Middle East Technical University | Department of Computer Engineering

CENG 334

Introduction to Operating Systems

Spring 2019-2020

Homework 2 - Elevator Simulator

Due date: 27/04/2020, Monday, 23:59

1 Introduction

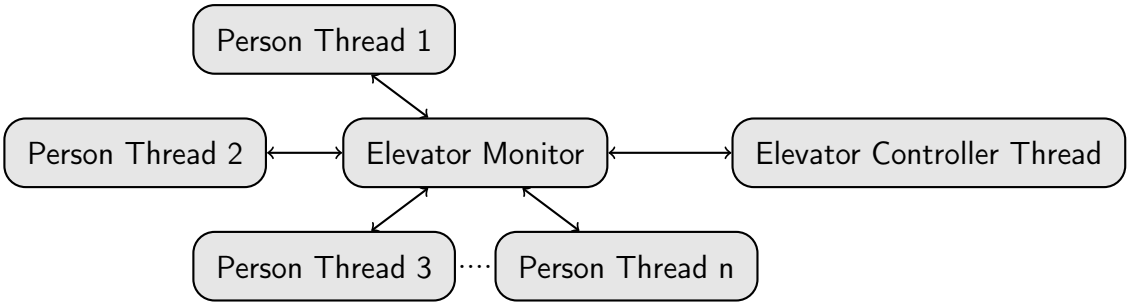
The objective of this assignment is to gain hands-on experience on solving **synchronization problems** through simulating an elevator in a multi-threaded program.

An elevator is "a type of vertical transportation machine that moves people or goods between floors, levels or decks of a building, vessel, or other structure"[1]. In this assignment, you will be implementing a *person elevator simulator*. The elevator has a fixed capacity on the weight and the number of people it can carry. It will receive asynchronous requests from people, which have different priority levels and different characteristics, and aims to transport them to their destination floors.

You will be using a monitor to simulate interactions between the elevator and the people subject to the synchronization constraints of the problem.

2 Components of the Program

The simulator will consist of 3 main components to perform this task; an *Elevator Monitor*, an *Elevator Controller* thread and multiple *Person* threads.



The *Elevator Monitor* regulates the access to the critical regions. It should be created through inheriting a C++ class, that is already implemented. Check the github repository[2] of Dr. Onur T. Şehitoğlu's to access examples on the Monitor implementation as well a its usage to solve some classical synchronization problems such as Barbershop and Readers-Writers.

[1] https://en.wikipedia.org/wiki/Elevator
[2] https://github.com/onursehitoglu

The *Elevator Controller* controls the movement of the elevator based on the requests coming from people. Those requests are stored in a destination queue which is filled with requests of people and emptied when those destinations are reached. It starts at the beginning of the simulation in the *idle* state, and continues to serve all the requests made by the *Person* threads until all requests are carried out.

In the *idle* state, the elevator will wait for requests to come for a fixed amount of time before becoming active (*IDLE_ TIME*). If no request comes after expiration of *IDLE_ TIME* it stays in *idle* state and waits again. If the elevator is moving in between floors it will wait for a fixed amount of time (*TRAVEL_ TIME*) before increasing/decreasing the current floor. If the elevator reaches to a destination it will notify the corresponding threads to allow them to leave/enter by waiting for a fixed amount of time (*IN_ OUT_ TIME*). Note that (*IN_ OUT_ TIME*) will be long enough to let all the *Person* threads enter/leave the elevator.

The *Person* threads make transportation requests to the *Elevator Controller* and will terminate after they are transported to their destination floors. Multiple *Person* threads make requests independently from one another.

3 The Elevator Controller

The *Elevator Controller* can be in three different states: *idle*, *moving-up* and *moving-down*. It will start in the *idle* state at the beginning of the program at *Floor 0* and will wait for requests from *Person* threads. The initial state of the simulator is shown in Figure 1. Elevator Controller is mainly responsible from the movement of the elevator. While it is active, it checks whether there is any destination to go in the destination queue. If there is no destination in the queue it lets other threads to take control to make calls to the elevator, otherwise moves the elevator up/down one floor at a time with *TRAVEL_ TIME*. Then it checks if the current floor is an *initial floor* or *destination floor* for any of the person threads. If that floor happens to be an *initial floor* for any person it deletes that floor from its destination queue (since it reached to one of its destinations) and notifies the *Person* threads that want to enter. After notifying those threads it waits for *IN_ OUT_ TIME* to allow *Person* threads to leave/enter. Elevator Controller terminates only after *all of the persons reach to their destinations.*

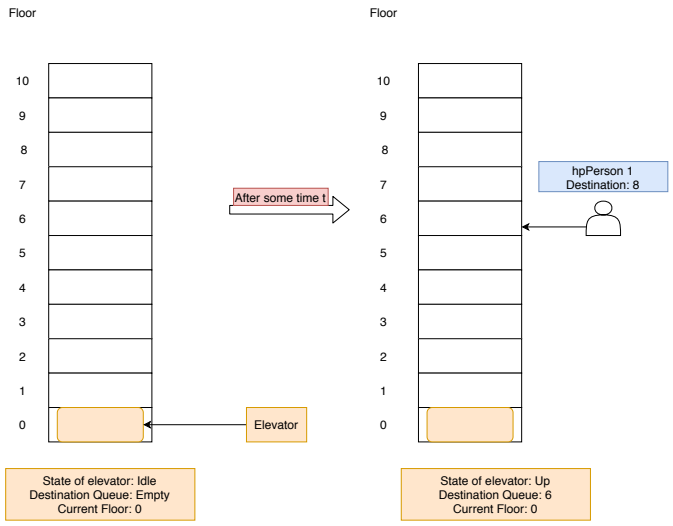


Figure 1: Initial state of the Elevator Simulator.

The pseudo code for the *Elevator Controller* is given in **Algorithm 1.**

```
while (Not all people have been served) do
    if (elevator is stationary at floor x) then
        while (No one is calling the elevator) do
            Wait for requests to come ;
        end
        Person at floor y wants to go to floor z;
        if (x < y) then
            Move upwards after some delay ;
        end
        if (x > y) then
            Move downwards after some delay ;
        else
            Move towards the floor z;
        end
    end
    if (Another person wants to enter in currentFloor) then
        if (Person does not violate entrance constraints) then
            Person enters the elevator ;
        end
    end
    if (Elevator reaches to a destination of a person inside the elevator) then
        Person leaves the elevator ;
    end
end
```

Algorithm 1: The pseudo code for the *Elevator Controller*.

Note that the entrance constraints are explained in Section 4.

4 Person Threads

There will be multiple *Person* threads making transportation requests. Each *Person* thread will make a transportation request to be transported from its current floor to a destination floor and will terminate after its request is carried out. A *Person* thread can make many requests until its request is carried out since the elevator will probably be full. These threads will make requests *as long as they are eligible to enter the elevator*. They conditions for eligibility are:

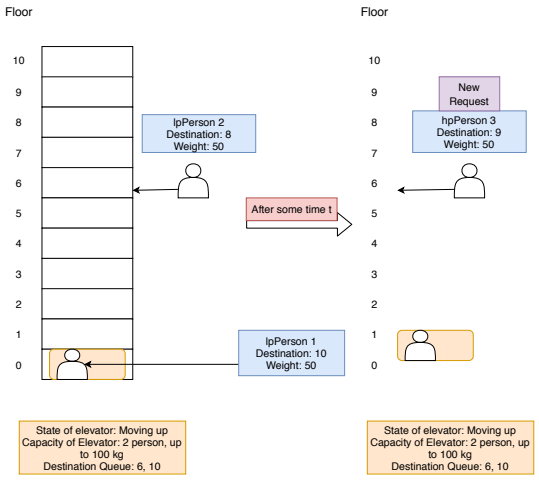
1. They want to go in a direction that is the direction of the elevator (if elevator is idle they can make a request).

2. They are not behind of elevator in terms of location and direction.

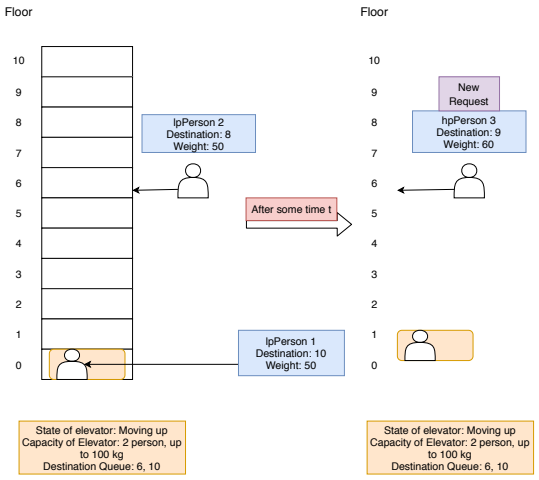
For example, if the elevator is *moving-up* from floor 2 to floor 9, it will not stop to pick up a person who is currently at floor 5 and wants to go to floor 3 or a person who is currently at floor 1 and wants to go to floor 7. It would however stop to pick up another person that is currently at floor 3 and wants to go to floor 4[3]. A *Person* thread is initialized with unique *person ID*, *weight*, *initial* and *destination floors*, and a *priority*. When a *Person* thread makes its request, the *Elevator Controller* will process it based on its own status, and the requests that it already has.

When a *Person* thread enters the elevator, destination queue is updated with the destination of the person unless the elevator already has that destination in its queue (**No duplicate destinations in the queue**).

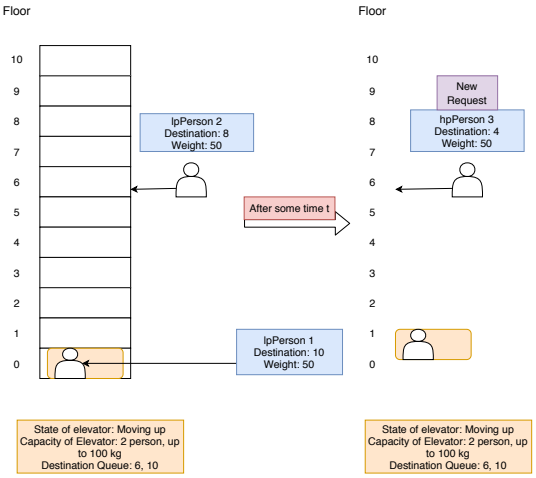
[3]You may find it amusing, how most people in this country are not aware or ignorant of this fact and would get into an elevator when it stops, disregarding the current direction of the elevator.



(a) Scenario one



(b) Scenario two



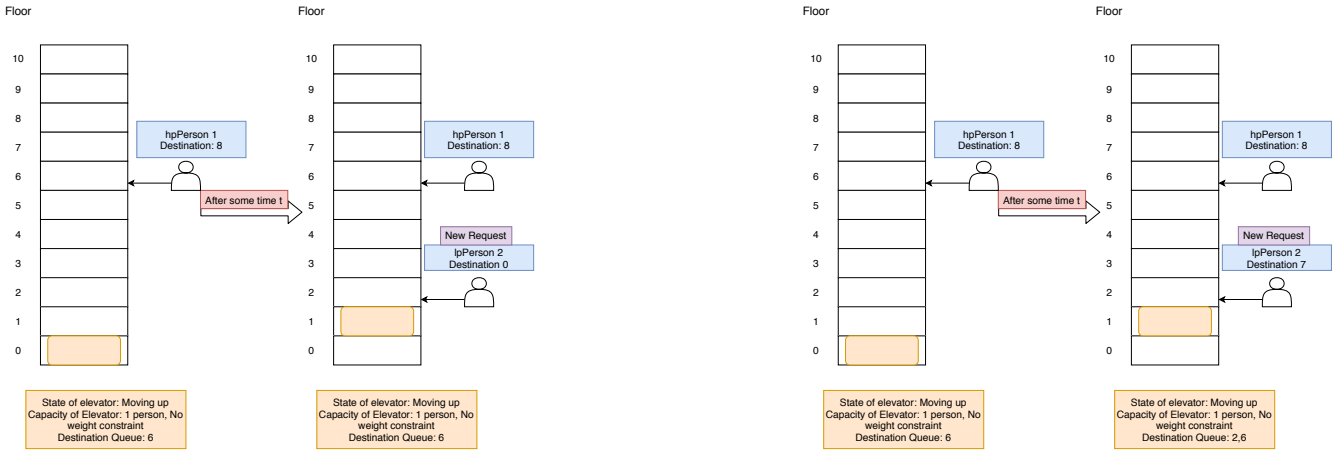
(c) Scenario three

Figure 2: Three different scenarios to handle priorities in a floor

The elevator stops at a floor either because it's a destination floor for a person that it is transporting, or It has a person which made a request to be transported in the current moving direction of the elevator. When the elevator stops, the people in the elevator leave (if it is their destination), before the people waiting for the elevator enter.

A *Person* can either be of *low-priority* (*lp*) or *high-priority* (*hp*). When the elevator stops at a floor, *high-priority* people will enter the elevator before the *low-priority* people at that floor. Entrance into the elevator is subject to the capacity (both in terms of weight and in the number of people) constraints of the elevator. A person, whose entrance into the elevator would exceed the allowed capacity, will be denied from entering the elevator and has to wait. Note that, priority level changes the order only within the people waiting at the same floor during their entrance into the elevator. It does not apply to the order or service to people in different floors. For instance, when the elevator is *moving-up* from floor 0 to floor 9, it will stop at floor 3 first and then at floor 5 to pick up people want to get to floor 8. The fact that high-priority people are waiting at floor 5, as opposed to low-level people waiting at floor 3 does not change the order of picking up.

The handling of priorities should in three different scenarios is shown in Figure 2.



(a) Scenario one

(b) Scenario two

Figure 3: Two different scenarios that can occur after Figure 1

5 Sample Scenarios of Interaction

In this section, we will focus on two main mechanisms about our simulation. These are 1) how elevator should move and when to stop at floors to let people enter, and 2) how the priorities should be handled in specific cases.

5.1 Scenarios About the Movement and Whom to Pick Up

- Figure 1: The elevator is initially *idle* and is waiting for requests. Then a request comes from a high-priority person who wants to go from floor 6 to floor 8. Since the elevator is initially idle, it adds floor 6 to its destination queue, and starts *moving-up*. Two possible scenarios that follows this one are given in Figure 3.

- In Scenario one of Figure 3a a low priority person calls the elevator from floor 2 and wants to go to 0th floor. Since elevator currently moving upwards, it ignores the call from lpPerson 2 and continues without changing its destination queue. lpPerson 2 will only be served after hpPerson 1 reaches to its destination, then elevator begins moving downwards towards lpPerson2.

- In Scenario two of Figure 3b, this time lpPerson 2 requests to go upwards and eventually lpPerson 2 will enter the elevator. As a result, elevator first updates its destination queue by adding floor 2, then stops at floor 2 to allow lpPerson 2 to enter. After lpPerson 2 enters, the destination queue changes again, this time to 6,7. Notice that this time hpPerson 1 will wait until elevator becomes empty(only after lpPerson 2 leaves the elevator at floor 7) then the elevator moves towards hpPerson 1.

5.2 Scenarios About the Priorities

In each of the three scenarios the elevator is at floor 0, carrying lpPerson 1, moving upwards and gets a request from lpPerson 2 at floor 6. After some time passes while traveling in between the floors 0 and 1, a new request comes from a high priority person at floor 6.

- In Scenario one that can be seen in Figure 2a hpPerson 3 can enter the elevator since it has the priority over lpPerson 2 and does not violate any of the capacity or direction constraints. As a

result, when the elevator comes to floor 6 hpPerson 3 enters and the destination queue becomes 8, 10.

- In Scenario two which can be seen in Figure 2b hpPerson 3 checks whether he/she can enter the elevator. However, this time he/she violates the capacity constraints and eventually lpPerson 2 tries to enter. Since lpPerson 2 does not violate any of the capacity or direction constraints he/she is eligible to enter. lpPerson 2 eventually enters and updates the destination queue which becomes 8, 10.
- In Scenario three which can be seen in Figure 2c even if hpPerson 3 has the priority over lpPerson 2 and satisfies the capacity constraints he/she is not eligible due to the violation of direction constraints. Therefore, lpPerson 2 is able to enter the elevator and destination queue becomes 8, 10.

6 Implementation Specifications

- Each *Person* should run as a separate thread.
- Input for priorities will be 1, and 2 for the high-priority and low-priority people respectively.
- The *Elevator Control* should run as a thread, should controls the movement of the elevator until all *People* have been served.
- **Elevator information** should be printed in the following form: "Elevator (*state of elevator*, *carried weight*, *number of people inside*, *current floor -> destination floors seperated witch comma*) terminated with newline character.
- Whenever elevator moves up/down by 1 floor, each time you need to print the elevator information afterwards.
- If elevator reaches to a destination floor, after letting people leave/enter you need to print out the the elevator information.
- If a person is eligible to enter to elevator (and makes a request), then print: "Person (*person ID*, *hp/lp depending on priority*, *initial floor -> destination floors seperated witch comma*, *weight*) made a request" terminated with newline. Additionally, you need to print out the elevator information.
- If a person enters the elevator print: "Person (*person ID*, *hp/lp depending on priority*, *initial floor -> destination floors seperated witch comma*, *weight*) entered the elevator" terminated with newline. In addition, you need to print the elevator information.
- If a person leaves the elevator print: "Person (*person ID*, *hp/lp depending on priority*, *initial floor -> destination floors seperated witch comma*, *weight*) has left the elevator" terminated with newline. Additionally, you need to print out the elevator information.

7 Input Specifications

The input will be read from a text file. The first line of input will consist of num_floors (N_D), num_people(N_P), weight_capacity(W_C), person_capacity(P_C), $TRAVEL\_TIME(TT)$, $IDLE\_TIME(IT)$ and IN_OUT_TIME (IOT), all time values are in terms of microseconds. Rest of the input will consists of N_P many lines each of which with the form weight_person(W_P), initial_floor(I_F), destination_floor(D_F) and priority(P). A sample input is as follows:

- 10 3 150 2 10000 5000 15000 -> First line, ($N_D, N_P, W_C, P_C, TT, IT, IOT$)

- 50 2 5 2 -> First person, (W_P, I_F, D_F, P)
- 60 1 5 1 -> Second person, (W_P, I_F, D_F, P)
- 50 2 5 1 -> Third person, (W_P, I_F, D_F, P)

Please note that all lines end with a white space followed by a newline character.

8 Homework Specifications

- Your code must be written in `C++`. However you can use `C`'s functionalities after making sure that you correctly inherited from Monitor class.
- Your programs will be compiled with `g++`. Make sure that you also use the `-lpthread` flag when compiling your program.
- Your solution should be built using the Monitor class provided. You do not need and hence should not use Semaphores and Mutexes.
- Although it is possible to have correct results with different outputs due to the concurrent nature of threads, please follow the output instructions carefully, the grading will mostly be black box and make sure you follow these rules.
- Busy wait solutions will be penalized. Deadlocks for a particular input, will get no points for that case.
- Do not delete anything from the code that is provided to you. You may need to add some small piece of code into `monitor.h`.
- **All code (except the code provided to you) submitted should completely be your own! Using code from your friends, previous homeworks, or the internet will be considered as cheating. All source codes will be automatically cross-checked for similarity. Keep in mind that This class has a Zero tolerance against cheating and disciplinary action will be taken for submissions with similarity values above the class average.**
- Follow the course page on COW for update and clarifications. Please post your questions on COW instead of e-mailing if the question does not contain code or solution.

9 Submission

Submission will be done via ODTUClass. You should submit a tar file called `hw2.tar.gz` that contain all your source code together with your `makefile`. Your tar file should not contain any folders. Your makefile should be able to create a single executable named Elevator and it should be able to run using the following command sequence.

```
> tar -xf hw2.tar.gz
> make all
> ./Elevator inp.txt
```

You can assume that all the executables and library files are present for the required compilation and execution. **Any errors generated during the 3 steps listed above will be penalized with 10 points.**

10 Sample Execution

Given input:

```
10 3 150 2 10000 5000 15000
50 2 5 1
60 1 5 2
50 2 5 2
```

The corresponding output (**not a unique correct output**) is:

```
Person (0, hp, 2 -> 5, 50) made a request
Elevator (Moving-up, 0, 0, 0 -> 2)
Person (1, lp, 1 -> 5, 60) made a request
Elevator (Moving-up, 0, 0, 0 -> 1,2)
Person (2, lp, 2 -> 5, 50) made a request
Elevator (Moving-up, 0, 0, 0 -> 1,2)
Elevator (Moving-up, 0, 0, 1 -> 2)
Person (1, lp, 1 -> 5, 60) entered the elevator
Elevator (Moving-up, 60, 1, 1 -> 2,5)
Elevator (Moving-up, 60, 1, 2 -> 5)
Person (0, hp, 2 -> 5, 50) entered the elevator
Elevator (Moving-up, 110, 2, 2 -> 5)
Elevator (Moving-up, 110, 2, 3 -> 5)
Elevator (Moving-up, 110, 2, 4 -> 5)
Elevator (Idle, 110, 2,  5 ->)
Person (0, hp, 2 -> 5, 50) has left the elevator
Elevator (Idle, 60, 1, 5 ->)
Person (2, lp, 2 -> 5, 50) made a request
Elevator (Moving-down, 60, 1, 5 -> 2)
Person (1, lp, 1 -> 5, 60) has left the elevator
Elevator (Moving-down, 0, 0, 5 -> 2)
Elevator (Moving-down, 0, 0, 4 -> 2)
Elevator (Moving-down, 0, 0, 3 -> 2)
Elevator (Idle, 0, 0, 2 ->)
Person (2, lp, 2 -> 5, 50) made a request
Elevator (Idle, 0, 0, 2 ->)
Person (2, lp, 2 -> 5, 50) entered the elevator
Elevator (Moving-up, 50, 1, 2 -> 5)
Elevator (Moving-up, 50, 1, 3 -> 5)
Elevator (Moving-up, 50, 1, 4 -> 5)
Elevator (Idle, 50, 1, 5 ->)
Person (2, lp, 2 -> 5, 50) has left the elevator
Elevator (Idle, 0, 0, 5->)
```