COMP2006 - Operating Systems

CURTIN UNIVERSITY

School of Electrical Engineering, Computing and Mathematical Sciences

Customer Queue

Name: Dhilisha Flavio Ranatunga
Student ID: 20908391

Introduction

- The assignment is to develop a bank simulator which consists of 4 tellers and a queue of customers. The requirements of the simulator are for the program to be able to handle several customers who are added and removed from the queue repeatedly. A teller should provide service to a customer while also been in sync with the other tellers and checking the queue status repeatedly. This is done by creating different threads for each teller and customer function, the threads are synchronized using pthread mutual exclusion functions.
- This report's objective is to discuss the program solution offered as part of the specified Customer Queue assignment. The source code consists of files cq.c, queue.c, teller.c and mutexes.c. In-depth explanations of the use of pthread mutual exclusion functions in multithreaded programming are discussed in this study, including how they can be used to synchronize shared variables and resources between threads. To guarantee that only one thread at a time can access shared resources or variables, the report also provides in-depth information on thread safety, access control, and mutual exclusion techniques. It also covers how threads acquire and release mutexes and condition variables, as well as the roles and permissions of various threads when gaining access to shared resources or variables.

Procedure

- A thorough examination of the provided requirements was done to address the problem at hand, with a focus on discovering shared resources and variables. The job involved creating a model where customers line up in a queue at periodic intervals and are assisted by one of four tellers. The

following techniques, data structures, and programming ideas were applied to this goal:

- Multithreading: To simulate the system's continuous execution, multithreading was used. There were two different kinds of threads made: teller threads and a customer thread. Customer objects were created by customer threads, which also queued them up. Customer objects are then taken out of the queue and served by teller threads.
- Mutexes: When accessing shared resources like the customer queue, teller service count, and the mutexes themselves, mutexes were used to ensure mutual exclusion and prevent race conditions. The mutexes were built using the pthread_mutex_t structure and were acquired and released using the pthread_mutex_lock and pthread_mutex_unlock methods, respectively. Mutexes were used as global variables so other functions can also utilize the corresponding mutex. These were implemented in mutexes.h and mutexes.c
- Condition variables: To keep the threads synchronized, condition variables were implemented. The condition variables were built using the pthread_cond_t structure, and the wait and signal conditions were handled by the pthread_cond_wait and pthread_cond_signal, functions, respectively. To signal whether the queue was neither empty nor full, two condition variables, notEmpty and notFull, were used.
- queue: To hold customer items waiting to be served, a queue was used. A front and rear index were used to implement the queue by utilizing a LinkedList. This was done by creating a struct called node_q which contained data of customer and a pointer to the next customer of the list/queue. The elements were added to the queue using the enqueue and dequeue methods, respectively.
- Global variables: Command line arguments were set variables which were externed using the cq.h file to set the sleep times of several variable such as tc(period time to add customer to queue), tw(time for cash withdrawal), td (time to deposit cash), ti(time to get information).
- Various challenges had to be solved during the process; most of them had to do with testing and debugging the solution. These were fixed by adding delay functions to slow down the simulation and examine its behavior, using printf statements to produce debug messages, and using the valgrind tool to check for memory leaks and make sure the solution was stable.

Implementation

- As mentioned before the implementation has four files used to solve the problem

1. Queue.c - Contains all the structures for queue, node and customer. Main purpose of the file is to run the customer () function without obstructing the program's flow.
2. Teller.c- Contains structures to store teller data and customer data to be used by threads. The teller function is implemented in teller.c is run by 4 teller threads.
3. Mutexes.c- Used to initialize the pthread mutex and condition functions.
4. Qc.c- Executable file which reads arguments and create threads accordingly for the simulator to work

- A LinkedList of customer data structures is used to implement the queue data structure in queue.c. The queue and its elements can be changed using the createQueue, enqueue, and dequeue functions. Mutual exclusion is necessary to guarantee that only one thread at a time can access the queue because it is shared by the client and teller threads. It also consists of the customer thread function which adds customer data to the customer queue periodically from the read file cq_file. Mutexes are utilized to establish mutual exclusion because the customer thread function also has access to the shared customer queue.
- The teller thread function implements a loop that repeatedly checks for customers in the queue and the flag generated by the customer function to indicate that all customers are enqueued from the cq_file. This is done in the teller.c file. It removes a customer from the line, goes to sleep for the duration of the service, and updates the statistics on the teller. The teller thread function uses mutexes to ensure that only one teller serves a customer at a time while accessing shared variables like the customer queue and teller statistics.
- The initialize Mutexes method in mutexes.c is used to initialize mutexes and conditional variables. Only one thread at a time can access the shared resources with the mutex and condition variables.
- Cq.c consist of the main method and pthread_create functions and parses the relevant customer and teller data to the threads, also responsible for cleaning up resources which were created, this is don't after all the threads are terminated.
- The secure and proper access of the threads to the shared resources was one of the primary challenges faced by implementation. Mutexes and condition variables were used to ensure synchronization and mutual

exclusion, and this was accomplished. The proper design of the queue data structure needed careful study of the enqueue and dequeue operations to make sure that they are handled appropriately, which created another issue. Logging to the file was also required to be locked and unlocked using mutual exclusion functions to avoid two or more threads from accessing the r_log file at the same time. This was done using another pthread mutex function called logMutex.

Synchronization

- Using mutexes and condition variables together, synchronization is achieved in this software solution. Mutexes are utilized to prevent concurrent access by several threads to shared resources like the customer queue and log file. To indicate changes in the status of shared resources, such as when a customer joins the queue or a teller completes serving a customer, condition variables are employed.
- In this software solution, numerous threads access the following common resources which are accessed by the customer and teller threads:

1. Customer Queue.
2. R_log file.

- The mutex is utilized to protect the customer queue from concurrent access. The customer thread notifies when the queue is not empty using the notEmpty condition variable, and the teller thread informs other threads when the queue is not full using the notFull condition variable. A thread must first obtain the mutex to enter the customer queue. The teller thread waits on the notEmpty condition variable if the queue is empty. The customer thread waits on the notFull condition variable if the queue is full (i.e., the customer count has reached the queue size and the customer thread cannot add more customers to the queue at the moment). The customer thread notifies the notEmpty condition variable to wake up any waiting processes when a client arrives and is added to the queue. The customer threads add a client to the queue and signals the notEmpty conditions so other processes wake up and continue its operation.
- The mutex lock and unlock functions are applied before condition variables to avoid any unnecessary block or wait signals from other threads and ensure only one thread access the customer queue at a time. The mutex is immediately unlocked and signals the notEmpty or notFull after a customer is added or removed from the queue. This method ensures other threads

will not be idle for long and for the customer to be responded to by the teller much faster.

- The logMutex mutex is applied when threads tend to access the r_log file to log its actions. A similar mechanism is used for handling the customer queue however condition variables are not needed in file writing. The log file is written in several instances such as when a customer arrives at the queue, the customer reaches the teller, and after the completion of the service. Mutex is acquired before opening the file and unlocked after closing the file. This ensures that only one thread can be written to the r_log at a time.
- Challenges that were faced during the implementation of synchronizing included deadlocks and race conditions. Deadlocks occur when multiple threads wait indefinitely for a shared resource that is held by another thread. Deadlocks mainly occur when the last thread is held by the notEmpty wait condition, this is because there are no more available threads to signal the condition. A flag was used to indicate that all customers have been enqueued in the customer function, this will assist the teller thread to terminate. The thread which exits the loop signals the notEmpty condition therefore tellers which are been blocked could terminate as well.

Results

- Different arguments were used to test the program, for example using different number of customers, changing the queue size, using a high queueing time than service time, using a queue size less than 4 and finally checking if the number of total customers served is same as the number of customers in the cq_file. All these tests created the correct log files. The samples folder consists of sample log outputs with different arguments.

Conclusion

- In conclusion, using multithreading and synchronization approaches, the software solution described in this report properly replicates a bank with numerous tellers and customers. The code structure is made up of many modules that carry out particular tasks including building and managing the queue, the tellers, and the customers.
- Several mutexes and conditional variables that manage access to shared resources including the queue, teller, and customer status are used to

synchronize processes. The customer and teller threads are the ones that have access to shared resources.

- Different parameters were used to test the code, and the results demonstrated that the bank simulation functioned as intended. Overall, this solution's ability to simulate a bank is reasonable, and it might be made much better with the addition of new features and functionalities.