The Sleeping Teaching Assistant

Project Description:

the sleeping Teaching Assistant problem is a classic synchronization problem in operating systems. 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 .

The challenge in this scenario is to design a system where students and the TA can work concurrently but maintain certain conditions:

1. **If a student arrives and the TA is asleep, the student should wake up the TA.**
2. **If a student arrives and finds the TA awake, they should wait until the TA is available.**
3. **After the TA finishes helping a student, they should either go back to sleep if no other students are waiting or help the next student in line.**

What we have actually did:

It took us a while to figure out what to do in this project, it wasn't easy but we did it, we found a way of cooperating with one another as a team and we have done the project exactly as it was described. We made a model of the interaction between teaching assistants and students in a scenario where students seek help from the teaching assistant, who may be either available or busy assisting other students. The code uses Semaphores and Mutex locks to coordinate the behavior of multiple teaching assistants and students, ensuring proper synchronization to avoid race conditions and deadlocks. The simulation involves students arriving after a specified program time, attempting to acquire the teaching assistant's availability, and either receiving help or waiting based on the availability of the teaching assistant and the number of chairs in the waiting area.

Team members roles:

Documentation :

1. **Ahmed Zienhom**
2. **Arwa Ibraheem**

Coding :

1. **Abdelrahman Hazem**

2. **Yousef Ibraheem**
3. **Abdelrahman Ahmed**
4. **Mohamed Ahmed**
5. **Nour Ehab**

Testing:

1. **Arwa Ibraheem**
2. **Ahmed Zienhom**

Recording the video: Arwa Ibraheem

Code Documentation :

1.teaching assistant class:

```
public class TeachingAssistant implements Runnable {

    private Mutexlock wakeup;
    private Semaphore chairs;
    private Semaphore available;
    private Thread t;
    private int numberofTA;
    private int numberofchairs;
    private int helpTime=5000;


    public TeachingAssistant (Mutexlock wakeup, Semaphore chairs, Semaphore available, int numberofTA,int numberofchairs) {
        t = Thread.currentThread();
        this.wakeup = wakeup;
        wakeup = new Mutexlock(numberofTA);
        this.chairs = chairs;
        this.available = available;
        this.numberofTA = numberofTA;
        this.numberofchairs=numberofchairs;
    }

    @Override
    public void run() {
        while (true) {
            try {
                wakeup.release();
                t.sleep(helpTime);

                if (chairs.availablePermits() != numberofchairs) {
                    do {
                        t.sleep(helpTime);
                        chairs.release();
                    } while (chairs.availablePermits() != numberofchairs);
                }
            } catch (InterruptedException e) {
                continue;
            }
        }
    }
```

This class represents the Teaching Assistant and implements the Runnable interface.

It has attributes such as wakeup (mutex lock), chairs (Semaphore representing the number of chairs in the waiting area), available (Semaphore representing the availability of the TA),

numberofTA (number of TAs),

numberofchairs (number of chairs in the waiting area), and helpTime (time the TA spends helping a student).

The run method contains the logic for the TA's behavior. The TA will wake up, sleep for a specified time (helpTime), and then release waiting students from the chairs

2.student class:

```
public class Student implements Runnable {

    private int programTime;
    private int count = 0;
    private int studentNum;
    private Mutexlock wakeup;
    private Semaphore chairs;
    private Semaphore available;
    private int numberofchairs;
    private int numberofTA;
    private int helpTime=5000;
    private Thread t;

    public Student(int programTime, Mutexlock wakeup, Semaphore chairs, Semaphore available, int studentNum, int numberofchairs, int numberofTA) {
        this.programTime = programTime;
        this.wakeup = wakeup;
        wakeup = new Mutexlock(numberofTA);
        this.chairs = chairs;
        this.available = available;
        this.studentNum = studentNum;
        t = Thread.currentThread();
        this.numberofchairs = numberofchairs;
        this.numberofTA = numberofTA;
    }

    @Override
    public void run() {...۳۰ lines }
}
```

It has attributes such as **programTime** (time the student spends on a program), **wakeup** (mutex lock), **chairs** (Semaphore representing the number of chairs in the waiting area), **available** (Semaphore representing the availability of the TA), **studentNum** (unique identifier for the student), **numberofchairs** (number of chairs in the waiting area), **numberofTA** (number of TAs), and **helpTime** (time the TA spends helping a student).

- The **run** method contains the logic for the student's behavior. Students arrive after a certain program time, attempt to acquire the TA's availability, and either get help or wait based on the availability of the TA and chairs.

The run method

```
        @Override
        public void run() {
            while (true) {
                try {
                    t.sleep(programTime * 1000);
                    if (available.tryAcquire()) {
                        try {
                            wakeup.take();

                            t.sleep(helpTime);
                        } catch (InterruptedException e) {
                            continue;
                        } finally {
                            available.release();
                        }
                    } else {

                        if (chairs.tryAcquire()) {
                            try {

                                available.acquire();
                                t.sleep(helpTime);

                                available.release();
                            } catch (InterruptedException e) {
                                continue;
                            }
                        } else {
                            t.sleep(helpTime);
                        }
                    }
                } catch (InterruptedException e) {
                    break;
                }
            }
        }
    }
```

The **run** method in the provided code represents the

behavior of a student in a simulation of the Sleeping Teaching Assistant problem. Within an infinite loop, the student periodically arrives after a specified program time, attempts to acquire the availability of the teaching assistant (**tryAcquire()** on the **available** Semaphore). If successful, the student enters a critical section where it signals the teaching assistant to take action (**wakeup.take()**), sleeps for a predefined help time (**t.sleep(helpTime)**), and then releases the teaching assistant's availability. If the TA is not available, the student checks if there are available chairs in the waiting area (**chairs.tryAcquire()**). If a chair is available, the student enters the waiting area, acquires the TA's availability, receives help, and releases the availability after assistance. If no chairs are available, the student waits for a specified time before retrying. The process is repeated in the infinite loop, and if the thread is interrupted, the student gracefully exits the simulation. This design ensures the proper synchronization of students seeking assistance in a concurrent environment.

3.mutex class:

```
class Mutexlock {

    private final ReentrantLock entlock;
    private final Condition self[];
    private int num;
    private boolean signal = false;

    public Mutexlock(int taNum) {
       num=taNum;
       entlock=new ReentrantLock();
       self=new Condition[taNum];
     for(int i=0;i<num;i++){
       self[i]=entlock.newCondition();
     }
    }

    public void take() {

        entlock.lock();
        try{
            this.signal = true;
            try {
                self[num-1].signal();

            }catch(NullPointerException ex){}

        }finally {
            entlock.unlock();
        }
}


    public  void release(){...\∀ lines }

}
```

- This class implements a mutex lock using **ReentrantLock** and **Condition**.

- It has attributes such as **entlock** (ReentrantLock), **self** (an array of Conditions for signaling), **num** (number of TAs), and **signal** (a boolean indicating whether the TA is available).
- The **take** and **release** methods handle the synchronization of the TA's availability.