Programming Assignment #2: Elevator Simulation

Due: see WebCT for the deadline (TA in-charge: shitai.liu@cs.mcgill.ca)

Why this assignment?

For solving certain problems, concurrency needs to be exploited. Multi-threaded programming is a well-known approach for handling concurrent computations in a computer program. While multi-threaded programming makes concurrent programs easier to understand, they can also introduce hard to detect timing related errors. The standard approach for handling timing related problems is to use synchronization primitives to control concurrent computations.

What is required as part of this assignment?

In this assignment, you will develop an elevator simulator. The elevator simulator is a multi-threaded program to simulate the events that happen as people use elevators to move between the floors of a tall building with *N* floors. Consider one elevator for the whole building. Initially, the elevator is parked in the first floor (at the start of the simulation). In the simulator, threads represent people such that one thread represents a person. A person goes through two different activities: working and taking the elevator to reach another floor. Once the person reaches the other floor, he/she is going to work there for a certain amount of time and then take the elevator again to another floor. Requests for the elevator come continuously from the people. Because each person works for a random amount of time before requesting the elevator, not all riders will show up at the same time. At the beginning of the simulation, you can also assume that the riders arrive into the building at different times.

The elevator itself uses a simple sweeping algorithm. When the first request comes in, it starts moving to service that request. After all the requests are serviced (no pending requests), the elevator is parked at the last serviced floor. You can assume that the elevator is strong enough to carry large number of passengers (no upper limit on the capacity). The elevator takes a clock period to move from a floor to another. It takes *2N* clock tics for an elevator car to make a full sweep of the building. The elevator may not continue its sweep in one direction if it finds that all the pending requests are in the other direction. That is, the elevator that is moving upwards might change direction and move downwards when it finds that there are no pending requests for higher floors.

In this simulation, you are using threads. Certain updates are local to a thread (i.e., the variables are updated by a single thread). You don't need to use synchronization primitives to protect updates (that are only performed by a single thread) that are purely local to a thread. Any time multiple threads are bound to update a variable, then that variable needs to be protected by synchronization primitive. In this assignment, you are required to use a monitor. In C, POSIX threads do not provide an explicit monitor construct. As discussed in the class, it is easy for you to construct a monitor like structure using POSIX Pthread synchronization primitives. Although the elevator problem can be solved in alternative ways, we request you to do it this way so it is

easy for us to grade the assignment. Correctness of a multi-threaded program is hard to establish and the TA has limited amount of time to evaluate your assignment!

Objectives in detail and proposed approach

Here is a suggested plan. You could deviate from the plan. However, it is important that you provide equivalent functionality in the approach you choose to implement (that is not simplify the requirements any further).

Implement a thread for the clock. The clock is responsible for maintaining time, which is a global variable that is being advanced by this thread. Note that this global variable in *not* modified by more than one thread. In addition this thread should also drive the elevator. That is the clock thread calls a clock input of the elevator to let the elevator know about the passing of the time. You could also use a similar approach to let the people know about the time passage or may be use a simpler approach. It should be correct (in terms of synchronization). The clock thread sleeps for a while and then advances the clock and repeats the process.

Implement a thread for a person. There are alternatives. However, this seems like the simplest solution that is easier to understand. The thread would undergo different activities. For a random amount of time a person would work and then decide to take the elevator to a random floor. How do the person thread know about the passage of time? You decide on a solution. A simple solution that does not cause synchronization issues will be acceptable.

Implement a monitor like structure to manage the elevator. You will use POSIX Pthread_mutex_lock and Pthread_cond_wait for implementing the monitor like structure. It is important to note that you may not be able to use the cond_wait supplied as part of the POSIX pthreads directly. The pseudo code shown below for the monitor that implements the elevator uses a prioritized wait. Many threads can be waiting on such a prioritized wait and the releasing order depends on the priority value used at waiting.

```
monitor elevator {
     int direction=1, up=1, down=-1, position=1, busy=0;
     condition upsweep, downsweep;
     request(int dest) {
          if (busy) {
               if ((position < dest) ||
                    ((position==dest) && (direction==up)))
                         upsweep.wait(dest);
               else downsweep.wait(-dest);
          }
          busy = 1;
          position = dest;
}
```

```
release() {
     busy = 0;
     if (direction==up) {
          if (!empty(upsweep)) upsweep.signal;
          else {
               direction = down;
               downsweep.signal;
          }}
     else if (!empty(downsweep)) downsweep.signal;
          else {
               direction = up;
               upsweep.signal;
          }
}
```

The above pseudo-code is *intentionally* ***incomplete***. Very easily you will realize that the above pseudo-code does not handle clock advances. The pseudo-code also uses prioritized waits. Use this pseudo-code as a starting point for your solution.

Your simulator should take number of people and number of floors as the arguments. It should print output at three different levels. In the most detailed level (which can be used as diagnostics) it should show what is happening at each clock tic. Your output should have the following information: requests from the people, who is riding the elevator and who got out. In the next level, provide entry and exit from the elevator. In the third level, you provide aggregated data such as people transported for 100 clock tics. You can also pass this parameter as an argument to the program.

In the second part of the assignment, you modify the simulator so that the maximum capacity of the elevator can be enforced. In the first part, you assumed that the elevator could carry unlimited number of passengers. You need to limit it. The capacity is passed in as an input to the simulator. Limiting the capacity can introduce new synchronization requirements to the elevator. You need to solve them using appropriate synchronization schemes.

What to Hand in

Submit the following files separately:

1. A README file with:
    1. Your name and McGill ID number
    2. If you have not fully implemented all, then list the parts that work so that you can be sure to receive credit for the parts you do have working. Indicate any issues you ran into doing this assignment. If you point out an error that you know occurs in your problem, it may lead the TA to give you more partial credit.

2. All source files needed to compile, run and test your code. If multiple source files are present, provide a Makefile for compiling them. Do not submit object or executable files.
3. Output from your testing of your program.

Suggestions

1. Make sure your source code could compile in Linux. Test them on the Trottier lab machines.

Part of the grading scheme you might want to know (to keep the TA happy):

1. 0 points if it does not compile.
2. A README text file must come with the assignment, especially if it does not implement all the features or has some known errors. (In this file, you can tell the TA about any issues you ran into doing this assignment. If you point out an error that you know occurs in your problem, it may lead the TA to give you more partial credit. On the other hand, it also may lead the TA to notice something that otherwise he or she would not.). Otherwise, the highest mark you can get is half of the total.
3. Name your parameters and functions properly by following one of the naming rules, such as Camel, Pascal, and even Hungarian notation (not recommended), to make your program clear enough to readers. (It could be you in 6 months later.). Otherwise, the highest mark you can get is half of the total.
4. Proper comments should be written in your program, unless your program is highly well-organized. Otherwise, the highest mark you can get is half of the total.
5. Use indentation correctly in your program, otherwise you will lose 20 percents out of 100.