CS 340
Lecturer: Dr. Simina Fluture

Lecture # 14

Topics: **The Concept of Semaphores**
Operations on Semaphores
Types of Semaphores
Implementation of Mutual Exclusion using semaphores
Implementation of Operation - Serialization and process - Synchronization
The Active V operation

Read: **Lecture Notes**
Textbook: **Semaphores**

Semaphores

We have seen how Mutual Exclusion can be implemented with no operating system support.

Semaphores provide a primitive yet powerful tool for enforcing mutual exclusion and for coordinating processes. Semaphores need operating system support.

Apart from **initialization**, **two atomic operations** can be performed on semaphores: **wait** and **signal**.

Initial definition of wait and signal (P and V)

```
wait(S):    while S ≤ 0 do no-op;
            { S = S--; }

signal(S):   S = S++;
```

Mutual Exclusion implementation: n processes share a semaphore, named **mutex**.
Initially

```
While(true) {
        wait(mutex);                // P(mutex)
                CS
        signal(mutex);              // V(mutex)
        remainder section;
}
```

While a process is in its CS, any other process that attempts to enter its CS, must loop continuously in the entry code.

The process is busy waiting, wasting CPU cycles. This kind of semaphore is also called a **spinlock semaphore**.

Overcoming the need for busy waiting:
Rather then busy waiting, the process can block itself.

Classical definition of wait and signal

Note: this is the definition that will be considered from now on in the implementation of any problem.

There are two fields associated with the semaphore: semaphore value (integer) and semaphore queue (which contains a list of processes blocked on the semaphore).

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


 V(S){
     S.value ++;
     if (S.value ≤ 0) {
         remove a process P from  S.queue;
         wakeup(P);                                    // release //
         }
 }
```

There are two kinds of semaphores:
Counting semaphores: can assume (initialized) nonnegative integer values.
Wait - resource has been allocated.
Signal - resource has been returned to the pool.

Binary semaphores: can assume only values 1 and 0.

Lecture Homework

What will be the outcome of changing the Signal definition for Binary Semaphores to:

```
V(S) {
        s.value = 1;
        remove a process P from s.queue;
        wakeup(P);
    }
```

Using semaphores for solving synchronization problems (completed in class)
Consider two concurrently running threads. T1 executes statement S1 and T2 executes statement P2.
Using semaphores, synchronize the execution of the two threads such that S2 will be executed only after S1 has been completed.

```
 T1                      T2
..................     .........................
 S1                      S2
.................      ..........................
```

Serialization between read and write **operations** (completed in class)
Each of the threads below reads or writes a specific variable. The read operation should be done only after the variable has been updated (written).

Semaphores:

```
ThreadA                                   ThreadB

       compute A1                              read x;
       write x;                                compute B1;
       compute A2;                             write y;
       read y;                                 compute B2;
```

Deadlock and Starvation (completed in class)
*) Deadlock
S and Q are two semaphores set to value 1.

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


signal(S);      signal(Q);
signal(Q);      signal(S);
```

Starvation
This is likely to happen in a multiprogramming (one processor) environment if the process that executes the signal does not immediately release the CPU after incrementing the semaphore.