Introduction to Semaphores

In 1965, Dijkstra proposed a new and very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes. This integer variable is called a **semaphore**. So it is basically a synchronizing tool and is accessed only through two low standard atomic operations, **wait** and **signal** designated by P(S) and V(S) respectively.

The classical definitions of **wait** and **signal** are:

- **Wait**: This operation decrements the value of its argument S, as soon as it would become non-negative(greater than or equal to 1). This Operation mainly helps you to control the entry of a task into the critical section. In the case of the negative or zero value, no operation is executed. wait() operation was originally termed as P; so it is also known as **P(S) operation**. The definition of wait operation is as follows:

```
wait(S)

{

   while (S<=0);//no operation

   S--;

}
```

When one process modifies the value of a semaphore then, no other process can simultaneously modify that same semaphore's value. In the above case the integer value of S(S<=0) as well as the possible modification that is S-- must be executed without any interruption.

- **Signal**: Increments the value of its argument S, as there is no more process blocked on the queue. This Operation is mainly used to control the exit of a task from the critical section.signal() operation was originally termed as V; so it is also known as **V(S) operation**. The definition of signal operation is as follows:

```
signal(S)

{

S++;

}
```

Properties of Semaphores

1. It's simple and always have a non-negative integer value.
2. Works with many processes.
3. Can have many different critical sections with different semaphores.
4. Each critical section has unique access semaphores.
5. Can permit multiple processes into the critical section at once, if desirable.

Types of Semaphores

Semaphores are mainly of two types in Operating system:

1. **Binary Semaphore:**

   It is a special form of semaphore used for implementing mutual exclusion, hence it is often called a **Mutex**. A binary semaphore is initialized to 1 and only takes the values 0 and 1 during the execution of a program. In Binary Semaphore, the wait operation works only if the value of semaphore = 1, and the signal operation succeeds when the semaphore= 0. Binary Semaphores are easier to implement than counting semaphores.

2. **Counting Semaphores:**

   These are used to implement **bounded concurrency**. The Counting semaphores can range over an **unrestricted domain**. These can be used to control access to a given resource that consists of a finite number of Instances. Here the semaphore count is used to indicate the number of available resources. If the resources are added then the semaphore count automatically gets incremented and if the resources are removed, the count is decremented. Counting Semaphore has no mutual exclusion.

Advantages of Semaphores

Benefits of using Semaphores are as given below:

- With the help of semaphores, there is a flexible management of resources.
- Semaphores are machine-independent and they should be run in the machine-independent code of the microkernel.
- Semaphores do not allow multiple processes to enter in the critical section.
- They allow more than one thread to access the critical section.
- As semaphores follow the mutual exclusion principle strictly and these are much more efficient than some other methods of synchronization.
- No wastage of resources in semaphores because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if any condition is fulfilled in order to allow a process to access the critical section.

Disadvantages of Semaphores

- One of the biggest limitations is that semaphores may lead to priority inversion; where low priority processes may access the critical section first and high priority processes may access the critical section later.

- To avoid deadlocks in the semaphore, the Wait and Signal operations are required to be executed in the correct order.
- Using semaphores at a large scale is impractical; as their use leads to loss of modularity and this happens because the wait() and signal() operations prevent the creation of the structured layout for the system.
- Their use is not enforced but is by convention only.
- With improper use, a process may block indefinitely. Such a situation is called **Deadlock**. We will be studying deadlocks in detail in coming lessons.