Homework 3 Wet

**Due Date:** 31/12/2017, 23:30

Teaching assistant in charge:
- **Idan Yaniv**

**Important:** the Q&A for the exercise will take place at a public forum Piazza only. Critical updates about the HW will be published in pinned notes in the piazza forum. These notes are mandatory and it is your responsibility to be updated. A number of guidelines to use the forum:
- Read previous Q&A carefully before asking the question; repeated questions will probably go without answers
- Be polite, remember that course staff does this as a service for the students
- You’re not allowed to post any kind of solution and/or source code in the forum as a hint for other students; In case you feel that you have to discuss such a matter, please come to the reception hour
- When posting questions regarding hw3, put them in the hw3 folder

Only Arie, the TA in charge, can authorize postponements. In case you need a postponement, contact him directly.
Part I - Barrier

Introduction

An N-process barrier is a synchronization mechanism that allows N threads, where N is a fixed number, to wait until all of them have reached a certain point. Once all threads have reached this certain point (the barrier), they may all continue. For example, we can use barriers to synchronize N people who watch a movie, if each person runs the following thread:

```cpp
buy_popcorn();
buy_something_to_drink();
barrier.wait();   // wait for everyone else before we start the movie
watch_the_movie();
barrier.wait();   // wait until everyone leaves the cinema
close_the_door();
```

In the above example, each thread that calls barrier.wait() blocks until the last thread enters the synchronization barrier, and only then all threads are released. Note that the same barrier is used twice to synchronize all threads, so once a thread unblocks, it can re-enter the barrier and block again.

Implementation Requirements

Implement the following Barrier API, as defined in the Barrier.h header file:

```cpp
Barrier(unsigned int num_of_threads);
Constructor.
Parameters:
(1) num_of_threads - the number of threads that waits on the barrier.
Return value: none.
```

```cpp
void wait();
Block and wait until N threads reach this point.
Parameters: none.
Return value: none.
```

```cpp
~Barrier();
Destructor.
Parameters: none.
Return value: none.
```
Important Notes and Tips

- Don’t modify the signatures of the methods defined in the Barrier.h header file. We will test your code according to this interface.
- You may add any class members you need (integers, boolean, data structures, …), but your implementation should use only one synchronization primitive: semaphores. To put it another way, you are not allowed to use any of the following types and their associated functions: pthread_mutex_t, pthread_cond_t, and obviously not pthread_barrier_t.
- You can use as many semaphores as you need. Hint: remember that you can implement a simple lock with a semaphore.
- In your implementation, you can ignore any errors that the semaphore functions may return. For instance, according to the man pages, sem_init() returns the value EINVAL when the initial value of the semaphore exceeds SEM_VALUE_MAX; we will not check such cases in our tests.
- Use only standard POSIX threads and synchronization functions. You are not allowed, of course, to use the C++11 thread support or the C++11 atomic operations libraries.
- You should try to make your implementation as concurrent and as efficient as possible. The performance of your solution can affect your grade. However, efficiency must not come at the cost of correctness!
- Make sure your code is running on csl3 rather than your virtual machine. We will check your code only on csl3.
- csl3 may not be reachable outside the Technion network; but you connect to csl2 from any network and then ssh to csl3.

Part II - Reception Hour

Introduction

As you probably know, each TA in the operating systems course holds a weekly reception hour. The TA sleeps as long as there are no students in his office. When a student arrives he wakes up the TA and starts asking a question. To synchronize between the many students that may arrive simultaneously, the following must hold:

1. Only one student asks a question at any given moment.
2. After asking a question, the student waits for the TA’s answer. The student leaves only when the TA finished answering.
3. A student may only ask a question after the TA finished answering the previous student’s question (or the student was the first to arrive at the reception hour).
4. The TA’s office is small and contains only N (will be given as a parameter) seats. If a student arrives and can’t find a free seat, he leaves immediately.
(5) At a certain point, the TA decides to finish the reception hour by closing the door. After closing the door, the TA finishes answering the students who are left in his room. Students that arrive when the door is closed leave immediately.

Implementation Requirements

Implement the following ReceptionHour API, as defined in the ReceptionHour.h header file:

```cpp
ReceptionHour(unsigned int max_waiting_students);

Constructor. Starts a new thread for the TA, which should call the following member methods:
- waitForStudent()
- waitForQuestion()
- giveAnswer()

Parameters:
(1) max_waiting_students - the number of seats in the room of the TA.
Return value: none.

~ReceptionHour();

Destructor.
Parameters: none.
Return value: none.

void startStudent(unsigned int id);

Starts a new student thread with the given id. The student thread should call the following member methods:
- waitForTeacher()
- askQuestion()
- waitForAnswer()

Parameters:
(1) id - the student id.
Return value: none.

StudentStatus finishStudent(unsigned int id);

Finishes the student thread with the given id.
Parameters:
(1) id - the student id.
Return value:
- ENTERED if the student entered the reception hour and asked a question.
- LEFT_Because_NO_SEAT if the student left because there was no room,
- LEFT_Because_DOOR_CLOSED if the student left because he arrived after the reception hour was over.
```
void closeTheDoor();
Closes the door. After this function returns, new arriving students will not be able to enter.
Parameters: none.
Return value: none.

bool waitForStudent();
Makes the TA wait for a student. The students are not ordered in a queue, so the operating system will choose the next student according to the thread scheduling.
Parameters: none.
Return value:
  ● True if a new student has arrived.
  ● False if there are no more students and the door has closed.

void waitForQuestion();
Makes the TA wait for a question. Returns only after a student thread called askQuestion().
Parameters: none.
Return value: none.

void giveAnswer();
Answer a student question.
Parameters: none.
Return value: none.

StudentStatus waitForTeacher();
Makes the student wait for the TA.
Parameters: none.
Return value:
  ● ENTERED if the student entered the reception hour and asked a question.
  ● LEFT_Because_NO_SEAT if the student left because there was no room,
  ● LEFT_BECAUSE_DOOR_CLOSED if the student left because he arrived after the reception hour was over.

void askQuestion();
Ask the TA a question.
Parameters: none.
Return value: none.

void waitForAnswer();
Makes the student wait for an answer. Returns only after the TA thread called giveAnswer().
Parameters: none.
Return value: none.
Important Notes and Tips

- Don’t modify the signatures of the methods defined in the ReceptionHour.h header file. We will test your code according to this interface.
- You may add any class members you need (integers, boolean, data structures, …), but your implementation should use only mutexes and condition variables as synchronization primitives. To put it another way, you are not allowed to use semaphores or any synchronization primitives (besides pthread_mutex_t and pthread_cond_t) that are provided in the pthreads library.
- You can use as many mutexes and condition variables as you need.
- In your implementation, you can ignore any errors that the pthreads functions may return; we will not check such cases in our tests.
- Use only standard POSIX threads and synchronization functions. You are not allowed, of course, to use the C++11 thread support or the C++11 atomic operations libraries.
- You should try to make your implementation as concurrent and as efficient as possible. The performance of your solution can affect your grade. However, efficiency must not come at the cost of correctness!
- You can use any of the C++11 STL containers library. We will build your code with “g++ -std=c++11”, so you should too.
- Make sure your code is running on csl3 rather than your virtual machine. We will check your code only on csl3. [csl3 default compiler is GCC 5.5, so it fully supports C++11.]
- csl3 may not be reachable outside the Technion network; but you connect to csl2 from any network and then ssh to csl3.
Submission

- You should edit and submit all source files (including headers) that were provided in the course website under the “HW3 Wet” section.
- You should electronically submit a zip file that contains the above files.
- We only require you to submit the implementation of your classes, but not any test programs that use these classes. Of course, this does not mean you will not need a test program for debugging!
- You should not submit a printed version of the source code. However, you should document your source code!
- A file named submitters.txt which includes the ID, name and email of the participating students. The following format should be used:

  Linus Torvalds linus@gmail.com 234567890
  Ken Thompson ken@belllabs.com 345678901

Important Note: Make the outlined zip structure exactly. In particular, the zip should contain only the following files (no subdirectories):

```
zipfile
  |  
  +- Barrier.h
  +- Barrier.cxx
  +- ReceptionHour.h
  +- ReceptionHour.cxx
  +- submitters.txt
```

If you missed a file and because of this, the exercise is not working, you will get 0 and resubmission will cost 10 points. In case you missed an important file (such as the file with all your logic) we may not accept it at all. In order to prevent it you should open the zip file in a new directory and try to build and test your code in the new directory, to see that it behaves as expected.

Important Note: when you submit, retain your confirmation code and a copy of the file(s), in case of technical failure. Your confirmation code is the only valid proof that you submitted your assignment when you did.

Have a Successful Journey,

The course staff