Student Details

- **uid**: 'u7108792'
- **name**: 'Quoc Nguyen'

References

- [1] comp2310 (2024), Condition Variables and Threaded Echo Server Sample Solutions: Sample Solutions for Lab 9 & Lab 10, accessed at: https://comp.anu.edu.au/courses/comp2310/resources/06-sample-sols/
- [2] Bryant R, O'Hallaron D (2016), Computer Systems: A Programmer's Perspective, 3/E, chapter 11-12 accessed at: https://www.cs.sfu.ca/~ashriram/Courses/CS295/assets/books/CSAPP_2016.pdf

Report

Overview of the Air Traffic Control Network

This air traffic control (ATC) system prototype simulates the communication between a **central controller node** and a **network of airport nodes** to facilitate flight scheduling and gate management. The **controller node** acts as a proxy, receiving requests from clients and forwarding them to appropriate airport nodes, which are implemented as **child processes** to mimic a distributed system. Each airport node contains a set number of gates, and each gate manages a schedule with **48 half-hour slots** representing a single day.

The key features implemented include:

- **Flight scheduling**: Requests to assign landing times to gates in airports, ensuring no conflicts.
- **Flight status lookup**: Requests to retrieve a plane's scheduled gate and timeslot.
- **Gate information**: Requests to check the occupancy status of specific gates for given time ranges.
- **Concurrency support**: Implemented **multithreading** in airport nodes using **thread pools**, avoided controller threading for simplicity .
- **Robust error handling**: Proper handling of invalid requests, incorrect airport/gate numbers, and network connection issues.

This prototype ensures that multiple clients can interact with the system concurrently, and it guarantees thread-safe access to shared data structures like flight schedules, avoiding scheduling conflicts or deadlocks.

Request Format Used

To maintain simplicity and consistency, the **controller forwards requests** in the same format as they are received from clients, with the airport number retained. This choice improves **debuggability** and allows for easier tracking of requests during testing, as it ensures consistency between forwarded and received requests.

The request format follows the guidelines below:

1. **SCHEDULE** requests: Used to assign a flight to a gate with the following parameters: SCHEDULE [airport_num] [plane_id] [earliest_time] [duration] [fuel]
2. **PLANE_STATUS** requests: Used to query the gate and time slots occupied by a flight: PLANE_STATUS [airport_num] [plane_id]
3. **TIME_STATUS** requests: Used to check the status of specific time slots within a gate: TIME_STATUS [airport_num] [gate_num] [start_idx] [duration]

This **verbatim forwarding strategy** simplifies both the controller and airport node implementations, as no preprocessing or parsing beyond routing is required. This also reduces potential parsing errors and ensures requests remain traceable for debugging purposes.

Extensions

Multithreading within Airport Nodes

To enhance the **concurrency** and **performance** of the airport nodes, a **thread pool** was implemented within each airport process. This allows the airport nodes to handle multiple client requests simultaneously, thereby improving the system's ability to manage high volumes of scheduling and status queries without significant delays.

Implementation Details:

1. **Thread Pool Initialization:**

- A fixed number of worker threads (`MAX_THREADS` , set to 4) are created at the startup of each airport node.
- These threads are initialized in **detached mode** to ensure they do not consume additional resources after completion.

2. **Connection Queue:**

   - A **bounded queue** (`conn_queue_t`) with a maximum capacity (`MAX_QUEUE` , set to 16) is implemented to store incoming client connections.
   - The queue utilizes **mutexes** and **condition variables** to manage concurrent access, ensuring thread-safe operations when adding or removing connections.

3. **Worker Threads:**

   - Each worker thread continuously waits for new connections to be enqueued.
   - Upon receiving a connection (`connfd`), a thread dequeues it and processes the associated client requests by parsing commands (`SCHEDULE` , `PLANE_STATUS` , `TIME_STATUS`) and interacting with the gate schedules accordingly.

4. **Locking Strategy:**

   - A **fine-grained locking mechanism** is employed at the **gate level**. Each `gate_t` structure contains its own **mutex** to protect access to its `time_slots` .
   - This approach allows multiple threads to operate on different gates concurrently without interfering with each other, thereby maximizing parallelism and reducing contention.
   - Specifically by implementing consistency via pthread_mutex mutex, allowing only one thread to access these resources, we ensure that the integrity of `conn_queue_t` is maintained.

5. **Request Handling:**

   - **SCHEDULE** requests are processed by searching for available time slots within the specified gate, ensuring that no overlapping schedules occur.
   - **PLANE_STATUS** and **TIME_STATUS** requests retrieve and report the current scheduling information for specific planes or gates.
   - Proper synchronization ensures that updates to gate schedules are atomic and consistent across all threads.

Impact on Performance:

Implementing a thread pool within airport nodes significantly **increases the system's throughput** by allowing multiple client requests to be handled in parallel. The fine-grained locking mechanism ensures that operations on separate gates do not block each other, thus **optimizing resource utilization** and **minimizing latency** for client requests.

Testing

Challenges Encountered

The largest challenge encountered was setting up the inital connection, reasons only the universe knows the initial testing script refused to run on my machine and due to my inexpereince with netcat at the time of starting the assignment. This was by far the largest challenge as I was not able to effectively test any of the stuff I was coding untill about 11 am this morning (30/10/2024) as it turns out it was a combination of specific kernel settings on my machine in conjunction with the make file that rendered my controller build useless, my connections would always time out as no child processes were every created. I managed to resolved that by praying to the arch gods and sheer luck. up untill 1pm today I was coding this assignment off pure intuition and will power as I couldnt even debug if i wanted to. I have about 4 different versions of this assignment all in differnet states of diaaray from a purely sequential implementaion to one with controller threading, however, since the ability to debug was granted to me this morning by the lord, I have managed to produce this mess (version that was the most stable).

Additional Tests I Created in 1 hour

While automated testing scripts were developed beyond the included testing script, extensive (as much as i could think) **manual testing** was conducted using **netcat** (`nc`) to simulate client interactions with the controller and airport nodes. This approach allowed for real-time verification of request handling, response accuracy, and concurrency management.

Testing Scenarios Included:

1. **SCHEDULE Requests:**
    - Tested scheduling flights with various parameters to ensure flights are assigned to the earliest possible time slots and lowest gate numbers.
    - Verified that overlapping schedules are correctly rejected with appropriate error messages.
2. **PLANE_STATUS Requests:**
    - Queried the status of scheduled planes to confirm accurate reporting of gate assignments and time slots.
    - Checked responses for planes that were not scheduled to ensure correct error messages are returned.
3. **TIME_STATUS Requests:**
    - Requested time slot statuses for specific gates to verify that allocations and free slots are accurately reported.
    - Tested edge cases, such as querying the full range of time slots to ensure comprehensive coverage.
4. **Concurrent Connections:**
    - Simulated multiple clients connecting simultaneously to the controller to assess the system's ability to handle concurrent requests without data corruption or scheduling conflicts.
    - Observed the behavior of airport nodes under concurrent access to ensure thread-safe operations.
5. **Error Handling:**
    - Sent malformed requests and invalid parameters to test the robustness of error detection and response mechanisms.
    - Verified that the system gracefully handles invalid airport or gate numbers, as well as incorrect request formats.

Known Bugs or Limitations

1. **Single-Threaded Controller Node:**
    - The **controller node** currently operates in a **single-threaded** manner, handling one client connection at a time. This design choice limits the system's ability to handle multiple concurrent clients efficiently.
    - **Impact:** Under high load, the controller may become a **bottleneck**, causing increased response times as each client must wait for the previous one to be processed.
    - **(IMPORTANT) Design Flaw:** Due to the way controler connections, to the client is set up, it seems that past a certain amount of events there is tendancy for connections to be lost and then be unable to connect to the airport/client again. Initial it was believed that this was due to the thread de-queuer not effectively resolving the connections from the pool. After extensive testing it seems that the de-queuer is actively performing well and the bug may be elsewhere such as within the controller.
2. **Potential Race Conditions in Schedule Assignment:**
    - Although **fine-grained locks** are implemented at the gate level, there is a **possibility of race conditions** when multiple threads attempt to schedule flights on the same gate simultaneously.
    - **Impact:** Without additional safeguards, two threads might still find the same time slots available before either has a chance to lock and assign them, potentially leading to **overlapping flight schedules**.

3. **Limited Error Handling in Airport Nodes:**
    - The airport node's request processing functions primarily check for the correct number of arguments and basic value ranges but may **not cover all edge cases** or **invalid input scenarios**.
    - **Impact:** This could result in **undefined behavior** or **unexpected crashes** if malformed requests are received.
4. **Non-Deterministic Scheduling Under High Concurrency:**
    - Due to the concurrent nature of request handling, the order in which scheduling requests are processed may vary between runs.
    - **Impact:** This leads to **non-deterministic scheduling outcomes**, making it challenging to **reproduce specific scheduling scenarios** during testing.
5. **Hardcoded Maximum Limits:**
    - Constants like `MAX_THREADS` and `MAX_QUEUE` are hardcoded, limiting the **scalability** of the system.
    - **Impact:** In environments requiring higher concurrency, these limits could **restrict performance**, necessitating manual adjustments to the source code.
6. **No Graceful Shutdown Mechanism:**
    - The current implementation lacks a mechanism to **gracefully shut down** the controller and airport nodes, ensuring that all threads and resources are properly released.
    - **Impact:** This could lead to **resource leaks** or **incomplete processing** of ongoing requests when terminating the system.

Conclusion

The **Air Traffic Control (ATC)** system prototype successfully emulates a **distributed air traffic control network** by integrating a central **controller node** with multiple **airport nodes**. The implementation effectively manages flight scheduling, status lookups, and gate queries, ensuring that operations are both **reliable** and **robust**. By incorporating **thread pools** within the airport nodes, the system adeptly handles multiple concurrent client requests, enhancing overall **performance** and **scalability**.

The use of **fine-grained mutexes** at the gate level ensures **thread-safe access** to shared resources, preventing race conditions and maintaining the integrity of flight schedules. This locking strategy allows multiple threads to operate on different gates simultaneously, optimizing **resource utilization** and minimizing **latency** in request processing.

During development, extensive **manual testing** was conducted using **netcat** to simulate client interactions. These tests validated the correct handling of various request types and the system's ability to manage concurrent connections without scheduling conflicts. While manual testing provided valuable insights into the system's behavior under typical usage scenarios, the absence of automated testing scripts means that some edge cases and high-concurrency situations may not have been thoroughly examined.

Known limitations include the **single-threaded** nature of the controller node, which could become a bottleneck under heavy load, and the potential for rare **race conditions** despite the implemented mutex scheme. Additionally, the lack of a **graceful shutdown mechanism** poses challenges for resource management during system termination.

Future enhancements could involve transitioning the controller node to a **multithreaded architecture** to better handle simultaneous client connections, implementing **automated testing frameworks** to ensure comprehensive coverage of edge cases, and refining the **locking mechanisms** to further eliminate any residual race conditions. Introducing **slot-level locking** within gates could provide even finer control over schedule management, though it would require meticulous design to prevent deadlocks.

Overall, this prototype demonstrates the foundational capabilities of a **robust and scalable ATC network**, laying the groundwork for more advanced features and optimizations necessary for real-world deployment.

Test Results

Test basic-1

```
Running make all:               ok!
Running test basic-1:           passed!
Total: 1/1 tests passed.
```

Test basic-2

```
Running make all:               ok!
Running test basic-2:           passed!
Total: 1/1 tests passed.
```

Test basic-3

```
Running make all:               ok!
Running test basic-3:           passed!
Total: 1/1 tests passed.
```

Test basic-4

```
Running make all:               ok!
Running test basic-4:           passed!
Total: 1/1 tests passed.
```

Test basic-5

```
Running make all:               ok!
Running test basic-5:           failed!
  - Actual output does not match expected output
 === Output of file './output/basic-5/response0' ============
 === Warning: Only showing first and last 20 lines. =========
SCHEDULED 1 at GATE 0: 00:00-22:30
SCHEDULED 2 at GATE 1: 00:00-01:30
SCHEDULED 3 at GATE 2: 00:00-09:00
SCHEDULED 4 at GATE 1: 09:00-19:00
SCHEDULED 5 at GATE 3: 00:30-23:30
Error: Invalid 'duration' value (47)
SCHEDULED 6 at GATE 4: 02:30-23:30
SCHEDULED 7 at GATE 0: 00:00-00:00
SCHEDULED 8 at GATE 0: 00:00-00:00
SCHEDULED 9 at GATE 0: 00:00-00:00
SCHEDULED 10 at GATE 2: 09:30-21:30
SCHEDULED 11 at GATE 5: 00:00-12:00
SCHEDULED 12 at GATE 6: 00:00-18:00
SCHEDULED 13 at GATE 0: 00:00-00:00
SCHEDULED 14 at GATE 5: 12:30-22:30
SCHEDULED 15 at GATE 7: 00:00-23:30
```

```
SCHEDULED 16 at GATE 8: 00:00-20:00
SCHEDULED 17 at GATE 1: 02:00-06:30
Error: Could not connect to airport 0
Error: Could not connect to airport 0
[ ... removed output ... ]
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
Error: Could not connect to airport 0
 ============================================================
 === Output of file './output/basic-5/server_out' ===========
[Controller] Airport 0 assigned port 1102
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
```

```
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
[Controller] Failed to connect to airport 0
 ============================================================
  - What is being run:  ./controller -p 1101 -n 1 -- 10
Total: 0/1 tests passed.
```

Test basic-6

```
Running make all:               ok!
Running test basic-6:           passed!
Total: 1/1 tests passed.
```

Test multi-1

```
Running make all:               ok!
Running test multi-1:           passed!
Total: 1/1 tests passed.
```

Test multi-2

```
Running make all:               ok!
Running test multi-2:           passed!
Total: 1/1 tests passed.
```

Test concurrent-1

```
Running make all:               ok!
Running test concurrent-1:      passed!
Total: 1/1 tests passed.
```

Test concurrent-2

```
Running make all:               ok!
Running test concurrent-2:      passed!
Total: 1/1 tests passed.
```

Test concurrent-3

```
Running make all:               ok!
Running test concurrent-3:      passed!
Total: 1/1 tests passed.
```