Exp 9: Semaphores

1. Objective:

To learn how to synchronize processes using Semaphores.

2. Introduction:

Semaphore is a technique for coordinating or synchronizing activities in which multiple processes compete for the same operating system resources. A semaphore is a value in a designated place in operating system (or kernel) storage that each process can check and then change. Depending on the value that is found, the process can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphores can be binary (0 or 1) or can have additional values. The advantage of semaphores over other synchronization mechanisms is that they can be used to synchronize two related or unrelated processes trying to access the same resource.

2.1 Binary semaphores

Pthread.h offers some primitives that control the declaration, management, synchronization of threads through binary semaphores. The built-in declaration statements and functions below enable semaphore management under *pthread* library.

Statement	Explanation
`Pthread_mutex_t`	
`Pthread_mutex_init(&semaphone_name, NULL)`	
`Pthread_mutex_lock(&semaphore_name)`	
`Pthread_mutex_unlock(&semaphore_name)`	

2.2 Counting semaphores

semaphore.h offers some primitives that control the declaration, management, synchronization of threads through counting semaphores. A counting semaphore may be initialized to a value that is larger than one. However, if the semaphore is initialized to one it will be a binary semaphore. The built-in functions below enable semaphore management under *semaphore library*.

Function	Explanation
sem_t	
Sem_init(&semaphore_name,0,semaphore_value)	
Sem_wait(&semaphore_name)	
Sem_post(&semaphore_name)	
sem_destroy(&semaphore_name)	

2.4 Example:

```
/*************************************
Semaphore creation
compile use: gcc simple.c -o simple -lpthread
File Name: simple.c
*************************************/

#include <unistd.h>     /* Symbolic Constants */
#include <sys/types.h>  /* Primitive System Data Types */
#include <errno.h>      /* Errors */
#include <stdio.h>      /* Input/Output */
#include <stdlib.h>     /* General Utilities */
#include <pthread.h>    /* POSIX Threads */
#include <string.h>     /* String handling */
#include <semaphore.h>  /* Semaphore */


/* prototype for thread routine */
void handler ( void *ptr );

/* global vars */
/* semaphores are declared global so they can be accessed
   in main() and in thread routine,
   here, the semaphore is used as a mutex */

sem_t mutex;
int counter; /* shared variable */

int main()
{
    int i[2];
    pthread_t thread_a;
    pthread_t thread_b;

    i[0] = 0; /* argument to threads */
    i[1] = 1;

 sem_init(&mutex, 0, 1);

/* initialize mutex to 1 - binary semaphore */
/* second param = 0 - semaphore is local */


  pthread_create (&thread_a, NULL, (void *) &handler, (void *) &i[0]);
  pthread_create (&thread_b, NULL, (void *) &handler, (void *) &i[1]);

    pthread_join(thread_a, NULL);
```

```
    pthread_join(thread_b, NULL);

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

        exit(0);
}

void handler ( void *ptr )
{
    int x;
    x = *((int *) ptr);
    printf("Thread %d: Waiting to enter critical region...\n", x);
    sem_wait(&mutex);       /* down semaphore */
    /* START CRITICAL REGION */
    printf("Thread %d: Now in critical region...\n", x);
    printf("Thread %d: Counter Value: %d\n", x, counter);
    printf("Thread %d: Incrementing Counter...\n", x);
    counter++;
    printf("Thread %d: New Counter Value: %d\n", x, counter);
    printf("Thread %d: Exiting critical region...\n", x);
    /* END CRITICAL REGION */
    sem_post(&mutex);       /* up semaphore */

    pthread_exit(0); /* exit thread */
}
```