3.1 Concurrency

Issues with Concurrency

Concurrency refers to the simultaneous execution of multiple processes or threads in a system. In operating systems, concurrency allows multiple tasks to run in parallel, improving resource utilization and responsiveness. However, managing concurrent processes or threads introduces several challenges:

1. **Race Conditions**: A race condition occurs when two or more processes or threads attempt to modify shared data simultaneously, and the final outcome depends on the timing of their execution. Race conditions can lead to unpredictable and incorrect results.
2. **Deadlock**: Deadlock arises when two or more processes are blocked because each is waiting for a resource that the other process holds. This leads to a system where no progress can be made.
3. **Starvation**: In some scheduling algorithms, a process may be indefinitely delayed because other processes are continually being given preference. This occurs when higher-priority tasks monopolize the CPU.
4. **Livelock**: Similar to deadlock, in livelock, processes are not blocked but constantly change states without making any progress.
5. **Data Consistency**: When multiple processes access and modify shared data concurrently, ensuring data consistency is a significant challenge. Inconsistent data can lead to system failures, incorrect outputs, or crashes.

Concurrency is difficult to manage due to the unpredictability of process scheduling, varying execution times, and potential conflicts over shared resources. Effective solutions are needed to handle these issues and ensure correct behavior in concurrent systems.

Principles of Concurrency

Concurrency can be achieved through the use of **processes**, **threads**, or **distributed systems**. Several principles must be followed to manage concurrency safely and efficiently:

1. **Synchronization**: Synchronization is the process of coordinating the execution of multiple processes to ensure they do not conflict with one another when accessing shared resources. Synchronization mechanisms such as **locks**, **semaphores**, and **monitors** ensure that critical sections are executed by only one process at a time.
2. **Atomicity**: An operation is atomic if it is indivisible. This means that the operation must be executed as a single, uninterruptible unit. For example, modifying shared data should be atomic to prevent other processes from accessing it midway through the modification.
3. **Isolation**: Isolation ensures that each process or thread appears to run in isolation, even if other processes are executing concurrently. This prevents the execution of one process from affecting the behavior of another.
4. **Order of Execution**: Proper ordering of operations is crucial to ensure that processes execute in a consistent and predictable manner. When two processes depend on each other's outputs, it is important to enforce a specific order of execution to avoid race conditions.

Critical Section and Race Conditions

A **critical section** is a segment of code where a process accesses shared resources or modifies shared data. Since multiple processes or threads may attempt to enter the critical section simultaneously, it is crucial to ensure that only one process is allowed to execute in the critical section at a time. Failure to do so leads to **race conditions**.

A race condition occurs when the outcome of a program depends on the sequence or timing of uncontrollable events, such as the order in which processes access shared data. For example, consider two processes attempting to increment a shared variable at the same time. If both processes read the value of the variable simultaneously and increment it without synchronizing their actions, the final value may be incorrect due to overlapping operations.

To solve this, synchronization mechanisms such as **locks** and **semaphores** are used to ensure **mutual exclusion**, which guarantees that only one process can access the critical section at a time.

Pipes and Types of Pipes

**Pipes** are a form of inter-process communication (IPC) that allow data to be passed between processes. Pipes create a unidirectional communication channel where one process writes data to the pipe, and another process reads data from it.

There are two main types of pipes:

1. **Unnamed Pipes**: Unnamed pipes are typically used for communication between parent and child processes. They exist only while the processes are running and are created using system calls like pipe() in UNIX-based systems. Unnamed pipes are unidirectional, meaning data can flow in only one direction (from the writer to the reader).
2. **Named Pipes (FIFOs)**: Named pipes, also known as FIFOs (First In, First Out), allow communication between unrelated processes. Named pipes persist beyond the lifetime of the processes, and they are identified by a name in the file system. Data written to the pipe by one process can be read by another process, making them useful for inter-process communication across different applications.

3.2 Mutual Exclusion

Hardware and Software Approaches

**Mutual exclusion** ensures that only one process can access shared resources (like critical sections) at a time. Both hardware and software approaches are used to implement mutual exclusion.

Hardware Approaches

1. **Disabling Interrupts**: One hardware-based solution is to disable interrupts during critical sections, ensuring that the CPU does not switch between processes during that time. However, this approach is impractical for long critical sections and is not suitable for multiprocessor systems.
2. **Test-and-Set Lock**: The **test-and-set** instruction is a hardware-based atomic operation that checks and modifies a memory location in a single step. This ensures that no other process can interrupt or alter the memory location while it is being tested and set. This technique is efficient and widely used in mutual exclusion algorithms.
3. **Compare-and-Swap**: Another hardware mechanism is the **compare-and-swap** instruction, which compares the value of a memory location to a given value and swaps it with a new value if they are equal. This operation is atomic and prevents race conditions when multiple processes try to update the same variable.

Software Approaches

1. **Peterson’s Algorithm**: Peterson’s Algorithm is a simple software-based solution for mutual exclusion in a two-process system. It uses two variables, flag and turn, to ensure that only one process enters the critical section at a time. While simple, Peterson's Algorithm is limited to two processes and may not scale well in multiprocessor environments.
2. **Bakery Algorithm**: The Bakery Algorithm extends Peterson’s Algorithm to multiple processes. Each process is assigned a “ticket,” and the process with the smallest ticket gets access to the critical section. This method works in multiprocessor systems, but it can become inefficient when many processes are contending for access.

OS/Programming Language Support: Semaphores, Mutex, and Monitors

Operating systems and programming languages provide built-in synchronization mechanisms like **semaphores**, **mutexes**, and **monitors** to enforce mutual exclusion and synchronization.

1. **Semaphores**: A semaphore is a synchronization primitive that uses an integer variable to control access to shared resources. Semaphores can be either **binary** (0 or 1) or **counting** (any integer value). A semaphore is initialized with a value representing the number of available resources. Processes use the wait() operation to decrease the semaphore and signal() to increase it when they are done using the resource. Semaphores are commonly used to solve problems like deadlocks and resource contention.
2. **Mutex (Mutual Exclusion Lock)**: A **mutex** is a binary synchronization primitive that ensures mutual exclusion by allowing only one process to hold the lock at any time. When a process holds a mutex, other processes are blocked until the mutex is released. Mutexes are lightweight and efficient for protecting critical sections in multithreaded programs.
3. **Monitors**: A **monitor** is a higher-level synchronization construct that combines mutual exclusion and condition synchronization. Monitors provide a locking mechanism, ensuring that only one thread can execute a method in the monitor at a time. Monitors are supported natively in programming languages like **Java** and **C#**, where they are implemented using synchronized blocks or methods.

3.3 Classical Problems of Synchronization

Readers-Writers Problem

The **readers-writers problem** is a classical synchronization problem that occurs when multiple processes (readers and writers) access a shared resource like a database. The challenge is to allow multiple readers to read the resource simultaneously, but only one writer to modify it at a time. There are two variations of the problem:

1. **First Readers-Writers Problem**: Readers get priority over writers. If there are readers in the system, a writer must wait until all readers have finished.
2. **Second Readers-Writers Problem**: Writers get priority over readers. Once a writer is ready to write, no new readers are allowed to start until the writer is finished.

Both versions of the problem can lead to starvation of either readers or writers if not handled properly. Solutions involve using semaphores or monitors to coordinate access and prevent starvation.

Producer-Consumer Problem

The **producer-consumer problem** (also known as the **bounded-buffer problem**) involves a producer process that generates data and a consumer process that uses that data. They share a fixed-size buffer where the producer places items, and the consumer removes items. The challenge is to ensure that the producer does not add items to a full buffer and the consumer does not remove items from an empty buffer.

This problem can be solved using semaphores to synchronize access to the buffer:

* A **mutex** semaphore ensures mutual exclusion when accessing the buffer.
* A **full** semaphore tracks the number of items in the buffer.
* An **empty** semaphore tracks the number of available slots in the buffer.

Dining Philosophers Problem

The **dining philosophers problem** involves five philosophers sitting around a table, each alternating between thinking and eating. There are five forks on the table, and each philosopher needs two forks to eat. The challenge is to design a solution that prevents deadlock and ensures that no philosopher starves.

The dining philosophers problem can be solved using semaphores, where each philosopher picks up the two forks (semaphores) next to them before eating. However, improper handling can lead to deadlock, where all philosophers hold one fork and wait for the second, leading to starvation. Solutions involve techniques like **limiting resource acquisition** or **using wait-and-signal mechanisms** to avoid deadlock.

3.4 Deadlock

Principles of Deadlock

**Deadlock** occurs when two or more processes are blocked because each process is holding a resource and waiting for another resource that another process holds. In this state, none of the processes can make progress. Four conditions must hold simultaneously for a deadlock to occur:

1. **Mutual Exclusion**: At least one resource must be held in a non-sharable mode.
2. **Hold and Wait**: A process holding one resource is waiting for another resource held by another process.
3. **No Preemption**: Resources cannot be forcibly removed from a process holding them.
4. **Circular Wait**: A circular chain of processes exists, where each process holds one resource and waits for another.

Deadlock Prevention

Deadlock prevention techniques focus on breaking one or more of the four conditions necessary for deadlock:

1. **Eliminate Mutual Exclusion**: This is difficult since some resources (like printers) are inherently non-shareable.
2. **Eliminate Hold and Wait**: Processes must request all resources at once or release resources before requesting new ones.
3. **Eliminate No Preemption**: Resources can be forcibly taken from processes if needed.
4. **Eliminate Circular Wait**: Resources can be numbered, and processes are required to request resources in increasing order.

Deadlock Avoidance

Deadlock avoidance techniques ensure that the system never enters a deadlock state. The most common approach is **Banker’s Algorithm**, which requires each process to declare the maximum number of resources it will need. The OS allocates resources only if it determines that the system will remain in a safe state, where there is no risk of deadlock.

Deadlock Detection

In systems where deadlock prevention and avoidance are not feasible, deadlock detection mechanisms are used to periodically check for deadlocks. This involves creating a **wait-for graph** that tracks which processes are waiting for resources held by other processes. If the graph contains a cycle, a deadlock is present.

Deadlock Recovery

Once deadlock is detected, the system must take action to recover from it. Recovery techniques include:

1. **Resource Preemption**: Taking resources away from processes and reassigning them to other processes.
2. **Process Termination**: Terminating one or more processes to break the deadlock cycle.
3. **Rollback**: Rolling back processes to a safe state before the deadlock occurred.