Section 3 : Concurrency/Synchronization

Synchronization Constructs

Mutex (Mutual Exclusion Lock)

A mutex is a simple lock that makes sure only one thread can enter the critical section at a time.

How does it work?

- A thread that wants to enter a critical section locks the mutex
- If another thread tries to lock it while its already taken, it has to wait
- Once the thread is done, it unlocks the mutex, letting another one in.

Ensures mutual exclusion. May cause busy waiting or put threads to sleep depending on implementation

Semaphores

Flexible synchronization tool based on an integer counter. Allows multiple threads to access shared resources up to a limit.

How does it work?

- wait() (P) decreases semaphore value, if value is already 0, thread blocks or spins (waits)
- signal() (also called V) increases the value, possibly waking up a waiting thread.

Two types

1. Counting Semaphore
   - Value can be greater than 1, used when you'd have multiple identical resources
2. Binary Semaphore
   - Value is either 0 or 1, acts exactly like a mutex

Analogy: Think of a parking lot

- wait() means you enter and use a spot (counter goes down)
- signal() means you leave, freeing a spot (counter goes up)
- If its full (value = 0), you have to wait until someone leaves

Classic Synchronization Problems

a. **Producer-Consumer (Bounded Buffer)**
   - Shared buffer between a producer (writes data) and consumer (reads it)
   - Use semaphores to track
     - Empty slots, Filled slots, Mutual exclusion for buffer access

b. **Readers-Writers**
   - Multiple readers can access data simultaneously
   - But only one write can update at a time - and no readers during writing

   Variations
   - First readers' problem (writer starvation possible
   - First writers' problem (reader starvation possible)

c. **Dining Philosophers**
   - Five philosophers sharing five forks
   - Must grab two forks (shared resources) to eat
   - Can lead to: deadlock, starvation, or livelock depending on implementation

Solutions to the critical section problem

a. **Software-only (Peterson's Algorithm)**

Peterson's algorithm is a class software based method that provides a solution to the critical section problem for two threads

Key Ideas

- Each thread has a flag to say "I'm interested in the critical section"
- A shared turn variable decides whose turn it is to enter the critical section
- Both threads cooperate and check these two variables before proceeding

Downside

- It relies on busy waiting, meaning the thread constantly checks a condition in a loop (also called a **spinlock**). This wastes CPU cycles while waiting

  Def - Spinlock - A type of lock here a thread waits in a loop (spins), constantly checking if it can enter the critical section.
  - It keeps checking a condition until the lock becomes available
  - No sleeping, just checking over and over, which wastes CPU cycles, but can be fast if the wait time is short.

b. **Hardware Support**

a. **Test and set**

This is an atomic instruction that test the value of a variable and sets it to true (locked)

- If the var was already true (locked), the thread waits, if false, locks and proceeds
- Useful for implementing basic locks, but also uses busy waiting

b. **Compare and Swap (CAS)**

Also atomic, checks whether a value matches an expected value, and only then updates

- This is more flexible and powerful than test and set
- Commonly used in **lock free data structure**, and modern mutex implementations

c. **Disabling Interrupts (Kernel Only)**

Sometimes need to disable interrupts before entering critical section so no other thread can preempt current one.

- This works only in single CPU systems or inside the kernel
- Not recommended in user space or multicore environments

Critical Section

A critical section is any code that accesses shared resources that must not be concurrently modified. They key rule is : only one thread/process should access the critical section at a time.

The problem

If multiple threads/processes access shared resources **w/out coordination** → race conditions

Def - Race Condition - Happens when two or more threads race to read and write the same data at the same time. It can lead to unpredictable results.

Def - Atomic Instruction - A low level CPU operation that runs completely without interruption - meaning no other thread or process can interfere while its happening/

- It is indivisible - either it finishes entirely or it doesn't happen at all
- This is important for synchronization, because it avoids race conditions.

Monitors (High level Abstraction)

Built in synchronization feature provided by some programming languages that combines

- A mutex to control who can enter the critical section
- Condition variables for threads to wait and signal each other inside the monitor

Features

- Monitors abstract away the complexity - you don't need to manually lock/unlock
- They bundle data and synchronization logic together
- Internally, they use mutexes and condition variables

Def - Condition Variable - Allows threads to wait for certain conditions to become true and lets other threads signal when those conditions are met

Def - trylock() - Non blocking version of lock().

- It tries to acquire the lock, if the lock is already held by another thread, it doesn't wait - it just returns an error (or false), if it succeeds, the thread enters the critical section.

What is Concurrency

Concurrency is the ability of a system to allow multiple tasks (processes or threads) to make progress a the *same logical time*. It doesn't require multiple CPUs.

Logical concurrency can be achieved even on a single core processor by interleaving execution.

Def: Interleaving - Alternating the execution of instructions from multiple tasks on a single CPU, so that it appears as if tasks are running at the same time.

- A single core processor can only execute one single instruction at a time, the OS rapidly switches between tasks - this is called **context switching**. Doing this frequently creates the illusion of tasks progressing simultaneously, even though only one is running.

Types of Concurrency

- Logical Concurrency - Tasks *appear* to run simultaneously
- Physical Concurrency - Tasks *actually* run at the same time

Why do we care?

- OS's must handle multiple active entities at once : processes, IO, user interaction, etc
- It enables parallelism, responsiveness, and efficiency.
  - **Def** - Parallelism - When multiple tasks are literally executed at the same time using multiple processing units (multiple cores for ex). Different than **interleaving**, when tasks just appear to run at the same time but actually take turns.

Blocking vs Busy Waiting

Busy Waiting

- The thread spins checking a condition repeatedly
- Wastes CPU cycles - bad for performance

Blocking

- The OS suspends the thread until it can enter the critical section
- More efficient, especially with many threads

Def - Buffer - Region of memory used to temporarily store data while it is being transferred between two place. Such as between a producer and a consumer.

Def: Livelock-2+ threads keep change state in response to each other, but don't make progress

What is Synchronization

Synchronization is the coordination of multiple threads or processes so shared resources are accessed safely & program behaves correctly even when many things are happening at once.

Requirements for a correct solution

Mutual Exclusion - Only one process/thread in the critical section (CS) at a time

Why its important - If two threads modify the same resource at once, you get data corruption or race conditions.

Progress - If no process is in critical section, and some threads want to enter, the system must allow one of them to go in without unnecessary delay

Why its important - We want the system to be fair and efficient. If the critical section is free, we shouldn't block everyone just because the program is indecisive or stuck

Bounded waiting - Every thread should get a fair turn. If a thread wants to enter critical section, there should be a limit on how many times other threads cut in line before it gets a chance

Why its important - Without bounded waiting, threads can be starved, never get their turn