Air Traffic Control Simulator

Berke Kurtuldu 76351, Burak Can Sahin 76824,

Beyza Erdogan 76426

Introduction

The purpose of this project is to simulate an air traffic control (ATC) system that manages the take-off and landing of planes using a single runway. The simulator ensures collision-free and deadlock-free operation while preventing starvation of planes either waiting to land or take off. The implementation utilizes POSIX threads (pthreads) to manage multiple planes and the control tower concurrently.

Objectives

1. Part I: Implement a basic ATC simulator with real-time scheduling, favoring landing planes to minimize fuel consumption.

2. Part II: Extend the simulator to prevent starvation of planes waiting on the ground by incorporating a maximum wait-time and counter.

Environment Setup

1. Directories:

- include/: Header files
- src/: Source files
- obj/: Object files
- bin/: Executable files

2. Files:

- include/atc_simulator.h: Header file for the ATC simulator.
- include/pthread_sleep.h: Header file for the pthread sleep function.
- src/atc_simulator.cpp: Main source file for the ATC simulator.
- src/pthread_sleep.cpp: Source file for the pthread sleep function.
- Makefile: Build automation tool.
- README.md: Project documentation.
- planes.log: Log file for plane activities.

Implementation

PART 1

In our main function, we first control the command line parameters to be sure that correct ones are given. Below you can see the -s -p -n flag controls.

```
// Parse command line arguments
for (int i = 1; i < argc; i++) {
    if (strcmp(argv[i], "-s") == 0 && i + 1 < argc) {
        simulationTime = atoi(argv[++i]);
    } else if (strcmp(argv[i], "-p") == 0 && i + 1 < argc) {
        p = atof(argv[++i]);
    } else if (strcmp(argv[i], "-n") == 0 && i + 1 < argc) {
        n = atoi(argv[++i]);
    }
}
```

Then, we create the tower thread, and initial plane threads. We have 3 different functions for threads to run when created. These are towerThread for tower, landingThread for landing planes and departureThread for departing planes. For randomness, we used uniform real distribution. Then we go into for loop which runs until simulation time is finished. We get the simulation time from the command line. In the case of any error, there is a default value for simulation time which is 200, default probability equalized to 0.5 and default log snapshot time equal to 0.

```
int simulationTime = 200;  // Default simulation time
double p = 0.5;            // Default probability
int n = 0;                 // Default log snapshot time
```

To achieve real time simulation we make the master thread sleep for 1 second in every iteration of the for loop. Then for probability p, we created a landing plane. We assign even numbers as IDs to landing planes. And we created departing planes for possibility 1-p. Both landing planes and departing planes are added to a thread vector to efficiently cancel all the threads at the end of the simulation without any data leak. Every n second which is snapshot time, we print the planes on the ground and the planes in the air. We used mutex locks to read the planes properly since at the time of reading there can also be another write process which can cause inconsistency. After the main loop finishes we join all the slaves thread to the master thread, then simulation ends.

```
for (int time = 0; time <= simulationTime; time++) {
    pthread_sleep(1);

    double prob = distribution(generator);
    if (prob < p) {
        int* id = new int(landingID);
        landingID += 2;
        pthread_t plane;
        pthread_create(&plane, NULL, landingThread, id);
        planeThreads.push_back(plane);
    }
    if (prob < 1-p){
        int* id = new int(departureID);
        departureID += 2;
        pthread_t plane;
        pthread_create(&plane, NULL, departureThread, id)
        planeThreads.push_back(plane);
    }

    if (time % n == 0) {
        std::unique_lock<std::mutex> lock(mtx);
        std::cout << "At " << time << " sec ground: ";
        std::queue<int> tempQueue = departureQueue;
        while (!tempQueue.empty()) {
            std::cout << tempQueue.front() << " ";
            tempQueue.pop();
        }
        std::cout << "\nAt " << time << " sec air: ";
        tempQueue = landingQueue;
        while (!tempQueue.empty()) {
            std::cout << tempQueue.front() << " ";
            tempQueue.pop();
        }
        std::cout << std::endl;
    }
}
```

In the towerThread function our main purpose is to manage the air traffic. Tower thread blocks until one of the queues is not empty. Since both of them are empty, then there is nothing to do, so it blocks. In the below screenshot, you can see our air traffic algorithm which will be explained in part 2. But for part 1, the priority is given to the landing planes. If the landingQueue is not empty then we notify landing planes to land. The way we do that is notifying all threads with cv.notify_all() which is a function of control variable cv. Since we call that, all threads are notified, we add another 2 variables, landFree and departFree.They are both initialized to false, each time tower notify threads, if landing planes should be prioritized, landFree turns true, if departing planes should be prioritized, departFree turns true. Then we break the lock and let the thread sleep for 1 second. 1 second is given in the pdf.

```
void* towerThread(void* arg) {
    while (!terminateFlag.load()) {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, []() { return !landingQueue.empty() || !departureQueue.empty(); });

        //We will prioritize landing planes until there are 5 planes waiting to take off on the ground
        //and then we will give a chance for planes to land after 2 consecutive departs.
        //This way we prevent starvation of both landing and departing planes.
        if (!landingQueue.empty() && (consecutiveCounter >= 2 || groundCounter < 5)) {
            landFree = true;
            consecutiveCounter = 0; // Reset ground processed counter
            cv.notify_all();
        } else if (!departureQueue.empty()) {
            departFree = true;
            cv.notify_all();
        }

        lock.unlock();
        pthread_sleep(1);  // Check queues every second
    }
    return NULL;
}
```

We implemented a plane struct for keeping track of the arrival and departure times of the planes correctly. This struct will be used in the departingThread and landingThread functions.

```
//Plane struct to keep times correctly
struct Plane{
        int id;
        char status;
        std::string requestTime;
        int completionTime;
};

// Part 2 starvation prevention
int groundCounter = 0;
int consecutiveCounter = 0;
```

When landing, landingThread function is invoked. Then real time is acquired by the chrono library in cpp. Status of the plane is set to landing is the struct. Then the plane is added to the landing queue. To convert the output given by the chrono library into a better/readable output, we adjusted the output such that it would meet the requirements in the project PDF. Our output looks like real-life departure announcements, displaying the time it was requested and its runway as a HH:MM:SS, and the difference between runway and request is displayed based on the difftime result.

```
int id = *(int*)arg;
Plane *plane = new  Plane;

std::unique_lock<std::mutex> lock(mtx);

auto requestTime = std::chrono::system_clock::now();
time_t requestTimeT = std::chrono::system_clock::to_time_t(requestTime);
std::string timeStr = std::ctime(&requestTimeT);
timeStr.pop_back();
plane->requestTime = timeStr;
plane->status = 'L';

// Notify the tower
landingQueue.push(id);
cv.notify_all();

// Wait for the runway to be free and permission from the tower
cv.wait(lock, [id]() { return terminateFlag.load() || (landFree && runwayFree && landingQueue.front() == id); });
if (terminateFlag.load()) {
    return NULL;
}
```

```
// Wait for the runway to be free and permission from the tower
cv.wait(lock, [id]() { return terminateFlag.load() || (landFree && runwayFree && landingQueue.front() == id); });
if (terminateFlag.load()) {
    return NULL;
}

// Use the runway
runwayFree = false;
lock.unlock();

pthread_sleep(2);  // Simulate time on the runway
```

After being added to the landing queue, tower thread is notified to work since there is a plane in the queue. Before landing, the plane is blocked until both landFree and runwayFree variables are true. runwayFree variable represents the availability of the runway. Then if the id is equal to the plane's id in the front of the landing queue, plane breaks the block and continues to run the code.In this moment the thread sleeps for 2 seconds to simulate the landing takes 2

seconds. After landing time is acquired again and runway time is calculated by subtracting the commit time from the completion time. After completion, the plane is removed from the queue and runwayFree assigned to be true. On the other hand, landFree is assigned to be false, since the tower should manage all the traffic. A plane cannot take control from another plane.

When departing, the departingThread function is invoked. It is the same thing with the landingThread function. There are some other differences that will be explained in part 2.

In the both functions, when landingQueue, departingQueue, runwayFree, departFree or landFree is changed the mutex locks are used to prevent concurrent writing.

```
void* departureThread(void* arg) {
    int id = *(int*)arg;
    Plane *plane = new Plane;
    std::unique_lock<std::mutex> lock(mtx);

    auto requestTime = std::chrono::system_clock::now();
    time_t requestTimeT = std::chrono::system_clock::to_time_t(requestTime);
    std::string timeStr = std::ctime(&requestTimeT);
    timeStr.pop_back();
    plane->requestTime = timeStr;
    plane->status = 'D';

    // Notify the tower
    departureQueue.push(id);
    groundCounter++;
    cv.notify_all();
```

Same format adjustment is used while using the chrono library in the departure just like we did in landing.

PART 2

To implement the below logic, we changed our tower thread function.

- (a) No more planes is waiting to land,
- (b) 5 planes or more on the ground are lined up to take off.

```
if (!landingQueue.empty() && (consecutiveCounter >= 2 || groundCounter < 5)) {
    landFree = true;
    consecutiveCounter = 0; // Reset ground processed counter
    cv.notify_all();
} else if (!departureQueue.empty()) {
    departFree = true;
    cv.notify_all();
}
```

We added a new control to our tower thread. If the groundCounter is smaller than 5, then we need to let landing planes operate, otherwise we should let departing planes. The groundCounter is incremented by 1 when a plane tries to depart. groundCounter is decremented by 1 after the plane takes off. The groundCounter variable is protected by the mutex lock since different threads can try to modify it at the same time.

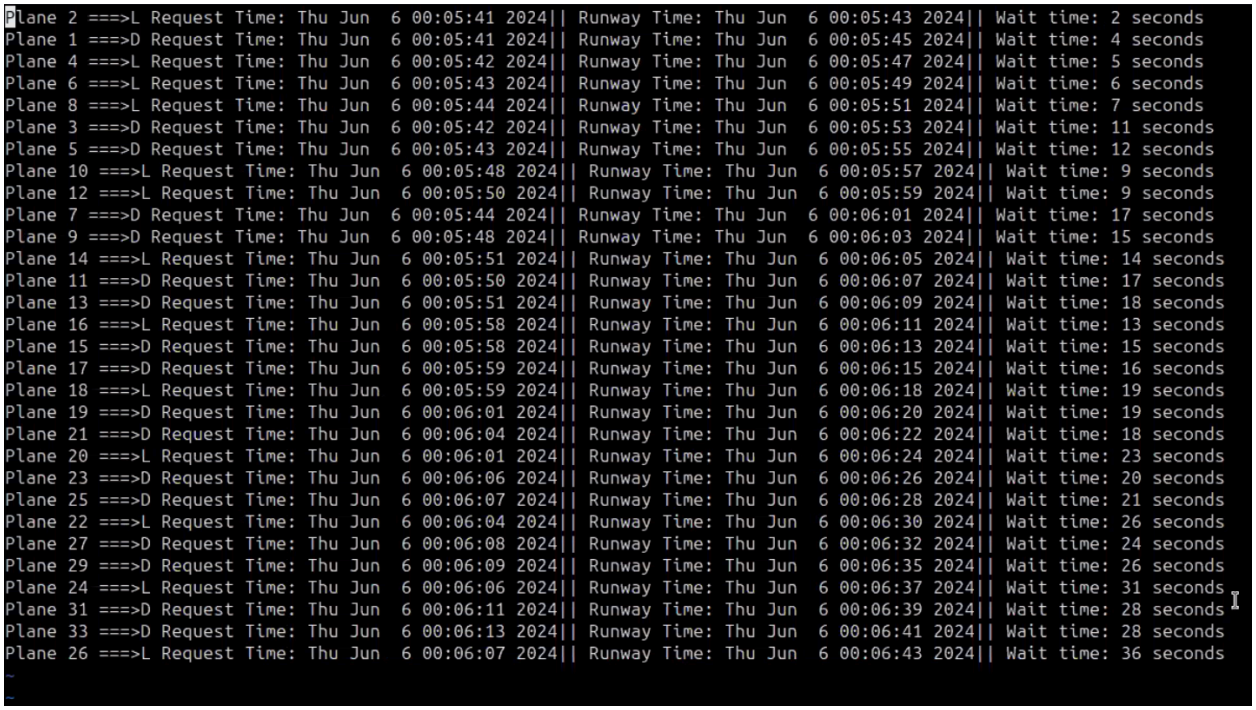
```
// Notify the tower
departureQueue.push(id);
groundCounter++;
cv.notify_all();
```

```
// Remove plane from queue
departureQueue.pop();
groundCounter--;  // Decrement grou
consecutiveCounter++; // Increment
departFree = false;
runwayFree = true;
```

Without the presence of consecutiveCounter, the starvation for landing planes occurs.The reason for that is after a certain time, there will always be more than 5 planes on the ground waiting to take off. Since every 1 second, with a 0.5 probability, a departing plane is created, we face starvation. To prevent this, we add another counter which is consecutiveCounter. The aim of this counter is to let a plane land, after a certain time of consecutive departures. We decided this number to be 2 in our simulation. So, after 2 consecutive departures, there must be a landing if landingQueue is not empty. Consecutive counter is incremented by 1 when a departing plane completes its process. And it is set to 0 in each landing. This way we can prevent starvation for landing planes.

```
 2 files changed, 30 insertions(+), 26 deletions(-)
berke@berke-ThinkPad-T15-Gen-1:~/Desktop/comp-304-operating-systems-project-2-3bornagain$ git push origin main
Username for 'https://github.com': Bkurtuldu
Password for 'https://Bkurtuldu@github.com':
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 8 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 913 bytes | 456.00 KiB/s, done.
Total 5 (delta 3), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To https://github.com/KocUniversity/comp-304-operating-systems-project-2-3bornagain.git
   cfd0f6a..3bb8b94  main -> main
berke@berke-ThinkPad-T15-Gen-1:~/Desktop/comp-304-operating-systems-project-2-3bornagain$ git pull origin main
Username for 'https://github.com': Bkurtuldu
Password for 'https://Bkurtuldu@github.com':
From https://github.com/KocUniversity/comp-304-operating-systems-project-2-3bornagain
 * branch            main       -> FETCH_HEAD
Already up to date.
berke@berke-ThinkPad-T15-Gen-1:~/Desktop/comp-304-operating-systems-project-2-3bornagain$ ./bin/atc_simulator -s 60 -p 0.5 -n 10
At 0 sec ground: 1
At 0 sec air: 2
At 10 sec ground: 3 5 7 9 11 13
At 10 sec air: 10 12 14
At 20 sec ground: 9 11 13 15 17 19
At 20 sec air: 14 16 18 20
At 30 sec ground: 15 17 19 21 23 25 27 29 31
At 30 sec air: 18 20 22 24 26 28 30 32
At 40 sec ground: 21 23 25 27 29 31 33 35 37 39 41 43 45
At 40 sec air: 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48
At 50 sec ground: 27 29 31 33 35 37 39 41 43 45 47 49 51
At 50 sec air: 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54
At 60 sec ground: 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63
At 60 sec air: 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66
```

Here is our print operation with 0.5 probability, 60 second simulation time and 10 second snapshot time.



The screenshot above demonstrates our planes.log file. After Plane 12 lands, we see that a pattern forms which is 2 departures and 1 landing. In this screenshot, we understand that until plane 12 there is a number of planes smaller than 5 on the ground waiting for take off. After plane 12, there are more, so our consecutiveCounter algorithm works and prevents starvation for landing planes.

IMPORTANT NOTE - TO ACHIEVE CORRECT RESULTS:
FIRST
make clean
AND THEN
make all
SHOULD BE RUN. AFTER THAT
./bin/atc_simulator -s 60 -p 0.5 -n 10
SHOULD BE RUN TO SEE THE OUTPUT.