**Classic Problem of Synchronization:**

**SLEEPING TEACHING ASSISTANT**

**CMPE 180 -94 PROJECT REPORT**

**Fall, 2014**

**Submitted by:**

**Preeti Krishnan**

**Sandyarathi Das**

**Saranya Mohan**

**Shruthi Narayanan**

**Table of Contents**

ABSTRACT: 3

UNDERSTAND CONCEPTS OF MULTI-THREADING AND PROCESS SYNCHRONIZATION: 4

Critical section problem: 4

POSIX Thread Library: 4

CREATING THREADS: 5

Thread Join : 6

Mutex Locks: 6

Semaphores: 7

Semaphore Routines: 7

PROBLEM STATEMENT 8

Solution Approach 8

Description : 9

SINGLE SLEEPING TA SCENARIO : 10

PSEUDO CODE: 10

Source Code : 11

Output screen shots : 14

MULTIPLE SLEEPING TA SCENARIO 15

PSEUDO CODE: 15

Source Code : 17

Output Screen Shots : 20

CONCLUSION 21

References : 21

# ABSTRACT:

This project ventures into the understanding the concepts of Multi-threading and thread synchronization and implement a solution using POSIX mutex locks and semaphores. A multithreaded process environment involves quite a few important aspects to be taken care of. As these parallel or concurrent processes feed on the same-shared data, the most important aspect is to maintain the integrity of the data. Multithreaded program acting on the data may try to update multiple related variables without a separate thread making conflicting changes to that data. This is typically a critical section. The solution to avoid a critical section problem must employ mutual exclusion, progress of process requests, bounded waiting.

When multiple cooperating sequential processes share the same set of data, mutual exclusion must be provided to ensure that a critical section of code is used by only one process or thread at a time. Mutex locks and semaphores can be used to solve various synchronization problems and can be implemented efficiently. It can also deploy inter process communication between the participating processes and threads.

The Teaching Assistant-Student problem overlays a platform to understand the above mentioned objectives, through which inter-process communication and synchronization problems between multiple operating processes could be demonstrated and resolved effectively. Using POSIX threads, mutex locks and semaphores we intend to design and implement a solution that coordinates the activities of the Teaching Assistant and the students. The initial plan of the report is to implement a solution for coordination of activities of a single Teaching Assistant and Students and we are gradually building up for deploying the same solution in coordinating activities of multiple Teaching Assistants and Students.

# UNDERSTAND CONCEPTS OF MULTI-THREADING AND PROCESS SYNCHRONIZATION:

A thread is nothing but execution of smallest unit of programming instruction by an Operating System. Multiple threads can exist independently within a system and can either share a logical address space or shared data. Multithreading allows execution of multiple threads within single processor. It allows faster responsiveness, effective system utilization, sharing and communication as well as parallel execution of multiple tasks. As multiple processes can access same data, it may result in data integrity issues and race condition where execution of certain part of the code depends on the order in which the code was accessed by several threads. This results in the need for process synchronization.

## Critical section problem:

Every process has a segment code termed as critical section, in which the process may update/ read/ write a file or change some variable or value. When multiple processes are running simultaneously, if all the processes access their critical section simultaneously, it may lead to issues of data integrity and race condition. Hence when one process is accessing its critical section, other process must be prevented from accessing theirs’. This is done by creating an entry section where a process must request the system before executing its critical section, followed by exit section. The rest of the code is executed in remainder section. Here, the processes must have mutual exclusion, progress and bounded waiting.

do

{

Entry section;

// Critical section

Exit section;

// Remainder section

} while (true);

Critical section may have software-based tools like Mutex locks and Semaphores.

## POSIX Thread Library:

**POSIX Threads**, or **Pthreads**, is a POSIX standard for threads. The standard, POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995), defines an API for creating and manipulating threads. Implementations of the API are available on many Unix-like POSIX systems such as **FreeBSD**, **NetBSD**, **GNU/Linux**,**Mac OS X** and **Solaris**, but **Microsoft Windows** implementations also exist. For example, the pthreads-w32 is available and supports a subset of the Pthread API for the Windows 32-bit platform.

The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification. The latest version is known as IEEE Std 1003.1, 2004 Edition.

Pthreads are defined as a set of C language programming types and procedure calls, implemented with a **pthread.h**header file. In GNU/Linux, the pthread functions are not included in the standard C library. They are in **libpthread**, therefore, we should add **-lpthread** to link our program.

Pthread API can be grouped into following four types.

* Thread management : Routines that directly work on creating, detaching, joining or any other thread manipulations. They also include functions to set/query thread attributes such as joinable, scheduling etc.
* Mutexes:  
  They are like a “mutex” used for thread synchronization. They for used for creating, destroying, locking and unlocking mutexes. It also includes other attribute functions that set or modify mutex attributes.
* Condition variables: They are associated with thread communications and share a mutex, based on the condition. They include functions to create, destroy, wait and signal based upon specified variable values and functions to set/query condition variable attributes.
* **Synchronization:**  
  They are routines that manage read/write locks in a process.

## CREATING THREADS:

The main() routine itself is a single, default thread. All other threads must be explicitly created by the programmer. The function used to create thread is **pthread\_create.** Using pthread\_create routine we generate threads that share the data from the master thread and perform concurrent operations, exit their routines and then join back to the master thread. This routine can be called any number of times from anywhere within our code. The parameters associated with the pthread\_create() routine is:

***pthread\_create (pthread\_t \*thread, pthread\_attr\_t \*attr, void \*(\*start\_routine)(void \*), void \*arg)***

The arguments passed are:

* **thread** : An identifier for the new thread returned by the subroutine. Pointer to thread returns the memory location to which the variable points when the thread is created. This identifier enables us to refer to the thread.
* **attr**: An attribute object that may be used to set thread attributes. Null for the default values.
* **start\_routine**: This specifies a particular routine that the thread will execute once created.
* **arg**: A single argument that may be passed to start\_routine. It must be passed as a void pointer. Null may be used if no argument is to be passed.

The maximum number of threads that may be created by a process is implementation dependent. Once created, threads can create more threads and are independent.

## Thread Join :

A thread can wait for the termination of other thread using pthread\_join() routine. This can be defined as:

***int pthread\_join (pthread\_t th, void \*\*thread\_return)***

* The first parameter is the thread for which to wait, the identified that **pthread\_create** filled in for us.
* The second argument is a pointer to a pointer that itself points to the return value from the thread. This function returns zero for success and an error code on failure.

Thread join can be used if a parent wants to join with one of its child threads. Thread join has the following activities:

* Suppose a parent thread executes thread\_join in to join with a child thread, which is still executing. Here, parent thread is suspended until child terminates. The parent resumes only when the child exits.
* When parent thread executes a thread join and child thread has already terminated, parent will continue its execution.

Thus, pthread\_join() subroutine blocks the calling thread until the specified thread terminates.

## Mutex Locks:

These are basically used for providing mutual exclusion between the processes to avoid race conditions and prevent critical section problem. The process must first acquire lock for entering critical section using acquire() function and must release the lock once done using release() function so that other processes can access the critical section. While a process is executing its critical section, other process waiting for the lock enters a continuous loop leading to a spinlock. This is termed as “busy waiting”. This is the main disadvantage using a mutex lock.

## Semaphores:

Semaphores are more robust and sophisticated tools for process synchronization. A semaphore S is an integer variable that can be initialized and performs 2 atomic operations termed as wait() and signal(). It is defined as follows:

wait(S)

{

while (S <= 0)

; // busy wait

S--;

}

signal(S)

{

S++;

}

While one process is modifying the semaphore value, other process cannot simultaneously modify it. There are 2 types of semaphores:

1. **Binary semaphore** : Can be either 0 or 1. They are similar to mutex locks.
2. **Counting semaphore** : Can have an unrestricted domain range. They can be used to control access to a resource having finite instances.

### Semaphore Routines:

Semaphore Initiation:

A semaphore can be initiated using sem\_init() routine. The routine can be defined as follows:

***int sem\_init(sem\_t \*sem, int pshared, unsigned value);***

It takes a reference to the semaphore, a pshared integer value which is 1 for indicating the semaphore is being shared by processes and 0 for indicating it is being shared by threads and the third argument is an initial value given to the semaphore.

Semaphore Manipulation:

A semaphore can be manipulated with two routines to follow the POSIX standard:

* sem\_wait()
* sem\_post()

Sem\_wait() is synonymous to the wait function defined for semaphores.

int **sem\_wait**(sem\_t \*s)

{

// wait until value of semaphore s is greater than 0

// decrement the value of semaphore s by 1

}

Sem\_post() is synonymous to the signal function defined for semaphores.

int **sem\_post**(sem\_t \*s)

{

// increment the value of semaphore s by 1

// if there are 1 or more threads waiting, wake 1

}

# PROBLEM STATEMENT

The Operating Systems course under Computer Science department in San Jose State University has Teaching Assistants (TAs) who helps graduate students with their course work, responds to emails regarding doubts posted by the students and assesses the students during regular office hours.

The TA’s office is quite small and has room to accommodate a maximum of 3 TA's. The number of waiting seats available for the students depends on the number of TAs present for the day. Each TA has a waiting capacity of 3 waiting chairs so if all the TAs are available for the day then there are 9 waiting chairs available in the hallway outside the office where students can sit and wait. When there are no students waiting for help during office hours, the TAs sleeps.

If a student arrives during office hours and finds out that at least one of the TA is available, then the student enters the TA's office and approaches the available TA. If a student arrives and finds that all of the TAs is currently occupied, then the student sits on the chairs in the hallway and wait until one of the TA becomes available. If no chairs are available, the student will come back at a later time.

## Solution Approach

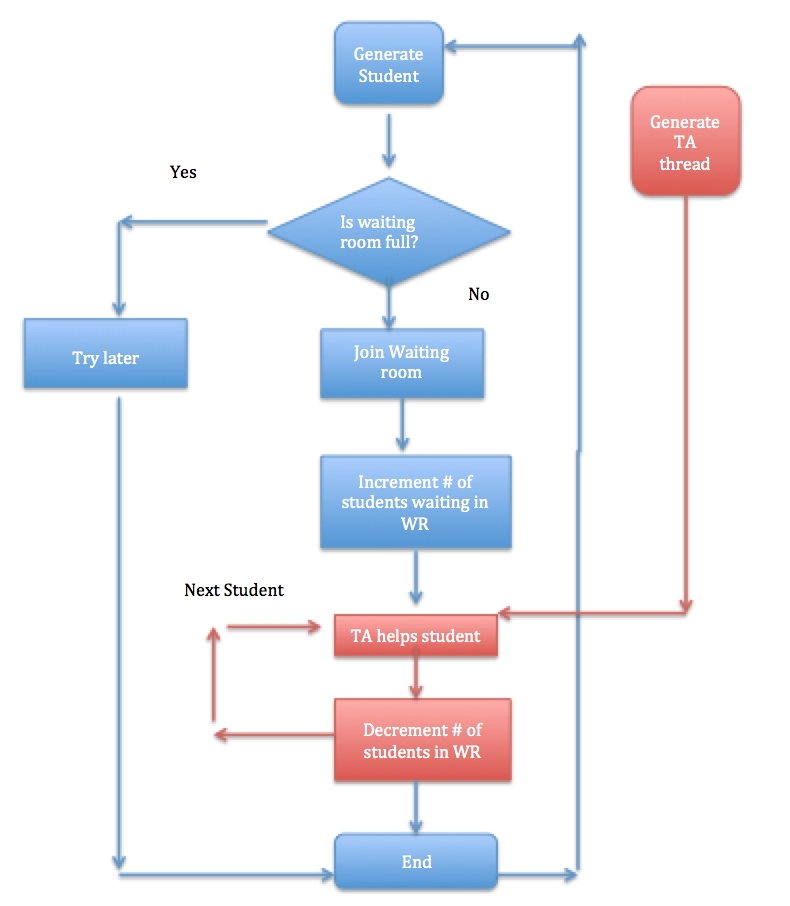
Using Pthreads begin by creating n students. Each will run as a separate thread. The TA will run as a separate thread as well. Student threads will alternate between programming for a period of time and seeking help from the TA. If the TA is available, they will obtain help. Otherwise, they will either sit in a chair in the hallway or, if no chairs are available, will resume programming and will seek help at a later time. If a student arrives and notices that the TA working alone, the student must notify the TA using a semaphore. When the TA finishes helping a student, the TA must check to see if there are students waiting for help in the hallway. If so, the TA must help each of these students in turn. If no students are present, the TA may return to his work.

### Description :

The flow chart can be explained as follows:

* Separate threads are created for both students and teachers.
* The blue paths indicates the student thread flow, the red path indicates TA thread flow.
* The student thread checks if waiting room is available and joins the waiting room if the room is not full, thereby incrementing the number of students in the waiting room.
* If the waiting room is full, the student leaves and returns later.
* The TA checks the waiting room if student is available. If there are students in waiting room, TA helps students one by one, thereby decrementing number of students in the waiting room.

The flow chart for Sleeping TA problem is as follows:



# SINGLE SLEEPING TA SCENARIO :

In Single TA problem, the course has just one TA who addresses students’ queries and responds to their mails during his office hour. The TA’s office is small and can maximum have one student inside. There is a waiting room outside the TA office having 3 waiting chairs. The students approaching the TA first enter the waiting room and check the availability of the room. If the TA is available for the day, then the student enters the TA's office and approaches the available TA. Other students wait in waiting room chairs for their chance. If all the chairs in the waiting room are full, then the student leaves the room.

The TA checks if there are any students in the waiting room and starts addressing them one by one. When there are no students waiting for help during office hours, the TA sleeps.

## PSEUDO CODE:

#define CHAIRS 5 */\* # chairs for waiting students \*/*

typedef int semaphore; */\* declare semaphores\*/*

semaphore studentAvailable = 0; */\* # of student waiting for help\*/*

semaphore taReady = 0; */\* ta waiting for student\*/*

Semaphore waitingRoom = 1; */\* for mutual exclusion of waiting room access\*/*

int studentsInWaitingRoom = 0; */\* number of students waiting\*/*

void taRoutine(void)

{

while (TRUE)

{

wait (&studentAvalable); */\* go to sleep if # of students is 0 \*/*

wait (&waitingRoom); */\* acquire access to "waiting room \*/*

studentsInWaitingRoom = studentsInWaitingRoom -1;

*/\* decrement count of waiting students \*/*

*/\*ta is helping student\*/*

signal (&taReady); */\* ta is ready for next student\*/*

signal (&waitingRoom); */\* release access to 'waiting’ room \*/*

}

}

void studentRoutine(void)

{

wait (&waitingRoom); */\* enter critical region i.e access waiting room \*/*

if (waiting < CHAIRS) */\* if there are no free chairs, leave \*/*

{

studentsInWaitingRoom = studentsInWaitingRoom + 1;

*/\* increment count of waiting students\*/*

signal (&studentAvailable); */\* wake up ta if necessary \*/*

signal (&waitingRoom); */\* release access to 'waiting' \*/*

wait (&taReady); */\* go to sleep if # of free barbers is 0 \*/*

}

Else

{

signal (&waitingRoom); */\* room is full; do not wait \*/*

}

}

## Source Code :

**#include** <pthread/pthread.h>

**#include** <stdlib.h>

**#include** <sys/\_pthread/\_pthread\_t.h>

**#include** <sys/\_types/\_time\_t.h>

**#include** <sys/semaphore.h>

**#include** <unistd.h>

**#include** <ctime>

**#include** <iostream>

**#include** <thread>

**using** **namespace** std;

**#define** NUMBER\_OF\_STUDENTS 5

/\*Indicate student Id of the student currently with TA\*/

**int** stuID=100;

/\*Time variable to control the TA's working hours.\*/

time\_t end\_time;

/\*Semaphore declaration :

\*waitingRoom semaphore : Access the waiting room.

\*taReady semaphore : indicate TA is free or TA is helping student.

\*studentAvailable semaphore :indicate student is waiting for help in room.

\*/

sem\_t waitingRoom,studentAvailable,taReady;

/\*The number of student waiting for TA in waiting room\*/

**int** studentsInWaitingRoom=0;

/\*The TA thread checks if student is available.

\*Attends to one student and then chooses the next.

\*This routine first checks the studentAvailable semaphore,

\*acquires the waiting room semaphore to acquire a student and

\*then releases it after helping the student and again

\*indicates its availability using taReady semaphore.

\*/

**void** \***teachingAssistantRoutine**(**void** \*);

/\*The studentRoutine thread first attempts to access the waiting room ,

\* and then after gaining the lock,it occupies one of the chairs,

\* indicates that it is ready and then waits for the TA to be ready.

\* In case the waiting room is full,it leaves the waiting room to try later.

\*/

**void** \***studentRoutine**(**void** \*);

**int** **main**()

{

/\*TA working hours is set to 20s\*/

end\_time=**time**(NULL)+20;

/\*Semaphore initialization\*/

/\*The waiting room lock is initially made available.\*/

**sem\_init**(&waitingRoom,0,1);

/\*The student Available indicator is initially set to 0 i.e no student.\*/

**sem\_init**(&studentAvailable,0,0);

/\*The TA signal is initially set to 1 i.e TA is available.\*/

**sem\_init**(&taReady,0,1);

/\*TA thread creation and initialization\*/

pthread\_t taThreadId,studentThreadId;

**int** taThreadStatus=

**pthread\_create**(&taThreadId, NULL, teachingAssistantRoutine, (**void** \*)0);

**if**(taThreadStatus!=0)

cerr<<"\nCreation of TA thread failed!!"<<endl;

/\*Student thread creation and initialization\*/

**int** studThreadStatus=

**pthread\_create**(&studentThreadId,NULL,studentRoutine,(**void** \*)0);

**if**(studThreadStatus!=0)

cerr<<"\nCreation of student thread failed!!"<<endl;

/\*Student threads are blocked first.\*/

**pthread\_join**(studentThreadId,NULL);

/\*TA thread is blocked next.\*/

**pthread\_join**(taThreadId,NULL);

**exit**(0);

}

/\*TA thread routine \*/

**void** \***teachingAssistantRoutine**(**void** \*)

{

/\*routine runs till the end of TA hrs or

\* till a student is available in the waiting room.

\*/

**while**(**time**(NULL)<end\_time || studentsInWaitingRoom>0)

{

**sem\_wait**(&studentAvailable);/\*Check if resource is available\*/

**sem\_wait**(&waitingRoom);/\*Checks if waiting room is accessible\*/

studentsInWaitingRoom--;/\*Make one chair free in the waiting room.\*/

stuID++;

//cout<<"TA is helping student with Id: "<<this\_thread::get\_id()<<endl;

cout<<"TA is helping student with Id: "<<stuID<<endl;

cout<<"Number of students in waiting room :"<<studentsInWaitingRoom<<endl;

**sem\_post**(&waitingRoom);/\*Release waiting room access\*/

**sem\_post**(&taReady);/\*Indicate its availability for the next student\*/

**sleep**(3);

}

**return** 0;

}

/\*Student thread routine\*/

**void** \***studentRoutine**(**void** \*)

{

/\*routine runs till the end of TA working hours\*/

**while**(**time**(NULL)<end\_time)

{

**sem\_wait**(&waitingRoom);/\*access the waiting room\*/

/\*Run if there are free seats to be occupied in the waiting room.\*/

**if**(studentsInWaitingRoom<NUMBER\_OF\_STUDENTS)

{

studentsInWaitingRoom++;//Occupy a free chair

cout<<"Student joins Waiting room!!Number of students in waiting :"

<<studentsInWaitingRoom<<endl;

**sem\_post**(&waitingRoom);/\*Release waiting room access\*/

**sem\_post**(&studentAvailable);/\*Indicate student availability\*/

**sem\_wait**(&taReady);/\*check for ta availability\*/

}

/\*Run if there are no free seats to be occupied in the waiting room.\*/

**else**{

/\*release waiting room access, if it is full and leave\*/

**sem\_post**(&waitingRoom);

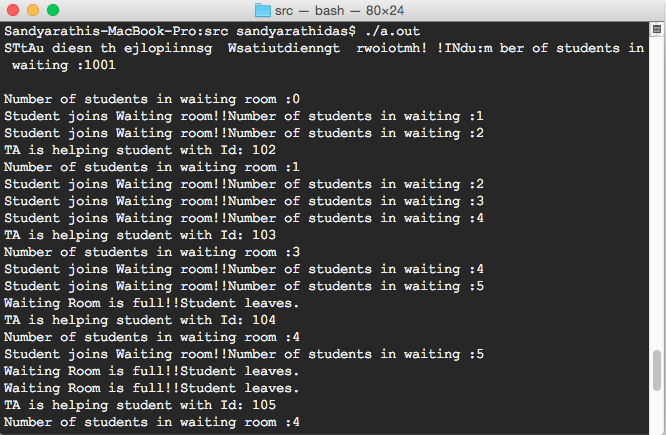
cout<<"Waiting Room is full!!Student leaves."<<endl;

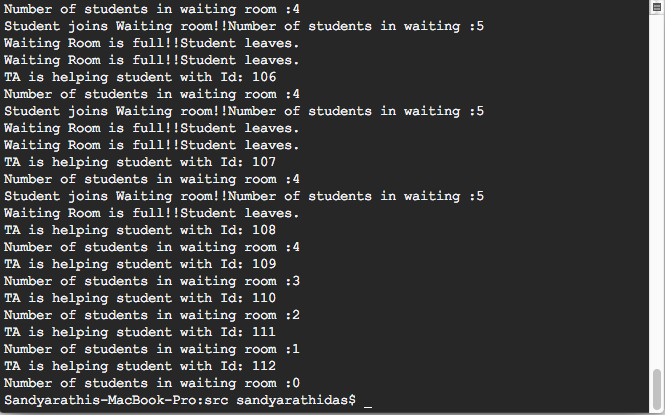
}

}**return** 0;

}

## Output screen shots :





# MULTIPLE SLEEPING TA SCENARIO

This is an extended version of our Single TA problem. In this case, there are multiple TAs and many students approaching them. Students come to address their queries with multiple TAs depending on the their availabilityA TA’s office room has m chairs, a computer and a waiting room with n chairs. If no students to be addressed, all the TAs sleeps. As the students approach TA for help, they check if the TA is available. They enter the waiting room and check its availability as well. If all waiting room chairs are occupied, then the student leaves the office. If all TAs are busy but chairs are available, then the student waits in one of the waiting room chairs.

The functionality of the TA remains the same as that of single TA problem, except there are multiple TAs for addressing the students.

## PSEUDO CODE:

waiting\_queue\_size=9

semaphore ta\_sem,student\_sem,wait\_sem; *//declare semaphores*

main()

{

*//initialize semaphores*

init(ta\_sem,3);

init(student\_sem,0);

int(wait\_sem,1); *//binary semaphore to synchronize access to waiting queue*

pthread\_t ta\_id[3], student\_id[13]; *//ids for TA and Student threads*

*//create threads for TAs and Students*

 for(int i=1;i<=3;i++)

{

*//create TA threads*

  pthread\_create(ta\_id[i],NULL,(void \*)ta\_function, (void\*)&i);

}

for(int j=1;j<=13;j++)

{

*//create Student threads*

  pthread\_create(student\_id[i],NULL,(void \*)student\_function, (void\*)&j);

 }

*//Joining the student threads*

 for(int k=1;k<=13;k++)

{

  join(student\_id[k]);

 }

 ta\_function(void \*i)

{

 while(1)

{

  wait(student\_sem); *//wait for a student "j" to approach*

  signal(ta\_sem) *//TA "i" gets ready to help student*

 }

}

student\_function(void \*j)

{

  wait(wait\_sem); *//getting access to modify the wait queue*

  if(waiting\_queue\_size>0)

{

  waiting\_queue\_size--;

  signal(wait\_sem); *//releasing the wait queue*

  signal(student\_sem); *//Student "j" approaches TA*

  wait(ta\_sem) *//wait for TA "i" to help*

  wait(wait\_sem);

  waiting\_queue\_size++;

  signal(wait\_sem); *//releasing the wait queue*

  }

Else

{

signal(wait\_sem);

*//The wait queue is full hence release the wait queue. The student leaves and comes back later*

}

}

## Source Code :

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

#include<semaphore.h>

#include<sys/ipc.h>

// Maximum capacity of the waiting room.

# define MAX\_WAITING\_ROOM\_SIZE 9;

//Function to create the TA threads and control the activities

void ta\_function(void \*arg);

//Function to create the student threads and control the activities

void student\_function(void \*arg);

/\*Semaphore declarations for synchronizing the access to the shared variable

'waiting\_room\_count' and synchronizing the activities of ta and students\*/

sem\_t wait\_queue\_sem, student\_sem, ta\_sem;

//Initializing the shared variable to the maximum capacity of waiting room

int waiting\_room\_count = MAX\_WAITING\_ROOM\_SIZE;

// variable to keep track of the number of students approching TAs

int stud\_count = 0;

int main(){

//Declaring the ids for TA and student threads

pthread\_t ta\_id[3], student\_id[13];

int thread\_status = 0;

int i, j, k;

//Initializing the semaphores

sem\_init(&wait\_queue\_sem, 0, 1);//Binary semaphore to control the access to the shared variable 'waiting\_room\_count'

sem\_init(&student\_sem, 0, 0);//semaphore to indicate if a student is available or not

sem\_init(&ta\_sem, 0, 3);//semaphore to indicate if a TA is free or helping student.

//Creating threads for TAs

for (i = 1; i <= 3; i++){

thread\_status = pthread\_create(&ta\_id[i], NULL, (void \*)ta\_function, (void\*)&i);

sleep(1);

if (thread\_status != 0)

printf("Creating TA Thread failed!\n");

}

//Creating threads for students

for (j = 1; j <= 13; j++){

thread\_status = pthread\_create(&student\_id[j], NULL, (void \*)student\_function, (void\*)&j);

sleep(1);

if (thread\_status != 0)

printf("Creating student Thread failed!\n");

}

//Blocking student threads

for (k = 1; k <= 13; k++)

pthread\_join(student\_id[k], NULL);

exit(0);

}

/\*Initially each TA waits for the student to approach.Once the student approaches,

the TA starts helping the student.\*/

void ta\_function(void \*arg)

{

int j = \*(int \*)(arg);

printf("TA %d is waiting for student.\n", j);

while (1)

{

sem\_wait(&student\_sem);//Checks if student available

printf("TA %d is helping student.\n", j);

sem\_post(&ta\_sem);//Indicates that a TA is available

sleep(1);

}

pthread\_exit(0);

}

/\*The student arrives and enters the waiting room. If the waiting room is full, the student leaves.If waiting room is not full, the student waits for the TA. Once a TA is available,

the student approaches the TA and gets his help.\*/

void student\_function(void \*arg)

{

int n = \*(int \*)(arg);

sem\_wait(&wait\_queue\_sem);

stud\_count++;

printf("Student %d arrives.\n", n);

if (waiting\_room\_count>0){//Checks if the waiting room is full.

waiting\_room\_count--;//Decreasing the number of free seats in the waiting room when a student occupies it.

printf("Student %d waits for TA.\n", n);

sem\_post(&wait\_queue\_sem);

printf("Student %d approaches TA.\n", n);

sem\_post(&student\_sem);//Indicates that a student is available

sleep(2);

sem\_wait(&ta\_sem);//Checks if TA available.

printf("Student %d leaves.\n", n);

sem\_wait(&wait\_queue\_sem);

waiting\_room\_count++; // Increases the number of free seats in the waiting room, once the TA helps the student.

sem\_post(&wait\_queue\_sem);

}

else{

sem\_post(&wait\_queue\_sem);

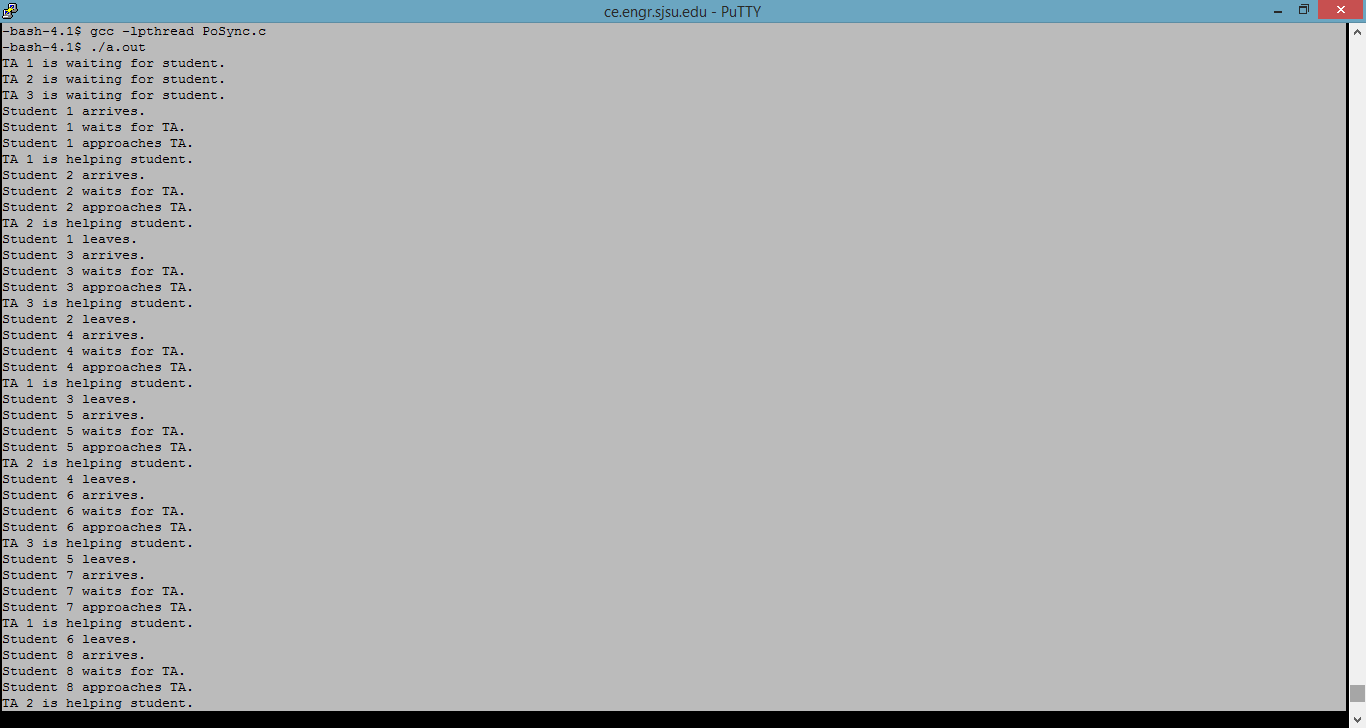
printf("No seats avaialble to wait.Student %d leaves\n", n);

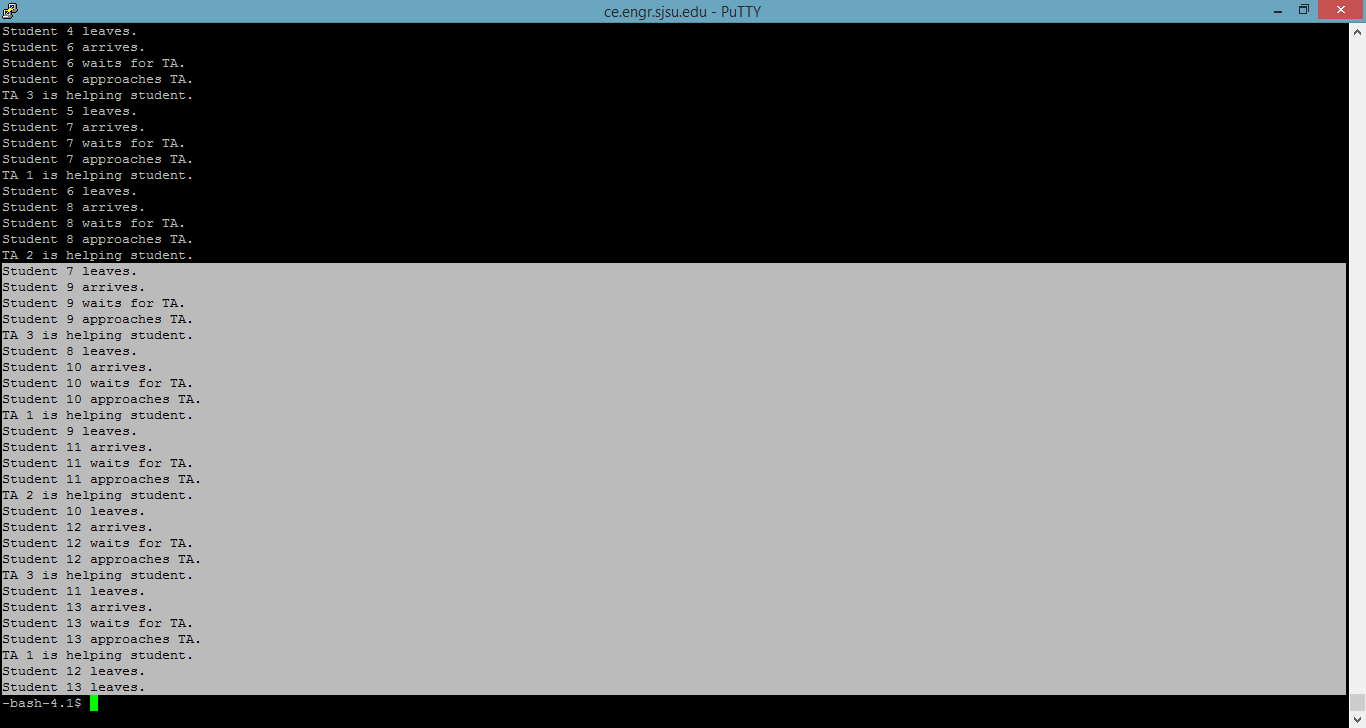
}

pthread\_exit(0);

}

## Output Screen Shots :





# CONCLUSION

Thus by using POSIX threads and semaphores, we can solve the single TA and multiple TAs problem based on multithreading and synchronization.

For solving the single TA or multiple TA problems, we need to first create separate threads for TA and n students. Students either seek help from the TA, if he is available; else sit in the waiting room chairs. If no chairs are available, the students will leave and come again later. All this can be done using Pthread implementation.

In order to notify the TA when the students arrive, help TA monitor the number of students in the waiting room and also TA sleeps when no students are in the waiting room. These can be implemented with the help of semaphores.

# References :