❖ Semaphores

A **semaphore S** is an integer variable that is accessed only through two standard **atomic** operations: **wait( )** and **signal( )**. The definition of **wait( )** is as follows:

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

The definition of **signal( )** is as follows:

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

All modifications to the integer value of the semaphore (**S**) in the **wait**( ) and **signal**( ) operations must be executed **indivisibly** (without interruption). That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

❖ Semaphore Usage

There are two types of semaphore:

1. **Binary semaphore:** integer value range between 0 and 1 (equivalent to a mutex lock).

```
wait(sync);   // sync initialize to 1
Critical section
signal(sync);
Remainder section
```

2. **Counting semaphore:** integer value range over an unrestricted domain. It 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( )** on the semaphore (decrementing the count). When a process releases a resource, it performs a **signal( )** (incrementing the count). When the count 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**.

Example: consider two concurrently running processes:

- ***P*1** with a statement ***S*1** and ***P*2** with a statement ***S*2**.
- Suppose we require that ***S*2** be executed only after ***S*1** has completed.
- ***P*1** and ***P*2** share a common semaphore **synch**, initialized to 0.

***P*1:**
```
S1;
signal(synch);
```

***P*2:**
```
wait(synch);
S2;
```

Because *synch* is initialized to 0, *P*2 will execute *S*2 only after *P*1 has invoked signal(synch), which is after statement *S*1 has been executed.

Exercise: Consider a system consisting of two process **Pi** and **Pj**, each accessing two semaphores, **S** and **Q**, both are set to the value **1**. Give the output after executing the two processes concurrently.

***P*i:**
```
wait(S);
wait(Q);
printf ("Process i");
signal(S);
signal(Q);
```

***P*j:**
```
wait(Q);
printf ("Process j");
signal(Q);
```

❖ Deadlock and Starvation

- **Deadlock:** two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

Let ***S*** and ***Q*** be two semaphores initialized to **1**:

P_0
```
wait(S); //exec 1st
wait(Q); //exec 3rd
   ...
signal(S);
signal(Q);
```

P_1
```
wait(Q); //exec 2nd
wait(S); //exec 4th
   ...
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 (Indefinite Blocking)**: a situation in which processes wait indefinitely within the semaphore.

❖ The Bounded-Buffer Problem

- Assume that we have **n** buffers, each capable of holding one item.
- **mutex** semaphore provides mutual exclusion for accessing buffers (initialized to **1**).
- **empty** and **full** semaphores count the number of empty and full buffers. Semaphore **empty** is initialized to the value **n**; while semaphore **full** is initialized to the value **0**.

▪ **The structure of the producer process:**

```
do {
            /* produce an item in next_produced */
     wait(empty);
     wait(mutex);
    ...
  /* add next produced to the buffer */
    ...
     signal(mutex);
     signal(full);
} while (true);
```

▪ **The structure of the consumer process:**

```
do {
  wait(full);
  wait(mutex);
    ...
  /* remove an item from buffer to next_consumed */
    ...
  signal(mutex);
  signal(empty);
    /* consume the item in next consumed */
  } while (true);
```

❖ Problems with Semaphores

- Incorrect use of semaphore operations can be difficult to detect, e.g. :
  - Inversed order: **signal(mutex) ... wait(mutex)**
  - Repeated calls: **wait(mutex) ... wait(mutex)**
  - Omitted calls: **wait(mutex)** or **signal(mutex)** (or both)
- Results in deadlock and/or starvation