Lab No. 9
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 semaphore
```

2) Solved Lab Activities

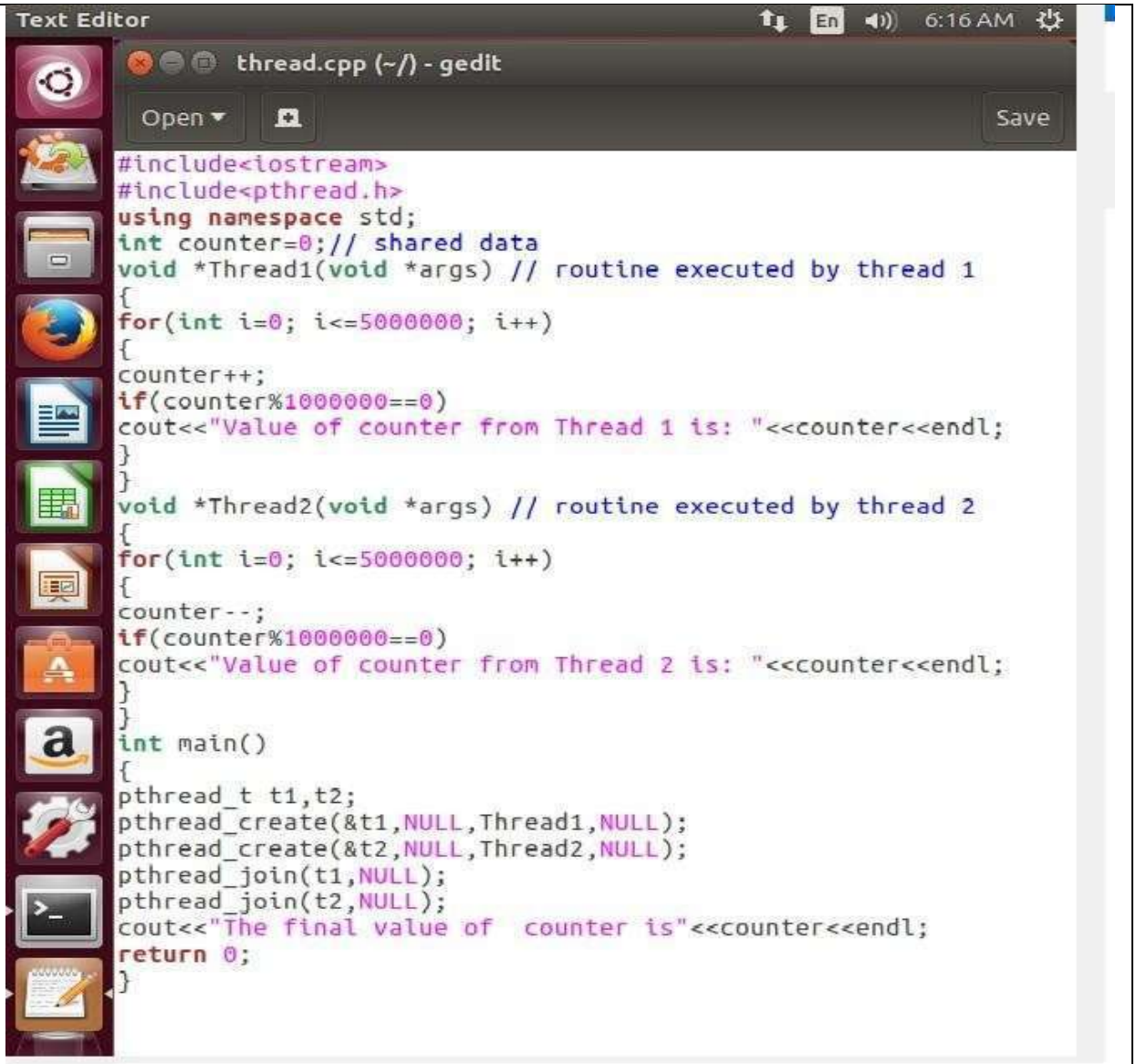
Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*20*	*Medium*	*CLO-7*
2	*9*	*Medium*	*CLO-7*
3	*15*	*Medium*	*CLO-7*
4	*15*	*Medium*	*CLO-7*

Activity 1:

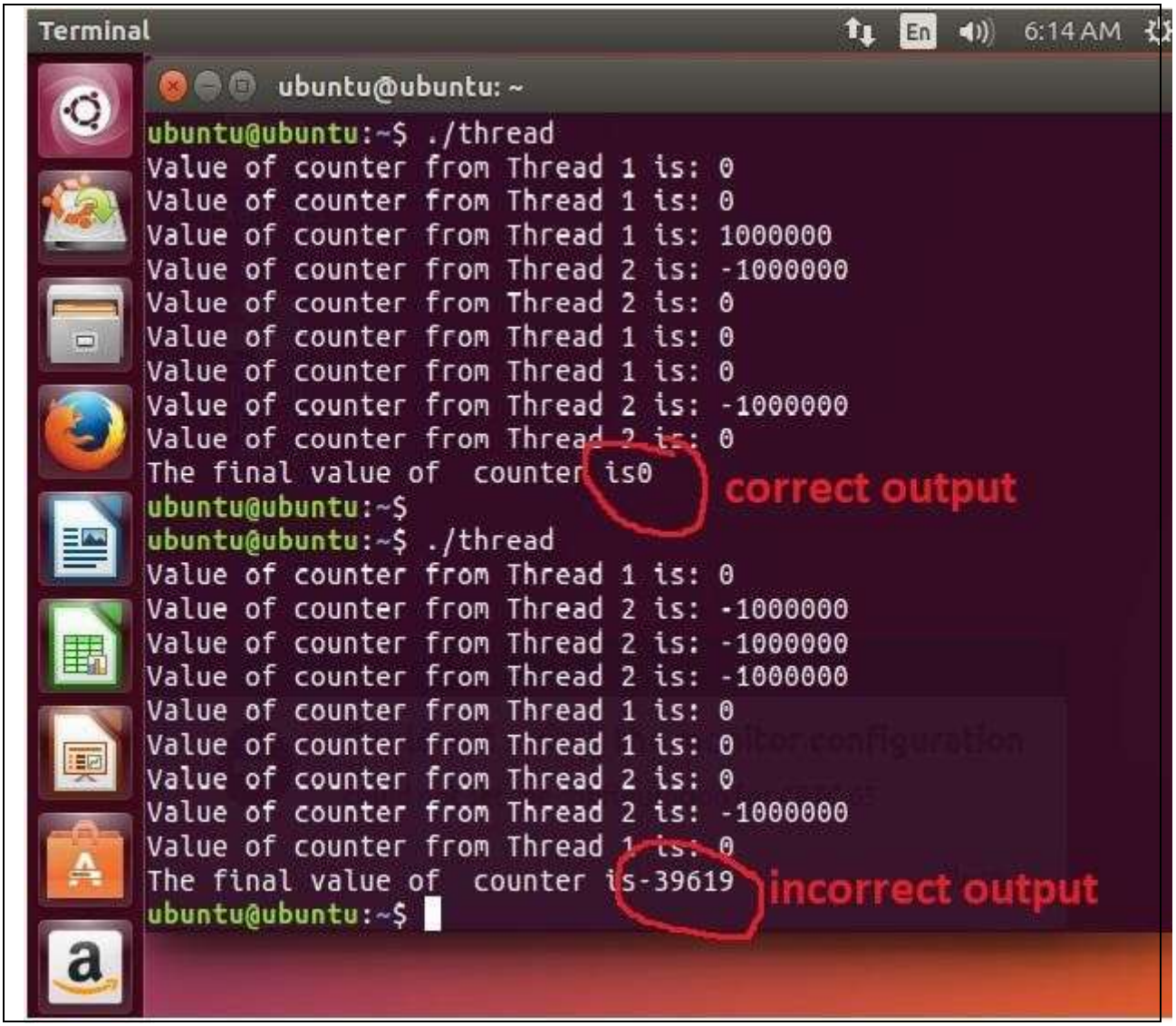
In this activity, we identify how the concurrent access to shared data may result in data inconsistency. We create two threads that manipulate a shared variable counter concurrently. As a result, sometimes we get correct output while sometimes the output is not c

Solution:

Code


Text Editor
En
6:16 AM
thread.cpp (~/) - gedit
Open
Save
#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:

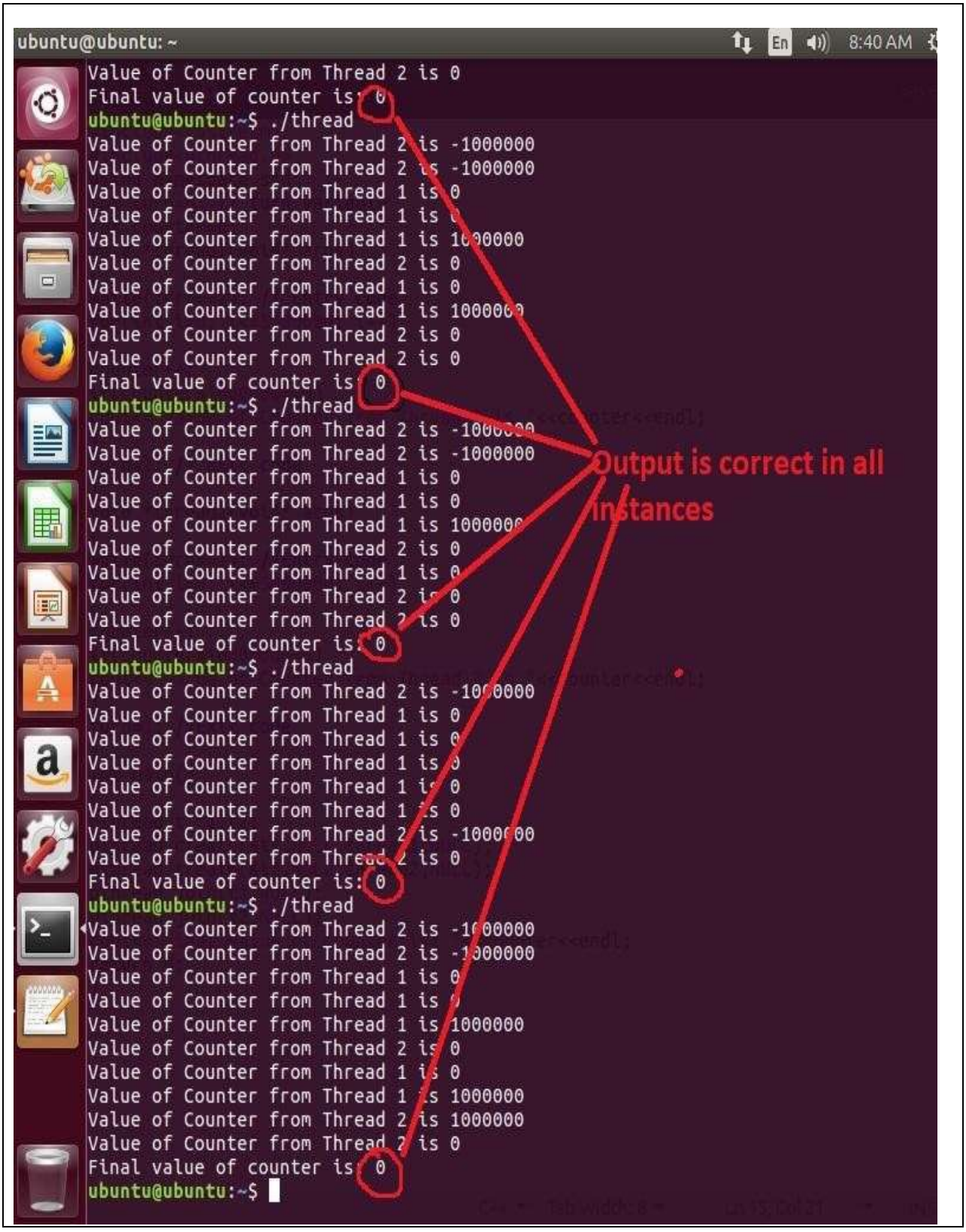
Code

thread.cpp (~/) - gedit

Open

```
#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:

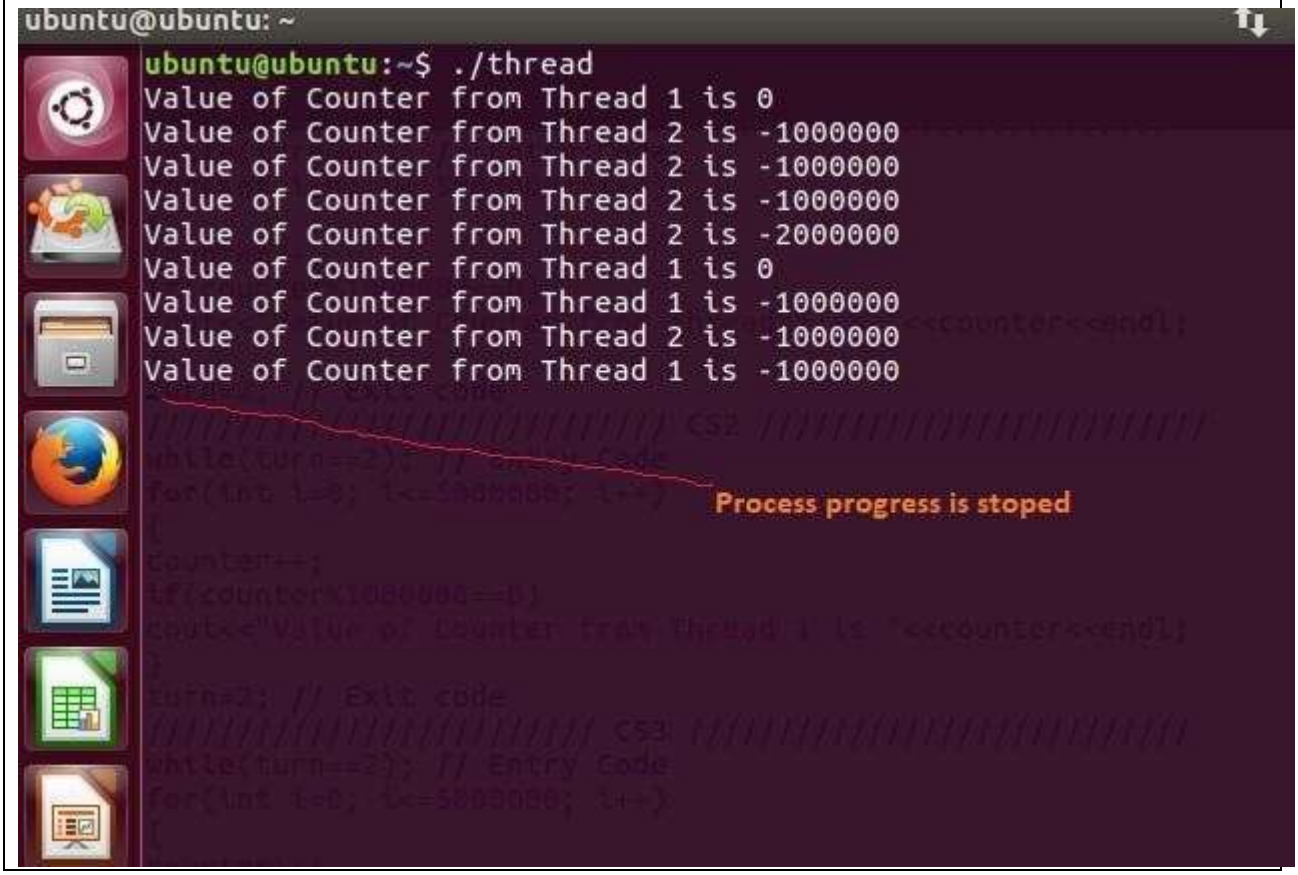
In this activity, we show that how the progress in not satisfied in simple solution

Solution:

Code



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:

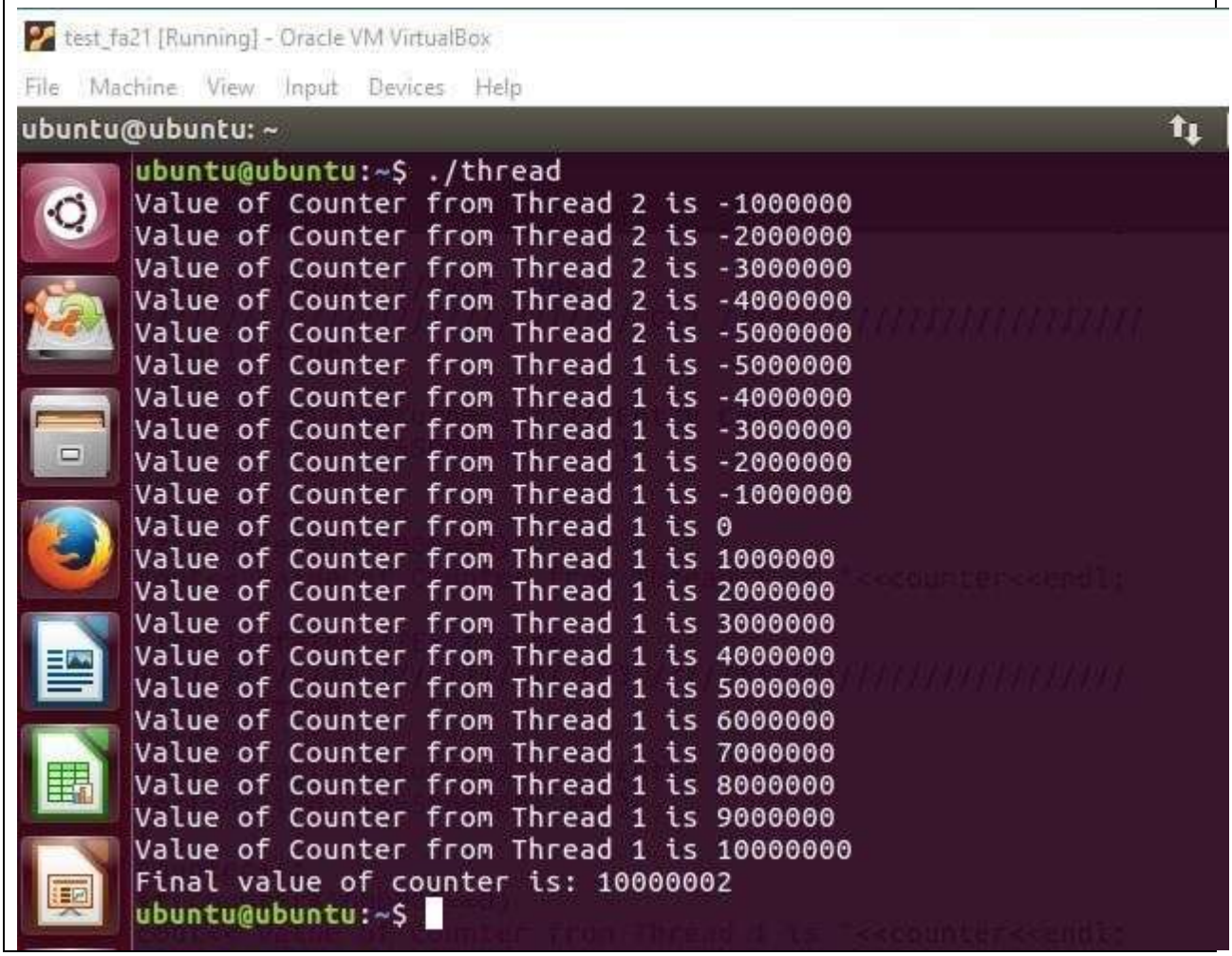
Code

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

Lab Exercise and Summary

Summary should cover Introduction, Procedure, Data Analysis and Evaluation

LABORATORY SKILLS ASSESSMENT (Psychomotor)
Total Marks: 90

(Max Marks)	Level 1 0% ≤ S < 50%	Level 2 50% ≤ S< 70%	Level 3 70% ≤ S< G0%	Level 4 G0%≤ S ≤90%	Score (S)
Procedural Awareness (20)	Selects inappropriate Linux commands, shell scripting techniques, or process management methods.	Selects and applies partially appropriate Linux commands and techniques	Selects and applies considerably appropriate Linux commands and techniques.	Selects and applies completely appropriate Linux commands and techniques	
Practical Implementation (20)	Makes major critical errors in executing Linux commands, scripting, and system processes.	Makes numerous critical errors in executing commands and process management.	Makes minor non-critical errors in executing Linux commands and system operations.	Executes Linux commands and manages processes correctly with no errors.	
Process Management and Shell Scripting (20)	Program logic contains major errors with incorrect or contradictory script flow.	Program logic has some errors with occasional contradictions in process execution.	Program logic is mostly correct but may contain occasional redundancy or minor errors.	Program logic is completely correct with no contradictions or redundant processes.	
Syntax Correctness and Results (20)	Program does not follow proper syntax for Linux commands and shell scripting, leading to incorrect outputs	Program partially follows proper syntax, producing correct results for few inputs.	Program adequately follows proper syntax, producing correct results for most inputs.	Program fully follows proper syntax, producing accurate results for all inputs.	
Use of OS Tools (9)	Uses OS tools (like terminal, process manager) with limited competence.	Uses OS tools with some competence.	Uses OS tools with considerable competence.	Uses OS tools proficiently with a high degree of competence.	
Safety (9)	Requires constant reminders to follow system safety procedures (e.g., file permissions, process handling).	Requires some reminders to follow system safety procedures.	Follows system safety procedures with minimal reminders.	Routinely follows system safety procedures.	
				Marks Obtained	

LABORATORY SKILLS ASSESSMENT (Affective)
Total Marks: 40

Marks)	Level 1 0% ≤ S < 50%	Level 2 50% ≤ S < 70%	Level 3 70% ≤ S < G0%	Level 4 G0% ≤ S ≤ 90%	Score (S)
Attitude s Engagement (5)	Shows little interest in lab activities; does not participate actively.	Participates occasionally but lacks enthusiasm and consistency.	Engages actively in most lab activities with interest.	Highly motivated, participates enthusiastically, and shows a proactive approach	
Responsibility s Punctuality (5)	Frequently misses deadlines and is often late to lab sessions.	Occasionally late or misses deadlines but tries to catch up.	Submits work on time and attends lab sessions regularly.	Always punctual, meets deadlines, and takes full responsibility for assigned tasks.	
Collaboration s Teamwork (9)	Rarely collaborates, struggles to work in a team, and does not contribute effectively.	Works with team members occasionally but struggles with communication.	Cooperates well, contributes effectively, and maintains professional interactions.	Actively engages in teamwork, supports peers, and demonstrates excellent collaboration.	
Communicatio n s Presentation Skills (9)	Struggles to explain concepts, unclear verbal/written communication.	Communicates ideas with some clarity but lacks confidence or coherence.	Presents ideas effectively with minor issues in clarity or structure.	Communicates clearly, confidently, and effectively in all aspects of lab work.	
Report Quality (9)	Report contains many errors.	Report is somewhat organized with some spelling or grammatical errors.	Report is well organized and cohesive but contains some grammatical errors.	Report is well organized and cohesive and contains no grammatical errors. Presentation seems polished.	
				Marks Obtained	