Final Project
Elevator Operating System

Problem

For this final project, you will write a C/C++ program that will act as the scheduler for an Elevator Operating System. While the OS is running on the same machine as your scheduler, the core functionality as well as the simulation environment are written entirely in Python. As such, data communications between the scheduler and the OS will be handled via asynchronous API calls.

Your solution should accept the building configuration data file as a required command line argument and display an appropriate error message if the argument is not provided or the file does not exist. The command to run your application will look something like this:

Form:	scheduler_os <path_to_building_file>
Example:	scheduler_os simple.bldg

Major Requirements

Your solution is required to meet the following to be considered correct. Failure to meet all the requirements could lead to a reduction in points to your "Correction Score" regardless of how correct the final output is.

Multithreading

Your solution must make use of at least 3 concurrently running threads along with any necessary protection from race conditions. You must have **at minimum** the following three threads running concurrently:

- Input Communication Thread
  - This thread is used to communicate with the /NextInput API.
  - You may want to insert a small wait either after every attempt or everytime you get a NONE response to keep from weighing down the main OS with too many data communication attempts.
- Output Communication Thread
  - This thread is used to communicate with /AddPersonToElevator or /AddPersonToElevator_A3 APIs.
- Scheduler Computation Thread
  - This thread is used to make decisions about when and where to assign the people coming from /NextInput.
  - This thread should be pulling data from the Input Communication Thread, making decisions, then pushing data to the Output Communication Thread.

Testing Requirements

Your solution must be able to compile using GNU version 5.4.0 running in the HPCC Quanah cluster. Your solution must also meet the following additional requirements:

- Your solution will be tested in the HPCC using a bash script to compile your code, execute the Python Conda environment for the primary OS, and then run your solution against one or more test cases.
- Your solution must be capable of running within the confines of a single Quanah node, this means the final grading script will request the following resources for your job, these should be considered your maximum limits when determining how to write your solution:
    - 5-hour maximum runtime
    - 36 cores
    - 192 GB of memory
- Keep in mind that my Python OS will consume 2 threads and upwards of 1 GB of memory.

Timing Requirements

Your solution must be capable of completing the entire simulation in the required runtime (which may be different that what SBATCH requests). Each solution will have a maximum runtime determined by me running a simplistic (dumb) scheduling algorithm through the test and then your solution must not run any slower than 10% over my algorithm.

Timing examples:

- If my code runs the algorithm in 200 timesteps, then your code must complete it within 220 timesteps.
- If my code runs the algorithm in 462 timesteps, then your code must complete it within 509 timesteps.

README Requirements

Your solution must contain a README document that describes the scheduling algorithm you chose, why you chose it over other possible options, and what (if any) improvements you think you could have made had you optimized your algorithm. This file can be submitted as a Word document, a text document (standard ASCII, Unicode, or Markdown), or a PDF.

Group Requirements

For those who opt to work in a group of up to four members, you will have 2 additional requirements:

1) A weekly summary report as described in class.
    a. The specific template to use will be emailed to Project Managers after Tuesday, April 9.
2) Your solution must be constructed within GitHub to make it clear who worked on which portions of the code.
    a. The weekly summary will help explain discrepancies with multiple people work together but only one person is submitting the code.

Output Rules

Your code can print whatever you want it to print to the screen – the grading will be done based on the report files generated by my Elevator OS which is controlled through your API calls. As such, printing to the screen is purely to be used for your debugging purposes and will be ignored during grading.

Elevator OS Building Configuration File (.bldg) Specification

File Format

- The file is a plain text ASCII file with a .bldg extension.
- Data is organized in a tab-delimited format.
- Each line represents a single elevator bay's configuration.

Line Format

Each line contains the following fields, in order, separated by tabs (\t):

1. Elevator Bay Name:
    a. Type: Alphanumeric String
    b. Description: A unique identifier for each elevator bay within the building.
    c. Restrictions: Must not contain tabs. Should be concise yet descriptive.
2. Lowest Floor:
    a. Type: Integer
    b. Description: The lowest floor number to which the elevator travels.
    c. Restrictions: Can be negative (for basements), zero, or positive.
3. Highest Floor:
    a. Type: Integer
    b. Description: The highest floor number to which the elevator travels.
    c. Restrictions: Must be greater than the lowest floor.
4. Current Floor:
    a. Type: Integer
    b. Description: The starting floor number for the elevator when the simulation begins.
    c. Restrictions: Must be between the lowest and highest floors, inclusive.
5. Total Capacity:
    a. Type: Positive Non-Zero Integer
    b. Description: The maximum number of people that can be on board the elevator simultaneously.
    c. Restrictions: Must be a positive integer, reflecting the capacity for humans.

API Specification

Overview

This API controls and monitors an elevator simulation. It provides endpoints to start or stop the simulation, check elevator and simulation status, process inputs, and assign people to elevators.

Base URL

The base URL for the API is not explicitly defined in the provided code and will depend on where the service is hosted. For development, it typically defaults to http://localhost:5432.

Endpoints

Start or Stop Simulation

- **URL**: /Simulation/<string:command>
- **Method**: GET | PUT
- **URL Params**: command can be either "start" or "stop" for PUT, command should be "check" for GET.
- **Success Response**:
  - **Code**: 202 (Accepted) | 200 (OK)
  - **Content**: "Simulation started" or "Simulation is already running." or "Simulation stopping"
- **Error Response**:
  - **Code**: 400 (Bad Request)
  - **Content**: "Invalid command"
- **Notes**: For GET, returns the current status of the simulation.

Elevator Status

- **URL**: /ElevatorStatus/<string:elevatorID>
- **Method**: GET
- **URL Params**: elevatorID is the identifier of the elevator.
- **Success Response**:
  - **Code**: 200 (OK)
  - **Content**: Elevator status as a string.
  - **Content Format**: <bayID> | <currentFloor> | <directionString> | <passengerCount> | <remaingingCapacity>
  - **Direction String**: S, U, D, or E (Stationary, Up, Down, Error)
- **Error Response**:
  - **Code**: 400 (Bad Request)
  - **Content**: "Simulation is not running." | "DNE"

Next Input

- **URL**: /NextInput
- **Method**: GET
- **Success Response**:
  - **Code**: 200 (OK)
  - **Content**: Next person in queue's data or "NONE".
  - **Content Format**: <personID> | <startFloor> | <endFloor>
- **Error Response**:
  - **Code**: 400 (Bad Request)
  - **Content**: "Simulation is not running."

Add Person to Elevator

- **URL**: /AddPersonToElevator/<string:personID>/<string:elevatorID>
- **Method**: PUT
- **URL Params**:
  - personID: Identifier of the person.
  - elevatorID: Identifier of the elevator.
- **Success Response**:
  - **Code**: 200 (OK)
  - **Content**: Confirmation message.
- **Error Response**:
  - **Code**: 400 (Bad Request)
  - **Content**: "Simulation is not running."

Legacy Endpoints (Old Style API)

These endpoints behave similarly to the updated ones but are tailored to work like those encountered in Assignment #3.

- **Simulation Control**: /Simulation_A3
- **Elevator Status**: /ElevatorStatus_A3
- **Add Person to Elevator**: /AddPersonToElevator_A3

The request data format and responses for these endpoints are similar to their newer versions, with the exception that data is expected in the request body for PUT operations, and the personID and elevatorID for adding a person to an elevator are expected to be delimited by a | in the request body for the old-style API.

API Notes

- This API is designed for internal use in an elevator simulation environment.
- Responses are primarily plain text for simplicity.
- The simulation must be started with a PUT request to /Simulation/start before other actions can be taken.
- A call to /Simulation/stop will abort all work being performed and terminate the OS. Only use this API for testing – do not use it in your final submission.

What to turn in to BlackBoard

A zip archive (.zip) named FinalProject.zip that contains the following files:

- Your C / C++ source code
    - You can select any names you want, your makefile will dictate the executable name.
- `makefile`
    - The makefile required to compile your application – ensure your final executable is called *scheduler_os.*