Zeynep Türkmen - 29541

Programming Assignment 3

I wrote my program using C++.

Explanation of global variables:

2 mutex locks initialized.

- Cool_lock1 = atomicity of numStu and numAss
- Cool_lock2 = atomicity of awakeStu and awakeAss

3 semaphores

- enterLeave = checking the validity for entering leaving the classroom (3classAss >= classStu)
- semStudent = for sleeping/waking the students until a demo session (numStu >= 2 and numAss >=1)
- semAssistant = for sleeping/waking the assistants until a demo session (numStu >= 2 and numAss >=1)

4 counters initialized as zero.

- numStu = eligible students for a demo session (they are inside the classroom)
- numAss = eligible assistants for a demo session
- awakeStu = number of students that are a member of a demo session
- awakeAss = number of assistants that are a member of a demo session

1 barrier vector to store barriers of all demo sessions.

Below is the Pseudocode of the entire program:

```
/*
Student function:
    Create a variable called barrIdx that will represent the index of the barrier that belongs to this student's demo
session.
    Print the student wants to enter the classroom.
    Wait semaphore(enterLeave) to signal
    //either when another student leaves, or an assistant enters
    Print "entered classroom"
    Lock cool_lock to check available students for a demo
    Increment numStu
    If (numStu < 2 or numAss < 1)
```

```
        Unlock cool_lock
        Wait for student semaphore to be signaled
    Else
        Reduce the number of available students by 2 (numStu)
        Reduce the number of available assistants by 1 (numAss)
        Wake a student sem_post(student)
        Wake an assistant sem_post(assistant)
        Unlock cool_lock

    Lock cool_lock2
    Increment the awake students by 1
    Assign the barrier index acordingly to the number of awake students (they get indexes accordingly to the order
they wake up)
    Unlock cool_lock2
    Print “participating”
    Wait for barrier at barrIdx
    Print “left the classroom”
    Let another student in its place enter sem_post(enterLeave)
    Return null;

Assistant function:
    Print “entered classroom”
    Signal enterLeave semaphore 3 times to let 3 students enter.
    Lock cool_lock
    Increment numAss (available for demo)
    If (numStu < 2 or numAss < 1)
        Unlock cool_lock
        Wait for assistant semaphore to be signaled
    Else
        Reduce the number of available students by 2 (numStu)
        Reduce the number of available assistants by 1 (numAss)
        Wake 2 students x  student sem_post(student)
        Unlock cool_lock
    Lock cool_lock2
    Increment the awake assistants by 1
    Assign the barrier index acordingly to the number of awake assistants (they get indexes accordingly to the order
they wake up)
```

```
   Unlock cool_lock2
   Print “participating”
   Wait for barrier at barrIdx
   Print “demo is over”
   Wait for enterLeave semaphore 3 times, either students leave or an assistant enters or its already available
   Print “left the classroom”
   Return null.

In main:
   Create a vector of thread identifiers
   Assign console arguments to numA and numS
   Print “Works with all conditions”
   Initialize all semaphores with initial value zero
   Check arguments validity
   If more than 2 inputs
      Print “only enter 2 arguments”
   Else
      If numA < 0
         Print “numA should be positive”
      Else if numA*2 != numStu
         Print “num stu should be double of ass”
      Else
         Create barriers of size 3 for all demo sessions
         Push them to the barrier veector
         Create numA amount of asssistant threads
         Create numS amount of student threads
         Push all threads to thread vector

   Join all threads to the main
   Clear thread vector
   Destroy all barriers
   Clear barrier vector
   Destroy the semaphores
   Print “main terminates”
   Terminate the main
*/
```

Synchronization points and the explanations:

Entering/leaving the classroom:

- I used a semaphore with initial value of zero for this point. (enterLeave)
- The reason => Initally theres no available resource until an assistant enters because the condition states that numStu <= 3*numAss should hold at all times. So, a student cannot enter until theres at least 1 assistant in. If he does, condition will become 0>=1.
- So, when a student first tries to enter the classroom, s/he first waits for a resource to be available using sem_wait.
- When student does leave, she uses sem_post to give up the resource she has.
- When an assistant enters the classroom, it means 3 resources are open for 3 student to grab. They can pass through the sem_wait part.
- When an assistant is leaving, he needs to make sure to close 3 resources so that the condition will hold when he leaves. So it calls sem_wait 3 times before leaving to make sure it reduces the available resources before he leaves.
- As a result the resource increases by 3 when assistant enters and by 1 when a student leaves. Similarly, it decreases by 3 when assistant leaves and by 1 when a student gets in. This makes sure the condition numStu <= 3*numAss (as stated on page 1) is always true.

Joining a demo session:

- I used 2 semaphores for this point. First to wake up/sleep the assistants semAssistant. The other is to wake up/sleep the students.
- They both start with the initial value of zero. As theres no resource available at first and they need to sleep until woken up by a valid demo session.
- I also used counter variables that denote the number of students/assistants eligible to do a demo session. (numStu and numAss) (Meaning: they are inside the classroom and they are not currently assigned to a demo session)
- To make sure the increments/decrements on these counters are atomic, I created a mutex cool_lock.
- Student:

    If student is inside the classroom:

    It first obtains the lock and increments the numStu, it then checks if there are enough people to initiate a demo session (numStu >= 2 && numAss >=1).

    If not, it gives up the lock and calls sem_wait(semStudent).

    Else, It proceeds to initiate a demo session, it wakes up an assistant and a student and decrements the eligible students by 2 and assistant by 1. Then it gives up the lock.

- Assistant:

  It first obtains the lock and increments the numAss, it then checks if there are enough people to initiate a demo session (numStu >= 2 && numAss >=1).

  If not, it gives up the lock and calls sem_wait(semAssistant).

  Else, It proceeds to initiate a demo session, it wakes up an 2 students and decrements the eligible students by 2 and assistant by 1. Then it gives up the lock.

- The demo part only gets initiated when there are enough people to do so, unless that condition holds, all threads will get to sleep until there are enough people to do a demo session with. Since the numAss*2 = numStu as given by the inputs, all threads will be waken up eventually. (Meaning = for a demo session 2 out of 3 members sleep and the third member initiates by waking the sleeping 2, this means all threads will be waken up by their demo partners.)
- As a result this condition will always hold.

Waiting for other people in your own demo session:

- For this point I used a pthread barrier for each demo session and set their initial values as 3. All of these barriers are stored inside a vector called cool_barriers
- In order to make sure each demo session waits for their own barrier I created something like an indexing system.
- For that I added 2 variables awakeStu and awakeAss. Of course, to keep the increments atomic I made another mutex called cool_lock2. (since it's not related with other counters it's better for it to have its own lock for efficiency)
- When a demo session is about to begin, each thread increments their awakeStu and awakeAss accordingly. Accordingly, to the current number of this global variable, they achieve a barrier given by the order of incrementation. With my implementation, only 2 students and an assistant can get the same index. This makes sure all 3 people (group of demo session) waits for one another. Each demo session has its own index and they are all unique as the awakeAss and awakeStu is always incremented. Since the barrIdx is kept as a local variable for each thread, and the assignment operations are done inside the mutex, it will always work.
- After that point they simply wait for their assigned barriers before proceeding, hence this condition will hold.

As a result, the code is free of deadlocks, and it works with all conditions ☺. (though my report took me longer than implementing my solution..)