Lab 4: Concurrent Elevator Simulation Using Pthreads

IUPUI CSCI 503 – Fall 2019
Assigned: November 7 Thursday, 2019
Due time: 11:59PM, November 25 Monday, 2019

1. Introduction

You need to write a multi-threaded program using pthreads to simulate how *multiple elevators* will move a lot of people up and down in a tall building.

The input to your simulator should be the same as follows (in exactly the following order):

1. Number of elevators (must be > 0).
2. Number of floors (must be > 0).
3. How often people arrive (*p* seconds per person): After every *p* seconds, a new person arrives at a random floor and waits for an elevator. Any one of the elevators can satisfy that person if multiple elevators arrive at his or her floor simultaneously. Time interval *p* > 0.
4. Elevator speed (e seconds per floor): An elevator takes *e* seconds to move from one floor to the next floor. Speed *e* > 0.
5. Total time to simulate (*s* seconds): The total time you want to run the simulation. After *s* seconds, your program should stop and prints out the required statistics information (See Section 6). *s* > 0.
6. Seed of the random function: A *random number generator* is needed to generate a person's random *from-floor#* as well as her *to-floor#*. A different seed changes the generated random number sequence.

In the simulator program, you just need one single thread to generate persons (yes, this is a simplified version). Every elevator has its own thread to move up and down to serve people waiting on different floors.

2. How the *People-Generation Thread* Works

You will need a while loop. For every p seconds, the thread creates a new person and adds it to a global double linked list. Person has a data structure (defined later) and its content has to be filled appropriately. ~~(in a more realistic simulator, each person will be represented by an independent thread, which calls wait_to_get_on, get_on, wait_to_get_off, and finally get_off in a sequence.)~~

3. How the *Elevator Thread* Works

Each elevator thread executes a while loop. It checks the global double linked list: if the list is empty, blocks itself on the elevator's conditional variable until a new person arrives; if not empty, removes the first person from the global list, then the elevator moves to the person's from-floor, then moves to the person's destination floor, and finally drops the person. Obviously, this moving method is not efficient because this elevator only serves one person at a time!

4. Next, an Improved Version

A solution to the inefficiency problem is to simulate real-world elevators. That is, when an elevator is moving up, it will only pick up people who are on their way going up. Similarly, when an elevator is moving down, it will only pick up people who are on their way going down. On each floor where it stops, an elevator can pick up as many people as possible (i.e., there is no capacity limit). The elevator should not pick up any person who is going down if the elevator is moving up, and vice versa. Note that you need to search the double linked list to find who should be picked up.

Note: Whenever in doubt, try to imagine how a real-world elevator is working.

5. Suggested Data Structures

```
struct person {

  int id;                       /* 0, 1, 2, 3, … */

  int from_floor, to_floor; /* i.e., from where to where */

  double arrival_time;      /* The time at which the person arrives */

  …                             /* You can add more fields if you want */

};

struct elevator {

    int id;                      /* 0, 1, 2, 3, … */

   int current_floor;          /* Current location of the elevator */

   pthread_mutex_t  lock;

   pthread_cond_v     cv;    /* Will be used to block the elevator if there is no request */

   struct person   *people;/* All the people currently inside the elevator */

   …                            /* You can add more fields if you want */

};
```

```
struct gv { /* All the global information related to the elevator simulation */
   int num_elevators;
   int num_floors;
   int beginning_time;
   int elevator_speed;
   int simulation_time;
   int random_seed;
   int num_people_started;  /* statistics */
    int num_people_finished; /* statistics */
   pthread_mutex_t *lock;
    …
};
```

These data structures may not be complete. You can define additional data structures.

6. Simulation Print-out and Statistics Output (required)

- In the people-generation thread:
  - For every new person, print "[time] Person id arrives on floor A, waiting to go to floor B".
- In each elevator thread:
  - Print "[time] Elevator id starts moving from P to Q …"
  - Print "[time] Elevator id arrives at floor Q."
  - Print "[time] Elevator id picks up Person id1, Person id2, and so on."
  - Print "[time] Elevator id drops Person id1, Person id2, and so on"
  - Note: In the basic version, an elevator picks up and drops only one person. In the improved version, it picks up and drops multiple persons.

  When the simulation starts, [time] is equal to 0. So you need to deduct current_time by beginning_time to get [time].

- When the program exits, you must print out:
  - "Simulation result: x people have started, y people have finished during z seconds."

7. Score Distribution

- Coding style and error checking: **10%** (as described in the "Labs Grading Policy")
- The basic version of simulation: **50%**
- The improved version: **40%**

Note: This lab is more likely to cause segmentation faults than the previous labs. Make sure you test your code thoroughly.

The total grade is 100 points.

8. Deliverable

There should be **two subdirectories**: **one for the basic version** and **the other for the improved version**.

In each subdirectory, there should be source code in C, a Makefile, and a README. Make one tar ball for the two subdirectories and submit it to Canvas. The tar ball name should be "Lab4_your_name.tar".