Project one

Sleeping Teaching assistant problem

—

Team roles

Student and Teaching assistant classes:يوسف اسامه وعمر عزت

Work class:اسلام خالد وفارس ادريس

GUI:منه الله اسامه

Documentation and video:محمود رافت و محمود شحات

Description

A university computer science department has multiple teaching assistants (TA) who help 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 per TA. There are 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.

Requirements

Using JAVA threads, mutex locks, and semaphores, implement a solution that coordinates the activities of the TA and the students. The number of disks in the TA's room (aka no. of

TAs), number of chairs for waiting students and number of students that have questions should be provided as an input to your program.

Specifications

1 Entities

- Students (Threads): Modeled as threads, representing individuals seeking assistance.
- TAs (Mutex/Semaphore): Represent the teaching assistants. A mutex or semaphore can be used to control access to the TAs.
- Chairs (Mutex/Semaphore): Modeled as a limited resource (semaphore or mutex) to control the number of students waiting.

2 Logic

- Student Entering TA's Room:
    - A student attempts to enter the TA's room to seek assistance.
- TA Availability Check:
    - If a TA is available, the student receives assistance.
- Chairs Availability Check:
    - If no TA is available, the student checks for available chairs.
    - If chairs are available, the student waits.
    - If no chairs are available, the student leaves and plans to return later.
- Repeat Logic:
    - If the student leaves, they repeat the same logic when they return.

3. GUI

3.1 Input

- # Students: Number of students seeking assistance.
- # Chairs: Number of chairs available in the waiting room.

- # TAs: Number of teaching assistants available.

3.2 Output

- # TAs Working: Real-time count of TAs currently assisting students.
- # TAs Sleeping: Real-time count of TAs currently taking a nap.
- # Students Waiting on Chairs: Real-time count of students currently waiting in chairs.
- # Students That Will Come Later: Real-time count of students who left and plan to return later.

3.3 Real-Time Updates

- The GUI should dynamically update the counts as students seek assistance, TAs become available/busy, and chairs are occupied.

Implementation

Overview of Classes

Student Class (`Student.java`):

- Represents a student seeking assistance.
- Utilizes "`MutexLock`" for synchronization.
- Implements the "`Runnable`" interface.

TeachingAssistant Class (`TeachingAssistant.java`):

- Represents a teaching assistant.
- Utilizes "`Mutexlock`", semaphores, and conditions for synchronization.
- Implements the "`Runnable`" interface.

Mutexlock Class (`Mutexlock.java`):

- Implements a custom mutex lock with conditions.
- Used for synchronization between students and teaching assistants.

Work Class(`Work.java`)

- The `Work` class serves as the graphical user interface (GUI) to configure and initiate the simulation of the Sleeping Teaching Assistants problem. It takes input parameters, such as the number of teaching assistants, students, and chairs, and provides real-time updates on the simulation's progress.

Analysis of Student Class (`Student.java`)

Attributes:

```
    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;
```

- `programTime`: Represents the time interval between attempts to seek assistance.
- `wakeup`: A `Mutexlock` instance representing the availability of teaching assistants.
- `chairs`: A semaphore representing the available waiting chairs.
- `available`: A semaphore representing the availability of teaching assistants.
- `studentNum`: Unique identifier for the student.
- `t`: The current thread instance.
- `numberofchairs`: Total number of available chairs.
- `numberofTA`: Total number of teaching assistants.
- `helpTime`: Duration for which a student receives assistance.

Constructor:

```
    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;
    }
```

Run Method:

```
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;
            }
        }
    }
}
```

Analysis of TeachingAssistant Class (`TeachingAssistant.java`)

Attributes:

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

- `wakeup`: A `Mutexlock` instance representing the availability of teaching assistants.
- `chairs`: A semaphore representing the available waiting chairs.
- `available`: A semaphore representing the availability of teaching assistants.
- `t`: The current thread instance.
- `numberofTA`: Total number of teaching assistants.
- `numberofchairs`: Total number of available chairs.
- `helpTime`: Duration for which a teaching assistant helps a student.

Constructor:

```
    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;
    }
```

Run Method:

```
@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;
            }
        }
    }
```

- Utilizes a continuous loop to model a teaching assistant's behavior.
- Releases the wakeup signal and simulates helping a student for a specified time.
- After helping the first student, it checks for students waiting in chairs if there are students it continuously helps the first on the line and frees it's chair if the chairs are empty it goes to sleep .

Analysis of Mutexlock Class (`Mutexlock.java`)

Attributes:

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

- `entlock`: A `ReentrantLock` instance used for mutual exclusion.
- `self`: An array of conditions for signaling.

Methods:

```
 public void take() {

        entlock.lock();
        try{
            this.signal = true;
            try {


                self[num-1].signal();

            }catch(NullPointerException ex){}

        }finally {
            entlock.unlock();
        }
}
```

- `take()`: Acquires the lock and signals conditions.

```
 public  void release(){
        try{
        entlock.lock();
        while(!this.signal){
            try {
                 self[num-1].await();
            }
```

```
            catch(NullPointerException ex){}
            catch (InterruptedException ex) {

Logger.getLogger(Mutexlock.class.getName()).log(Level.SEVERE, null,
ex);
            }
        }
        this.signal = false;
        } finally {
            entlock.unlock();
        }
    }
}
```

- `release()`: Releases the lock and awaits conditions.

Analysis of `Work` Class

Attributes:

```
 numberofTA = Integer.parseInt(t1.getText());
        numberofStudents = Integer.parseInt(t2.getText());
        numberofchairs = Integer.parseInt(t3.getText());

        Mutexlock wakeup = new Mutexlock(numberofTA);
        Semaphore chairs = new Semaphore(numberofchairs);
        Semaphore available = new Semaphore(numberofTA);
        Random studentWait = new Random();
```

Create threads for both students and TAs

```
        for (int i = 0; i < numberofStudents; i++) {
            Thread student = new Thread(new
Student(studentWait.nextInt(20), wakeup, chairs, available, i + 1,
numberofchairs, numberofTA));
            student.start();
```

```
        }
        for (int i = 0; i < numberofTA; i++) {

            // Create and start TA Thread.
            Thread ta = new Thread(new TeachingAssistant(wakeup,
chairs, available, numberofTA,numberofchairs));
            ta.start();
        }
```

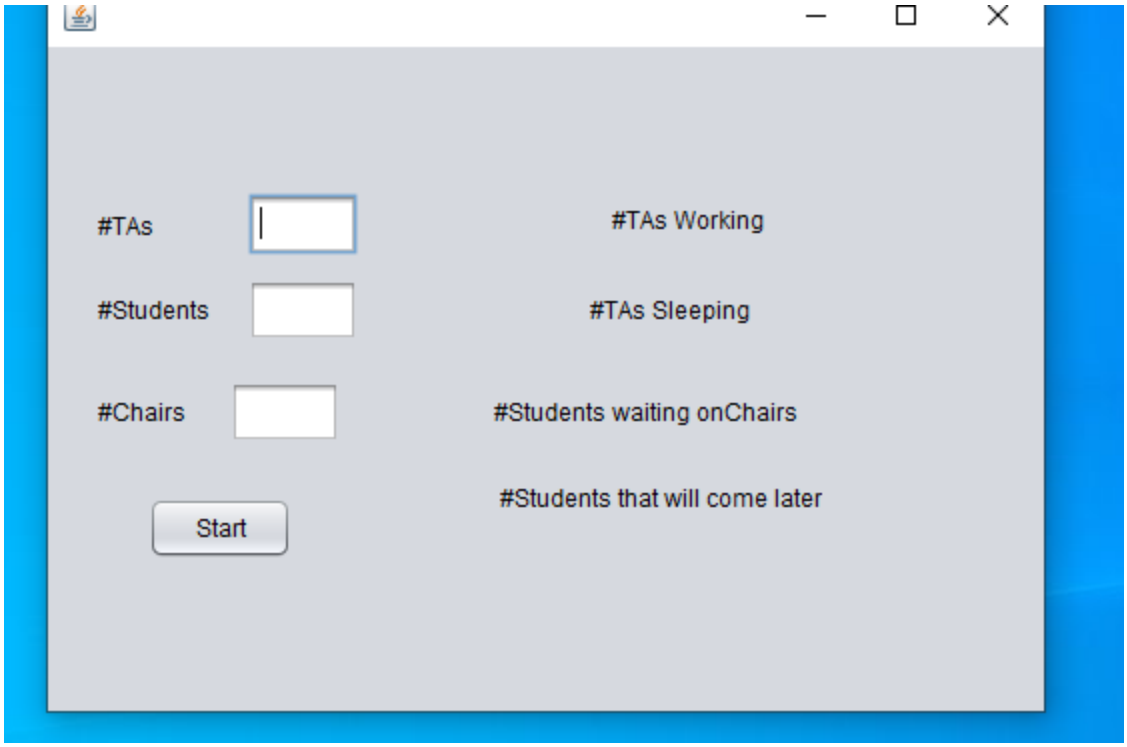
Create a third thread (`print`)

```
  Thread print = new Thread() {
            public void run() {
                while (true) {

                    try {
                        Thread.sleep(2000);
                    } catch (InterruptedException ex) {
                    }
                    jLabel5.setText(String.valueOf(numberofTA -
available.availablePermits()));

jLabel4.setText(String.valueOf(available.availablePermits()));
                    jLabel7.setText(String.valueOf(numberofchairs -
chairs.availablePermits()));
                    jLabel6.setText(String.valueOf(numberofStudents -
((numberofTA - available.availablePermits()) + (numberofchairs -
chairs.availablePermits()))));
                }
            }
        };
```

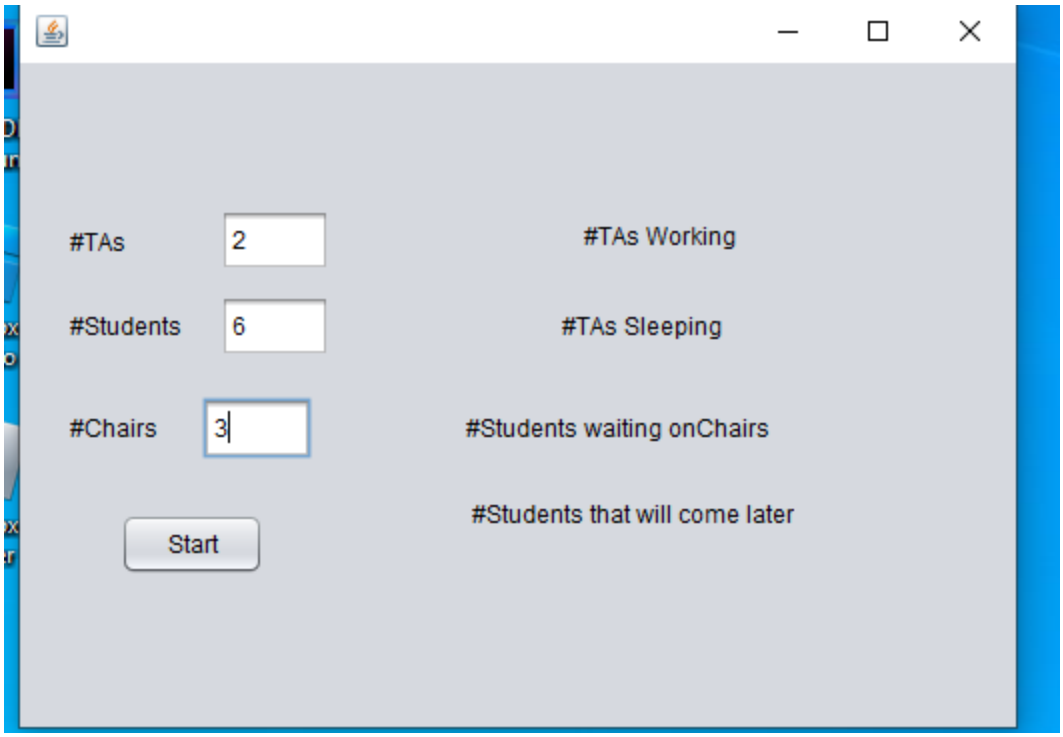
to periodically update the GUI with real-time data on the number of TAs working, TAs sleeping, students waiting on chairs, and students planning to come later.

Testing

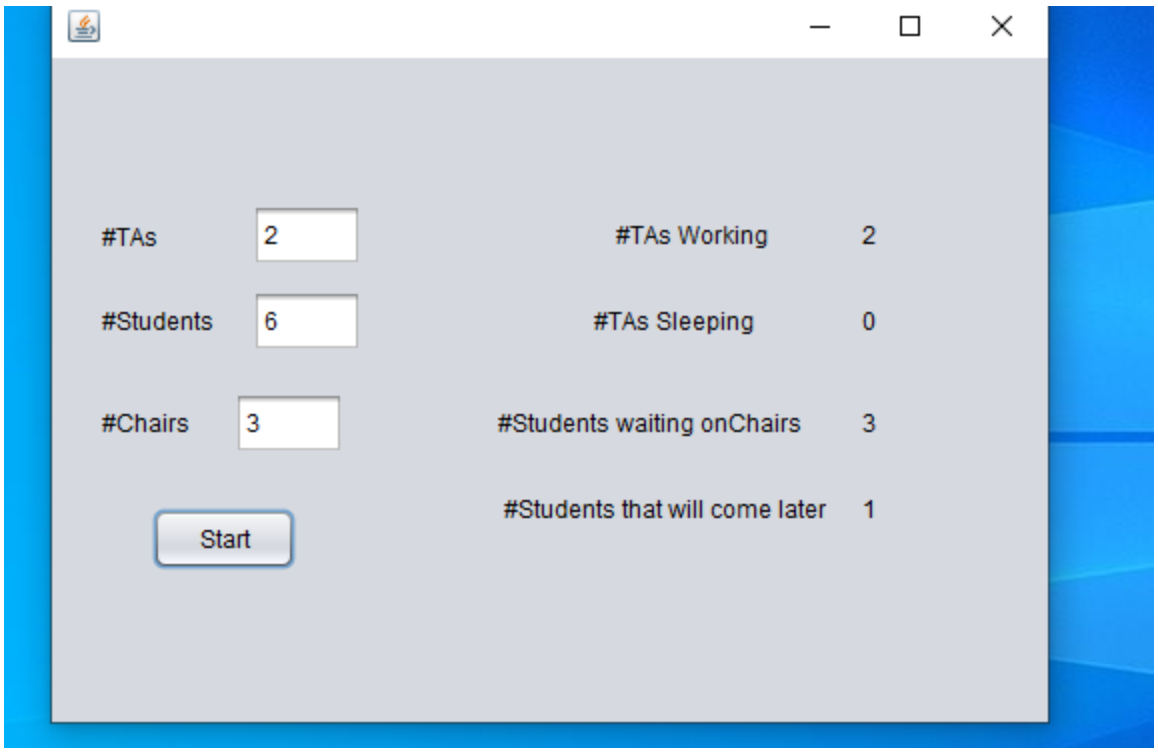
When you run the program



when we put the inputs


#TAs
2
#TAs Working
#Students
6
#TAs Sleeping
#Chairs
3
#Students waiting onChairs
#Students that will come later
Start

The outputs


#TAs
2
#TAs Working
2
#Students
6
#TAs Sleeping
0
#Chairs
3
#Students waiting onChairs
3
#Students that will come later
1
Start