The Sleeping Teaching Assistant

Statement of the Problem and the Solution

## **(1)Overview:**

In this tutorial, we’ll explain the Sleeping Teaching Assistant.

It’s a famous inter-process communication and synchronization problem that takes place in a TA ’s office.

## **(2) Problem Definition**

the problem takes place in a TA ’s office. A university computer science department has a teaching assistant ( TA ) who helps undergraduate students with their programming assignments 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 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. 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.

- Let’s look at the characteristics of the problem:

* We need to synchronize the Teaching Assistant and the students so there wouldn’t be any race conditions. Since we have limited resources and waiting tasks, this problem has so many similarities with various queueing situations.
* students can enter the waiting hallway and should wait for whether the teacher is available or not
* If other students keep coming while the teacher is helping a student, they sit down in the waiting room
* students leave if there is no empty chair in the hallway

## **(3) Solution:**

Our solution uses three semaphores, students, which counts waiting students(excluding the students in the teacher desks, who is not waiting), teacher, the number of teachers (0 or 1) who are sleeping, waiting for students, and mutex, which is used for mutual exclusion. We also need a variable, waiting, which also counts the waiting students. The reason for having waiting is that there is no way to read the current value of a semaphore. In this solution, a student entering the hallway has to count the number of waiting students. If it is less than the number of chairs, he stays; otherwise, he leaves.

* We can use the following variables for synchronisation and mutual exclusion and 2 threads .

#define CHAIRS 5

typedef int semaphore;

semaphore customers = 0;

semaphore teacher= 0;

semaphore mutex = 1;

int waiting = 0;

void teachers(void) {

while (TRUE) {

down(&students);

down(&mutex);

waiting = waiting - 1;

up(&teachers);

up(&mutex);

Helps();

} }

void customer(void) {

down(&mutex);

if (waiting < CHAIRS) {

waiting = waiting + 1;

up(&students);

down(&teacher);

get\_Help();

else {

up(&mutex);

} }

void students(void)

{ down(&mutex);

if (waiting < CHAIRS) {

waiting = waiting + 1;

up(&students);

up(&mutex);

down(&teachers);

get\_Help();

} else { up(&mutex); } }

## **4. Conclusion:**

In this tutorial, we’ve given a brief definition of the problem and then shared the solution. It’s another important problem in concurrent programming because it applies to other types of queuing problems in computing, networks, industrial engineering, telecommunication, and traffic engineering.

5. Main methods:

1-teachingAssitant : a method simulate TAS sleeping until students arrive .TAs wake up upon a student`s request for help ,assist ,then return to sleep

2-students : this method represents students arriving at the office . they check for available ,get help from a TA ,and leave afterward.

6.threads:

In this code, the number of threads is determined by two loops: 1. **Teaching Assistant Threads:** - The first loop (for(int i = 1; i <= desks; i++)) creates threads for the teaching assistants. It creates desks number of threads, each representing a teaching assistant. 2. **Student Threads:** - The second loop (for(int i = 1; i <= num; i++)) creates threads for the students. It creates num number of threads, each representing a student seeking help. Overall, the code dynamically generates threads for each teaching assistant and each student, using the Thread class and lambda expressions to run the teachingAssistant and students methods from the AssistantProfessorOffice class concurrently. The threads are employed to simulate multiple concurrent actions happening in the assistant professor's office: teaching assistants assisting students and students seeking help simultaneously. This concurrency is achieved by running each assistant and student interaction in its own thread.