MARMARA UNIVERSITY

CSE 4034 – Advanced Unix Programming

Programming Assignment # 1

Group Members
150114040 – Taha Bilal Özbey
150115009 - Ferhat Özkan
150116828 – Emre Kumaş

Supervised by
Assistant Professor Betül Demiröz Boz
06.05.2019

Implementation Details:

```
int main(){

    //Before beginning we need to make sure that the program create different random numbers on every sequence.
    srand((unsigned int) time(0));

    //Reading the input file.
    read_file();

    //Creating necessary variables.
    create_necessary_variables();

    //Creating the threads.
    create_threads();

    //Managing threads;
    manage_threads();

    //When the job is done, we need to join all threads.
    join_threads();

    //Printing the summary and cleaning up spaces allocated.
    print_summary();
    clean_up();

    return 0;
}
```

- The function *srand()* is used to create a seed for random numbers that will be generated in the code.
- The function *read_file()* reads the file and fills parameters such as *number_of_customers, number_of_sellers, number_of_products* etc.
- The function *create_necessary_variables()* creates several variables according to the variables extracted from the input file.
- The function *create_threads()* function creates threads and mutex locks. We have two kind of threads seller threads and customer threads. These threads are all together in our *customer_tread* and *seller_thread* arrays. We have *transaction_mutex, reserve_mutex* and an array of mutexes called *seller_mutex* (Each seller has a mutex associated with itself).
- The function *manage_threads()* keeps running till the given number of simulation days are completed. After each simulation day it reads the file again and initializes the all variables to their initial value. It cleans the *reserve_struct* which stores all reservations. At the end of the day all reservations will be discarded and initialized as no reservations are existing.
- The function *join_threads()* is used to join all threads created and destroy all mutexes created.
- The function *print_summary()* is used to print all logs.
- The function clean_up() free's all memory that has been allocated for variables used in the code.

Detailed Information about the core functions

Our core functions are *manage_threads()*, *customer_thread()* and *seller_thread()*.

manage_threads()

```
void manage_threads(){

    //We created all threads. Since 10 seconds correspond to one simulation day, we need to suspend the main thread for 10 seconds.
    while(current_simulation_day < number_of_simulation_days){
        sleep(1);
        current_simulation_day++;
        printf("Day %d ended.\n",current_simulation_day);

        //After the current day finishes, we need to reset all information.
        //To be able to do that, we need to wait for all customers and sellers to finish their current job.
        for(int i = 0; i < number_of_sellers; i++){
            while(seller_to_customer[i] != -1);
        }

        //Now, all jobs has finished, we can reset all values back to original.
        read_file();

        //We need to clear the reserve list, as well.
        clean_reserve_list();

        //Also, we will set the current days flag to true if the simulation has not ended.
        if(current_simulation_day != number_of_simulation_days) current_day_initialized[current_simulation_day] = true;
    }
}
```

We determined a simulation day as 1 sec. This time is enough for the threads to complete their operations. After all operations are done and the system is stable we increment the *current_simulation_day* and wait till all customers and sellers finish their job. Then initialize all values for a new day and clean also the reserve list since reservations are discarded at the end of each day in our system.

customer_thread()

```
void *customer_thread(void *argument){

    int i = 0;
    int operation_type, product_type, product_amount;
    int status = -1;
    int customer_no = (int)(intptr_t) argument;

    //While the simulation days is not over...
    while(current_simulation_day != number_of_simulation_days){

        //Before doing anything, we should check if the current day has been initialized.
        while(current_day_initialized[current_simulation_day] == false);

        int current_day = current_simulation_day;

        if(customer_status[customer_no] == false){

            //If this is the case, this customer cannot do anything else until the current day finishes. So we need to suspend it.
            while(current_day == current_simulation_day);

            goto end_of_customer;
        }

        //For each operation, we need another random number for the type of the operation. We have 3 different operations.
        operation_type = (int) random() % 3;
```

The customer thread runs till we have reached the number of simulation days given in the input file. Before the customer thread starts to operate it checks whether the *current_simulation_day* is initialized or not. The *current_day_initialized* is a boolean array witch each value initialized as false except the first day. After each day finishes the *manage_threads()* function sets the *current_day_initialized[current_simulation_day]* to true. By doing so the threads all wait in a while loop

till the new day is initialized and they are allowed to operate. Then we check if the customer status is false. The customer status can only be set to *false* if the customer has no more operation resources. This means the customer can't do any operations till the next day. If this is the case the customer thread waits in the while loop till the *current_simulation_day* has been re-initialized to the next day.

If the thread can overcome these controls, it can do an operation for that day. It produces a number from 0 to 2 identifying one of the following operations: *BUY PRODUCT, RESERVE PRODUCT* and *CANCEL RESERVATION*

```
    if(operation_type == 0){ // BUY PRODUCT

        product_type = (int) random() % number_of_products;
        product_amount = (int) (random() % 5) + 1;

        customers[customer_no].operation_type = 0;
        customers[customer_no].product_type = product_type;
        customers[customer_no].product_amount = product_amount;

    }else if(operation_type == 1){ // RESERVE PRODUCT

        product_type = (int) random() % number_of_products;
        product_amount = (int) (random() % 5) + 1;

        customers[customer_no].operation_type = 1;
        customers[customer_no].product_type = product_type;
        customers[customer_no].product_amount = product_amount;

    }else{ // CANCEL RESERVATION

        customers[customer_no].operation_type = 2;
    }

    //Customer needs to find an empty seller as long as we're on the same day.
    while(current_day == current_simulation_day){

        for(i = 0; i < number_of_sellers; i++){

            status = pthread_mutex_trylock(&seller_mutex[i]);

            //If we able to lock anything which means we found an empty seller, we will exit the loop.
            if(status != EBUSY)
                break;
        }

        if(status != EBUSY)
            break;
    }

    //We need to reset the loop if we're not on the same day.
    if(current_day != current_simulation_day){
        if(status != EBUSY && status != -1) pthread_mutex_unlock(&seller_mutex[i]);
        continue;
    }

    //Now, we need a way to communicate with the seller. So we set the arrays value.
    seller_to_customer[i] = customer_no;

    //We need to keep the customer waiting until its job finishes.
    while(seller_to_customer[i] == customer_no);

    end_of_customer:;
}

pthread_exit(NULL);
}
```

We used an array of customer struct called *customers*. This array contains the information of all customers current operations. For *BUY PRODUCT* and *RESERVER PRODUCT* the *customer_thread* randomly picks a

product type (the range depends on the input file) and a product amount between 1 and 5. After doing so it fills the customer struct array (index is its customer_id) with these values and operation type. For *CANCEL RESERVATION* it only sets the operation type of the struct nothing else is needed. Now the customer thread has to find a seller that is idle at the moment. While we are still in the up to date day we search in a for loop for such a seller. If we find an idle seller and we are sure that the day still doesn't changed we set a value of the *seller_to_customer* array (index is *sellers_id*). This is array is later used in *seller_thread* to identifiy which customer is doing the request to the *seller_thread*. Now the *customer_thread* waits till his request is handled by the *seller_thread* in a while loop. The *seller_thread* will set the value of *seller_to_customer[i]* to -1 when it has handled the request. This is the basic routine for the *customer_thread*.

seller_thread()

```
void *seller_thread(void *argument){

    int customer_to_serve, operation_type;
    int seller_no = (int)(intptr_t) argument;

    //While the simulation days is not over...
    while(current_simulation_day != number_of_simulation_days){

        //The seller waits for a customer to lock its mutex until the end of its execution.
        while(seller_to_customer[seller_no] == -1)
            if(current_simulation_day == number_of_simulation_days) goto end_of_seller;

        //Now, the seller can do a job.
        customer_to_serve = seller_to_customer[seller_no];
        operation_type = customers[customer_to_serve].operation_type;

        //Firstly, we need to check if that customer has any operation right.
        if(customer_information[customer_to_serve][1] <= 0){

            //Means the customer has no right to do its job.
            customer_status[customer_to_serve] = false;
```

The *seller_thread* runs till all simulation days are over. It waits in a while loop till a customer has done a request on that *seller_thread*. The seller catches the request by constantly checking the *seller_to_customer* array (index is the *seller_no*). If the value is -1 meaning it has no request it waits. If that's not the case it means that a customer has done a request and written its id in the index. When the seller catches a request it first starts by extracting the id of the customer that has done a request. Know it has to know which operation was requested. But before doing any operation it has to check whether the customer has any operation right left or not. If not the seller sets the customer status to false meaning this customer can no longer do any request for that day.

```
}else{

    if(operation_type == 0){ // BUY PRODUCT

        //We need to check if that amount exists.
        if(num_of_instances_of_product[customers[customer_to_serve].product_type] < customers[customer_to_serve].product_amount){

            //Means wanted amount do not exist. Unsuccessful transaction.
            add_to_transaction_list(create_transaction(customer_to_serve, current_simulation_day, false, seller_no));

        }else{

            //We can make this transaction successfully.
            add_to_transaction_list(create_transaction(customer_to_serve, current_simulation_day, true, seller_no));

            //Lastly, we need to decrease the amount from product amount.
            pthread_mutex_lock(&product_mutex[customers[customer_to_serve].product_type]);
            num_of_instances_of_product[customers[customer_to_serve].product_type] -= customers[customer_to_serve].product_amount;
            product_sales[customers[customer_to_serve].product_type][0] += customers[customer_to_serve].product_amount;
            pthread_mutex_unlock(&product_mutex[customers[customer_to_serve].product_type]);
        }

    }else if(operation_type == 1){ // RESERVE PRODUCT

        //We need to check if that amount exists.
        if(num_of_instances_of_product[customers[customer_to_serve].product_type] < customers[customer_to_serve].product_amount){

            //Means wanted amount do not exist. Unsuccessful transaction.
            add_to_transaction_list(create_transaction(customer_to_serve, current_simulation_day, false, seller_no));

        }else if(customer_information[customer_to_serve][2] < customers[customer_to_serve].product_amount){

            //Secondly, we need to check if customers reserve amount is enough. If this is the case, unsuccessful transaction.
            add_to_transaction_list(create_transaction(customer_to_serve, current_simulation_day, false, seller_no));

        }else{

            //We can make this transaction successfully.
            add_to_transaction_list(create_transaction(customer_to_serve, current_simulation_day, true, seller_no));

            //We need to decrease the amount from product amount.
            pthread_mutex_lock(&product_mutex[customers[customer_to_serve].product_type]);
            num_of_instances_of_product[customers[customer_to_serve].product_type] -= customers[customer_to_serve].product_amount;
            product_sales[customers[customer_to_serve].product_type][1] += customers[customer_to_serve].product_amount;
            pthread_mutex_unlock(&product_mutex[customers[customer_to_serve].product_type]);

            //Also, we need to add this to the reserve list.
            add_to_reserve_list(create_reserve(customer_to_serve));

            //We need to decrease from customers allowed reservation count.
            customer_information[customer_to_serve][2] -= customers[customer_to_serve].product_amount;
        }
```

If the seller thread has received a request that is valid to be operated, it checks which operation is requested. The request types were mentioned in the *customer_thread* detailed explanation. For BUY PRODUCT request the seller checks the store and checks whether the requested amount is available in the store or not. If it is not available, we create a transaction and add it to the *transaction_list*. This list will be later used for printing all transactions. If the amount is available in the store, we lock the *product_mutex* since only one thread should edit *num_of_sales* and *product_sales* arrays at the same time. Same product mutex is used for *RESERVE PRODUCT* operation and *CANCEL RESERVATION* operations.

```
        }else{ // CANCEL RESERVATION

            int product_amount = cancel_reservation(customer_to_serve);
            bool is_successful;

            if(product_amount == -1) is_successful = false;
            else is_successful = true;

            //We need to make the transaction.
            add_to_transaction_list(create_transaction(customer_to_serve, current_simulation_day, is_successful, seller_no));

            pthread_mutex_lock(&product_mutex[customers[customer_to_serve].product_type]);
            if(is_successful == true)
                product_sales[customers[customer_to_serve].product_type][2] += product_amount;
            pthread_mutex_unlock(&product_mutex[customers[customer_to_serve].product_type]);
        }

        //We need to decrease from customers allowed operation count.
        customer_information[customer_to_serve][1]--;
    }

    //Since seller finished its job, we need to reset its status.
    seller_to_customer[seller_no] = -1;

    //We will release the mutex.
    pthread_mutex_unlock(&seller_mutex[seller_no]);

    end_of_seller:;
}

pthread_exit(NULL);
```