Race Conditions and Deadlocks

Modern Operating Systems, by Andrew Tanenbaum — Chap 2.3 & 6

Operating Systems: Three Easy Pieces (a.k.a. the OSTEP book) — Chap 26 & 28

Parallelism Continuum

Sequential

- Many things one after another

CPU 1
Task 1
Task 1
Task 2
Task 2

Concurrent

- Interleaved resource usage

CPU 1
Task 1
Task 2
Task 1
Task 2

Parallel

- Simultaneous resource usage

CPU 1	CPU 2
Task 1	Task 2
Task 1	Task 2

Parallel is a subset of Concurrent

Concurrency and Synchronization

- Independent tasks can execute independently
- Many tasks have dependencies
  - Task 1 : Select all CS-550 students from enrollment database
  - Task 2 : Sort CS-550 students by last name
- Concurrent tasks may need to synchronize (communicate)
  - Not always, but probably more than once
- Synchronization requires access to shared resources
  - Shared memory (buffers), Pipes, Signals, etc.

Critical Section

- A section of code that modifies or accesses a resource that is shared with another task
  - Also called "critical region"

- Examples:
  - A piece of code that reads from or writes to shared memory
  - Code that modifies or traverses a shared linked list

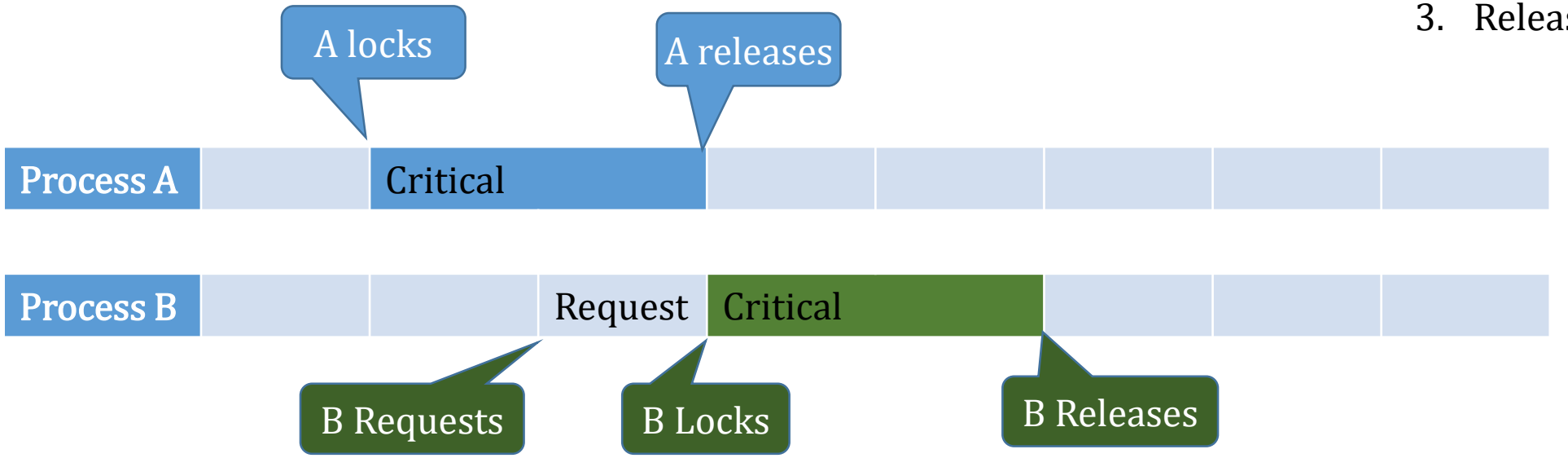
Race Conditions

- Cases where errors can be introduced because two programs execute critical sections concurrently
- Example: Task 1 writes CS-550 student name to shared memory while Task 2 is reading old CS-550 student name
- Prevent races by locking resources to a specific task

Mutual Exclusion

Don't allow two or more processes to execute critical section (on same resource) concurrently

Steps
1. Acquire lock
2. Critical Section
3. Release lock



Deadlock

- Two or more processes cannot make progress indefinitely because they are waiting for each other to do something

```
Task 1
while (new student) {
        wait for name lock
        lock name
        update name
        wait for count lock
        lock count
        update count
        release count lock
        release name lock
}
```

```
Task 2
while (1) {
        wait for count lock
        lock count
        if (count>used) {
                wait for name lock
                lock name
                use name
                release name lock
        }
        release count lock
}
```

Correct Mutual Exclusion

1. No two processes are simultaneously in their critical sections
2. No assumptions made about speed or numbers of CPUs
3. No process waits forever to enter its critical section (No deadlocks)
4. No process running outside its critical region may block another process running in its critical section

- 1. and 2. enforced by the operating system lock implementation
- 3. and 4. must be enforced by the programmer using locks!

Mutual Exclusion for Readers and Writers

- General rule: If a thread is writing to a shared resource, other threads should not read from or write to the same resource

Thread 1	Thread 2	Allowed
Read	Read	Allowed
Read	Write	Disallowed
Write	Read	Disallowed
Write	Write	Disallowed

- Exceptions may be allowed for special lockless data structures

Types of Locks

- Blocking Locks
  - No progress until lock is available
  - OS may swap out process/task until lock is available
- Non-Blocking Locks
  - Allows status query
  - Program decided how to proceed
- Spin lock
  - A non-blocking lock of the form: while(lockAvailable()==False) {}

Blocking Locks

- Usage:

```
lock(resource); // Block until resource is available, then lock it
critical_section(resource); // access or modify shared resource
unlock(resource); // Allow others to use the resource
```

- Advantage: Simple to use, assumes locking ultimately succeeds
- Disadvantages:
  - Unpredictable blocking duration
  - Overhead to swap out and swap in if lock becomes available soon after blocking

Non-Blocking locks

- Usage:

```
if (trylock(resource)==available) {
   critical_section(resource);
   unlock(resource);
} else {
   plan_b();
}
```

- Advantage: No unpredictable wait time
- Disadvantage: Need a "Plan B" to handle locking failure

Spin Locks

- Usage:

```
spin_lock(resource);
critical_section(resource);
unlock(resource);
```

- Advantage: Very efficient if lock released quickly
  - Implies short critical sections so that locks are released quickly
- Disadvantage: Wastes CPU cycles
  - If critical sections are long, wastes lots of CPU cycles
  - Counter productive on uniprocessor machines

Locking Recommendations

1. ***Associate locks with shared resource, NOT code***
   - e.g. lock protects a linked list, not insert() and remove() functions
2. ***Guard each shared resource with a separate lock***
   - Improves concurrency
   - Allows critical section code to be used on multiple resources
     - e.g. list1 is guarded by lock1, list2 is guarded by lock2, shared insert() and remove()
   - Be careful about deadlock!

- OS cannot enforce these recommendations
  - OS doesn't understand the application-level semantics

Deadlock on Multiple Resources

Two processes, P_1 and P_2, both need two resources controlled by locks L_1 and L_2

Deadlock	
P_1	P_2
lock(L_1)	
	lock(L_2)
lock(L_2)	
	lock(L_1)

Solution: Lock Ordering

Define order (L_1, L_2, ...)
Acquire locks in order
Release locks in reverse

No Deadlock	
P_1	P_2
lock(L_1)	
	lock(L_1)
lock(L_2)	
critSect	
unlock(L_2)	
unlock(L_1)	
	lock(L_2)
	critSect
	unlock(L_2)
	unlock(L_1)

Generalizing Lock Ordering

- Given n locks, L_1, L_2, ..., L_n and k processes, P_1, P_2, ... , P_k

  All processes must acquire any subset of locks in sorted order

(A process doesn't need to acquire ALL locks, but whatever locks it does acquire must be acquired in sort order)

- Example, n=10
  - Allowed: P_i acquires L_1, then L_5, then L_{10}
  - Allowed: P_j acquires L_1, then L_3, then L_{10}
  - Not Allowed: P_k acquires L_5, then L_1, then L_2

Priority Inversion

Three processes using priority based scheduling: P_h, P_m, P_l

Priority Inversion		
P_l	P_m	P_h
lock(L)	idle	idle
critSect	ready	ready
		lock(L)

Solution: Priority Inheritance

Temporarily increase priority of P_l to high priority so P_h can obtain lock

Priority Inheritence		
$P_{l(h)}$	P_m	P_h
lock(L)		
critSect		
		lock(L)
critSect		
unlock(L)		
		critSect
		unlock(L)

Spinlocks on Multi-CPU

Single CPU

Slowlock

P_1/CPU_1	P_2/CPU_1
lock(L_1)	
	spinlock(L_1)
unlock(L_1)	

Deadlock

P_1/CPU_1	P_2/CPU_1
lock(L_1)	
	spinlock(L_1)

CPU Intensive
High Priority

Multi-CPU

No Deadlock

P_1/CPU_1	P_2/CPU_2
lock(L_1)	
	spinlock(L_1)
unlock(L_1)	

Threads must use interrupt disabling version of spinlock (spinlock_irqsave)… Why?

Problem: Interrupts and Deadlocks

- Interrupts invoke Interrupt Service Routines (ISR) in the kernel
  - ISR must process interrupt quickly and return
  - ISR must never block or spin on a lock
  - What if ISR needs a lock to process the interrupt?

Interrupt Deadlock	
P	ISR
lock(L)	
critSect	interrupt
	lock(L)

Interrupt Deadlock Solutions

1. Don't lock in ISR! – defer locking work to thread context
   - See "softirqs" in Linux
2. If you must lock, use non-blocking lock

```
if (tryLock(L)==available) { lock(L); do work }
else { handle unavailable lock (e.g. write message }
```

3. Disable interrupts before locking in process
   - No deadlock because ISR can't run when process locks resource
   - When ISR runs, it assumes resource is unlocked
   - Disabling interrupts for extended periods is not a good idea