CmpE 322 - Operating Systems

Project#3 - Simulation of a Fun Fair Payment System

BOFUN Fair - It's Fun!

(Due Date: 30 December 2022, 23:59)

1. Introduction

In this project, you will implement a **multi-threaded application** for simulating a fun fair payment system, where the customers can make the prepayments of the various rides through the available ticket vending machines. Since the prepayments can be made simultaneously through different ticket vending machine instances, the **synchronization and the data consistency** should be taken into the consideration throughout the implementation.

Our Fun Fair consists of many different joyful rides. However, every ride does not belong to the same company. The Fun Fair is directed by 5 major companies named: "Kevin", "Bob", "Stuart", "Otto" and "Dave". They are all named after the little sons of their owners. Customers who arrive at the BOFUN Fun Fair must purchase an electronic ticket, right before entering the fun fair. The ticket is an electronic card in which the customer needs to prepay (pay beforehand) the amount that will be used in the fair. We guarantee that every customer does this prepaying option only and exactly once. Also, as those five companies' owner are kind of "different" in the way of thinking, they each have their own separate accounts and they force every customer to prepay their amounts for a specific company that they select.

A sample scenario of the simulation can be expressed as follows (the details are presented in Section 2): There is a number of customers, which should be specified by the input configuration file as an argument provided to the program, that seeks for the prepaid ticket prepayment service. There are 5 different companies that the prepayments can be chosen to: "Kevin", "Bob", "Stuart", "Otto", and "Dave" In our system, there will be 10 different ticket vending machines that provide the necessary services for realizing the prepayments requested by the customers.

The customers arrive sequentially, but each sleep (you can think of this sleeping period for thinking about which rides to take) for a certain duration (the amount of time for sleeping is also specified by the input configuration file) after the arrival, and then pick a ticket vending machine. Each customer makes only one prepayment, of a predefined company with an amount to be paid, and leaves the environment. The customer provides the necessary information about the prepayment, and the corresponding ticket vending machine makes the prepayment on behalf of the customer. If there is an ongoing prepayment operation made by another customer on that particular ticket vending machine, the newly arriving customer should wait until the operation of the customer at the head of the queue is completed. Therefore, the queue mechanism of the ticket vending machines should be implemented as a **FIFO queue**. (**Hint**: The mutex unlock operation behaves like FIFO queue in Linux and MacOS pthread implementation. Therefore, you don't need implement an extra linked list or similar structure to imitate the FIFO queue functionality.)

However, there is a single bank account for each company So, the simultaneous prepayments made through different ticket vending machine instances should not lead to an inconsistent situation.

You need to implement this program in **C/C++ programming language** that should be compiled by **gcc/g++ compilers**.

The rest of this project description document organizes as follows. Section 2 provides the detailed information about the program and implementation. Section 3 provides the format of the log file that should be prepared by your program. Section 4 provides the list of submission criteria and overall submission process.

2.Implementation Details

Example program execution command: `./simulation input.txt`

As stated in the previous section, the program should be executed with an input configuration file (the name of the file can be anything). The input file includes the information that is required for simulation to progress, and it should reside in the same directory with your program. The first line of the input file represents the total number of customers to be simulated in the following run. You can assume that the number of customers is always a positive integer value that does not exceed 300**.**

After the execution, the specified number of customers begin to arrive. Each of the customers should be represented as **separate threads**. The main thread begins to create the customer threads **sequentially, without any waiting time**. However, **a customer does not begin the execution immediately**. Each customer thread sleeps for a time (**in milliseconds**) upon creation. The amount of time for each customer to sleep is also specified by the input configuration file.

Then, after waking up, that customer **picks a ticket vending machine instance**, that is depicted in the input configuration file again. If the determined ticket vending machine is currently empty and waits for an incoming request, it will become ready to process the operation instructed by the customer. The operations of a prepayment are as follows:

1. **The name of the company to be prepaid should be determined through the input configuration file**. There will be 5 different companies as stated in the previous section: Kevin, Bob Stuart, Otto, and Dave.
2. After determining the company name, the customer finds the amount to be prepaid for that company in the configuration file.

The ticket vending machine then processes the instructions, and **realize the prepayment operation on behalf of the customer**. In the project, it is important to represent the **ticket vending machine instances as separate serving threads** to carry out the prepayment operations. The customer threads should not make the prepayments themselves, their only role is providing the necessary information about the prepayment. After a customer provides the necessary information, the corresponding ticket vending machine thread will be the one that executes the prepayment operation. You should create the ticket vending machine threads in advance, before the first customer arrives. Besides, the ticket vending machine machines as server threads should not read the input configuration file in order to carry out the prepayment operations.

If the ticket vending machine that is picked by the customer is currently serving to another customer, **the incoming customer should wait until it becomes the head of the queue** (as stated in Section 1, the queue mechanism can be represented through mutex lock and unlock operations that behaves like a FIFO queue. You don't need to implement an additional structure for this objective).

After the request of a customer is handled and the corresponding prepayment is made, **the client thread exits without any additional operation**.

Please consider the specific format of the **input configuration file** as given below. As aforementioned, the first line specifies the number of customers to be simulated, while the rest of the configuration file represents the customers. The order of the lines should be aligned with the order of customer thread creation. A particular line includes the following information about a customer that arrives for a prepayment: **<sleep time, ticket vending machine instance, prepayment company name, amount>**. It should be noted that the ticket vending machine instances are enumerated as [1, 10]. The format of the configuration file is exact. For example, there will be no whitespaces within lines, and there will be always **NumberOfCustomers+1 lines** in the configuration file.

```
58
92,1,Bob,300
44,1,Kevin,231
2,2,Otto,50
322,9,Bob,30
123,8,Stuart,65
90,9,Kevin,100
1,4,Bob,40
…
```

3. Log of the Operations

After completing a prepayment operation, the ticket vending machine instance should log the operation to an external file that should be named as **<nameOfTheInputFile>_log.txt**. This output file should reside in **the same directory where the program resides**. Below, you can find the exact format of the output log file. Please consider that the **customers are enumerated according to their arrival sequence, not the sequence of waking up or serving time**. The first customer that arrives at the environment will be Customer1, the following customer will be represented as Customer2 and so on…

```
Customer2,231TL,Kevin
Customer5,65TL,Stuart
Customer1,300TL,Bob
…
…
All prepayments are completed.
Kevin: 744TL
Bob: 900TL
Stuart: 144TL
Otto: 500TL
Dave: 199TL
```

As observed, **the output log is not sorted according to the arrival sequence of the customers**. The output log and the lines should be **sorted according to the prepayment time**, not customer arrival time.

After all of the customers are served, a line that represents the end of the operations are printed into the file. **This line of the output log ("All prepayments are completed.") should be written by the main thread of the program**, not by any other thread since it represents the end of the program. After that, **for each prepayment type, total amount of prepayments that are made are printed one by one in the log file** as presented above. This information should be also printed by the main thread of the program.

It is possible to read the log file in the end for calculating the total prepayments made for each type of prepayment. However, it is not a good practice. You should keep a separate variable as the balance for each prepayment type (suppose there is a single bank account for each company to keep track of the overall balance, and each of the prepayments for a particular company will be added to that balance).

Your program should **handle the synchronization issues, including the balance control, ticket vending machine usage and logging part**. You should avoid the following type of implementation: implementing the whole program as a single critical section. In order not to restrict the functionality of the program, **different critical sections should be implemented as it**

should be. Of course it is possible to implement whole program as a single critical section, but it does not provide the necessary functionality. So this type of implementation is forbidden.

4. Submission Process

- You need to upload a .zip file on Moodle until the deadline specified at the top of this document **30 December 2022, 23:59)**. Important note: No .rar files or other extensions are accepted.
- There is no late submission policy for this project. The projects submitted after 30 December 2022, 23:59 will not be evaluated.
- This is an individual assignment, no group submission is allowed. Plagiarism policy of the course applies for this project, which means that your code will be analyzed for plagiarism. No excuses will be accepted.
- The name of the zip file should be **StudentID.zip**, without any brackets (e.g. 2020000000.zip). The files that must be included in the submission package are:
  - Code files (C or C++)
  - A Makefile to create the executable (cmake files are not accepted)
- You should document your code.
- Here is the grading policy:
  - Coding & implementation (90%)
  - Code documentation (10%)
- If you conflict with any provided rule in this section, it will negatively affect your grade.
- Good luck!