

BGL312E - Assignment 2

Ömer Malik Kalembası - 150180112

13/05/2023

1 Introduction

The subject of this assignment is to design an online shopping routine using multiprocessing and multithreading concepts in operating systems (OS). In an online shopping routine, more than one customer can order at the same time and this can cause a problem called race condition. When more than one customer wants to purchase the same product at the same time, this flow should be managed with a proper synchronization mechanism avoiding any race condition. Otherwise, the program may have some unexpected/inconsistent outputs. For instance, two customers may order product1 which is n in stock initially. They may reach and modify the same product (shared resource) concurrently and both processes/threads read the stock as n, and the count of the product may be decreased just one time instead of two times. To prevent race condition in OS, mutual exclusion methods are used. In this assignment, you are asked to perform a properly working online shopping routine using mutexes.

2 Multiprocess

The program is designed to simulate an online shopping routine, where multiple customers can simultaneously order products, potentially leading to a race condition. To avoid this, synchronization mechanisms like mutexes are used to ensure proper management of these concurrent requests.

The entire system is composed of various key components, represented by structs and functions. The **Customer** and **Product** structs represent customers and products, respectively, each containing relevant fields like customer ID, balance, ordered and purchased items for customers, and product ID, price, and quantity for products.

The `initialize_products()` and `initialize_customers()` functions are used to initialize the products and customers with random values. The product prices and quantities, as well as the customer balances, are assigned random values within specified ranges. The customer's ordered and purchased items are initialized to zero.

The `order_product()` function simulates a customer ordering a particular product in a specified quantity. It checks whether the product is in stock and

whether the customer has enough balance. If both conditions are met, it proceeds to deduct the price from the customer's balance, reduce the product quantity, and update the customer's ordered and purchased items. If the conditions are not met, it prints an appropriate error message.

The `order_products()` function is for ordering multiple products. It takes a customer and an array of quantities as arguments, and for each product, if the quantity is greater than zero, it calls the `order_product()` function to order that product.

The `print_initial_balances()`, `print_initial_inventory()`, and `print_latest_balances()` functions are used to print the initial balances of all customers, the initial inventory of all products, and the latest balances of all customers, respectively.

The main function initializes the customers and products by allocating shared memory for them and attaching them to the process address space. It then calls the initialization functions and the print functions to display the initial state. It then forks a new process for each customer, and each child process calls the `order_products()` function to order a set of products. The parent process waits for all child processes to finish, then prints the remaining inventory, the sales made by each customer, and the latest balances.

This program utilizes the concept of multiprocessing, where multiple processes are created using the `fork()` function, with each child process representing a customer placing an order. By using separate processes, the program can perform multiple operations concurrently, enhancing performance. Shared memory is used to allow these processes to communicate and share data (customers and products), with each process able to read and write to this shared memory. This shared memory is allocated with `shmget()` and attached to the process address space with `shmat()`. Finally, it is detached and deallocated with `shmdt()` and `shmctl()`.

Synchronization is crucial in a system like this to avoid race conditions, where two or more processes try to access and manipulate shared data simultaneously, leading to inconsistent and unpredictable results. This is achieved using mutex locks. Before accessing shared data (such as when ordering a product), a process must acquire a lock. If another process is already holding the lock, the process will block until the lock is released. Once it has finished manipulating the shared data, the process releases the lock, allowing other processes to acquire it.

3 Multithread

The `Customer` and `Product` structs representations, `initialize_products()`, `initialize_customers()`, `print_initial_balances()`, `print_initial_inventory()`, `print_latest_balances()` functions are similar with `multiprocess` which explained above.

The `order_product()` function simulates a customer ordering a particular product in a specified quantity. It checks whether the product is in stock and whether the customer has enough balance. If both conditions are met, it proceeds to deduct the price from the customer's balance, reduce the product quan-

tity, and update the customer's ordered and purchased items. If the conditions are not met, it prints an appropriate error message.

The `order_products()` function is for ordering multiple products. It takes a customer and an array of quantities as arguments, and for each product, if the quantity is greater than zero, it calls the `order_product()` function to order that product.

The main function initializes the customers and products, then creates a thread for each customer using the `pthread_create()` function, with each thread representing a customer placing an order. It then waits for all threads to finish using `pthread_join()` before it prints the remaining inventory, the sales made by each customer, and the latest balances.

This program utilizes the concept of multithreading, where multiple threads are created, each representing a customer. By using separate threads, the program can perform multiple operations concurrently, enhancing performance. Threads within a process share the same data space, which means they can access the memory of the same struct instances. This is beneficial because it allows easy communication between threads and also reduces the overhead of copying data.

Synchronization is crucial in a system like this to avoid race conditions, where two or more threads try to access and manipulate shared data simultaneously, leading to inconsistent and unpredictable results. This is achieved using mutex locks. Before accessing shared data (such as when ordering a product), a thread must acquire a lock using `pthread_mutex_lock()`. If another thread is already holding the lock, the thread will block until the lock is released. Once it has finished manipulating the shared data, the thread releases the lock using `pthread_mutex_unlock()`, allowing other threads to acquire it.

4 Compile & Run

You can compile both C programs with "make all" command.

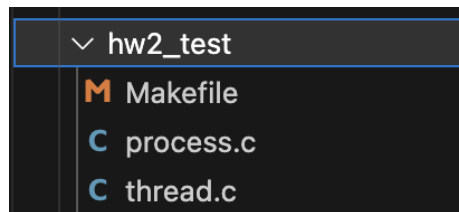
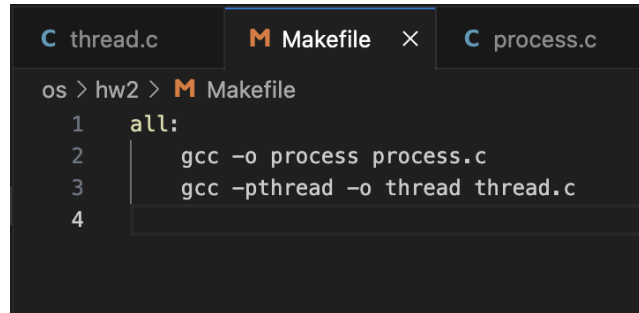
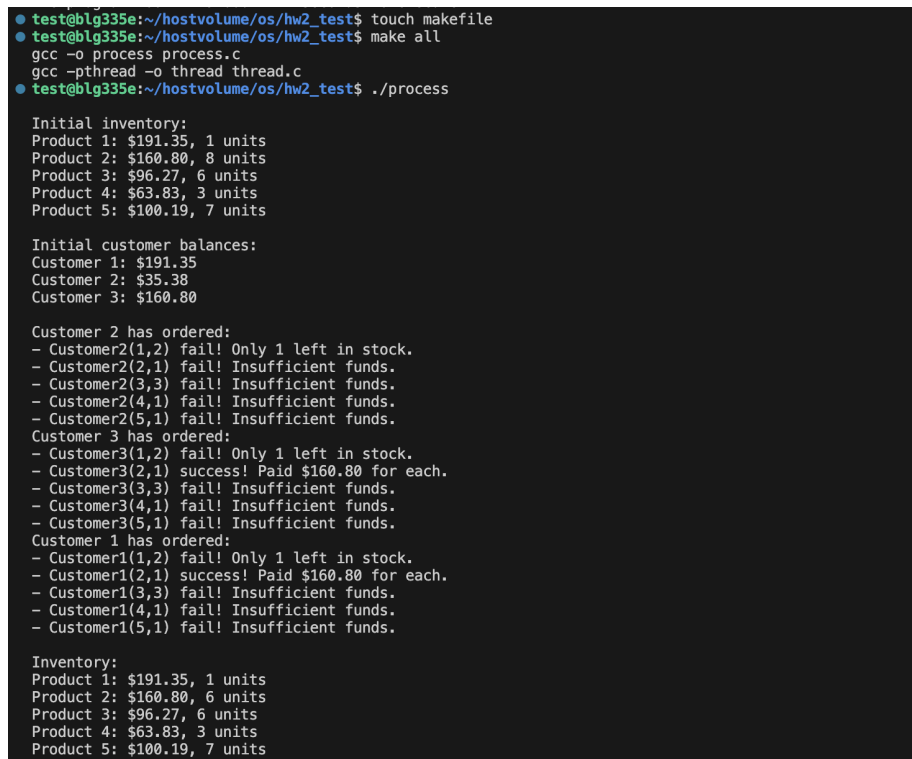


Figure 1: file position in directory



```
os > hw2 > Makefile
1 all:
2     gcc -o process process.c
3     gcc -pthread -o thread thread.c
4
```

Figure 2: Makefile



```
test@blg335e:~/hostvolume/os/hw2_test$ touch makefile
test@blg335e:~/hostvolume/os/hw2_test$ make all
gcc -o process process.c
gcc -pthread -o thread thread.c
test@blg335e:~/hostvolume/os/hw2_test$ ./process

Initial inventory:
Product 1: $191.35, 1 units
Product 2: $160.80, 8 units
Product 3: $96.27, 6 units
Product 4: $63.83, 3 units
Product 5: $100.19, 7 units

Initial customer balances:
Customer 1: $191.35
Customer 2: $35.38
Customer 3: $160.80

Customer 2 has ordered:
- Customer2(1,2) fail! Only 1 left in stock.
- Customer2(2,1) fail! Insufficient funds.
- Customer2(3,3) fail! Insufficient funds.
- Customer2(4,1) fail! Insufficient funds.
- Customer2(5,1) fail! Insufficient funds.
Customer 3 has ordered:
- Customer3(1,2) fail! Only 1 left in stock.
- Customer3(2,1) success! Paid $160.80 for each.
- Customer3(3,3) fail! Insufficient funds.
- Customer3(4,1) fail! Insufficient funds.
- Customer3(5,1) fail! Insufficient funds.
Customer 1 has ordered:
- Customer1(1,2) fail! Only 1 left in stock.
- Customer1(2,1) success! Paid $160.80 for each.
- Customer1(3,3) fail! Insufficient funds.
- Customer1(4,1) fail! Insufficient funds.
- Customer1(5,1) fail! Insufficient funds.

Inventory:
Product 1: $191.35, 1 units
Product 2: $160.80, 6 units
Product 3: $96.27, 6 units
Product 4: $63.83, 3 units
Product 5: $100.19, 7 units
```

Figure 3: Run "make all", Output

```
Sales:
Customer 1:
- Product 2: 1 units, $160.80 each
Customer 2:
No purchases made.
Customer 3:
- Product 2: 1 units, $160.80 each

Latest customer balances:
Customer 1: $30.55
Customer 2: $35.38
Customer 3: $0.00

The program took 4.486000 milliseconds to execute
● test@blg335e:~/hostvolume/os/hw2_test$ ./thread

Initial inventory:
Product 1: $193.40, 8 units
Product 2: $184.44, 0 units
Product 3: $103.61, 2 units
Product 4: $169.72, 6 units
Product 5: $27.65, 5 units

Initial customer balances:
Customer 1: $193.40
Customer 2: $13.39
Customer 3: $184.44

Customer 3 has ordered:
- Customer3(1,2) fail! Insufficient funds.
- Customer3(2,1) fail! Only 0 left in stock.
- Customer3(3,3) fail! Only 2 left in stock.
- Customer3(4,1) success! Paid $169.72 for each.
- Customer3(5,1) fail! Insufficient funds.

Customer 1 has ordered:
- Customer1(1,2) fail! Insufficient funds.
- Customer1(2,1) fail! Only 0 left in stock.
- Customer1(3,3) fail! Only 2 left in stock.
- Customer1(4,1) success! Paid $169.72 for each.
- Customer1(5,1) fail! Insufficient funds.
```

Figure 4: Output continues, time elapsed in multiprocess run is 4.48 miliseconds.

```
Customer 2 has ordered:
- Customer2(1,2) fail! Insufficient funds.
- Customer2(2,1) fail! Only 0 left in stock.
- Customer2(3,3) fail! Only 2 left in stock.
- Customer2(4,1) fail! Insufficient funds.
- Customer2(5,1) fail! Insufficient funds.

Inventory:
Product 1: $193.40, 8 units
Product 2: $184.44, 0 units
Product 3: $103.61, 2 units
Product 4: $169.72, 4 units
Product 5: $27.65, 5 units

Sales:
Customer 1:
- Product 4: 1 units, $169.72 each
Customer 2:
No purchases made.
Customer 3:
- Product 4: 1 units, $169.72 each

Latest customer balances:
Customer 1: $23.67
Customer 2: $13.39
Customer 3: $14.72

The program took 2.811000 milliseconds to execute
● test@blg335e:~/hostvolume/os/hw2_test$
```

Figure 5: Output continues, time elapsed in multiprocess run is 2.81 miliseconds.

5 Performance Comparison Between Multiprocessing and Multithreading

The speed of multiprocess versus multithreaded programs depends largely on the specifics of the task being performed and the hardware on which it's running.

However, generally speaking, multithreading can be faster than multiprocessing for a few reasons:

- **Context switching:** Switching from one thread to another within the same process is typically quicker than switching between processes. The operating system needs to change less information in the CPU registers during a thread context switch than during a process context switch.
- **Resource sharing:** Threads within the same process share the same address space, which means they can access each other's data directly, without needing to use inter-process communication (IPC) mechanisms, which can be slower.
- **Startup time:** It's generally quicker to start a new thread than a new process.

In this assignment, the multithreaded program performed as anticipated and executed faster. As evidenced by the output images depicted in Figures 3, 4, and 5, the execution time for `process.c` was approximately 4.48 milliseconds, while `thread.c` completed its execution in approximately 2.81 milliseconds.

6 Efficiency of Multiprocessing vs Multithreading in High Load Scenario

In a scenario where there are 10,000 active customers in the shopping system, it is likely that multithreading would be a more efficient methodology to use. The reasons for this are as follows:

1. **Memory Usage:** In a multiprocess system, each process has its own memory space. This implies that with 10,000 processes, the memory space would be duplicated 10,000 times. On the other hand, threads within the same process share the same memory space. This difference can lead to significant memory savings in a multithreaded approach.
2. **Context Switching:** Context switching is generally faster between threads than between processes, because threads share the same address space. With 10,000 active customers, the overhead of context switching would be less with threads.
3. **Inter-thread Communication:** Threads within the same process can communicate with each other more easily and efficiently than processes can. If the 10,000 customers need to share data, such as a shared inventory of products, using threads can simplify and speed up this process.

However, it is important to note that threads may bring their own challenges, such as increased complexity of programming due to the need to manage shared state and potential issues with race conditions.

Ultimately, the choice between multiprocessing and multithreading depends on several other factors. These include the specific requirements of the application, the nature of the tasks being performed, the architecture of the system on which the application is running, and the proficiency of the development team with multithreaded programming.