2022 Fall CMPE 322 Project 3

Simulation of a Fun Fair Payment System

Ömer Faruk Çelik

This program simulates a fun fair prepayment system by creating separate threads for each of the customers and 10 ticket vending machines, while also considering synchronization and data consistency.

Implementation Details:

1-Global Variables

```
// Global variables.
ofstream outputf;

int totalAmount[5] = {0, 0, 0, 0, 0};
int numberOfCustomers;
int machine_volume[11]; //holds remaining prepayment actions for each vending machine

pthread_mutex_t vending_machine_lock[10];
pthread_mutex_t customer_lock[10];

customer cus_at_vending[10]; //holds customer info assigned to each vending machine

pthread_mutex_t total_amount_lock[5];
pthread_mutex_t output_write_lock;
```

totalAmount: An array of size 5 that holds the total amount of prepayments earned by each company which will be printed to output file at the end of the execution.
numberOfCustomers: The total number of customers.
machine_volume: An array of size 11 that holds the remaining number of prepayment actions for each vending machine. Size is 11 because ids are starting from 1.
cus_at_vending: An array of size 10 that holds the customer information assigned to each vending machine.

The program also includes the following mutexes:

vending_machine_lock: An array of size 10 that is used to ensure synchronization by mainly blocking the vending mahine thread.
customer_lock: An array of size 10 that is used to ensure synchronization by mainly blocking the customer thread.
Together, these two mutex locks provide synchronization of data transfer between the vending machine and the customer.

total_amount_lock: An array of size 5 that is used to synchronize access to the totalAmount array.
output_write_lock: A mutex used to synchronize access to the output file.

2- Main Function:

```
int main(int argc, char *argv[])
{
    if(argc != 2) {
        return 1;
    }
    ifstream inputf(argv[1]);
    string outputf_str= argv[1];
    outputf_str.erase(outputf_str.end()-4, outputf_str.end());
    outputf_str += "_log.txt";
    outputf.open(outputf_str);

    string line;
    getline(inputf, line);
    numberOfCustomers = stoi(line);

    customer customers[numberOfCustomers];
    for(int i = 0; i < numberOfCustomers; i++) {    //read input file and assign customer info
        customers[i].id = i+1;
        string line;
        getline(inputf, line);
        stringstream ss(line);
        string token;
        int j = 0;
        while(getline(ss, token, ',')) {
            if(j == 0) {
                customers[i].sleepTime = stoi(token);
            } else if(j == 1) {
                customers[i].ticketVendingMachineInstance = stoi(token);
                machine_volume[customers[i].ticketVendingMachineInstance] += 1;     //update remaining prepayment actions for each vending machine
            } else if(j == 2) {
                if(token == "Kevin") {
                    customers[i].prepaymentCompany = 0;
                } else if(token == "Bob") {
                    customers[i].prepaymentCompany = 1;
                } else if(token == "Stuart") {
                    customers[i].prepaymentCompany = 2;
                } else if(token == "Otto") {
                    customers[i].prepaymentCompany = 3;
                } else if(token == "Dave") {
                    customers[i].prepaymentCompany = 4;
                }
            } else if(j == 3) {
                customers[i].amount = stoi(token);
            }
            j++;
        }
    }
```

The first part of the main function accesses the input file via the command line arguments and creates the output file with the desired name format.
After getting the number of customers from the first line of the input file, it iterates through the remaining lines and creates the customer struct instances by accessing the necessary fields with the help of stringstream.

```
output_write_lock = PTHREAD_MUTEX_INITIALIZER;  //initialize locks
for(int i = 0; i < 10; i++) {
    vending_machine_lock[i] = PTHREAD_MUTEX_INITIALIZER;
    customer_lock[i] = PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_lock(&vending_machine_lock[i]);   //firstly lock vending machines and don't let them run until customers are assigned
}

pthread_attr_t attr;
pthread_attr_init(&attr);

pthread_t vending_machine_threads[10];
pthread_t customer_threads[numberOfCustomers];

int vending_ids[10];

for(int i = 0; i < 10; i++) {
    vending_ids[i] = i+1;
    pthread_create(&vending_machine_threads[i], &attr, threadVendingMachine, &vending_ids[i]);  //create vending machine threads
}

for(int i = 0; i < numberOfCustomers; i++) {
    pthread_create(&customer_threads[i], &attr, threadCustomer, &customers[i]); //create customer threads
}

for(int i = 0 ; i < numberOfCustomers ; i++){
    pthread_join(customer_threads[i], NULL);    //wait for all customers to finish
}

for(int i = 0 ; i < 10 ; i++){
    pthread_join(vending_machine_threads[i], NULL); //wait for all vending machines to finish
}
```

In the second part of the main function, all mutex locks are initialized first. The lock that blocks the vending machines is locked after initialization, so the vending machine threads are locked until the customers are assigned to the vending machine.

Then threads are created for customers and vending machines.

Program waits until all threads are completed to execute the last part of the program.

```
    pthread_mutex_lock(&output_write_lock);  //write total amount of each company to output file
    outputf<<"All prepayments are completed."<<endl;
    outputf<<"Kevin: "<<totalAmount[0]<<"TL"<<endl;
    outputf<<"Bob: "<<totalAmount[1]<<"TL"<<endl;
    outputf<<"Stuart: "<<totalAmount[2]<<"TL"<<endl;
    outputf<<"Otto: "<<totalAmount[3]<<"TL"<<endl;
    outputf<<"Dave: "<<totalAmount[4]<<"TL"<<endl;
    pthread_mutex_unlock(&output_write_lock);

    inputf.close();
    outputf.close();

    for (int i = 0; i < 10; i++) {  //destroy locks
        pthread_mutex_destroy(&vending_machine_lock[i]);
        pthread_mutex_destroy(&customer_lock[i]);
        if (i < 5) {
            pthread_mutex_destroy(&total_amount_lock[i]);
        }
    }
    pthread_mutex_destroy(&output_write_lock);

    return 0;
}
```

In the last part of the program, the total prepayments made to the companies are taken from the global variables and written to the output file.

Files are closed and mutexes are destroyed. Program returns 0.

3- threadCustomer

```
void *threadCustomer(void *param)  //customer thread runner
{
    customer cus = *((struct customer *) param);
    usleep(cus.sleepTime * 1000);   //sleep for customer's sleep time

    pthread_mutex_lock(&customer_lock[cus.ticketVendingMachineInstance]);
    cus_at_vending[cus.ticketVendingMachineInstance] = cus;     //assign customer to vending machine
    pthread_mutex_unlock(&vending_machine_lock[cus.ticketVendingMachineInstance]);

    pthread_exit(0);
}
```

This function is the thread runner for the customer threads. It sleeps for the specified time.

Takes customer data as parameter. Before assigning the customer to the specified vending machine locks the customer_lock. Makes sure no other customer thread is assigning to same

vending machine until assignment operation is done. After assignment it unlocks the vending_machine_lock so that vending machine can makes necessary operations. Finally exits the thread.

Note: The assignment operation is performed over the cus_at_vending global array.

4- threadVendingMachine

```
void *threadVendingMachine(void *param)  //vending machine thread runner
{
    int vending_id = *((int *) param);
    while(machine_volume[vending_id] > 0) {  //while there are still prepayment actions to be done

        if(cus_at_vending[vending_id].id == 0){
            continue;
        }

        pthread_mutex_lock(&vending_machine_lock[vending_id]);
        customer cus = cus_at_vending[vending_id];        //get customer info
        machine_volume[vending_id]--;
        pthread_mutex_unlock(&customer_lock[vending_id]);

        pthread_mutex_lock(&total_amount_lock[cus.prepaymentCompany]);
        totalAmount[cus.prepaymentCompany] += cus.amount;       //update total amount
        pthread_mutex_unlock(&total_amount_lock[cus.prepaymentCompany]);

        string prepaymentCompany;
        if(cus.prepaymentCompany == 0) {
            prepaymentCompany = "Kevin";
        } else if(cus.prepaymentCompany == 1) {
            prepaymentCompany = "Bob";
        } else if(cus.prepaymentCompany == 2) {
            prepaymentCompany = "Stuart";
        } else if(cus.prepaymentCompany == 3) {
            prepaymentCompany = "Otto";
        } else if(cus.prepaymentCompany == 4) {
            prepaymentCompany = "Dave";
        }

        pthread_mutex_lock(&output_write_lock);
        outputf<<"Customer"<<cus.id<<","<<cus.amount<<"TL,"<<prepaymentCompany<<endl;   //write to output file
        pthread_mutex_unlock(&output_write_lock);

        if(machine_volume[vending_id] == 0){  //if there are no more prepayment actions to be done for this vending machine free the lock
            pthread_mutex_unlock(&vending_machine_lock[vending_id]);
        }

    }
    pthread_exit(0);
}
```

This function is the thread runner for the vending machine threads.
Takes the vending machine id as a parameter. It runs while there are still prepayment actions to be done at this machine.
Before accesing the customer at cus_at_vending array it locks the vending_machine_lock. And after getting customer data unlocks the customer_lock so next customers can continue to assign this machine.

Updates the totalAmount array with total_amount_lock so makes sure no other thread is updating same company value at the same time.

It prepares the log and prints it to the output file.
With output_write_lock it makes sure no other thread is writing to output file at the same time.
Finally it controlls the machine volume and if work is done unlocks it if it’s locked.

When machine volume (queue) ends, exits the thread.