UNIT V: SYNCHRONIZATION AND CONCURRENCY CONTROL

CONCURRENCY:

Principle : It refers to the simultaneous execution of multiple process or threads to improve system efficiency, resource utilization and responsiveness. It is essential for multitasking operating system and distributive system.

Issues with Concurrency:

1. Race Condition: Occurs when multiple process/threads access shared resource and the result depend on execution order, potentially leading to inconsistent states.
2. Deadlock : A situation where two or more process are stuck waiting for resources held by each other preventing further progress.
3. Starvation: A process is perpetually denied access to critical resources due to resource allocation policy or priority scheduling.
4. Live lock : (similar to deadlock) The process keeps changing states (eg. retrying operations) without progress.
5. Resource Contention: Multiple process competing for limited resource, leading to delays or bottlenecks.

MUTUAL EXCLUSION:

It ensures that only one process/thread can access a shared resource or critical section at a time. preventing conflicts and maintaining data consistency.

Approach to achieve Mutual Exclusion:

1. HARDWARE APPROACH.
   - Disable Interrupts: Prevents context switching during critical section execution by disabling CPU interrupts. This approach is simple but not suitable for long critical sections or multi core systems.
   - Test and set lock: A hardware instruction that atomically tests and Modifies a memory location to implement locks It prevents race condition but can cause busy waiting
   - Compare and swap: Updates and modifies a memory location only if its current value matches an expected value, enabling atomic operation and avoiding inconsistent states.

SOFTWARE APPROACH

- Peterson's Algorithm : A classic solution for two process mutual exclusion using shared flags and turn variables.
- Dekker's Algorithm : Another early algorithm for achieving mutual exclusion for two processes
- Bakery Algorithm : Designed for multiple processes; it uses ticket number to decide which process enters the critical section next.

SYNCHRONIZATION TOOLS

1. Semaphore : A synchronization primitive used to control access to shared resource.

Types :

- Binary : Can take values 0 or 1 similar to a lock.
- Counting : Maintains a count of available resources.

Operations :

- wait() : Decreases semaphore value; block if zero
- signal() : Increases semaphore value; unblocks waiting.

2. Mutex : A mutual exclusion ~~box~~ lock that ensures only one thread can access the critical section. Unlike semaphore, a mutex is owned by a thread that locks it and must be explicitly released.
   - use : provides exclusive access to resource in multithread application.

3. Monitor : A high level synchronization construct that allows process to safely share resources. Monitors encapsulate shared data, procedures and synchronization mechanism.
   Automatically ensures mutual exclusion by allowing only one process to execute a monitor procedure at a time.

Classical Syncronization/Concurrency Problems

1. Reader Writer problem:

rules:
- Only one writer may write at a time.
- when writer is writing no read operations
- when reader is reading no write operations
- Multiple readers may read at a time.

Challenge: Avoid starvation of readers & writers

Solution: Use semaphore to implement reader or writer priority.

2. Producer Consumer problem:

A producer produces. A consumer consumes

- when a producer produces more than capacity of consumer (buffer overflow)
- when a consumer doesnt have any resource to consum (buffer underflow)

Challenge: prevent buffer overflow/underflow.

Solution: use semaphore and mutex to synchronize producer and consumer.

eg. halt production when overflow
halt consumption when underflow.

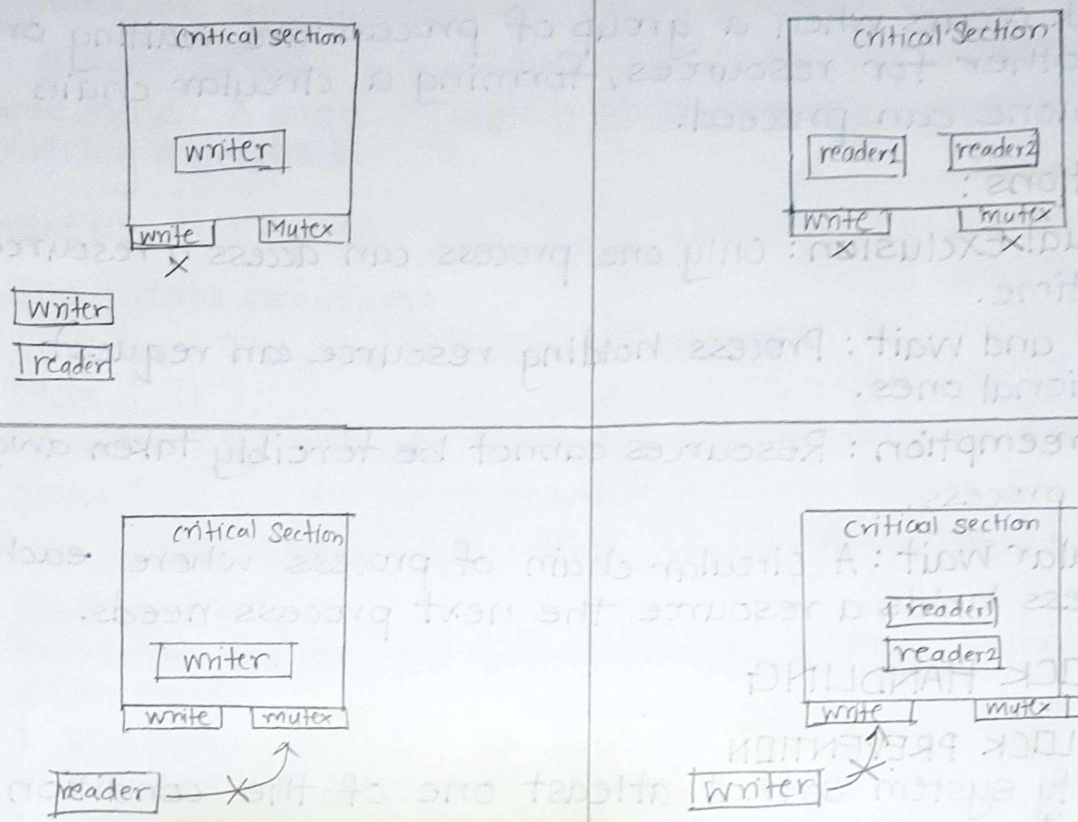
3. Dining Philosopher problem:

Philosophers sitting around a table alternate between thinking and eating requiring shared fork (resources) to eat.

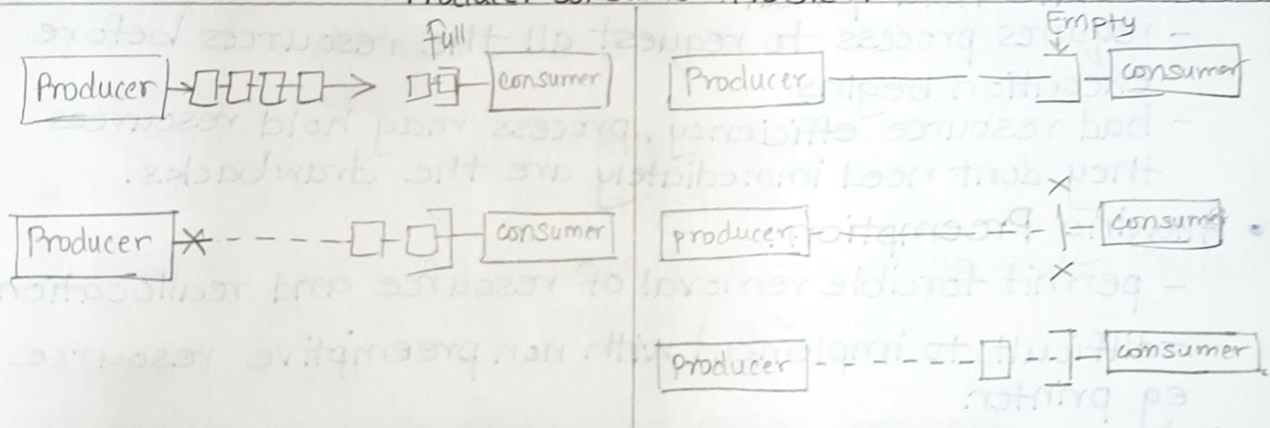
Challenge: prevent deadlock or starvation.

Solution: Implement ~~sta~~ strategies like resource hierarchy, limit the number of philosophers or implement waiter process for resource allocation.

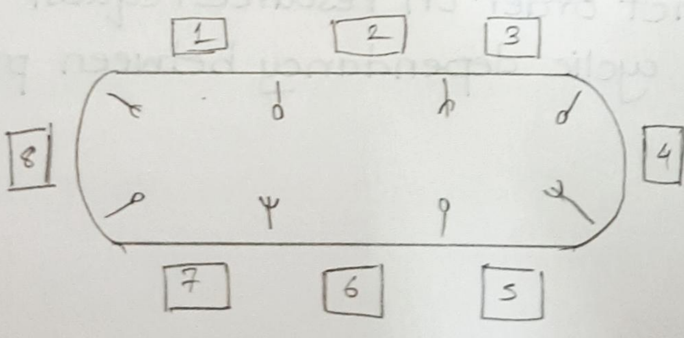
Reader Writer Problem



Producer Consumer Problem



Dining Philosopher Problem.



DEADLOCKS

Deadlock occurs when a group of process are waiting on each other for resources, forming a circular chain and none can proceed.

Conditions:

1. Mutual Exclusion: only one process can access a resource at a time.
2. Hold and wait: Process holding resource can request additional ones.
3. No Preemption: Resources cannot be forcibly taken away from process.
4. Circular wait: A circular chain of process where each process holds a resource the next process needs.

DEADLOCK HANDLING

1. DEADLOCK PREVENTION

Modify system sothat atleast one of the condition cannot occur.

- Eliminate Hold & wait:
  - requires process to request all the resources before execution begins
  - bad resource efficiency, process may hold resources they dont need immediately are the drawbacks.
- Allowing Preemption:
  - permit forcible removal of resource and reallocation
  - difficult to implement with non preemptive resource eg printer.
- Eliminating Circular wait:

Impose strict order on resource request

Ensures no cyclic dependancy between process.

2. DEADLOCK AVOIDANCE

Dynamically allocate resoure while ensuring system is in a safe state

Safe state: A state in which all process are running without entering a deadlock.

BANKERS ALGORITHM:

1. Input data structures:

   Available [] : units of available resource
   Max [][] : Max demand of each process for resource type
   Allocation [][]: Resource currently allocated for each process.
   Need [][] : Remaining resource needs for each process.

   Calculation: Need [i][j] = Max [i][j] − Allocation [i][j]

2. Safety check:
   simulate granting request to ensure system remains in safe state.
   1. Initialize work [] as Available []
   2. Mark all process with finish = false
   3. Find a process Pi such that need[i] <= work & finish[i] = false.
   4. If such process is found then allocate resources to work [] as:
      work[j] = Work[j] + Allocation [i][j]
   5. Mark process Pi as finished = true
   5. Repeat untill all process marked finish = true (safe) or no process can proceed (unsafe)

3. Grant request:
   If system in safe state the grant request else deny request.

3. DEADLOCK DETECTION.

Allow deadlock to occur but detect and resolve when they happen.

- Resource Allocation Graphs (RAG
  - Represents process and resources as nodes.
  - A deadlock exists if graph has a cycle.
- Wait for Graphs (WFG)
  - Simplified version of RAG used for system with multiple resources.
  - A cycle indicates deadlock.
- Periodic deadlock checks:
  - perform deadlock detection at regular intervals or when resource request is denied.

4. DEADLOCK RECOVERY

After detecting a deadlock, recover the system by breaking the cycle and freeing resources.

- Process termination:
  - Abort all deadlocked Process: simplifies recovery but can result in significant data loss or incomplete task.
  - Abort one process at a time: incrementally terminate process until the deadlock is resolved.
  - Selection Criteria for termination:
    - Priority of the process
    - Execution time (shortest to longest)
    - Resources used or required
    - Impact on other processes.
- Resources Preemption.
  - temporarily remove resources from a process and reallocate to other processes to break the cycle.
  - Requires mechanism to roll back the states of preempted processes
- Challenges
  - Resource preemption may not always be feasible
  - Process termination risks losing critical data or functionality.