**SEMAPHORES**

In order to manage a concurrent processes in an operating system in 1965, Dutch computer scientist Edsger Dijkstra proposed a new technique known as Semaphore.

A semaphore is an integer variable that indicates the number of resources that are available in a system at a particular time and this semaphore variable is generally used to achieve the process synchronization. It is generally denoted by " S ". You can use any other variable name of your choice. This integer variable is accessed only through two standard atomic operations: wait () and signal (). The wait() operation was originally termed P (from the Dutch proberen, "to test"); signal() was originally called V (from verhogen, "to increment").

**Wait:** The wait() function is used to decrement the value of the semaphore variable " S " by one if the value of the semaphore variable is positive. If the value of the semaphore variable is 0 or less than 0, then no operation will be performed.

The definition of wait() is as follows:

wait(S)

{

while (S <= 0); // busy wait

S--;

}

**Signal:** This operation increments the actual value of its argument s.

The definition of signal() is as follows:

signal(S)

{

S++;

}

All the modifications to the integer value of the semaphore in the wait () and signal () operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait(S), the testing of the integer value of S (S < 0), and its possible modification (S—), must also be executed without interruption.

**i) Semaphore Usage:**

Semaphores are of two types:

1. Binary Semaphore
2. Counting Semaphore

**Binary Semaphore:**

In Binary semaphores, the value of the semaphore variable can range only between 0 and 1. Initially, the value of semaphore variable is set to 1 and if some process wants to use some resource, then the wait () function is called and the value of the semaphore is changed to 0 from 1. The process then uses the resource and when it releases the resource then the signal () function is called and the value of the semaphore variable is increased to 1. If at a particular instant of time, the value of the semaphore variable is 0 and some other process wants to use the same resource then it has to wait for the release of the resource by the previous process. In this way, process synchronization can be achieved.

**Counting Semaphore:**

The value of a counting semaphore can range over an unrestricted domain. Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. In Counting semaphores, firstly, the semaphore variable is initialized with the number of resources available. After that, whenever a process needs some resource, then the wait () function is called and the value of the semaphore variable is decreased by one. The process then uses the resource and after using the resource, the signal () function is called and the value of the semaphore variable is increased by one. So, when the value of the semaphore variable goes to 0 i.e all the resources are taken by the process and there is no resource left to be used, then if some other process wants to use resources then that process has to wait for its turn. In this way, we achieve the process synchronization.

**ii) Semaphore Implementation:**

The implementation of mutex locks suffers from busy waiting. The definitions of the wait() and signal() semaphore operations just described present the same problem. To overcome the need for busy waiting, we can modify the definition of the wait() and signal() operations as follows: 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.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a **wakeup() operation**, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

To implement semaphores under this definition, we define a semaphore as follows:

typedef struct

{

int value;

struct process \*list;

} semaphore;

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process.

Now, the wait() semaphore operation can be defined as

wait(semaphore \*S)

{

S->value--;

if (S->value < 0)

{

add this process to S->list;

block();

}

}

The signal() semaphore operation can be defined as

Signal(semaphore \*S)

{

S->value++;

if (S->value <= 0)

{

remove a process P from S->list;

wakeup(P);

}

}

The block () operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

Note that in this implementation, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.

**iii)Deadlocks and Starvation:**

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be deadlocked.

To illustrate this, consider a system consisting of two processes, P0 and P1, 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);

Suppose that P0 executes wait(S) and then P1 executes wait(Q).When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S). Since these signal() operations cannot be executed, P0 and P1 are deadlocked.

Another problem related to deadlocks is indefinite blocking or starvation, a situation in which processes wait indefinitely with in the semaphore. Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

**iv)Priority Inversion:**

A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process—or a chain of lower-priority processes. Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource. The situation becomes more complicated if the lower-priority process is preempted in favour of another process with a higher priority.

**Example:** Assume we have three processes—L, M, and H—whose priorities follow the order L < M < H. Assume that process H requires resource R, which is currently being accessed by process L. Ordinarily, process H would wait for L to finish using resource R. However, now suppose that process M becomes runnable, thereby preempting process L. Indirectly, a process with a lower priority—process M—has affected how long process H must wait for L to relinquish resource R.

This problem is known as **priority inversion.** It occurs only in systems with more than two priorities, so one solution is to have only two priorities. That is insufficient for most general-purpose operating systems, however. Typically, these systems solve the problem by implementing a **priority-inheritance protocol**. According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources. When they are finished, their priorities revert to their original values.

In the above example, a priority-inheritance protocol would allow process L to temporarily inherit the priority of process H, thereby preventing process M from preempting its execution. When process L had finished using resource R, it would relinquish its inherited priority from H and assume its original priority. Because resource R would now be available, process H—not M—would run next.