1 Introduction

To get started, use this invitation link
`https://classroom.github.com/a/7UdbbY78` to create your turn-in repository. As usual, run `make C` or `make rust` to select your language.

An M story high-rise has N elevators in a single "elevator bank", each able to serve every floor of the building. Contrary from most elevator designs, there are no buttons to choose a destination floor inside the elevator. Instead, the elevator lobby at each floor has M individual "destination floor" buttons, and passengers choose their destination by pushing the appropriate button. Once a button is pushed, a display near the buttons tells the passenger which elevator door to wait by. In this assignment, elevators serve a preconfigured maximum number of passengers at a time (default is 3).

Your assignment is to create an elevator controller that ensures that all passengers receive (safe, reliable) service, minimizing the amount of time it takes to serve all of the passengers, and how much CPU time is used to run the controller.

2 The Code

The homework 7 template is a multithreaded program which simulates passengers and elevator travel. There is one thread per elevator, and one thread per passenger. There is a fixed number of passengers, that embark on a number of trips each. These threads interact with each other through a central elevator controller, which is your responsibility to implement.

The central controller responds to the following two function calls:

```
/* called whenever a passenger pushes a button in the elevator lobby.
   call enter / exit to move passengers into / out of elevator return
   only after the passenger has been delivered to requested floor */

void passenger_request(int passenger, int from_floor, int to_floor,
    void (*enter)(int, int), void(*exit)(int, int));

/* called whenever the elevator needs to know what to do next.
   call move_direction with direction -1 to descend, 1 to ascend.
   must call door_open before letting passengers in, and door_close
   before moving the elevator */

void elevator_ready(int elevator, int at_floor,
    void(*move_direction)(int, int), void(*door_open)(int), void(*door_close)(int));
```

2.1 Rust version

The rust version of this assignment uses the C infrastructure, but you write your solution in a Rust library crate. The crate creates a static library, with

which the infrastructure code is linked to create the final executable. Simply modify the `src/lib.rs` file in the provided `elevator` crate. If you need other dependencies, this is allowable, go ahead and edit the `Cargo.toml` as needed.

To build the Rust version, use the Makefile in the rust folder. You may pass in configuration parameters as environment variables to make, like this `ELEVATORS=2 PASSENGERS=14 make elevators`.

Note: solving this assignment in Rust is likely to be considerably easier than solving it in C, as long as you prevail in your battles with the compiler. Not only are you likely to make fewer mistakes, but you have a rich standard library providing handy collections making it much easier to express whatever scheduling logic you would like to implement.

3 Performance Expectations

The template code works correctly, but is terribly slow. In part, this is because passengers often "miss" the elevator (don't get in in time), or miss their floor (stay on the elevator). In part, it is because the elevator stops at every floor, we only use one elevator, and we only allow one passenger in the elevator at a time. In fact, only one passenger is allowed to move at all at any given time.

For a full score, have a fully working solution that uses all available elevators, and finishes each test case in less than (**TBA**) seconds without any busy waiting (aka uses less than 1As a guideline: for the default configuration, below 10 seconds is tolerable, below 7 is pretty good, below 5 good and below 4 excellent. The actual grading criteria will be revealed later.

There will also be a leaderboard for this assignment. The leaderboard is only for fun. The leaderboard score is computed as 10000 - (execution time for the leaderboard test case in milliseconds).

The autograder runs a slightly modified version of the Makefile given in the handout - all of the test case variables are the same, but it does not print to the screen, which will likely cause execution time and CPU usage to go down by a very small amount compared to running on systemsX. You should be able to write this code on any modern Linux machine.

3.1 Notes

- Every passenger pushes a button.
- Passengers can't change their minds, and will wait until they are told to enter the elevator, using the enter() callback function
- Passengers will stay in the elevator until they are told to leave, using the exit() callback function.
- Elevators can hold at most `CAPACITY` passengers.
- The program finishes after all passengers have been served, and all functions have returned.

3.2 Hints

- Start by making sure your passengers have time to get on and off the elevator. You can do this with condition variables (pthread_cond_t), or barriers (pthread_barrier_t). To support multiple passengers in the elevator, a condition variable-based solution will be easier.
- The template supports only one passenger using the elevators at any given time, due to the passenger lock. Relax this by only holding the passenger lock when modifying shared data or calling enter/exit, rather than while waiting. Passengers will now crowd in over the elevator capacity - add code to avoid this.
- The template has passengers simply wait until the elevator shows up. Add code to let passengers tell elevators where they are, and where they are going, to make elevator dispatch more efficient.
- The template opens and closes the door on every floor, and door opening/closing is slow. When an elevator is occupied, try going straight to the destination floor and only opening the door there.
- The template solution has a single set of global variables for elevator state. You'll probably want to have one set per elevator. Define an elevator struct that holds all your necessary state per elevator, and make an array of such structs.
- Handling multiple elevators should be the last item on your TODO list. An easy way to do it is to randomly decide, for each passenger, which elevator they should use, independent of everything else. Then you can treat each elevator+passengers group separately.
- Alternatively, use a shared "elevator scheduling" lock, and let elevators share the pool of passengers. This may be considerably more efficient than statically assigning passengers to elevators.

4 Logging

To add more logging output, use the `log()` macro instead of printf. `log(loglevel, format_string, parameters)` works just like fprintf(logfile,...) except it prints both to `elevator.log` and to the screen during execution. Moreover, `log()` only prints if the configured log level is higher than the loglevel argument provided. To set the log level to, for example, 8 use the following flag to gcc -DLOGLEVEL=8

5 Testing

Test your program varying the configuration settings `ELEVATORS`, `FLOORS`, `PASSENGERS` and `CAPACITY`. The autograder may test for correctness with one elevator, multiple elevators, single-passenger capacity and larger capacities.

6 Turn-In

Commit and push your solution to your github turn-in repo, then submit it to gradescope using the github submission method. Name your solution `hw7.c` (for a C submission). Do not modify any other files: your solution will be graded against the original versions of the other files.

6.1 Rust submission

For rust, simply modify the "elevator" crate provided, placing your code in the lib.rs file.