Green Elevator

Ludvig Larsson

March 14, 2022

1 Introduction

I have made this project on my own and designed the controller by myself without using external sources that could have helped me to develop an efficient algorithm to assign elevators to specific requests. I thought that would be funny to come up with myself, so I did. I used VS Code as text editor and compiled with gcc. I ran all programs on macOS and used pthreads library to control synchronization. To run the elevator GUI I ran elevator.jar via IntelliJ with appropriate arguments. To build the controller I used a Makefile to simplify.

2 Design and implementation

Implementation specific details are discussed in this section as well as the main algorithm for assigning elevators to service requests.

2.1 Implementation

I decided to develop my elevator controller in C with help of the `pthread.h` library to achieve concurrency and parallelism. If the program is started with 3 elevators there will be 4 threads in total running concurrently, since a main thread is needed too.

The main thread will first initiate the hardware connection to the client via predefined function `hw_init()` in file `hardwareAPI.c`, to set up the socket communication channel to the GUI. Next step in execution is to create the threads with appropriate structures needed for thread communication. Each thread will be assigned a personal `struct`, and main thread will have references to each of them.

2.1.1 Elevator struct

```
typedef struct elevator_state_t {
    pthread_mutex_t mutex;      #1
    pthread_cond_t cond;        #2
```

```
    int eid;                    #3
    int emergency_stop;         #4
    double current_position;    #5
    int direction;              #6
    queue_t *up_now;            #7
    queue_t *down_now;          #8
    queue_t *up_later;          #9
    queue_t *down_later;        #10
    queue_t *current_queue;     #11
} elevator_state_t;
```

Description and usage of fields in elevator struct

- **elevator_state_t fields 1 & 2:** Thread private synchronization primitives. Mutex is used when reading/writing shared resources in the struct (this elevator and main thread). Conditional variable is used to minimize busy-waiting state of elevators. Will be explained further later.
- **elevator_state_t field 3:** Elevator id. Logical which corresponds to numbers received from the GUI, e.g. `"b 1 2"`. Used for printouts mainly, no actual functionality.
- **elevator_state_t field 4:** Binary flag. Cabin might press emergency stop (32000) and elevator will quit servicing when this when this flag is set.
- **elevator_state_t field 5:** Current position of elevator. Updated every time the main thread received position data from the GUI. Elevator needs this variable to stop at correct position.
- **elevator_state_t field 6:** Current servicing direction of elevator, doesn't mean that it is actually moving in the direction. Controlled by the elevator itself. Direction is 0 (idle), -1 (servicing down) or 1 (servicing up).
- **elevator_state_t fields 7 & 9:** The current queued stops in direction up or down. Might not be serviced at currently, but once the elevator is servicing this direction this queue should be conformed to.
- **elevator_state_t fields 8 & 10:** Queues for stops in current direction being serviced, but unable to be serviced this elevator run, stops put on hold basically. Will be serviced once opposite direction "NOW"-queue is finished, but will be moved to corresponding "NOW"-queue once the current "*_NOW"-queue is empty and be dequeued from there instead.

- **elevator_state_t field 11:** Pointer to current queue that the elevator is following. Used in multiple places in the program to determine which queue that is being used currently, without branching.

It is not necessary to place these elevator_state_t structs in global scope. Elevators solely care about their own state, and main will have references to each elevator state. Avoiding global scope is great. Elevators doesn't need state of main thread nor other elevators and therefore I liked this implementation decision.

```
elevator_state_t *elevator_states =
    (elevator_state_t*)malloc(ELEVATORS * sizeof(elevator_state_t));

// assign default values to elevator states

pthread_create(..., &elevator_states[tid]);
```

2.1.2 Queue and queue entry structs

```
typedef struct queue_entry_t {
    type_of_req type_of_req;
    int floor;
    struct queue_entry_t *next;
} queue_entry_t;

typedef struct queue_t {
    queue_type type;
    queue_entry_t *first;
} queue_t;
```

Description and usage of fields in queue and queue entry structs

- **queue_entry_t:** The primitive used in queues the program. Field type decides if it is a pick up (SRC) or drop off (DEST), floor is where to go and next pointer allows for a linked queue.
- **queue_t:** Holds an arbitrary amount of queue_entry_t, references by first pointer. Type defines which queue (`UP_NOW, DOWN_NOW, ...`). Type is used for printouts mostly.

```
void enqueue_sorted(queue_t *queue, queue_entry_t *elem, int direction);
queue_entry_t* dequeue(queue_t *queue);
void set_type(queue_t *queue, int floor, int type);
int is_empty(queue_t *queue);
```

```
int contains(queue_t *queue, int floor);
int peek_next_stop(queue_t *queue);
int peek_last_stop(queue_t *queue);
int get_num_stops(queue_t *queue);
```

Queues have their own API defined by the above header file, made by myself. It is used to handle queues in the program. Functions are most self-explained, though worth mentioning `enqueue_sorted(...)` is sorting entries based on direction (so that elevators stop in correct order when servicing). Also, `set_type(...)` explicitly sets a queue entry to "type" argument given (SRC/DEST).

2.1.3 Main thread's run loop

After starting the threads, the main thread will mainly listen for incoming events from the elevator system, including both cabin-/floor- button presses from clients and events describing the elevators' positioning.

Overview of functionality for each incoming action event:

- **FloorButton:** Find a suitable elevator by calculating cost for each elevator. Lock each elevator's mutex first. Then save the cost (explained in next subsection) and queue (appropriate queue to when the request can be handled, e.g. `UP_NOW`)) for each elevator. Allocate a queue_entry_t pointer, with type "SRC" (pick up here) and add it to the appropriate queue corresponding to the elevator who could provide lowest cost. If the queue already contains the entry the already existing entry will be set to (SRC) to allow for cabin-button pressing once arrived. Elevators are waiting on its conditional variable if it's idle and might therefore have to be awakened. Lastly, unlock all mutexes, since main is now done reading and writing queue contents.

- **CabinButton:** The elevator who issued this action event will have the cabin-button request added to the current queue being serviced by the elevator. Before that the main thread has to lock the elevator's mutex and create the queue entry with type "DEST". After the enqueueing the elevator will be signaled (elevator sleeps during wait for cabin-button press) and lock will also be released since altering is done. In case the requested floor is "32000" the main thread will increment its local counter of emergency stopped elevators, and when all elevators have been emergency stopped the main thread will **exit** its run loop.

- **Position:** Position is related to one elevator, lock its mutex, update the elevator's current position and unlock the mutex.

2.1.4 Elevator threads' run loop

Elevator threads will continuously run and serve button presses issued by people outside (floor-button press) or inside (cabin-button press) the elevator. Best explained with pseudo code. Handling of doors and elevator motors are done via the `hardwareAPI.c` file. I had no good option to avoid busy-waiting while polling to stop at the next stop, as I didn't want to put elevator logic inside the main thread. I have cond wait at the other two while loops which is good enough I thought. Could possibly be changed to

```
endless loop
    take the lock (check lists)
    if elevator is idle
        while no list is populated with stops to service
            if UP_NOW isn't empty
                set fields current_queue=UP_NOW and direction=UP
                break loop
            if DOWN_NOW isn't empty
                set fields current_queue=DOWN_NOW and direction=DOWN
                break loop
            cond wait for main thread to signal new stop available

    *has lock when woken up*
    while no emergency stop && current queue isn't empty (NOTE: polling)
        save peek of next stop from current queue
        give back lock (done reading queue)
        get motor direction to serve next stop
        if motor action should alter
            handleMotor(...) via API
        if stop is in +-0.04 range of current position
            handleMotor(STOP)
            handleDoor(OPEN)
            take lock (alter queue)
            remove stop from current queue (dequeue)
            if type=SRC
                cond wait for cabin-press and signal from main
            unlock (done altering queue)
            handleDoor(CLOSE)
        take lock (read queue for while)

    *has lock when exited above loop*
    (NOTE: should release lock before upcoming if and acquire after)
    if emergency stop
```

```
    handleMotor(DOWN)
    give back mutex (elevator will exit)
    break out of run loop altogether

change first pointer of NOW-queue to instead point to LATER-queue's first pointer
set UP_LATER's first to null ("LATER-queue empty")

get opposite NOW-queue
if opposite NOW-queue isn't empty
    update current queue
    direction *= -1 (opposite)

if current queue is empty (up and down empty)
    handleMotor(STOP)
    current queue null
    direction=0
give back mutex (locked in beginning again)
```

2.2 Algorithm for assigning elevators

To control which elevator to service an incoming floor-button press we need to implement a function that will be able to tell which elevator is best suited to take action at a given point in time. We should take into account that people do not want to wait for too long to enter an elevator and people inside the elevator do not want to be interrupted during the travel, because of increasing travel time. Furthermore motors controlling each elevator should not be using an excessive amount of power just to service people as fast as possible. If floors at BV, 1 and 2 all want to take an elevator to the 10th floor it is not a good idea to prioritize that floor 1 instantly elevates to floor 10 without stopping at floors 2 and 3. Reasoning is that it can possibly make all 3 elevators, move from initial position at ground floor (0) to 10th floor, which is power consuming.

The following algorithm I created is not perfect at optimizing which elevators should be used by the means of sometimes maybe having 2 elevators go from BV to 10th floor since there might be very many people interested in going upwards at a point in time. This, in contrast to the situation where only one elevator should be enough.

Instead I have made the calculation of cost rather static and includes the total distance an elevator has to travel before servicing the incoming request and also how many stops it has been assigned. Each elevator in the elevator system is evaluated and the elevator with lowest cost is assigned the job to service the floor-button request. Based on which state an elevator currently is in there are different movement paths they have to complete before being able to service the request. The cost-algorithm does not predict or take into account that current stops being evaluates might be floor-button presses which can possibly lead

to additional distances to travel before being able to service the floor-button request. In other words, there is no foreseeing involved and also, once an elevator has been assigned the stop will not be handed over to another elevator dynamically, despite being a better decision in the current state.

- "CASE 1": The elevator is currently idle and can service instantly.
- The elevator is currently servicing in the correct direction corresponding to the floor-button request.
  - "CASE 2": The elevator has already passed by the floor-button request. Elevator has to service current direction and opposite direction. Then it can service the floor-button request.
  - "CASE 3": The elevator has no passed by the floor yet and will service the floor-button request instantly as soon as it gets there.
- "CASE 4": The elevator is currently servicing opposite direction of floor-button request and has to finish those first before being able to service.

There are lists corresponding to each of the elevators as mentioned in previous section *Implementation, 2.1*, namely `UP_NOW, UP_LATER, DOWN_NOW, DOWN_LATER`. The cost-function must evaluate lists based elevator case in contrast to the floor-button request. As an example, we consider "CASE 4". Elevator is currently servicing down in the elevator, i.e. servicing people who want to go down with the elevator. A person is requesting floor-button UP, which means that CASE 4 is relevant. Proceed as following, we evaluate the distance from the elevator's current position to the last stop in the current direction. Then also the distance from the last stop in current direction to the last stop in the correct direction (the requested direction of floor-button). But since the algorithm always picks up requests on the way the last stop in current direction might not be the endpoint of travel, the requested floor-button might be positioned before which leads to requested floor-button being the endpoint instead.

It is also possible that the elevator is moving downwards just to pick up people in the up-going direction which needs to be handled correctly with branches. Furthermore and lastly, if the elevator has to go up-down-up or down-up-down (change direction twice, CASE 2) the "wrong" direction might not have any stops which leads to additional branching.

The last 30 minutes I coded the algorithm for calculating cost I also implemented an evaluation for how many stops an elevator had currently to also be accounted for, instead of just calculating travel distance. This was done by simply by counting elements in each direction queue.

When people enter the elevator and presses a cabin-button, there is no possibility of different execution in the program. Once an elevator stops at an assigned floor where the floor-button has been pressed, the following cabin-button press is simply placed in the queue currently in use by the elevator. The logic has been

3 Conclusion

As I added number of stops for each elevator when calculating cost it will often change elevator assignments and put additional elevators in use. On the downside, the power consumption is increased and that is higher power consumption as a higher number of elevators are assigned now. I'm not entirely sure which of the cost implementation I like most, and it is after all the main question asked in the project formulation.

I really enjoyed programming a really concrete real life example of elevators. I had a hard time making homework 1 work with pthreads but now I got it right from the start and learned from hw1 mistakes so that was a sweet revenge! I'm not entirely satisfied with how I am polling (busy-waiting) on the elevator's current position to know if it should stop at a floor or not. I do believe that it could have been solved in another way but I couldn't come up with one during the project. I saw that other course projects had an algorithm outline and this elevator could also have had one. It took me very long to come up with a not-so-good (?) solution. I did what worked for me basically and have not had time to evaluate if it was good and how it could be changed. Really funny project overall but very time consuming.

I wonder how a big project combined with 5(4) homeworks only gives 3HP credits. Too high workload in contrast to low HP credits.