KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

Information and Computer Science Department

ICS 431 Operating Systems

Lab # 9

Semaphores

Objectives:

In this lab, we will use semaphore to solve various synchronization problems. Semaphores are counters for resources shared between processes/threads. The basic operations on semaphores are: increment the counter atomically, and wait until the counter is non-null and decrement it atomically.

This lab has two primary objectives:

1. How to use semaphore to solve various synchronization problems
2. Implementing the POSIX.1b semaphore functions which include
   - Initializing semaphores
   - Decreasing the count of a semaphore.
   - Increasing the count of a semaphore.
   - Destroying semaphores.

Introduction

So far we have implemented mutexes and condition variables as the tools of synchronization and managing critical sections of code. These are not the only tools that can be used for the job. Semaphore is another tool to synchronize activities in a computer.

The concept of semaphores as used in computer synchronization is due to the Dutch computer scientist *Edsgar Dijkstra*. They have the advantages of being very simple, but sufficient to construct just about any other synchronization function you would care to have.

There are several versions to implement semaphores. The most common versions are:

1) POSIX semaphores
2) System V IPC semaphores

This lab will consider only POSIX semaphore, since POSIX semaphores has very clear API functions to perform semaphore operations. However, it is more efficient to use System V semaphore than POSIX semaphore when semaphores are shared between processes.

What is a Semaphore?

A semaphore is an integer variable with two atomic operations:

1) **wait**. Other names for **wait** are **P**, **down** and **lock.**
2) **signal**: Other names for **signal** are **V**, **up**, **unlock** and **post**.

The simplest way to understand semaphore is, of course, with code. Here is a little pseudo-code that may help.

```
typedef struct SEMAPHORE {
     int value;     /* the integer value for semaphore */
     other stuff;
} sem_t;

init(sem_t *S, int i)
{
     S->value = i;
}

void wait(sem_t *S)
{
     S->value--;
     if (S->value < 0)
          block on semaphore
}

void signal(sem_t *S)
{
     S->value++;
     if (S->value <= 0)
          unblock one process or thread that is blocked on semaphore
}
```

Basic Synchronization Patterns

This section presents a series of examples of basic synchronization problems. It shows ways of using semaphore to solve them.

Example 1 (Signaling)

This example represents the simplest use for a semaphore (which is **signaling**), in which one process/thread sends a signal to another process/thread to indicate that something has happened. Suppose process one must execute statement **a** before process two executes statement **b**. To solve this synchronization problem, we need a semaphore (say **sync**) with initial value **0**, and that both processes have shared access to it. The following pseudo-code describes the solutions:

```
init(&sync, 0);
```

Process One

```
a;
signal(&sync);
```

Process Two

```
wait(&sync);
b;
```

Example 2 (Rendezvous)

Puzzle: Generalize the signal pattern so that it works both ways. Process One has to wait for Process Two and vice versa. In other words, given this code

Process One

```
a1;
a2;
```

Process Two

```
b1;
b2;
```

we want to guarantee that **a1** happens before **b2** and **b1** happens before **a2**. Please try to think how to solve this puzzle before proceeding to next paragraph.

Hint: You need to use two semaphores

```
init(&sem1, 0);

init(&sem2, 0);
```

Deadlock:

While working on the previous problem, you might have tried something like this

Process One

```
a1;
wait(&sem2);
signal(&sem1);
a2;
```

Process Two

```
b1;
wait(&sem1);
signal(&sem2);
b2;
```

if so, I hope you reject it quickly, because it has a serious problem (**DEADLOCK**). Can you explain how?

Solution 1:

Process One

```
a1;
signal(&sem1);
wait(&sem2);
a2;
```

Process Two

```
b1;
signal(&sem2);
wait(&sem1);
b2;
```

Solution 2:

Process One

```
a1;
wait(&sem2);
signal(&sem1);
a2;
```

Process Two

```
b1;
signal(&sem2);
wait(&sem1);
b2;
```

This solution is probably less efficient, since it might have to switch between Process One and Process Two more than necessary.

Example 3 (Mutual exclusion)

This example illustrates how we can use a semaphore to guarantee that only one process/thread accesses a shared memory (or global variables).

Suppose two threads try to access a global variable `count`. To enforce mutual exclusion, we need a semaphore (say **`mutex`**) with initial value **`1`**. The following pseudo-code describes the solution:

```
init(&mutex, 1);
```

Thread A

```
wait(&mutex);
/* CS */
count = count + 1;
signal(&mutex);
```

Thread B

```
wait(&mutex);
/* CS */
count = count + 1;
signal(&mutex);
```

Quiz

Consider the following pseudo code:

```
Process 1: executes
     for (;;) {
          wait(&S);
          a;
          signal(&Q);
     }
```

```
Process 2: executes
     for (;;) {
          wait(&Q);
          b;
          signal(&S);
     }
```

1) What happens if the semaphores S and Q are both initially 1?
2) What happens if one semaphore is initially 1 and the other 0?
3) What happens if both semaphores are initially 0?

POSIX Semaphore

Semaphores are part of the POSIX.1b standard adopted in 1993. The POSIX.1b standard defines two types of semaphores: *named* and *unnamed*. A POSIX.1b *unnamed semaphore* can be used by a single process or by children of the process that created them. A POSIX.1b *named semaphore* can be used by any processes. In this section, we will consider only how to initialize unnamed semaphore.

The following header summarizes how we can use POSIX.1b unnamed semaphore:

Header file name	`#include <semaphore.h>`
Semaphore data type	`sem_t`
Initialization	`int sem_init(sem_t *sem, int pshared, unsigned value);`

Semaphore Operations	`int sem_destroy(sem_t *sem);`
	`int sem_wait(sem_t *sem);`
	`int sem_post(sem_t *sem);`
	`int sem_trywait(sem_t *sem);`

Compilation	`cc filename.c -o filename -lrt`

All of the POSIX.1b semaphore functions return `-1` to indicate an error.

`sem_init` function initializes the semaphore to have the value `value`. The `value` parameter cannot be negative. If the value of `pshared` is not `0`, the semaphore can be used between processes (i.e. the process that initializes it and by children of that process). Otherwise it can be used only by threads within the process that initializes it.

`sem_wait` is a standard semaphore wait operation. If the semaphore value is 0, the `sem_wait` blocks unit it can successfully decrement the semaphore value.

`sem_trywait` is similar to `sem_wait` except that instead of blocking when attempting to decrement a zero-valued semaphore, it returns `-1`.

`sem_post` is a standard semaphore signal operation. The POSIX.1b standard requires that `sem_post` be reentrant with respect to signals, that is, it is asynchronous-signal safe and may be invoked from a signal-handler.

Example (Signaling):

Lab1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

/* Global variables */
int x = 0;
sem_t sync;

/* Thread function */
void *my_func(void *arg)
{
    /* wait for signal from main thread */
    sem_wait(&sync);
    printf("X = %d\n", x);
}

void main ()
{
    pthread_t thread;

    /* semaphore sync should be initialized by 0 */
    if (sem_init(&sync, 0, 0) == -1) {
        perror("Could not initialize mylock semaphore");
        exit(2);
    }

    if (pthread_create(&thread, NULL, my_func, NULL) < 0) {
        perror("Error: thread cannot be created");
        exit(1);
    }

    /* perform some operation(s) */
    x = 55;
    /* send signal to the created thread */
    sem_post(&sync);

    /* wait for created thread to terminate */
    pthread_join(thread, NULL);

    /* destroy semaphore sync */
    sem_destroy(&sync);

    exit(0);
}
```

Example (Critical Section):

Lab2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

/* Global variables */
int   x = 0;
sem_t m;

/* Thread function */
void *thread(void *arg)
{
    /* critical section */
    sem_wait(&m);   /* lock the mutex m */
    x = x + 1;
    sem_post(&m);   /* unlock the mutex m */
}

void main ()
{
    pthread_t tid[10];
    int i;

    /* semaphore m should be initialized by 1 */
    if (sem_init(&m, 0, 1) == -1) {
        perror("Could not initialize mylock semaphore");
        exit(2);
    }
     /* create TEN threads */
    for (i=0; i<10; i++)
    {
        if (pthread_create(&tid[i], NULL, thread, NULL) < 0) {
           perror("Error: thread cannot be created");
           exit(1);
        }
    }
    /* wait for all created thread to terminate */
    for (i=0; i<10; i++) pthread_join(tid[i], NULL);
    printf("Final value of x is %d\n", x);
    exit(0);
}
```

Exercises

Note:

Lab problems will be given during the lab based on material covered in this lab manual.