Jonathan Chen & Adam Tecle

DESIGN

This program works by first creating a database of customer information in “create_db”. Structs of type customer are hashed by integer customer ids using uthash. Customer structs contain that customers name, their current funds, zipcode, and address. The categories are read **from the command line** and put into another hashtable, each category stored into the table as a type struct Cat, containing the index of the array of queues that corresponds to the category. **You cannot have more than 100 categories at once, as the array of queues is capped at 100 indices.**

The order txt file is read in “read_order”, each order being contained in a struct Order object. The category for that order is queried in the Category hashtable, which gives us the index of the queue array. Once we know the index of the queue array that corresponds to that category, we enqueue that order as a struct Queue item. Our queue is implemented as a linked list using utlist. The function of read order is to set up the array of queues. The category hash table assists in this by making lookup an O(1) operation.
We process orders in main.c by iterating through the category array, getting the index in the queue array that corresponds to one category, and passing that to pthread_create. To do this, we have an array of pthreads **capped at 100, matching the limit of categories**. Each thread deals with the queue of orders for one category, processing orders one by one.

While processing orders, we create a database of struct Report items, which keys by customer id, keeping track of that customer’s successful orders, failed orders, and the funds associated with the purchase of each book. After we process every order, the report hashtable is complete and is used to create the final report. The final report is stored in a text file named finalreport.txt.

THREAD SYNCHRONIZATION

The synchronization of our threads is facilitated by the use of pthread_mutex_t. When updating a customer’s funds, we first lock that customer’s information, then proceed to update. Threads trying to access that customer’s information simultaneously wait until pthread_mutex is unlocked, and only then update. After creating the threads corresponding to each category, we call pthread_join on each thread, in the order they were created, to assure deadlocks would not occur -- pthread_join returns an error code to detect deadlocks