AirControlX: Design and Implementation Report

Operating Systems Project

Muhammad Fatik Bin Imran (23I-0655)

Muhammad Kaleem Akhtar (23I-0524)

Section: BCS-4C

May 7, 2025

1 Introduction

The AirControlX project is an automated air traffic control system developed as part of an Operating Systems course. Designed to simulate real-world air traffic management, it handles aircraft scheduling, runway assignments, violation detection, and fine management using C++ and the SFML library for graphical visualization. This report outlines the design choices and implementation details, emphasizing the operating system concepts utilized, such as multithreading, synchronization, and priority-based scheduling.

2 Design Choices

The design of AirControlX was driven by the need for modularity, scalability, and real-time responsiveness. Below are the key design decisions:

2.1 Object-Oriented Architecture

- **Rationale**: An object-oriented approach was chosen to model real-world entities like aircraft, runways, and airlines, ensuring code reusability and maintainability.
- **Implementation**: Classes such as `Aircraft`, `Runway`, `Airline`, and `AVN` encapsulate data and behavior. For example, `Aircraft` includes attributes like `flightNumber`, `phase`, and `priority`, with methods to update status and simulate movement.
- **Benefit**: This modular structure allows easy extension, such as adding new aircraft types or flight phases, without altering core logic.

2.2 Priority-Based Scheduling

- **Rationale**: To mimic real-world air traffic control, aircraft are prioritized based on emergency status, fuel levels, and aging (time spent waiting).
- **Implementation**: A custom `ComparePriority` functor is used with `priority_queue` to order aircraft by priority (`EMERGENCY_PRIORITY`, `HIGH_PRIORITY`, `NORMAL_PRIORITY`). The `rebuildQueue` function dynamically adjusts queue order when priorities change, such as upgrading `NORMAL_PRIORITY` to `HIGH_PRIORITY` after an `AGING_THRESHOLD` of 30 seconds.
- **Benefit**: Ensures critical flights (e.g., emergencies) are processed first, reducing delays and enhancing safety.

2.3 Multithreading and Synchronization

- **Rationale**: Real-time simulation requires concurrent execution of simulation steps, user input handling, and visualization updates.
- **Implementation**: A dedicated `simulationThread` runs the `runSimulation` function, updating aircraft states every `TIME_STEP` (1 second). Mutexes (`coutMutex`, `simulationMutex`, `avnListMutex`) prevent race conditions when accessing shared resources like `avnList` or console output. For instance, `coutMutex` ensures thread-safe console logging.

- **Benefit**: Enables smooth simulation execution while allowing user interaction (e.g., pausing or spawning aircraft) without data corruption.

2.4 Runway Management

- **Rationale**: Efficient runway allocation is critical to avoid bottlenecks and ensure fair access.
- **Implementation**: Three runways (`RWY_A`, `RWY_B`, `RWY_C`) are designated for specific operations: `RWY_A` for arrivals, `RWY_B` for departures, and `RWY_C` for cargo, emergencies, or overflow. The `assignRunway` function prioritizes runway assignment based on aircraft type and direction, with waiting queues for occupied runways.
- **Benefit**: Balances runway usage, minimizing wait times and prioritizing high-priority flights.

2.5 Violation Detection and Fine System

- **Rationale**: To enforce air traffic regulations, the system detects speed and altitude violations and issues fines.
- **Implementation**: The `checkForViolations` function evaluates aircraft parameters against phase-specific thresholds (e.g., speed ≤ 600 km/h in `HOLDING`). Violations trigger `generateAVN`, creating an `AVN` struct with details like `avnID`, `fineAmount`, and `dueDate`. Fines are calculated with a 15% service fee (`SERVICE_FEE`) and managed via a mock `mockStripePay` function.
- **Benefit**: Provides a realistic penalty system, with an airline portal for viewing and paying fines, enhancing simulation fidelity.

2.6 Graphical Visualization with SFML

- **Rationale**: A visual interface improves user understanding of the simulation state.
- **Implementation**: The `visualizeSimulation` function uses SFML to render runways, aircraft, and status text. Aircraft are color-coded by priority (red for `EMERGENCY_PRIORITY`, yellow for `HIGH_PRIORITY`, blue for `NORMAL_PRIORITY`) and positioned dynamically based on their `phase` and `timeInFlight`.
- **Benefit**: Offers an intuitive view of runway occupancy, aircraft movement, and queue sizes, aiding debugging and user engagement.

2.7 Non-Blocking Input Handling

- **Rationale**: Users need to interact with the simulation (e.g., exit to menu) without interrupting the simulation loop.
- **Implementation**: The `setNonBlockingInput` function configures the terminal for non-blocking input, allowing `kbhit` to detect keypresses (e.g., 'q' to exit). This uses `termios` and `fcntl` for POSIX-compliant input handling.
- **Benefit**: Ensures responsive user interaction while maintaining simulation continuity.

3 Implementation Details

The implementation leverages C++17 features and operating system concepts to achieve a robust simulation. Key aspects include:

3.1 Data Structures

- **`priority_queue`**: Used for `arrivalQueue` and `departureQueue`, ensuring efficient priority-based scheduling with $O(\log n)$ insertion and removal.
- **`map` and `vector`**: `aircraftStatusMap` provides $O(\log n)$ lookup for aircraft status, while `vector` stores `airlines`, `runways`, and `activeAircrafts` for sequential access.
- **`queue`**: Each runway maintains a `waitingQueue` for aircraft awaiting runway access, supporting FIFO ordering within priority levels.

3.2 Simulation Loop

The `runSimulation` function drives the simulation, executing `updateSimulationStep` every `TIME_STEP`. This updates aircraft states, checks violations, and manages runway assignments. The loop terminates after `SIMULATION_DURATION` (300 seconds) or user-initiated cleanup via `cleanupSimulation`, which deallocates resources and resets global variables.

3.3 Input Validation

Robust input validation is implemented using regular expressions (`regex`) and custom functions like `isValidInteger` and `isValidTimeFormat`. For example, flight numbers are restricted to 10 characters, and AVN IDs follow the format `AVN-YYYYMMDD-NNN`. This prevents invalid inputs from disrupting the simulation.

3.4 Error Handling

- **Memory Management**: Dynamic allocation of `Aircraft` objects is carefully managed, with `cleanupSimulation` and `removeAircraftFromEverywhere` ensuring no memory leaks.
- **Fault Handling**: Ground faults are simulated with a 5% probability during `TAXI` or `AT_GATE` Phases, marking aircraft as `isFaulty` and removing them from active simulation.

3.5 Operating System Concepts

- **Threading**: The simulation runs in a separate thread, demonstrating process scheduling and concurrency.
- **Synchronization**: Mutexes ensure thread-safe access to shared resources, addressing critical section problems.
- **Priority Scheduling**: The priority queue implements a preemptive scheduling algorithm, prioritizing high-priority tasks (aircraft).

- **I/O Management**: Non-blocking input handling showcases asynchronous I/O operations, a key OS concept.

4 Challenges and Solutions

- **Challenge**: Synchronizing console output from multiple threads caused garbled text.
- **Solution**: Introduced `coutMutex` to serialize console writes, ensuring clear output.
- **Challenge**: Dynamic priority changes (e.g., due to aging) disrupted queue ordering.
- **Solution**: Implemented `rebuildQueue` to reconstruct the priority queue when priorities change, maintaining correct order.
- **Challenge**: Visualizing aircraft movement smoothly in SFML.
- **Solution**: Used a hash-based positioning algorithm in `visualizeSimulation` to assign unique circular paths for `HOLDING` and `APPROACH` phases, preventing overlap.

5 Conclusion

AirControlX successfully simulates an air traffic control system, integrating operating system concepts like multithreading, synchronization, and priority scheduling. The object-oriented design, coupled with robust data structures and SFML visualization, ensures scalability and user-friendliness. Future enhancements could include network-based airline portals, real-time weather impacts, or advanced scheduling algorithms to further emulate real-world complexities.