COMP 304 Project 2 Report - Ege Erdogan 64004

Overview

My solution is divided into three modules:

- `solution.c` contains the main running code.
- `queue.h` contains the circular array FIFO queue implementation used for landing, departing, and emergency queues.
- `logging.h` contains some helper methods to keep logs.

The first thing that the `main` method does is to parse the arguments: `-s` for total simulation time, `-p` for the probability of a landing plane arriving, and `-n` for when to display snapshots. They can be given in any order.

Then some variables are initialized and the ATC thread is created along with a landing plane and a departing plane. The plane-generating loop begins afterwards. Which type of plane to generate is decided by getting a random number from the `rand()` method, dividing it by the `RAND_MAX` constant to get a number between 0 and 1, and finally comparing it with teh `p` parameter provided. A landing plane arrives with probability `p`, and a departing plane arrives with probability `1-p`. Both types of planes can arrive simultaneously as well.

Locks, Condition Variables and Other Structures

Three different queues are used to store waiting planes. One for landing, one for departing, and one for emergency planes. There is a mutex lock for each queue. Any plane adding itself to the queue should first obtain the respective lock to avoid race conditions with other planes. Queues and their locks are initiated in the `main` method after parsing the arguments.

Each plane also has its own mutex & condition variable pair. The ATC signals a plane's `available` condition variable when it is allowed to land/depart. Those variables are initiated when the `Plane` struct is firt created in `landing` and `departing` methods.

Air Traffic Controller

The ATC has its own thread that runs the `traffic_control` method. It initially waits for the first plane to signal, and then enters a `while` loop until the simulation is over, permitting planes to land/depart according to the following rules:

- If there is an emergency landing plane, allow it to land.
- If there is a plane waiting to land, and either there are less than 5 planes waiting to depart, or the last two planes allowed were departing planes, allow it to land.
- If there is a departing plane, allow it to depart.

```
if (emergency_queue->size > 0) {
  // allow emergency landing
} else if (landing_queue->size > 0 && (departs == MAX_CONSEC_DEPARTS ||
departing_queue->size < 5)) {
  // allow landing
} else if (departing_queue->size > 0) {
  // allow departure
}
```

Outline of the ATC code to implement the logic above. `departs` is a local variable to keep track of consecutive departing planes.

The maximum consecutive departing planes constraint is added to avoid starvation of landing planes. Without that, when the arrival rate of departing planes is larger than their approval rate, the queue keeps increasing in size and never goes below 5. In that case, a landing plane is never allowed to land. This extra contraint makes sure that a landing plane is allowed to land no matter how many planes are waiting to depart. The number can be changed by modifying the `MAX_CONSEC_DEPARTS` constant. Any value would avoid starvation, but a value too large could increase the waiting time of landing planes to undesired levels.

Planes

```
struct Plane {
  int id;
  char status;
  int request_time;
  int completed_time;
  pthread_cond_t available;
  pthread_mutex_t mutex;
};
```

Each plane is represented by the struct above in its own thread. Landing planes have even `id`s and departing planes have odd `id`s. The status char is L for landing, E for emergency, and D for departing planes.

In the `landing` and `departing` methods, first a `Plane` struct is created and its values are initialized according to the type of the plane. Then the thread waits to acquire the lock for its respective queue, pushes the plane into the queue, and starts waiting for the signal from the ATC thread.

An emergency landing is disinguished from a regular landing by passing an argument to the `landing` method when creating the thread. If `emergency` is true, an emergency landing is created. Another solution would be to write a new method to be called when the emergency landing thread is created, but passing an argument is more efficient since a large part of the code is the same for emergency and regular landings.

Queue

```c
struct Queue {
  int capacity;
  int size;
  int head;
  int tail;
  struct Plane *queue_array[CAPACITY];
};
```

I implemented a fixed-size FIFO queue with a circular array in the `queue.h` file. The `head` and `tail` indices keep track of the start and end of the queue in the array. Available operations are `is_full`, `pop`, `push` , and `print_queue`.

`head` and `tail` are both -1 when the queue is empty. Then at each push, `tail` is incremented by one, looping back to 0 after reaching the end of the array. Similarly, `head` is increased by one after each pop operation.

Logging

The helper methods to keep logs are found in the file `logging.h`. I kept two different log files, `planes.log` and `tower.log`. A plane is logged to `planes.log` when it is approved to land/depart, and to `tower.log` when it arrives. So in the end, `tower.log` keeps a history of all the planes with their IDs, arrival times, and types, sorted by arrival time. `planes.log` keeps a history of all planes that has landed or departed, sorted by their approval times.