Project 5 - Multithreaded Book Order

Kyle Suarez and Indraneel Purohit
kds124@rutgers.edu, ip132@scarletmail.rutgers.edu

November 25, 2013

0.1 Overview

0.1.1 Design

For the customer database, we spend time at the beginning parsing the database text file. The database is an array of customer structures that contains relevant customer information. Since each customer has a unique customer ID, we use that ID number as the index of the customer in the array. (Note: we assume that customer IDs can be directly be used as array indices. For IDs containing extremely large or negative numbers, we would need some sort of hash function and turn the array into a hash table to deal with collisions.)

The producer and consumer share a single queue. This way, we guarantee that the order in which the producer generates the book orders is the same order in which the consumers will process them (as opposed to having multiple queues, one for each category). For efficiency's sake, the queue is implemented as a circular linked list. There is a mutex that protects the queue from concurent modification by the producers and consumers. There is also a condition variable `queue.nonempty` that represents the condition that the queue actually contains something inside.

The producer thread's code can be found in `producer_thread()`. The expected argument is a `FILE` pointer to the order text file. For each line in the file, we parse it, getting relevant information and performing some sanity checks. We then acquire the queue's mutex, enqueue the order, and then use `pthread_cond_signal` to alert the consumer threads that the queue now has something ready for processing. Once the producer is finally finished parsing the entire file, it sets the global flag `is_done` to a truthy value and then uses `pthread_cond_broadcast` to wake up all threads, forcing them to empty out the queue and then finally exit.

The consumer threads' code can be found in `consumer_thread()`. The expected argument is a null-terminated string representing the name of the category that this particular consumer thread is dealing with. The consumers then begin working in the main while loop, which keeps looping until `is_done` is true and the queue is empty. Each thread contends for the queue's mutex. If the queue is empty and the producer says it's finished, we stop. Otherwise, if the producer is still working, we relinquish the mutex via `pthread_cond_wait()` and wait for the producer to signal that there's actually something ready for processing. Once we're finally awake and have the mutex again, we peek at the top of the queue. If the category of the order matches the consumer's category, we go ahead by dequeuing it and processing it by modifying the database. We also leave a receipt for the order, failed or not, in the database as well. Otherwise, it's meant for someone else, so the consumer will unlock the mutex and `sched_yield()` to allow another thread to use the queue.

It would appear that the database would also need a mutex so that threads who are processing orders could access it in a thread-safe way. However, our code is written in a way so that the queue mutex also extends protection for the database. When a consumer claims the lock on the queue mutex, *it does*

not release it until it is finished modifying the database. Thus, one and only one producer is running in its critical section at any one time; therefore the database retains its internal integrity. This also solves the problem of two different orders being processed out-of-order; this code will always produce the same successful orders, failed orders, and final revenue after every run.

0.2 Analysis

0.2.1 Runtime Analysis

Since the shared queue is the focal point of the producers and consumers, we worked to make queue accesses as efficient as possible. Because the queue is a circular linked list, we achieve $O(1)$ enqueue, dequeue, and peek operations because we have immediate, constant-time access to both the front and back of the queue. Similarly, database operations are also extremely fast. Because each customer has a customer ID used as its index in the database, looking up a customer or finding its proper spot in the database takes $O(1)$ time.

At the end, we traverse the database and print out relevant information for each customer, including successful and unsuccessful orders. Because accessing each customer takes $O(1)$ time and fetching the receipts from the customer receipt queues are also $O(1)$, it ends up being that printing all of the information takes $O(n)$ time for n orders submitted to the program.

Finally, we address our threaded code. Because we use condition variables and yield scheduling at appropriate times, there is no busy waiting or spin-locking in our code. Threads that are scheduled that have no work to do immediately yield; if they realize that they are in a finished state, they exit fully.

0.2.2 Memory Usage

The parsing for this project is pretty memory-efficient. Because we use `getline` to read one line from a file stream at a time, only some parts of the file are in memory at any given time. Furthermore, there is no duplication of data in that for every customer and every order, there is one customer structure and one order structure to represent it. For n orders placed and a constant k number of customers, we allocate space for $n * k = O(n)$ data structures over the lifetime of the program.