CMPE 180C Project (Semester: Spring 2019)

**The Sleeping Teaching Assistant Problem**

**PROJECT GROUP 04**



Submitted By:

Vatsal Makani (013731614)

Jay Parsana (013779389)

Yash Amin (013006253)

**Table of Contents**

[1. Introduction 1](#_Toc531343165)

[2. Background and Objectives 2](#_Toc531343166)

[3. Algorithm 3](#_Toc531343167)

[4. Results 4](#_Toc531343168)

[5. Conclusion and Recommendation 6](#_Toc531343169)

[References 7](#_Toc531343170)

[Appendix: Code 8](#_Toc531343171)

## **Introduction**

The Sleeping Teacher Assistant problem is analogous to the critical section problem where Mutex and Semaphores are used to avoid the Race Condition or Data Inconsistency.

The problem states that a university computer science department has a teaching assistant (TA) who helps undergraduate students with their programming assignments during regular office hours. The Teaching Assistant (TA) is analogous to the critical section of the processor where one or more process executes. The TA’s office is rather small and has room for only one desk with a chair and computer, which means that the processor can execute only one program in its critical section at a time. There are three chairs in the hallway outside the office where students can sit and wait if the TA is currently helping another student. This is analogous to the Queue, where the other process wait if they find that the processor is busy executing other processes. The processes follow the First In First Out Algorithm when entering the queue and the process enters the critical section as soon as the processor finishes the execution of the process already in the critical section.

When there are no students who need help during office hours, the TA sits at the desk and takes a nap. If a student arrives during office hours and finds the TA sleeping, the student must awaken the TA to ask for help. This is analogous to the producer consumer problem where the consumer goes to sleep if there is no item in the Buffer. If the producer produces an item and it finds the consumer to be sleeping, it awakens the consumer. 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. This is analogous to the wait() signal, in which the process enters and waits in the queue if it founds the critical section to be occupied by the other processes. If no chairs are available, the student will come back at a later time.

We need to use POSIX mutex locks and semaphores to design a solution to the sleeping TA problem. A simple solution can use one mutex lock and two semaphores as follows:

* A mutex lock for controlling access to the chairs. (This of course can be simplified by keeping track of the number of waiting students which roughly resembles the number of students sitting in chairs.)
* One semaphore for signaling the sleeping TA that a student has shown up and second semaphore for signaling a waiting student that the TA can now help them.

## **Background and Objectives**

The Sleeping TA problem involves the concepts of Threading (POSIX Threads), Mutex and Semaphores to resolve the Critical Section problem. Here the critical section is the Teaching Assistant office, where only one student can sit and ask the queries.

* **Threading:** A thread is a path of execution within a process. A process contains multiple threads. A thread is also known as lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc.
* **Semaphores and Mutex:** A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa. At any point of time, only one thread can work with the entire buffer. The concept can be generalized using semaphore. Strictly speaking, a mutex is locking mechanism used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with mutex, and only the owner can release the lock (mutex). A semaphore is a generalized mutex. In lieu of single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. The consumer and producer can work on different buffers at the same time. Semaphore is signaling mechanism. For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wake up.

**Objectives**

The Objective of the problem is to achieve an efficient solution to the above-mentioned Teaching Assistant Problem. The Teaching Assistant should be solving the problem for one student at a time and others should be waiting outside, till the student inside the TA office comes out. The program should implement a logic that prevents other students from entering the Teaching Assistant office, if the office is already occupied by another student.

The Program should implement logic to avoid the busy waiting. This should be implemented by asking the students to wait outside till the Teaching Assistant office becomes vacant. The Program should also implement the logic to accommodate no more than three students in the waiting room. If the number of students is more than three, then the remaining students should resume working if all the three chairs in the waiting room are occupied by the students. These processes/ students can resume working and should come back once there is space available in the waiting room.

The solution mentioned above also adheres to the three critical requirements to avoid a race condition. The implementation follows Mutual Exclusion, Bounded Waiting and Continuous Progress.

## **Algorithm**

**Structure:**

* Student: student threads have own semaphore.
* Teaching Assistant(TA): TA thread has own semaphore.

**Functions:**

* *void \*Student\_Activity(void \*threadID)*: students is waiting in random time (they are programming in this scenario), and go to TA for help is they need one. If TA is helping a student already, other students are waiting chairs in hallway, but if chairs are full, students go to their programming task back and wait for random time and come again.
* *void \*TA\_Activity()*: TA is sleeping until one student wakes him up. if any student wakes the TA up, TA will attend that student. If helping is finished and chairs are empty, then TA goes to sleeping back until any student comes. This process will continue repeatedly.

**Simple process flows/Algorithm:**

* Make student threads and TA thread (initialize)
* Each thread is running own thread function (student is programming, TA is sleeping)
* Student wakes the TA up after randomly time, then TA helps the arrived student (change student's semaphore to 1 and wait TA’s semaphore). but if chairs are full, students go to their programming work back and come again after random amount of time.
* If TA is done with helping, TA checks the remaining students. if there is(are) student(s) TA attends next student who is waiting on the chair and if not, TA goes to sleeping back (change TA’s semaphore to 1 and wait student's semaphore)
* Repeat 3-4 steps.

## **Results**

A screenshot of a cell phone

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## **Conclusion and Recommendation**

**Semaphore:**

Use a semaphore when thread wants to sleep till some other thread tells it to wake up. Semaphore 'down' happens in one thread (producer) and semaphore 'up' (for same semaphore) happens in another thread (consumer) e.g.: In this project we are using semaphore to wake up TA thread which is done by Student thread.

**Mutex:**

Use a mutex when thread want to execute code that should not be executed by any other thread at the same time. Mutex 'down' happens in one thread and mutex 'up' must happen in the same thread later. For example, in this project when TA thread is helping one student at the same time no other student should not interfere TA. Due that reason we are using mutex lock in that case.

## **References**

1. Operating System Concepts by Abraham-Silberschatz-9th edition.
2. https://www.geeksforgeeks.org/mutex-vs-semaphore/
3. <https://sjsu.instructure.com/courses/1313987/files/folder/unfiled>
4. <https://github.com/mingrammer/os-assign#structure>

## **Appendix: Code**

﻿#include <pthread.h>

#include <time.h>

#include <unistd.h>

#include <semaphore.h>

#include <stdlib.h>

#include <stdio.h>

//#include <windows.h>

#ifdef \_unix\_

#include <unistd.h>

#elif defined \_WIN32

//#include <windows.h>

#define sleep(x) Sleep(750 \* (x))

#endif

pthread\_t \*Students; //N threads running as Students.

pthread\_t TA; //Separate Thread for TA.

int ChairsCount = 0; //all three chairs are empty

int CurrentIndex = 0; //

//Declaration of Semaphores and Mutex Lock.

sem\_t TA\_Sleep;

sem\_t Student\_Sem;

sem\_t ChairsSem[3];

pthread\_mutex\_t ChairAccess;

//Declared Functions

void \*TA\_Activity();

void \*Student\_Activity(void \*threadID);

int main(int argc, char\* argv[])

{

int number\_of\_students; //a variable taken from the user to create student threads. Default is 5 student threads.

int id;

srand(time(NULL));

//Initializing Mutex Lock and Semaphores.

sem\_init(&TA\_Sleep, 0, 0);

sem\_init(&Student\_Sem, 0, 0);

for(id = 0; id < 3; ++id) //Chairs array of 3 semaphores.

sem\_init(&ChairsSem[id], 0, 0);

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

if(argc<2)

{

printf("Number of Students not specified. Using default (4) students.\n");

number\_of\_students = 4;

}

else

{

number\_of\_students = atoi(argv[1]);

printf("Number of Students specified. Creating (%d) threads.\n", number\_of\_students);

}

//Allocate memory for Students

Students = (pthread\_t\*) malloc(sizeof(pthread\_t)\*number\_of\_students);

//Creating TA thread and N Student threads.

pthread\_create(&TA, NULL, TA\_Activity, NULL);

for(id = 0; id < number\_of\_students; id++)

pthread\_create(&Students[id], NULL, Student\_Activity,(void\*) (long)id);

//Waiting for TA thread and N Student threads.

pthread\_join(TA, NULL);

for(id = 0; id < number\_of\_students; id++)

pthread\_join(Students[id], NULL);

//Free allocated memory

free(Students);

return 0;

}

void \*TA\_Activity()

{

while(1)

{

sem\_wait(&TA\_Sleep); //TA is currently sleeping.

printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");

printf("[TA has been awakened by a student]\n");

while(1)

{

// lock

pthread\_mutex\_lock(&ChairAccess);

if(ChairsCount == 0)

{

//if chairs are empty, break the loop.

pthread\_mutex\_unlock(&ChairAccess);

break;

}

//TA gets next student on chair.

sem\_post(&ChairsSem[CurrentIndex]);

ChairsCount--;

printf("One Student left chair. Remaining Chairs %d\n", 3 - ChairsCount);

CurrentIndex = (CurrentIndex + 1) % 3;

pthread\_mutex\_unlock(&ChairAccess);

// unlock

printf("\t\t TA is currently helping one of the student.\n");

sleep(5);

sem\_post(&Student\_Sem);

usleep(1000);

}

}

}

void \*Student\_Activity(void \*threadID)

{

int ProgrammingTime;

while(1)

{

printf("Student %ld is doing programming assignment.\n", (long)threadID);

ProgrammingTime = rand() % 10 + 1;

sleep(ProgrammingTime); //Sleep for a random time period.

printf("Student %ld needs help from the TA\n", (long)threadID);

pthread\_mutex\_lock(&ChairAccess);

int count = ChairsCount;

pthread\_mutex\_unlock(&ChairAccess);

if(count < 3) //Student tried to sit on a chair.

{

if(count == 0) //If student sits on first empty chair, wake up the TA. sem\_post(&TA\_Sleep);

else

printf("Student %ld sat on a chair waiting for the TA to finish. \n", (long)threadID);

// lock

pthread\_mutex\_lock(&ChairAccess);

int index = (CurrentIndex + ChairsCount) % 3;

ChairsCount++;

printf("Student sat on chair.Chairs Remaining: %d\n", 3 - ChairsCount);

pthread\_mutex\_unlock(&ChairAccess);

// unlock

sem\_wait(&ChairsSem[index]); //Student leaves his/her chair.

printf("\t\t Student %ld is getting help from the TA. \n", (long)threadID);

sem\_wait(&Student\_Sem); //Student waits to go next.

printf("Student %ld left TA room.\n",(long)threadID);

}

else

printf("Student %ld will return after some time. \n", (long)threadID);

//If student didn't find any chair to sit on.

}

}