Concurrency: Running Together

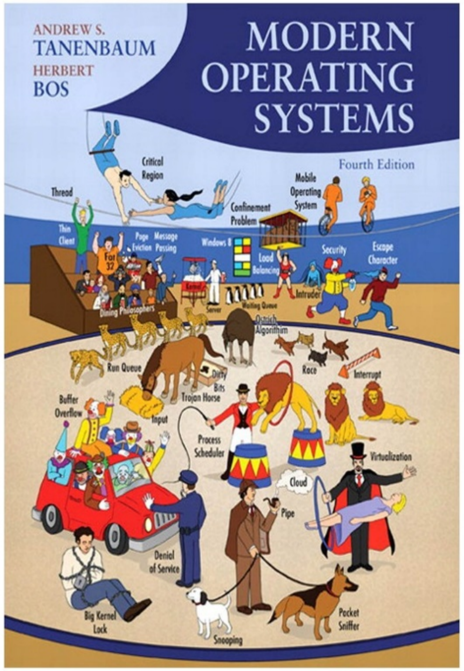
Methods & Tools for Software Engineering (MTSE)
Fall 2019

Arie Gurfinkel

UNIVERSITY OF WATERLOO

References

- (not comprehensive!)
- Modern Operating Systems by Andrew S. Tanenbaum, 4th Edition
  - Section 2.1.7
  - Sections 2.2.3 & 2.2.4
  - Sections 2.3.1 & 2.3.2 & 2.3.3
  - Sections 2.3.5 & 2.3.6
  - Section 6.2
- Slides & Demo credit:
  - Carlos Moreno (cmoreno@uwaterloo.ca)
  - Reza Babaee



MULTIPROGRAMMING

Multiprogramming

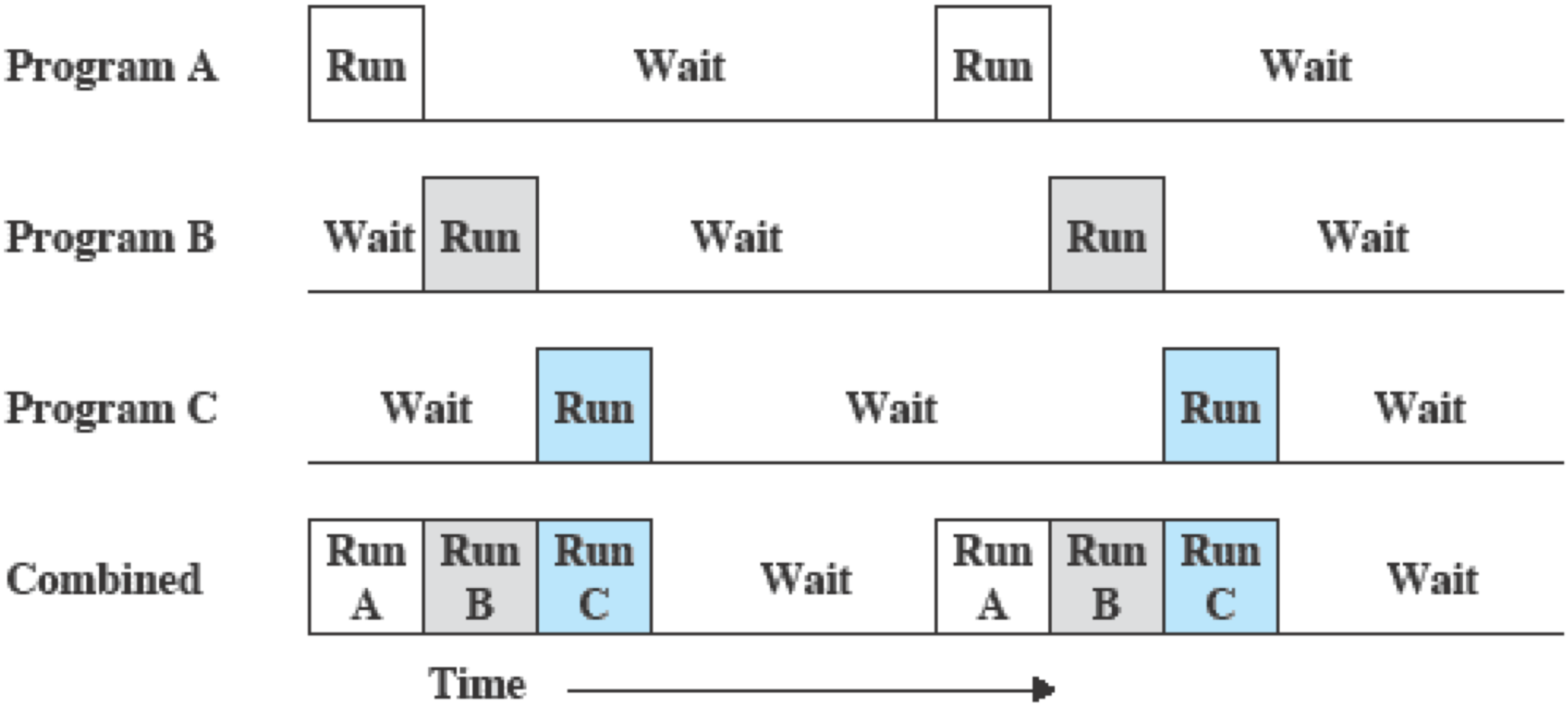
Concurrent execution of multiple tasks (e.g., processes)

- Each task runs as if it was the only task running on the CPU.

Benefits:

- When one task needs to wait for I/O, the processor can switch to the another task.
- (why is this potentially a *huge* benefit?)

Multiprogramming



(c) Multiprogramming with three programs

Multiprogramming

Example / case-study:

- Demo of web-based app posting jobs and a simple command-line program processing them.
  - Can run multiple instances of the job processing program.
  - Or we can have the program use **`fork()`** to spawn multiple processes that work concurrently

MULTITHREADING

Multithreading: Process versus Thread

Process provides an execution context for the program

- **unit of ownership**
- Memory, I/O resources, console, etc
- Process pretends like it is a single entity controlling the execution environment
- Inter Process Communication (IPC) is “like” communicating between individual machines (but connected with super-fast network)

Thread represent a single execution unit (i.e., CPU)

- unit of scheduling
- ancient time: a process has one thread running on one physical CPU
- old time: a process has many threads sharing one physical CPU
- today: a process has many threads sharing many physical CPUs (multicore)
- all threads of a process share the same memory space!

Threads: Programmer's Perspective

A thread is a function that is ran concurrently with other functions

- It is like fork() followed by a call to a child process function
- Except: no new process is created. The new thread can access all the data of the **current** process

```
void * foo(void*) {...}
void * bar(void*) {...}

int main(void) {
  pthread_t t1, t2;
  void *data;
  ...
  pthread_create(&t1, NULL, foo, data);
  pthread_create(&t2, NULL, bar, data);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
}
```

UNIVERSITY OF WATERLOO

Threads: Programmer's Perspective

A thread is a function that is ran concurrently with other functions

- It is like fork() followed by a call to a child process function
- Except: no new process is created. The new thread can access all the data of the **current** process

```
void * foo(void*) {...}
void * bar(void*) {...}

int main(void) {
  pthread_t t1, t2;
  void *data;
  ...
  pthread_create(&t1, NULL, foo, data);
  pthread_create(&t2, NULL, bar, data);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
}
```

Code that will execute concurrently

Start of concurrent execution

Main thread waits for others to finish

Multithreading

Example/demo:

- With the multithreading demo, we'll look at a different application/motivation for the use of concurrency: performance boost through parallelism.
  - Possible when we have multiple CPUs (e.g., multicore processors)
  - Important to have multiple CPUs when the application is CPU-bound.

CONCURRENCY ISSUES

Race Condition

A situation where concurrent operations access data in a way that the outcome depends on the order (the timing) in which operations execute.

- Doesn't necessarily mean a bug! (like in the threads example with the linked list)

- In general it constitutes a bug when the programmer makes any assumptions (explicit or otherwise) about an order of execution or relative timing between operations in the various threads.

Race Condition – Example

Race condition:

Assume that `x` is a variable shared between the two threads

Thread 1:

```
x = x + 1;
```

Thread 2:

```
x = x - 1;
```

(what's the implicit assumption a programmer could make?)

Race Condition – Example

Race condition:

Thread 1:

```
x = x + 1;
```

Thread 2:

```
x = x - 1;
```

In assembly code:

Thread 1:

```
R1 ← x
inc  R1
R1 → x
```

Thread 2:

```
R1 ← x
dec  R1
R1 → x
```

Race Condition – Example

And this is how it could go wrong:

Thread 1:

```
x = x + 1;
```

Thread 2:

```
x = x - 1;
```

In assembly code:

Thread 1:

```
R1 ← x
inc  R1
R1 → x
```

Thread 2:

```
R1 ← x
dec  R1
R1 → x
```

UNIVERSITY OF WATERLOO

Atomicity/ Atomic Operations

Atomicity is a characteristic of a fragment of a program that exhibits an observable behavior that is non-interruptible – it behaves as if it can only execute entirely or not execute at all, such that no other threads deal with any intermediate outcome of the atomic operation.

- Non-interruptible applies in the context of other threads that deal with the outcome of the operation, or with which there are race conditions.
- For example: in the pthreads demo, if the insertion of an element in the list was atomic, there would be no problem.

Atomicity/ Atomic Operations – Examples

- Renaming / moving a file with
  int rename (const char * old, const char * new);
  Any other process can either see the old file, or the new file – not both and no other possible "intermediate" state.

- **open**ing a file with attributes **O_CREAT** and **O_EXCL** (that is, creating a file with exclusive access). The operation atomically attempts to create the file: if it already exists, then the call returns a failure code.

UNIVERSITY OF WATERLOO

Mutual Exclusion

Atomicity is often achieved through mutual exclusion – the constraint that execution of one thread excludes all the others.

- In general, mutual exclusion is a constraint that is applied to sections of the code.
- For example: in the pthreads demo, the fragment of code that inserts the element to the list should exhibit mutual exclusion: if one thread is inserting an element, no other thread should be allowed to access the list
  - That includes main, though not a problem in this particular case (why?)

Mutual Exclusion – How?

Attempt #1: We disable interrupts while in a critical section (and of course avoid any calls to the OS)

- There are three problems with this approach
  - Not necessarily feasible (privileged operations)
  - Extremely inefficient (you're blocking everything else, including things that wouldn't interfere with what your critical section needs to do)
  - *Doesn't always work!!* (keyword: multicore)

Mutual Exclusion – How?

Attempt #2: We place a flag (sort of telling others "don't touch this, I'm in the middle of working with it).

```
int locked;  // variable shared between threads
...
if (! locked)
{
    locked = 1;
    // insert to the list (critical section)
    ...
    locked = 0;
}
```

Why is this flawed? (there are *several* issues)

Mutual Exclusion – How?

One of the problems: does not really work!

This is what the assembly code could look like:

```
R1 ← locked
tst R1
brnz somewhere_else
R1 ← 1
R1 → locked
```

Mutual Exclusion – How? → Mutex

A mutex (for MUTual EXclusion) provides a clean solution: In general we have a variable of type mutex, and a program (a thread) attempts to *lock* the mutex. The attempt *atomically* either succeeds (if the mutex is unlocked) or it *blocks* the thread that attempted the lock (if the mutex is already locked).

- As soon as the thread that is holding the lock unlocks the mutex, this thread's state becomes ready.

Mutual Exclusion – How? → Mutex

Using a Mutex:

```
lock (mutex)
```
critical section
```
unlock (mutex)
```

Mutual Exclusion – How? → Mutex

Using a Mutex:

With POSIX threads (pthreads):

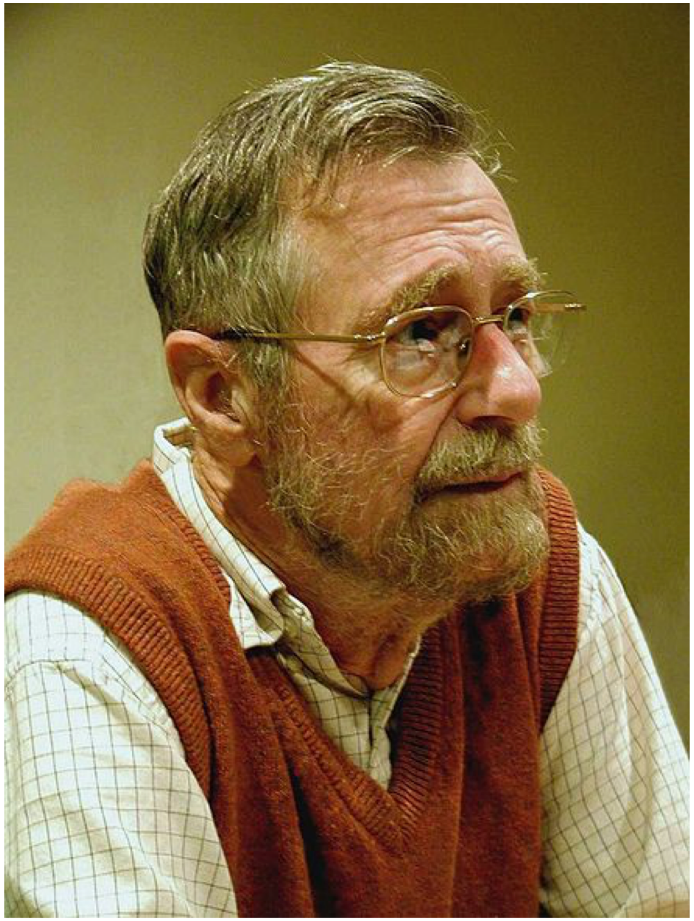
```
pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;
...
pthread_mutex_lock (&mutex);
... critical section
pthread_mutex_unlock (&mutex);
```

Mutual Exclusion – How? → Mutex

- One issue is that POSIX only defines mutex facilities for threads --- not for processes!

- We could still implement it through a "lock file" (created with **open** using flags **O_CREAT** and **O_EXCL**)
  - Not a good solution (it *does* work, but is has the same issues as the lock variable example)

Edsger W. Dijkstra



(image courtesy of wikipedia.org)

Another synchronization primitive

SEMAPHORES

UNIVERSITY OF WATERLOO

Semaphore: Definition

Semaphore is a counter with the following properties

- Initialized with a non-negative value
  - `count := 2`
- Atomic increment
  - `count := count + 1`
- Atomic decrement
  - `count := count - 1`

Operations

`wait` operation decrements count and causes caller to block if count becomes negative (if it was 0)

`signal` (or `post`) operation increments count. If there are threads blocked (waiting) on this semaphore, it unblocks one of them.

Example

Producer / consumer with semaphores

```
semaphore items = 0;
mutex_t mutex;  // why also a mutex?
```

```
void producer()
{
    while (true)
    {
        produce_item();
        lock (mutex);
        add_item();
        unlock (mutex);
        sem_signal (items);
    }
}
```

```
void consumer()
{
    while (true)
    {
        sem_wait (items);
        lock (mutex);
        retrieve_item();
        unlock (mutex);
        consume_item();
    }
}
```

Implementing Mutex with a Semaphore

Interestingly enough – Mutexes can be implemented in terms of semaphores!

```
semaphore lock = 1;

void process ( ... )
{
    while (1)
    {
        /* some processing */
        sem_wait (lock);

        /* critical section */

        sem_signal (lock);
        /* additional processing */
    }
}
```

Exercise

Producer / consumer with semaphores only

POSIX Semaphores

- Defined through data type `sem_t`
- Two types:
  - Memory-based or unnamed (good for threads)
  - Named semaphores (system-wide — good for processes synchronization)

POSIX Semaphores – unnamed

- Declare a (shared – possibly as global variable) `sem_t variable`
- Give it an initial value with `sem_init`
- Call `sem_wait` and `sem_post` as needed.

```
sem_t items;
sem_init (&items, 0, initial_value);
// ...
sem_wait (&items)  //or  sem_post (&items)
```

POSIX Semaphores – named

– Similar to dealing with a file: have to “open” the semaphore – if it does not exist, create it and give it an initial value.

```
sem_t * items = sem_open (semaphore_name, flags,
                          permissions, initial_value);
// should check if items == SEM_FAILED

// ...

sem_wait (items)  or  sem_post (items)
```

POSIX Semaphores – Example

Producer-consumer:

- We'll work on the example of the web-based demo as a producer-consumer with semaphores.
- Granularity for locking?
  - Should we make the entire `process_requests` a critical section?
    - Clearly overkill! No problem with two separate processes working each on a different file!
    - We can lock the file instead — no need for a mutex, since this is a *consumable* resource.
    - For a reusable resource, we'd want a mutex – block while being used, but then want to use it ourselves!

STARVATION & DEADLOCKS

Starvation

- One of the important problems we deal with when using concurrency:
- An otherwise ready process or thread is deprived of the CPU (it's *starved*) by other threads due to, for example, the algorithm used for locking resources.
  - Notice that the writer starving is *not* due to a defective scheduler/dispatcher!

Deadlocks

- Consider the following scenario:
- A Bank transaction where we transfer money from account A to account B and vice versa at the same time
- Clearly, there is a (dangerous) race condition
  - Want granularity — can not lock the entire bank so that only one transfer can happen at a time
  - We want to lock at the account level:
    - Lock account A, lock account B, then proceed!

Deadlocks – cont.

- Problem with this?
- Two concurrent transfers — one from Account A to Account B ($100), and the other one from account B to account A ($300).
  - If the programming is written as:
    Lock source account
    Lock destination account
    Transfer money
    Unlock both accounts

Deadlocks – cont.

- Problem with this?
- Two concurrent transfers — one from Account A to Account B ($100), and the other one from account B to account A ($300).
  - Process 1 locks account A, then locks account B
  - Process 2 locks account B, then locks account A

UNIVERSITY OF WATERLOO

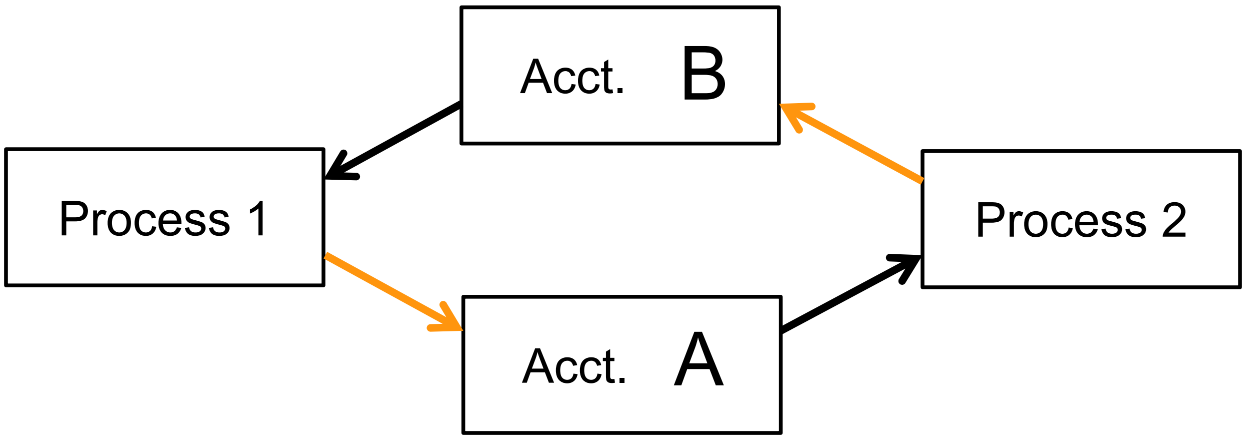
Deadlocks – cont.

- What about the following interleaving?
  - Process 1 locks account A
  - Process 2 locks account B
  - Process 1 attempts to lock account B (blocks)
  - Process 2 attempts to lock account A (blocks)

- When do these processes unblock?

- Answer: under some reasonable assumptions, *never!*

Deadlocks – cont.



- Solution in this case is really simple:
  - Lock the resources in a given order (e.g., by ascending account number).

INTER-PROCESS COMMUNICATION

Shared Memory

- Mechanism to create a segment of memory and give multiple processes access to it.
- **shmget** creates the segment and returns a handle to it (just an integer value)
- **shmat** creates a logical address that maps to the beginning of the segment so that this process can use that memory area
  - If we call **fork()**, the shared memory segment is inherited shared (unlike the rest of the memory, for which the child gets an independent copy)

Message Queues

- Mechanism to create a queue or "mailbox" where processes can send messages to or read messages from.
- **mq_open** opens (creating if necessary) a message queue with the specified name.
- **mq_send and mq_receive** are used to transmit or receive (receive by default blocks if the queue is empty) from the specified message queue.
- Big advantages:
  - Allows multiple processes to communicate with other multiple processes
  - Synchronization is somewhat implicit!