

**Department of Computer Science and Engineering**

**FACULTY OF ENGINEERING AND TECHNOLOGY  
UNIVERSITY OF LUCKNOW  
LUCKNOW**



**CS-501**

**Dr. Zeeshan Ali Siddiqui**  
**Assistant Professor**  
**Deptt. of C.S.E.**

# PROCESS SYNCHRONIZATION

**(Semaphores)**

# Semaphores

- A semaphore  $S$  is *a non-negative integer variable* that, apart from initialization, is accessed only through two standard atomic operations:

- `wait()`

- `signal()`

- `wait()` operation was originally termed as  $P$  (from the Dutch *proberen*, “to test”);

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

- `signal()` was originally called  $V$  (from *verhogen*, “to increment”).

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

# Types of Semaphores

- The two common kinds of *semaphores* are:
  - Counting semaphore
    - integer value can range over an *unrestricted* domain.
    - It may be used to implement the *solution* of critical section problem with multiple processes.
  - Binary semaphore
    - integer value can range only between *0 and 1*; can be simpler to implement, also known as *mutex* locks.
    - It may be used to control access to a resource that has *multiple* instances.

# Busy waiting<sup>1/3</sup>

- *The repeated execution of a loop of code while waiting for an event to occur is called busy-waiting.*
- CPU is not engaged in any real *productive* activity during this period and the process does not progress towards completion.
- When a process executes the wait() operation and finds that the semaphore value is not positive, it must *wait*.

# Busy waiting<sup>2/3</sup>

- 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.

# Busy waiting<sup>3/3</sup>

- 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:

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

# Implementation

- Semaphore Implementation with no *Busy waiting*

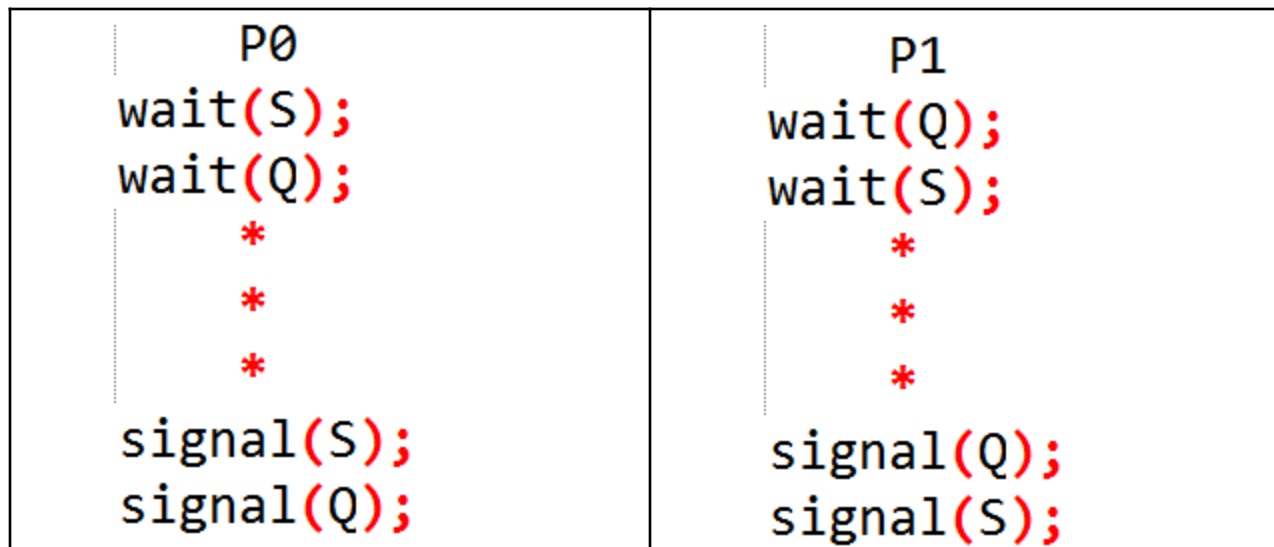
```
wait (S)
{
    S.value--;
    if (S.value < 0)
    {
        add this process to waiting queue
        block();
    }
}
```

```
Signal (S)
{
    S.value++;
    if (S.value <= 0)
    {
        remove a process P from the waiting queue
        wakeup(P);
    }
}
```



# Deadlock<sub>1/2</sub>

- Two or more processes are waiting *indefinitely* for an event that can be caused only by one of the waiting processes.
- Consider a system consisting of two *processes*, P0 and P1, each accessing two semaphores, S and Q, set to the value 1:



# Deadlock<sub>2/2</sub>

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.

# Starvation

- *Indefinite blocking.*
- A process may never be removed from the *semaphore queue* in which it is suspended.

# References

1. Silberschatz, Galvin and Gagne, “Operating Systems Concepts”, Wiley.
2. William Stallings, “Operating Systems: Internals and Design Principles”, 6<sup>th</sup> Edition, Pearson Education.
3. D M Dhamdhere, “Operating Systems: A Concept based Approach”, 2<sup>nd</sup> Edition, TMH.

**Thank You.**

