Assessment 2 component: Elevator controller

Summary

The elevator controller is responsible for the operation of a single elevator - opening and closing the doors and directing it to travel up and down floors. The elevator will receive commands to go to a particular floor from the overseer via TCP - so, for example, if someone on level 10 hits the destination select to travel to level 4, the elevator will be directed to go to level 10 then 4. Physical buttons inside the elevator can still be used to open the doors, but this happens automatically and is not the responsibility of the elevator controller.

Program name

```
elevator
```

Command-line arguments

```
{id} {address:port} {wait time (in microseconds)} {door open time (in microseconds)} {shared memory path} {shared memory offset} {overseer address:port}
```

Shared memory structure

```
struct {
    char status; // 'O' for open, 'C' for closed and not moving, 'o' for opening, 'c' for closing, 'M' for moving
    char direction; // 'U' for up, 'D' for down, '-' for motionless.
    uint8_t floor;
    pthread_mutex_t mutex;
    pthread_cond_t cond_elevator;
    pthread_cond_t cond_controller;
};
```

Initialisation

On startup, this component will bind the TCP port it was supplied with and start listening on it. It should be non-blocking - use `fcntl (https://man7.org/linux/man-pages/man2/fcntl.2.html) (fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK);` after opening the socket so that accept() does not block.

It will then send the following initialisation message to the overseer via TCP:

`ELEVATOR {id} {address:port} HELLO#`

The default status on initialisation will be `C`.

Normal operation

After initialisation, the component will perform the following loop:

1. If there is a TCP connection waiting:
   A. Accept the TCP connection and receive a message from the overseer (of the format `FROM {from floor} TO {destination floor}#`)
   B. Close the TCP connection
   C. Enqueue the floors (see the 'Elevator scheduling' section below)
   D. Loop back to 1
2. If the current queue is empty, skip to 8
3. Lock the mutex
4. If the floor value from shared memory is equal to the next entry in the queue and the current status is `C`
   A. Set the status to `o`

   B. Signal cond_elevator
   C. Remove the next entry from the queue
   D. Skip to 8
5. If the next entry in the queue is a higher floor than the value from shared memory and the current status is `C`
   A. Set direction to `U`
   B. Signal cond_elevator
   C. Skip to 8
6. If the next entry in the queue is a lower floor than the value from shared memory and the current status is `C`
   A. Set direction to `D`
   B. Signal cond_elevator
   C. Skip to 8
7. If the current status is `O`
   A. Timed wait on cond_controller for {door open time} microseconds
   B. If the current status is not `O`, skip to 8
   C. If the next entry in the queue is a higher floor number than the current floor, set direction to `U`
   D. If it is lower, set direction to `D`
   E. If it is the same floor number, set direction to `-`
   F. Set status to `C`
   G. Signal cond_elevator
8. Timed wait on cond_controller for {wait time} microseconds
9. Unlock the mutex
10. Loop back to 1

Elevator scheduling

The elevator controller will maintain a queue of floors to visit and will visit each floor in the order they appear in the queue. However, the queue is not a pure 'first come first serve' queue, as this would involve the elevator potentially moving back and forth inefficiently. Instead, newly queued floors will be added to an appropriate place in the queue to enable efficient service.

The elevator controller will receive floors in pairs from the overseer in the form of TCP messages of the form: `FROM {from floor} TO {destination floor}#`

This indicates that someone on {from floor} has requested an elevator to take them to {destination floor}. Both of these floors will be added to the queue, but with the following rules:

- If both floors are already in the queue and the *destination* floor appears **after** the *from* floor, nothing needs to be added to the queue
  - For example, FROM 4 TO 8# with the elevator on floor 2 and the queue is {3, 4, 6, 8, 9}, the queue does not need to be changed
- If the *from* floor already exists in the queue, the *destinatio*n floor will need to be added to the queue **after** the *from* floor (at an appropriate location, taking into account the efficiency rules listed below)
  - For example, FROM 9 TO 2# with the elevator on floor 6 and the queue is {7, 9, 11, 5, 1}, floor 2 will be added to the queue between 5 and 1
- If the *destination* floor already exists in the queue, it might be possible to add the *from* floor before it - this can be done as long as it does not cause the elevator to have to change directions midway through. Otherwise both the *from* floor and the *destination* floor will need to be added to the queue.
  - For example, FROM 10 TO 5# with the elevator on floor 12 and the queue is {11, 9, 5, 3}, floor 10 will be added to the queue between 11 and 9
  - On the other hand, FROM 3 TO 7# with the elevator on floor 10 and the queue is {9, 8, 7, 5, 4, 2}, floor 3 needs to be added between 4 and 2 and floor 7 needs to be added to the end of the queue, so the resulting queue is {9, 8, 7, 5, 4, 3, 2, 7}
- Floors will be added to the queue with a preference for minimising changes of direction.