SNS COLLEGE OF TECHNOLOGY

Coimbatore-35.

An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A+' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

COURSE NAME : 19CSB201 – OPERATING SYSTEMS

II YEAR/ IV SEMESTER

UNIT – II Process Scheduling And Synchronization

Topic: Process Synchronization: Semaphores

Semaphores

A semaphore is a variable or abstract **data type used to control access to a common resource** by multiple threads and avoid critical section problems in a concurrent system such as a multitasking operating system. Semaphores are a type of synchronization primitive.

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait() and signal() .**

definition of wait() is as follows:

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

The definition of signal() is as follows:

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

Semaphore Usage

Operating systems often distinguish between **counting and binary semaphores**.

The value of a counting semaphore can range over an unrestricted domain.

The value of a binary semaphore can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks.

Counting semaphores

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.

In process P1 , we insert the statements

```
S1;
signal(synch);
```

In process P_2, we insert the statements

```
wait(synch);
S2;
```

Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch) , which is after statement S1 has been executed.

Semaphore Implementation

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

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

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();
        }
}
```

and 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.

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, 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);
```

Suppose that P_0 executes `wait(S)` and then P_1 executes `wait(Q)`. When P_0 executes `wait(Q)`, it must wait until P_1 executes `signal(Q)`. Similarly, when P_1 executes `wait(S)`, it must wait until P_0 executes `signal(S)`. Since these `signal()` operations cannot be executed, P_0 and P_1 are deadlocked.

starvation

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

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 favor of another process with a higher priority.

As an 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.

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 in question. When they are finished, their priorities revert to their original values. In the example above, 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.

REFERENCES

TEXT BOOKS:

T1 Silberschatz, Galvin, and Gagne, “Operating System Concepts”, Ninth Edition, Wiley India Pvt Ltd, 2009.)

T2. Andrew S. Tanenbaum, “Modern Operating Systems”, Fourth Edition, Pearson Education, 2010

REFERENCES:

R1 Gary Nutt, “Operating Systems”, Third Edition, Pearson Education, 2004.

R2 Harvey M. Deitel, “Operating Systems”, Third Edition, Pearson Education, 2004.

R3 Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, “Operating System Concepts”, 9th Edition, John Wiley and Sons Inc., 2012.

R4. William Stallings, “Operating Systems – Internals and Design Principles”, 7th Edition, Prentice Hall, 2011


THANK
YOU!