Section 4 : Multi-object Synchronization (Deadlocks)

Indefinite blocking = Starvation

What is a Deadlock?

A deadlock happens when a group of processes (or threads) are all waiting on each other to release resources - but none ever do. So everything just stops

The Four Necessary Conditions for Deadlock

1. **Mutual Exclusion** - At least one resource is held in a non-shareable mode
2. **Hold and Wait** - A process is holding one resource and waiting to acquire others
3. **No Preemption** - Resources can't be forcibly taken away from a process; they must be released voluntarily
4. **Circular Wait** - There's a cycle of processes, each waiting for a resource held by the next.
   - This is usually the one we check and that is most important

Ignore the Problem

Just reboot if something hangs - This is literally what many operating systems used to do for rare cases. It is still common in embedded systems or when deadlocks are statistically rare

Deadlock Prevention

Disallow Hold and Wait

How? - Require that a process must request all of the resources it will ever need at once

Why this helps - Prevents a process from holding one resource while waiting for another

Downside - It can be very inefficient - what happens if the process ends up not needing all those resources? You've blocked them for nothing

Disallow Circular Wait

How? - Enforce a global ordering on all resources. Every process must request resources in increasing order of that ordering

Why this helps - Circular waiting can't happen if everyone always goes in the same direction - theres no loop

Downside - Requires discipline in coding and planning resource requests, and may not be flexible for dynamic need

Disallow No Preemption

How? -If a process holding some resources requests another one and can't get it, then force it to release all its resources

Why this helps - Prevents a process from hogging resources while waiting - it must give them up if it can't proceed

Downside - Hard to implement safely. For example, you can't just yank mem or a printer mid task - it might corrupt the data or cause a crash.

Disallow Mutual Exclusion

How? - Make resources shareable

Why this helps - If resources aren't exclusive, then there's no blocking

Downside - Not always possible. Some resources *must* be used by one process at a time

Pros of Deadlock Prevention

- Guaranteed safety - Deadlocks simply can't occur if you follow the rules

Cons of Deadlock Prevention

- Performance penalties - Forcing all resources at once or adding rigid ordering can make your system slower and more complex
- Not always practical - You can't always preempt or share certain resources

Deadlock Detection and Recovery

Lets deadlocks happen, but detects them and tries to fix the situation afterward.

Why would we use this

- In systems where performance or flexibility is more important than absolute safety, you might tolerate occasional deadlocks and clean them up later
- Good for long running systems (like operating systems or servers), where halting everything to prevent a rare deadlock would hurt performance

1. **Resource Allocation Graph (RAG)**

   What is it : A directed graph that shows processes (P1, P2, etc), resources (R1, R2,, etc)

   Edges

   - From process → resource : process is requesting a resource
   - From resource → process: resource is allocated to that process

2. **Matrix based Detection Algorithms**

   These are like the Bankers Algorithm, without the avoidance part

   - Data Structures - Available - Allocation - Request

   How it works - Simulate processes completing. If none can finish with what's available, and that never changes, you have a deadlock

   Used when - You have multiple resource instances and can't just rely on graph cycles

Recovery Options

1. **Kill Processes one by one** - Terminate one process at a time until the cycle breaks

   Choice of who to kill - Lowest priority, least work done, uses the fewest resources

   Risk - Can cause data loss or inconsistency

2. **Rollback to Checkpoints**

   - Periodically save snapshots (called checkpoints) of process states
   - If a deadlock is detected, roll some processes back to their last safe point, retry

   Risk: May involve heavy overhead, especially with frequent checkpoints

3. **Preempt and Reassign Resources**

   - Forcefully take resources from one process and give them to another
   - Usually requires support from OS (must pause, save state, and manage side effects)

   Risk : Data corruption or system instability if not carefully handled

Pros

- Flexible-doesn't require processes to behave in constrained way(prevention, avoidance)
- Lets you maximize usage and concurrency most of the time

Cons

- Potentially dangerous - Especially when killing or rolling back processes
- Expensive to implement - Detecting deadlocks is computationally heavy
- Can cause delays or inconsistencies if not carefully managed

Deadlock Avoidance

Deadlock avoidance assumes deadlocks could happen but actively analyzes each resource request to avoid entering a dangerous state.

Before granting a resource request, the system checks if doing so will still keep the system in a safe mode. If its safe → grant the resource. If not → deny or delay the request

Safe State

If there exists some sequence of process completions such that every process can finish using the currently available resources plus those held by others in the sequence.

- A safe sate = guaranteed no deadlock
- An unsafe state = deadlock might happen

The Banker's Algorithm

System only grants resources if it's sure it won't run out.

1. Each process tells the system, its maximum resource need ahead of time.
2. When a process makes a request, the system checks :
   - Will granting this leave enough resources for the rest to finish
   - If yes → grant it, if not → deny it for now
3. Simulates future executions using current allocation to determine if safe sequence exists

Def - Maximal Claim - A maximal claim is the maximum number of resources of each type that a process may ever request during its execution

Pros of Deadlock Avoidance

- More flexible than prevention - Doesn't rigidly block things up front
- Can allow higher concurrency - as long as its still safe

Cons of Deadlock Avoidance

- Needs prior knowledge - processes must state max resource needs upfront
- Expensive to check - system must simulate possible outcomes with every resource request
- Scalability issues - checking all sequences gets slow with many processes/resources