**Teaching Assistant (TA) for assistance**

**Project submitted to the**

**SRM University – AP, Andhra Pradesh**

**Submitted in partial fulfillment of the requirement for the award of the degree**

**of**

**Bachelor of Technology**

**in**

**Computer Science and Engineering**

**School of Engineering and Sciences**

**Submitted By**

**Student Name : CH.Mohith**

**Roll.No – AP21110010611**

**Student Name : CH.Rushendra sai**

**Roll.No – AP21110010612**

**Student Name : S.Jahnavi**

**Roll.No – AP21110010613**

Under the guidance of

Siva Krishna Prasad. M



**Department of Computer Science and**   **Engineering SRM University**

**Neerukonda, Mangalagiri, Guntur**

**Andhra Pradesh – 522 240**

**[Month, Year]**

**Table Of Contents:**

**1) Aim of the project**

**2) Description of the project**

**3) Individual members roles and responsibilities**

**4) Code**

**5) explanation about the code**

**6) test cases and their outputs**

**7) conclusion**

**8) references**

**1) Aim of the project:**

The aim of the project is to simulate a scenario where students are programming and visiting a Teaching Assistant (TA) for assistance. The project aims to demonstrate synchronization and mutual exclusion concepts in a multithreaded environment.

**2) Description of the project:**

The project involves creating a simulation where students and a TA interact in a waiting room. Students program for random durations and, if there are available chairs, take a seat in the waiting room. The TA periodically checks for waiting students and helps them one by one. The program terminates after a specified number of iterations.

**3) Individual members roles and responsibilities:**

Student Name : CH.Mohith

Roll.No – AP21110010611

Contribution:35% code and PPT

Student Name : CH.Rushendra sai

Roll.No – AP21110010612

Contribution:35% code and Word Document

Student Name : S.Jahnavi

Roll.No – AP21110010613

Contribution:30% code

**4)CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#include <semaphore.h>

#include <ctype.h>

#include <unistd.h>

#include <time.h>

void\* student\_actions(void\* student\_id);

void\* ta\_actions();

#define NUM\_WAITING\_CHAIRS 3

#define DEFAULT\_NUM\_STUDENTS 5

sem\_t sem\_students;

sem\_t sem\_ta;

pthread\_mutex\_t mutex\_thread;

int waiting\_room\_chairs[NUM\_WAITING\_CHAIRS];

int number\_students\_waiting = 0;

int next\_seating\_position = 0;

int next\_teaching\_position = 0;

int ta\_sleeping\_flag = 0;

int num\_executions = 1; // Number of executions for termination

int main(int argc, char\*\* argv) {

int i;

int student\_num;

if (argc > 1) {

if (isdigit(argv[1][0])) {

student\_num = atoi(argv[1]);

} else {

printf("Invalid input. Quitting program.\n");

return 0;

}

} else {

student\_num = DEFAULT\_NUM\_STUDENTS;

}

int student\_ids[student\_num];

pthread\_t students[student\_num];

pthread\_t ta;

sem\_init(&sem\_students, 0, 0);

sem\_init(&sem\_ta, 0, 1);

// Create threads

pthread\_mutex\_init(&mutex\_thread, NULL);

pthread\_create(&ta, NULL, ta\_actions, NULL);

for (i = 0; i < student\_num; i++) {

student\_ids[i] = i + 1;

pthread\_create(&students[i], NULL, student\_actions, (void\*)&student\_ids[i]);

}

// Join threads

pthread\_join(ta, NULL);

for (i = 0; i < student\_num; i++) {

pthread\_join(students[i], NULL);

}

return 0;

}

void\* ta\_actions() {

printf("Checking for students.\n");

while (num\_executions > 0) {

// If students are waiting

if (number\_students\_waiting > 0) {

ta\_sleeping\_flag = 0;

sem\_wait(&sem\_students);

pthread\_mutex\_lock(&mutex\_thread);

int help\_time = rand() % 5;

// TA helping student.

printf("Helping a student for %d seconds. Students waiting = %d.\n", help\_time, (number\_students\_waiting - 1));

printf("Student %d receiving help.\n", waiting\_room\_chairs[next\_teaching\_position]);

waiting\_room\_chairs[next\_teaching\_position] = 0;

number\_students\_waiting--;

next\_teaching\_position = (next\_teaching\_position + 1) % NUM\_WAITING\_CHAIRS;

usleep(help\_time \* 1000000);

pthread\_mutex\_unlock(&mutex\_thread);

sem\_post(&sem\_ta);

num\_executions--;

}

// If no students are waiting

else {

if (ta\_sleeping\_flag == 0) {

printf("No students waiting. Sleeping.\n");

ta\_sleeping\_flag = 1;

}

}

}

}

void\* student\_actions(void\* student\_id) {

int id\_student = \*(int\*)student\_id;

int executions = num\_executions;

while (executions > 0) {

// If student is waiting, continue waiting

if (waiting\_room\_chairs[id\_student - 1] == id\_student) {

continue;

}

// Student is programming.

int time = rand() % 5;

printf("\tStudent %d is programming for %d seconds.\n", id\_student, time);

usleep(time \* 1000000);

pthread\_mutex\_lock(&mutex\_thread);

if (number\_students\_waiting < NUM\_WAITING\_CHAIRS) {

waiting\_room\_chairs[next\_seating\_position] = id\_student;

number\_students\_waiting++;

// Student takes a seat in the hallway.

printf("\t\tStudent %d takes a seat. Students waiting = %d.\n", id\_student, number\_students\_waiting);

next\_seating\_position = (next\_seating\_position + 1) % NUM\_WAITING\_CHAIRS;

pthread\_mutex\_unlock(&mutex\_thread);

// Wake TA if sleeping

sem\_post(&sem\_students);

sem\_wait(&sem\_ta);

} else {

pthread\_mutex\_unlock(&mutex\_thread);

// No chairs available. Student will try later.

printf("\t\t\tStudent %d will try later.\n", id\_student);

}

executions--;

}

}

**5) Explanation about the code:**

The code simulates the interactions between students and a TA. It utilizes pthreads for multithreading support and semaphores for synchronization. Students and the TA are represented as separate threads. The code implements functions for student actions, TA actions, and the main program logic.

The main function initializes variables, creates threads, and waits for them to finish. The student\_actions function represents the actions of a student, including programming and interacting with the waiting room. The ta\_actions function represents the actions of the TA, checking for waiting students and providing assistance.

**6) Test cases and their outputs:**

To provide specific test cases and their outputs, it would be helpful to have a clear understanding of the expected behavior and functionality of the program. Since the provided code is partial, it's difficult to provide comprehensive test cases and outputs. However, potential test cases may include scenarios such as:

Testing the program with different numbers of students and chairs in the waiting room.

Checking if the TA helps students in the correct order.

Testing the program termination after a specified number of iterations.

Verifying the synchronization and mutual exclusion of threads.

It is important to note that the outputs of test cases would depend on the specific implementation and modifications made to the provided code.

**7) conclusion:**

The given C code is a multi-threaded implementation of the Sleeping Teaching Assistant (TA) problem. The problem simulates a scenario where a TA helps students with their programming assignments. The TA has an office with a limited number of chairs (3 in this case) for students to wait. If all chairs are occupied, the arriving student will leave and try again later. If there are no students waiting, the TA goes to sleep. When a student arrives and finds the TA sleeping, they wake the TA up.

The code uses POSIX threads (pthreads), semaphores, and mutexes to manage concurrency and synchronization. The main function initializes the required semaphores and mutex, creates the TA and student threads, and then joins them. The TA and student threads execute their respective actions in the ta\_actions and student\_actions functions.

In conclusion, this code demonstrates a solution to the Sleeping TA problem using multi-threading, semaphores, and mutexes for synchronization and concurrency control. The program simulates the interactions between the TA and students, ensuring that the TA helps students when they are waiting and goes to sleep when no students are waiting.

**8)References:**

The Sleeping Teaching Assistant problem is a classic synchronization problem in computer science, similar to other well-known problems like the Dining Philosophers and the Producer-Consumer problem. The problem and its solutions can be found in various textbooks and online resources related to operating systems, concurrency, and multi-threading. Here are some references:

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th ed.). John Wiley & Sons. (Chapter 6: Synchronization)

Tanenbaum, A. S., & Bos, H. (2014). Modern Operating Systems (4th ed.). Pearson. (Chapter 2: Processes and Threads)

Robbins, K. A., & Robbins, S. (2003). UNIX Systems Programming: Communication, Concurrency, and Threads (2nd ed.). Prentice Hall. (Chapter 11: Threads)

Butenhof, D. R. (1997). Programming with POSIX Threads. Addison-Wesley. (Chapter 3: Synchronization)

These resources provide a comprehensive understanding of synchronization problems, including the Sleeping Teaching Assistant problem, and discuss various solutions using different synchronization primitives like semaphores, mutexes, and condition variables.