**Sleeping teaching assistant**

A university computer science department has a teaching assistant (TA) that holds office hours where they provide help to undergraduate students with their programming assignments. The TA's office is rather small, and only has room for one desk with a chair and a computer. There are three chairs outside the office where students may sit and wait if the TA is currently helping another student. If there are no available chairs in the waiting area, the student shows up at a later time. When the TA has finished helping a student, she goes outside to the waiting area to see if there are waiting students. If so, she takes the next student and begins to help them. If there are no waiting students, she returns to her chair in he office and takes a nap. If a student shows up and sees the TA sleeping, they sit in her chair and wake her up.

Using POSIX mutex locks and sempahores, design a solution to the sleeping TA problem. A simple solution can use one mutex lock and two semaphores:

- the 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.)

- a semaphore for signalling the sleeping TA that a student has shown up.

- a semaphore for signalling a waiting student that the TA can now help them.

Students and the TA

Using Pthreads, begin by creating N students where each student runs as a separate thread. In addition, the TA will also run as a separate thread. Student threads will alternate between programming for a period of time, and then seeking help from the TA. If the TA is available, they will obtain help or will sit in a hallway while waiting their turn with the TA. If there are no chairs available, the student will resume programming on their own, and seek help at a later time. If a student arrives and notices the TA is sleeping, they must notify the TA using the appropriate semaphore.

If the TA is helping a student, upon finishing they must check if there are waiting students in the hallway. If so, they must help them as well. They will notify a waiting student by using the appropriate semaphore. If there are no waiting students, the TA can take a nap.

**Implementation in C**

#include <stdlib.h>

#include <stdio.h> //For Input and Output

#include <semaphore.h> //For creating semaphores

#include <pthread.h> //For Create POSIX threads.

#include <time.h> //Waiting for a random time.

#include <unistd.h> //Thread calls sleep for specified number of seconds.

//Declaration of Semaphores and Mutex Lock.

sem\_t TA\_Sleep;

sem\_t Student\_Sem;

sem\_t ChairsSem[3];

pthread\_mutex\_t ChairAccess;

//Declaration of N threads for running as Students.

pthread\_t \*Students;

//Declaration of a Separate Thread for TA.

pthread\_t TA;

//Number of chairs occupied is 0 initially.

int ChairsCount = 0;

//Chair number in which next student can wait.

int CurrentIndex = 0;

//Declaration of Functions.

void \*TA\_Activity();

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

void \*TA\_Activity()

{

while(1)

{

//TA is currently sleeping.

sem\_wait(&TA\_Sleep);

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.

//The teaching assistant can continue sleeping.

pthread\_mutex\_unlock(&ChairAccess);

break;

}

//When students are waiting.

//TA gets next student on chair by signalling.

sem\_post(&ChairsSem[CurrentIndex]);

//Decrease the chair count.

//Number of students waiting gets reduced by one.

ChairsCount--;

//Remaining chairs is total - number of chairs occupied.

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

//Update the chair number where next student can sit.

CurrentIndex = (CurrentIndex + 1) % 3;

pthread\_mutex\_unlock(&ChairAccess);

// unlock

printf("\t TA is currently helping 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 for a random time period.

sleep(ProgrammingTime);

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

pthread\_mutex\_lock(&ChairAccess);

int count = ChairsCount;

pthread\_mutex\_unlock(&ChairAccess);

//Student is trying to sit on a chair.

if(count < 3)

{

//If student sits on first empty chair, wake up the TA.

if(count == 0)

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;

//Number of chairs occupied is increased by one.

ChairsCount++;

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

pthread\_mutex\_unlock(&ChairAccess);

// unlock

//Student leaves his/her chair.

sem\_wait(&ChairsSem[index]);

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

//Student waits to go next.

sem\_wait(&Student\_Sem);

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

}

else

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

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

}

}

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

{

//a variable taken from the user to create student threads. Default is 5 student threads.

int number\_of\_students;

int id;

srand(time(NULL));

//Initializing Mutex Lock and Semaphores.

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

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

//Chairs array of 3 semaphores.

for(id = 0; id < 3; ++id)

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

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

if(argc<2)

{

printf("Number of Students not specified. Using default (5) students.\n");

number\_of\_students = 5;

}

else

{

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

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

}

//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;

}