Semaphores in Process Synchronization

Last Updated : 07 Jan, 2025

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.

To learn more refer: Types of Semaphores

Working of Semaphore

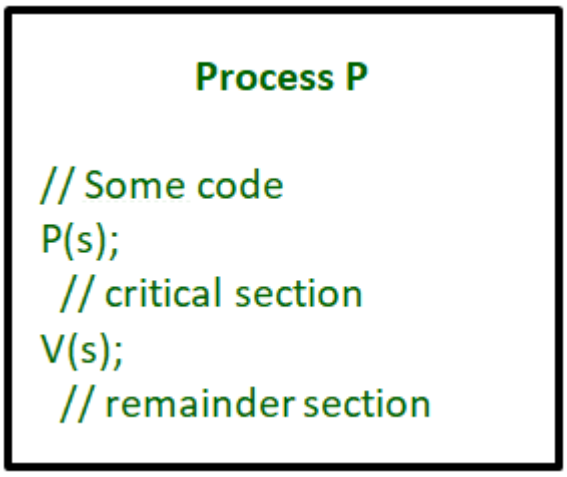
A semaphore is a simple yet powerful synchronization tool used to manage access to shared resources in a system with multiple processes. It works by maintaining a counter that controls access to a specific resource, ensuring that no more than the allowed number of processes access the resource at the 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.

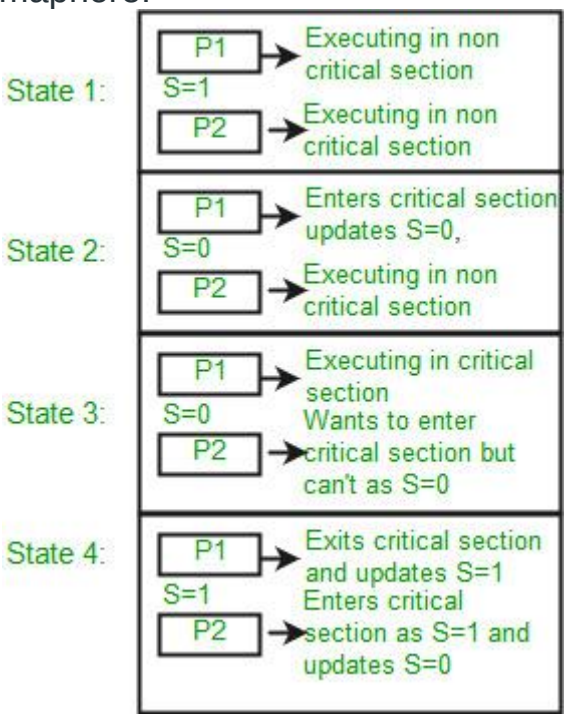
Now let us see how it does so. First, look at two operations that can be used to access and change the value of the semaphore variable.

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.



Implementation - Binary Semaphores

```
struct semaphore {

    enum value(0, 1);

    // q contains all Process Control Blocks (PCBs)
    // corresponding to processes got blocked
    // while performing down operation.
    Queue<process> q;

};
P(semaphore s)
{
    if (s.value == 1) {
        s.value = 0;
    }
    else {
        // add the process to the waiting queue
        q.push(P) sleep();
    }
}
V(semaphore s)
{
    if (s.q is empty) {
        s.value = 1;
```

```
    }
    else {

        // select a process from waiting queue
        Process p = q.front();
        // remove the process from waiting as it has been
        // sent for CS
        q.pop();
        wakeup(p);
    }
}
```

The description above is for binary semaphore which can take only two values 0 and 1 and ensure mutual exclusion. There is one other type of semaphore called counting semaphore which can take values greater than one.

Now suppose there is a resource whose number of instances is 4. Now we initialize S = 4 and the rest is the same as for binary semaphore. Whenever the process wants that resource it calls P or waits for function and when it is done it calls V or signal function. If the value of S becomes zero then a process has to wait until S becomes positive. For example, Suppose there are 4 processes P1, P2, P3, and P4, and they all call wait operation on S(initialized with 4). If another process P5 wants the resource then it should wait until one of the four processes calls the signal function and the value of semaphore becomes positive.