Name : Gregory Grzymski

Assignment 5: Multithreaded Book Order System

Setup: 1 producer thread and multiple consumer threads (no extra credit)

The program begins by checking if the user entered the correct inputs in order to properly run the program. If the inputs are valid, then a char **array is created based on the categories text file. Each category represents an index in the array, and the function is designed to take in any number of categories.

Once the categories are stored in the array data structure, the database file is parsed with all of the users in the file inserted into a linked list data structure. Each person represents a Customer node that contains variables for all of the user fields from its line in the text file. In addition to the user's information, the Customer node also contains two linked list pointers to Receipts lists. These lists represent the successful and unsuccessful orders that are processed by the program.

A single queue is created and acts as a shared buffer that is utilized by the producer and the multiple consumers. When initialized, the queue has a pthread mutex variable to regulate access between the multiple active threads, and two condition variables:

1. The check_empty variable regulates the communication between the consumer and producer when the queue is empty. This prompts the consumer to wait for a signal from the producer when the queue is empty during its run.

2. The check_full variable regulates the communication between the producer and consumer when the shared buffer queue has reached the max size of 10. This prompts the producer to wait until one of the consumer threads has dequeued an order from the buffer.

The amount of pthreads spawned during the program run is dependant on the amount of categories stored in the text file. In the sample file, four threads were spawned: 1 for the producer, and 3 for the categories. The producer thread's responsibility is to take the orders text file, parse the information and insert each order into an Orders variable, and then enqueue the order into the shared buffer queue. The Order struct contains the information interpreted from the line in a linked list data node.

As mentioned above, if the the buffer size of the queue has reached its capacity of 10, then the producer won't enqueue the order object and will call on the cond_wait condition variable. Once the producer receives a signal from the consumer that the buffer size is below 10, then the order will get enqueued. The enqueue operation is wrapped within a mutex lock/unlock access because the consumers are also simultaneously communicating with the queue. The producer also sends a conditional signal to the consumer(s) when an enqueue is successful to alert them that the the queue is possibly no longer empty. Once the producer completes the orders, the function sends a broadcast signal to all potentially blocked consumer threads to wake up and process the remaining orders.

Each consumer thread runs its function according to the individual category it was assigned. The function operates in a looping fashion while the queue remains filled or the producer isn't finished. If the thread encounters an empty queue, it calls a cond_wait variable and waits for the queue to be filled with some data by the producer. If the queue is empty and the producer is indicated (boolean flag) as finished, then the thread exits the function. During the simultaneous run of these threads, if the queue becomes empty after the the firt condition statement is called, then the current thread calls a sched_yields and allows another thread in the system queue to run.

Once the consumer has an order to process, the category of the order is used to compare to the current thread. If the categories do not match up, then the current thread calls a sched_yield and wakes up another consumer in the queue that will contain the matching category. If the categories do match up, then an order is dequeued from the buffer and the customer is pulled from the database according to the identification number. Once the customer has been found, a receipt value is created with the title, book price, and the customer's remaining credit limit when it is subtracted from the price of the order.

The customer's credit limit is compared to the price of the book, and two situations arise:

1. If the credit limit is lower than the price, then an order rejection is printed with the customer's name, book details (title/price), and the customer's remaining credit limit. The receipt is added to the customer's failed order linked list.

2. If the credit limit is greater than or equal to the price, then an order acceptance is printed with the customer's name, book name, book price, and shipping information. The receipt is added to the customer's successful order linked list.

The producer and consumer threads have pthread_join called on them in order to have the threads terminate properly amongst each other. Mutex locks and unlocks are placed on the critical sections of the consumer code in order to ensure that no variables are being used by more than one consumer at the same time.

The final report of the program is written into a file called finalreport.txt, with the successful/failed orders written for each customer. At the end of the text file, a total revenue is reported by calculating the total successful orders from all the customers.

All allocated memory throughout the program is freed properly without any memory leaks or uninitialized values. This was tested with valgrind test results.

Also, please note the statements:

• Producer waits because queue is full.
• Producer resumes when queue buffer space is available.
• Consumer waits because queue buffer is empty.
• Consumer resumes when queue has orders ready for processing.

are printed out on the command line when the consumer or producer thread are executing that behavior.

<u>Running-Time</u>

The queue operations of enqueue and dequeue are performed in O(1) time.

Finding the customer in the consumer thread may take worst case O(n) time, with the customer being the last item in the linked list.

Inserting a receipt into the successful/unsuccessful linked list also takes O(n) time because it is always appended at the very end of the linked list it belongs to.

If there are n Customers, it takes O(n) time to write the report for each customer. However, if there are s successful orders and f failed orders, then it takes O(f)+O(s) time for each customer. In total, the running time for writing the report is : O(n *(s+f)).