Green Elevator

Kevin Ammouri, Youssef Taoudi

March 2019

1 Introduction

Implementing a controller for an elevator that responds to button clicks. Depending on what button that is clicked and on the current positions of the elevators, the program will try to find the most cost-effective way of scheduling the elevators.

2 Environment

Two computers were used to test the program, both running Mac OS Mojave.

Computer 1:

- **OS** macOS, Mojave
- **Processor** 1,4 GHz Intel Core i5, 2 Cores
- **RAM** - 4 GB

Computer 2:

- **OS** macOS, Mojave
- **Processor** 2,3 GHz Intel Core i5, 2 Cores
- **RAM** 8 GB

3 Design and Implementation

The program was written in C and utilized the `pthreads` library for synchronization and locking. The work in the program was divided into 5 different kinds of threads, one for listening for new events (button clicks, elevator movement or door movement), event handler threads for handling each particular event and elevator threads for each concurrently running elevator.

3.1 Data Structures

3.1.1 Linked Lists

The main data-structures used in the program were doubly-linked lists acting as queues. A queue provides two functions, `enqueue(int type, Event* event)` and `dequeue(int type)` where type is which `queue` should be changed and `event` is a pointer to an event. An `enqueue(int type, Event* event)` queues an event pointer into the last position in the queue while `dequeue(int type)` removes and returns the first element in the list. A queue is always be locked with it's specified mutex lock before a queue method is used.

3.1.2 Mutex locks and Condition Variables

The synchronization in the program is done by using mutex locks in conjunction with condition variables. Whenever global structures or variables are used, they must always be locked and each global structure has it's own assigned lock that can be used.

For synchronization between threads (in most cases) where a thread can only continue if a condition is true, condition variables are used. The waiting thread will try to acquire a lock and if it succeeds, it will check if the condition is met. If the condition is not met, the thread will wait on the corresponding condition variable and lock. The signalling thread will then have to acquire the lock, set the condition, send the condition signal and finally release the lock. The waiting thread will then be woken up and if it successfully acquires the lock, it will be allowed to execute.

3.2 Event Listener

This particular thread listens to incoming events and `enqueue` them onto an event queue. There are multiple event queues that are used for different event types and the listener has to filter the incoming event before an `enqueue()` . The listener also checks if there are any events of the same time already running, if there are, the event will be ignored instead of en-queued. The listener will also run through the event lists to check if a received event is already in the queue to avoid button spamming. Before an event is received, the event listener must allocate space for an incoming event using `malloc(size)`.

3.3 Event Handlers

There are different kinds of buttons on that can be pressed on the elevator, buttons inside the cabin and buttons outside the cabin. These two types of buttons serve different purposes and are thus handled differently. Not only button clicks serve as events, in the elevator application, position movement by an elevator also registered a new event. Because of this, the program assigns different kind of threads to handle different type of events and they will be gone through step by step.

3.3.1 Floor Button

Each time a (non-duplicate) floor button event is received, it is pushed into a floor button event queue by the listener. As long as the event queue is not empty, the floor button thread will try to assign a new even to an available elevator. Of course, if the event queue is empty, the event handler will wait for a signal from the listener that will notify that the queue is no longer empty. The floor button handler look for available elevators and delegate an event to the most cost-efficient one (more on that in the algorithm section). The event is assigned by locking a shared pointer and pointing it to the newly `dequeued()` event, setting a boolean to true, sending a condition variable signal to the chosen elevator and then unlocking the mutex lock.

3.3.2 Cabin Button

When a cabin button is pushed, it will be placed on a queue of events only for that specific cabin. When there is something in the queue, the event will be taken care of a functions that delegates cabin button events to a specific cabin. If the cabin is busy; once it arrives at its destination the priority of the pushed cabin button mentioned before will begin to execute. If, however, the cabin is not busy it will listen to the event and act accordingly (execute the event). A cabin button will almost always have priority over a floor button except when the cost of the elevator in relation to the other elevators is less, hence, when the direction of the cabin is the same as the direction of the pushed floor button.

3.3.3 Position Event

A position event occurs whenever an elevator moves position, i.e whenever the remote method `handleMotor(N,o)` is invoked. Each time the method is invoked, a new event is received on the event listener that will forward the event through the queue onto the position event handler thread. The position event carries the new position of the cabin that moved and will write that position to the event pointer of the specific elevator. Since the elevator thread repeatedly reads that value, the operation must be done under mutual exclusion using locks. When the position of the elevator has been changed, the position event handler will send a condition signal to the elevator thread that will be waiting for it's position to be updated before calling `handleMotor(N,o)` again.

There are however other invocations of the position events. For some reason, the `handleDoor(N,o)` method creates new position events, but only when the status of the door is actually changed. To counter this, boolean values representing the status of the doors had to be created and checked every time a `handleDoor` method was used and every time a position event was received. When the door status is zero, the position event is an event for moving the elevator, otherwise it sent a condition signal to close the door on the elevator.

3.4 Elevator Cabins

Each cabin is a thread. The cabin thread is synchronized to each event handler and will wait for an event before waking up. The thread will first check which type of event it is processing and change it's variables accordingly. It will first try to close the door and start moving. If the door wasn't closed before, the thread will have to wait for the position event to be processed before it is allowed to continue. Once it has received a signal to continue from the process event handler, it will start moving the elevator. The elevator will keep moving until the position is close enough (≤ 0.04) to the targeted floor. Each time the position moves (`handleMotor`) it will wait for the event to be processed by the position event handler which will update it's position. Of course, all operations using the global event structure which hold the position value must be mutually exclusive. The doors will then be closed and the elevator will wait for the event to be processed by position event handler once again. Once it has received it's final signal from the event handler, it will signal that it is done running, decrement the counter for amount of busy elevators and it will signal the event handlers to tell them that an elevator is now free.

3.5 Algorithm

3.5.1 Floor Button Events

When a floor button attempts to find an elevator and all cabin queues are empty, it will first look for elevators on the same floor as where the person who clicked the button is on. If there are none on the same floor, it will look for the closest available elevator. If there are none, it will wait until one is available.

1. Same Floor

2. Closest Availalbe Elevator

3. Wait for available elevator

3.5.2 Cabin Button Events

When a cabin button is clicked, before running the cabin event, the event handler will look for any floor button events that may be going the same direction and that are in between the target destination and the current destination of the elevator. If this is true, it will let the floor button event run first, and then run the cabin button event afterwards. This way, both the person pushing the floor button and the person pushing the cabin button will arrive at their destination without any detours (except for stopping at the floor button floor).

4 Conclusion

This project was very difficult to debug and it was difficult to understand all remote methods. For example, there was a bug in the program where a position

event that occurred after closing a door was not handled properly because we did not know that closing a door could create new position events because it does not at all affect the position variable of an elevator. Most of the time there was an error in the program, it was because we had misunderstood the remote methods, not because of the synchronization and we believe that there should be much clearer descriptions of the methods than there currently are in the README in the hwAPI folder.

Github for the source code: https://github.com/parallellboyz/GreenElevator