Lecture - 4

Process Synchronization: Critical-Section problem - Synchronization Hardware – Semaphores – Classic Problems of Synchronization – Critical Region – Monitors.
Deadlock : Characterization – Methods for handling Deadlocks – Prevention, Avoidance, and Detection of Deadlock - Recovery from deadlock.

Process Synchronization in OS

When two or more process cooperates with each other, their order of execution must be preserved otherwise there can be conflicts in their execution and inappropriate outputs can be produced.

A cooperative process is the one which can affect the execution of other process or can be affected by the execution of other process. Such processes need to be synchronized so that their order of execution can be guaranteed.

The procedure involved in preserving the appropriate order of execution of cooperative processes is known as Process Synchronization. There are various synchronization mechanisms that are used to synchronize the processes.

1. 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.

2. 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 in OS :

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.

In order to synchronize the cooperative processes, our main task is to solve the critical section problem. We need to provide a solution in such a way that the following conditions can be satisfied.

Requirements of Synchronization mechanisms:

Entry in the critical section is handled by the **wait()** function, and the **exit** from the critical section is handled by the **signal()** function.

1. **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.

2. **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.

3. **Bound Waiting**

   There exists a limit on the number of times other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Solutions to Critical Section:

Some methods to solve the problem of Critical Section are as follows:

1. Peterson Solution
2. Semaphore Solution
3. Synchronization Hardware
4. MUTEX

Peterson's Solution:

Peterson's solution is one of the most widely used solutions to the critical section. It is a classical software-based solution.

In this solution, we use two shared variables:

1. int turn – For a process whose turn is to enter the critical section.

2. boolean flag[i] – Value of TRUE indicates that the process wants to enter the critical section. It is initialized to FALSE, indicating no process wants to enter the critical section.

   It allows two or more processes to share a single-use resource without conflict, using only shared memory for communication.

Disadvantages of Peterson's solution:

1. Peterson's solution is limited to two processes.
2. It involves Busy Waiting.

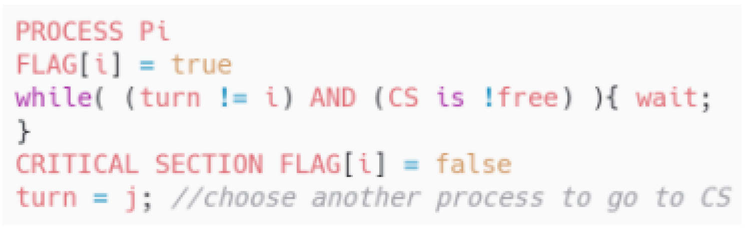
Advantages of Peterson's solution:

It is able to preserve all the three rules for the solution of the critical section:

1. It assures Mutual Exclusion, as only one process can access the critical section at a time.
2. It assures progress, as no process is blocked due to processes that are outside.
3. It assures Bound Waiting as every process gets a chance.

Example of Peterson's solution:

Let's assume there are n number of processes. Each process needs to enter the Critical Section at some point.



```
PROCESS Pi
FLAG[i] = true
while( (turn != i) AND (CS is !free) ){ wait;
}
CRITICAL SECTION FLAG[i] = false
turn = j; //choose another process to go to CS
```

In this case, a FLAG[] array of size n is added, which by default is at FALSE.

Whenever a process needs to enter the critical section, it is set to TRUE.

From the above example: if the process Pi needs to enter, it will set FLAG[i] = TRUE.

There is another variable TURN. Whenever a process is exiting the critical section it will change the TURN to another number from the list of ready processes.

Semaphore:

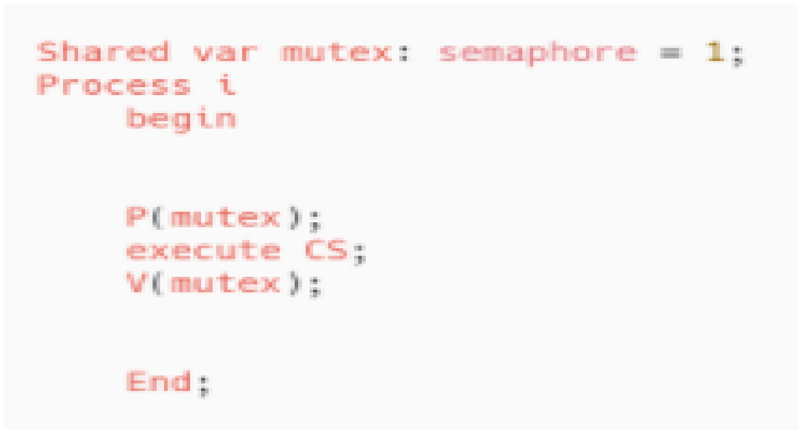
Semaphore is a variable used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment. It is a non-negative variable that is shared between the threads.

There are two types of Semaphores:

- **Binary Semaphore:** It is also known as **Mutex Lock.** It has two values 1 and 0, the value is initialized to 1.
- Counting Semaphore: The value can range over an unrestricted domain. Counting Semaphore is used to control access to a resource that has multiple instances.

Example of a Semaphore:

Let's look at the following example that involves step by step implementation of Semaphore:



```
Shared var mutex: semaphore = 1;
Process i
    begin


    P(mutex);
    execute CS;
    V(mutex);


    End;
```

There are two operations in Semaphores used to implement Process Synchronization:

- **Wait for Operation**
  It helps you control the entry of a task in the critical section.
- **Signal Operation**
  It helps you control the exit of a task from the critical section.

Counting Semaphore vs Binary Semaphore:

Counting Semaphore	**Binary Semaphore**
It provides more than one slot.	It provides one slot.
It provides a set of processes.	It has mutual exclusion mechanism.
Any value.	Only 0 and 1.

Advantages of Semaphore:

The advantages of Semaphore are as follows:

1. It allows flexible management of resources.
2. It is machine-independent.
3. It allows multiple threads to access the critical section.
4. It does not allow multiple processes to access the critical section at the same time.

Disadvantages of Semaphore:

The disadvantages of Semaphore are as follows:

1. Priority inversion.
2. The OS needs to keep track of Wait and Signal operations.
3. This is not a practical method for large-scale use.
4. It may cause deadlock if the Wait and Signal operations are executed in the wrong order.

Synchronization Hardware:

Process Synchronization refers to coordinating the execution of processes so that no two processes can have access to the same shared data and resources. A problem occurs when two processes running simultaneously share the same data or variable.

There are three hardware approaches to solve process synchronization problems:

1. Swap
2. Test() and Set()
3. Unlock and lock

Test and Set

In Test and Set the shared variable is a lock that is initialized to false.

The algorithm returns whatever value is sent to it and sets the lock to true. Mutual exclusion is ensured here, as till the lock is set to true, other processes will not be able to enter and the loop continues. However, after one process is completed any other process can go in as no queue is maintained.

Swap

In this algorithm, instead of directly setting the lock to true, the key is first set to true and then swapped with the lock.

Similar to Test and Set, when there are no processes in the critical section, the lock turns to false and allows other processes to enter. Hence, mutual exclusion and progress are ensured but the bound waiting is not ensured for the very same reason.

Unlock and Lock

In addition to Test and Set, this algorithm uses waiting[i] to check if there are any processes in the wait. The processes are set in the ready queue with respect to the critical section.

Unlike the previous algorithms, it doesn't set the lock to false and checks the ready queue for any waiting processes. If there are no processes waiting in the ready queue, the lock is then set to false and any process can enter.

Classical Problems of Synchronization:

The classical problems of synchronization are as follows:

1. Bound-Buffer problem
2. Sleeping barber problem
3. Dining Philosophers problem
4. Readers and writers problem

Bound-Buffer problem

Also known as the **Producer-Consumer problem**. In this problem, there is a buffer of n slots, and each buffer is capable of storing one unit of data. There are two processes that are operating on the buffer – Producer and Consumer. The producer tries to insert data and the consumer tries to remove data. If the processes are run simultaneously they will not yield the expected output.The solution to this problem is creating two semaphores, one full and the other empty to keep a track of the concurrent processes.

Sleeping Barber Problem

This problem is based on a hypothetical barbershop with one barber.

When there are no customers the barber sleeps in his chair. If any customer enters he will wake up the barber and sit in the customer chair. If there are no chairs empty they wait in the waiting queue.

Dining Philosopher's problem

This problem states that there are K number of philosophers sitting around a circular table with one chopstick placed between each pair of philosophers. The philosopher will be able to eat if he can pick up two chopsticks that are adjacent to the philosopher. This problem deals with the allocation of limited resources.

Readers and Writers Problem

This problem occurs when many threads of execution try to access the same shared resources at a time. Some threads may read, and some may write. In this scenario, we may get faulty outputs.

Deadlock

Deadlock is a situation that occurs in OS when any process enters a waiting state because another waiting process is holding the demanded resource. Deadlock is a common problem in multi-processing where several processes share a specific type of mutually exclusive resource known as a soft lock or software.

What is Circular wait?

One process is waiting for the resource, which is held by the second process, which is also waiting for the resource held by the third process etc. This will continue until the last process is waiting for a resource held by the first process. This creates a circular chain.

For example, Process A is allocated Resource B as it is requesting Resource A. In the same way, Process B is allocated Resource A, and it is requesting Resource B. This creates a circular wait loop.

Deadlock Detection in OS:

A deadlock occurrence can be detected by the resource scheduler. A resource scheduler helps OS to keep track of all the resources which are allocated to different processes. So, when a deadlock is detected, it can be resolved using the below-given methods:

Deadlock Prevention in OS:

It's important to prevent a deadlock before it can occur. The system checks every transaction before it is executed to make sure it doesn't lead the deadlock situations. Such that even a small change to occur dead that an operation which can lead to Deadlock in the future it also never allowed process to execute.

It is a set of methods for ensuring that at least one of the conditions cannot hold.

No preemptive action:

No Preemption – A resource can be released only voluntarily by the process holding it after that process has finished its task

- If a process which is holding some resources request another resource that can't be immediately allocated to it, in that situation, all resources will be released.
- Preempted resources require the list of resources for a process that is waiting.
- The process will be restarted only if it can regain its old resource and a new one that it is requesting.
- If the process is requesting some other resource, when it is available, then it was given to the requesting process.
- If it is held by another process that is waiting for another resource, we release it and give it to the requesting process.

Mutual Exclusion:

Mutual Exclusion is a full form of Mutex. It is a special type of binary semaphore which used for controlling access to the shared resource. It includes a priority inheritance mechanism to avoid extended priority inversion problems. It allows current higher priority tasks to be kept in the blocked state for the shortest time possible.

Resources shared such as read-only files never lead to deadlocks, but resources, like printers and tape drives, needs exclusive access by a single process.

Hold and Wait:

In this condition, processes must be stopped from holding single or multiple resources while simultaneously waiting for one or more others.

Circular Wait:

It imposes a total ordering of all resource types. Circular wait also requires that every process request resources in increasing order of enumeration.

Deadlock Avoidance Algorithms

It is better to avoid a deadlock instead of taking action after the Deadlock has occurred. It needs additional information, like how resources should be used. Deadlock avoidance is the simplest and most useful model that each process declares the maximum number of resources of each type that it may need.

Avoidance Algorithms

The deadlock-avoidance algorithm helps you to dynamically assess the resource-allocation state so that there can never be a circular-wait situation.

A single instance of a resource type.

- Use a resource-allocation graph
- Cycles are necessary which are sufficient for Deadlock

Multiples instances of a resource type.

- Cycles are necessary but never sufficient for Deadlock.
- Uses the banker's algorithm

Difference Between Starvation and Deadlock

Here, are some important differences between Deadlock and starvation:

Advantages of Deadlock

Here, are pros/benefits of using Deadlock method

- This situation works well for processes which perform a single burst of activity
- No preemption needed for Deadlock.
- Convenient method when applied to resources whose state can be saved and restored easily
- Feasible to enforce via compile-time checks
- Needs no run-time computation since the problem is solved in system design

Disadvantages of Deadlock

Here, are cons/ drawback of using deadlock method

- Delays process initiation
- Processes must know future resource need

Deadlock	Starvation
The deadlock situation occurs when one of the processes got blocked.	Starvation is a situation where all the low priority processes got blocked, and the high priority processes execute.
Deadlock is an infinite process.	Starvation is a long waiting but not an infinite process.
Every Deadlock always has starvation.	Every starvation does n't necessarily have a deadlock.
Deadlock happens then Mutual exclusion, hold and wait. Here, preemption and circular wait do not occur simultaneously.	It happens due to uncontrolled priority and resource management.

- Pre-empts more often than necessary
- Dis-allows incremental resource requests
- Inherent preemption losses.

Banker's Algorithm is used majorly in the banking system to avoid deadlock. It helps you to identify whether a loan will be given or not.

- This algorithm is used to test for safely simulating the allocation for determining the maximum amount available for all resources. It also checks for all the possible activities before determining whether allocation should be continued or not.
- For example, there are X number of account holders of a specific bank, and the total amount of money of their accounts is G.
- When the bank processes a car loan, the software system subtracts the amount of loan granted for purchasing a car from the total money (G+ Fixed deposit + Monthly Income Scheme + Gold, etc.) that the bank has.
- It also checks that the difference is more than or not G. It only processes the car loan when the bank has sufficient money even if all account holders withdraw the money G simultaneously.