DEPARTMENT OF AIML

23CST202- OPERATING SYSTEMS

II YEAR IV SEM AIML-B

UNIT 2-PROCESS SCHEDULING AND SYNCHRONIZATION

TOPIC –SEMAPHORES AND CLASSICAL PROBLEMS OF SYNCHRONIZATION

SEMAPHORES

Semaphores are a tool used in operating systems to help manage how different processes (or programs) share resources, like memory or data, without causing conflicts. A semaphore is a special kind of synchronization data that can be used only through specific synchronization primitives. Semaphores are used to implement critical sections, which are regions of code that must be executed by only one process at a time. By using semaphores, processes can coordinate access to shared resources, such as shared memory or I/O devices.

What is Semaphores?

A semaphore is a synchronization tool used in concurrent programming to manage access to shared resources. It is a lock-based mechanism designed to achieve process synchronization, built on top of basic locking techniques.

Semaphores use a counter to control access, allowing synchronization for multiple instances of a resource. Processes can attempt to access one instance, and if it is not available, they can try other instances. Unlike basic locks, which allow only one process to access one instance of a resource. Semaphores can handle more complex synchronization scenarios, involving multiple processes or threads. It help prevent problems like race conditions by controlling when and how processes access shared data.

The process of using Semaphores provides two operations:

- **wait (P):** The wait operation decrements the value of the semaphore
- **signal (V):** The signal operation increments the value of the semaphore.

When the value of the semaphore is zero, any process that performs a wait operation will be blocked until another process performs a signal operation.

When a process performs a wait operation on a semaphore, the operation checks whether the value of the semaphore is >0. If so, it decrements the value of the semaphore and lets the process continue its execution; otherwise, it blocks the process on the semaphore. A signal operation on a semaphore activates a process blocked on the semaphore if any, or increments the value of the semaphore by 1. Due to these semantics, semaphores are also called counting semaphores. The initial value of a semaphore determines how many processes can get past the wait operation.

Semaphores are required for process synchronization to make sure that multiple processes can safely share resources without interfering with each other. They help control when a process can access a shared resource, preventing issues like **race** conditions.

Types of Semaphores

Semaphores are of two Types:

- **Binary Semaphore:** This is also known as a mutex lock, as they are locks that provide mutual exclusion. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes and a single resource.

Counting Semaphore: Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Its value can range over an unrestricted domain.

Working of Semaphore

A semaphore is a simple yet powerful synchronization tool used to manage access to shared resources in a system with multiple processes. It 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.

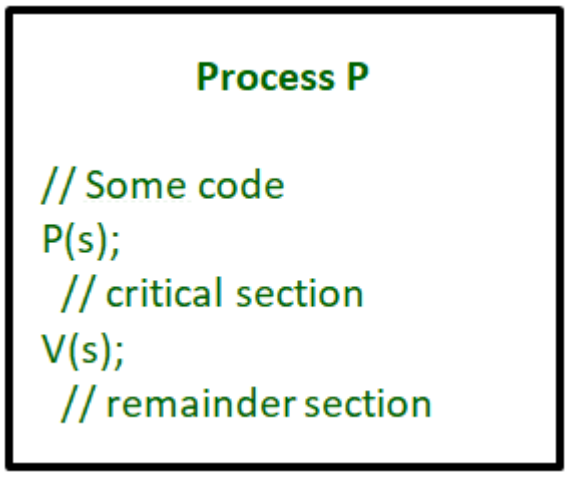
There are two primary operations that a semaphore can perform:

1. **Wait (P operation)**: This operation checks the semaphore's value. If the value is greater than 0, the process is allowed to continue, and the semaphore's value is decremented by 1. If the value is 0, the process is blocked (waits) until the semaphore value becomes greater than 0.
2. **Signal (V operation)**: After a process is done using the shared resource, it performs the signal operation. This increments the semaphore's value by 1, potentially unblocking other waiting processes and allowing them to access the resource.

Now let us see how it does so. First, look at two operations that can be used to access and change the value of the semaphore variable.

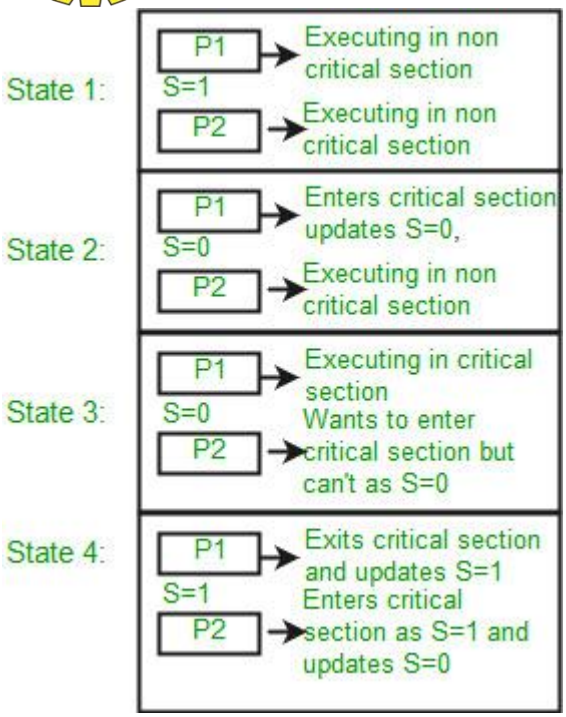
The picture can't be displayed.

A critical section is surrounded by both operations to implement process synchronization. The below image demonstrates the basic mechanism of how semaphores are used to control access to a critical section in a multi-process environment, ensuring that only one process can access the shared resource at a time



Now, let us see how it implements mutual exclusion. Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore s.

This way mutual exclusion is achieved. Look at the below image for details which is a Binary semaphore.



Limitations

- One of the biggest limitations of semaphore is priority inversions.
- Deadlock, suppose a process is trying to wake up another process that is not in a sleep state. Therefore, a deadlock may be blocked indefinitely.
- The operating system has to keep track of all calls to wait and signal the semaphore.

The main problem with semaphores is that they require busy waiting, If a process is in the critical section, then other processes trying to enter the critical section will be waiting until the critical section is not occupied by any process. Whenever any process waits then it continuously checks for semaphore value (look at this line while (s==0); in P operation) and wastes CPU cycle.

There is also a chance of "spinlock" as the processes keep on spinning while waiting for the lock.

Uses of Semaphores

- **Mutual Exclusion** : Semaphore ensures that only one process accesses a shared resource at a time.
- **Process Synchronization** : Semaphore coordinates the execution order of multiple processes.
- **Resource Management** : Limits access to a finite set of resources, like printers, devices, etc.
- **Reader-Writer Problem** : Allows multiple readers but restricts the writers until no reader is present.
- **Avoiding Deadlocks** : Prevents deadlocks by controlling the order of allocation of resources.

Advantages of Semaphores

- Semaphore is a simple and effective mechanism for process synchronization
- Supports coordination between multiple processes. By controlling the access to critical sections, semaphores help in managing multiple processes without them interfering with each other.
- When used correctly, semaphores can help avoid deadlocks by managing access to resources efficiently and ensuring that no process is indefinitely blocked from accessing necessary resources.
- Semaphores help prevent race conditions by ensuring that only one process can access a shared resource at a time.
- Provides a flexible and robust way to manage shared resources.

Disadvantages of Semaphores

- It Can lead to performance degradation due to overhead associated with wait and signal operations.
- If semaphores are not managed carefully, they can lead to deadlock. This often occurs when semaphores are not released properly or when processes acquire semaphores in an inconsistent order.
- It can cause performance issues in a program if not used properly.
- It can be difficult to debug and maintain. Debugging systems that rely heavily on semaphores can be challenging, as it is hard to track the state of each semaphore and ensure that all processes are correctly synchronized
- It can be prone to race conditions and other synchronization problems if not used correctly.
- It can be vulnerable to certain types of attacks, such as denial of service attacks.

Classical Synchronization Problems

These are the classical synchronization problem using the concept of semaphores.

1. Producer-Consumer Problem

The producer-consumer problem involves two types of processes producers that generate data and consumers that process data. The Producer-Consumer Problem is like a restaurant where the chef (producer) makes food and the customer (consumer) eats it. The counter (buffer: A fixed-size queue where producers place items and consumers remove items.) holds food temporarily. A special lock (semaphore) ensures the chef doesn't overflow the counter and the customer doesn't take food when it's not available. In This way, everything runs smoothly and efficiently and gives faster results.

Semaphores Used

- empty: Counts the number of empty slots in the buffer
- full: Counts the number of full slots in the buffer
- mutex: Locks access to the buffer (mutual exclusion)

2. Traffic Light Control

Description: Traffic lights at an intersection can be managed using semaphores to control the flow of traffic and ensure safe crossing.

Example:

Traffic Lights: Represented by semaphores that control the green, yellow, and red lights for different directions.

Semaphores Used: Each light direction is controlled by a semaphore that manages the timing and transitions between light states.

Implementation

Light Controller: Uses semaphores to cycle through green, yellow, and red lights. The controller ensures that only one direction has the green light at a time and manages transitions to avoid conflicts.

3. Bank Transaction Processing

- **Multiple Transactions:** Processes that need to access shared resources (accounts)
- **Account Balance:** Shared resource represented by a semaphore
- **Semaphore value:** 1 (only one transaction can access and modify an account at a time is crucial for maintaining data integrity).

Process

- **Acquire Semaphore:** A transaction must need to acquire the semaphore first before modifying an account balance.
- **The account** is **:** Then operations like debiting or crediting the account are performed by the transaction.
- **Release Semaphore:** After Completing the transactions the semaphore is released to allow other transactions to proceed.

4. Print Queue Management

- **Multiple Print Jobs:** Processes that need to access shared resources (printers)
- **Printer:** Shared resource represented by a semaphore
- **Semaphore Value:** 1 (only one print job can access the printer at a time)

Process

- **Acquire Semaphore:** First, acquire a semaphore for the printer to begin the print job.
- **Print Job Execution:** The printer processes the print job.
- **Release Semaphore:** Now the semaphore is released after the job is done.

5. Railway Track Management

- **Multiple Trains:** Processes that need to access shared resources (tracks)
- **Track:** Shared resource represented by a semaphore
- **Semaphore Value:** 1 (only one train can access the track at a time)

Process

- **Acquire Semaphore:** A train acquires the semaphore before entering a track.
- **Travel Across Track:** train Travels across the track.
- **Release Semaphore:** The semaphores are released after the train passes the track.

6. Dining Philosopher's Problem

- **Multiple Philosophers:** Process that needs to access shared resources(forks).
- **Forks:** Shared resources represented by semaphores, that need to eat. Each philosopher needs two forks to eat.
- **Semaphore Value:** 1 (Only one philosopher can access a fork at a time preventing deadlock and starvation).

Process

- **Acquire Forks:** Philosophers acquire the semaphore for both the left and right forks before eating.
- **Eat** eat **:** When both the forks are acquired then the philosopher eats.
- **Release Forks:** After eating, both the forks are released for others to use them.

Solution – Dining Philosopher Problem Using Semaphores

7. Reader-Writer Problem

In Reader-Writer problem Synchronizing access to shared data where multiple readers can read the data simultaneously, but writers need exclusive access to modify it. In simple terms imagine a library where multiple readers and writers come and all readers want to read a book, and some people(writers) want to update or edit the book. That's why we need a system to ensure that these actions are done smoothly without errors or conflicts.

- **Multiple Readers and Writers:** Processes that need to access a shared resource(data).
- **Data:** Shared resources represented by a Semaphore.
- **Semaphore Value for Reader:** Process that reads shared data, Can access simultaneously(value>1, more than one reader can access at the same time)
- **Semaphore Value for Writers:** Processes that modify shared data need exclusive access(value =1, one at a time)

Solution –

- Readers Preference Solution
- Writers Preference Solution

CLASSICAL PROBLEMS OF SYNCHRONIZATION

With the evolution of multi-processors, tasks suffer due to overheads and traffic. To manage the resources efficiently, synchronization is needed. Synchronization ensures perfect execution of processes and controlled access to shared resources. Synchronization in operating systems is often explained with the help of real-life examples.

In this article, we will see a number of classical problems of synchronization as examples of a large class of concurrency-control problems. In our solutions to the problems, we use semaphores for synchronization, since that is the traditional way to present such solutions. However, actual implementations of these solutions could use mutex locks instead of binary semaphores.

Synchronization Problems with Semaphore Solution
These problems are used for testing nearly every newly proposed synchronization scheme. The following problems of synchronization are considered as classical problems:

1. Bounded-buffer (or Producer-Consumer) Problem,
2. Dining-Philosophers Problem,
3. Readers and Writers Problem,
4. Sleeping Barber Problem

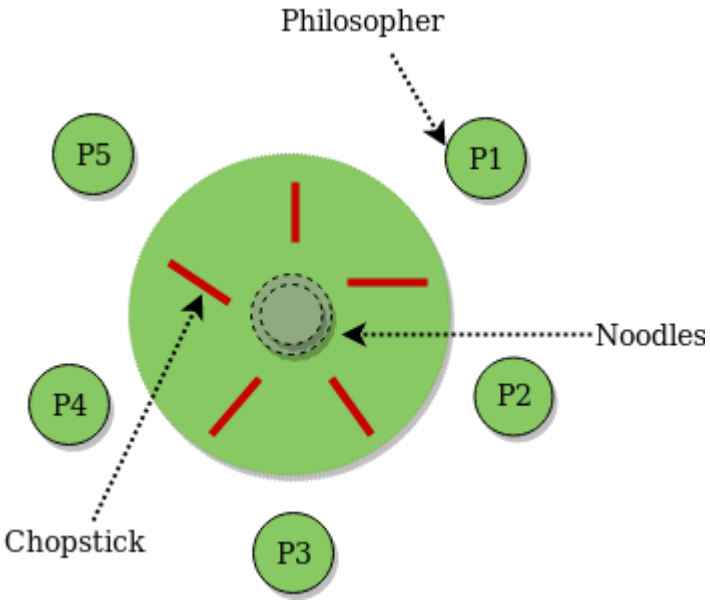
These are summarized, for detailed explanation, you can view the linked articles for each.

Bounded-Buffer (or Producer-Consumer) Problem

The Bounded Buffer problem is also called the producer-consumer problem. This problem is generalized in terms of the Producer-Consumer problem. The solution to this problem is, to create two counting semaphores “full” and “empty” to keep track of the current number of full and empty buffers respectively. Producers produce a product and consumers consume the product, but both use of one of the containers each time.

Dining-Philosophers Problem

The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both. This problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.



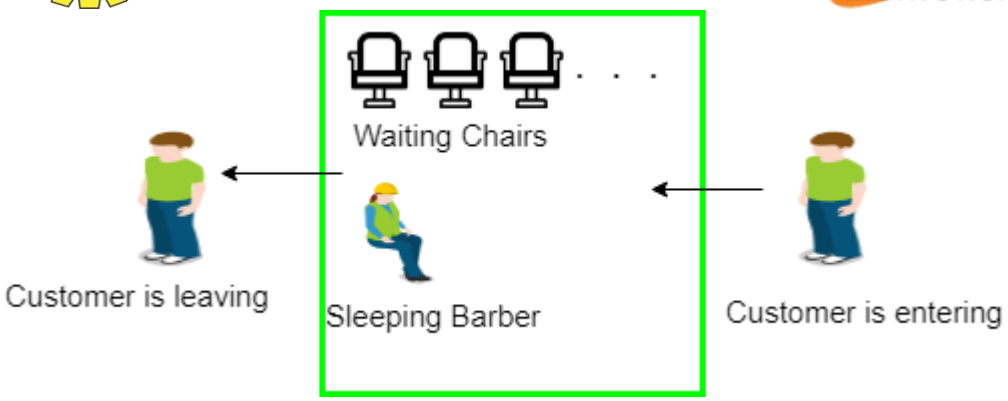
Readers-Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Precisely in OS we call this situation as the readers-writers problem. Problem parameters:

- One set of data is shared among a number of processes.
- Once a writer is ready, it performs its write. Only one writer may write at a time.
- If a process is writing, no other process can read it.
- If at least one reader is reading, no other process can write.
- Readers may not write and only read.

Sleeping Barber Problem

- Barber shop with one barber, one barber chair and N chairs to wait in. When no customers the barber goes to sleep in barber chair and must be woken when a customer comes in. When barber is cutting hair new customers take empty seats to wait, or leave if no vacancy. This is basically the Sleeping Barber Problem.


Waiting Chairs
Customer is leaving
Sleeping Barber
Customer is entering