Computer Operating Systems Assignment 2 Report

Emre Çamlıca - 150210071

1 Introduction

In this homework, I designed a concurrent online shopping routine simulator using multithreading and multiprocessing concepts of the C programming language.

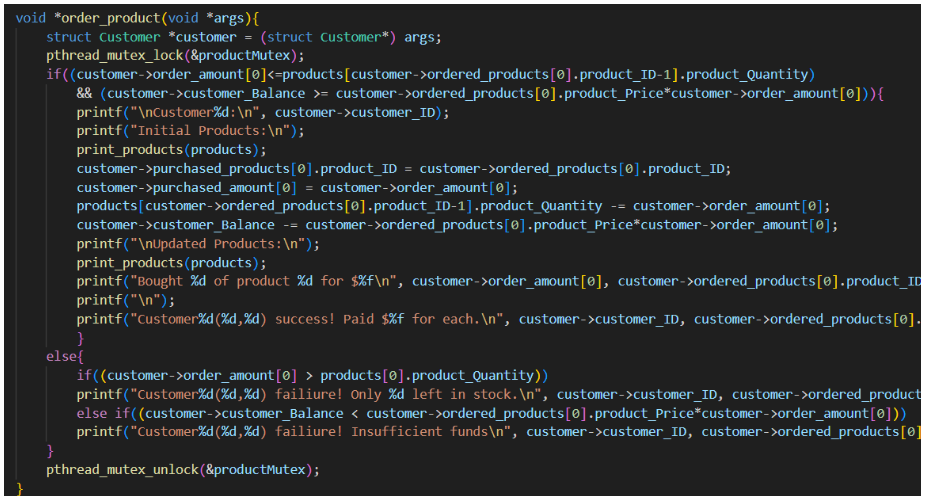
2 Explanation

2.1 Multithreading

In the multithreading part, I first defined two structs named "Product" and "Consumer". My program has 5 products and 3 consumers indicated by the global definitions of product and customer sizes. I used a global array of products in order to share resource between the customers. In the main function, I initialized the array of products and created an array of customers of size 3 and Initialized them the way it is specified in the homework, with the help of "srand()" function. For the ordered products array, I initialized the product ID's to 0 initially, before randomly initializing some of them, in order to use these 0 values for condition checks later on. Each customer also have their 2 arrays of ordered and purchased products, with corresponding integer arrays accounting for the amount of products they ordered/purchased. I also initialized the mutex that I send to the product ordering functions in the main function.

After the initialization step, I used a for loop to create threads that simulate the online shopping routine. In each iteration, I sent the address of one of the two product ordering functions to the thread creating function, depending on the number of products they ordered. Furthermore, I sent the address of customer struct, which is one of the elements of the customer struct array I created, to the product ordering functions.

1 shows the single product ordering function for the multithreading part. This function takes a reference to a customer struct as parameter as a void pointer and casts it back to a pointer to a customer struct. Then, before entering the critical section, it locks the global product mutex. Using an if statement it checks the availability of necessary amount of products and the customer balance. If both are not satisfied, it writes the failiure messages to screen. If



```c
void *order_product(void *args){
    struct Customer *customer = (struct Customer*) args;
    pthread_mutex_lock(&productMutex);
    if((customer->order_amount[0]<=products[customer->ordered_products[0].product_ID-1].product_Quantity)
        && (customer->customer_Balance >= customer->ordered_products[0].product_Price*customer->order_amount[0])){
        printf("\nCustomer%d:\n", customer->customer_ID);
        printf("Initial Products:\n");
        print_products(products);
        customer->purchased_products[0].product_ID = customer->ordered_products[0].product_ID;
        customer->purchased_amount[0] = customer->order_amount[0];
        products[customer->ordered_products[0].product_ID-1].product_Quantity -= customer->order_amount[0];
        customer->customer_Balance -= customer->ordered_products[0].product_Price*customer->order_amount[0];
        printf("\nUpdated Products:\n");
        print_products(products);
        printf("Bought %d of product %d for $%f\n", customer->order_amount[0], customer->ordered_products[0].product_ID
        printf("\n");
        printf("Customer%d(%d,%d) success! Paid $%f for each.\n", customer->customer_ID, customer->ordered_products[0].
        }
    else{
        if((customer->order_amount[0] > products[0].product_Quantity))
        printf("Customer%d(%d,%d) failiure! Only %d left in stock.\n", customer->customer_ID, customer->ordered_product
        else if((customer->customer_Balance < customer->ordered_products[0].product_Price*customer->order_amount[0]))
        printf("Customer%d(%d,%d) failiure! Insufficient funds\n", customer->customer_ID, customer->ordered_products[0]
    }
    pthread_mutex_unlock(&productMutex);
}
```

Figure 1: Order Product Function for Multithreading

one is not satisfied the function updates the purchased product array with the corresponding amount array of the customer. It also decreases the amount of products available to purchase. It prints the necessary messages on the screen. After the critical section is over, it unlocks the product mutex.

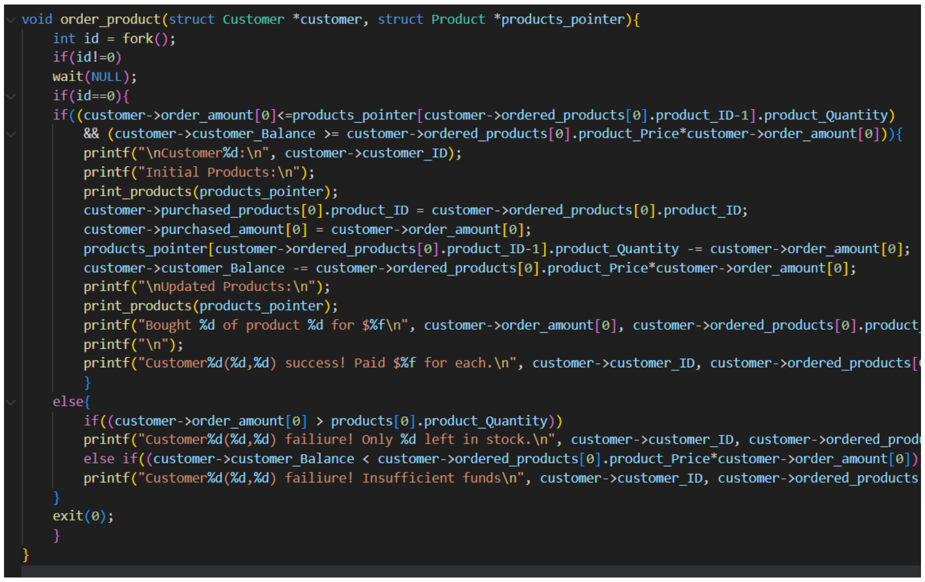
The function for ordering multiple products work in a similar way. The only difference is that it loops through the ordered products array of the customer using a while loop and an integer "i". Because I initialized the product ID's of the ordered products array to 0 while creating the customers, it was helpful to check the product ID's to decide when to stop. The loop also breaks when "i" reaches the maximum amount of products.

2.2 Multiprocessing

The code for multiprocessing is very similar to the multithreading one. This time I created two pairs of global products and customers pointers, one to be used as an array and one to be used to share memory. In the main function I used the "mmap" function to share memory between the parent and child processes. I initialized the products and the customers the same way I did in the multithreading part. After that, I again used a for loop to simulate the shopping routine. This time I sent the products array along with the address of a customer struct to the product ordering functions.

After creating the threads and simulating the routines, I used the "join" function to terminate the threads and destroyed the product mutex.

I created and used helpful functions to print the products and the customers. The function for printing products takes a product array and the customer printing function takes a pointer to struct customer as arguments. They both print the necessary products, looping through the product arrays.



```c
void order_product(struct Customer *customer, struct Product *products_pointer){
    int id = fork();
    if(id!=0)
    wait(NULL);
    if(id==0){
    if((customer->order_amount[0]<=products_pointer[customer->ordered_products[0].product_ID-1].product_Quantity)
        && (customer->customer_Balance >= customer->ordered_products[0].product_Price*customer->order_amount[0])){
        printf("\nCustomer%d:\n", customer->customer_ID);
        printf("Initial Products:\n");
        print_products(products_pointer);
        customer->purchased_products[0].product_ID = customer->ordered_products[0].product_ID;
        customer->purchased_amount[0] = customer->order_amount[0];
        products_pointer[customer->ordered_products[0].product_ID-1].product_Quantity -= customer->order_amount[0];
        customer->customer_Balance -= customer->ordered_products[0].product_Price*customer->order_amount[0];
        printf("\nUpdated Products:\n");
        print_products(products_pointer);
        printf("Bought %d of product %d for $%f\n", customer->order_amount[0], customer->ordered_products[0].product_
        printf("\n");
        printf("Customer%d(%d,%d) success! Paid $%f for each.\n", customer->customer_ID, customer->ordered_products[
        }
    else{
        if((customer->order_amount[0] > products[0].product_Quantity))
        printf("Customer%d(%d,%d) failiure! Only %d left in stock.\n", customer->customer_ID, customer->ordered_prod
        else if((customer->customer_Balance < customer->ordered_products[0].product_Price*customer->order_amount[0])
        printf("Customer%d(%d,%d) failiure! Insufficient funds\n", customer->customer_ID, customer->ordered_products
    }
    exit(0);
    }
}
```

Figure 2: Order Product Function for Multiprocessing

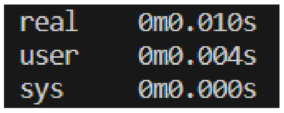
2 shows the single product ordering function for the multiprocessing part. The code looks very similar to its multithreading counterpart. The difference is that here I used “fork” and “wait” and “exit” functions instead of locks and mutexes. The function first creates an integer “id” variable to store the result of the “fork” call. If the ID is not 0, implying that it is the parent, it waits for the child to finish its execution. If the ID is 0, implying that it is the child, it executes the shopping routine. At the end of the routine, the child terminates using the “exit” function. This way it prevents other processes from entering the critical section. The multiple product ordering function works the same way.

At the end of the main function, I used the “munmap” function to deallocate the shared memory for products and customers arrays.

3 Discussion

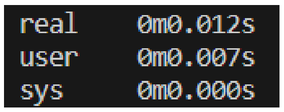
3.1

The figures 3 and 4 show the execution times for the multithreading and multiprocessing programs, respectively. It can be observed that the multithreading program executes a little bit faster, that might be because of the fact that the context switching between different processes take a longer time than the switching between different threads in the same process.



```
real    0m0.010s
user    0m0.004s
sys     0m0.000s
```

Figure 3: Excecution Time for the Multithreading Program



```
real    0m0.012s
user    0m0.007s
sys     0m0.000s
```

Figure 4: Excecution Time for the Multithreading Program

3.2

If the number of customers is as high as 10000, multithreading will be more efficient. Firstly, it will take shorter time to execute tasks in the multithreading system. Secondly, multiprocessing system will use more memory to allocate resources for each process, even though some of these resources will be shared, some of them will not. So it is more efficient to use multithreading for the case of high amount of customers.