Elevator Simulation Using Threads

BSCSSE

Instructor:
Doc. Hadji Tejuco

Submitted By:

Von Florendo
Justin Duane Cleofas

Date Submitted:
27/01/2025

Introduction

Purpose:

This project simulates a multithreaded elevator system to model realistic elevator behavior, with an emphasis on efficient floor request handling, direction control, and load management. The significance stems from its use as a teaching tool for multithreaded programming and synchronization concepts in C++.

Objectives

- Simulate the behavior of an elevator moving between floors in a building.
- Implement a passenger limit and dynamic sorting of floor requests based on direction going up or down.
- Ensure proper synchronization between threads representing user requests and elevator operations.
- Optimize system responsiveness and accuracy in handling requests.

Scope

The project includes a simulation of a single elevator system with a capacity limit that services floors 1-9. Implemented features include floor request sorting, direction control, and passenger management. This project's scope excludes advanced concepts such as multi-elevator coordination, operations scheduling, and implementation of logic to call the elevator to the middle of the building if it has been idle for too long and hasn't moved in a direction.

Project Overview

Traditional elevator systems face issues such as inefficient floor request handling and the inability to manage multiple requests effectively. This project addresses these issues using a multithreaded simulation, which provides insights into synchronization and task prioritization.

Key Features

- Dynamic floor request queue sorted by direction depending on going up or down in ascending or descending order.
- Direction changes logic at boundary floors (e.g., switching from up to down at the top floor).
- Passenger capacity management for load limit simulation.
- Thread synchronization for handling user inputs and elevator operations simultaneously.

Requirement Analysis

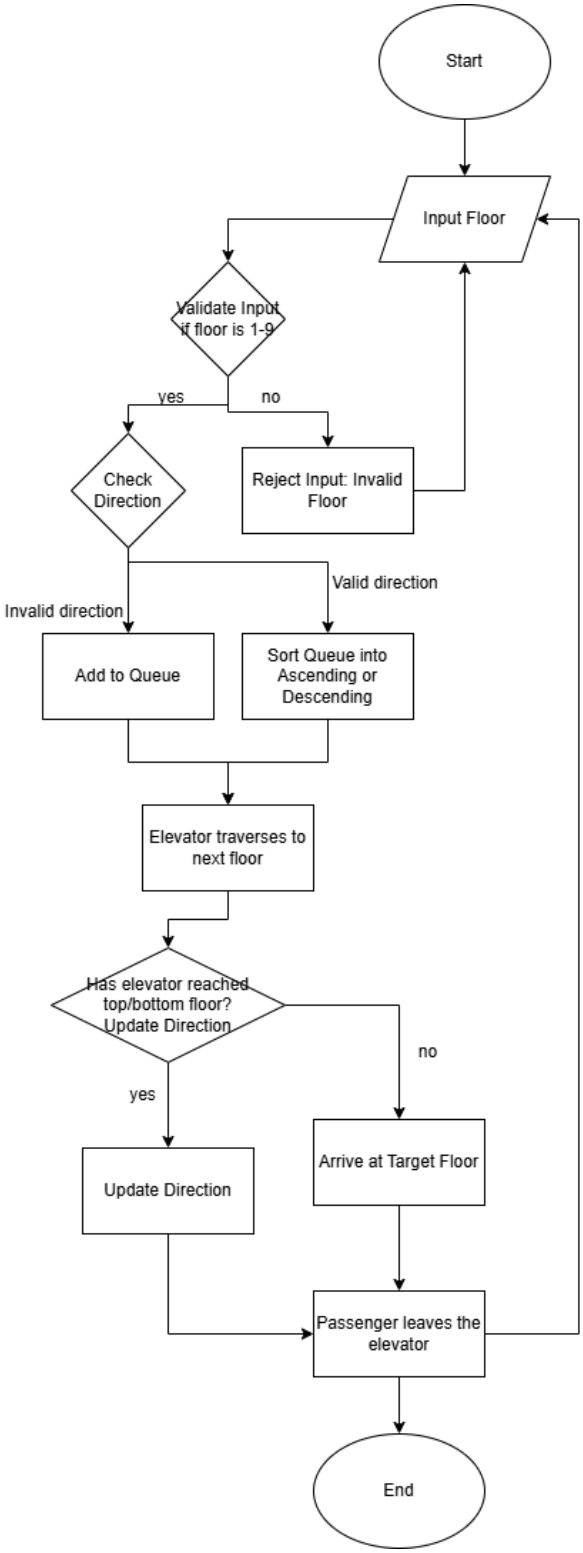
Functional requirements:

- Allow users to request floors dynamically.
- Simulate elevator movement between floors using time delays based on distance.
- Handle requests in a systematic manner based on the elevator's current direction.
- Please provide feedback on passenger entry, exit, and current load.

Non-functional Requirements:

- Performance: The system should be able to handle multiple inputs simultaneously while responding quickly.
- Usability: Offer clear prompts and feedback for user inputs and elevator status updates.
- Security: Avoid invalid inputs (such as out-of-range floors and duplicate floor entries).

System Design



Implementation

Technologies Used:

Programming Language:

- C++.

Libraries:

- Threads - for multithreading
- Mutex - for thread synchronization
- Chronos - for delay simulation between travelling of floors
- Algorithm - for sorting the user input
- Vector - for temporary storage of sorted queue
- Queue - for storing of user floor input

Screenshots:

Code:

```
#include <iostream>
#include <thread>
#include <chrono>
#include <mutex>
#include <queue>
#include <algorithm>
#include <vector>

/*
    Logic Explanation:
    elev starts at ground floor 1 going up to 9th floor, has a passenger limit of <depending on const variable>.


    elev will first sort the queue into ascending order when entered by the user.
    example: 2 passengers entered. 1st passenger wants to go floor 8 but 2nd passenger presses floor 2

    once the elev reaches 9th floor, the elev will now go down. with the sorting now being descending order

    the elev will ignore passed floors (example: you entered 2nd floor going up when the elev is at 4th floor)
    elev will also ignore duplicate floors (why enter floor to only leave at the same one?)
    the elev's threads will wait until the elev reaches a floor (time remaining is current floor between next floor)

*/

using namespace std;

mutex mtx;
int currFloor = 1;
queue<int> requestFloor;
bool running = true;
bool goingUp = true;
int passengers = 0;
const int passengerLoadLimit = 3; // load limit

// Elevator functions
void elevator();
void requestFloors();

// Sorting functions
void sortQueue(queue<int> &requestFloor, int currFloor, bool goingUp);

int main() {
    cout << "Max elevator passenger is "<<passengerLoadLimit<<endl;
    cout << "Starting at floor 1 / ground floor\n";
    cout << "Enter -1 to quit\n";

    thread elev_t(elevator);
```

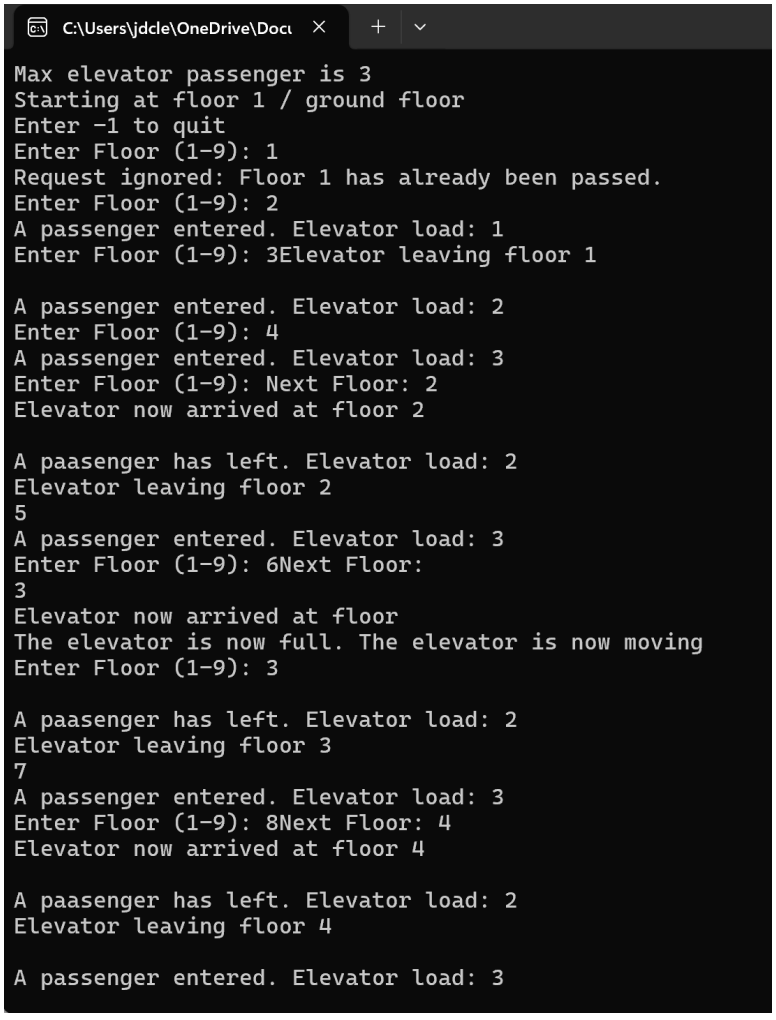
```cpp
using namespace std;

mutex mtx;
int currFloor = 1;
queue<int> requestFloor;
bool running = true;
bool goingUp = true;
int passengers = 0;
const int passengerLoadLimit = 3; // load limit

// Elevator functions
void elevator();
void requestFloors();

// Sorting functions
void sortQueue(queue<int> &requestFloor, int currFloor, bool goingUp);

int main() {
    cout << "Max elevator passenger is "<<passengerLoadLimit<<endl;
    cout << "Starting at floor 1 / ground floor\n";
    cout << "Enter -1 to quit\n";

    thread elev_t(elevator);
    thread request_t(requestFloors);
    elev_t.join();
    request_t.join();

    cout << "\nElevator simulation done.\n";
    return 0;
}

//function to request floors [START]
void requestFloors() {
    int i = 1;
    int floor = 0;
    do {
        cout << "Enter Floor (1-9): ";
        cin >> floor;

        if (floor == -1) {
            cout << "\nThe elevator is now shutting down\n";
            running = false;
            break;
        }

        if (floor < 1 || floor > 9) {
            cout << "\nInvalid floor request. Must be 1-9\n";
```

```
//function to request floors [START]
void requestFloors() {
    int i = 1;
    int floor = 0;
    do {
        cout << "Enter Floor (1-9): ";
        cin >> floor;

        if (floor == -1) {
            cout << "\nThe elevator is now shutting down\n";
            running = false;
            break;
        }

        if (floor < 1 || floor > 9) {
            cout << "\nInvalid floor request. Must be 1-9\n";
            continue;
        }

        mtx.lock();
        if ((goingUp && floor > currFloor) || (!goingUp && floor < currFloor)) { //conditional statement determining if floor is going up or down
            if (passengers < passengerLoadLimit) { //lets say elevator has a load limit
                requestFloor.push(floor);
                passengers++; //means someone has entered the elevator
                sortQueue(requestFloor, currFloor, goingUp);
                cout << "A passenger entered. Elevator load: " << passengers << endl;
            } else {
                cout << "\nThe elevator is now full. The elevator is now moving\n";
            }
        } else {
            cout << "Request ignored: Floor " <<floor<< " has already been passed.\n";
        }
        mtx.unlock();

    } while (running);
}

//function to simulate elevator
void elevator() {
    while (true) {
        mtx.lock();
        if (!running && requestFloor.empty()) {
            mtx.unlock();
            break;
        }
```

```
void elevator() {
    while (true) {
        mtx.lock();
        if (!running && requestFloor.empty()) {
            mtx.unlock();
            break;
        }

        if (!requestFloor.empty()) {
            int nextFloor = requestFloor.front();
            requestFloor.pop();

            if ((goingUp && nextFloor <= currFloor) || (!goingUp && nextFloor >= currFloor)) {
                mtx.unlock();
                continue;
            }

            mtx.unlock();

            if (currFloor != nextFloor) {
                cout<<"Elevator leaving floor "<<currFloor<<endl;
                this_thread::sleep_for(chrono::seconds(abs(currFloor - nextFloor))); //wait between floors

                currFloor = nextFloor;

                cout<<"Next Floor: "<<currFloor<<endl;
            }

            cout<<"Elevator now arrived at floor " << currFloor << endl;

            mtx.lock();
            passengers--;
            cout<<"\nA paasenger has left. Elevator load: "<<passengers<<endl;

            //determines whether floor reaches floor 1 or 9 to change direction going up or down
            if (currFloor == 9) {
                goingUp = false;
                cout << "\nThe elevator is now going down\n";
            } else if (currFloor == 1) {
                goingUp = true;
                cout << "\nThe elevator is now going up\n";
            }
            mtx.unlock();

        } else {
            mtx.unlock();
            this_thread::sleep_for(chrono::milliseconds(500));
```

```cpp
void sortQueue(queue<int> &requestFloor, int currFloor, bool goingUp) {
    vector<int> floors; // create temporary vector for sorting
    while (!requestFloor.empty()) {
        floors.push_back(requestFloor.front());
        requestFloor.pop();
    }

    if (goingUp) {
        sort(floors.begin(), floors.end()); // sort ascending order
    } else {
        sort(floors.rbegin(), floors.rend()); // sort descending order
    }

    for (int floor : floors) {
        requestFloor.push(floor); //push temporary vector elements into requestFloor queue after sorting
    }
}
```

Sample Output:



Testing

Test Case 1: Basic Elevator Operation

Input:

- User requests floors in the following sequence: 3, 5, 2, and 9.
- Elevator starts at floor 1.

Expected Output:

- The elevator moves to floor 2, then 3, then 5.
- The elevator moves to floor 9, changes direction, and begins moving down.

Actual Output:

- Matches the expected output. Requests were handled correctly, and direction changed at the 9th floor.

Test Case 2: Passenger Limit Test

Input:

- User requests floors with 4 passengers: 2, 4, 6, and 8.
- Passenger load limit: 3.

Expected Output:

- Floors 2, 4, and 6 are accepted.
- Floor 8 is rejected with a message: "The elevator is now full. The elevator is now moving."

Actual Output:

- Matches the expected output. Capacity handling and error message displayed correctly.

Test Case 3: Invalid Inputs

Input:

- User enters invalid floor requests: -3, 10, and 0.
- A valid floor (5) is entered after invalid inputs.

Expected Output:

- Requests for -3, 10, and 0 are rejected with the message: "Invalid floor request. Must be 1-9."
- Floor 5 is queued and accepted by the elevator.

Actual Output:

- Matches the expected output. Invalid inputs were rejected, and valid input was accepted.

Results:

All test cases were successfully completed, demonstrating proper handling of floor requests, passenger limits, and invalid input scenarios. The elevator system performed as expected, meeting the functional requirements.

User Manual

Using the github repo link, look for the folder of parallel_comp_act1.cpp in the main branch. Copy and Paste the code in the DevC++ compiler or onlinegdb compiler. For DevC++, compiler option TDM-GCC 10.3.0 and Language standard (-std) must be GNU C++11 selected. Simply compile then run. For user inputs, adhere to the instructions by only entering numbers between 1-9. Elevator has a passenger limit, but can be altered for your preference by changing the const int variable in the IDE.

Challenges and Solutions

Challenges:

- Implementation of sorting algorithm depending on direction
- Implementation of Passenger limit
- Implementation of Direction changing and Thread floor logic

Solutions:

- Solved the sorting algorithm by using the Algorithm library instead of implementing a user-defined algorithm. Then adapted the function into checking the validity of changing direction instead before sorting in ascending or descending order.
- Solved the implementation of the passenger limit by hardcoding the limit into a const int variable. Originally the elevator relied on the length of the queue, which was a mistake on the programmer's part as the function did not run due to the condition statement used as a function to return queue's size and expecting it to

contain element indexes to contain a variable that can be counted using comparison operators.

- Solved the issue of direction changing and floor processing by validating the user input beforehand and where the elevator traverses per input entered. Originally the program faced a problem wherein the floors would somehow stop after reaching the boundary, preventing the user from inputting any more values to represent the floors. Eventually forcing the user to end the program as the only fix. It was resolved by debugging the logic in the thread's function and finding out about the thread logic not exiting properly within a loop.

Future Enhancements

Potential Improvements:

- Support for multiple elevators with expanded logic.
- better visualization of the simulation.
- Logic implementation of a cycle of inputs to simulate flow of passengers entering various floors.
- Implement logic wherein the elevator must service a request in the middle of the building after being idle for too long and not moving in a direction.

Conclusion

The project successfully simulates a multi-threaded elevator system, demonstrating efficient request handling and direction control. It demonstrates synchronization and multithreading concepts in C++. This project provided invaluable insights into managing shared resources, implementing thread-safe operations, and designing responsive systems. It emphasized the importance of balancing functionality, performance, and scalability in multithreaded programming.

References

- https://www.w3schools.com/cpp/cpp_ref_vector.asp
- https://www.geeksforgeeks.org/multithreading-in-cpp/
- https://www.geeksforgeeks.org/std-mutex-in-cpp/
- https://www.geeksforgeeks.org/chrono-in-c/
- https://www.w3schools.com/cpp/cpp_algorithms.asp