Concurrency Control & Deadlock

Department of Computer Science & Engineering (Data Science)

R. C. Patel Institute of Technology, Shirpur

(An Autonomous Institute)

Unit-IV 09 Hrs.

- Concurrency control: Concurrency: Principles of Concurrency,
- Mutual Exclusion: S/W approaches, H/W Support,
- Semaphores, Monitors,
- Classical Problems of Synchronization: Readers Writers and Producer Consumer problems and solutions.
- Deadlock: Principles of deadlock, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, Dining Philosopher problem.
- Comparative study of concurrency control in Windows, Linux and An droid OS.

Concurrency

Concurrency refers to the ability of a system to execute multiple operations or tasks simultaneously. This doesn't always mean true parallel execution (which requires multiple processors or cores); it can also mean interleaving execution on a single processor through context switching.

Concurrency improves efficiency, responsiveness and user experience.

Applications:

1. **Web Browsers:** Load multiple tabs at once.
2. **Operating Systems:** Run music, downloads and editing software at the same time.
3. **Databases:** Multiple users accessing records simultaneously.
4. **Gaming:** Handle graphics, sound and user input together.
5. **Airline Reservation Systems:** Thousands of users booking tickets at once.

Principles of Concurrency

The principles of concurrency in operating systems are designed to ensure that multiple processes or threads can execute efficiently and effectively, without interfering with each other or causing deadlock.

- **Mutual exclusion**

Mutual exclusion refers to the principle of ensuring that only one process or thread can access a shared resource at a time. This is typically implemented using locks or semaphores to ensure that multiple processes or threads do not access a shared resource simultaneously.

- **Synchronization**

Synchronization refers to the coordination of multiple processes or threads to ensure that they do not interfere with each other. This is done through the use of synchronization primitives such as locks, semaphores, and monitors. These primitives allow processes or threads to coordinate access to shared resources such as memory and I/O devices.

- **Interleaving**

Interleaving refers to the interleaved execution of multiple processes or threads. The operating system uses a scheduler to determine which process or thread to execute at any given time. Interleaving allows for efficient use of CPU resources and ensures that all processes or threads get a fair share of CPU time.

Principles of Concurrency

- **Deadlock avoidance**

Deadlock is a situation in which two or more processes or threads are waiting for each other to release a resource, resulting in a deadlock. Operating systems use various techniques such as resource allocation graphs and deadlock prevention algorithms to avoid deadlock.

- **Process or thread coordination**

Processes or threads may need to coordinate their activities to achieve a common goal. This is typically achieved using synchronization primitives such as semaphores or message passing mechanisms such as pipes or sockets.

- **Resource allocation**

Operating systems must allocate resources such as memory, CPU time, and I/O devices to multiple processes or threads in a fair and efficient manner. This is typically achieved using scheduling algorithms such as round-robin, priority-based, or real-time scheduling.

Problems in Concurrency

- Race conditions occur when the output of a system depends on the order and timing of the events, which leads to unpredictable behavior. Multiple processes or threads accessing shared resources simultaneously can cause race conditions.
- Deadlocks occur when two or more processes or threads are waiting for each other to release resources, resulting in a circular wait. Deadlocks can occur when multiple processes or threads compete for exclusive access to shared resources.
- Starvation occurs when a process or thread cannot access the resource it needs to complete its task because other processes or threads are hogging the resource. This results in the process or thread being stuck in a loop, unable to make progress.

Problems in Concurrency

- Priority inversion occurs when a low-priority process or thread holds a resource that a high-priority process or thread needs, resulting in the high-priority process or thread being blocked.
- Memory consistency refers to the order in which memory operations are performed by different processes or threads. In a concurrent system, memory consistency can be challenging to ensure, leading to incorrect behavior.
- Deadlock avoidance techniques can prevent deadlocks from occurring, but they can lead to inefficient use of resources or even starvation of certain processes or threads.

Real-World Applications of Concurrency

- **Multithreaded web servers**

Web servers need to handle multiple requests simultaneously from multiple clients. A multithreaded web server uses threads to handle multiple requests concurrently, improving its performance and responsiveness.

- **Concurrent databases**

Databases are critical for many applications, and concurrency is essential to support multiple users accessing the same data simultaneously. Concurrent databases use locking mechanisms and transaction isolation levels to ensure that multiple users can access the database safely and efficiently.

- **Parallel computing**

Parallel computing involves breaking down large problems into smaller tasks that can be executed simultaneously on multiple processors or computers. This approach is used in scientific computing, data analysis, and machine learning, where parallel processing can significantly improve the performance of complex algorithms

Synchronization Mechanism

- **Race Condition :**

A **Race Condition** typically occurs when two or more threads try to read, write and possibly make the decisions based on the memory that they are accessing concurrently.

- **Critical Section :**

The regions of a program that try to access shared resources and may cause race conditions are called critical section. To avoid race condition among the processes, we need to assure that only one process at a time can execute within the critical section.

Critical Section Problem

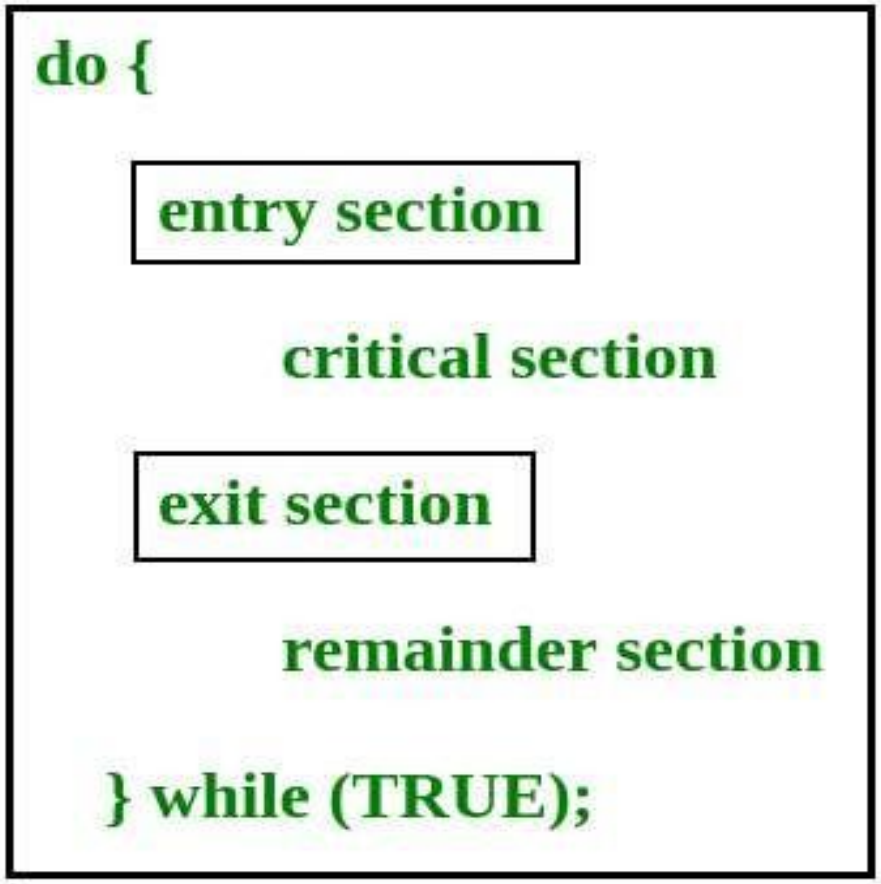
- Critical Section is the part of a program which tries to access shared resources. That resource may be any resource in a computer like a memory location, Data structure, CPU or any IO device.

- The critical section cannot be executed by more than one process at the same time; operating system faces the difficulties in allowing and disallowing the processes from entering the critical section.

- The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise.

Critical Section Problem

- Critical section is a code segment that can be accessed by only one process at a time. Critical section contains shared variables which need to be synchronized to maintain consistency of data variables.



- In the entry section, the process requests for entry in the Critical Section.
- **Any solution to the critical section problem must satisfy three requirements:**

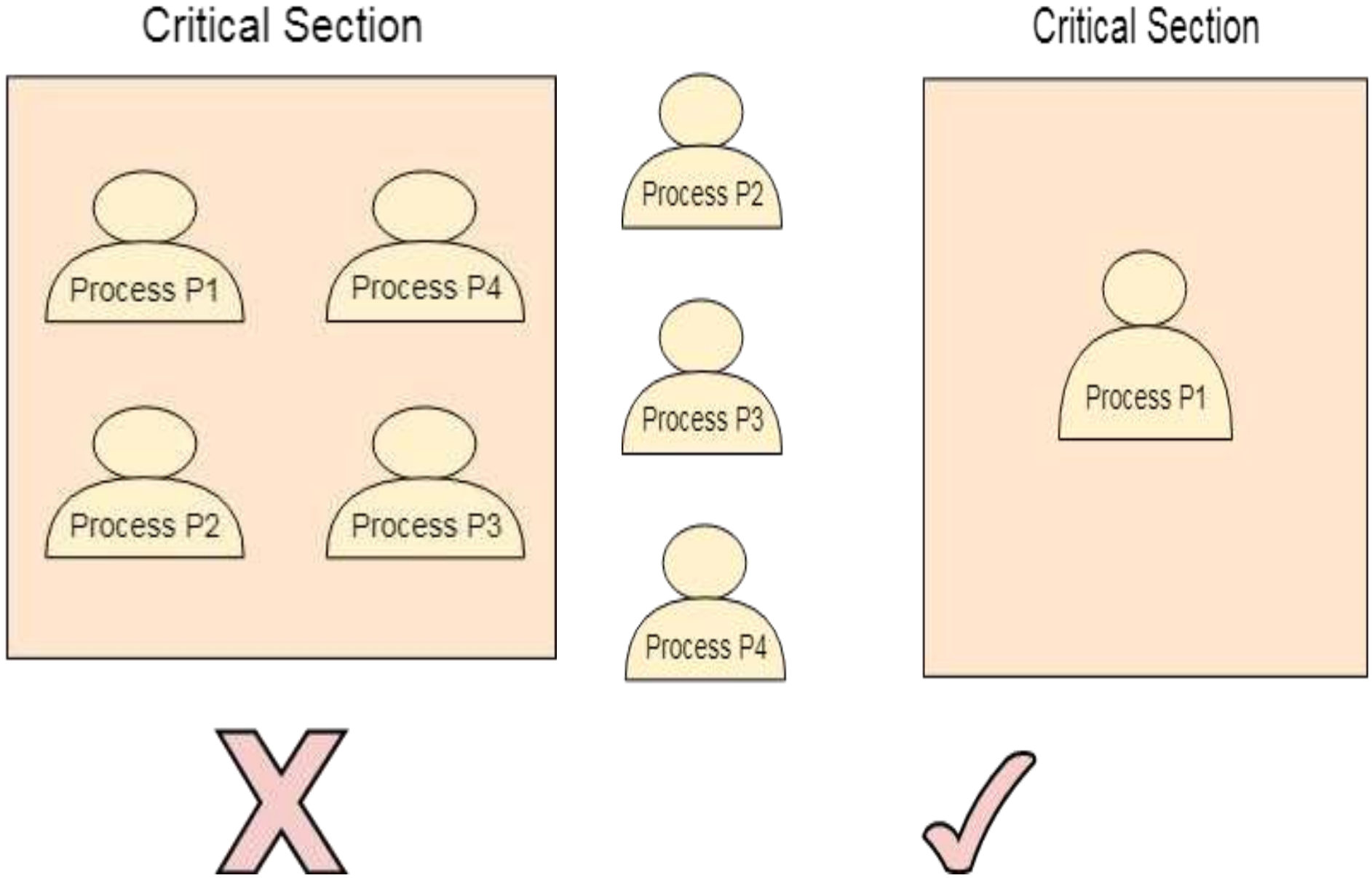
1. Mutual exclusion
2. Progress
3. Bounded waiting

Requirements of Synchronization mechanisms

- **Primary**

- **Mutual Exclusion**

Our solution must provide mutual exclusion. By Mutual Exclusion, we mean that if one process is executing inside critical section then the other process must not enter in the critical section.


Critical Section
Process P1
Process P4
Process P2
Process P3
Process P2
Process P3
Process P4
Critical Section
Process P1

Requirements of Synchronization mechanisms

- **Primary**
- **Progress**

Progress means that if one process doesn't need to execute into critical section then it should not stop other processes to get into the critical section.

Requirements of Synchronization mechanisms

- **Secondary**
- **Bounded Waiting**
- We should be able to predict the waiting time for every process to get into the critical section. The process must not be endlessly waiting for getting into the critical section.

- **Secondary**
- **Architectural Neutrality**

Our mechanism must be architectural natural. It means that if our solution is working fine on one architecture then it should also run on the other ones as well.

Mutual Exclusion: S/W approaches, H/W Support

1. Software Approaches

These are implemented using programming techniques without direct hardware support.

a. Peterson’s Algorithm

•Works for two processes.
•Uses two shared variables to ensure mutual exclusion.
•Provides fairness and avoids deadlock.
•Not practical for modern multiprocessor architectures due to compiler and hardware optimizations.

b. Dekker’s Algorithm

•Early solution to mutual exclusion for two processes.
•Uses flags and a turn variable to control access.
•More complex and less efficient than Peterson’s.

c. Lamport’s Bakery Algorithm

•Generalizes mutual exclusion for **n processes**.
•Each process takes a "number" (like a bakery ticket) before entering the critical section.
•Processes with smaller numbers get access first.
•Ensures fairness and avoids starvation.

Mutual Exclusion: S/W approaches, H/W Support

2. Hardware Approaches

These rely on low-level atomic instructions provided by the CPU.

a. Disabling Interrupts

•Used in uniprocessor systems.
•Disable interrupts when entering critical section to prevent context switch.
•Not suitable for multiprocessor systems and not scalable.

b. Test-and-Set Instruction

•An atomic CPU instruction that tests and sets a lock variable.
•If the lock is free (false), sets it to true and enters critical section.
•Can lead to **busy waiting** (spinlock), which wastes CPU time.

c. Compare-and-Swap (CAS) / Atomic Swap

•Atomically compares a memory location with a value and swaps it if they match.
•Used in many modern architectures.
•Basis for many lock-free and wait-free data structures.

Semaphores

A Semaphore is simply a variable (integer) used to control access to a shared resource by multiple processes in a concurrent system. It ensures that only the allowed number of processes can use a resource at a given time. It was introduced by **Edsger Dijkstra**.

There are two types:

•**Binary Semaphore (Mutex):** Value is either 0 or 1.

•**Counting Semaphore:** Value can be greater than 1, often used to manage access to a limited number of resources.

Operations

- `P()` or `wait()` : Decreases the semaphore. If the value becomes negative, the process is blocked.
- `V()` or `signal()` : Increases the semaphore. If the value is ≤ 0, a blocked process is unblocked.

Semaphores

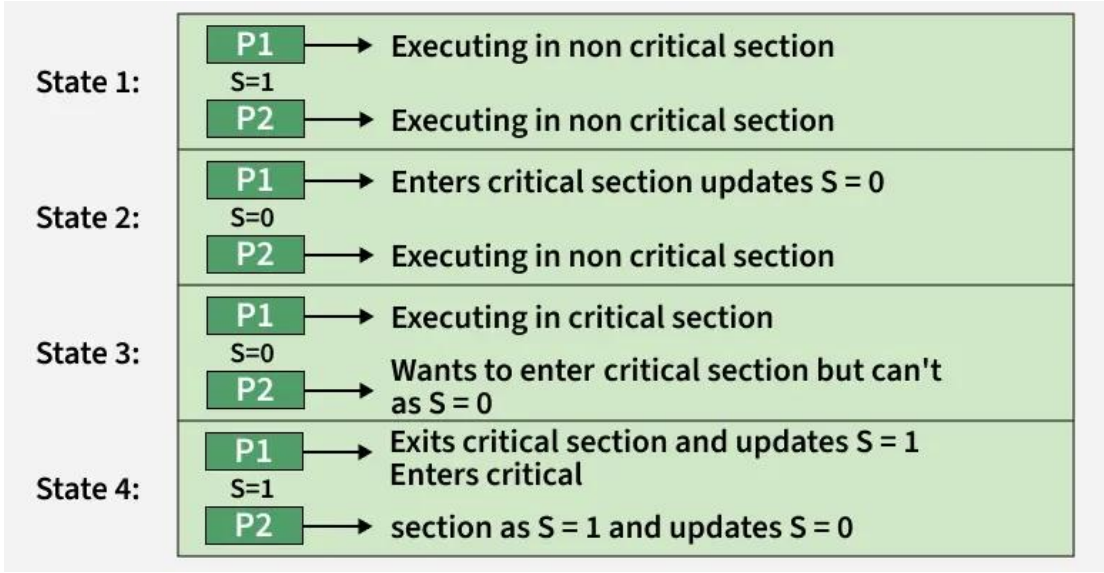
Working of Semaphore

A semaphore works by maintaining a counter that controls access to a specific resource, ensuring that no more than the allowed number of processes access the resource at the same time.

To achieve synchronization, every critical section of code is surrounded by two operations- Wait (P operation) and Signal (V operation) that are discussed above.

Example: Let's consider two processes P1 and P2 sharing a semaphore S, initialized to 1:

- **State 1**: Both processes are in their non-critical sections, and S = 1.
- **State 2**: P1 enters the critical section. It performs wait(S), so S = 0. P2 continues in the non-critical section.
- **State 3**: If P2 now wants to enter, it cannot proceed since S = 0. It must wait until S > 0.
- **State 4**: When P1 finishes, it performs signal(S), making S = 1. Now P2 can enter its critical section and again sets S = 0.



Monitors

In operating systems, a monitor is a **high-level synchronization construct** used to manage concurrent access to shared resources, preventing data corruption and ensuring orderly execution. It works by encapsulating shared data and the procedures that operate on it, allowing only one process or thread to execute within the monitor at a time, which ensures mutual exclusion. This is a more user-friendly alternative to lower-level mechanisms like semaphores.

Key characteristics and components

•**Encapsulation:** A monitor bundles shared data variables and the functions or procedures that operate on them into a single unit.

•**Mutual Exclusion:** A monitor guarantees that only one thread can be active inside its procedures at any given time. Other processes must wait in a queue until the active one exits.

•**Condition Variables:** Monitors use condition variables to allow threads to wait for certain conditions to be met before proceeding. A thread can use wait() to pause its execution and signal() to wake up another waiting thread.

•**Data Protection:** The shared data within a monitor is protected from direct access by external programs, as it can only be manipulated by the monitor's internal procedures.

•**Implementation:** Monitors are typically implemented at the programming language level, not directly by the operating system itself. Languages like Java have built-in support for monitor-like behavior through keywords like synchronized.

•**Alternative to Semaphores:** Monitors provide a higher-level and often safer alternative to semaphores for managing concurrency. They simplify the programming of concurrent systems by handling much of the complexity of synchronization automatically.

Monitors

Key Points:

•A monitor is similar to a **class/module** that groups shared variables and the functions that operate on them.
•Only one thread can execute inside a monitor at a time, ensuring automatic mutual exclusion.
•In Java, monitors are implemented using classes and synchronized methods.
•There is no monitor keyword in Java – the functionality is achieved through synchronized.

How to Implement Monitors

•Monitors are implemented at the programming language level, not directly by the operating system.
•In Java, monitor-like behavior is achieved using the synchronized keyword ensures that only one thread can execute inside the monitor at a time.
•A monitor encapsulates both shared data (critical resource) and the operations that access or modify it.
•Mutual exclusion is enforced automatically, unlike semaphores where programmers must explicitly call wait() and signal().
•Synchronization can be applied through synchronized methods or synchronized blocks.
•Condition variables are used to control thread waiting and signaling.
•The methods wait(), notify(), and notifyAll() provide process coordination.

```
class Monitor{                  //Name of Monitor

  variables;
  condition variables;

  process p1 {....}             //synchronized method
  process p2 {....}             //synchronized method

}
```

Monitors

Condition Variables in Monitors

A condition variable allows threads to wait until a certain condition becomes true. They are always used inside a monitor. There are three main condition variables:

- **wait()**: temporarily releases the monitor lock and puts the thread to sleep until it is signaled.
- **signal()**: wakes up one waiting thread (if any).
- **broadcast()** (in some languages): wakes up all waiting threads.

Limitations of Monitors

- **Language Dependency:** Monitors must be built into the programming language itself; they cannot be added externally like libraries.
- **Compiler Burden:** The compiler has to generate extra code to manage monitor functionality.
- **OS Dependency:** The compiler also needs awareness of operating system facilities that handle critical section access.
- **Limited Language Support:** Only some languages support monitors directly, such as Java, C#, Visual Basic, Ada, and Concurrent Euclid.
- **Reduced Flexibility:** Since monitors are tightly coupled with language and compiler design, portability across different environments is limited.

Classical Problems of Synchronization

Readers Writers Problem

The Readers-Writers Problem is a classic synchronization issue in operating systems. It deals with coordinating access to shared data (e.g., database, file) by multiple processes or threads.

•**Readers**: Multiple readers can read the shared data **simultaneously** without causing inconsistency (since they don't modify data).

•**Writers**: Only **one writer** can access the data at a time, and no readers are allowed while writing (to prevent data corruption).

The challenge is to design a synchronization scheme that ensures:

1.Multiple readers can access data together if no writer is writing.

2.Writers have exclusive access no other reader or writer can enter during writing.

Variants of the Problem

1. Readers Preference

•Readers are given priority.

•No reader waits if the resource is available for reading, even if a writer is waiting.

•Writers may suffer from *starvation*.

2. Writers Preference

•Writers are prioritized over readers.

•Ensures writers won't starve, but readers may wait longer.

Classical Problems of Synchronization

Readers Writers Problem

Solution When Reader Has The Priority Over Writer

Here priority means, no reader should wait if the shared resource is currently open for reading. There are four types of cases that could happen here.

Case	Process 1	Process 2	Allowed/Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Writing	Reading	Not Allowed
Case 3	Reading	Writing	Not Allowed
Case 4	Reading	Reading	Allowed

Synchronization Variables

- **mutex:** Ensures mutual exclusion when updating the reader count (readcnt).
- **wrt:** Controls access to the shared data (used by both readers and writers).
- **readcnt:** Number of readers currently in the critical section.

Semaphore Operations

- **wait():** Decreases semaphore value, blocks if < 0.
- **signal():** Increases semaphore value, wakes waiting process if any.

Classical Problems of Synchronization

Readers Writers Problem

Reader Process(Reader Preference)

1. Reader requests the entry to critical section.
2. If allowed:

•it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.

•It then, signals mutex as any other reader is allowed to enter while others are already reading.

•After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore "wrt" as now, writer can enter the critical section.

3. If not allowed, it keeps on waiting.

```
do {
   wait(mutex);          // Lock before updating readcnt
   readcnt++;
   if (readcnt == 1)
      wait(wrt);         // First reader blocks writers
   signal(mutex);        // Allow other readers in
   // ---- Critical Section (Reading) ----
   wait(mutex);
   readcnt--;
   if (readcnt == 0)
      signal(wrt);       // Last reader allows writers
   signal(mutex);        // Unlock
} while(true);
```

Explanation:

•When a reader enters, it locks mutex to update readcnt.

•If it’s the first reader, it locks wrt so writers are blocked.

•Multiple readers can now read the data simultaneously.

•When a reader exits, it decrements readcnt.

•If it’s the last reader, it unlocks wrt so writers can proceed.

The first reader blocks writers, the last reader allows writers, and all readers in between share the resource. This gives preference to readers, but writers may starve.

Classical Problems of Synchronization

Readers Writers Problem

Writer's Process

```
do {
   wait(wrt);   // Lock resource

   // ---- Critical Section (Writing) ----

   signal(wrt); // Release resource
} while(true);
```

Explanation:

•**wait(wrt):** Writer locks the resource to get exclusive access.

•**Critical Section:** Writer performs writing (no other reader or writer can enter).

•**signal(wrt):** Writer releases the resource after finishing.

Only one writer can write at a time, and no readers are allowed while writing. This ensures data integrity.

The Readers-Writers Problem highlights the need for proper synchronization when multiple processes access shared data.

•Readers Preference solution allows many readers to access simultaneously while blocking writers, improving read efficiency.

•However, writers may starve if readers keep arriving continuously.

•To avoid this, Writers Preference or a Fair solution (using queues or advanced semaphores) can be used to ensure no starvation and balanced access.

Producer Consumer problems

The Producer-Consumer problem is a classic example of a synchronization problem in operating systems. It demonstrates how processes or threads can safely share resources without conflicts. This problem belongs to the process synchronization domain, specifically dealing with coordination between multiple processes sharing a common buffer.

In this problem, we have:

- **Producers**: Generate data items and place them in a shared buffer.
- **Consumers**: Remove and process data items from the buffer.

The main challenge is to ensure:

1. A producer does not add data to a full buffer.
2. A consumer does not remove data from an empty buffer.
3. Multiple producers and consumers do not access the buffer simultaneously, preventing race conditions.

Producer Consumer problems

Semaphore: The Synchronization Tool

A semaphore is an integer-based signaling mechanism used to coordinate access to shared resources. It supports two atomic operations:

•**wait(S)**: Decreases the semaphore value by 1. If the value is ≤0, the process waits.

•**signal(S):** Increases the semaphore value by 1, potentially unblocking waiting processes.

```
wait(S){
    while(S <= 0);  // busy waiting
    S--;
}
signal(S){
    S++;
}
```

Producer Consumer problems

Problem Statement

Consider a fixed-size buffer shared between a producer and a consumer.

•The producer generates an item and places it in the buffer.

•The consumer removes an item from the buffer.

The buffer is the critical section. At any moment:

•A producer cannot place an item if the buffer is full.

•A consumer cannot remove an item if the buffer is empty.

To manage this, we use three semaphores:

•mutex – ensures mutual exclusion when accessing the buffer.

•full – counts the number of filled slots in the buffer.

•empty – counts the number of empty slots in the buffer.

Producer Consumer problems

Semaphore Initialization

```
mutex = 1;    // binary semaphore for mutual exclusion
full = 0;     // initially no filled slots
empty = n;    // buffer size
```

Producer

```
do {
    // Produce an item
    wait(empty);    // Check for empty slot
    wait(mutex);    // Enter critical section

    // Place item in buffer

    signal(mutex);  // Exit critical section
    signal(full);   // Increase number of full slots
} while (true);
```

Consumer

```
do {
    wait(full);     // Check for filled slot
    wait(mutex);    // Enter critical section

    // Remove item from buffer

    signal(mutex);  // Exit critical section
    signal(empty);  // Increase number of empty slots
} while (true);
```

Explanation:

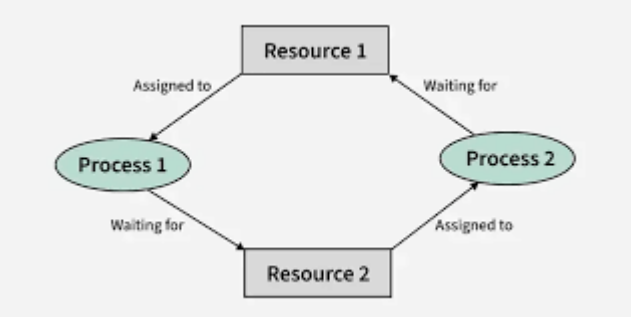
•Empty ensures that producers don't overfill the buffer.

•Full ensures that consumers don't consume from an empty buffer.

•Mutex ensures mutual exclusion, so only one process accesses the buffer at a time.

Deadlock

A deadlock in an operating system is a situation where two or more processes are blocked indefinitely because each is waiting for a resource that is held by another process in the group. This creates a circular dependency, halting the progress of all involved processes until the system intervenes.



How it happens

- **Resource contention:**
- Multiple processes compete for a limited number of system resources, like memory, CPU time, or I/O devices.
- **Circular wait:**
- A cycle is formed where process A needs a resource held by process B, and process B needs a resource held by process A.
- **Example:**
- Imagine two processes, P1 and P2, and two resources, R1 and R2. If P1 holds R1 and requests R2, while P2 holds R2 and requests R1, neither can proceed. P1 is waiting for P2 to release R2, and P2 is waiting for P1 to release R1.

Deadlock

Principles of deadlock

For a deadlock to occur, four conditions must be met simultaneously:

•**Mutual Exclusion:**
At least one resource must be held in a non-sharable mode; only one process can use the resource at a time.

•**Hold and Wait:**
A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

•**No Preemption:**
Resources cannot be forcibly taken from a process holding them; they must be released voluntarily by the process.

•**Circular Wait:**
A set of processes must exist such that each process is waiting for a resource held by the next process in the set, forming a closed chain.

Deadlock Prevention

Deadlock prevention is a strategy used in computer systems to ensure that different processes can run smoothly without getting stuck waiting for each other forever. Think of it like a traffic system where cars (processes) must move through intersections (resources) without getting into a gridlock.

As we have already discussed that deadlock can only happen if all four of the following conditions are met simultaneously:

•Mutual Exclusion

•Hold and Wait

•No Preemption

•Circular Wait

So we can prevent a Deadlock by eliminating any of the above four conditions.

1. Eliminate Mutual Exclusion

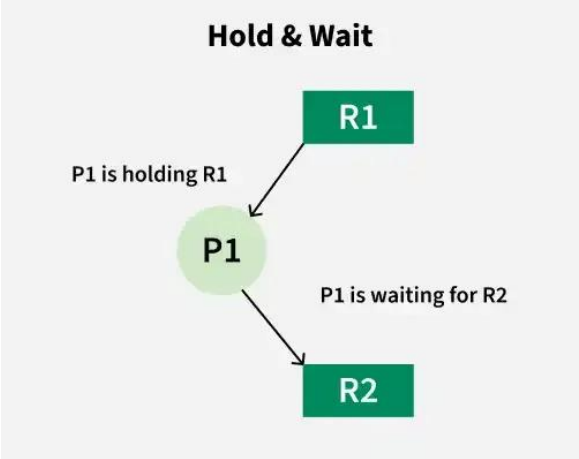
•Some resources, like a printer, are inherently non-sharable, so this condition is difficult to break.

•However, sharable resources like read-only files can be accessed by multiple processes at the same time.

•For non-sharable resources, prevention through this method is not possible.

2. Eliminate Hold and Wait

Hold and wait is a condition in which a process holds one resource while simultaneously waiting for another resource that is being held by a different process. The process cannot continue until it gets all the required resources.



There are two ways to eliminate hold and wait:

•**By eliminating wait**: The process specifies the resources it requires in advance so that it does not have to wait for allocation after execution starts.

For Example, Process1 declares in advance that it requires both Resource1 and Resource2.

•**By eliminating hold**: The process has to release all resources it is currently holding before making a new request.

For Example: Process1 must release Resource2 and Resource3 before requesting Resource1.

3. Eliminate No Preemption

No preemption means resources can’t be taken away once allocated. To prevent this:

•**Processes must release resources voluntarily**: A process gives up resources once it finishes using them.

•**Avoid partial allocation**: If a process requests resources that are unavailable, it must release all currently held resources and wait until all required resources are free.

4. Eliminate Circular Wait

Circular wait happens when processes form a cycle, each waiting for a resource held by the next. To prevent this:

•Impose a strict ordering on resources.

•Assign each resource a unique number.

•Processes can only request resources in increasing order of their numbers.

•This prevents cycles, as no process can go backwards in numbering.

Deadlock Avoidance

Deadlock Avoidance is a process used by the Operating System to avoid Deadlock.
The operating system avoids Deadlock by knowing the maximum resource requirements of the processes initially, and also, the Operating System knows the free resources available at that time. The operating system tries to allocate the resources according to the process requirements and checks if the allocation can lead to a safe state or an unsafe state. If the resource allocation leads to an unsafe state, then the Operating System does not proceed further with the allocation sequence.

Safe State and Unsafe State

A safe state refers to a system state where the allocation of resources to each process ensures the avoidance of deadlock. The successful execution of all processes is achievable, and the likelihood of a deadlock is low. The system attains a safe state when a suitable sequence of resource allocation enables the successful completion of all processes.
Conversely, an unsafe state implies a system state where a deadlock may occur. The successful completion of all processes is not assured, and the risk of deadlock is high. The system is insecure when no sequence of resource allocation ensures the successful execution of all processes.

Deadlock Avoidance Algorithms

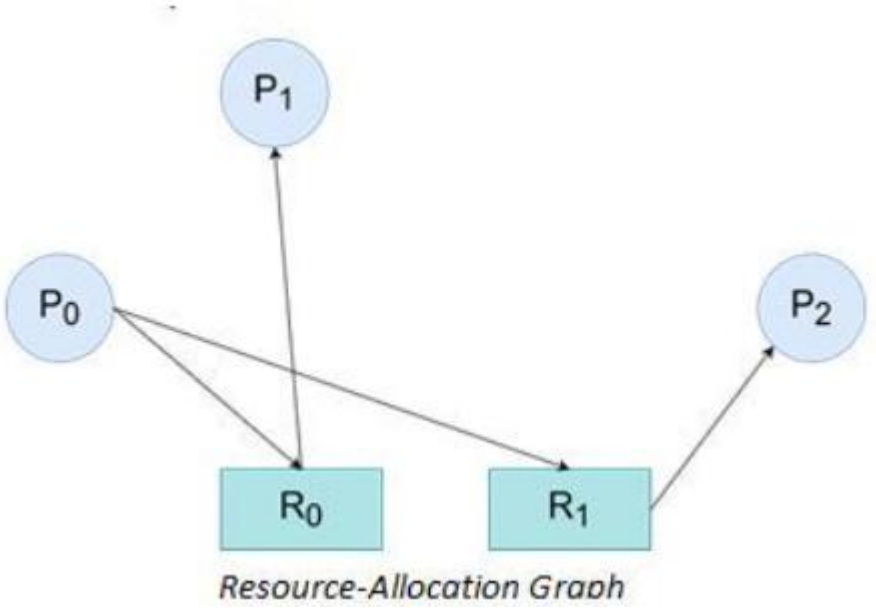
•When resource categories have only single instances of their resources, Resource- Allocation Graph Algorithm is used. In this algorithm, a cycle is a necessary and sufficient condition for deadlock.

•When resource categories have multiple instances of their resources, Banker's Algorithm is used. In this algorithm, a cycle is a necessary but not a sufficient condition for deadlock.

Resource-Allocation Graph Algorithm

Resource Allocation Graph (RAG) is a popular technique used for deadlock avoidance. It is a directed graph that represents the processes in the system, the resources available, and the relationships between them. A process node in the RAG has two types of edges, request edges, and assignment edges. A request edge represents a request by a process for a resource, while an assignment edge represents the assignment of a resource to a process.

To determine whether the system is in a safe state or not, the RAG is analyzed to check for cycles. If there is a cycle in the graph, it means that the system is in an unsafe state, and granting a resource request can lead to a deadlock. In contrast, if there are no cycles in the graph, it means that the system is in a safe state, and resource allocation can proceed without causing a deadlock.



Resource-Allocation Graph

Resource-Allocation Graph Algorithm

The RAG technique is straightforward to implement and provides a clear visual representation of the processes and resources in the system. It is also an effective way to identify the cause of a deadlock if one occurs. However, one of the main limitations of the RAG technique is that it assumes that all resources in the system are allocated at the start of the analysis. This assumption can be unrealistic in practice, where resource allocation can change dynamically during system operation. Therefore, other techniques such as the Banker's Algorithm are used to overcome this limitation.

Banker's Algorithm

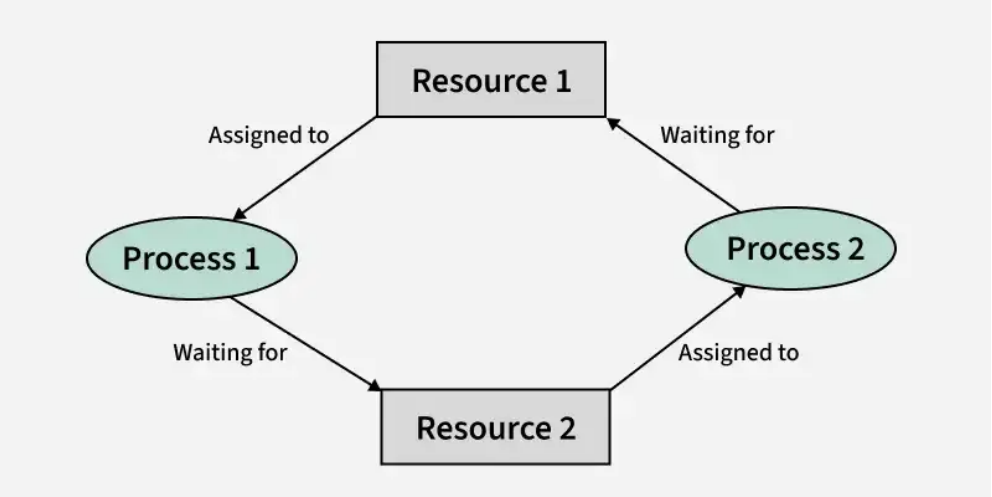
The banker's algorithm is a deadlock avoidance algorithm used in operating systems. It was proposed by Edsger Dijkstra in 1965.

- The Banker's algorithm assumes that each process declares its maximum resource requirements upfront.
- Based on this information, the algorithm allocates resources to each Resource-Allocation Graph process such that the total number of allocated resources never exceeds the total number of available resources.
- The algorithm does not grant access to resources that could potentially lead to a deadlock situation.
- The Banker's algorithm uses a matrix called the "allocation matrix" to keep track of the resources allocated to each process, and a "request matrix" to keep track of the resources requested by each process.
- It also uses a "need matrix" to represent the resources that each process still needs to complete its execution.
- To determine if a request can be granted, the algorithm checks if there is enough available resources to satisfy the request, and then checks if granting the request will still result in a safe state.
- If the request can be granted safely, the algorithm grants the resources and updates the allocation matrix, request matrix, and need matrix accordingly.
- If the request cannot be granted safely, the process must wait until sufficient resources become available.

Deadlock Detection

1. If Resources Have a Single Instance

In this case for Deadlock detection, we can run an algorithm to check for the cycle in the Resource Allocation Graph. The presence of a cycle in the graph is a sufficient condition for deadlock.



In the above diagram, resource 1 and resource 2 have single instances. There is a cycle R1 → P1 → R2 → P2. So, Deadlock is Confirmed.

2. If There are Multiple Instances of Resources

Detection of the cycle is necessary but not a sufficient condition for deadlock detection, in this case, the system may or may not be in deadlock varies according to different situations.

For systems with multiple instances of resources, algorithms like Banker's Algorithm can be adapted to periodically check for deadlocks.

3. Wait-For Graph Algorithm

The Wait-For Graph Algorithm is a deadlock detection algorithm used to detect deadlocks in a system where resources can have multiple instances. The algorithm works by constructing a Wait-For Graph, which is a directed graph that represents the dependencies between processes and resources.

Dining Philosophers Problem

The **Dining Philosopher Problem** is a classic synchronization and concurrency problem that deals with resource sharing, deadlock, and starvation in systems where multiple processes require limited resources. In this article, we will discuss the Dining Philosopher Problem in detail along with proper implementation.

Problem Statement

The **Dining Philosopher Problem** involves 'n' philosophers sitting around a circular table. Each philosopher alternates between two states: **thinking** and **eating**. To eat, a philosopher needs two chopsticks, one on their left and one on their right. However, the number of chopsticks is equal to the number of philosophers, and each chopstick is shared between two neighboring philosophers.

The standard problem considers the value of 'n' as 5 i.e. we deal with 5 Philosophers sitting around a circular table.

Constraints and Conditions for the Problem

•Every Philosopher needs two forks to eat.

•Every Philosopher may pick up the forks on the left or right but only one fork at once.

•Philosophers only eat when they have two forks. We have to design such a protocol i.e. pre and post protocol which ensures that a philosopher only eats if he or she has two forks.

•Each fork is either clean or dirty.

Dining Philosophers Problem

Solution

The correctness properties it needs to satisfy are:

- **Mutual Exclusion Principle:** No two Philosophers can have the two forks simultaneously.
- **Free from Deadlock:** Each philosopher can get the chance to eat in a certain finite time.
- **Free from Starvation:** When few Philosophers are waiting then one gets a chance to eat in a while.
- **No strict Alternation**
- **Proper utilization of time**

Dining Philosophers Problem

Solutions and concepts

•**Semaphores**:

A common solution is to use a semaphore for each chopstick.

- •**Pick up**: A philosopher executes a wait() operation on the semaphore for a chopstick to pick it up.
- •**Put down**: A philosopher executes a signal() operation on the semaphore to release the chopstick.
- •**Deadlock prevention**: To avoid deadlock, a common solution is to have one philosopher (or an even/odd numbered rule) pick up their right chopstick first and then their left, while others pick up their left first, then their right.

•**Monitors**:

Another approach is to use a monitor, which provides a higher-level synchronization mechanism. It can ensure that a philosopher only picks up chopsticks when both are available, preventing deadlock from occurring in the first place.

•**Real-world applications**:

The problem is a model for real-world scenarios like the deadlock in transaction management where multiple transactions need to access shared resources simultaneously, such as bank accounts.

Comparative study of concurrency control in Windows, Linux and Android OS

Feature / Aspect	Windows OS	Linux OS	Android OS
Kernel Type	Hybrid Kernel	Monolithic Kernel (modular)	Based on modified Linux kernel
Concurrency Model	Preemptive multitasking	Preemptive multitasking	Preemptive multitasking
Thread Management	Threads are managed by Windows Kernel (NT Kernel)	Implemented via POSIX threads (pthreads), kernel-level threads	Java threads (mapped to Linux pthreads), managed by Android Runtime (ART)
Synchronization Primitives	- Mutex - Critical Sections - Semaphores - Events - SRW Locks (Slim Reader/Writer)	- Mutex - Spinlocks - Semaphores - Futexes - Read/Write locks	- Java-level: synchronized, Lock - Native: Mutexes, Semaphores, Futexes (Linux)
Interprocess Communication (IPC)	- Named Pipes - Shared Memory - COM (Component Object Model) - Windows Messages	- Pipes - Message Queues - Shared Memory - Signals	- Binder IPC (Android-specific) - Shared Memory - Intents

Comparative study of concurrency control in Windows, Linux and Android OS

Feature / Aspect	Windows OS	Linux OS	Android OS
Scheduler	Round-robin, priority-based, dynamic	Completely Fair Scheduler (CFS)	Inherits Linux CFS, optimized for mobile and low-power
Deadlock Handling	Developers must manually handle deadlocks; tools like WinDbg help analyze them	No automatic deadlock prevention; use of detection tools and coding best practices	Same as Linux; ART helps in analyzing thread behavior
Scalability	Highly scalable on multi-core systems	Scalable, widely used in servers and HPC	Scaled for mobile devices; power-aware thread management
Priority Inversion Handling	Supports priority inheritance via real-time thread priorities	Priority inheritance with real-time mutexes (PREEMPT_RT patches)	Uses real-time features of Linux kernel for critical tasks
Real-time Support	Windows RT and real-time extensions (Windows IoT)	PREEMPT_RT patch makes Linux real-time capable	Not fully real-time, but tuned for responsiveness
Programming APIs	Win32, Windows Threads, .NET Threading	POSIX APIs (pthreads), syscalls	Java Threads, Android APIs, NDK for native threading
Debugging & Tools	WinDbg, Process Explorer, Performance Monitor	GDB, perf, strace, valgrind	Android Studio Profiler, systrace, logcat

THANK YOU