BLG 312E Computer Operating Systems Homework 2

Due: Wednesday, May 10, 2023

Lecturer: Prof. Kemal Bıçakçı

CRN: 23148

Fatih Baskın
150210710

Contents

Implementation

Introduction

In this homework, we were asked to implement a shopping program using C language in which there are customers and products. Customers must be able to access and make purchases at the same time, concurrently. We were asked to implement both multi-processed and multi-threaded programs.

Multi-threaded program

Creating the multi-threaded program was easier because each thread uses the same global and heap memory. Therefore there is no need to create shared memories. Pointers to products and customers are defined on the global memory and they are initialized with `malloc()` in the `main()` function via initializer functions. Initializations are made randomly within the limits.

Locks

Each customer is running on a thread (or process) and they concurrently run. Product quantity checking and modifying the quantity operations are critical sections so each product has a `product_Lock` mutex lock. Quantity checking requires a lock because while checking, another thread (or process) might change the value and the current assessment might be false.

There is also a global `printlock` which is used because if a message is required to be printed, another thread shouldn't print. Otherwise for example, while a thread is printing the list of products, another thread can write another message and the output would be messy.

Also, the changes in the product quantities are made inside `printlock` too. Because if a change in a product quantity is required, the initial state and the latest state of the products are printed. While printing another program might change a value of a product and the resulting output would be false. To prevent this, if a change is required to a value, `printlock` is held, initial state is printed, changes are made and finally resulting state is printed. But this logic makes the program act like a sequential one, not a parallel one if the list of products is large. Details are discussed in the discussion.

Multi-processed program

The multi-processed program on the other hand required me to create shared memory to be able to communicate between processes. And since heap memory is not shared between processes, dynamically allocated objects were initialized by using shared memory too. Also, `printlock` is a pointer to the shared memory address where it holds a mutex for the `printlock`. Dynamic memory allocation is made in the parent by using `mmap()` function provided by `#include <sys/mman.h>` library.

All mutex locks required some attribute changes while being initialized. They needed to be able to be shared between processes so they needed some attributes during initialization. Those attributes are defined using a `pthread_mutexattr_t attr` variable and required attributes are assigned to it by using the `pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED)` function with the parameters given. Then mutex is initialized with the attribute using `pthread_mutex_init(printlock, &attr)`, note that `printlock` is now a pointer pointing to shared memory space.

Implementation continued on next page...

Files and `makefile` Commands

There are 6 `.c` files and 2 `.h` files. `thread_head.h`, `thread_head.c`, `threaded_benchmark.c` and `threaded.c` are the source codes of the threaded program. `process_head.h`, `process_head.c`, `processed_benchmark.c` and `processed.c` are the source codes of the processed program.

Benchmarks are slightly modified versions of their base programs, they don't print messages to console, they append (if necessary, create) the runtime in microseconds to the `benchmark.txt` file. Parameters such as `PRODUCT_COUNT` and `CUSTOMER_COUNT` are macros defined in the `.h` files. Macros can be changed there to text program speeds with different counts.

For the `makefile` to work properly, all the source codes and the makefile must be on the same folder. There are four makefile commands:

- `make processed` creates the regular multi-processed program
- `make threaded` creates the regular multi-threaded program
- `make processed_benchmark` creates the benchmark for the multi-processed program
- `make threaded_benchmark` creates the benchmark for the multi-threaded program

Structs

There are three structs (like classes so I will refer them as classes) in this program:

- `Product:` This class is for holding the information of the products. There are two versions of initializers, regular and randomized. They dynamically allocate memory space and return a pointer to that space. In the processed version it allocates a shared memory address and returns it by a pointer. In my implementation, I used the randomized version. This struct contains:
  - `product_ID` is the ID of the product, it is assigned during the initialization. Initializers take a parameter which is a global counter and they assign the value of the counter and increment the counter.
  - `product_Lock` is the mutex lock for the product. It is required for quantity checking and quantity changing operations. Details of these operations are discussed later.
  - `product_Price` is the price of the product. It is a double value between 1 and 200. Those values are actually macros that are held in the header files.
  - `product_Quantity` is the quantity of the product, it is a value between 1 and 10 but again those values are actually macros defined in the header files.
- `Product_Customer:` This class is the simplified version of the `Product` class, It only holds `product_Price` and `product_Quantity`. This class is used to store ordered and purchased items of the customers and is used as a parameter for the ordering function. It doesn't have an initializer.
- `Customer:` This class is used for holding the information of the customers. Its initialization is similar to the `Product` class but its initializer has some more details for randomizing customer orders but overall logic is the same. Its initializer returns a pointer to dynamically allocated or shared memory space. It contains:
  - `customer_ID:` Same as the `product_ID`, a global counter is used to assign customer IDs.

- `customer_Balance`: This is a double value, ranging between 1 and 200. It is randomly assigned and its borders are actually macros defined in the header.
- `initial_Balance`: This is the copy of the `customer_Balance` but it is never modified again during the execution of the program it is printed to compare the initial and final balance of a customer.
- `ordered_Items_Counter`: `ordered_items` is actually an array and this counter is holding the size of it. The decision-making process of which product to buy and how many using `rand()` is complicated. First, a boolean array consisting of same number of elements as products is created with each element being false, then two local variables are created, `last_rand` and `rand_count`, initially they are equal to zero. A random value is chosen between `last_rand` and (`last_rand` + the number of products - 1) (inclusive). Since the array is 0-indexed, -1 is mandatory. The generated number indicates the index in the boolean array. If the generated number is greater or equal to the number of products then the loop that generates this number is stopped. If the generated number is in the bounds of the array, then `rand_count` is incremented and the boolean value indexed by the generated number is set to true. If a boolean value is true, the product that has the ID index + 1 will be ordered. Using this logic, orders are spread randomly and close to normal distribution. `ordered_Items_Counter` value is set to `rand_count` because it indicates the number of items that will be ordered.
- `ordered_Items`: Using the boolean array created above, it is dynamically allocated (or allocated via shared memory if the program is multi-processed), and its size is set according to the number of orders. Ordered items are kept as an array of `product_Customer` (actually a pointer but pointers can be used with indexing) because it occupies less space than the regular `Customer` class. Then its IDs are set using the indexes of the boolean array. Its quantity is set as a random value in the range of 1 to 5. Its borders are defined in the header file as a macro.
- `purchased_Items_Counter`: By default it is defined as zero. But later in the execution, its value can change as the customer purchases products.
- `purchased_Items`: This is again a pointer to `product_Customer`. It is set to `NULL` in the multi-threaded program and the space for it is allocated while ordering. In the multi-processed program, it is very difficult to dynamically allocate and transfer a dynamically allocated memory space to the parent process so in this case, the same size of space as `ordered_Items` is created via shared memory.

`order_Product(s)()` Function

After a thread (or a process) for a customer is created, it checks whether the current customer has one order or not. If the current customer has only one order, it calls `order_Product()` function, otherwise, it calls `order_Products()` function.

`order_Product()` function returns a bool value, a transaction (buying) is done or not done. It takes a pointer to `Product_Customer` as an input which contains ID and quantity for an order. It also takes a pointer to the customer balance and the customer ID.

First, it checks whether the customer has enough money to buy the order. Since products are defined on the heap (or the shared memory) and their pointers are on the global domain, the function can directly access the products. If the customer doesn't have enough money, the function tries to acquire the lock for the `printlock` and prints the message, then releases the lock and returns false.
If the customer has enough money, then the function acquires the `product_Lock` for the specific lock so while doing quantity checks (to check there are enough products in stock) and while doing so, another process or thread shouldn't change its value. Then if there are not enough products, it tries to acquire the `printlock`

[`order_Product(s)()` Function] continued on next page...

and prints the message and releases the `printlock` and the `product_Lock`. Then it returns false.

If the function passes all the checks (note that this is an if statement and `product_Lock` is acquired before this statement), firstly, it acquires the `printlock`, then it prints entire stocks before the transition. Then it changes the values of the quantity of the product and the customer balance and finally, it prints latest state of the product stocks after the transition. `product_lock` and `printlock` must be held during the entirety of this operation because while printing, another process (or thread) might change the value of another product, and since it is not part of this operation, the output would be false. After printing everything, it releases the locks and returns true.

`Order_Products()` function takes similar inputs and in addition, it takes customer's `ordered_Products_Counter` as an input. It creates a boolean array of the said counter using `malloc()` (since it is used locally by the thread/process it doesn't matter) and in a for loop, calls `order_Product()` function and assigns the return values to corresponding indexes in the boolean array. Pointer to `Product_Customer` can point to a single element or an array of elements so it passes the address of the current indexed element into the function.

Customer thread or process uses these return values, boolean value or array. If it is a thread, it creates `purchased_Products` by `malloc()` since it is shared among all threads. If it is a process, it sets the values inside the preallocated (shared) `ordered_Products`.

Performance

Hypothesis and Benchmarking Program

Since the multi-processed approach requires more things to work such as context switching, copying the entire memory space while being forked, and requires defining shared memory sections which also have overheads, I was expecting the multi-processed program to work slower. To test my hypothesis, I created non-printing versions of the multi-processed and multi-threaded programs to test them.

I removed printing because printing takes too long because printing to the console has its own overhead so in the end both programs lasted for too long to end. Also, I/O overheads are unpredictable, therefore results wouldn't be reliable.

I tested both programs without printing (commented the `printf()`s) and added timestamping to calculate elapsed time. For timestamping to work, code must be compiled with C11 standard, by giving `-std=c11` parameter to the compiler.

Benchmark Results

I created benchmarks with 700 customers and 1000 products. Here are the results (in microseconds):

According to the benchmark results, the average time of execution of a multi-threaded program is 56077.4 μs. The average time for the multi-processed program on the other hand is 2210505.1 μs. The ratio between multi-processed and multi-threaded programs is $\frac{2210505.1}{56077.4} \approx 39.39$ and this was expected.

Discussion

From the benchmark results, it is obvious that multi-processed programs work slower. But the benchmark ignores printing and printing itself has an overhead. Overheads of the multiprocessing might be ignored in

Performance [Discussion] continued on next page. . .

Test number	Multi-threaded	Multi-processed
1	47304 μs	2162949 μs
2	47239 μs	2146460 μs
3	47936 μs	2144002 μs
4	52235 μs	2193242 μs
5	59280 μs	2177616 μs
6	61484 μs	2181177 μs
7	61100 μs	2226367 μs
8	62808 μs	2302359 μs
9	53398 μs	2181783 μs
10	55190 μs	2204896 μs

Table 1: Benchmark results, time of the execution of the programs without printing.

the case of a small number of customers since the time for process creation and context switching will be small but printing would still have some overhead and overheads would be similar since the two approaches will yield similar outputs. But in the case of a large number of customers, process overhead will dominate.

From the efficiency standpoint, especially in the cases of many customers, i.e. 10000 customers, multiprocessing would be terrible in terms of performance because it needs to create 10000 processes, each process will occupy some memory (heap, stack) and a large amount of shared memory must be initialized. Therefore the multi-processed program will be wasting time with context-switching, process creation, and shared memory initialization. It will also waste a very large amount of memory since each process has its own memory space.

On the other hand, the multi-threaded program will work a lot faster since thread creation is much more simple than process creation, they share the same memory space and except for each thread having its own stack, memory usage will be very small compared to the multi-processed program.

One other problem in the processed program is, it required to allocate preallocated shared memory for `purchased_Products` and it is quite wasteful since not all orders going to take place and the majority of this space will be left unused but allocated via shared memory.