بسم الله الرحمن الرحيم

EE 463 : Operating Systems
Faculty Of Engineering
King Abdulaziz University
Department Of Electrical and Computer Engineering

LAB #7

Process Synchronization Tools

Semaphores

Objective:

Modern operating systems support multi-processing and provide programming language libraries to create processes using the `fork()` and `exec()` system calls. These processes may share some common data structures where they may have read and write access rights. To ensure data integrity, some way to *synchronize* access to the common data objects is needed. There are several techniques which can be used to do that, but in this lab you will learn how POSIX *semaphores* can be used to solve various synchronization problems.

You can skip the discussions and go directly to the numbered instructions written in RED

Semaphores:

Semaphores are counters for resources shared between processes and/or threads. Thus, a semaphore can be thought of as an integer variable, whose value is never allowed to fall below zero, with two basic operations:

1) **wait** (Other names are: **P**, **down** and **lock**): Where the value of the variable is decremented atomically, if non-zero; otherwise the function blocks until it becomes possible to perform the decrement operation.

2) **signal** (Other names are **V**, **up**, **unlock** and **post**): Where the value of the variable is incremented atomically.

The most commonly used semaphore system in UNIX is system V semaphores. They are quite powerful, but not easy to use. A much easier to use implementation of semaphores is POSIX semaphores. In this lab we are going to discuss the following topics related to using POSIX semaphores:

- Creating and initializing semaphores
- Operating on a semaphore through its wait and signal functions
- Destroying semaphores

POSIX Semaphores:

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

Using POSIX Semaphores:

A program that uses POSIX semaphore functions must start with the following line:

```
#include <semaphore.h>
```

It must also be compiled with **-lrt** option as follows:

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

OR

cc filename.c -o filename -pthread
```

Unnamed Semaphores:

This kind of semaphores (also known as *memory-based* semaphores), does not have a name. Instead the semaphore is placed in a region of memory that is shared between multiple threads (called, a thread-shared semaphore) or multiple processes (called, a process-shared semaphore).

- A thread-shared semaphore is placed in an area of memory shared between the threads of a process, for example, a global variable.
- A process-shared semaphore must be placed in a shared memory region (e.g., a shared memory segment created using **shmget()**).

Before being used, an unnamed semaphore must be *initialized* using **sem_init()**. It can then be *operated* on using **sem_wait()** and **sem_post()**. When the semaphore is no longer required, and before the memory in which it is placed is deallocated, the semaphore should be *destroyed* using **sem_destroy()**.

Next, we are going to describe some specific details of the POSIX semaphore functions. Note that all of the POSIX semaphore functions return **-1** to indicate an error.

Semaphore Initialization - sem_init():

The **sem_init()** function initializes the semaphore to have the initial value, **value**. It can be called using the following syntax:

```
int sem_init (sem_t *sem, int pshared, unsigned value);
```

Where, **sem_t** is the data type of the semaphore, whose pointer **sem** is returned by the function. 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. Initializing a semaphore that has already been initialized, results in an undefined behavior. When successful, the function returns **0**.

Semaphore Wait Operation - `sem_wait()`:

This is a standard semaphore wait operation. It can be called using the following syntax:

```
int sem_wait (sem_t *sem);
```

This function decrements (locks) the semaphore pointed to by **sem**. If the semaphore's value is greater than zero, then the decrement is done, and the function returns, immediately. If the semaphore value is **0**, then **sem_wait()** blocks until it can successfully decrement the semaphore value. When successful, the function returns **0**.

Semaphore Signal Operation - `sem_post()`:

This is a standard semaphore signal operation. The POSIX 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. It can be called using the following syntax:

```
int sem_post (sem_t *sem);
```

This function increments (unlocks) the semaphore pointed to by **sem**. If the semaphore's value, as a result, becomes greater than zero, then another process or thread blocked in a **sem_wait()** call will be woken up and proceed to lock the semaphore. When successful, the function returns **0**.

Semaphore Destruction - `sem_destroy()`:

The **sem_destroy()** function destroys the unnamed semaphore at the address pointed to by **sem**. It can be called using the following syntax:

```
int sem_destroy (sem_t *sem);
```

Only a semaphore that has been initialized by **sem_init()** should be destroyed using this function. Destroying a semaphore that other processes or threads are currently blocked on (in **sem_wait()**) produces undefined behavior. Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialized using **sem_init()**.When successful, the function returns **0**.

Basic Synchronization Patterns:

We present here two cases of basic synchronization problems, where semaphores can be used to solve them.

1. **Signaling:**

   This is the simplest use for a semaphore, in which one process or thread sends a signal to another process or thread to indicate that something has happened. Suppose thread **one** must execute statement S_1 before thread **two** executes statement S_2. To solve this synchronization problem, we use one semaphore (say **sync**) with initial value **0**, where both processes have shared access to it. The following code describes the solution:

Initialize by: `sem_init(&sync, 0, 0);`

Thread one

```
S1;
sem_post(&sync);
```

Thread two

```
sem_wait(&sync);
S2;
```

Experimental Program:

```
/* Semaphore Signaling example */
#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);
  }

  // wait for a while
  sleep(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);
}
```

1) Run the above program several times.
2) Observe and take a snapshot of the program output, every time it is different:

   X = 55

3) Remove the semaphore statements (in red color) from the above program.
4) Run the changed program several times.
5) Observe and take a snapshot of the program output, every time it is different.

2. Mutual exclusion:

In mutual exclusion problems, semaphores can be used to guarantee that only one process or thread accesses a shared memory segment (or global variables) at a time. Suppose two threads, **A** and **B**, try to access a global variable **`count`**. To enforce mutual exclusion, we need a semaphore (say **`mutex`**) with initial value **`1`**, where both processes have shared access to it. The following code describes the solution:

Initialize by: `sem_init(&mutex, 0, 1);`

Thread A

```
sem_wait(&mutex);
count = count+1;
sem_post(&mutex);
```

Thread B

```
sem_wait(&mutex);
count = count-10;
sem_post(&mutex);
```

Experimental Program:

```
/* Mutual exclusion semaphore example */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

// Global variables
int x = 0;
sem_t mutex;

// Thread function
void *thread(void *arg) {

   // critical section
   sem_wait(&mutex);          /* lock the mutex semaphore */
   x = x + 1;                 /* the critical section code */
   sem_post(&mutex);          /* unlock the mutex semaphore */
}

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

   // semaphore mutex should be initialized by 1
   if (sem_init(&mutex, 0, 1) == -1) {
      perror("Could not initialize the mutex 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);

   // destroy semaphore mutex
   sem_destroy(&mutex);
   exit(0);
}
```

6) Run the above program.
7) Observe the program output and take a snapshot of it:

   `Final value of x is 10`

8) Write your report to include all the snapshots you took in the previous steps labeled with the step number, along with your observation comments on each; submit as one .pdf file.