COMP 304 - Operating Systems: Project 2 Report

Barış Samed Yakar 69471
Umur Berkay Karakaş 69075

Part I

Simulation Time: The simulation time is controlled by a while loop that continues until the elapsed time (difference between the current time and the start time) reaches the specified simulation time.

```
while (difftime(time(0), start_time) < simulation_time);
```

Metro Lines and Sections: Sections A-C, B-C, D-E, and D-F are represented by separate queues (`section1Queue`, `section2Queue`, section3Queue, section4Queue). Each train is enqueued based on its source and destination.

```
if (source == 'A') {
    section1Queue.push(train);
} else if (source == 'B') {
    section2Queue.push(train);
} else if (source == 'E') {
    section3Queue.push(train);
} else {
    section4Queue.push(train);
}
```

Train Movement and Speed: The movement of trains through sections is implemented in the sectionFunction thread. Trains move through sections in a loop, and their speed is simulated using the `sleep(1)` function.

Train Arrival Probability: The arrival probability is determined by comparing a randomly generated threshold (threshold) with the specified probability 'p'.

```
float threshold = (float)rand() / RAND_MAX;
if (prob_create <= threshold) {
    // Train creation logic
}
```

Line Selection Probability: The line selection probability is implemented by generating a random number (rand_line) to determine the destination of the train.

```
if ((source == 'A') || (source == 'B')) {
    destination = (rand_line <= 0.5) ? 'E' : 'F';
} else {
```

```
    destination = (rand_line <= 0.5) ? 'A' : 'B';
}
```

Train Length Probability: The length of the train is determined by comparing a randomly generated value (rand_length) with the specified probabilities.

```
int length = (rand_length <= 0.7) ? 100 : 200;
```

Control Center Efficiency: The control center ensures efficiency by selecting the section with the largest number of trains using a priority queue (pq).

```
priority_queue<Train, vector<Train>, greater<Train>> pq;
addSectionQueuesToPriorityQueue(pq, section1Queue);
addSectionQueuesToPriorityQueue(pq, section2Queue);
addSectionQueuesToPriorityQueue(pq, section3Queue);
addSectionQueuesToPriorityQueue(pq, section4Queue);
```

Priority in Case of Tie: When there is a tie, the control center prioritizes trains based on their departure points (A > B > E > F). It is done by ordering in the conditions.

```
if (maxQueueSize == section1QueueSize) {
    // Train selection logic for section 1
} else if (maxQueueSize == section2QueueSize) {
    // Train selection logic for section 2
} else if (maxQueueSize == section3QueueSize) {
    // Train selection logic for section 3
} else {
    // Train selection logic for section 4
}
```

Part II

Clearance Mode Handling: In the `sectionFunction` thread, a check is implemented to determine whether the system is in clearance mode. If clearance is set to true, it indicates that the system is overloaded, and clearance mode is activated. During clearance mode, the system does not allow new trains to arrive at points A, B, E, and F until all trains clear the tunnel.

```
if (clearance) {
    continue; // Skip train arrival if in clearance mode
}
```

System Overload Notification: The decision to enter clearance mode is based on the total number of trains in the entire system, excluding the tunnel section. If the total count exceeds a specified threshold (10 trains in this case), the control center notifies trains to slow down, initiating the clearance mode.

```
pthread_mutex_lock(&trainMutex);
if (section1Queue.size() + section2Queue.size() + section3Queue.size() +
section4Queue.size() > 10) {
    if (!clearance) {
        time_to_clear = time(0);
        controlLogQueue.push({"System Overload", time(0), "#",
trainsWaiting()});
    }
    clearance = true;
}
pthread_mutex_unlock(&trainMutex);
```

Part III

Event Structure: The Event structure is designed to encapsulate control center events. It includes fields such as the event type, event time, train ID, and a string representing trains waiting for passage. This structure is crucial for organizing and recording events in the simulation.

```
typedef struct Event {
    string event;
    time_t event_time;
    string train_id;
    string trains_waiting_passage;
} Event;
```

Logging Threads:
Two separate threads, controlLog and trainLog, are responsible for logging events to the "control-center.log" and "train.log" files, respectively. These threads ensure the orderly recording of control center events and comprehensive train details during the simulation.

```
void* controlLog(void* /*arguments*/) {
    // Open control-center.log and write events
}
void* trainLog(void* arguments) {
    // Open train.log and write train details
}
```

Logging Events in Control Center:
The control center thread (controlCenter) generates events during the simulation, and these events are pushed into a queue (controlLogQueue). Key events, such as tunnel passages, breakdowns, system overloads, and tunnel clearance, are logged along with relevant details such as train IDs and waiting trains.

```
controlLogQueue.push({"Tunnel Passing", time(0), to_string(train.id),
trainsWaiting()});
controlLogQueue.push({"Breakdown", time(0), to_string(train.id),
trainsWaiting()});
```

```
controlLogQueue.push({"System Overload", time(0), "#", trainsWaiting()});
controlLogQueue.push({"Tunnel Cleared", time(0), "#", "# Time to clear: " +
to_string(clear_time) + " sec"});
```

TrainLog Function:The trainLog function is responsible for logging detailed information about each train's journey in the "train.log" file. This includes simulation arguments, train IDs, starting and destination points, lengths, departure times, and, if applicable, arrival times.

Data Structures:

- Train Struct: Represents the characteristics of a train, including ID, source, destination, length, departure time, and arrival time. It is defined as a C++ struct with overloaded comparison operators (>, <) for sorting.
- Event Struct: Represents events in the metro simulation, such as train arrivals, departures, breakdowns, system overloads, and tunnel clearances. It holds information about the event type, event time, train ID, and trains waiting for passage during tunnel events.
- Mutexes: Ensures thread safety and prevents race conditions during critical sections of the simulation, such as tunnel passage and train data manipulation. Three mutexes are implemented - `tunnelMutex` for tunnel-related operations, `trainMutex` for train data manipulation, and `coutMutex` for ensuring thread-safe cout during debugging.
- Queues: (`section1Queue`, `section2Queue`, `section3Queue`, `section4Queue`) Organizes trains waiting to pass through specific sections of the metro lines (A-C, B-C, D-E, D-F). Standard C++ queues are used to manage the order of trains waiting in each section.Priority Queue: Determines the priority of trains waiting to pass through the tunnel based on the number of trains in each section. Utilized in the `addSectionQueuesToPriorityQueue` function to prioritize the section with the most trains. Also, Control Log Queue (`controlLogQueue`) captures events in the metro simulation for detailed logging and analysis. Standard C++ queue used to store Event structs.
- Hash Map : `trainsHashMap` stores information about each train using its ID as the key for quick access and reference. Implemented as an unordered map in C++.
- Global Variables : (`start_time`, `clearance`, `time_to_clear`, `train_id`) stores global information, including the simulation start time, clearance status during system overloads, the time to clear the tunnel, and the current train ID. Utilized to track and control various aspects of the simulation.

Functions:

- **`sectionFunction`**: Simulates the behavior of trains arriving at different points (A, B, E, F) in the metro system. It creates trains, determines their characteristics, and adds them to the corresponding section queues.
- **`controlCenter`**: Manages the passage of trains through the tunnel, handles system overloads, and logs relevant events. It controls tunnel passage, handles breakdowns, prevents starvation during system overloads, and logs events.
- **`addSectionQueuesToPriorityQueue`**: Populates a priority queue with trains from multiple

section queues to determine the order of passage through the tunnel based on the number of trains in each section. Used in the controlCenter function to prioritize sections with more trains.

- **`trainLog`**: Logs detailed information about each train, including ID, source, destination, length, departure time, and arrival time. It iterates through the trains in the simulation, formats train information, and writes it to the "**train.log**" file.
- **`controlLog`**: Logs events in the metro simulation, including tunnel passages, breakdowns, system overloads, and tunnel clearances. It writes event information to the "**control-center.log**" file.
- **`main`**: Coordinates the initialization of mutexes, creation of threads, simulation execution, and cleanup. Manages the main flow of the program, including command-line argument parsing, thread creation, and joining threads after the simulation period.