**Assignment 6**

###### AIM:

###### Thread synchronization using counting semaphores and mutual exclusion using mutex. Application to demonstrate: producer-consumer problem with counting semaphores and mutex.

Top of Form

**Theory:**

Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as a critical section. Processes’ access to critical section is controlled by using synchronization techniques. When one thread starts executing the critical section (a serialized segment of the program) the other thread should wait until the first thread finishes. If proper synchronization techniques are not applied, it may cause a race condition where the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.

Mutex

* A Mutex is a lock that we set before using a shared resource and release after using it.
* When the lock is set, no other thread can access the locked region of code.
* So we see that even if thread 2 is scheduled while thread 1 was not done accessing the shared resource and the code is locked by thread 1 using mutexes then thread 2 cannot even access that region of code.
* So this ensures synchronized access of shared resources in the code.

**A mutex is initialized and then a lock is achieved by calling the following two functions :** The first function initializes a mutex and through second function any critical region in the code can be locked.

* + 1. **int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex, const pthread\_mutexattr\_t \*restrict attr) :**Creates a mutex, referenced by mutex, with attributes specified by attr. If attr is NULL, the default mutex attribute (NONRECURSIVE) is used.

**Returned value**  
If successful, pthread\_mutex\_init() returns 0, and the state of the mutex becomes initialized and unlocked.  
If unsuccessful, pthread\_mutex\_init() returns -1.

* + 1. **int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex) :**Locks a mutex object, which identifies a mutex. If the mutex is already locked by another thread, the thread waits for the mutex to become available. The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it. When the mutex has the attribute of recursive, the use of the lock may be different. When this kind of mutex is locked multiple times by the same thread, then a count is incremented and no waiting thread is posted. The owning thread must call pthread\_mutex\_unlock() the same number of times to decrement the count to zero.

**Returned value**  
If successful, pthread\_mutex\_lock() returns 0.  
If unsuccessful, pthread\_mutex\_lock() returns -1.

**The mutex can be unlocked and destroyed by calling following two functions :**The first function releases the lock and the second function destroys the lock so that it cannot be used anywhere in future.

* + 1. **int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex) :**Releases a mutex object. If one or more threads are waiting to lock the mutex, pthread\_mutex\_unlock() causes one of those threads to return from pthread\_mutex\_lock() with the mutex object acquired. If no threads are waiting for the mutex, the mutex unlocks with no current owner. When the mutex has the attribute of recursive the use of the lock may be different. When this kind of mutex is locked multiple times by the same thread, then unlock will decrement the count and no waiting thread is posted to continue running with the lock. If the count is decremented to zero, then the mutex is released and if any thread is waiting for it is posted.

**Returned value**  
If successful, pthread\_mutex\_unlock() returns 0.  
If unsuccessful, pthread\_mutex\_unlock() returns -1

* + 1. **int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex) :**Deletes a mutex object, which identifies a mutex. Mutexes are used to protect shared resources. mutex is set to an invalid value, but can be reinitialized using pthread\_mutex\_init().

**Returned value**  
If successful, pthread\_mutex\_destroy() returns 0.  
If unsuccessful, pthread\_mutex\_destroy() returns -1.

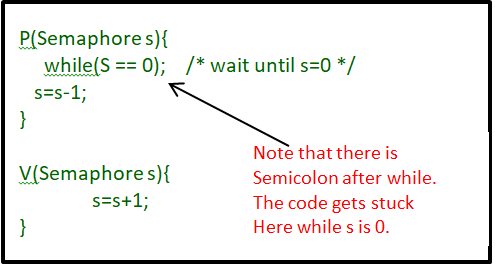
Semaphore was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a simple integer value, which is known as a semaphore. Semaphore is simply a variable which is non-negative and shared between threads. This variable is used to solve the critical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

Now let us see how it do so.

First, look at two operations which can be used to access and change the value of the semaphore variable.



**Some point regarding P and V operation**

1. P operation is also called wait, sleep or down operation and V operation is also called signal, wake-up or up operation.
2. Both operations are atomic and semaphore(s) is always initialized to one.Here atomic means that variable on which read, modify and update happens at the same time/moment with no pre-emption i.e. in between read, modify and update no other operation is performed that may change the variable.
3. A critical section is surrounded by both operations to implement process synchronization.See below image.critical section of Process P is in between P and V operation.

**Code:**

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

#include<unistd.h>

#define SIZE 10 //Buffer size

#define PC\_TIME 6 //producer and consumer Sleep time

#define MAIN\_TIME 10 //main function sleep time

int buffer[SIZE],count; //global declarations

pthread\_t tid[2]; //thread id

pthread\_mutex\_t lock; //mutex lock

sem\_t empty,full; //semaphores

void \*producer(void \*arg){

int item;

while(1){

sleep(PC\_TIME); //sleep

item=rand(); //generate random number

sem\_wait(&empty); //decrement semaphore empty

pthread\_mutex\_lock(&lock); //acquire lock

if(count<SIZE){

buffer[count]=item; //put item in buffer

count++;

printf("Producer produced item %d\n",item);

}

else{

printf("Buffer Full\n");

}

pthread\_mutex\_unlock(&lock); //release lock

sem\_post(&full); //increment semaphore full

}

pthread\_exit(NULL); //exit thread

}

void \*consumer(void \*arg){

int item;

while(1){

sleep(PC\_TIME); //sleep

sem\_wait(&full); //decrement semaphore full

pthread\_mutex\_lock(&lock); //acquire lock

if(count>0){

item=buffer[count-1]; //remove item from buffer

count--;

printf("Consumer consumed item %d\n",item);

}

else{

printf("Buffer Empty\n");

}

pthread\_mutex\_unlock(&lock); //release lock

sem\_post(&empty); //increment empty

}

pthread\_exit(NULL); //exit thread

}

int main(){

int p,c;

printf("Enter Number of producers: ");

scanf("%d",&p);

printf("Enter Number of consumers: ");

scanf("%d",&c);

printf("\n");

count=0; //initializing the global variabes

pthread\_mutex\_init(&lock,NULL);

sem\_init(&empty,0,SIZE);

sem\_init(&full,0,0);

for(int i=0;i<p;i++) //creating threads

pthread\_create(&tid[0],NULL,producer,NULL);

for(int i=0;i<c;i++)

pthread\_create(&tid[1],NULL,consumer,NULL);

sleep(MAIN\_TIME); //sleep

return 0;

}

**Output:**

