Project Title: Elevator Simulation
Course Name: Parallel and Distributed Computing
Instructor: Dr. Hadji Tejuco
Team Members:

- Sarmiento, Sebastian - Programmer
- Javier, Alexandra Jamie - Programmer

Submission Date: January 27, 2025

Table of Contents:

Introduction

Purpose:
The project simulates how an elevator system would work in a building with up to nine levels. The elevator functions by processing upward requests in ascending order of starting floors, followed by downward requests in descending order. Passengers are assigned according to their input. The goal of the program is to enhance understanding of parallel computing topics related to synchronization and multithreading using C++.

Objectives:

- Simulate realistic elevator movement, passenger interaction, and request handling for up and down directions.
- Apply and demonstrate synchronization and multithreading techniques. Showcase order-based request processing to ensure the elevator runs efficiently.

Scope:
The simulation covers:

- A single elevator servicing up to 9 floors.
- Time-based processing of up and down requests.
- Time-based processing of up and down requests.
- Console-based output for elevator movement and passenger handling.

Out of scope:

- Multiple elevators or dynamic capacity adjustment.
- Integration with external hardware or sensors.

Project Overview

Problem Statement:
It's important to handle elevator requests in a multi-story structure effectively while maintaining capacity and direction constraints. The problem of imitating such a process in a controlled software environment is addressed in this project.

Key Features:

- User input for generating elevator requests.
- Real-time synchronization using threads and condition variables.
- Directional movement handling for up and down requests.
- Capacity management for a maximum of 3 passengers.

Requirements Analysis

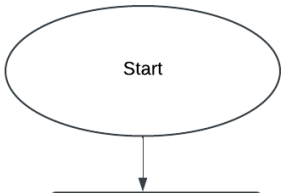
Functional Requirements:

- Receive and validate user inputs for specifying starting and destination floors.
- Efficiently handle requests for upward and downward travel directions based on floor sequence.
- Emulate real-time elevator functionality using multithreading for synchronized operations.

Non-Functional Requirements:

- Performance: Ensure timely processing of up to 3 simultaneous requests.
- Usability: Provide clear and user-friendly input validation and feedback.
- Scalability: Design the system to accommodate additional floors and requests with minimal modifications.

System Design



Implementation

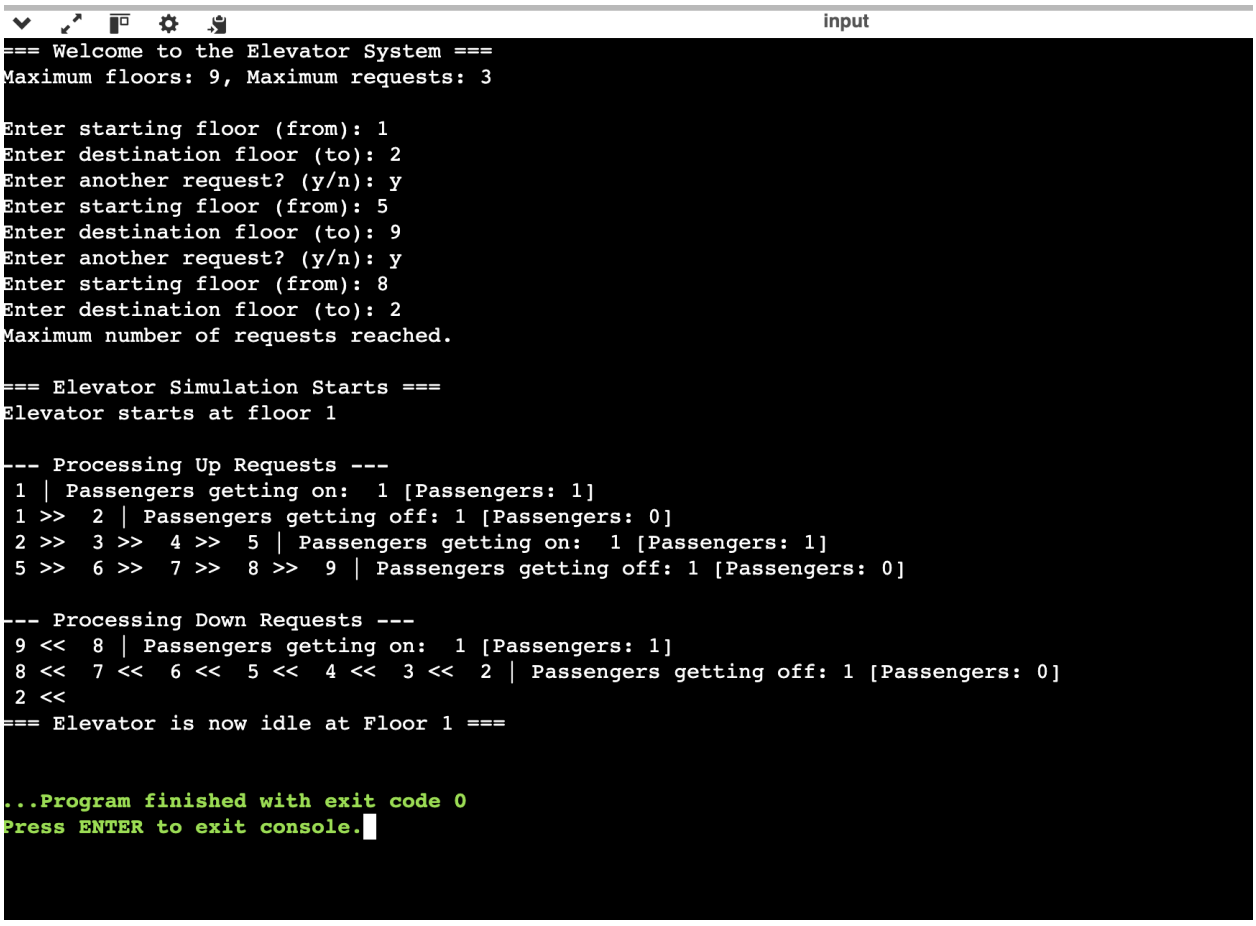
Technologies Used:

- Programming Language: C++
- Libraries: iostream, vector, algorithm, map, thread, mutex, condition_variable, iomanip, limits

Code Highlights:

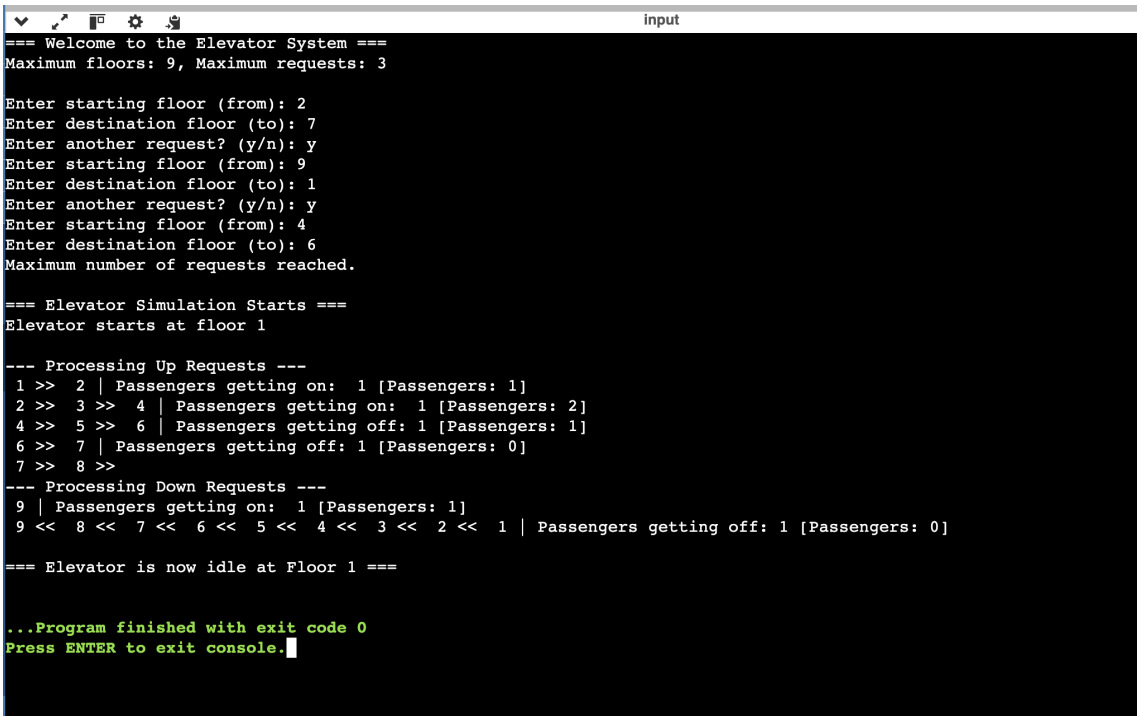
- Multithreading with std::thread.
- Synchronization using std::mutex and std::condition_variable
- Input validation to ensure correct request generation.

Sample Output:

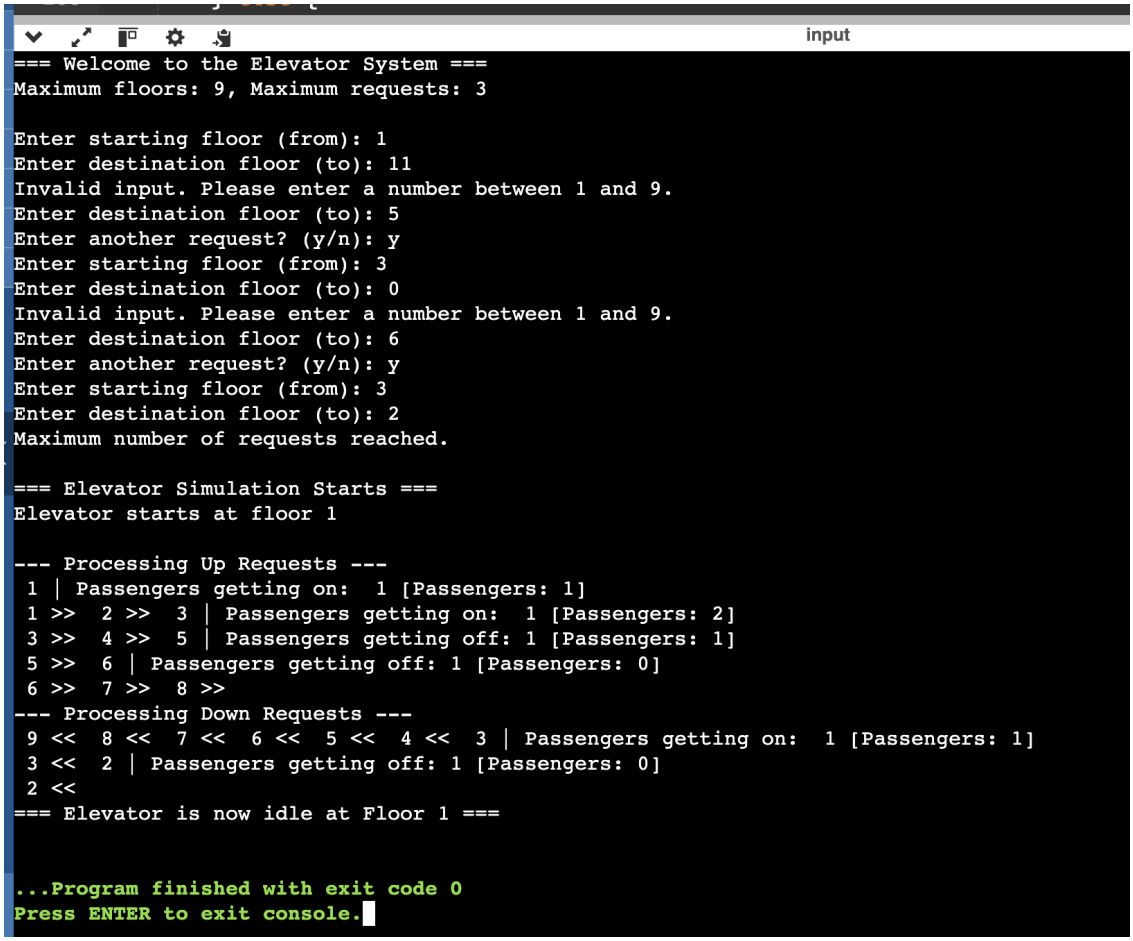


Testing

Test Cases:



2.



Results: All test cases were successfully executed, ensuring robust input validation and synchronization.

User Manual

Installation Guide:

1. Install a C++ compiler.
2. Copy the source code into a .cpp file.
3. Compile and run the program.

Usage Instructions:

1. Run the program.
2. Enter floor requests as prompted.
3. Observe the elevator's simulation.

Challenges and Solutions

Challenges Faced:

- avoiding deadlocks by synchronizing thread execution.
- ensuring strong validation of input.

Solutions Implemented:

- For controlled thread signaling, std::condition_variable was utilized.
- Loop-based input checks were put in place to stop invalid data.

Future Enhancements

Potential Improvements:

- Support for multiple elevators.
- Dynamic capacity adjustment based on building specifications.

Conclusion

Summary:
The following features of an elevator system are simulated in this project:

- It handles user requests to switch floors within a predetermined range.
- Requests are guaranteed to be valid by input validation (e.g., within floor limitations, not the same floor for start and destination).
- The elevator operates in two directions:
    - Upward requests are processed sequentially by sorting in ascending order.
    - Downward requests are processed sequentially by sorting in descending order.
- The system maintains a maximum passenger capacity while controlling the boarding and alighting of passengers at each floor.
- The elevator returns to an idle state at the first floor after fulfilling all requests.

In order to manage synchronization and provide seamless elevator movement simulation and requests handling, the project skillfully combines multithreading with mutex locks and condition variables.

Lessons Learned:

The use of std::mutex and std::condition_variable for thread safety, strong input validation to avoid errors, and data structures like std::vector and sorting algorithms for effective request handling are some of the most important lessons learnt. Reliability of the system depended on handling edge circumstances, such as incorrect inputs or overcapacity requests. The code became more manageable and easier to understand and debug when the system was divided into modular parts.

References

- https://www.geeksforgeeks.org/multithreading-in-cpp/
- https://www.javatpoint.com/multithreading-in-cpp-with-examples

---

Appendices

Appendix A: flowchart
Appendix B: Complete Code

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <iomanip>
#include <limits>
using namespace std;
const int MAX_FLOORS = 9;
const int MAX_REQUESTS = 3;
const int MAX_CAPACITY = 3;
struct Request {
    int from;
```

```
    int to;
};
mutex mtx;
condition_variable cv;
bool processing = false;
vector<Request> upRequests;
vector<Request> downRequests;
void simulateElevator() {
    unique_lock<mutex> lock(mtx);
    cv.wait(lock, [] { return processing; });
    int currentFloor = 1;
    int currentPassengers = 0;
    cout << "\n=== Elevator Simulation Starts ===\n";
    cout << "Elevator starts at floor 1\n";
    auto printPassengerCount = [&]() {
        cout << " [Passengers: " << currentPassengers << "]\n";
    };
    auto handleFloorActions = [&](map<int, int>& onFloors, map<int, int>& offFloors, int targetFloor) {
        // Handle passengers getting off
        if (offFloors[targetFloor] > 0) {
            int alighting = min(offFloors[targetFloor], currentPassengers);
            currentPassengers -= alighting;
            cout << setw(2) << targetFloor << " | Passengers getting off: " << alighting;
            printPassengerCount();
        }
        // Handle passengers getting on
        if (onFloors[targetFloor] > 0) {
            int boarding = min(onFloors[targetFloor], MAX_CAPACITY - currentPassengers);
            if (boarding > 0) {
                currentPassengers += boarding;
                cout << setw(2) << targetFloor << " | Passengers getting on:  " << boarding;
                printPassengerCount();
            }
        }
    };
    // Process up requests
    if (!upRequests.empty()) {
        sort(upRequests.begin(), upRequests.end(), [](const Request& a, const Request& b) {
            return a.from < b.from;
        });
        map<int, int> onFloors, offFloors;
        for (const auto& request : upRequests) {
            onFloors[request.from]++;
            offFloors[request.to]++;
        }
        cout << "\n--- Processing Up Requests ---\n";
        while (currentFloor <= MAX_FLOORS) {
            handleFloorActions(onFloors, offFloors, currentFloor);
            if (currentFloor == MAX_FLOORS) break; // Stop if the elevator is at the top floor
            cout << setw(2) << currentFloor << " >> ";
            currentFloor++;
        }
    }
    // Process down requests
```

```
    if (!downRequests.empty()) {
        sort(downRequests.begin(), downRequests.end(), [](const Request& a, const Request& b) {
            return a.from > b.from;
        });
        map<int, int> onFloors, offFloors;
        for (const auto& request : downRequests) {
            onFloors[request.from]++;
            offFloors[request.to]++;
        }
        cout << "\n--- Processing Down Requests ---\n";
        while (currentFloor >= 1) {
            handleFloorActions(onFloors, offFloors, currentFloor);
            if (currentFloor == 1) break; // Stop if the elevator is at the ground floor
            cout << setw(2) << currentFloor << " << ";
            currentFloor--;
        }
    }
    // Return to floor 1 and go idle
    if (currentFloor != 1) {
        cout << "\n--- Returning to Floor 1 ---\n";
        while (currentFloor > 1) {
            cout << setw(2) << currentFloor << " << ";
            currentFloor--;
        }
        cout << " 1\n";
    }
    cout << "\n=== Elevator is now idle at Floor 1 ===\n";
}
int main() {
    thread elevatorThread(simulateElevator);
    int requestCount = 0;
    cout << "=== Welcome to the Elevator System ===\n";
    cout << "Maximum floors: " << MAX_FLOORS << ", Maximum requests: " << MAX_REQUESTS << "\n\n";
    while (requestCount < MAX_REQUESTS) {
        Request request;
        // Input validation for "from" floor
        while (true) {
            cout << "Enter starting floor (from): ";
            cin >> request.from;
            if (cin.fail() || request.from < 1 || request.from > MAX_FLOORS) {
                cin.clear(); // Clear the error flag
                cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Discard invalid input
                cout << "Invalid input. Please enter a number between 1 and " << MAX_FLOORS << ".\n";
            } else {
                break;
            }
        }
        // Input validation for "to" floor
        while (true) {
            cout << "Enter destination floor (to): ";
            cin >> request.to;
            if (cin.fail() || request.to < 1 || request.to > MAX_FLOORS || request.from == request.to) {
                cin.clear(); // Clear the error flag
                cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Discard invalid input
```

```
            if (request.from == request.to) {
                cout << "Invalid input. Starting and destination floors cannot be the same.\n";
            } else {
                cout << "Invalid input. Please enter a number between 1 and " << MAX_FLOORS << ".\n";
            }
        } else {
            break;
        }
    }
    // Add the request
    if (request.from < request.to) {
        upRequests.push_back(request);
    } else {
        downRequests.push_back(request);
    }
    requestCount++;
    if (requestCount < MAX_REQUESTS) {
        cout << "Enter another request? (y/n): ";
        char moreRequests;
        cin >> moreRequests;
        if (moreRequests == 'n' || moreRequests == 'N') break;
    }
}
if (requestCount == MAX_REQUESTS) {
    cout << "Maximum number of requests reached.\n";
}
{
    lock_guard<mutex> lock(mtx);
    processing = true;
}
cv.notify_one();
elevatorThread.join();
return 0;
}
```