UNIT 3: SYNCHRONIZATION AND MEMORY MANAGEMENT

The Critical-Section Problem

The **Critical-Section Problem** is a fundamental issue in concurrent programming where multiple processes or threads attempt to access shared resources simultaneously. The critical section refers to a section of code where shared resources (such as variables or memory) are accessed, modified, or updated. If two or more threads/processes access this section at the same time, it can lead to inconsistencies or incorrect results.

Requirements for a solution to the Critical-Section Problem:

1. **Mutual Exclusion**: Only one thread/process can be in the critical section at any given time.
2. **Progress**: If no thread/process is in the critical section and multiple threads/processes want to enter it, then one of the threads/processes should be allowed to enter (no starvation).
3. **Bounded Waiting**: There must be a limit on how many times other threads are allowed to enter the critical section before a waiting thread can enter.

Peterson's Solution

Peterson's Solution is a software-based solution to the critical-section problem, specifically for two processes. It works by using two shared variables:

- **flag[]**: An array where each entry indicates whether a process is ready to enter the critical section.
- **turn**: A variable that indicates whose turn it is to enter the critical section.

Key idea:

- Each process sets its flag to `true` to indicate that it wants to enter the critical section.
- Then, the process sets `turn` to the index of the other process, signaling that it's the other process's turn to enter if both processes want to enter the critical section.
- If both processes want to enter, they will continuously check each other's flag and `turn` until one is allowed to enter.

Limitations of Peterson's Solution:

- Peterson's solution is limited to only two processes.
- It relies on busy-waiting, which can waste CPU cycles.
- It assumes a strict memory ordering, which may not be available on all architectures.

Synchronization Hardware

Synchronization hardware refers to special hardware features used to assist in the synchronization of processes/threads. These include atomic operations that ensure mutual exclusion without needing complex software algorithms.

Examples:

1. **Test-and-Set Lock**: An atomic instruction that reads a variable and sets it to a new value in one indivisible operation. Used to implement locks.
2. **Compare-and-Swap (CAS)**: Another atomic operation that compares the value of a memory location with a given value and, if they are equal, replaces it with a new value.
3. **Swap**: An operation that atomically swaps the contents of two memory locations.

These operations are often used to implement low-level synchronization primitives like **mutexes** and **spinlocks**.

Semaphores

A **Semaphore** is a synchronization tool used to manage access to a shared resource by multiple processes or threads. A semaphore is essentially an integer variable that can be manipulated only through two operations: **wait (P)** and **signal (V)**.

Operations:

- **wait(S)**: Decreases the value of semaphore S. If S is less than 0, the process is blocked (i.e., it waits until the semaphore is non-negative).
- **signal(S)**: Increases the value of semaphore S. If there are processes waiting (blocked), one of them is unblocked.

Types of Semaphores:

1. **Binary Semaphore**: Similar to a mutex, it takes values 0 and 1, often used for mutual exclusion.
2. **Counting Semaphore**: Can take any integer value and is used for managing access to a resource pool (e.g., a fixed number of identical resources).

Issues with Semaphores:

- **Deadlock**: If a set of processes wait indefinitely for resources held by each other.
- **Starvation**: A process may be indefinitely delayed from accessing the semaphore.
- **Priority Inversion**: Lower-priority processes may hold a semaphore needed by a higher-priority process.

Classic Problems of Synchronization

There are several well-known synchronization problems that are commonly used to teach synchronization techniques. Here are a few:

1. **The Producer-Consumer Problem**:
    - One process (the producer) generates data and places it in a shared buffer, while another process (the consumer) takes data from the buffer.
    - The buffer has a limited size, so synchronization is required to prevent overflows or underflows.

    Solution: Use a **counting semaphore** to track the number of empty and full slots in the buffer.

2. **The Reader-Writer Problem**:
    - Multiple readers can access the shared data simultaneously, but if a writer is modifying the data, no reader should access it.
    - Writers must have exclusive access to the data.

    Solution: Use a **reader-counting semaphore** for synchronization, ensuring that writers have exclusive access, while allowing concurrent reading.

3. **The Dining Philosophers Problem**:
    - A set of philosophers sit at a table with a fork between each pair. They need both forks to eat, but they must not hold both forks simultaneously to avoid deadlock.
    - Solution involves ensuring that philosophers acquire both forks without causing a deadlock.

Monitors

A **Monitor** is a higher-level abstraction for managing shared resources. A monitor is a synchronization construct that combines **mutexes** and **condition variables** to ensure mutual exclusion. It is a safer, more structured way of handling synchronization.

- **Condition Variables** are used inside monitors to allow threads to wait for certain conditions to be true.
- **Mutexes** are used to ensure mutual exclusion.

A monitor typically has:

- A set of shared variables.
- Procedures that operate on those variables, ensuring mutual exclusion when a procedure is executed.
- A set of condition variables for synchronizing processes inside the monitor.

Advantages:

- Easier to implement compared to semaphores because the monitor itself ensures that only one process can execute within a procedure at a time.
- Conditions can be easily managed via condition variables.

Atomic Transactions

Atomic Transactions are a set of operations that are guaranteed to be performed completely or not at all. This concept is essential in database management systems and systems that require high consistency and reliability.

Properties of Atomic Transactions (ACID properties):

1. **Atomicity**: All operations in a transaction are completed; otherwise, the transaction is aborted.
2. **Consistency**: A transaction brings the system from one consistent state to another.
3. **Isolation**: Transactions are isolated from each other, so the operations of one transaction do not interfere with others.
4. **Durability**: Once a transaction is committed, it cannot be rolled back (even in the case of a system crash).

In programming, atomic operations or transactions are used to avoid conflicts when multiple threads/processes are modifying shared resources.

- **Monitors**: Higher-level synchronization constructs that combine mutexes and condition variables.
- **Atomic Transactions**: Ensure that a series of operations are executed entirely or not at all, maintaining data consistency.

Deadlock

Deadlock is a situation in concurrent computing where a set of processes become blocked because each process is waiting for a resource held by another process in the set, forming a circular chain of dependencies. The processes involved in the deadlock are unable to proceed because they are all waiting on each other.

System Model for Deadlock

In the context of deadlock, the system can be modeled as a **Resource Allocation Graph (RAG)**:

- **Processes**: Represented by nodes in the graph.
- **Resources**: Represented by nodes in the graph.
- **Edges**:
  - **Request Edge**: From process PiP_iPi to resource RjR_jRj, indicating that process PiP_iPi is requesting resource RjR_jRj.
  - **Assignment Edge**: From resource RjR_jRj to process PiP_iPi, indicating that resource RjR_jRj is allocated to process PiP_iPi.

A **circular wait** occurs when there is a cycle in this graph, and this is a typical indicator of a deadlock situation.

Deadlock Characterization

Deadlock has four necessary conditions, all of which must be true for a deadlock to occur:

1. **Mutual Exclusion**: At least one resource must be held in a non-shareable mode (i.e., only one process can use a resource at a time).
2. **Hold and Wait**: A process holding one resource is waiting for additional resources held by other processes.
3. **No Preemption**: Resources cannot be forcibly taken from processes holding them; a process must release the resource voluntarily.
4. **Circular Wait**: A set of processes exists such that each process in the set is waiting for a resource held by the next process in the set, forming a cycle.

If all four conditions are present, a deadlock is possible.

Methods for Handling Deadlocks

There are several approaches to handle deadlocks in a system:

1. Deadlock Prevention

Deadlock prevention ensures that at least one of the four necessary conditions for deadlock does not hold. It can be achieved by:

1. **Eliminating Mutual Exclusion**: Not possible for many resources, as some resources must be non-shareable.
2. **Eliminating Hold and Wait**:
   - Require that a process requesting a resource must hold no other resources.

    - This can lead to resource starvation if a process requests all needed resources at once.
3. **Eliminating No Preemption**:
    - If a process holding some resources is requesting others, all resources held by the process are preempted.
    - Preempting resources can be difficult and might lead to inconsistent states.
4. **Eliminating Circular Wait**:
    - Impose an ordering on resource types (e.g., assigning a number to each resource type, and processes can only request resources in increasing order of their numbers).
    - This effectively eliminates cycles in the wait graph.

2. Deadlock Avoidance

Deadlock avoidance allows the system to dynamically examine resource allocation requests and decide whether to grant or deny them based on future behavior. A well-known technique for deadlock avoidance is the **Banker's Algorithm**.

- The Banker's Algorithm calculates whether allocating a resource will leave the system in a safe state. It checks if, for every possible future request, there exists a sequence of processes that can all finish without entering a deadlock state.
- **Safe State**: A state is safe if there exists at least one sequence of processes that can finish without any process being blocked.
- **Unsafe State**: A state is unsafe if no such sequence exists, meaning deadlock is possible.

3. Deadlock Detection

In systems where deadlock is not prevented or avoided, detection mechanisms are employed to identify when a deadlock occurs. This usually involves:

1. **Resource Allocation Graph (RAG)**: The system checks the graph for cycles, which indicates a deadlock.
2. **Wait-for Graph**: An alternative to RAG, where only processes and resource allocations are represented, and edges represent "waiting" relationships. A cycle in this graph indicates deadlock.

Once a deadlock is detected, the system can take action to recover from the deadlock.

4. Recovery from Deadlock

Once deadlock is detected, there are two primary recovery strategies:

1. **Process Termination**:
    - **Terminate all deadlocked processes**: This ensures no further waiting, but it might be costly.

- **Terminate one process at a time**: This involves identifying the least costly process to terminate first and continues until the deadlock is broken.

2. **Resource Preemption**:
   - **Preempt resources**: If a process is holding resources and is involved in a deadlock, resources can be preempted from it and given to other processes. However, this may require rolling back processes to a safe state.
   - **Rollback**: Roll back one or more processes to a safe state and restart them, ensuring that the deadlock cycle is broken.

Memory Management Strategies

Memory management strategies determine how memory is allocated and managed within a system. Effective memory management ensures that programs run efficiently and that memory is used optimally.

1. Swapping

Swapping is a memory management technique in which processes are moved between the main memory and secondary storage (such as a disk) to free up space for other processes.

- When the system runs out of memory, processes are temporarily swapped out to disk.
- This is useful for systems with limited RAM but can result in **thrashing**, where excessive swapping causes the system to become unresponsive.

2. Contiguous Memory Allocation

In **contiguous memory allocation**, each process is allocated a single contiguous block of memory. This is one of the simplest allocation schemes, but it has its limitations:

- **Fragmentation**: Over time, memory gets fragmented into small blocks that cannot be utilized efficiently. There are two types of fragmentation:
  - **External Fragmentation**: Free memory is split into small pieces, which cannot be used to satisfy larger memory requests.
  - **Internal Fragmentation**: Allocated memory may be larger than needed, causing wasted space inside the block.
- **Limitations**: It is difficult to allocate large blocks of memory as the system becomes more fragmented.

3. Paging

Paging is a memory management scheme that eliminates the problems of contiguous memory allocation by dividing physical memory into fixed-size blocks called **pages** and dividing logical memory into blocks of the same size, called **page frames**. The operating system keeps track of all pages and their locations in physical memory.

- **Page Table**: A table used by the operating system to map virtual memory addresses to physical memory addresses.

Advantages:

- Eliminates fragmentation issues.
- More flexible memory allocation.
- Processes can be allocated non-contiguous blocks of memory.

Disadvantages:

- **Page table overhead**: The operating system must maintain a table to map pages to frames.
- **Page Faults**: If a process accesses a page that is not in memory, a page fault occurs, and the system must load the page from disk.

4. Structure of the Page Table

The **page table** is used to map virtual pages to physical frames in memory. Each entry in the page table contains the address of the frame where the page is stored.

- **Page Table Entries (PTE)** can include:
  - **Frame Number**: The location in physical memory where the page is stored.
  - **Valid Bit**: Indicates whether the page is in memory or on disk.
  - **Access Control Bits**: Used for protection (e.g., read-only, read-write).
  - **Dirty Bit**: Indicates whether the page has been modified.

5. Segmentation

Segmentation is a memory management technique that divides a program’s memory into segments, which are logically related portions of a program, such as:

- **Code Segment**: Contains the program's executable instructions.
- **Data Segment**: Holds global and static variables.
- **Stack Segment**: Used for function calls and local variables.

Unlike paging, segmentation allows for different sizes for each segment, providing a more natural way of dividing a program.

- **Logical View**: Segmentation provides a logical view of memory, which is more aligned with how programmers think about memory.
- **External Fragmentation**: Since segments vary in size, segmentation can suffer from external fragmentation.