Computer Operating Systems Assignment 2

Project Report

BLG 312E

CRN: 23148

Beyza Aydeniz - 150200039

May 13, 2023

1 Introduction

The goal of this assignment is to develop an online shopping routine using the principles of multiprocessing and multithreading in operating systems. Since multiple customers may order simultaneously, this can lead to a race condition issue. Thus, a proper synchronization mechanism must be implemented to manage the flow of customers purchasing the same product at the same time, and prevent any race conditions. This assignment requires implementing a fully functional online shopping routine using mutexes to ensure proper synchronization.

Except for the main part, the other parts of my code are almost the same. The first function takes a void pointer as input, which is then cast into a pointer of type Customer using (Customer*) ptr. This means that the input argument can be of any type, and the function will try to interpret it as a pointer to a Customer struct. On the other hand, the second function takes a pointer to a Customer struct as input, which means that the input argument must be a pointer to a Customer struct.

```c
void* order_products(void * ptr){

    Customer *args = (Customer*) ptr;
    int ordered_quantity; //will be "Customer->ordered_Items[i].product_Quantity", used to make the code more readable
    int available_quantity; //will be "Products[Customer->ordered_Items[i].product_ID-1].product_Quantity", used to make the code more readable
    double cost; //will be "(Customer->ordered_Items[i].product_Quantity)*(Products[Customer->ordered_Items[i].product_ID-1].product_Price)", used to make the code more readable
    int initial_product_quantity[Num_of_Products];


    for(int i = 0; i<args->ordered_item_type_quantity; i++){


        pthread_mutex_lock(&lock[args->ordered_Items[i].product_ID-1]); //locks the product by checking its product ID. Product x is accessed by &lock(x-1).

        ordered_quantity = args->ordered_Items[i].product_Quantity;
        available_quantity = Products[args->ordered_Items[i].product_ID-1].product_Quantity;
        cost = (args->ordered_Items[i].product_Quantity)*(Products[args->ordered_Items[i].product_ID-1].product_Price);

        if(ordered_quantity<= available_quantity && cost<=args->customer_Balance){ //if there are enough money and quantity to purchase

            for(int j = 0; j<Num_of_Products; j++){
                initial_product_quantity[j]=Products[j].product_Quantity; //first it keeps the initial number of items to print later
            }

            order_product(args->ordered_Items[i].product_ID, ordered_quantity);


            args->purchased_Items[args->purchased_item_count]=args->ordered_Items[i]; //if the purchase is successful, it adds it to the array of purchases

            args->purchased_item_count++;
            args->customer_Balance -= cost;


            pthread_mutex_lock(&writelock); //write_lock to not interrupt outputs
            printf("Customer%d SHOPPING\n", args->customer_ID);
            printf("Initial Products \n");
            printf("Product ID  Quantity    Price \n");
            for(int j = 0; j <Num_of_Products; j++){
                printf("    %d          %d        %f\n", Products[j].product_ID, initial_product_quantity[j], Products[j].product_Price);
            }

            printf("Bought %d of product %d for %f. \n", ordered_quantity, args->ordered_Items[i].product_ID, cost);
            printf("\n");
            printf("Updated Products \n");
            printf("Product ID  Quantity    Price \n");
            for(int j = 0; j <Num_of_Products; j++){
                printf("    %d          %d        %f\n", Products[j].product_ID, Products[j].product_Quantity, Products[j].product_Price);
            }
            pthread_mutex_unlock(&writelock);
        }

        else if(ordered_quantity>available_quantity){ //if there is not enough stock to purchase
            pthread_mutex_lock(&writelock); //write_lock to not interrupt outputs
            printf("Fail! Customer %d cannot buy %d of product %d. Only %d left in stock.\n", args->customer_ID, ordered_quantity, args->ordered_Items[i].product_ID, available_quantity);
            pthread_mutex_unlock(&writelock);
        }
        else if(cost>args->customer_Balance){ //if there is not enough money to purchase
            pthread_mutex_lock(&writelock); //write_lock to not interrupt outputs
            printf("Fail! Customer %d cannot buy %d of product %d. Insufficient funds.\n", args->customer_ID, ordered_quantity, args->ordered_Items[i].product_ID);
            pthread_mutex_unlock(&writelock);
        }
        pthread_mutex_unlock(&lock[args->ordered_Items[i].product_ID-1]); //unlocks the product so that other costumers can purchase too

    }
}
```

Figure 1: Implementation of ordering multiple products in multithreading.c

2 Multithreading

In multithreading I allocated dynamic memory so that customers' data is not lost after using pthread join. I used mutex to perform a properly working online shopping routine. I created a for loop and lock the product by checking its product ID. Product x is locked by lock(x-1). Also I created write lock so that the texts do not interfere with each other.

```
Fail! Customer 1 cannot buy 5 of product 1. Only 2 left in stock.
Fail! Customer 2 cannot buy 7 of product 2. Only 4 left in stock.
Fail! Customer 3 cannot buy 5 of product 2. Only 4 left in stock.
Customer1 SHOPPING
Initial Products
Product ID  Quantity    Price
    1         2       15.000000
    2         4       26.000000
    3         5       5.000000
    4         2       14.000000
    5         5       7.000000
Bought 2 of product 5 for 14.000000.

Updated Products
Product ID  Quantity    Price
    1         2       15.000000
    2         4       26.000000
    3         5       5.000000
    4         2       14.000000
    5         3       7.000000
Fail! Customer 2 cannot buy 6 of product 2. Only 4 left in stock.
Customer3 SHOPPING
Initial Products
Product ID  Quantity    Price
    1         2       15.000000
    2         4       26.000000
    3         5       5.000000
    4         2       14.000000
    5         3       7.000000
Bought 1 of product 2 for 26.000000.

Updated Products
Product ID  Quantity    Price
    1         2       15.000000
    2         3       26.000000
    3         5       5.000000
    4         2       14.000000
    5         3       7.000000
Fail! Customer 1 cannot buy 9 of product 3. Only 5 left in stock.
Customer3 SHOPPING
Initial Products
Product ID  Quantity    Price
    1         2       15.000000
    2         3       26.000000
    3         5       5.000000
    4         2       14.000000
    5         3       7.000000
Bought 1 of product 2 for 26.000000.

Updated Products
Product ID  Quantity    Price
    1         2       15.000000
    2         2       26.000000
    3         5       5.000000
    4         2       14.000000
    5         3       7.000000
Fail! Customer 3 cannot buy 2 of product 4. Insufficient funds.
Fail! Customer 3 cannot buy 10 of product 1. Only 2 left in stock.
```

Figure 2: Sample output of multithreading.

```
Customer1 Information

---------------------
Initial Balance: $ 182.000000
Updated Balance: $ 168.000000
Ordered Products
ID      Quantity
1       5
5       2
3       9
Purchased Products
ID      Quantity
5       2

Customer2 Information

---------------------
Initial Balance: $ 120.000000
Updated Balance: $ 120.000000
Ordered Products
ID      Quantity
2       7
2       6
Purchased Products
ID      Quantity
-       -

Customer3 Information

---------------------
Initial Balance: $ 62.000000
Updated Balance: $ 10.000000
Ordered Products
ID      Quantity
2       5
2       1
2       1
4       2
1       10
Purchased Products
ID      Quantity
2       1
2       1
```

Figure 3: Sample output of multithreading.

3 Multiprocessing

To provide a synchronous online shopping experience, I first created a shared memory segment and attached it. Once that was done, I implemented a child process using fork to create three customers. After that, I waited for the child processes to complete. To prevent any issues with multiple customers trying to purchase the same product simultaneously, I used a mutex. I created a for loop to iterate over the products and locked each product by checking its product ID. In this way, I ensured that product x was locked by lock(x-1).

```
Fail! Customer 1 cannot buy 3 of product 5. Only 2 left in stock.
Customer1 SHOPPING
Initial Products
Product ID  Quantity    Price
    1         2       3.000000
    2         4       15.000000
    3         3       9.000000
    4         5       25.000000
    5         2       17.000000
Bought 4 of product 4 for 100.000000.

Updated Products
Product ID  Quantity    Price
    1         2       3.000000
    2         4       15.000000
    3         3       9.000000
    4         1       25.000000
    5         2       17.000000
Customer1 SHOPPING
Initial Products
Product ID  Quantity    Price
    1         2       3.000000
    2         4       15.000000
    3         3       9.000000
    4         1       25.000000
    5         2       17.000000
Bought 1 of product 2 for 15.000000.

Updated Products
Product ID  Quantity    Price
    1         2       3.000000
    2         3       15.000000
    3         3       9.000000
    4         1       25.000000
    5         2       17.000000
Fail! Customer 1 cannot buy 6 of product 2. Only 3 left in stock.
Fail! Customer 2 cannot buy 8 of product 2. Only 4 left in stock.
Fail! Customer 2 cannot buy 4 of product 4. Insufficient funds.
Customer2 SHOPPING
Initial Products
Product ID  Quantity    Price
    1         2       3.000000
    2         4       15.000000
    3         3       9.000000
    4         5       25.000000
    5         2       17.000000
Bought 1 of product 1 for 3.000000.

Updated Products
Product ID  Quantity    Price
    1         1       3.000000
    2         4       15.000000
    3         3       9.000000
    4         5       25.000000
    5         2       17.000000
Fail! Customer 2 cannot buy 7 of product 3. Only 3 left in stock.
Fail! Customer 2 cannot buy 10 of product 3. Only 3 left in stock.
Fail! Customer 3 cannot buy 9 of product 5. Only 2 left in stock.
Fail! Customer 3 cannot buy 4 of product 4. Insufficient funds.
```

Figure 4: Sample output of multiprocessing.

```
Customer1 Information
---------------------
Initial Balance: $ 149.000000
Updated Balance: $ 34.000000
Ordered Products
ID      Quantity
5       3
4       4
2       1
2       6
Purchased Products
ID      Quantity
4       4
2       1

Customer2 Information
---------------------
Initial Balance: $ 17.000000
Updated Balance: $ 14.000000
Ordered Products
ID      Quantity
2       8
4       4
1       1
3       7
3       10
Purchased Products
ID      Quantity
1       1

Customer3 Information
---------------------
Initial Balance: $ 64.000000
Updated Balance: $ 64.000000
Ordered Products
ID      Quantity
5       9
4       4
Purchased Products
ID      Quantity
-       -
```

Figure 5: Sample output of multiprocessing.

4 Discussion

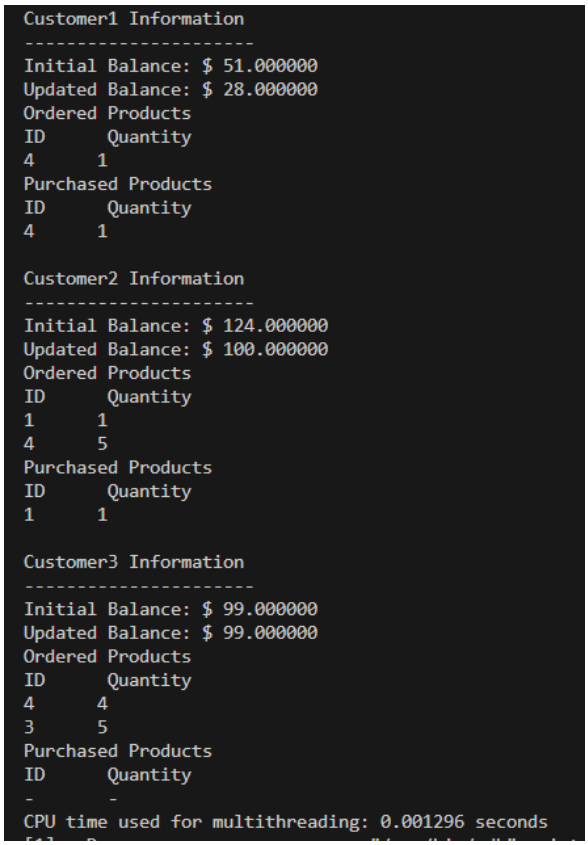

Figure 6: 0.001296 second execution time for multithreading.

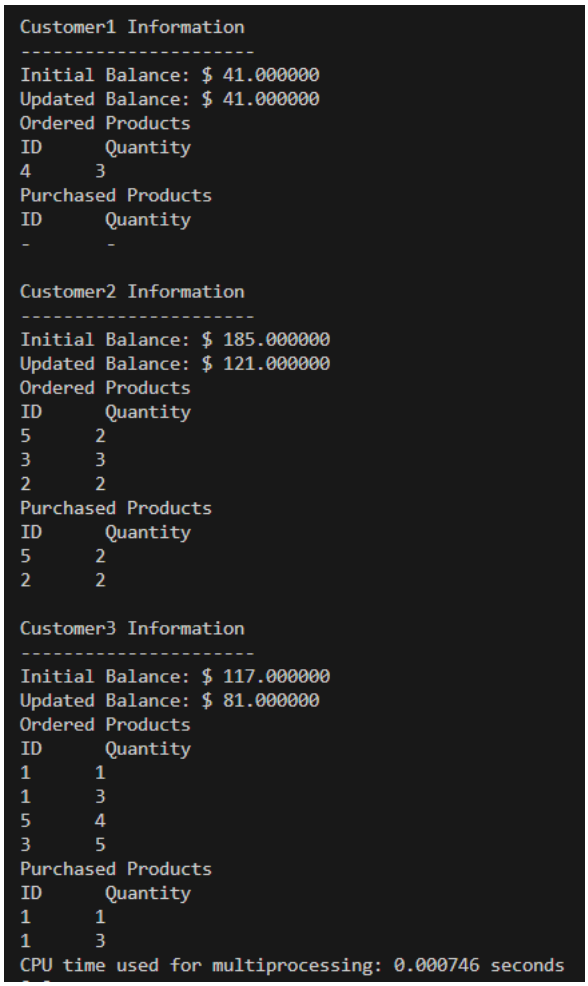

Figure 7: 0.000746 second execution time for multiprocessing.

Different from what I expected, despite requiring more inputs and processes than the multithreading method, the multiprocessing method was able to complete the task almost twice as fast as we see from figure 6 and figure 7.

Multiprocessing is the more efficient method than multithreading for a 1000-active-customers online shopping system. Multithreading can be limited according to the the number of available processors and can result in performance degradation if too many threads are created, leading to overhead costs such as thread creation and context switching. On the other hand multiprocessing allows the system to utilize multiple processors simultaneously which is proper for larger processes and datas.