Semaphores

- So, semaphore is a synchronization mechanism which does not require busy waiting.

- ***What is a Semaphore?***

Think of this as an integer variable, a kernel variable, a shared variable, is accessed using 2 standard (atomic) operations - wait() and signal() [alternative references: P() and V()]

-- Semaphore variable cannot be directly modified;

-- wait() and signal() are atomic operations - implemented as a part of kernel;

-- A semaphore variable is initialized with an integer value initially;

Semantics of the operations wait() & signal()

- wait(S) operation: /* Indivisible (until the calling process is blocked) */

  - Decrement S

  - If S>0: return will not cause the process to block;

  - if S < 0: the calling process is put to sleep(blocked) by this wait() until some process does a signal() and this sleeping process is selected to wake-up;

- signal(S) /* Indivisible (never blocks the calling process) */

  - S is incremented by 1;

  - If S < 0: a blocked process on S is selected and woken up and signal() returns;

Note- Sleeping activity does not cause CPU cycles wasted;

Implementing the operations wait() & signal()

A **waiting queue** is associated with each semaphore

- It stores the processes waiting on the semaphore

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

Two operations:

- **block** – place the process invoking the operation on the appropriate waiting queue
- **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Implementing the operations wait() & signal()

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block(); //suspends self, sleeps, avoids CPU cycles
   }
}
```

Sleep();

```
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Types of Semaphores

Two kinds: Counting Semaphore & Binary Semaphore;

Counting Semaphore: Assumes an integer value greater than 0 initially; Integer value can range over an unrestricted domain;

Use: Can control access to resource that has a finite number of instances; Say, it can be initialized to the number of resources available;

Binary Semaphore: Assumes integer values 0 / 1; Special case of counting semaphore;

Usage of Semaphores: CS access

- Binary semaphores can be used to solve CS problem;
- A semaphore variable (say, mutex) is shared by N processes and it is initialized to 1;
- Each process is structured as follows:

```
do {
        wait(mutex);
        Critical Section;
        signal(mutex);
        remainder section;
}while(True);
```

How does this ensure that there is only one process in the CS?

Usage of Semaphores: CS access (Example)

Tracking S value

S value	P0	State	P1	State
1		Running		Ready
1	Call wait(S)	Running		Ready
0	Wait() executed	Running		Ready
0	CS Enter	Running		Ready
0	ProSw(1)	Ready		Running
0		Ready	Call wait(S)	Running
-1		Ready	Wait() executed	Running
-1		Ready	If S<0: sleep()	Sleeping
-1		Running	ProSw(0)	Sleeping

Example (cont'd)…

S value	P0	State	P1	State
-1	CS: end	Running		Sleeping
-1	Call signal(S)	Running		Sleeping
0	Inc S	Running		Sleeping
0	wake(1)	Running		Ready
0	ProSw(1)	Ready		Running
0		Ready	wait(S) returns	Running
0		Ready	CS Enter	Running
0		Ready	Call signal(S)	Running
1		Ready	Inc S	Running

CS completes after this step

Usage of Semaphores: Solving Producer-Consumer Problem

Definitions for the semaphore variables:

A - Provides mutual exclusion; U & B - allows Prod() & Cons() to communicate and help in synchronization;

U = prevents underflow - indicates the number of items in the buffer to consume; If it goes negative it gives an indication of the number of threads that are blocked on this Semaphore;

B - amount of free space left in the buffer which helps to prevent overflow; As items are added the B count goes down and when it becomes negative, it indicates the num of threads blocked to add items to the buffer;

Usage of Semaphores: Solving Producer-Consumer Problem

```
int buffSize = "size";
Semaphore A = 1; // Controls buffer Access
Semaphore U = 0; // Controls buffer Underflow
Semaphore B = buffSize;  // Avoids buffer Overflow
```

```
void producer() {
        while(true){
                X = produce();
                SemWait(B);
                SemWait(A);
                append(X);
                SemSignal(A);
                SemSignal(U);
        }
        return;
}
```

Exclusive access to buffer

```
void consumer() {
        while(true){
                SemWait(U);
                SemWait(A);
                X = read();
                SemSignal(A);
                SemSignal(B);
                consume(X);
        }
        return;
}
```

Exclusive access to buffer

Solving Producer-Consumer Problem (Cont'd)

- Producer() Code description:

Producer() generates an item (copies to X to be written to the buffer eventually) and calls SemWait(B); Thus it decrements and if decrement does not cause any issues (meaning free slot is still available) it executes SemWait(A) through which it gets exclusive access to buffer to write. Then it appends X in the buffer and then releases the buffer by calling SemSignal(A); Then it calls SemSignal(U); Note that U indicates how many items are available for consumption; Hence, by incrementing it says items are available to consume - This is the way to communicate to the consumer() code.

Solving Producer-Consumer Problem (Cont'd)

- Consumer() Code description:

First thing it does is that it waits on semaphore U by calling SemWait(U) - rather it is forced to wait until something is produced and that signal comes from the producer as soon as the prod() writes it in the buffer (See the prod() code); Since there will be no block when something is there to consume, it calls SemWait(A) to get exclusive rights and consumes the item and then releases the buffer by calling SemSignal(A); Then call SemSignal(B) to signal the producer that it has created space for the next item to be produced in the buffer;

Drawbacks with Semaphores

- Incorrect use of semaphore operations:

(1) Program can be more complex to debug

- Wrong order: signal (mutex) …. Wait (mutex);
- Wrong calls: wait (mutex) … wait (mutex)

(2) Omitting of wait (mutex) or signal (mutex) (or both)

  - Timing errors –could occur only under certain circumstances and won't occur otherwise
  - Run a program and it crashes, run it again and it doesn't!

(3) Deadlock and starvation are possible

Solution ? - Use of Monitors

Monitors

Programming paradigm that exercises control on signaling mechanism between processes / threads, towards synchronization.

Monitor is a synchronization construct that achieves Mutual Exclusion;

It uses Conditional Variables(CVs): User-specific objects that are not shared among the processes; CVs are fundamental synchronization primitives that triggers an event depending on particular conditions that occur;

Using CVs, processes/threads attempt to acquire and release locks;

Programming language specific design; Example – Java monitors, using Java Virtual Machine