# Concurrency

- Race Conditions and Deadlocks

Kartik Gopalan

Chapters 2 (2.3) and 6
Tanenbaum's Modern OS

# Sequential

- Loosely, doing many things, but one after another
    - E.g. Finish one assignment, then another

- For example, two tasks executed on one CPU one after another.

Time

| CPU 1 |
|---|
| Task 1 |
| Task 1 |
| Task 2 |
| Task 2 |

# Concurrent

- Loosely, concurrency is “juggling” many things within a time window.
    - E.g. switching your attention back-and-forth between two different assignments.

- For example, two tasks share a single CPU over time.

| CPU 1 |
|---|
| Task 1 |
| Task 2 |
| Task 1 |
| Task 2 |

# Parallel

- Loosely, parallelism is doing many things **simultaneously**.
    - E.g. working on the computer and chewing gum at the same time.

- Parallelism is a subset of concurrency.
    - All parallelism is concurrency
    - But not all concurrency is parallelism.

- For example, two threads executing on two different CPUs **simultaneously.**

| CPU 1 | CPU 2 |
|---|---|
| Task 1 | Task 2 |
| Task 1 | Task 2 |

# Concurrency and Synchronization

- Concurrent tasks may either execute independently
- Or, concurrent tasks may need to synchronize (communicate) now and then
- Synchronization requires access to **shared resources**
  - Shared memory (buffers)
  - Pipes
  - Signals, etc

# Critical Section

- Also called critical region.

- A section of code in a concurrent task that modifies or accesses a resource shared with another task.

- Examples
  - A piece of code that reads from or writes to a shared memory region

  - Or a code that modifies or traverses a shared linked list.

# Race Condition and Deadlocks

- Race Condition
  - Incorrect behavior of a program due to concurrent execution of critical sections by two or more threads.
  - E.g. if thread 1 deletes an entry in a linked list while thread 2 is accessing the same entry.

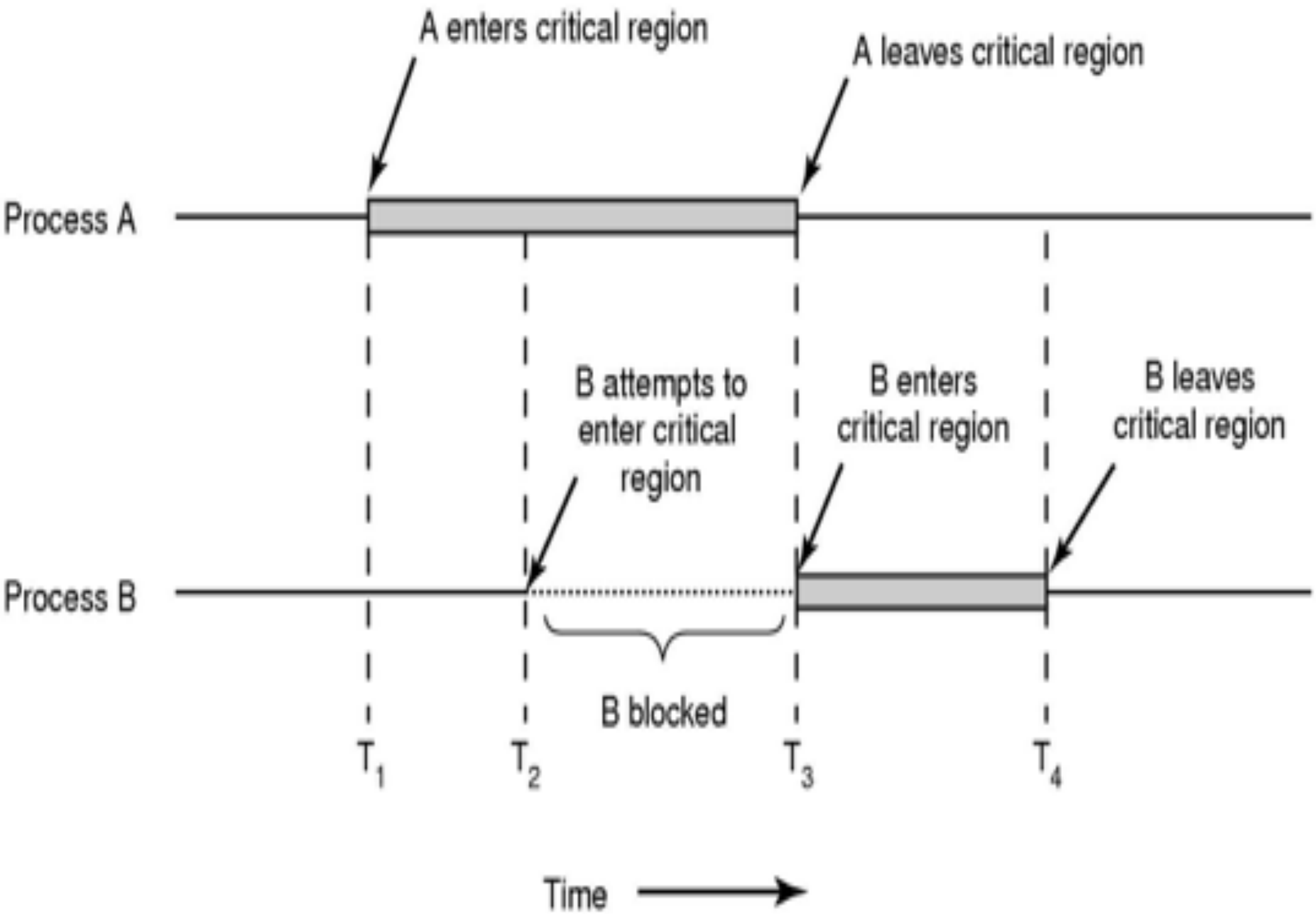
- Deadlocks
  - When two or more processes stop making progress *indefinitely* because they are all waiting for each other to do something.
  - E.g.
    - If process A waits for process B to release a resource, and
    - Process B is waiting for process A to release another resource at the same time.
    - In this case, neither A not B can proceed because both are waiting for the other to proceed.

# Race Conditions and Locking

- Race Condition: Incorrect behavior of a program due to concurrent execution of critical sections by two or more threads.

# Solution: Mutual Exclusion

Don't allow two or more processes to execute their critical sections concurrently (on the same resource).




<u>Steps</u>

1. Acquire Lock
2. Critical Section
3. Release Lock

# Conditions for correct mutual exclusion

1. No two processes are simultaneously in the critical region
2. No assumptions are made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process running in the critical region
4. No process must wait forever to enter its critical region
   - Waiting forever indicates a deadlock

(1) and (2) are enforced by the operating system’s implementation of locks

- Programmers assume that locks satisfy (1) and (2)

(3) and (4) have to be ensured by the programmer using the locks.

- OS cannot enforce these.

# Mutual Exclusion among Readers and Writers

- General rule
  - If any thread is writing to a shared resource, other threads are disallowed from reading or writing to the same resource.

| Thread 1 | Thread 2 | Allowed/Disallowed |
|---|---|---|
| Read | Read | Allowed |
| Read | Write | Disallowed |
| Write | Read | Disallowed |
| Write | Write | Disallowed |

- Exceptions may be allowed for special types of lockless data structures.

# Blocking Locks

- Give up CPU till lock is available

```
while(lock unavailable)
  yield CPU to others; // or block till lock available
return success;
```

- Usage:

```
Lock(resource); // Claim a shared resource
Execute Critical Section;//access or modify the shared resource
Unlock(resource); // unclaim shared resource
```

- Advantage: Simple to use. Locking always succeeds…ultimately.
- Disadvantage: Blocking may be indefinite.
  - Process is moved out of “Running” state to “Blocked” state.
  - Also, too much overhead in getting back to running state if lock may be available soon.
    - running—>blocked—>ready—>running

# Non-blocking locks

- Don't block if lock is unavailable

```
if(lock unavailable)
  return failure;
else
  return success
```

- Usage

```
if(TryLock(resource) == success)
  Execute Critical Section;
  Unlock(resource);
else
  Do something else; // plan B
```

- Advantage: No unbounded blocking
- Disadvantage: Need a "plan B" to handle locking failure

# Spinlocks

- Don't block. Instead, constantly poll the lock for availability.

```
while (lock is unavailable)
    continue; // try again
return success;
```

- Usage: Just like blocking locks

```
SpinLock(resource);
Execute Critical Section;
SpinUnlock(resource);
```

- Advantage
  - Very efficient with short critical sections
    - if you expect a lock to be released quickly
- Disadvantage
  - Doesn't yield the CPU and burns CPU cycles
    - Bad if critical sections are long.
  - Efficient only if machine has multiple CPUs.
    - Counterproductive on uni-processor machines

# Best practices for locking

1. ***Associate locks with shared resources, NOT code.***
   - E.g. a lock is for protecting a linked list
   - NOT for protecting insert() and remove() functions.
   - That way, you can use the same critical sections to operate on different shared resources having different locks.

2. ***Guard each shared resource by a separate lock***
   - to improve concurrency
   - E.g. Linked list 1 should be guarded by Lock 1
   - Linked List 2 by Lock 2
   - and so on.

- OS cannot enforce these properties
  - Up to the programmer to ensure these properties

# Deadlocks

- When two or more processes stop making progress *indefinitely* because they are all waiting for each other to do something.

# Deadlock when using multiple locks

- Say you have two processes P1 and P2
- Both need to acquire two locks L1 and L2 to access a resource.
- **Problem: Deadlock**
  - P1 acquires L1
  - P2 acquires L2
  - P1 tries to acquire L2 and blocks
  - P2 tries to acquire L1 and blocks
  - We have a deadlock!

- **Solution**: **Lock Ordering**
  - Sort the locks in a fixed order (say L1 followed by L2)
  - Always acquire locks in the sorted order.
- Lock ordering example:
  - P1 acquires L1
  - P2 tries to acquire L1 and blocks
  - P1 acquires L2
  - P1 executes critical section
  - P1 releases L2
  - P1 releases L1
  - P2 wakes up
  - P2 acquires L1
  - P2 acquires L2
  - P2 executes critical section
  - P2 releases L2
  - P2 releases L1
  - No deadlock!

# Generalizing the lock-ordering solution

- Given
  - N Locks: L1, L2, …, LN
  - K Processes: P1, P2, …, Pk

- A process must acquire any subset of locks in sorted order
  - A process doesn't need to acquire ALL the locks.
  - But whatever locks it needs, it MUST acquire in sorted order.

- E.g. Assume N=10, i.e. you have 10 Locks
  - (Allowed) Pi acquires L1, then L5, then L10
  - (Allowed) Pj acquires L1, then L3, then L10
  - (NOT Allowed) Pk acquires L5, then L2, then L1

# Priority Inversion

- Say there are three processes using priority based scheduling.
  - Ph – High priority
  - Pm – Medium priority
  - Pl – Low priority

- Pl acquires a lock L
- Pl starts executing critical section
- Ph tries to acquire lock L and blocks
- Pm becomes “ready” and preempts Pl from the CPU.
- Pl might never exit critical section if Pm keeps preempting Pl
  - So Ph might never enter critical section

- **Problem: Priority Inversion**
  - A high priority process Ph is blocked waiting for a low priority process Pl
  - Pl cannot proceed because a medium priority process Pm is executing.

- **Solution: Priority Inheritance**
  - Temporarily increase the priority of Pl to HIGH PRIORITY
  - Pl will be scheduled and will exit critical section quickly
  - Then Ph can execute.

# Interrupts and Deadlocks — Problem

- Interrupts invoke interrupt service routines (ISR) in the kernel.
  - ISR must process the interrupt quickly and return.
  - So ISRs must never block or spin on a lock.
  - But what if ISRs need a lock to process the interrupt?

- The problem:
  - A kernel thread T acquires lock L
  - An interrupt fires and ISR preempts T
  - ISR tries to acquire L and blocks (or spins)
  - T is also blocked because ISR cannot return
  - Deadlock!

# Interrupts and Deadlocks — Solutions

1. Don’t lock in ISR!
    - Defer any locking work to thread context (softirqs in Linux)

2. If you must, use try_lock() instead of lock() in ISR
    - try_lock() = if lock is available then get it, else return with error.
    - Be ready to handle error return

3. Or disable interrupts in thread T before locking
    - If ISR cannot run when lock is acquired by T, then there’s no deadlock.
    - When ISR runs, it assumes that T doesn’t have the lock.
    - But, disabling interrupts too long is also not a good idea.

4. Or, on multi-CPU systems, use spinlocks, but carefully
    - ISRs on different CPUs can compete for locks without blocking
    - Threads must use interrupt disabling versions of spinlock (spinlock_irqsave). Why?