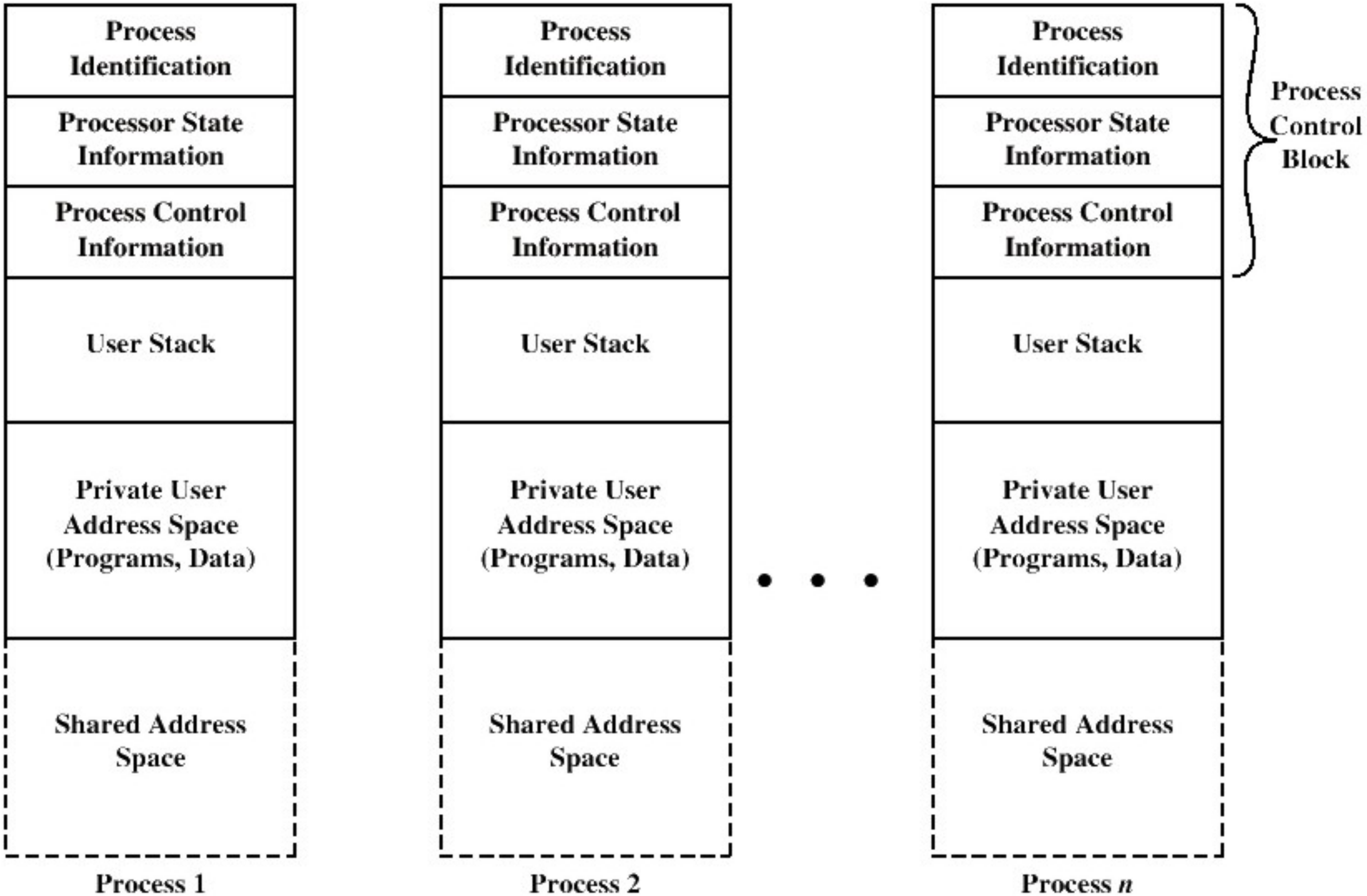
Process Table

- Where the object representing the process is located

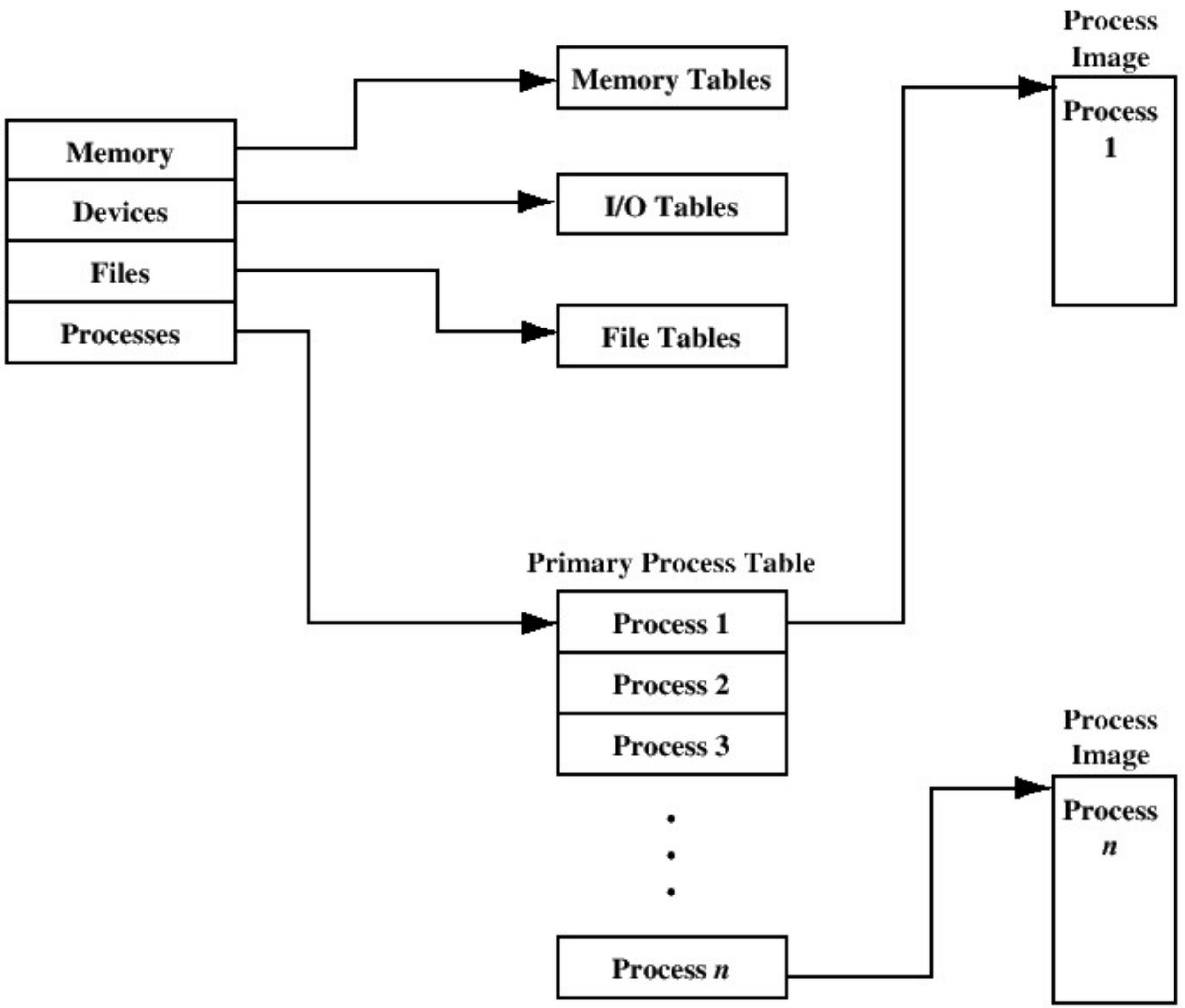
- Attributes necessary for individual process management
  - process ID
  - process state
  - location in memory

Process Location

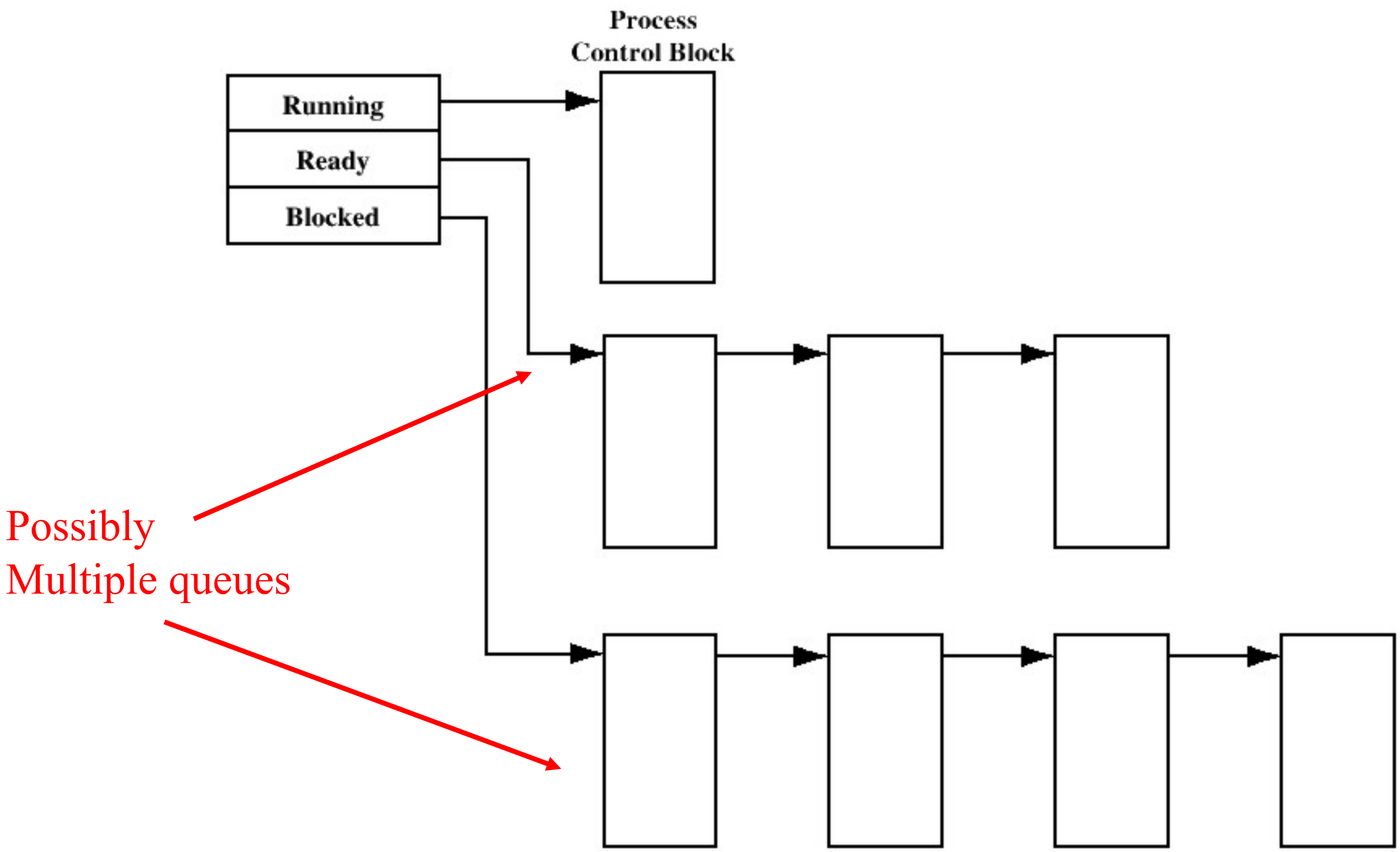
- Process includes set of programs to be executed
  - Program location
  - Data locations for local and global variables
  - Any defined constants
  - Stack
- Process control block
  - Per previous discussion
    - Collection of attributes frequently needed by OS
- Process image
  - Collection of program, data, stack, and attributes



User Processes in Virtual Memory



General Structure of Operating System Control Tables



Process List Structures

Changing Process State

- Process executes system call
  - *e.g.* file open
- Clock interrupt
  - process has executed for maximum time slice
- I/O interrupt
- Page fault
  - memory address accessed is not currently in main memory
- Error trap
  - process execution caused an error trap

Context Switch

- More than just switching from user mode to kernel mode
  - The currently executing process is to be changed
  - Which means: execution context must be switched
- Save the context of processor in the PCB
  - including program counter, other registers
- Update the PCB and put it to appropriate queue - ready, blocked, ...

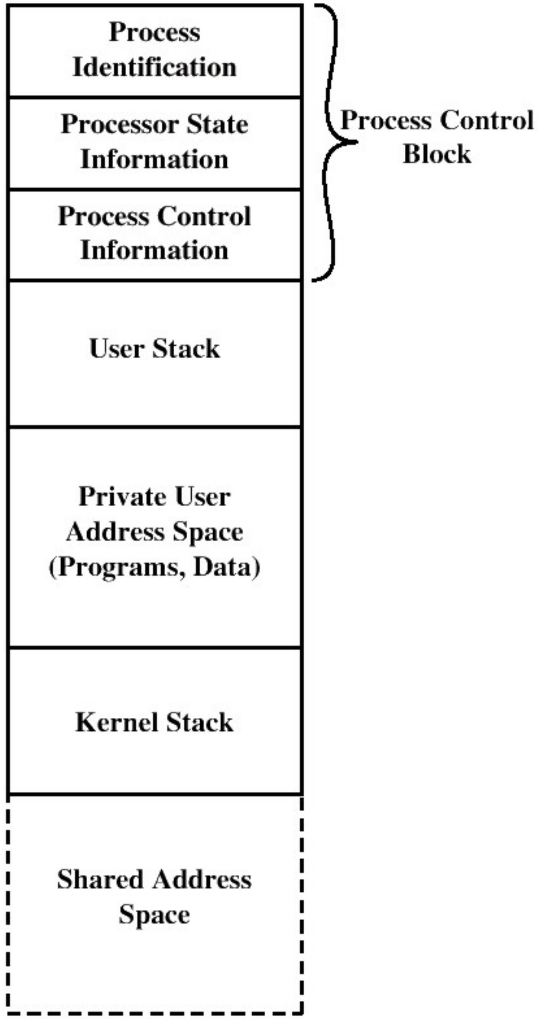
Context Switch [2]

- Select another process for execution (another PCB)
- Update the PCB of the process selected
- Update memory-management data structures
- Restore processor context to that of the selected process

- Question:
  - What does this imply for the contents of
    - Cache?
    - Main memory (as opposed to virtual memory)?

OS Execution

- Execution Within User Process
  - OS software executes within the context of a user process
  - process executes in privileged mode when executing OS code
- Kernel execution outside process
  - execute kernel outside of any process
  - operating system code is executed as a separate entity that operates in privileged mode

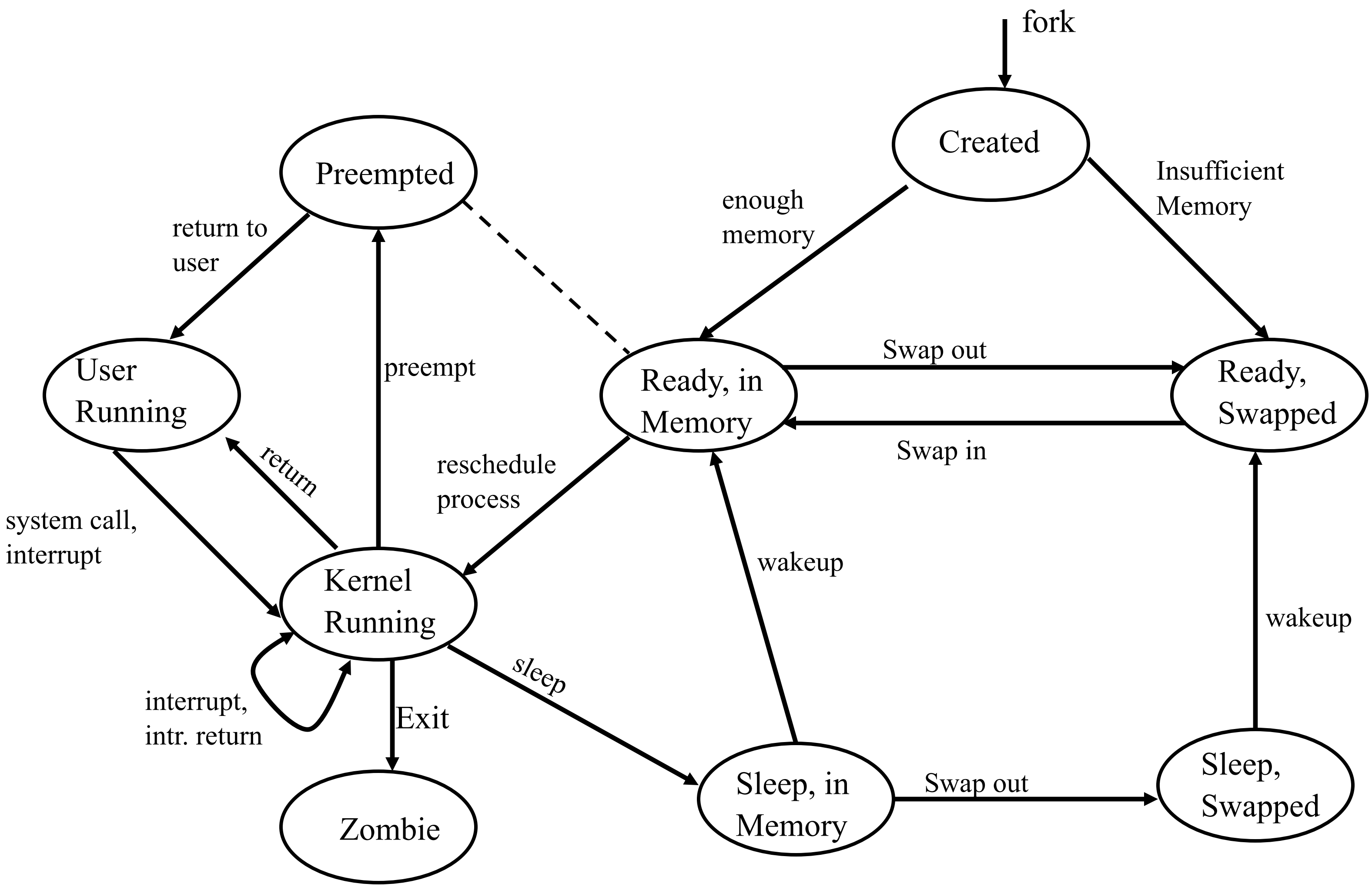


Process Image: Operating System Executes Within User Space

UNIX Process States

User Running	Executing in user mode.
Kernel Running	Executing in kernel mode.
Ready to Run, in Memory	Ready to run as soon as the kernel schedules it.
Asleep in Memory	Unable to execute until an event occurs; process is in main memory (a blocked state).
Ready to Run, Swapped	Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
Sleeping, Swapped	The process is awaiting an event and has been swapped to secondary storage (a blocked state).
Preempted	Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
Created	Process is newly created and not yet ready to run.
Zombie	Process no longer exists, but it leaves a record for its parent process to collect.

UNIX Process State Transitions



Problem

- What happens when I do a blocking system call?
  - *e.g.*, a browser does "`gethostbyname()`"
  - That process blocks
    - What if I have nothing else useful to run?
    - What if the "hostname" was a typo and I want to abort the call?
    - What if I want my application to execute faster?
    - What if my application is I/O bound?

Threads

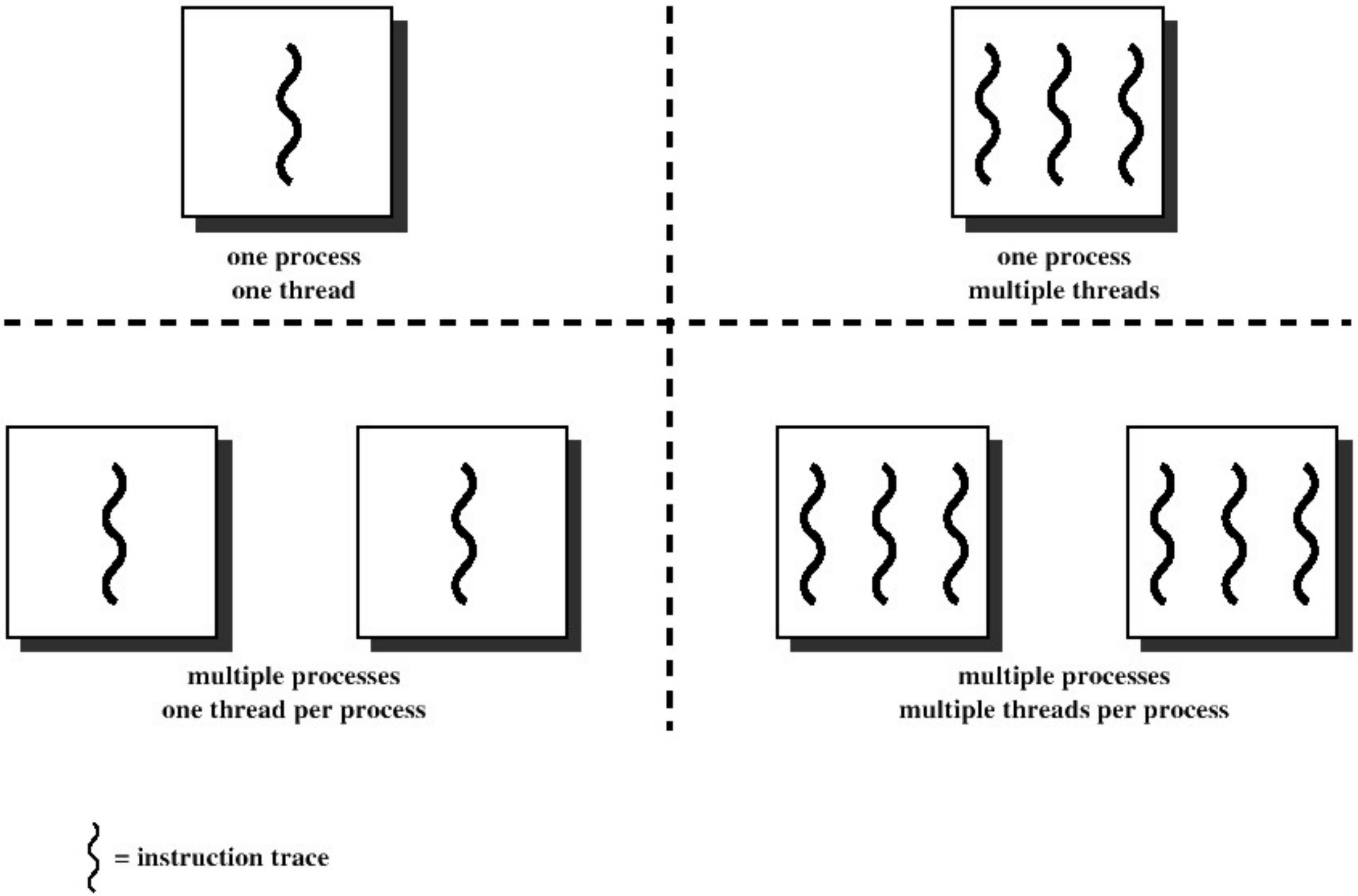


Figure 4.1 Threads and Processes [ANDE97]

Who gets what?

- the **unit of scheduling** is the thread
  - sometimes (*e.g.*, on Solaris) the lightweight process

- the unit of **resource ownership** is the process

Process

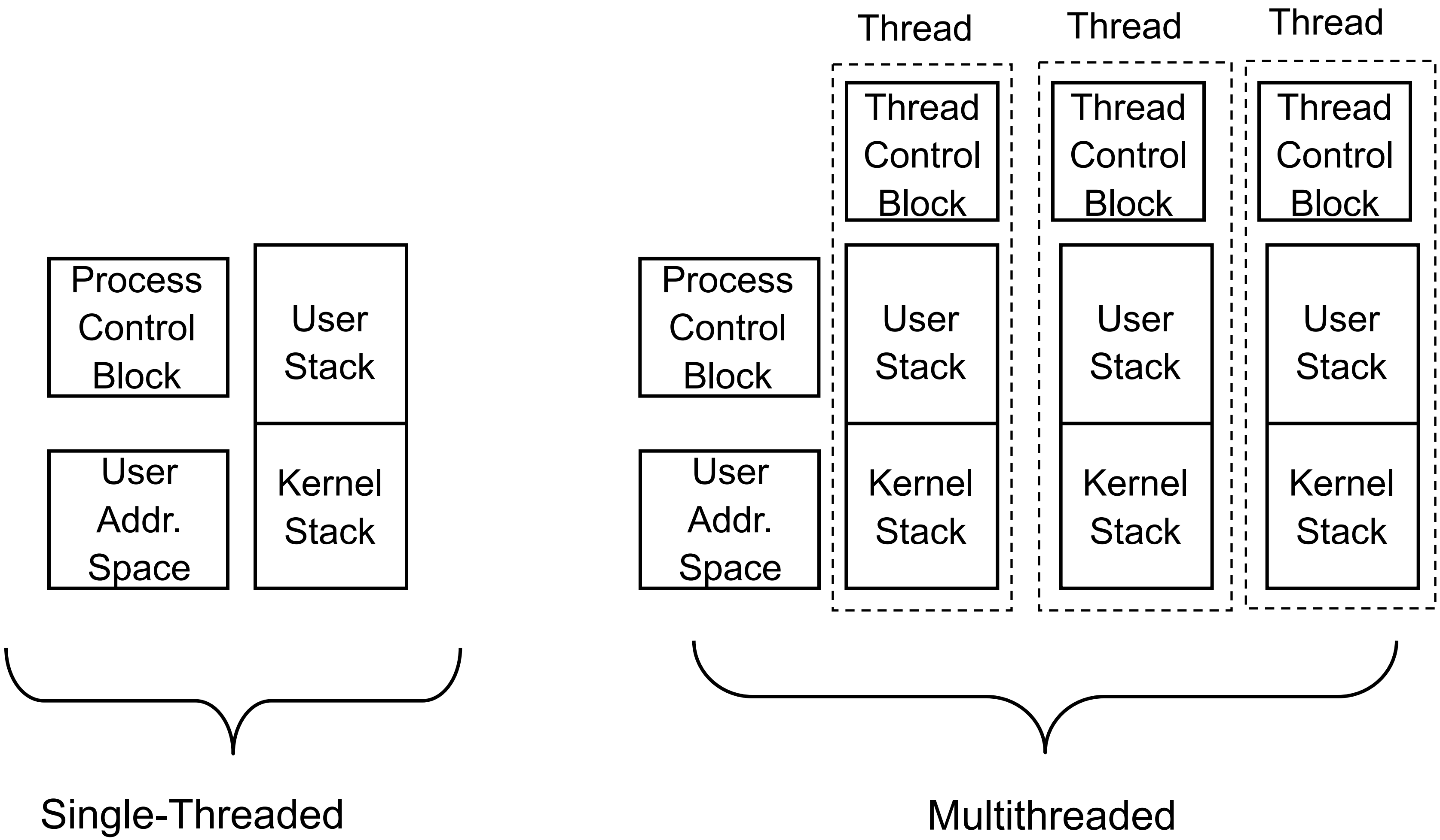
- Has a virtual address space which holds the process image

- Protected access to files, I/O resources and other processes

Thread

- has execution state (running, ready, *etc.*)
- thread context saved when thread not running
- has an execution stack
- some per-thread static storage for global variables
- access to the entire memory and all resources of its process
  - all threads of a process share this

Uni-, Multi-threaded Process Models



Benefits of Threads

- process = a heavy-weight entity
  thread = a lighter weight entity
  - Why?
  - less time needed
    - to create a new thread than a process
    - to terminate a thread than a process
    - to switch between two threads within the same process
- since threads within the same process share memory and files, they can communicate with each other without involving the kernel
  - Also, cache and main memory performance benefits
- penalty: lesser inter-thread protection

Thread States and Operations

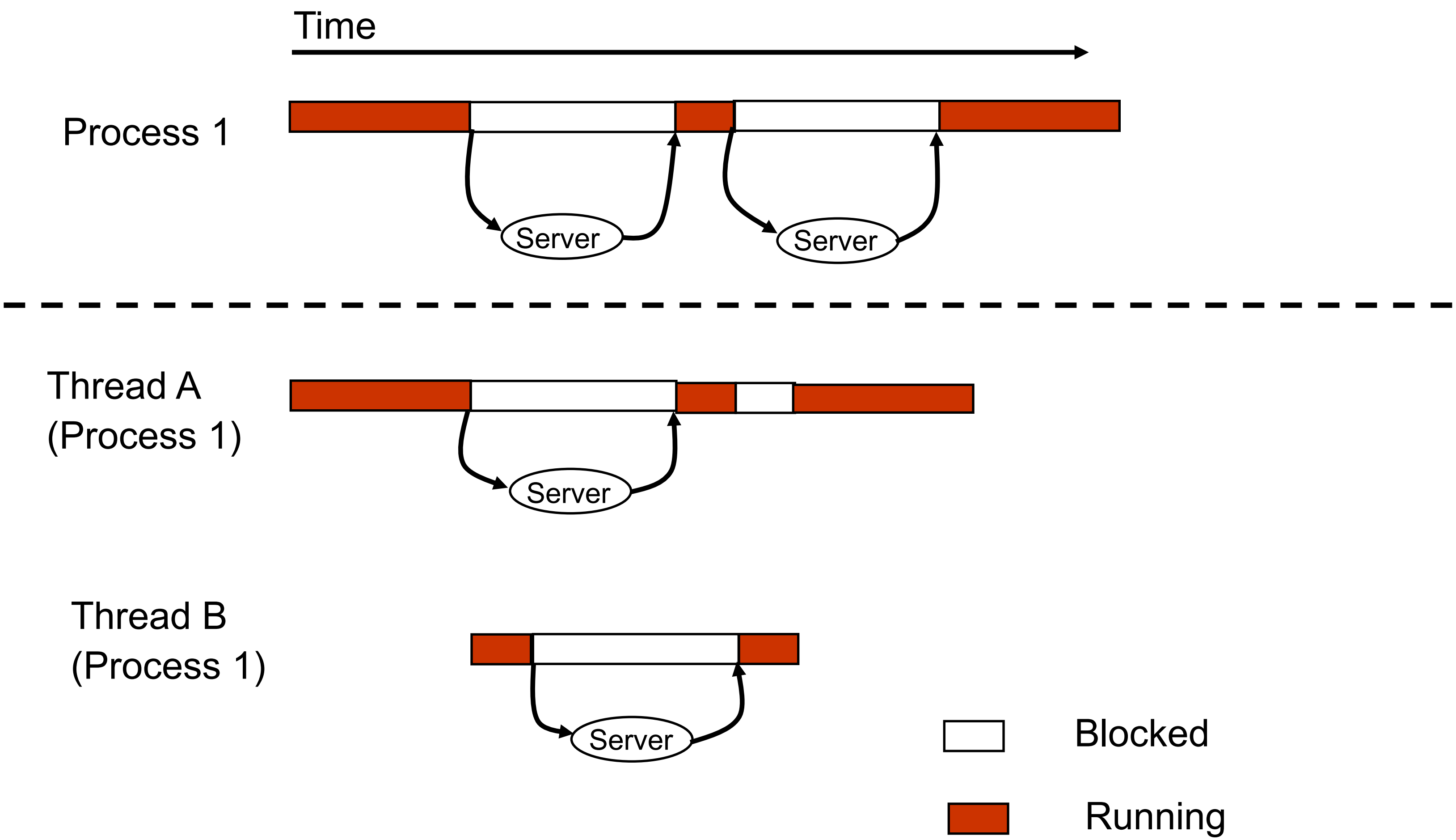
- key thread states: running, ready, blocked
- basic thread operations
  - spawn
    - spawn another thread
  - block
  - unblock
  - finish
    - deallocate space for register context and stacks

BIG QUESTION: if thread blocks, does entire process block???

Thread-Process Coupling

- Process $\rightarrow$ Thread
  - suspending a process involves suspending all threads of the process
  - termination of a process terminates all threads within the process
- Thread $\rightarrow$ Process
  - relationship between thread blocking and process blocking; two alternatives

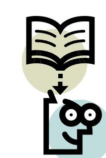
Example: RPC



Time
Process 1
Server
Server
Thread A
(Process 1)
Server
Thread B
(Process 1)
Server
Blocked
Running

Thread Design Alternatives

- **User-level threads**
  - all thread management is done by the application process
  - kernel is not aware of the existence of threads
- **Kernel-level threads**
  - kernel maintains context information for the threads (and the process)
  - scheduling is done by kernel on a thread basis
  - W2K, Linux, and OS/2 are examples of this approach



Combined Approaches

- example: Solaris
- both kernel and user-level threads (KLTs and ULTs)
- one or more ULTs mapped onto one KLT
- most activity at user level
  - thread creation is done completely at user level
  - also, the bulk of scheduling and synchronization of threads done at user level
- programmer can define the degree of parallelism in the process execution
  - state how many KLTs will the process run with

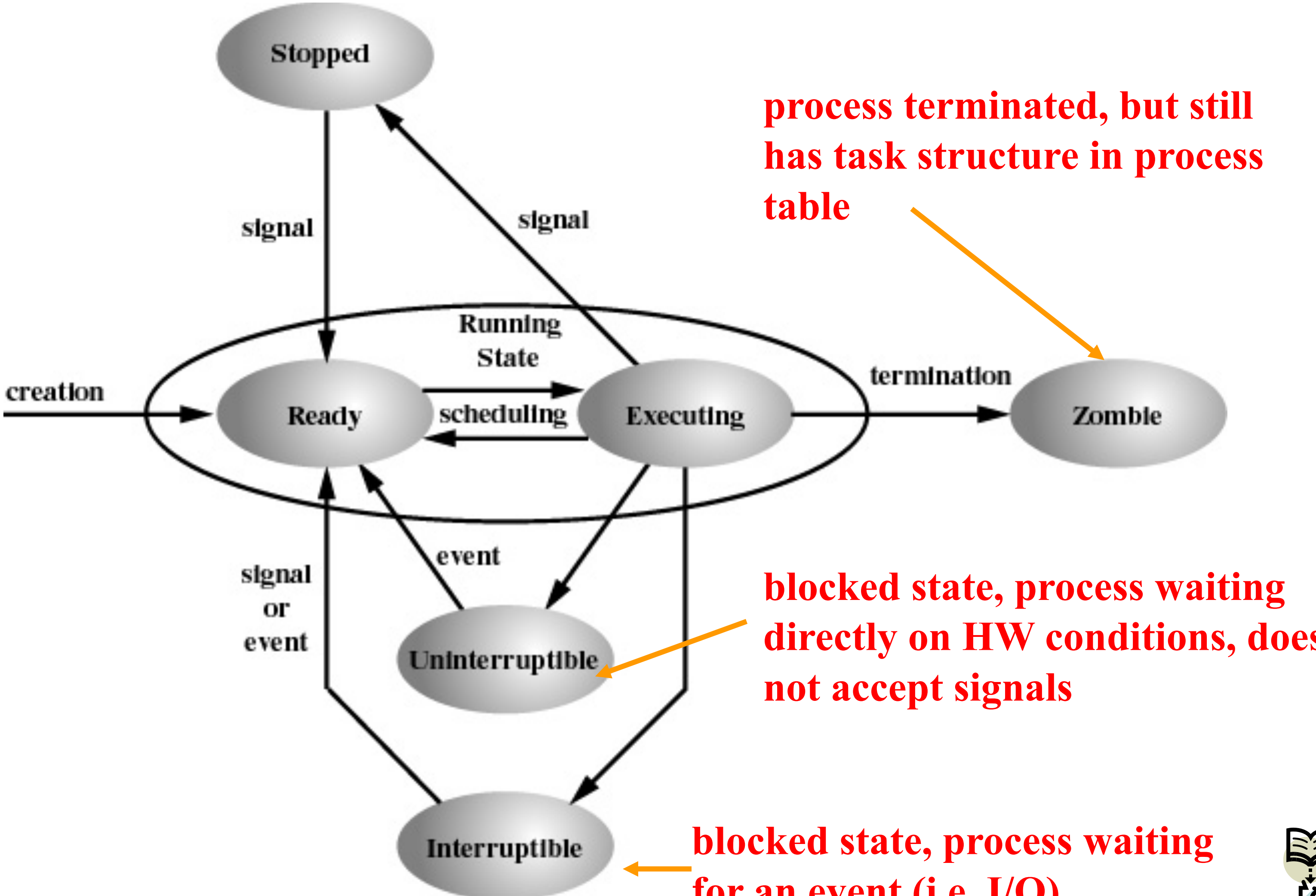


Figure 4.18 Linux Process/Thread Model

Inter-Process Communication and Concurrency

- Desire: communication among processes
  - Why?
    - Sharing resources
      - *E.g.*, printing: need application to pass item to printer daemon
    - Work together to solve a bigger problem
      - *E.g.*, "`ls | wc -l`"
- Two issues:
  1. Mechanism
  2. Synchronization
     1. Not tromping on each other
     2. Correct sequencing
- Implications on allocation of processor time

Mechanisms [1]

- Shared Memory
  - With processes, need explicit API
    - shm_open()
    - shmget()
    - shmctl()
    - shmat()
    - shmdt()
    - mmap()
    - ….
  - With threads, operating in a shared-memory space

Mechanisms [2]

- Message Passing
  - Named pipes
  - Send
  - Recv
  - Recvfrom
  - Putmsg
  - Recvmsg
  - ….

Difficulties Arising From Concurrency

- need to control sharing of global resources

- need to manage allocation of resources

- need to spend more resources on debug & test
  - bugs only show under some scenarios which are difficult to identify, reproduce
  - consequently, bugs are difficult to locate

A Simple Example

```
void echo()
{
   chin = getchar();
   chout = chin;
   putchar(chout);
}
```

- **echoing of keyboard input is common operation in OS**
- `echo()` **is a kernel routine**

A Simple Example

Process P1

```
//invokes echo()
chin = getchar();
.
chout = chin;
putchar(chout);
.
.
```

Process P2

```
 .
 //invokes echo()
   chin = getchar();
   chout = chin;
    .
   putchar(chout);
    .
```