BLG 312E Homework 2 Report

Mohamad Chahadeh - 150220901

09 - 05 - 2023

1 Introduction

In this Homework, We implemented an online shopping routine using two different methods to create the concurrency needed, these are **multi-threading**, and **multi-processing**.

1.1 Program Workflow

At the start of the program, 5 products with random prices and quantities, 3 customers with random balances are generated and stored in their appropriate storing mechanism. **then**, for each customer, a new branch, i.e. either a process or thread, is created, this branch will randomly generate a 1-4 item basket of products, **after that**, the ordering method is called taking in the basket as a parameter. The program keeps running and providing updates in the console until all orders are taken care of, it **then** gives the user the option to show a summary of all orders and their outcomes.

1.2 Differences Between Implementations

While both ways are meant to achieve the same outcome, There are a few key differences between the two approaches.

In the **multi-processing** approach, each **customer** obtains their own separate process that runs parallel to other processes, i.e. other customers.
as for the **multi-threading** approach, new **thread** is created **within the main process** for every single transaction that occurs.

1.3 Locking Mechanism

Mutexes are used in both implementation to apply a lock on the products that are being accessed, mutexes are available using the "pthread.h" library in C. The way the locking will take place is that, whenever an order is placed by a customer, the customer will attempt to obtain the mutex lock of that product, once the lock is obtained, the purchasing algorithm takes place and data is updated, and when that is done the customer releases the lock so that other customers can access the product.

1.4 Purchasing Algorithm

When the Customer orders a product or multiple products, either the **order_products()** or the **order_product()** function is called, for the **order_products()** function, the parameters of each order item is extracted and a for loop is used to iterate over them, ordering each one separately using the **order_product()** function. As for that, first, the customer_id, product_id, and quantities are extracted from the parameter **ThreadArgs** shown later, after that, a lock is obtained for the product specified, then, the purchase result is determined based on the following path:

1. If there isn't enough quantity in stock for the entire order, the transaction is fully terminated.

2. If there is enough stock, then if the user has insufficient funds to finance the entire order, the order is terminated for insufficient funds.

3. If the above contingencies are met, the transaction is moved forward with and the balance of the customer as well as the quantity of the product in stock is updated to their new values.

2 Implementation

The routines were implemented in C language on a Unix-based machine. The two routines have code snippets that are common between them.

2.1 Commonalities

2.1.1 Macros

```
#define NO_OF_PRODUCTS 5
#define NO_OF_CUSTOMERS 3
```

Figure 1: Macros of both programs

2.1.2 Customer and Product Object Structs

```
struct Product {
  int product_id;
  int price;
  int quantity_in_stock;
}

struct Customer {
  int customer_id;
  int balance;
  int ordered_items[99][2];
  int purchased_items[99][2];
  int ordered_items_size;
  int purchased_items_size;
}
```

Figure 2: Implementation of the Product and Customer Structs

2.1.3 ThreadArgs Object

The **ThreadArgs** Data object is used to pass order details to the order_product method.

```
typedef struct {
  int customer_id;
  int product_id;;
  int product_quantity;
  int direct;
} ThreadArgs;
```

Figure 3: Implementation of ThreadArgs Object

2.1.4 ArrayArgs Object

The **ArrayArgs** Data object is used to pass multiple orders to the order_products() method which is used to make multiple orders at once.

```
typedef struct {
  int customer;
  ThreadArgs* orders;
  int size;
} ArrayArgs;
```

Figure 4: ArrayArgs Implementation

2.1.5 Locking

Locking is done inside the order_product function using mutexes as shown below:

```
 ...
 pthread_mutex_lock(&(product_locks[product_id-1])); // obtaining the lock
 ...  //Purchasing Algorithm
 pthread_mutex_unlock(&(product_locks[product_id-1])); // releasing the lock
 ...
```

Figure 5: Implementation of mutexes in order_product function

2.2 Multi-processing Implementation

implementing the online shopping routine using the multi-processing method means that **separate processes should be created** for every customer or order so that all transaction can take place **asynchronously**, However, the processes created will **have separate memories allocated to them**, this means that changes applied by one process will only be visible to that process and the other processes will not see the changes. In order to resolve this issue, A **shared memory** that all processes can access needs to be created and accessed by these processes.

2.2.1 Shared Memory Object

A shared memory usually takes one data type and cannot store multiple data types at once. In order to resolve that, we need to create a new data type using the Typedef struct keyword so that we can store our data in any way we see fit.

```
typedef struct {
  pthread_mutex_t product_locks[NO_OF_PRODUCTS];
  struct Product products[NO_OF_PRODUCTS];
  struct Customer customers[NO_OF_CUSTOMERS];
} shared_data_t;
```

Figure 6: Shared Memory Object Implementation

The following is the data stored inside the shared memory object:

- **product_locks:** an array of mutexes for every product.
- **products:** an array containing the objects of all the products available for purchase.
- **customesr:** an array containing the objects of all customers.

2.2.2 Forking Processes

For each customer, a new process is forked and order_products() function is called after the orders of the customers are randomly generated real-time.

```
for(int i = 0; i<NO_OF_CUSTOMERS; i++) {
 ...
  else if(pid == 0){ // Child Process
    srand(time(NULL) ^ (getpid()<<16)); // new seed is generated for every process
    int no_of_orders = (rand()%4) + 1;
    ThreadArgs* orders = (ThreadArgs*) malloc(no_of_orders*sizeof(ThreadArgs));
    int customer = i;
    for(int j = 0; j<no_of_orders; j++) {
      orders[j].customer_id = customer;
      orders[j].product_id = (rand() % NO_OF_PRODUCTS);
      orders[j].product_quantity = (rand() % 5) + 1;
      orders[j].direct = 0;
    }
    all_args[i].customer = customer;
    all_args[i].orders = orders;
    all_args[i].size = no_of_orders;
    order_products(shmid, (void *) &all_args[i]);
    exit(0);
  } else continue; // parent process
}
```

Figure 7: Forking Method Implementation

2.3 Multi-threading Implementation

implementing the online shopping routine using the multi-threading method means that **threads within the main process** are created for every customer or order so that all transaction can take place **asynchronously**. Since These threads **exist within the same process**, global variables will be **shared** amongst them so any change of global variables in one thread will instantly **appear in all threads**, therefore, there is no need to implement a shared memory in this case, instead, we can just use global variables.

2.3.1 Thread Creation

For every customer, a random basket of orders is created, then a new thread is instantiated using the **pthread_create()** method. The method takes in the **order_products()** function as a thread function, and the basket as its parameters.

```
pthread_t my_threads[NO_OF_CUSTOMERS];

for(int i = 0; i< NO_OF_CUSTOMERS; i++) {

  int no_of_orders = rand()%4 + 1;
  ThreadArgs* orders = (ThreadArgs*) malloc(no_of_orders*sizeof(ThreadArgs));
  int customer = i;

  for(int i = 0; i<no_of_orders; i++) {
    orders[i].customer_id = customer;
    orders[i].product_id = (rand() % NO_OF_PRODUCTS);
    orders[i].product_quantity = (rand() % 4 + 1);
    orders[i].direct = 0;
  }

  ArrayArgs* array_args = (ArrayArgs*) malloc(sizeof(ArrayArgs));
  array_args->customer = customer;
  array_args->orders = orders;
  array_args->size = no_of_orders;

  int err1 = pthread_create(&my_threads[i], NULL, order_products, (void *)
      array_args);
  if(err1) {
    perror("thread create error\n");
    exit(1);
  }
}
```

Figure 8: Implementation and Creation of Threads

3 Discussion

As it was shown in the previous parts of this report, the two implementations have a lot of commonalities between them, and choosing between them in seek for better performance can seem challenging on the surface.

3.1 Which One is Faster?

By using the **gettimeofday()** function from the library **sys/time.h** we can get the time in milliseconds. By taking the time before and after a transaction, we can find the time elapsed of a transaction, here are the results:

1. **Multi-threading**: A single successful transaction took an average of **2.78 milliseconds**, while an unsuccessful transaction took 7 microseconds, or **0.007 milliseconds**

2. **Multi-processing**: A single successful transaction took an average of 530 microseconds or **0.53 milliseconds**, while an unsucessfull transaction took 7 microseconds or **0.007 milliseconds**.

3.2 Efficiency on Large Scales

It is clear from the results mentioned before that the m**multi-processing method is significantly faster** than the multi-threading method, and so based on that, it is safe to assume that **the multi-processing approach will perform better on a large scale application of 10000 active users**, and that's due to several other reasons:

- Utilization of multiple CPUs/cores.
- Better management of resources.
- Improved Scalability.