University of Central Punjab

FOIT (Operating Systems)

GL- 10

Synchronization

Mutex & Semaphores

Objectives

Students will be able to use **Mutex** and **Semaphore** libraries to solve critical section problem.

Students will be able to solve some classical synchronization problem using pthread **Mutex & Sempahore** libraries.

Mutex Lock

Mutual exclusion is the method of *serializing access* to shared resources. You do not want a thread to be modifying a variable that is already in the process of being modified by another thread! Another scenario is a dirty read where the value is in the process of being updated and another thread reads an old value.

Synopsis

#include <pthread.h>

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
           const pthread_mutexattr_t *restrict attr);

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Description

The pthread_mutex_init() function initialises the mutex referenced by mutex with attributes specified by attr. If attr is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialisation, the state of the mutex becomes initialised and unlocked.

The mutex object referenced by mutex shall be locked by calling pthread_mutex_lock(). If the mutex is already locked, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

The *pthread_mutex_trylock*() function shall be equivalent to *pthread_mutex_lock*(), except that if the mutex object referenced by *mutex* is currently locked (by any thread, including the current thread), the call shall return immediately.

The *pthread_mutex_unlock*() function shall release the mutex object referenced by *mutex*.

If successful, the pthread_mutex_lock() and pthread_mutex_unlock() functions shall return zero; otherwise, an error number shall be returned to indicate the error.

RETURN VALUE

The *pthread_mutex_trylock*() function shall return zero if a lock on the mutex object referenced by *mutex* is acquired. Otherwise, an error number is returned to indicate the error.

In the following example, we will implement producer-consumer problem with Mutex lock.

Producer-Consumer problem using Mutex Lock

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

pthread_mutex_t mutex;
int n=10; int buffer[10];
int in=0, out=0,count=0;

void* producer (void* ptr)
{
	for(unsigned int i=1;i<15;i++)
	{
		while(count==n);
		buffer[in]=i;
		in=(in+1)%n;
		pthread_mutex_lock(&mutex);
		count++;
		pthread_mutex_unlock(&mutex);
	}
	pthread_exit(0);
}

void* consumer (void* ptr)
{
	pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,NULL);

	for(unsigned int i=1;i<15;i++)
	{
		while(count==0);
		printf("%d\n",buffer[out]); /*print that data*/
		out=(out+1)%n; /*increment counter */
		pthread_mutex_lock(&mutex);
		count--;
```

```
          pthread_mutex_unlock(&mutex);

     }
     pthread_exit(0);
}
int main()
{
pthread_t pid1,pid2;
pthread_mutex_init(&mutex,NULL);     /*initializing     mutex
variable*/
pthread_create(&pid2,NULL,&consumer,NULL);
pthread_create(&pid1,NULL,&producer,NULL);
pthread_join(pid1,NULL);
pthread_join(pid2,NULL);
return 0;
}
```

Semaphore

Semaphore was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a simple integer value, which is known as a semaphore. Semaphore is simply a variable which is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

Semaphores are of two types:

1. **Binary Semaphore –** This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.
2. **Counting Semaphore –** Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

SYNOPSIS

#include <semaphore.h>

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

int sem_wait(sem_t *sem);

int sem_trywait(sem_t *sem);

int sem_post(sem_t *sem);
```

sem_init() initializes the unnamed semaphore at the address pointed to by *sem*. The *value* argument specifies the initial value for the semaphore. The *pshared* argument indicates whether this semaphore is to be shared between the threads of a process, or between processes. If *pshared* has the value 0, then the semaphore is shared between the threads of a process, and should be located at some address that is visible to all threads (e.g., a global variable, or a variable allocated dynamically on the heap).If *pshared* is nonzero, then the semaphore is shared between processes, and should be located in a region of shared memory.

sem_wait() decrements (locks) the semaphore pointed to by *sem*. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

sem_trywait() is the same as **sem_wait**(), except that if the decrement cannot be immediately performed, then call returns an error (*errno* set to **EAGAIN**) instead of blocking.

sem_post() increments (unlocks) the semaphore pointed to by *sem*. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem_wait(3) call will be woken up and proceed to lock the semaphore.

RETURN VALUE

All of these functions return 0 on success; on error, the value of the semaphore is left unchanged, -1 is returned, and *errno* is set to indicate the error.

In the following example, we will implement bounded-buffer problem using counting semaphore.

Bounded-Buffer problem using Semaphore

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

sem_t full,empty;
int n=10; int buffer[10];
int in=0, out=0;

void* producer (void* ptr)
{
      for(unsigned int i=1;i<8;i++)
      {
            sem_wait(&empty);
            buffer[in]=i;
            in=(in+1)%n;
            sem_post(&full);
      }
      pthread_exit(0);
}

void* consumer (void* ptr)
{
      pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,NULL);

      for(unsigned int i=1;i<8;i++)
      {
            sem_wait(&full); /* check if there is new data*/
            printf("%d\n",buffer[out]); /*print that data*/

            out=(out+1)%n; /*increment counter */
            sem_post(&empty); /* create vacancy */

      }
      pthread_exit(0);
}
int main()
{
pthread_t pid1,pid2;
```

```
sem_init(&full, 0, 0); /*initializing semaphore variable with 
0*/
sem_init(&empty, 0, 10); /*initializing semaphore variable 
with 10*/
pthread_create(&pid2,NULL,&consumer,NULL);
pthread_create(&pid1,NULL,&producer,NULL);
pthread_join(pid1,NULL);
pthread_join(pid2,NULL);
return 0;
}
```

University of Central Punjab
FOIT (Operating Systems)
GL- 10
Synchronization
Mutex & Semaphores

Graded Tasks

Task 1

Solution to dinning-philosopher problem is given below. Although this solution guarantees that no two neighbours are eating simultaneously, it nevertheless must be rejected because it could create deadlock. Suppose that all five philosophers become hungry at the same time and each grabs her right chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her left chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are replaced by:

1. Allow at most four philosophers to be sitting simultaneously at the table.
2. Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
3. Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

Dinning-philosopher basic solution
wait(chopstick[i]); **wait(chopstick[(i+1) % 5]);** ... **/* eat for a while */** ... **signal(chopstick[i]);** **signal(chopstick[(i+1) % 5]);** ... **/* think for a while */** ...

Problem: Incorporate all of the three solutions one by one to the basic solution for deadlock free execution of the code. You are required to submit each solution in a separate file.

Task 2

If all the philosophers come together, two of them can eat simultaneously. However, the incorporation of deadlock free solution may not guarantee this. Write a solution that ensures maximum possible number of philosophers could eat at a time.