OPERATING SYSTEMS

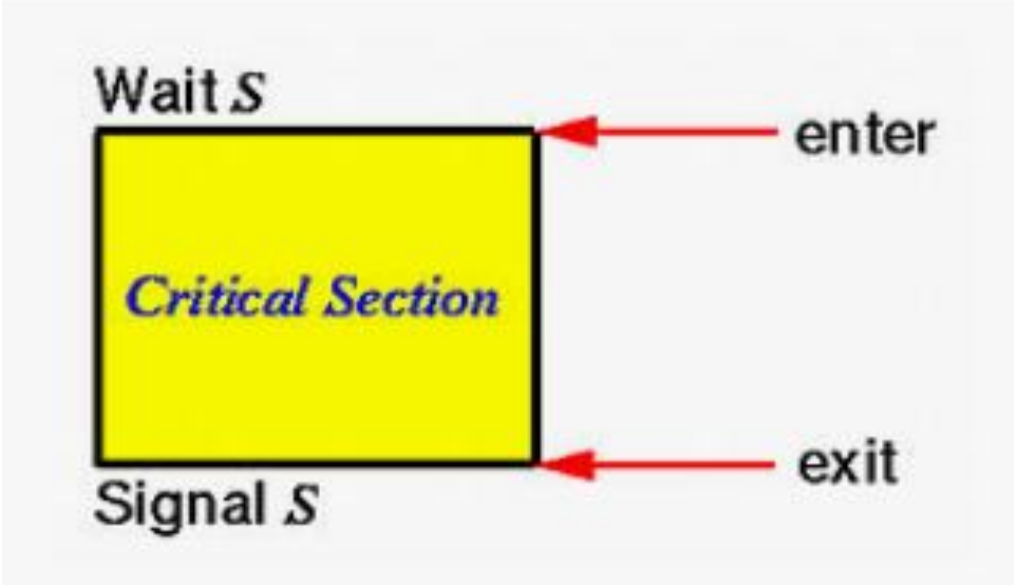
Razi Uddin

Lecture # 17

SEMAPHORES

SEMAPHORES

A semaphore S is an integer variable that, apart from initialization is accessible only through two standard atomic operations: wait and signal. These operations were originally termed P (for wait) and V (for signal).



SEMAPHORES

```
wait(S) {
   while(S<=0)
   ;// no op
   S--;
}
```

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

SEMAPHORES

- Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly.
- That is, when one process is updating the value of a semaphore, other processes cannot simultaneously modify that same semaphore value.
- In addition, in the case of the wait(S), the testing of the integer value of S (S<=0) and its possible modification (S--) must also be executed without interruption.

SEMAPHORES

- We can use semaphores to deal with the n-process critical section problem.
- The n processes share a semaphore, mutex (standing for mutual exclusion) initialized to 1.
- Each process Pi is organized as follows:

```
do
{
        wait(mutex);
                Critical section
        signal(mutex);
                Remainder section
} while(1);
```

SEMAPHORES

Is it a good solution?

- Answer is no.
- Mutual exclusion==Yes
- Progress== Yes
- Bounded Waiting== No

SEMAPHORES

- In a uni-processor environment, to ensure atomic execution, while executing wait and signal, interrupts can be disabled.

- In case of a multi-processor environment, to ensure atomic execution is one can lock the data bus, or use a soft solution such as the Bakery algorithm.

SEMAPHORES (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 may be able to use productively.
- **This type of semaphore is also called a spinlock** (because the process spins while waiting for the lock).
- Spinlocks are useful in multiprocessor systems.
- The advantage of a spinlock is that no context switch is required when a process must wait on a lock, and a context switch may take considerable time.
- Spinlocks are useful when they are expected to be held for short times.

SEMAPHORES (MODIFIED)

- To overcome the need for busy waiting, we can modify the definition of semaphore and the wait and signal operations on it.
- When a process executes the wait operation and finds that the semaphore value is not positive, it must wait.
- However, rather than 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.

SEMAPHORES (MODIFIED)

- 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. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU scheduling algorithm.)

SEMAPHORES (MC

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

- Each semaphore has an integer value and a list of processes.
- 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 the waiting processes and awakens that process.

```
void wait(semaphore S) {
  S.value--;
  if(S.value < 0) {
     add this process to S.L;
     block();
  }
}
```

```
void signal     (semaphore S) {
  S.value++;
  if(S.value <= 0) {
     remove a process P from S.L;
     wakeup(P);
  }
}
```

SEMAPHORES (MODIFIED)

- 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.
- The negative value of S.value indicates the number of processes waiting for the semaphore.
- A pointer in the PCB needed to maintain a queue of processes waiting for a semaphore.
- As mentioned before, the busy-waiting version is better when critical sections are small and queue-waiting version is better for long critical sections (when waiting is for longer periods of time).

Problems

PROCESS SYNCHRONIZATION (EXAMPLE-1)

Consider, for example, that you want to execute statement B in Pj only after statement A has been executed in Pi. You can solve this problem by using a semaphore S initialized to 0.

Solution: S=0

Pi	pj
-	-
-	Wait(S)
A	**B**
Signal(S)	-
-	-

PROCESS SYNCHRONIZATION (EXAMPLE-2)

We want to ensure that statement S1 in P1 executes only after statement S2 in P2 has been executed, and statement S2 in P2 should execute only after statement S3 in P3 has been executed.

Solution: S1=0,S2=0

P1	P2	P3
-	-	-
wait(S2)	wait(S1)	
S1	S2	S3
	signal(S2)	signal(S1)
-	-	-

PAST PAPER QUESTIONS

Consider multiple threads executing the following two functions. These threads print a string containing any number of a's and b's in any order. Synchronize the threads (using Semaphores) so that the string becomes a concatenation of the substring "ab".

- Following are few examples:
- ab (correct)
- ab ab (correct)
- ba (incorrect)
- ab ba (incorrect)

Solution: S1=1,S2=0

T1

```
void fun_1() {
  wait(S1);
 cout<<"a";
  signal(S2);
}
```

T2

```
void fun_2() {
   wait(S2);
 cout<<"b";
   signal(S1);
}
```

Now change the same functions given above so that the string becomes a concatenation of the substring ”abb”.

Following are few examples:

- abb (correct)
- abb abb (correct)
- ab (incorrect)
- ab ab (incorrect)
- bba (incorrect)
- bba bba (incorrect)

Solution: S1=1,S2=0 **static int counter=0;**

T1

```
void fun_1() {
wait(S1);

cout<<“a”;

signal(S2);
}
```

T2

```
void fun_2() {
wait(S2);
counter++;
cout<<“b”;
if(counter%2==0){
signal(S1);
}
else{ signal(S2); }

}
```

PROBLEMS WITH SEMAPHORES

- Semaphores provide a powerful tool for enforcing mutual exclusion and coordinating processes.
- The wait(S) and signal(S) operations are scattered among several processes. Hence, it is difficult to understand their effects.
- Usage of semaphores must be correct in all the processes.
- One bad (or malicious) process can fail the entire system of cooperating processes.
- Incorrect use of semaphores can cause serious problems.

DEADLOCKS

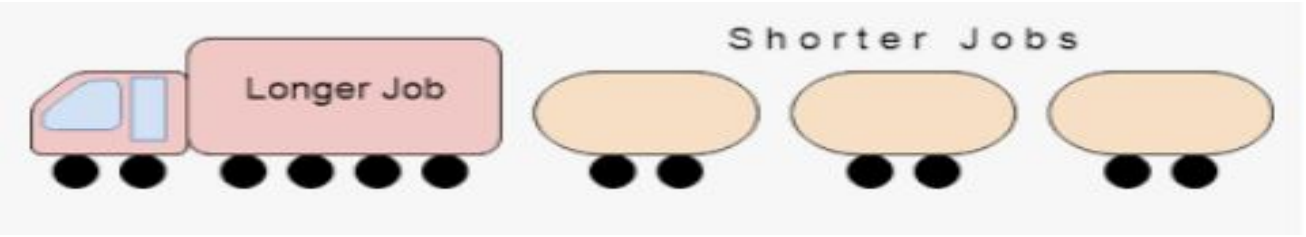
- A set of processes are said to be in a deadlock state if every process is waiting for an event that can be caused only by another process in the set
- Example of Deadlock

✔ Traffic deadlocks

✔ One-way bridge-crossing

P0	P1
wait(S);	wait(Q);
wait(Q);	wait(S);
...	...
signal(S);	signal(Q);
signal(Q);	signal(S);
...	...

STARVATION

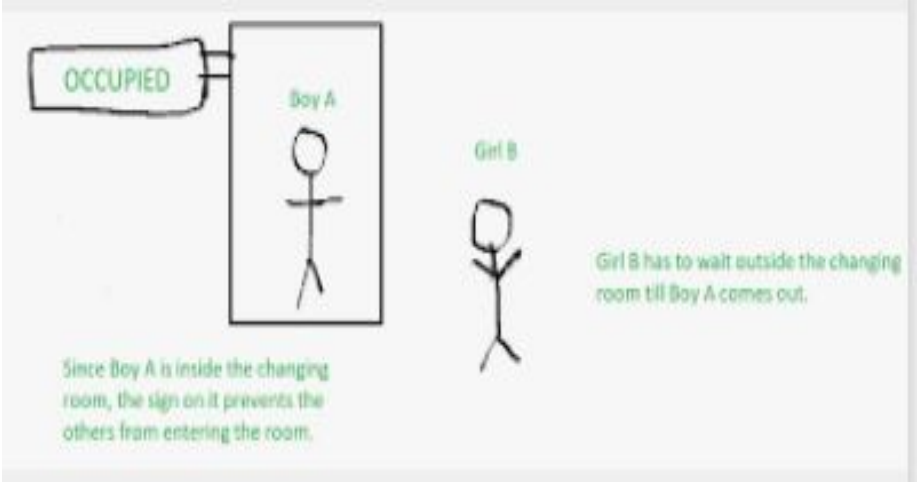


- Starvation is infinite blocking caused due to unavailability of resources.

```
P0                P1

wait(S);          wait(S);
...               ...
wait(S);          signal(S);
...               ...
```

VIOLATION OF MUTUAL EXCLUSION




```
P0                P1
signal(S);        wait(S);
...               ...
wait(S);          signal(S);
...               ...
```

- These problems are due to programming errors because of the tandem use of the wait and signal operations.
- The solution to these problems is higher-level language constructs such as critical region (region statement) and monitor.