CS307 – PA2 Report

Queue Implementation

I picked an existent concurrent queue algorithm which is called Michael & Scott queue. The C code was implemented in the book, and I converted it to the C++ code.

The algorithm ensures thread safety because it wraps critical sections using head lock and tail lock. When some thread tries to dequeue from the head, it gets the lock, so no other threads can own the head lock which results in a thread safe situation. When it comes to dequeue, some thread dequeue from the tail which gets the tail lock. The threads that are coming after that thread must wait for that thread to finish their work which is possible by tail lock. The implementation could be achieved by only one lock, but it would slow down the process. For example, a thread tries to dequeue, and the other thread wants to enqueue. When one thread gets the lock, the other thread must wait for the other thread to finish its job. However, by using two locks that are called head lock and tail lock, our job will be done much faster because enqueue and dequeue operations do not need to wait for each other.

FIFO (First In First Out) is preserved since the algorithm follows the standard queue implementation. It enqueues to the back and dequeues from the head which is the standard behavior of the queues. In addition, the algorithm uses one lock for head and one lock for tail which disables concurrent access, therefore, more than one thread cannot enqueue or dequeue simultaneously which data loss or duplication.

Mutex Implementation

In the MLFQ mutex implementation, the algorithm will have different queues (according to user input) and single quantum value. The algorithm utilizes test and set method by changing atomic flag variable *guard* which make sure that only one thread goes to the critical section.

In the beginning, the algorithm initializes multiple priority level queues according to the user input. As you can remember, the queue algorithm was written in template. In *MLFQMutex* implementation, the algorithm uses *pthread_t* as a template to keep track of the thread ids. Moreover, *guard* is set to 0 which imply that no thread is in critical section.

In lock function, it first looks if the guard equals to 1. If it equals to 1, it means that the other thread is performing its critical section, and no other thread can enter which lead threads to wait. After another thread finishes its job, one of the waiting threads can enter. This mechanism was implemented by the help of the *atomic* library. In the algorithm, flag variable is also kept. But it is only checked in a situation where the queue is empty, or the threads are initialized not so long ago. If flag is 0, it changes flag to 1 and enter its critical section. The algorithm starts the timer to keep track of the time period of the thread. If the flag is taken, the thread will be enqueued to the tail by using its thread id. In this part, the algorithm keeps an unordered map which is a hash map to keep track of the threads' priority levels because when the threads come to the lock function again, the algorithm looks if they exceed their quantum value time and decide which priority level this thread belongs to. Using this method, the algorithm makes sure that it is fair. After enqueue, the threads will be parked, and wait for its turn to be unparked. When it is unparked, it means the thread can go in critical section and for this reason we start the timer.

In unlock function, same as lock function, it checks for the guard first. When a thread tries to unlock. It will look for all the queues to dequeue and unpark a thread. It starts with the highest priority queue and finishes with the lowest priority queue. When it finds a queue that is not empty, it dequeue from this queue. After dequeue, it calculates the duration of the thread. If the duration is more than quantum value, its priority level will be changed. The algorithm increases the priority level by the mathematic floor operation of duration plus its previous priority level. If this calculation surpasses the lowest priority level, the algorithm makes this thread's priority level as the lowest priority by using the hash map. When it tries to lock again, its priority level will be decided from this hash map to be placed in the right queue. On the other hand, all the queues can be empty. In this case, flag is set to 0 again.

In print function, it starts from the highest priority queue and go to the lowest priority queue while printing every level's content (thread ids).

In conclusion, it ensures fairness since it keeps priority queues and hash map to correctly assign the threads' priorities. When it comes to performance, there is a spin wait in the test and set method, but the algorithm tries to lower that as possible by utilizing *yield* to the CPU. Lastly, the algorithm ensures that it is working correctly. In the algorithm, when a thread calls for lock function, only one thread can get the lock and do their job. Other waiting threads will be enqueued. Also, when a thread releases the lock, only one thread can have the lock. These properties ensures that the algorithm works correctly.