CSTechnocrat
+91-8354820003

Semaphore

The various hardware & software based solutions to the critical-section problem are having some limitations and complexities for application programmer to use. To overcome this difficulty, we can use a synchronization tool called a '*semaphore*'.
A semaphore S is an integer variable that can be accessed only through two standard atomic operations:

1. wait ( )
2. signal ( )

wait( )

The wait ( ) operations was originally termed 'P'.
Definition of wait ( )

```
wait(S)
 {
        while (S <= 0)
        ; // do nothing
        S - -;
}
```

signal( )

The signal ( ) operations was originally termed 'V'.
Definition of signal ( )

```
signal(S)
 {
        S ++;
}
```

When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Types of Semaphores

There are two types of semaphores:

1. **Counting Semaphore**
   The value of a counting semaphore can range over an unrestricted domain.

2. **Binary Semaphore / Mutex Locks**
   The value of a binary semaphore can be only 0 and 1.

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. Each process that wishes to use a resource performs a *wait ( )* operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a *signal( )* operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

CSTechnocrat
+91-8354820003

CSTechnocrat +91-8354820003

Binary Semaphore Solution for Critical Section

To deal with the problem of critical section for multiple processes we use binary semaphore.

Shared data

```
        semaphore mutex;
```

Initially

```
        mutex=1;
```

The Structure of Process P_i

```
do
{
        wait (mutex);
        //critical section
        signal (mutex);
        //remainder section
} while (true);
```

Disadvantage of semaphore

It requires **busy waiting**. While a process is in its critical section any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. Busy waiting cannot be avoided altogether.

This type of semaphore is also called a '**spinlock**' because the process "spins" while waiting for the lock. (Spinlocks do have an advantage in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful; they are often employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor).

To overcome the need for busy waiting, we can modify the definition of the *wait ( )* and *signal ( )* semaphore operations. When a process executes the *wait ( )* operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then, control is transferred to the CPU scheduler, which selects another process to execute.

The *block( )* operation suspends the process that invokes it. The *wakeup(P)*operation resumes the execution of a blocked process P. Under the classical definition of semaphores with busy waiting the semaphore value is never negative; this implementation may have negative semaphore values. If the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.

Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in two problems:

1. Deadlock
2. Starvation/ Indefinite Blocking

CSTechnocrat +91-8354820003

CSTechnocrat
+91-8354820003

Deadlock

Two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. E.g. consider a system consisting of two processes, P_0 and P_1, each accessing two semaphores, S and Q, set to the value 1:

```
        P0                      P1
wait (S) ;              wait (Q) ;
wait (Q) ;              wait (S) ;
…                       …
…                       …
…                       …
signal (S) ;            signal (Q);
signal (Q) ;            signal (S) ;
```

Starvation/ Indefinite Blocking

A process may never be removed from the semaphore queue in which it is suspended. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO order.

Questions asked in semester exam:

Question: Explain what semaphores are, their usage, implementation given to avoid busy waiting and binary semaphores.

[2018-2019] [7 Marks]

Question: Define Busy Waiting? How to overcome busy waiting using semaphore operations.

[2017-2018] [2 Marks]

Question: What is busy waiting?

[2015-2016] [2 Marks]

Question: What are semaphores?

[2015-2016] [2 Marks]

Question: Explain the Binary Semaphores with an example.

[2014-2015] [10 Marks]

Question: What is semaphore? Differentiate between binary and counting semaphore.

[2014-2015] [5 Marks]

Question: What are semaphores? What is the usage of semaphores? Explain with a suitable example.

[2014-2015] [10 Marks]

Question: Define semaphore with suitable examples.

[2013-2014] [5 Marks]

Question: Explain the binary semaphores with an example.

[2009-2010] [5 Marks]

Question: Define a critical section problem and its solution by using Semaphore. Use this approach to solve Producer/Consumer problem.

[2006-2007] [5 Marks]

CSTechnocrat
+91-8354820003