Lab No. 10
Synchronization: Two-Process Solutions, MUTEX, and Semaphore

Objective

This lab is designed to implement the solutions to the critical-solution problem.

Activity Outcomes:

On completion of this lab students will be able to

- Implement two process solutions to critical-section problem
- Solve the CS problem using MUTEX and Semaphore

Instructor Notes

As pre-lab activity, read the content from the following (or some other) internet source: https://www.geeksforgeeks.org

1) Useful Concepts

The Critical-Section Problem

Cooperating processes or threads share some data with each other. If two or more threads or processes access and manipulate the shared data concurrently then this may result in data inconsistency. To avoid such data inconsistencies, we need to make it sure that threads/processes must be synchronized and if one thread/process is manipulating the shared data then no other thread/process should be allowed to access that data.

Each cooperating thread/process has some segments of critical code that is the segment of code where shared data is accessed and manipulated. These segment of codes are called critical-sections. We need to make it sure that if one thread/process is executing its critical section then no other process should be allowed to execute its critical section. Designing solutions to ensure this; is called the CS problem. We can define the CS problem as:

The critical section problem is used to design a protocol followed by a group of processes, so that when one process has entered its critical section, no other process is allowed to execute in its critical section.

The general structure of a solution to critical-section problem is:

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Two Process Solutions to CS problem

A simple solution:

First, we discuss a simple solution. This solution uses a variable turn. The value of turn decides, whose turn it is to enter in CS. The solution is given below:

```
Code for process i
do {
     while (turn == j); // Entry Code
      critical section
       turn = j;  //Exit code
     remainder section
   } while (true);
```

```
Code for process j
do {
     while (turn == i); // Entry Code
      critical section
       turn = i;  //Exit code
     remainder section
   } while (true);
```

This simple solution ensure mutual exclusion and bounded waiting conditions but it fail to ensure progress condition.

Peterson's Solution:

Peterson's solution satisfies all the conditions for a good solution. It is also a two process solution. The two processes share two variables: int turn and Boolean flag[2]. The variable turn indicates whose turn it is to enter the critical section while the flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready. The pseudo code of the Peterson's solution is given below:

```
Code for process i
do
{
       flag[ i ] = true;
       turn = j;
       while (flag[ j ] && turn == j);
      critical section
       flag[ i ] = false;
      remainder section
} while (true);
```

```
Code for process j
do
{
       flag[ j] = true;
       turn = i;
       while (flag[ i ] && turn == i);
      critical section
       flag[ j ] = false;
      remainder section
} while (true); while (true);
```

MUTEX Lock

MUTEX lock is software based solution to CS problem and is applicable on n threads/processes. MUTEX is a shortened form of the words "mutual exclusion". MUTEX variables are one of the primary means of implementing thread synchronization. A MUTEX variable acts like a "lock" protecting access to a shared data resource. The basic concept of a MUTEX; as used in pthreads is that only one thread can lock (or own) a MUTEX variable at any given time. Thus, even if several threads try to lock a MUTEX only one thread will be successful. No other thread can own that MUTEX until the owning thread unlocks that MUTEX.
A typical sequence in the use of a MUTEX is as follows:

- Create and initialize a MUTEX variable

- Several threads attempt to lock the MUTEX
- Only one succeeds and that thread owns the MUTEX
- The owner thread performs some set of actions
- The owner unlocks the MUTEX
- Another thread acquires the MUTEX and repeats the process
- Finally the MUTEX is destroyed.

The routines to perform these tasks are given below:

pthread_mutex_init(pthread_mutex_t var, pthread_mutexattr_t attr) // to initialize MUTEX variable
pthread_mutex_lock (pthread_mutex_t var) // to lock CS
pthread_mutex_unlock (pthread_mutex_t var) // to unlock CS
pthread_mutex_destroy(pthread_mutex_t var) // to destroy MUTEX variable

Semaphore

Semaphore is another synchronization tool that can be used to solve several synchronization problems. Semaphore is an integer variable but it can be accessed only through two functions which are wait( ) and signal( ).

```
Pseudo code for wait function

wait(S) {

    while (S <= 0)

       ; // busy wait

    S - -;

}


Pseudo code for signal function

signal (S) {

    S++;

}
```

```
Pseudo code for solution to CS
problem using semaphore

do {

    wait(s)  //entry code

       critical section

    signal(S)  // exit code

      remainder section

 } while (true);
```

The following routines are used to implement POSIX semaphore

#include <semaphore.h> // header-file
sem_t // semaphore data type
int sem_init(sem_t *sem, int pshared, unsigned value); // to initialize of semaphore
sem_wait() // wait function
sem_post() // signal function
int sem_destroy(sem_t *sem); // to destroy semahpore

2) Solved Lab Activities

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	20	Medium	CLO-7
2	10	Medium	CLO-7
3	15	Medium	CLO-7
4	15	Medium	CLO-7

Activity 1:

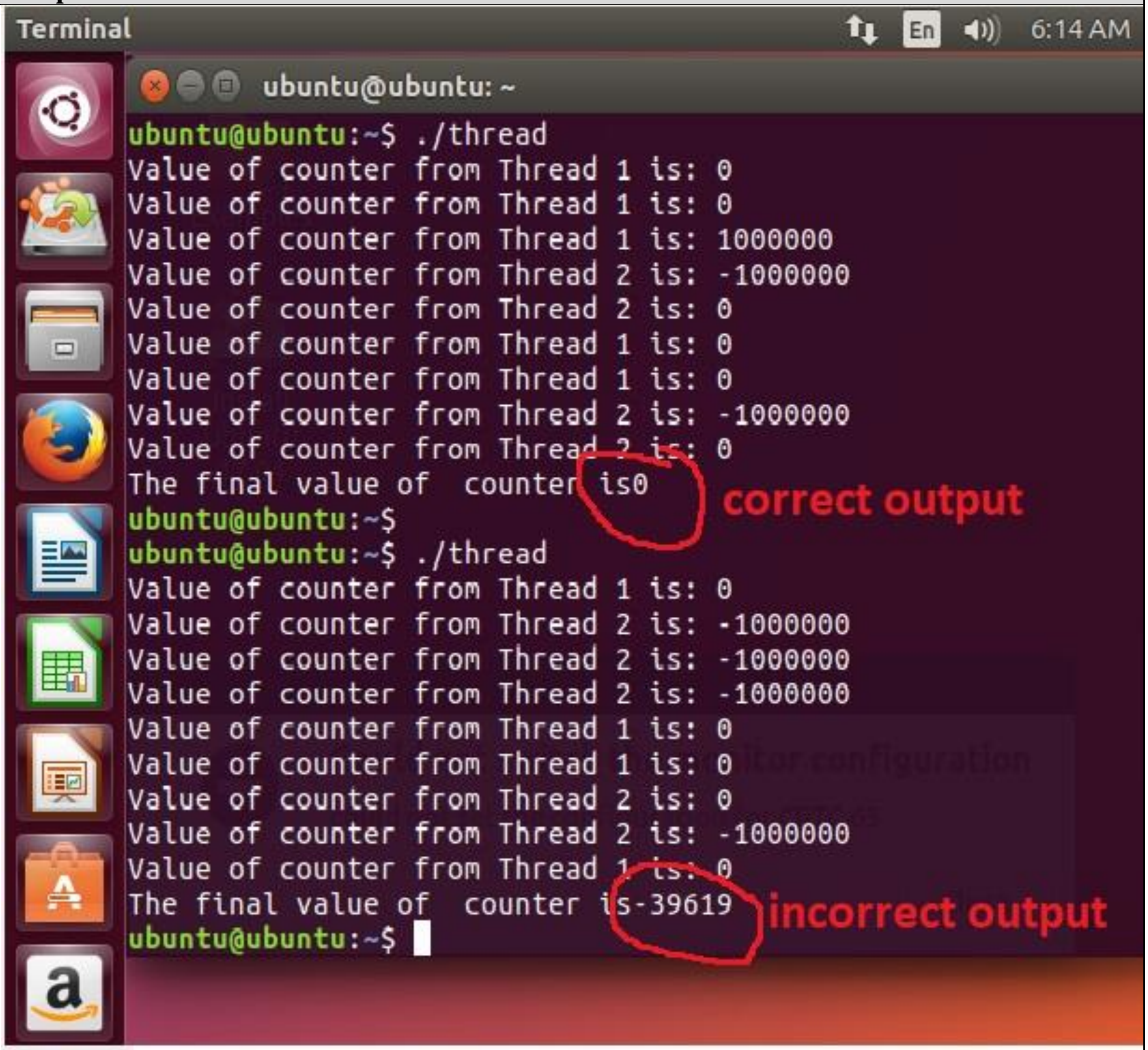
In this activity, we identify how the concurrent access to shared data may result in data inconsistency. We create two threads that manipulate a shared variable counter concurrently. As a result, sometimes we get correct output while sometimes the output is not c

Solution:

Code

```
#include<iostream>
#include<pthread.h>
using namespace std;
int counter=0;// shared data
void *Thread1(void *args) // routine executed by thread 1
{
for(int i=0; i<=5000000; i++)
{
counter++;
if(counter%1000000==0)
cout<<"Value of counter from Thread 1 is: "<<counter<<endl;
}
}
void *Thread2(void *args) // routine executed by thread 2
{
for(int i=0; i<=5000000; i++)
{
counter--;
if(counter%1000000==0)
cout<<"Value of counter from Thread 2 is: "<<counter<<endl;
}
}
int main()
{
pthread_t t1,t2;
pthread_create(&t1,NULL,Thread1,NULL);
pthread_create(&t2,NULL,Thread2,NULL);
pthread_join(t1,NULL);
pthread_join(t2,NULL);
cout<<"The final value of  counter is"<<counter<<endl;
return 0;
}
```

Out-put


Terminal
En
6:14 AM
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ ./thread
Value of counter from Thread 1 is: 0
Value of counter from Thread 1 is: 0
Value of counter from Thread 1 is: 1000000
Value of counter from Thread 2 is: -1000000
Value of counter from Thread 2 is: 0
Value of counter from Thread 1 is: 0
Value of counter from Thread 1 is: 0
Value of counter from Thread 2 is: -1000000
Value of counter from Thread 2 is: 0
The final value of counter is0
correct output
ubuntu@ubuntu:~$
ubuntu@ubuntu:~$ ./thread
Value of counter from Thread 1 is: 0
Value of counter from Thread 2 is: -1000000
Value of counter from Thread 2 is: -1000000
Value of counter from Thread 2 is: -1000000
Value of counter from Thread 1 is: 0
Value of counter from Thread 1 is: 0
Value of counter from Thread 2 is: 0
Value of counter from Thread 2 is: -1000000
Value of counter from Thread 1 is: 0
The final value of counter is-39619
incorrect output
ubuntu@ubuntu:~$

Activity 2:

Now, we implement the simple solution in the code written in Activity 1 to protect the CS

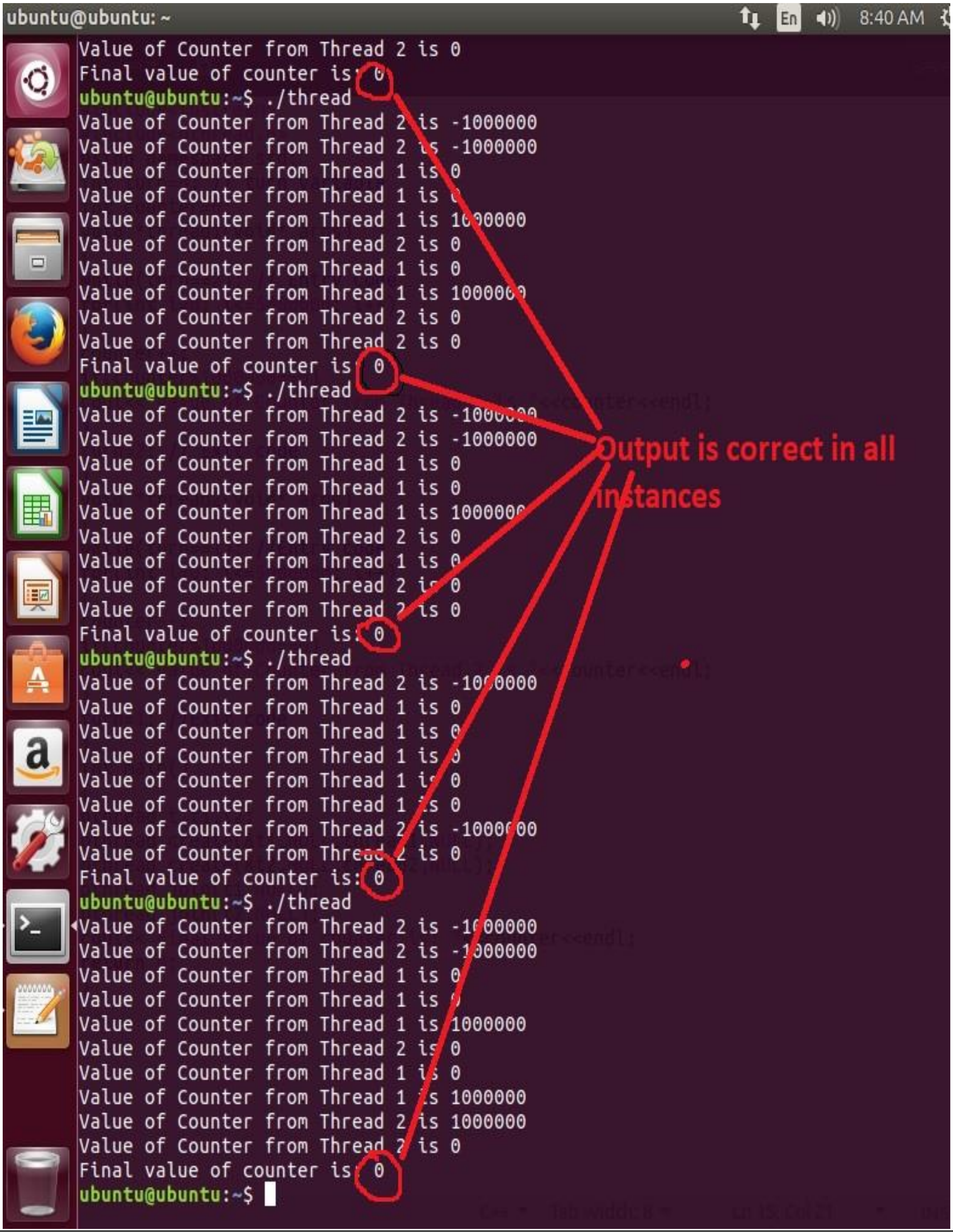
Solution:

Code

thread.cpp (~/) - gedit

```
#include<iostream>
#include<pthread.h>
using namespace std;
int turn=0; // turn variable
int counter=0;
void *thread1(void* args)
{
while(turn==2); // Entry Code
for(int i=0; i<=5000000; i++)
{
counter++;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 1 is "<<counter<<endl;
}
turn=2; // Exit code
}
void *thread2(void* args)
{
while(turn==1); //Entry code
for(int i=0; i<=5000000; i++)
{
counter--;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 2 is "<<counter<<endl;
}
turn=1; //Exit code
}
int main()
{
pthread_t t1,t2;
pthread_create(&t1,NULL,thread1,NULL);
pthread_create(&t2,NULL,thread2,NULL);
pthread_join(t1,NULL);
pthread_join(t2,NULL);
cout<<"Final value of counter is: "<<counter<<endl;
return 0;
}
```

Out-put



ubuntu@ubuntu: ~
En
8:40 AM
Value of Counter from Thread 2 is 0
Final value of counter is: 0
ubuntu@ubuntu:~$ ./thread
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 1000000
Value of Counter from Thread 2 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 1000000
Value of Counter from Thread 2 is 0
Value of Counter from Thread 2 is 0
Final value of counter is: 0
ubuntu@ubuntu:~$ ./thread
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 1000000
Value of Counter from Thread 2 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 2 is 0
Value of Counter from Thread 2 is 0
Final value of counter is: 0
ubuntu@ubuntu:~$ ./thread
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 2 is 0
Final value of counter is: 0
ubuntu@ubuntu:~$ ./thread
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 1000000
Value of Counter from Thread 2 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 1000000
Value of Counter from Thread 2 is 1000000
Value of Counter from Thread 2 is 0
Final value of counter is: 0
ubuntu@ubuntu:~$
Output is correct in all instances

Activity 3:

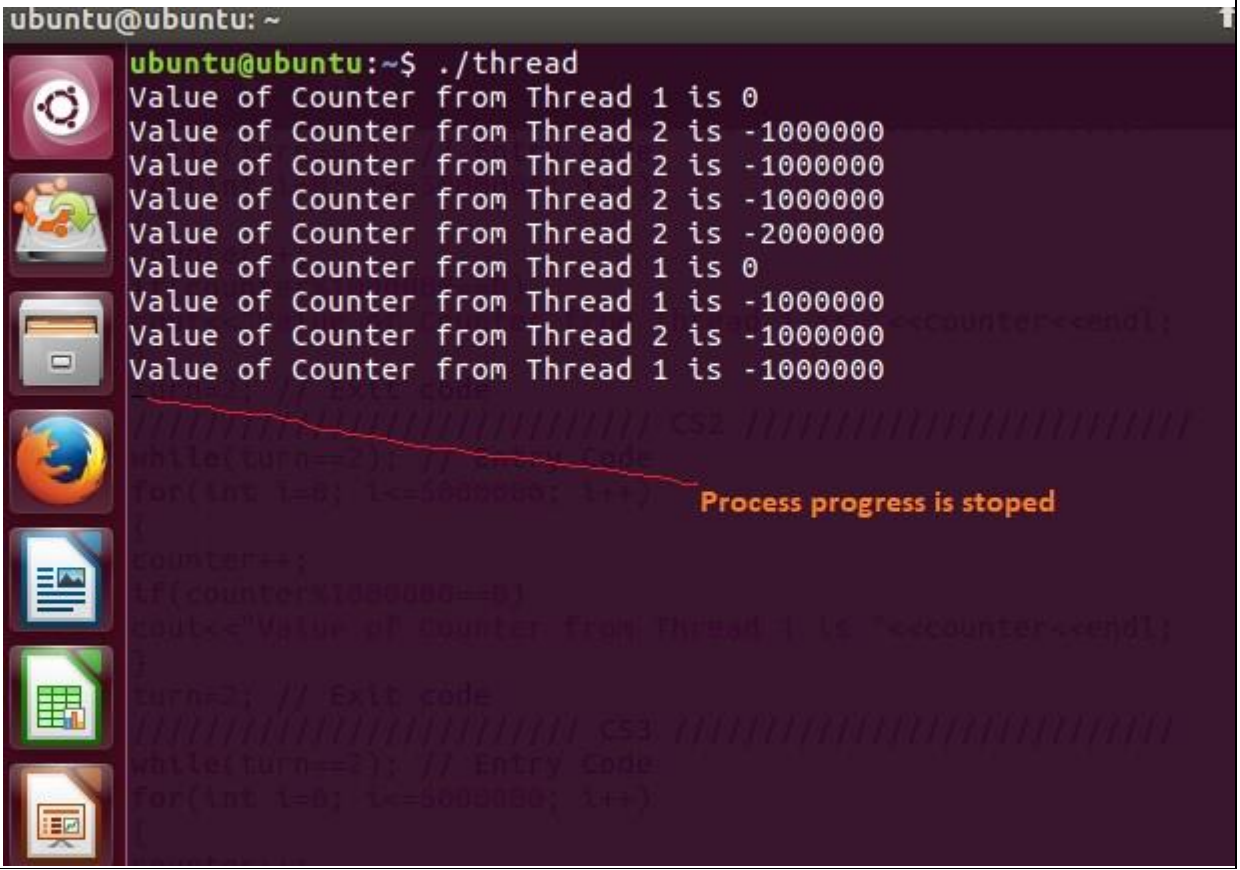
In this activity, we show that how the progress in not satisfied in simple solution

Solution:

Code

thread.cpp (~/) - gedit

```
#include<iostream>
#include<pthread.h>
using namespace std;
int turn=0; // turn variable
int counter=0;
void *thread1(void* args)
{
///////////////////////////  CS1 ////////////////////////
while(turn==2); // Entry Code
for(int i=0; i<=5000000; i++)
{
counter++;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 1 is "<<counter<<endl;
}
turn=2; // Exit code
///////////////////////////// CS2 /////////////////////////
while(turn==2); // Entry Code
for(int i=0; i<=5000000; i++)
{
counter++;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 1 is "<<counter<<endl;
}
turn=2; // Exit code
///////////////////////// CS3 ///////////////////////////
while(turn==2); // Entry Code
for(int i=0; i<=5000000; i++)
{
counter++;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 1 is "<<counter<<endl;
}
turn=2; // Exit code
}
void *thread2(void* args)
{
while(turn==1); //Entry code
for(int i=0; i<=5000000; i++)
{
counter--;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 2 is "<<counter<<endl;
}
turn=1; //Exit code
}
int main()
{
pthread_t t1,t2;
pthread_create(&t1,NULL,thread1,NULL);
pthread_create(&t2,NULL,thread2,NULL);
pthread_join(t1,NULL);
pthread_join(t2,NULL);
cout<<"Final value of counter is: "<<counter<<endl;
return 0;
}
```

Out-put


ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ ./thread
Value of Counter from Thread 1 is 0
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 2 is -2000000
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is -1000000
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 1 is -1000000
Process progress is stoped

Activity 4:

In this activity, we implement the Peterson's solution and show that progress condition is satisfied.

Solution:

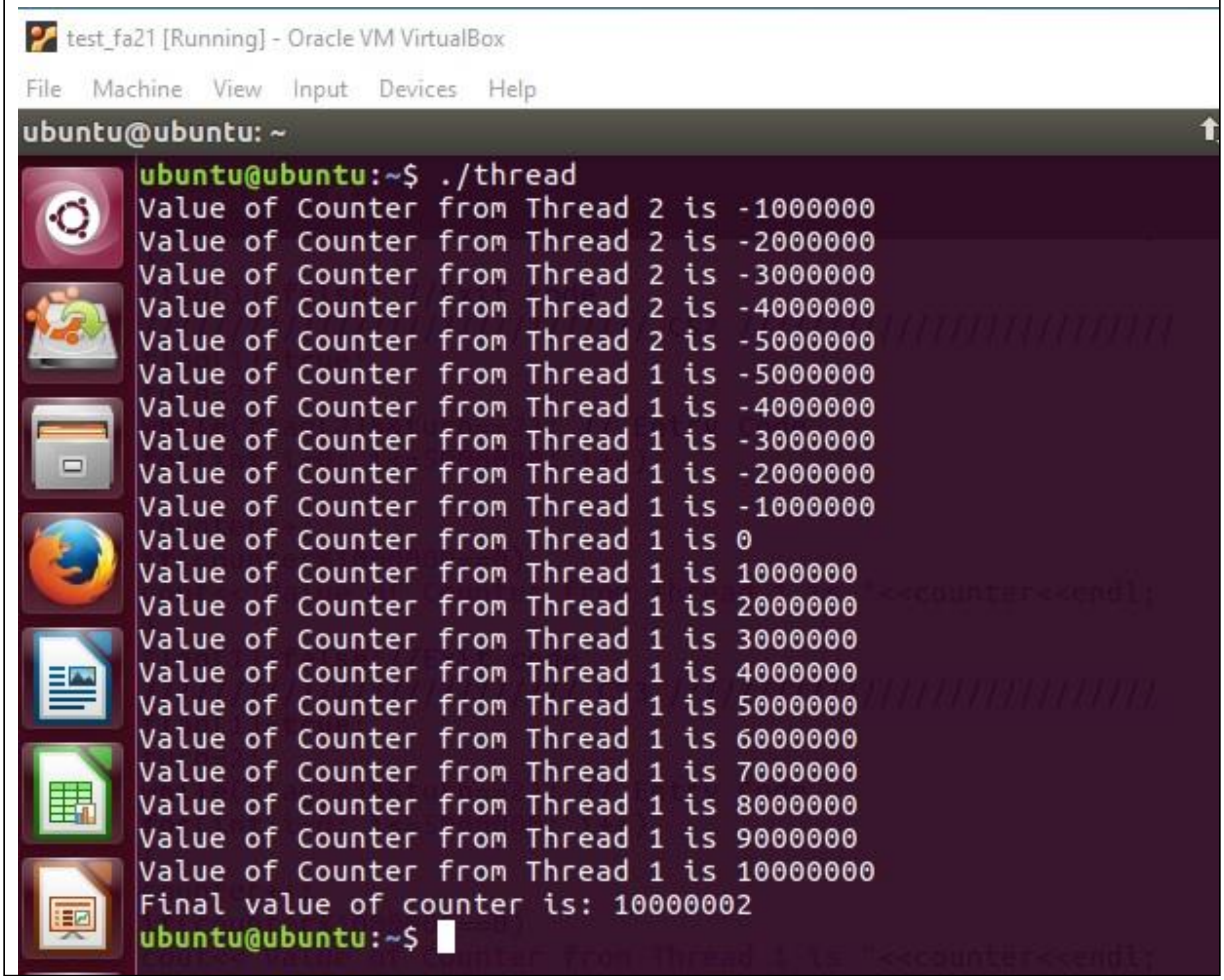
Code

thread.cpp (~/) - gedit

Open

```
#include<iostream>
#include<pthread.h>
using namespace std;
bool flag[3]={false,false,false}; // flag array
int turn=0; // turn variable
int counter=0;
void *thread1(void* args)
{
/////////////////////////////  CS1 /////////////////////////
flag[1]=true;
turn=2;
while(flag[2] && turn==2); // Entry Code
for(int i=0; i<=5000000; i++)
{
counter++;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 1 is "<<counter<<endl;
}
flag[1]=false; //Exit code
/////////////////////////////// CS2 //////////////////////////
flag[1]=true;
turn=2;
while(flag[2]&&turn==2); // Entry Code
for(int i=0; i<=5000000; i++)
{
counter++;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 1 is "<<counter<<endl;
}
flag[1]=false;//Exit code
/////////////////////////// CS3 /////////////////////////////
flag[1]=true;
turn=2;
while(flag[2]&&turn==2); // Entry Code
for(int i=0; i<=5000000; i++)
{
counter++;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 1 is "<<counter<<endl;
}
flag[1]=false;//Exit code
}
void *thread2(void* args)
{
flag[2]=true;
turn=1;
while(flag[1]&&turn==1); // Entry Code
for(int i=0; i<=5000000; i++)
{
counter--;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 2 is "<<counter<<endl;
}
flag[2]=false; //Exit code
}
int main()
{
pthread_t t1,t2;
pthread_create(&t1,NULL,thread1,NULL);
pthread_create(&t2,NULL,thread2,NULL);
pthread_join(t1,NULL);
pthread_join(t2,NULL);
cout<<"Final value of counter is: "<<counter<<endl;
return 0;
}
```

Out-put



3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Task 1:

Write the code for the problem given in Activity 3 and protect the CS's using the MUTEX lock

Task 2:

Write the code for the problem given in Activity 3 and protect the CS's using the Semaphore

Extra notes for students.
Mutex help:

Creating / Destroying Mutexes :

pthread_mutex_init (pthread_mutex_t mutex, pthread_mutexattr_t attr)

pthread_mutex_destroy (pthread_mutex_t mutex)

pthread_mutexattr_init (pthread_mutexattr_t attr)

pthread_mutexattr_destroy (pthread_mutexattr_t attr)

pthread_mutex_init( ) creates and initializes a new mutex mutex object, and sets its attributes according to the mutex attributes object, attr. The mutex is initially unlocked.

Mutex variables must be of type pthread_mutex_t. The attr object is used to establish properties for the mutex object, and must be of type pthread_mutexattr_t if used (may be specified as NULL to accept defaults). If implemented, the pthread_mutexattr_init( ) and pthread_mutexattr_destroy( ) routines are used to create and destroy mutex attribute objects respectively.
pthread_mutex_destroy( ) should be used to free a mutex object which is no longer needed.

Locking / Unlocking Mutexes :

pthread_mutex_lock (pthread_mutex_t mutex)

pthread_mutex_trylock (pthread_mutex_t mutex)

pthread_mutex_unlock (pthread_mutex_t mutex)

pthread_mutex_lock( ) routine is used by a thread to acquire a lock on the specified mutex variable. If the mutex is already locked by another thread, the call will block the calling thread until the mutex is unlocked.

pthread_mutex_trylock( ) will attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately. This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.
Mutex contention: when more than one thread is waiting for a locked mutex, which thread will be granted the lock first after it is released? Unless thread priority scheduling (not covered) is used, the assignment will be left to the native system scheduler and may appear to be more or less random. pthread_mutex_unlock( ) will unlock a mutex if called by the owning thread. Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data.
An error will be returned:

If the mutex was already unlocked

If the mutex is owned by another thread.

Sample code:

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

```
/* Function run when the thread is created */
void* compute_thread (void*);
/* This is the lock for thread synchronization */
    pthread_mutex_t my_sync;
main( )
{
/* thread creation */
    pthread_t tid;
    pthread_attr_t attr;
    char hello[ ] = {"Hello, "};
    char thread[ ] = {"thread"};
/* Initialize the thread attributes */
    pthread_attr_init (&attr);
/* Initialize the mutex (default attributes) */
    pthread_mutex_init (&my_sync,NULL);
/* Create another thread. ID is returned in &tid */
/* The last parameter is passed to the thread function */
    pthread_create(&tid, &attr, compute_thread, hello);
    sleep(1); /* Let the thread get started */
/* Lock the mutex when it's our turn to do work */
    pthread_mutex_lock(&my_sync);
     printf(thread);
     printf("\n");
    pthread_mutex_unlock(&my_sync);
    exit(0);
}
/* The thread function to be run */
void* compute_thread(void* dummy)
{
/* Lock the mutex when its our turn */
    pthread_mutex_lock(&my_sync);
    printf(dummy);
    pthread_mutex_unlock(&my_sync);
    sleep(1); return;
}
```

Semaphore help:

This lab will consider only POSIX semaphore, since POSIX semaphores has very clear API functions to perform semaphore operations. However, it is more efficient to use System V semaphore than POSIX semaphore when semaphores are shared between processes.

What is a Semaphore?

A semaphore is an integer variable with two atomic operations:

1) **wait**. Other names for **wait** are **P**, **down** and **lock.**
2) **signal**: Other names for **signal** are **V**, **up**, **unlock** and **post**.

POSIX Semaphore

Semaphores are part of the POSIX.1b standard adopted in 1993. The POSIX.1b standard defines two types of semaphores: *named* and *unnamed*. A POSIX.1b *unnamed semaphore* can

be used by a single process or by children of the process that created them. A POSIX.1b *named semaphore* can be used by any processes. In this section, we will consider only how to initialize unnamed semaphore.
The following header summarizes how we can use POSIX.1b unnamed semaphore:

Header file name	**`#include <semaphore.h>`**
Semaphore data type	**`sem_t`**
Initialization	**`int sem_init(sem_t *sem, int pshared, unsigned value);`**
Semaphore Operations	**`int sem_destroy(sem_t *sem);`** **`int sem_wait(sem_t *sem);`** **`int sem_post(sem_t *sem);`** **`int sem_trywait(sem_t *sem);`**
Compilation	**`cc filename.c -o filename -lrt`**

All of the POSIX.1b semaphore functions return **`-1`** to indicate an error.
`sem_init` function initializes the semaphore to have the value `value`. 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.
`sem_wait` is a standard semaphore wait operation. If the semaphore value is 0, the `sem_wait` blocks unit it can successfully decrement the semaphore value.
`sem_trywait` is similar to `sem_wait` except that instead of blocking when attempting to decrement a zero-valued semaphore, it returns `-1`.
`sem_post` is a standard semaphore signal operation. The POSIX.1b 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.

Sample code:

```
// in this code, two threads are displaying output by calling
sem_wait() and sem_post() to ensure M.E //

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h> // semaphore library
#include <unistd.h>
//using namespace std;
sem_t lock; //lock is defined as semaphore

//routine for thread 1
void *thread1(void *varg)
{
    sem_wait(&lock);
      printf("this from Thread 1\n"); // CS
    sem_post(&lock);
    return NULL;
}
//routine for thread 2
void *thread2(void *varg)
{
```

```
    sem_wait(&lock);
      printf("this from Thread 2\n");  //CS
    sem_post(&lock);
    return NULL;
}

int main()
{
    sem_init(&lock,0,1); // lock is initialized as 1 with last
argument. middle argument is for threads(i.e. 1 for processes
and 0 for threads)
    pthread_t t1, t2;
      printf("Before Thread\n");
    pthread_create(&t1, NULL, thread1,NULL );
    pthread_create(&t2, NULL, thread2,NULL );
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
      printf("After Thread\n");
return 0;
}
```