CS 451 – Operating Systems

Project 3

Due: 11:59 PM on Saturday, 12 September 2019

Objectives

- To familiarize yourself with threads and semaphores.
- To implement thread synchronization
- To implement a multithreaded program that simulates the movement of the people in the building and their use of the elevator (The Elevator Problem)

Teaming up!

For this project, you have the option to work with a partner (you may change your teammate). Please email the instructor with your teammate's name.

Overview

In a medium-sized city somewhere in Michigan stands a building with ***N*** floors, numbered from ***0*** to ***N-1***, and an elevator. The ***P*** people who work in the building use the elevator to move from one floor to another. At exactly 8 o'clock every weekday morning, these people enter the building and take the elevator to the floors they work on. The elevator is waiting with open doors at 8 o'clock and is big enough to hold all ***P*** people at once. From 8 o'clock in the morning until 5 o'clock in the afternoon, the people wander around the building, using the elevator to move from floor to floor as needed. The wandering time is *at least* 1 second in any floor they visit and is less than **`maxWanderTime`**. At the end of the day at 5 o'clock they always come back to floor 0 and wander there and then go home.

Your task is to write a multithreaded program that simulates the movement of the people in the building and their use of the elevator. Use semaphores to synchronize the elevator thread and the people threads.

Project Details

The program operates in a deterministic mode, where the itinerary of each person for the day is provided ahead of time.

In the deterministic mode, each person operates according to an input script, which determines which floor the person goes to and how long the person wanders around on that floor before using the elevator to go to another floor. Each person always starts at floor 0, and the last floor they come to is also 0.

When a person wants to use the elevator to go to another floor, the person pushes the elevator call button, waits for the elevator to arrive and open its doors, enters the elevator, pushes the button of the destination floor, and waits for the elevator to arrive at that floor. Because this is

an old building, each floor has just a single button to call the elevator, not separate “up” and “down” buttons we are more accustomed to seeing.

The elevator continually goes up and down, picking up and dropping off passengers. The elevator goes up until it reaches the highest floor N-1 and then goes down to the lowest floor 0. While going up and down like this, the elevator picks up and drops off passengers along the way. The important feature of the elevator algorithm is that the elevator does not reverse direction until it has gone all the way to the end in one direction as needed to pick up and/or drop off passengers.

The elevator takes one second to move from a floor to the next higher or lower floor. If the elevator needs to stop at a particular floor to pick up or drop of people, it takes one second to do so, no matter how many people embark or disembark. The elevator stops at a floor only if there are any passengers that need to get off or get on at that floor.

After each person returns to floor 0 and wanders and leaves the system, the person exits. When the elevator travels up and down the last time and sees that it has no passengers were picked up or dropped off, it sleeps for **`maxWanderTime`** and then checks to see anyone is waiting at any of the floors. If no one is seen still, it exits the system

Implementation Requirements

- Your program will have an elevator thread instantiated, and a person thread instantiated for each person. Then place all the semaphores and other state variables as data fields.
- The main method starts everything.
- The person thread blocks on a semaphore until picked up by the elevator.
- When picked up, the person object indicates the desired destination floor.
- The person thread blocks on a semaphore until the elevator arrives at the floor.
- When arriving at a floor, the elevator thread naps for one second to let passengers get on and off.

Tips

An outline of person thread. This is one way you can think about what a person thread does in the program.

```
void person(…) {
      while (notDone) {
            sleep(wanderTime);
            wait for the elevator to come to current floor get on the elevator;
            figure out the next floor to go to;
            wait for elevator to reach there;
            get off the elevator;
      }
      When done print good bye message and exit thread;
}
```

Input

The input to your program is the number of people in the building, the maximum wander time (napping time) and the number of floors in the building. Place these numbers on the command line for your program to parse.

$ elevator -p N_{people} -w maxWanderTime -f N_{floors} < file

-p <numberOfPeople> If –p is not given numOfPeople is set to 1 as default

-w <maxWanderTime> If –w is missing the maximum wander Time is set to 10 seconds

-f <NumberOfFloors> If –f is missing, the number of floors is set to 10

< file helps read the data from a file

Note: Remember to give the corresponding variables default values in your program to handle missing command line options.

In the *file*, for each person, one or more lines of the form are present.

n
f_1 t_1
f_2 t_2
f_3 t_3
...
f_n t_n

Here, n is the number of elevator ride/floor wandering pairs for the person and the pair f_i t_i means the person takes the elevator to floor f_i and wanders around on that floor for exactly t_i seconds before using the elevator to go to another floor. A sample input and output files are attached in the project dropbox.

Internal Requirements

1. Use sleep for wander time and alighting time for elevator.
2. DO NOT USE ANY OTHER SYNCHRONIZATION like locks or anything else to synchronize
3. Note that you can have global variables and semaphores that are global, however NO GLOBAL VARIABLES CAN BE READ OR WRITTEN w/o semaphore protection.
4. Note that elevator thread simply goes all the way up and comes down all the way checking to see if it should let passengers in or out along the way. If it just left floor 3, then some person wanted to get on at floor 3, that person has to wait until the elevator goes all the way up and the elevator comes downward. One the way down the elevator will pick up the passenger at floor 3.
5. NOTE: Since people do not know which direction the elevator is headed, it is possible that a person takes elevator to Floor 10 when the elevator is headed down to Floor 0. However, next time the elevator goes upward and reaches Floor 10, this person gets off.

Output Requirements

1. Output the input that is read in the following format
   `Person i: List of Floors to Visit:`
   `Person i: Amount of Time to Spend`
2. All outputs from elevator thread must be labeled "**`Elevator:`**…"
   It must also be printed several spaces to the right to distinguish it from output from person threads. (use 6 – 7 tab spaces)
   Output actions of the elevator when
   a. elevator stops at certain floor
   b. when it starts its upward journey
   c. when downward journey
   d. when it leaves the system
3. All outputs from person thread must be labeled "**`Person <i>:`** **`….`** "It must be left aligned to distinguish it from output of the elevator threads.
   Output actions of the people with their id,
   a. when person starts to wait to embark on the elevator
   b. when they embark the elevator
   c. when they wait to disembark the elevator
   d. when they start to wander
   e. when they leave the system.
4. Each time the elevator starts its upward journey, output the number of people waiting at each floor.

Getting Started with Semaphores

You will be using the library **<semaphore.h>** for the semaphore functions. You do not need to do anything special to compile the code with semaphore.h functions. The following functions would be most useful.

sem_t mutex; // declaring semaphore

sem_int(&mutex, 0, 1); // will initialize the value of the mutex to 1. You can initialize the mutex to any value of your choice.

sem_wait(&mutex); // If the mutex value is 1 or more the thread does not block, and it reduces the mutex value by 1; If the mutex value is 0 or below the thread executing this code will block until another thread wakes it up executing sem_post

sem_post(&mutex) // Increase the mutex value by 1 and unblock a thread that is blocked at sem_wait.

What to turn in

For this project, you must upload a **zip archive** to the course Blackboard assignment. This zip archive must contain the following items:

- One teammate from the team can submit
- Your **project3.c** and **.h** files, if any (no object files or executables, please!)
- A **Makefile** for compiling your source code.
- A **README** file (include your teammate's name) with some basic documentation about your code