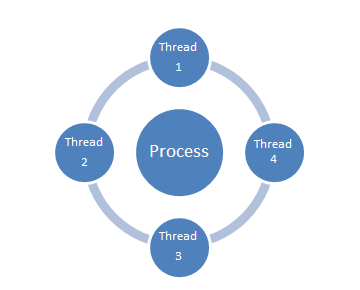
**PoSync-**

**Project on Multi-Thread Synchronization**

****

**Project Draft v 1.0**

**Fall semester, 2014**

|  |  |
| --- | --- |
| **Team Members** | **Email address** |
| Saranya Mohan | saranya.mohan@sjsu.edu |
| Preeti Krishnan | preeti.krishnan@sjsu.edu |
| Shruthi Narayanan | shruthi.narayanan@sjsu.edu |
| Sandyarathi Das | sandyarathi.das@sjsu.edu |

**Table of Contents**

Abstract 3

Introduction 4

Problem Statement 4

Explanation 5

Multi-threading 5

Thread Synchronization 6

Deadlock 7

**Solution to Single TA-Student Problem…………………………………………………………… ………. 8**

*Algorithm for single ………………………………………………………………………………………………...8*

*Implementation of the Code(C++)…………………………………………………………………………….9*

**Extension of the Problem……………………………………………………………………………………………..**

Approach 8

Thread Activities 8

TA thread 9

Student thread activities 9

Wrap Up & References: 11

**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.

**INTRODUCTION:**

**Problem Statement:**

A university computer science department has a teaching assistant (TA) who helps graduate students with their course work,  responds to emails regarding doubts posted by the students and assess the students during regular office hours.The TA’s office is rather small and has room for only one desk with a chair and computer. There are three chairs in the hallway outside the office where students can sit and wait if the TA is currently helping another student. When there are no students who need help during office hours, the TA sits at the desk and finishes other works like responding to student’s mails and grading students assignments. If a student arrives during office hours and finds the TA working alone then he could ask for help without waiting. If a student arrives and finds the TA currently helping another student, the student sits on one of the chairs in the hallway and waits. If no chairs are available, the student will come back at a later time.

Using POSIX threads, mutex locks and semaphores, implement a solution that coordinates the activities of the TA and the students.

**Description:**

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.

## Multi-threading :

## Pthreads

* **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 **libpthrea**, therefore, we should add **-lpthread** to link our program.

THE PTHREAD API

Pthreads API can be grouped into four:

* **Thread management:**  
  Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes such as joinable, scheduling etc.
* **Mutexes:**  
  Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
* **Condition variables:**  
  Routines that address communications between threads that share a mutex. Based upon programmer specified conditions. This group includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.
* **Synchronization:**  
  Routines that manage read/write locks and barriers.

CREATING THREADS

* Our **main()** program is a single, default thread. All other threads must be explicitly created by the programmer.
* **pthread\_create** creates a new thread and makes it executable. This routine can be called any number of times from anywhere within our code.
* **pthread\_create (pthread\_t \*thread, pthread\_attr\_t \*attr, void \*(\*start\_routine)(void \*), void \*arg)**arguments:
  + **thread:**   
    An identifier for the new thread returned by the subroutine. This is a pointer to **pthread\_t** structure. When a thread is created, an identifier is written to the memory location to which this variable points. This identifier enables us to refer to the thread.
  + **attr:**   
    An attribute object that may be used to set thread attributes. We can specify a thread attributes object, or NULL for the default values.
  + **start\_routine:**   
    The routine that the thread will execute once it is created.

**void \*(\*start\_routine)(void \*)**

We should pass the address of a function taking a pointer to void as a parameter and the function will return a pointer to void. So, we can pass any type of single argument and return a pointer to any type.   
While using **fork()** causes execution to continue in the same location with a different return code, using a new thread explicitly provides a pointer to a function where the new thread should start executing.

**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 are peers, and may create other threads. There is no implied hierarchy or dependency between threads.

JOIN

* **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.
* When a thread is created, one of its attributes defines whether the thread is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.

A thread can execute a thread join to wait until the other thread terminates. In our case, you - the main thread - should execute a thread join waiting for your colleague - a child thread - to terminate. In general, thread join is for a parent (**P**) to join with one of its child threads (**C**). Thread join has the following activities, assuming that a parent thread **P** wants to join with one of its child threads **C**:

* When **P** executes a thread join in order to join with **C**, which is still running, **P** is suspended until **C** terminates. Once **C**terminates, **P** resumes.
* When **P** executes a thread join and **C** has already terminated, **P** continues as if no such thread join has ever executed (i.e., join has no effect).

A parent thread may join with many child threads created by the parent. Or, a parent only join with some of its child threads, and ignore other child threads. In this case, those child threads that are ignored by the parent will be terminated when the parent terminates.

* The **pthread\_join()** subroutine **blocks the calling thread** until the specified thread terminates..

## Semaphores :

A **semaphore** is a **counting** and **signaling** mechanism. We use it to allow threads access to a specified number of items. If there is a single item, then a semaphore is virtually the same as a mutex.

However, it is more commonly used in a situation where there are multiple items to be managed. Semaphores can also be used to signal between threads or processes. For example, to tell another thread that there is data present in a queue. There are two types of semaphores: named and unnamed semaphores.

The semaphores form a classic system for confining access to shared assets (e.g. capacity) in a multi-processing environment. They were formulated by Dijkstra .

A semaphore is a protected variable (or abstract data type ) which can only be accessed using the following operations:

P(s) Semaphore s; while (s == 0) ; /\* wait until s>0 \*/ s = s-1;

V(s) Semaphore s; s = s+1;

Init(s, v) Semaphore s; Int v; s = v;

P and V stand for Dutch "Proberen", to test, and "Verhogen", to increment. The value of a semaphore is the number of units of the resource which are free (if there is only one resource a "binary semaphore" with values 0 or 1 is used). The P operation busy-waits (or maybe sleeps ) until a resource is available whereupon it immediately claims one. V is the inverse, it simply makes a resource available again after the process has finished using it. Init is only used to initialise the semaphore before any requests are made. The P and V operations must be indivisible, i.e. no other process can access the semaphore during the their execution.

To avoid busy-wait ing, a semaphore may have an associated queue of processes (usually a FIFO ). If a process does a P on a semaphore which is zero the process is added to the semaphore's queue. When another process increments the semaphore by doing a V and there are tasks on the queue, one is taken off and resumed.

A semaphore is as an object with an integer value that we can manipulate with two routines sem\_wait() sem\_post() to follow the POSIX standard). Because the initial value of the semaphore determines its behavior, before calling any other routine to interact with the semaphore, we must first initialize it to some value, as this code below does:

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

If the *pshared* argument is zero, then the semaphore is shared between threads of the process; any thread in this process can use *sem* for performing [*sem\_wait*()](http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_wait.html), [[TMO](javascript:open_code('TMO'))] Option Start][*sem\_timedwait*()](http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_timedwait.html), Option End] [*sem\_trywait*()](http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_trywait.html), [*sem\_post*()](http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_post.html), and [*sem\_destroy*()](http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_destroy.html) operations.

After a semaphore is initialized, we can call one of two functions to interact with it, sem\_wait() or sem\_post() [4]. The behavior of these two functions is described here: -------------------------------------------------------------------------------- int sem\_wait(sem\_t \*s) { wait until value of semaphore s is greater than 0 decrement the value of semaphore s by 1 } int sem\_post(sem\_t \*s) { increment the value of semaphore s by 1 if there are 1 or more threads waiting, wake 1 }

### The Student Routine :

**sem\_wait**(&waitingRoom) :

Wait until the value of semaphore waitingRoom is > 0

Then decrement the value of semaphore by 1

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

Increment the value of semaphore by 1

if there are 1 or more threads waiting, wake any one

**sem\_post**(&studentAvailable)

Increment the value of semaphore by 1

if there are 1 or more threads waiting, wake any one

**sem\_wait**(&taReady)

wait for the value of semaphore to be >0

and then decrement the value by 1

The TA routine :

**sem\_wait**(&studentAvailable)

Wait until the value of semaphore studentAvailable is > 0

Then decrement the value of semaphore by 1

**sem\_wait**(&waitingRoom)

Wait until the value of semaphore waitingRoom is > 0

Then decrement the value of semaphore by 1

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

Increment the value of semaphore by 1

if there are 1 or more threads waiting, wake any one

**sem\_post**(&taReady)

Increment the value of semaphore by 1

if there are 1 or more threads waiting, wake any one

**Single TA-Student Solution Flow:**

Generate Student

Generate TA thread

Yes

Is waiting room full?

No

Join Waiting room

Try later

Increment # of students waiting in WR

Next Student

TA helps student

Decrement # of students in WR

End

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;

}

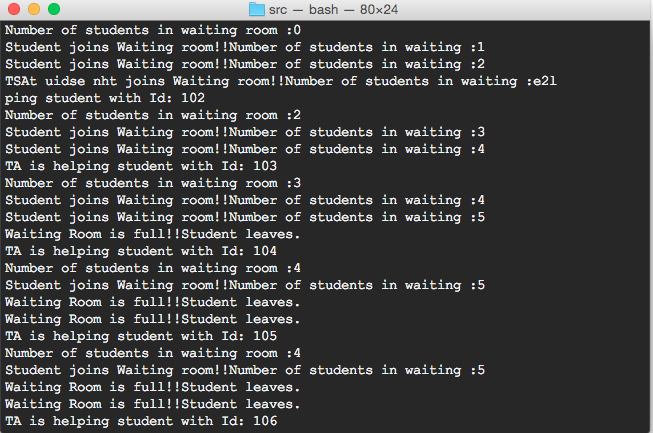
**sleep**(1);

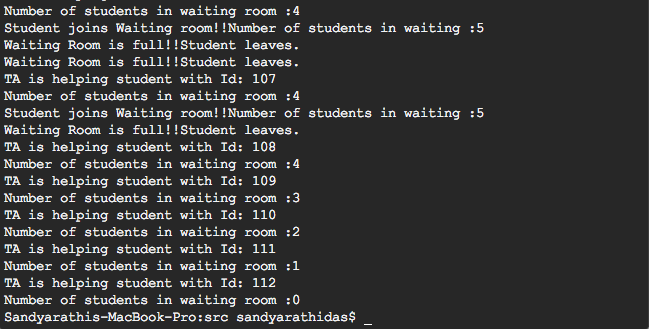
}

**return** 0;

}

Output Screens :





**Extended Problem Statement:**

**Purpose:**

Here, we will extend the initial Single-TA problem to a multiple TAs problem where many students come to address their queries with multiple TAs depending on the TA availability. Using the concept of POSIX threads, mutex locks and semaphores, we are extending our implementation for finding solution that coordinates activities of multiple TAs and Students.

**Our Extended Multiple Sleeping-TAs Problem:**

We will change our initial problem as follows:

A TA consists of an office room with 3 chairs and a computer and a waiting room with 9 chairs. If there are no students to be addressed, all the TAs resumes there work like grading assignments and responding to mails .If a student enters the TA office and all chairs are occupied, then the student leaves the office. If all the TAs are busy but chairs are available, then the student sits in one of the free chairs. If the TAs are working alone, then student can contact one of the TA’s.

**Multiple TA-Student Solution algorithm:**

waiting\_queue\_size=9

**//declare semaphores**

semaphore ta\_sem,student\_sem,wait\_sem;

main(){

**//initialize semaphores**

init(ta\_sem,0);

init(student\_sem,0);

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

pthread\_t ta\_id[3], student\_id[10]; **//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<=10;j++){

**//create Student threads**

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

}

**//Jining the student threads**

for(int k=1;k<=10;k++){

join(student\_id[k]);

}

}

ta\_function(void \*i){

while(1){

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

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

waiting\_queue\_size++;

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

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

}

}

student\_function(void \*j){

while(1){

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

if(waiting\_queue\_size>0){

waiting\_queue\_size++;

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

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

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

}

else{

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

}}}

**References :-**

* <http://en.wikipedia.org/wiki/Sleeping_barber_problem>
* <http://programmers.stackexchange.com/questions/213101/sleeping-barber-problem-with-multiple-barbers>
* <http://stackoverflow.com/questions/19692515/sleeping-barber-algorithm-with-multiple-barbers>