Chris Fretz, Kenny Udovic
November 23th, 2014

Program Summary

Book.c contains the main function, which initializes the data structures and controls the flow of the program. It starts by opening all three input files.

construct_database

The database file parsed and pulled apart by it's | delimiter. It reads the information into the stack, and O(n) to insert each user into the array list responsible for storing the database of users. Each user is also accounted for in the program by creating customer structs. The customer struct contains the information of the respective user, creates two empty doubly linked lists that will contain the information of the approved and rejected orders, as well as assigns a mutex to the customer.

generate_customers

Next, the program creates consumer structs – which contains a pointer to array list of users and the file containing the categories, and a queue (doubly linked-list) which will hold the orders of the struct's respective category until proccessed. Consumers are created and inserted into a hash table by generate_consumers, hashed by the added ascii value of the category modded by the table size. The hash table requires O(1) to insert to, and O(n) to build. The consumer struct assigns a thread to each category of item, creates a doubly linked list as a queue to keep track of the orders its thread needs to processes, and handles passing of pointers of the users struct and the its respective queue to the thread contained within itself. A custom struct named void_args is created in order to pass both these values as a single parameter to the thread.

produce

Orders are then processed. Again, the program requires O(n) to read in the input and create a struct out of each order. The order struct contains all the information provided by the file about each order. Having hashed the consumers by category, produce() fetches the respective consumer using the category provided by the order. After all orders are produced and handed off to the threads, stop orders are enqueued for every thread.

consume

Threads operate by using the consume() function. The function is responsible for popping values off the order queue in the consumer, (O(1) because it is a doubly-linked list) and placing them either in the approved or rejected list contained within each customer based on the balance the customer has at the time the process occurs.

The program then gets all the keys placed into the consumer table and uses that information to join the threads together.

handle_output

Then the program handles creating the output file iterating through the users array list and translating the orders in the approved and rejected lists in each customer struct. Output is written to output.txt.

deallocations

Finally the program destroys all data structures running a number of destruction methods and closes all three files.

Thread Synchronization

The program utilizes doubly-linked lists in multiple parts of the program. When a doubly-linked list is created, a boolean is passed to the constructor determining whether it will be interacting with a thread. If it is determined so, the list mallocs space for a mutex and a signal.

As mentioned earlier, threads are contained by the consumer struct which utilizes a doubly linked list as its order queue, and operates by using the consume() function. The doubly-linked list never has to reallocate as it can't reach a capacity, making thread sychronization a bit easier. Unfortunately, a maximum capacity has been set for the linked list as the project required.

As produce pushes orders onto a consumer struct's queue, a mutex locks and the data structure is only then manipulated. When manipulation is complete, a signal is sent to the consumer thread that data has been added and the mutex is unlocked. Consume() can then operate.

When produce locks a mutex, it checks whether the linked list has reached capacity. If so pthread_cond_wait() is called, thus unlocking the mutex, and the thread waits on a signal from the rpop() method called by consume(). Once the signal is received, the program resumes producing orders.

Consume() runs rpop(). Before popping off the tail of the doubly-linked list (O(1)), a mutex locks. Next, it determines whther the list contains any orders. If not, immediately unlock the mutex. If so, manipulate the data structure and remove the tail of the list, unlock the mutex, and free the memory associated with the removed tail.

Consume() also runs lpush(), adding orders to a customer's approved or rejected list. Since multiple consumer structs can be operating on a user's orders, ie. a user places an order in multiple different categories of books, consume will lock the mutex contained by the users struct before adding to its approved or rejected list.

With the appropriate use of mutexes and signals, thread synchronization works consistently without deadlocks.