CS307 Programming Assignment 2

Akif Işıtan 29354

This programming assignment involves a mutex implementation using the Multi-Level Feedback Queue (MLFQ) strategy. This report is made up of two parts. The first part explains the concurrent queue implementation, and the second one details the mutex implementation.

The queue that was used in the MLFQMutex implementation is the Michael and Scott concurrent queue. The Michael and Scott concurrent queue implementation ensures thread safety and FIFO order through the use of 2 separate mutex locks, one for the head of the queue, and one for the tail of the queue. The enqueue operation is done using the tail lock, which only locks the operations which involve the tail, leading to finer grained locking and improved concurrency compared to a single coarse lock. The dequeue operation is done with the help of the head lock, which ensures elements are removed in the order they were added by a single thread at a time, ensuring FIFO ordering and preventing data races. The neat trick about this queue implementation is that potential problems that might stem from the concurrent dequeue operation are nullified with the use of a dummy node which the head always points to. It ensures uniform handling of enqueue and dequeue operations, and avoids null references which can complicate synchronization. The dummy node can be seen in figures 1 and 2.

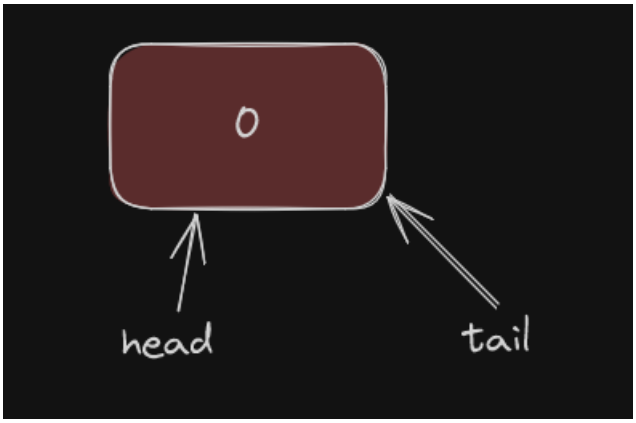

Figure 1. Empty Michael and Scott queue

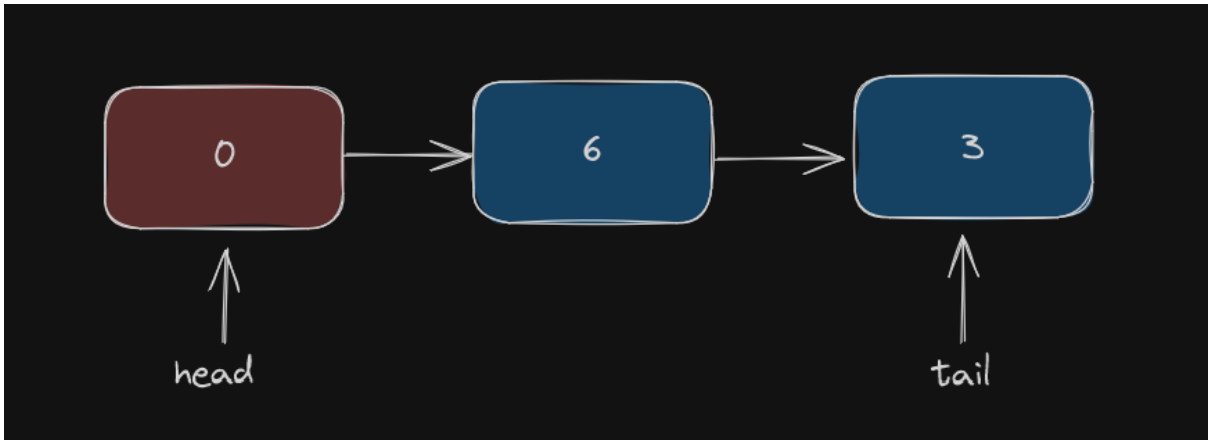

Figure 2. Michael and Scott Queue with 2 elements (6 and 3)

The MLFQMutex implementation is based off of the Solaris mutex implementation, with the MLFQ logic added in. Before going into the implementation, it would be wise to mention the member variables and the constructor method. The members variables consist of an integer flag, representing the state of the mutex with 0 being free and 1 taken; an atomic flag guard, which allows locking the lock() and unlock() method bodies atomically, a double value qVal which represents the quantum value, a vector of queues of thread id to keep the multiple levels of queues (see figure 3), a garage object to put threads to sleep and wake them up, an unordered map to store priorities of threads, and finally timestamp values for start and end to measure critical section length. The constructor takes in the number of priority levels in the queue and the quantum (time slice) value and initializes the members.

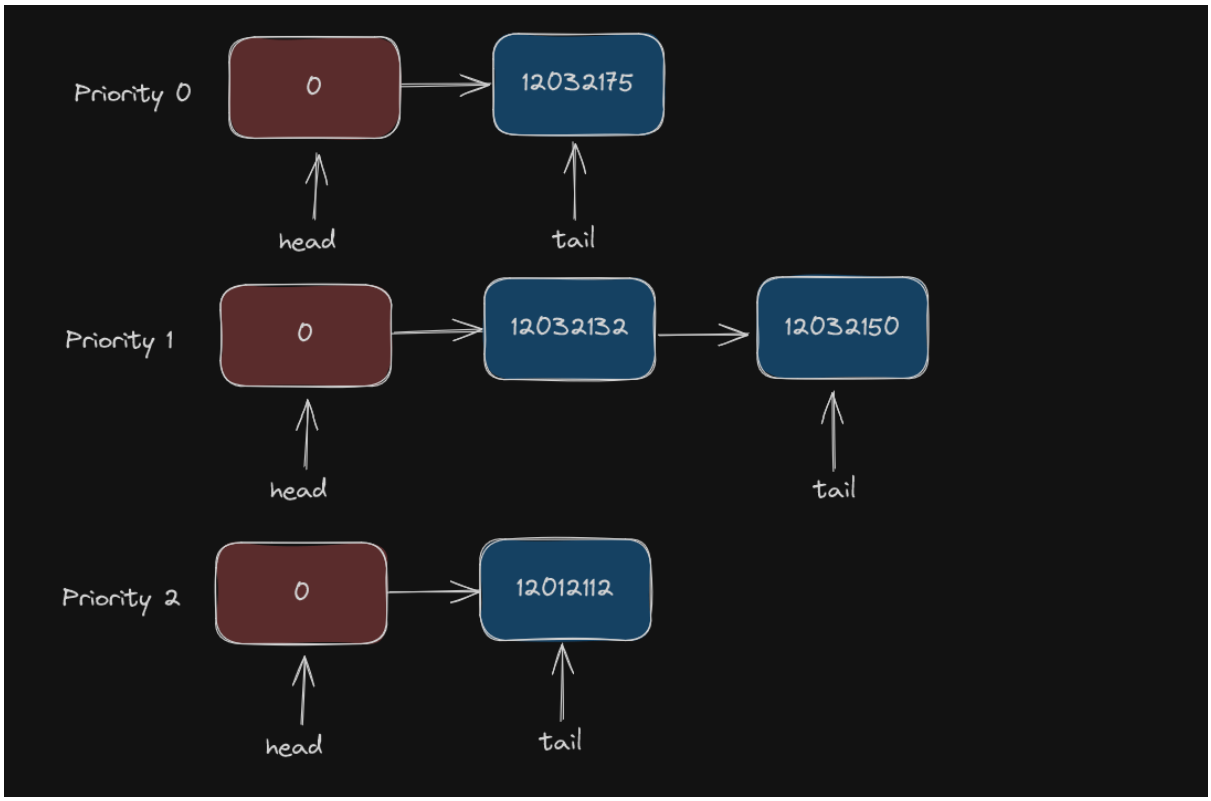

Figure 3. MLFQ with 3 priority levels and 4 parked threads

The lock() method allows the caller thread to take the mutex. It begins by spin waiting while trying to atomically set the guard to true. Once a thread sets it to true, it is guaranteed to be the single thread inside the remainder of the lock() body. The thread first checks for the flag, if its 0 it takes the mutex by setting the flag to 1, starts the timer to measure its critical section and sets guard to false, and returns from the lock function. If the flag is 1, it means that the mutex is already taken, so in order to avoid the busy waiting problem, it will put itself to sleep. However before doing that, it checks the map to find out if it has a previous priority level, otherwise its priority is set to the default of 0, which is the highest priority. After finding its priority, it gets added to the corresponding queue by using the priority as the index. It then calls setPark() using the garage object to signal that its going to park, releases the guard lock and calls park() to put the thread to sleep. It is important to note that the thread will continue executing from this point after being woken up by another thread during the unlock() method, so it can start the timer for itself right after the park() statement.

The unlock() method allows the caller thread to either release the mutex if there are no sleeping threads available or pass the lock to the next thread in the queue by waking it up. It begins by spin waiting while trying to atomically set the guard to true. Once a thread sets it to true, it is guaranteed to be the single thread executing the remainder of the unlock() body. The thread first stops the timer and calculates its critical section execution time. It then finds its previous priority level using the map, or 0 if its not in the map. It then checks if the execution time is higher than the allotted quantum value, if it is, the priority of the thread is decreased (increased in value) by floor(exec_time / quantum). If the new value exceeds the minimum, it is set to the minimum instead. The new priority value is stored in the map. The thread then tries to find the next thread to wake up by checking each level of the queue from highest to lowest. If it cannot find a sleeping thread, it sets flag to 0, releasing the mutex, releases the guard lock and returns from unlock(). If it does find a sleeping thread, it calls unpark() to wake up the next thread and returns from unlock(). As mentioned previously, the next thread continues from where it called park() and starts its timer to measure its critical section.

This implementation guarantees correctness through the exclusive control provided by atomic flags. The guard atomic flag ensures that only a single thread can enter the critical sections of the lock() and unlock() methods at any time, preventing race conditions. The mutex lock is acquired only if the flag variable is 0, and it is set immediately upon entry, ensuring mutual exclusion. If the lock is not available, threads are parked using the park method of the Garage class, which utilizes atomic booleans to safely manage thread sleep and wake states, ensuring that no thread is accidentally skipped or woken prematurely.

Fairness is addressed by integrating a priority-based queue system, which gives the name to the mutex. Threads are assigned priority levels, and they are placed in corresponding queues based on these levels. This mechanism ensures that threads are served in an order that respects their priority, preventing starvation. Threads that exceed their execution time relative to a predefined quantum, are dynamically reprioritized,

increasing their priority level (lower numerical value) if they run shorter than expected, and decreasing it (higher numerical value) if they exceed.

Spin waiting can result in the busy waiting problem in which the CPU does nothing but spin resulting in wasted CPU cycles, however this implementation aims to reduce the spin wait time to the minimum by putting itself to sleep and allowing a finished thread to wake it up, resulting in a more performant implementation.