Refresher Lab: Semaphores

In this lab you will learn the following

1. Creating a semaphore using POSIX libraries
2. Using the semaphore to run mutually exclusive critical sections
3. Implement producer-consumer problem with semaphores

Semaphores

The POSIX system in Linux presents its own built-in semaphore library. To use it, we have to :

1. Include #include <semaphore.h>
2. Compile the code by linking with -lpthread –lrt

Data type for the semaphore variable

Similar to shared variable, semaphores are defined by a specific data type **sem_t**

Initializing Semaphores

To initialize a semaphore we use the following function:

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

Where,

1. sem : Specifies the semaphore to be initialized.
2. pshared : This argument specifies whether or not the newly initialized semaphore is shared between processes or between threads. A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.
3. value : Specifies the value to assign to the newly initialized semaphore.

Wait function

To lock a semaphore or wait we can use the **sem_wait** function:

int sem_wait(sem_t *sem);

Signal function

To release or signal a semaphore, we use the **sem_post** function:

int sem_post(sem_t *sem);

Destroy a semaphore

To destroy a semaphore, we can use sem_destroy.

sem_destoy(sem_t *mutex);

An example of semaphore use

In the following program we will create two threads. We will use the semaphore to make sure once one thread is working the other thread waits.

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

sem_t mutex;

void* thread(void* arg)
{
        char *str = (char *) arg;
        printf("IN %s\n", str);
   //wait
        sem_wait(&mutex);
        printf("\n%s entered critical section\n",str);

   //critical section
        printf("\n%s working on the critical section\n",str);
        sleep(4);

   //signal
        printf("\n%s exiting critical section\n",str);
        sem_post(&mutex);
}


int main()
{
        // initialize mutex to value 1
        sem_init(&mutex, 0, 1);
        pthread_t t1,t2;
        pthread_create(&t1,NULL,thread,(void*)"Thread 1");
        pthread_create(&t2,NULL,thread,(void *)"Thread 2");
        pthread_join(t1,NULL);
        pthread_join(t2,NULL);
        sem_destroy(&mutex);
        return 0;
}
```

Q1: Incorporate semaphores into the following program (from Lab 4) to make sure that the final sum is always correct.

The following program creates 10 threads and each of the thread increments the value of a particular global variable. The main process will print the value of the global variable after all the threads executions are completed.

```
#define THREADS 10

void *incr();

int a = 0;
int main()
{

      pthread_t thread[THREADS];
      int  iret,i;
```

```
    /* Create independent threads each of which will execute function  incr*/

        for(i=0;i<THREADS;i++){
              iret = pthread_create( &thread[i], NULL, incr, (void*) NULL);

        }


        for(i=0;i<THREADS;i++){
              pthread_join( thread[i], NULL);
        }

        printf("All thread returns and a = : %d\n",a);
        exit(0);
}

void * incr()
{
        a = a + 1;

}
```

Change the number of threads to 20,000. Compile and run the program a few times. DO you get the same answer all the time?

Q2. Implement producer-consumer program with the help of threads. Create 3 threads for 3 producers and 3 threads for 3 consumers running infinitely. The producers keep producing numbers and store in an array of 5 elements and consumers keep consuming the numbers from the array. Use semaphores to make sure that there are no synchronization issues.