Queens College Brian Aguilar

CSCI340 6/22/2019

Project 2- Pseudo-Code Implementation

I will start off by explaining my initial implementation on how I thought the student and the teacher should be implemented. I labeled each action that would require semaphores (the extensive pseudo-code implementation will be shown for those actions that require semaphores), but the implementation of the semaphores will be shown in the next few pages. If you would rather see the semaphore implementation, then skip the first few pages. **I WILL HOWEVER WILL BE REFERRING TO THIS CODE TEMPLATE WHEN I TALK ABOUT THE SEMAPHORES!!**

The two threads I will be implementing will be a student thread and a teacher thread. The teacher thread however will be control by a timer which is a separate thread itself, but for the sakes of keeping this project pseudo-code like, timer implementation is implemented in the teacher thread and each timer action would be in a separate thread itself.

```
Student i () {
//Requires Semaphore
while(!labOpen(){}
// Requires Semaphore
currentStudents++
  if (currentStudents != capacity) {
        if (#questions_A != 0) {
              askTypeAQuestions(); //simulated by sleep
        }
        if (#questions_B != 0) {
              // Requires Semaphore
              askTypeBQuestions();
        }
        // Requires Semaphore
        browseInternet()
  } else {
     Student does not enter in lab
  }
}
```

Variables: currentStudents = 0;

The student thread is simple. For every student thread that is created, they will execute this following code. We start off by checking if the current number of students in the lab is not equal to the lab capacity. If the number of students is equal to the lab capacity (if the lab is full), then we don't execute any of the following actions that a student would do in the lab and it should terminate (aka not enter in the lab). If the lab is not full however, the student should be able to enter in lab and we increase the number of students whenever the student enters in the lab. After that, we perform the following scenarios. If the student has any type A questions, they will ask a type A question and that send that type A question to the professors email inbox (stored in the teacher object and will be placed in some sort of queue). This will be simulated by a sleep. If they have a type B question, the student will enter in the chat session and will wait until it's their turn to chat with the teacher. If they get into the chat session, they will start chatting with the professor (extensive implementation will be shown in later pages). Once they are done chatting with the professor, they browse the internet and wait until the professor closes his online office hours. All students should be browsing the internet regardless if they have chatted with the teacher or not. Once the professor has closed his online office hours, all of the students leave.

The teacher thread is a little bit more complex because the teacher's actions will be based on the time interval that the teacher is currently in. We need a way to simulate the teacher's schedule. To simulate the teacher's schedule, we will simulate some sort of timer thread that will perform the teacher's actions if he is in that time interval. We will implement this by having a counter and if the counter hits a certain value, we will perform that teacher's action

e.g,) If the teacher is supposed to be in the chat session between time interval 5 and 10 and the counter value is at 7, it will chat with the next student that is in the session

Let's determine our fixed time interval as the following

Variables:

counter =0

arrivalToOffice = 4

startOfficeHours = 8 // he will also start the chat session when he starts his office hours

endOnlineChatSession = 14 // he will get to 65% of the students

endOfficeHours = 17 //giving the professor the opportunity to answer a couple of emails (type A questions)

```
Teacher() {
        while(counter!=arrivalToOffice)
                do not perform any actions
                 //sleep
        Teacher arrives to his office
        while(counter!=startOfficeHours){
```

```
            answerTypeAQuestions();
            //sleep
        }
        Teacher starts his office hours
        while(counter!=endOnlineChatSession){
            chatWithStudent()
            //sleep
        }
        Teacher ends the online chat session
        while(counter!=endOfficeHours) {
            answerTypeAQuestions();
            //sleep
        }
        Teacher leaves the office
}
```

If the teacher has not arrived at the office, we will simulate that by doing a sleep a couple of times before the professor has arrived. Once the professor arrives, he will be answering some type A questions if he has any in his inbox. He will only perform this action in between the time interval of 4 and 8 (Initially answer only 4 questions. We can make this longer if its required). In between time interval 8 to 14, we will simulate the professor being in the chat session. At time 8, we simulate the teacher opening up his office hours and him starting up the chat session. He will only chat with a student if there are any students waiting to chat with the professor. The professor then chats with the first student that is on line. They will chat with that student and then will chat with the next student in line (if there is any), until the online chat session is up. After the online chat session is over, he will answer any remaining type A questions (but not all) and then he will leave the office. When he leaves the office (ends the office hours) all of the students should be kicked out of the lab.

We initially begin implementing the semaphore while the students are waiting for the lab to open. If the lab hasn't opened yet, we don't want the students to start their processes or enter in the lab. **This can be implemented as a counting semaphore. Let's call this counting semaphore Students and let's set it equal to 0.** The way that this is going to work is for every student process we create, we will block it by the semaphore. It isn't until all of the students have started up and the teacher has started up that we will release all of the student threads, so they can enter in the lab. The following implementation will look something like this

Main(){ Initialize all student threads and start them Initial the teacher thread and start it **for(int i=0;i<#numOfStudents;i++)V(Student)**	Student i () { **P(Student)** …. …. ….

This semaphore will work because when we initially start all of the student threads, they will reach the block statement before the main thread can release all of the students and even if the main thread executes all of the releases before the student threads hit the block statement (P(student)), this indicates that the lab is already open, therefore allowing any student to pass by.

We then implement more semaphores when we increase the number of students. If we don't implement this line of code in a mutually exclusive way, there will be a data coherence problem and it will be possible for the lab to be full, but students can still enter in. **We will fix this by implementing a binary semaphore MUTEX where the value of MUTEX will equal 1.**

Student i () { **…..** **// Requires Semaphore** **currentStudents++** if (currentStudents != capacity) { **…..** **…..** **…..**	Student i () { **…..** **P(MUTEX)** **currentStudents++** **V(MUTEX)** if (currentStudents != capacity) { **…..** **…..** **…..**

With having this semaphore being implemented, we are guaranteed no data coherence problem, and this makes sure that we only have no more than 8 students in the lab. The good thing about this implementation too, let's say students 1-12 have arrived at the lab, but the lab hasn't opened yet. As soon as the lab opens, any of 8 of the 12 can come in (e.g. Student 1 could be the first one to arrive to the lab, but this student could be the 5th student to enter into the lab).

The next implementation of semaphores occurs when the students are waiting to chat with the professor and in the execution of the chat session. The reason being is students will be waiting to chat with the teacher, and if a student is chatting with the teacher, the teacher needs to wait for the student to ask a question and then the student needs to wait for the teacher to answer their question. We also need to block any other students from entering in to the chat session. These actions must be implemented in a synchronized way.

Below shows initial pseudo-code for askTypeBQuestions() [comes from student thread] and chatWithStudent() [comes from teacher thread]

```
askTypeBQuestions() {
  arrive to the chat session
  waits until its their turn to chat with the teacher
  if (online chat session is not over) {
     while(#typeBQuestions != asked) {
          ask type B question
          asked++
          wait for the teacher to answer
          get answer
     }
  } else {
     Leave the chat session due to not being able to
chat with the teacher
}
```

```
chatWithStudent() {
  if(there are students waiting to chat with the
professor) {
    Chat with the first student in line
    while(#typeBQuestions != received){
       Wait for student to ask a question
       Get Question
        received++
       Answer student's question //sleep
    }
}
```

Variables: #typeBQuestions, asked = 0, received =0

These two processes between the student and the teacher will be running concurrently whenever a student is chatting with the professor. The lines that will be simulate by a sleep will be asking a type B question from a student and the teacher answering the type B questions. The main issue with this part is keeping the student and the teacher in sync. We want to simulate the order where the student asks the question first, and then the teacher will answer their question. We never want the student to interrupt the teacher while answering the question and this is where the use of semaphores will come into play. The way that we can simulate this is by implementing two binary semaphores. The first binary semaphore will be for the student asking the question. **Let Asked be a binary semaphore with its initial value set to 0. The second binary semaphore will be for the teacher answering the question. Let Answer be a binary semaphore with its initial value set to 0. We will also be using variables asked which is a counter to keep track of the number of questions that the student has asked (implemented only in the student) and will be implementing another counter in the teacher called received to keep track of the number of questions the teacher has gotten (BOTH VARIABLES ARE NOT SHARED).** The other issue that we have to take care of is for all of the other students that are waiting to chat with the professor. If two students arrived to the chat and student 2 gets to chat with the professor first, we need to block student one from entering the chat and vice versa. We will take care of that using a counter and a binary semaphore. **Let ChatLine be a counting semaphore that is set equal 1. Let counter be a shared variable among all student threads and lets set that equal to 0. We will also be using the binary semaphore MUTEX from before and we know the value will be 1.** With these implementations, we get the following code

```
askTypeBQuestions() {
  arrive to the chat session
  P(Mutex)
  counter++
  V(Mutex)
   P(ChatLine)
  if(the online chat session is not over){
     while(#typeBQuestions != asked) {
          ask type B question
```

```
chatWithStudent() {
  if(there are students waiting to chat with the
professor) {
    Chat with the first student in line
    while(#typeBQuestions != received){
       Wait for student to ask a question
        P(Asked)
       Get Question
        received++
```

```
        asked++
        V(Asked)
        P(Answer)
        wait for the teacher to answer
        get answer
     }
    counter--;
    V(ChatLine)
  } else {
     //Leave the chat session because
    // the chat session is now over
   if(online chat session is over) {
    for(int i=0;i<counter;i++)V(ChatLine)
}
```

```
      Answer student's question //sleep
      V(Answer)
    }
}
```

For the students that are entering the chat session is going to work as follows. The reason why ChatLine is set to one is because we want to allow the first student that enters into the chat session and block all others. Before, we block the student we make sure we keep track of the number of students that are in the chat session. The student will go through the chat session and as soon as they are done, we decrease the counter by one and release the next student that is waiting to chat with the professor. The reason why we keep the counter is because if the chat session is over, but there are still students blocked in the chat line, we have to release all of them, so they can browse the internet and we only release the remaining students that are in the chat line. So, every time the student leaves the chat session, we check if the chat session is over and if it is, then that student process will release all of the other students that were waiting to chat with the professor and then since the chat session is over, they will reach the else statement and not execute the code that a student would normally do when chatting with the professor. There will be a Boolean variable that controls that, and it will be set to true after the teacher is done chatting with all of his students.

For the chat session, the student will ask a question. As soon as the student asks a question, they notify the professor that they have asked a question (hence V(Asked)). After they have asked a question, they will be blocked by P(Answer) because they need to wait until the professor has answered their question before they move on and ask another question. We transition to the professor then who waits for the student to ask the question. If they already asked a question before the teacher gets to P(Asked), the teacher will automatically continue its sequence. Otherwise they will be blocked until the student asks a question. The teacher will then be notified that they have gotten the question and will begin answering it which is simulated by a sleep. As soon as they finish answering the question, they will notify the student that they have answered the question and will wait for the student to ask the next question.

The reason why the semaphore values are set to 0 is because we want to control some sort of order. The first possible sequence that can happen is the teacher trying to answer the question first while they haven't gotten the question, but because Asked = 0 in the beginning. It will be blocked, and it won't continue until student signals (V(Asked)) the professor that they have answered a question before the teacher moves on.

The last semaphore that we need to implement is when all of the students are browsing the internet. They browse the internet until the teacher leaves the office and as soon as the professor leaves the office all of the students leave the lab. We will implement this by using a counting semaphore. **Let Internet be a counting semaphore with an initial value of 0.** If we implement this in the teacher and the student, this is the following code that we will get

// IMPLEMENTED IN THE STUDENT THREAD browseInternet() { browse the internet **P(Internet)** Leave the lab }	Teacher () { … Teacher leaves office for(int i=0;i<#numOfStudents;i++)**V(Internet)** }

For every student that enters browseInternet() will be automatically blocked until the teacher has finished is tasks. We know that all the student threads will arrive here before the teacher is because by the time the online chat session is over, all of the students will be browsing the internet and there is still some leeway because the teacher has a little bit more time to answer type A questions before he leaves. Once the teacher leaves, he releases all of the students and there will be a small algorithm implemented to make all of the student threads leave in decreasing order.