2023 – 2024 Spring Term

CS307 – Operating Systems Course

Programming Assignment 2 Report

-

Berke Ayyıldızlı

31018

Program Outline

In this programming assignment, we are asked to implement a Multi-Level-Feedback-Queue (MLFQ) scheduling algorithm in c++, utilizing different classes such as queue.h, a queue implementation using pointers in a linked list fashion, which I used a modified version of Michael&Scott's, and a park.h, which has methods to ensure that the threads do not fall into busy-waiting problems. The implementation and the use of queue here hold the upmost value, as it helps us to keep track of the thread that holds the head node. Meaning if the enqueue operations are correct, the dequeue operations would also be correct. Here, because we are not using the standart library <queue>, we needed to implement some functions to use in our disposal. These methods will be explained in their respected spots. The queue implementation also needed to have a template based structure, because rather than standart data types like strings or integers, we would be working with pthreads's threads. The park.h header file holds the garage class, as it helps us to not spin-wait threads, while they try to acquire the locks. It's methods like park() and unpark() is used to wait threads in a safe way. The last header file MLFQmutex.h holds the MLFQMutex class, as it is the main class that initializes and maintains the locking and unlocking methods. These methods properly create and enqueue threads to their respected priority queues. Another important aspect of this class is that it also uses another data structure such as unordered maps to hold the information of priorities of different threads. Without this kind of a structure, we wouldn't be able to change the priorities and move the threads to their respected queues. Again, it's methods will be explained later. An atomic bool is used to hold the information of who owns the lock, as explained in the instructions. When a thread acquires the mutex in the lock() method, a

timer starts, continueing until it's entry to the unlock() method. When the same thread enters the unlock() method, chrono computes the time it takes for the thread to do its job, and compares it to our quantum (qVal) value. If it is less than quantum, the thread can stay in the same priority level, but if not, a calculation is made to enqueue it to the correct spot. Whole operations are done in the same fashion, and the program execution time is printed in the end.

queue.h

When implementing the queue.h, I covered all method bodies with mutex locks. This approach was necessary to ensure the thread-safety and the prevention of concurrent access. How I achieved this is explained below.

This class holds 4 methods along with its constructor and destructor: enqueue(), dequeue(), isEmpty() and print(). Also because of the linked list fashion, I implemented a QueueNodes struct in the beginning, having a typename T object and a QueueNodes pointer. In the private section of the queue, we create 2 more QueueNodes pointers named as head and tail. These are going to be used in the following operations. Heavily influenced by the queue implementation by Michael&Scott from our book OSTEP, my enqueue() method works in a similar manner. After the mutex lock, a dummy node has been created to assign the head and tail if the tail is null. Else, the next node after the tail becomes the dummy node and tail node becomes the dummy node again. After the insertion of the new node, the method ends by unlocking the mutex. My dequeue() method also works in a similar fashion, starting with a mutex lock, the code first checks if the queue is empty by looking to the head node. After, it also creates a temporary node, and makes it equal to the next node after head. Than, the dequeued value gets equal to the value of the next node of the temporary node. After the operation is completed, we delete the temporary node and return the dequeued value. Other method isEmpty() also checks the head node, and if it is equal to a null value, it returns true. Specifically designed for the test case, print() method prints the information of each node one by one.

park.h

Although the garage class is given to us to both simplify the code and increase the performance, for the sake of my code to function true, I did not use it during the execution. Of course I know what the methods do, such as park() to be used as to change the flag and wait the threads and unpark() to both change flag and notify the threads. Garage class, as we have seen it before in the solaris application, holds an upmost value in the terms of conqurrent data structures and threads. Here, also utilizing a hash map, the garage class can be further utilized to decrease busy-waiting and ensuring an equal implementation.

MLFQmutex.h

The main implementation of the MLFQ, holds 3 methods with a constructor. We first start by declaring the required values such as execTime and qVal, along with our atomic bool and chrono variables. Here, I used a vector of queues of pthreads, which will hold our priority queues. I also declared a map with pthreads and integers. Also an owner thread is also declared here. In the constructor, I used .resize, to ensure that there is a queue for each priority level. In the lock() method, the id of the thread is taken, than, with a quick check from the map, if it is the first time of the thread to enter here, its priority is arranged as 0, and it is enqueued to the suitable queue. After this part, a never ending loop is used to spin-wait other threads. Here, thanks to the <atomic> library, I used the .compare_exchange_strong() function to make a compare and swap lock. If it returns true, owner becomes the current thread, and the clock starts to tick. In the unlock() method, we first start by checking if the thread is the owner of the lock, if not, it gives an expection. Of course that never happens, but let's continue. Because the owner of the lock comes in, we stop the timer, and find the execTime. If the execTime is bigger than quantum value qVal, we decrease it to the necessary queue by making the calculations. It is first done by dequeing the thread from its old queue, and enqueueing it to its new priority level's queue. After the end of the operation, owner bool is reinitialised and locked bool is also resetted to false. The last function print(), utilizes the print() function from the queue implementation, for each level, it calls for the queue's print, and prints the values that the queue holds.

The spinning fashion of the lock and the use of an atomic bool, ensures that the fairness and correctness are maintained. Performance wise, however, due to my implementation, not using park() and unpark() may have negative effects.

The Program Hierarchy

On the start of a test program, necessary queues and variables are initialized. When a thread first comes, knowing it wants to access to the critical section, it needs to get hold of the lock, for this aim, it gets to the lock() method in the MLFQmutex class. In the beginning, since it is the first time for it to get in the method, it is placed in the priority level 0 queue. Other threads that come after it would be also placed in the same queue because of the fifo fashion. After its work is done, the same thread tries to get in the unlock() method. At this point, it's time to complete is computed and classified as execTime. Before the next thread starts, a compare operation is done to know whether it is necessary to change the priority of the thread. If it is needed, the dequeue operation is called first from the queue.h class, and than it is enqueued to the respected priority queue. According to the needs of the test program, the print() method can be called here utilizing the print() from the queue.h header file. The hierarchy continues like this for all other threads, making them acquire the lock, if not, let them spin and wait them in their priority queues, while holding the priority information in the hash map. When all threads finish their execution and there isn't any threads left in the queues, the program terminates, printing the total time it takes for the program to run.