Sabanci University


Operating Systems

CS 307

# Programming Assignment - 2: MLFQ Mutex Implementation

Release Date: 15 April 2024
Deadline: 29 April 2024 23.55

# 1 Problem Description

Mutexes (locks) are basic synchronization primitives that are used for ensuring atomicity of critical sections. In the lectures we have seen several multi-threaded mutex implementations and evaluated them on three criteria:

- Correctness (Safety condition)
- Fairness (Liveness condition)
- Performance

In this Programming Assignment (PA), you are asked to develop your own mutex implementation in C++ language. We expect your implementation to satisfy above criteria as follows:

- **Correctness:** Your mutex class will contain two API methods: `lock` and `unlock`. As usual, these methods do not take any input parameters and do not return anything. We expect the standard correctness conditions from these methods:

  1. **Mutual Exclusion:** If multiple threads call `lock` method when the mutex is free, at most one of them must return from it. Moreover, when a thread owns the mutex , other threads have to block inside the `lock` method until the owner returns from `unlock`.
  2. **Deadlock Freedom:** If multiple threads call `lock` method when the mutex is free, at least one of them must return from it. Similarly, when the mutex is owned, there are other threads waiting inside the `lock` method and the owner returns from the `unlock` method, one of the waiting threads must acquire the ownership of the mutex .

- **Fairness:** In the lectures, we have seen fairness as a desirable property of the scheduler. A fair scheduler is an assumption we should expect from the Operating System (OS) for implementing a fair mutex . On top of it, we should develop our fairness mechanisms in our mutex implementation. In the lectures, we have seen a few fair mutex implementations such as *ticket lock*, *Solaris Lock* and *Futex*. In your mutex implementation, we expect you to use a basic version of Multi-Level

Feedback Queue (MLFQ) to ensure fairness. In the lectures, MLFQ was introduced for fair scheduling of jobs (threads, processes) in the kernel level. In this PA, you will implement your own MLFQ in the user-level and use it in your mutex implementation for its fairness. Details of the MLFQ algorithm will be explained in Section 1.1.

- **Performance:** In the lectures, we said that basic spin-locks suffer from busy-waiting problem. A common pattern we observed was that when the mutex is owned, other threads inside the `lock` method were looping in a `while` statement until the mutex was released and they were wasting CPU cycles doing no useful job. In order to solve this problem, we introduced performant mutex implementations (Solaris, Futex) using OS API methods like `park` and `unpark` to put threads waiting for a mutex into blocked state and unblock them again when they acquire the mutex . In your C++ implementation, you have to use similar mechanisms to avoid busy-waiting problems.

We complete the PA2 specification by explaining the MLFQ algorithm that you will implement.

## 1.1 MLFQ Algorithm

In the OSTEP book, Chapter 8, describes the MLFQ algorithm for scheduling multiple jobs. For your mutex implementation, you will use it for picking the next thread that will own the mutex after the current owner releases it if there are multiple waiting threads.

As you would remember, MLFQ algorithm keeps multiple queues with different priority levels and I/O bound jobs are kept in higher priority queues to increase their response time. Similarly, we want to keep threads with short critical sections in high priority queues.

Since we cannot know total runtime of jobs before running them, MLFQ tries to estimate a CPU-burst of a job from its previous CPU bursts. Similarly, we cannot predict the exact critical section length of a thread before running it. Hence, we need to keep track of running time of a thread while it is executing its critical section for future reference.

Once we have this information, then you can implement the MLFQ algorithm obeying the following rules:

1. If a thread is calling the `lock` method for this mutex for the first time, it is considered to be belonging to the highest priority queue which belongs to level 0.

2. Quantum value or the time-slice for each priority level is the same and it is called $Qval$. When the mutex owner does not consume all of its $Qval$ time during its critical section execution, it stays in the same priority level.

3. If the critical section execution time called $execTime$ of the current mutex owner exceeds $Qval$, then the priority level of this thread is incremented by $\lfloor \frac{execTime}{Qval} \rfloor$. However, a thread's priority level cannot exceed the maximum level. For instance, if the current mutex owner thread is already at queue level 3, its critical section took 70 ms to complete and the quantum value is 20 ms, then this thread moves to priority level $3 + \lfloor \frac{70}{20} \rfloor = 3 + 3 = 6$ after releasing the mutex . However, if the lowest priority level is 5, then it moves to level 5.

4. If there are multiple waiting threads, MLFQ always picks the one in the lowest level. If there are multiple threads in this level, they acquire the mutex in Round-Robin order.

# 2 C++ Implementation

In this PA, we expect you to submit three C++ header files along with your report.

## 2.1 Concurrent Queue: `"queue.h"`

Your MLFQ algorithm will maintain queues for different priority levels. For this purpose, you have to implement your own queue. As we will see in the lectures, sequential data structure implementations are usually incorrect when they are accessed and manipulated by concurrent threads. For instance, when multiple threads try to enqueue or dequeue to the same queue concurrently, problems like double insertions, double deletions, lost insertions or lost deletions are possible.

In order to avoid these problems, you have to implement a *thread-safe* queue. In order to achieve this, you have to use synchronization mechanisms.

The simplest way is to wrap method bodies of the queue library with mutex `lock` and `unlock` method calls. In this case, performance of the methods will be poor since they execute sequentially and you have to be careful about deadlocks.

Concurrent queues are well-studied data structures and there are various efficient algorithms. One of them is called Michael&Scott queue and you can find its C implementation in OSTEP book, Page 11, Chapter 29. If you wish, you can use other implementations like lock-free queues or develop your own efficient concurrent queue.

The most important point is that we will evaluate your queue implementations separately from the mutex implementation. We will let multiple concurrent threads to execute `enqueue` and `dequeue` operations on the same queue.

Your queue implementation must have the following features:

- In addition to `enqueue` and `dequeue` methods, your queue API must include `isEmpty` and `print` methods. `isEmpty` method does not take any input and returns a `bool` value. If the queue is empty, it returns `true`. Otherwise it returns `false`. `print` method does not have any input or output variables. It just prints the elements from head to tail.

- In `"queue.h"` file, you must implement the class named *Queue*. This class must be a template class in the sense that elements of the queue can be of any types.

## 2.2 Blocking and Unblocking Threads: `"park.h"`

When a thread is waiting for a mutex to be free, it might inform the OS about it and request the OS to block this thread until it acquires the mutex . For this communication, you have to implement a class called *Garage* in the file called `"park.h"`.

This class will implement two methods: `park` and `unpark`. `park` method will put the caller thread into the blocked state such that it cannot be picked by the scheduler and continue its execution. On the other hand, `unpark` method will get a thread identifier of type `pthread_t` as input and mark it as "runnable" again.

In the PA2 package, we already provide an implementation of this class using condition variables which we will learn about later on. You can use it

as is or replace it with your own `park` and `unpark` methods to make it more efficient.

In your submission, you have to include `"park.h"` file even though you did not modify the sample implementation.

## 2.3 The Mutex Implementation: `"MLFQmutex.h"`

In this file, you will implement the *MLFQMutex()* class. This class must have four public methods: the constructor, `lock`, `unlock and print` methods.

The constructor takes two parameters: an integer showing the number of priority levels and a double to denote the quantum value of the priority levels in **seconds**.

In the `lock` method, a thread first checks if the mutex is free. If this is the case, the caller thread owns the mutex , starts the timer to measure critical section length and returns. You can use a simple variable like `flag` we saw in the spin-locks to keep track of whether a mutex is free or not. If the mutex is already owned by some other thread, you have two options: either you can keep spinning and checking if the mutex becomes free or you can call the `park` method of the associated `Garage` object and put the caller thread into blocked state. If you choose the second option, you can assume that the thread returning from `park` method owns the mutex and then it can start the timer for measuring the execution time of its critical section.

For the owned mutex case, you do not actually have to choose one of these two options. Like Linux Futexes, you can implement a fast path and a slow path such that you first try several times to acquire the mutex and if these attempts fail, you can park the thread. Since the performance of your mutex will affect your grade (see Section 4 ), you can optimize the waiting times of threads by considering the cost of `park` and `unpark` methods.

In the `unlock` method, the first thing you should do is to stop the timer and measure the execution time of the critical section. Based on this value and the quantum length, you decide whether this thread remains in the same priority level or moves to another level considering the rules in Section 1.1. If it has to move to another level, you have to perform this operation inside the `unlock` method.

Please note that when the mutex owner calls the unlock method, you cannot insert it back to the one of queues keeping track of priority levels since these queues are reserved for threads who are waiting in the `lock` method. Hence, you need to keep a separate data structure like a hash-map

to determine which thread belongs to which priority level. If you do this, next time this thread calls the `lock` method when the mutex is taken, you can correctly decide the queue that this thread will be inserted to.

Next, we provide some implementation hints and details for the *MLFQ-Mutex* class:

- Implementations of the `lock` and `unlock` methods manipulate multiple objects like queue, hash-map and flag. Hence, concurrent calls to these methods might contain data races that cause incorrect behaviour of the mutex . In order to prevent this, you have to synchronize their method bodies. You can do this by using a simple spin-lock like the guard lock of the Solaris mutex we have analyzed in the lectures.

- In order to implement the inner guard lock as a simple spin-lock, you have to use instructions like Test-and-Set. For this purpose, you can include the `<atomic>` library of C++ . For more information on this library, please check the manual page. In this library, you can also find some mechanisms for blocking and unblocking threads that you can use in *Garage* class.

- Please do not use the default threading library of C++ (`<thread>`) in your mutex implementation. In the lectures, we learnt `<pthread>` library's API. Our automated grading scripts are consistent with it and we will assume that you used `<pthread>` library.

- You are not allowed to use time parametric synchronization primitives like `sleep` to synchronize threads.

- In order to measure the critical section execution times, you can utilize `<chrono>` library of C++ . You can find more information about this library here.

- When a thread is inserted to a waiting queue, we expect `lock` method to print a line in the following format:

  ```
  Adding thread with ID: <tid> to level <priority level>
  ```

  where $< tid >$ refers to the unique thread identifier assigned by the OS and $<$ *priority level* $>$ refers to the priority queue level in MLFQ that current thread is inserted to. Even though you ensure atomicity of your `lock` method implementation, printing this line might be buffered

and delayed. In that case, This line might be interrupted by other print statements and you can produce garbage output on the console. In order to prevent this, please use `cout.flush()` which enforces print statements to execute immediately.

Lastly, `print` method prints the contents of the queues by calling their print member methods. For more information on the printing format, please check sample runs in Section 5. You can safely assume that `print` methods are always called by a thread after acquiring the mutex so that it does not constitute a data race.

# 3 Submission Guidelines

For PA2, you are expected to submit four files:

- **queue.h**: C++ header file containing the *Queue*<*T*> template class including public methods with the following signatures: `"Queue()"` (constructor), `"void enqueue(T item)"`, `"T dequeue()"`, `"bool isEmpty()"` and `"void print()"`.
- **MLFQmutex.h**: C++ header file containing the *MLFQMutex* class including public methods with the following signatures: `"MLFQMutex(int, double)"` (constructor), `"void lock()"`, `"void unlock()"` and `void print()`.
- **park.h**: C++ header file containing the *Garage* class including public methods with the following signatures: `"Garage()"` (constructor), `"~Garage()"` (destructor), `"void park()"`, and `"void unpark(pthread_t))"`.
- **report.pdf**: Your report that explains your queue and mutex implementation. In the first part, please explain whether you picked an existing concurrent queue algorithm or developed your own version. Then, please discuss, how this algorithm ensures thread-safety i.e., how FIFO order is preserved, it does not allow data loss or duplication in case of concurrent accesses. In the second part, describe your mutex implementation and argue why it ensures the correctness, fairness and performance requirements. Please note that all the reports are read carefully to understand your program.

During the submission of this homework, you will see two different sections on SUCourse. You are expected to submit your files separately. You should NOT zip any of your files. Please submit your report.pdf to "PA2 – Report Submission" and your three C++ header files to "PA2 – Code Submission". The files that you submit should NOT contain your name or your id. SUCourse will not except if your files are in another format, so please be careful.

**IMPORTANT NOTE:** If the file names, class names or the public method signatures do not match with the format explained above, you might get 0 from this assignment since the automated tests will fail. Your C++ files must be able to be compiled and run with sample `"main.cpp"` files we provided in the PA2 bundle. If you need to modify `"#include"` statements or the code in those files, it means that you are not faithful to the format provided above.

**IMPORTANT NOTE 2:** In order to compile and test your implementation, please use the **makefile** provided in the PA2 bundle or use the commands in this files. We will use these commands for compiling your implementation during the automated grading and evaluation.

# 4 Grading

- **Compilation and termination (10 pts):** Given a `"main.cpp"` file that uses your mutex , when it is compiled with the command:

  ```
  g++ -o main main.cpp -I. -std=c++20 -pthread
  ```

  it successfully generates the executable file `"main"`. Moreover, all the tests successfully terminate. Successful termination entails that the program never deadlocks.

- **Queue implementation (20 pts):** Your queue implementation works correctly in the case of both sequential access and multi-threaded access scenarios.

- **Single layer MLFQ (10 pts):** When a *MLFQMutex* instance is initialized with a single layer and multiple concurrent threads try to acquire and relax it concurrently, it orders them in Round-Robin fashion.

- **Single Thread and Multi-layer MLFQ (10 pts):** When a multi-layer *MLFQMutex* instance is sequentially acquired and released by a single thread multiple times, it always inserts the thread in the correct priority level .

- **Correctness tests (30 pts)**: Your *MLFQMutex* implementation will be tested by multi-threaded programs in which threads try to acquire and release a multi-level mutex concurrently.

- **Performance tests (10 pts):** We expect your mutex implementation to be efficient and not suffer from busy-waiting problems. Therefore, we will run each test case several times with your implementation and take the average running times. If the average is less than %140 of our implementation's running time, you will get full score from this part. If it is between $\%140 - \%200$, you will get 5 points. Otherwise, you get 0. In Section 5, we provide the average running times of our implementation in the server for the sample programs provided in the PA2 bundle.

- **Report (10 pts):** We expect you to submit a report summarizing your work, satisfying the expectations explained in Section 3.

# 5 Sample Runs

In PA2 bundle, we provide some sample programs to test your implementations. In this section, we present the expected outcomes of these programs.

## 5.1 Sample Queue Test

The program in `"sampleQueue.cpp"` generates two threads each of which tries to push 100 elements to a shared queue. After the pusher threads terminate, the main thread pops 100 elements from the queue and prints the queue state.

The program uses *Queue* class implemented in `"queue.h"` file which refers to your implementation. When compiled and run with your queue implementation, you should expect 100 elements in the queue that does not have any duplicates. A sample output we obtained is as follows:

```
Hello, from main.
Thread with base: 0 started.
Thread with base: 100 started.
Threads terminated. Resulting queue state:
136 137 138 139 140 141 142 143 144 145 146 147 148
   149 150 151 152 153 154 64 155 65 66 67 68 69 70 71
    72 73 74 75 156 76 157 158 159 160 161 162 163 164
    165 77 166 78 167 79 168 80 169 81 170 171 172 173
    174 175 176 177 178 179 82 180 83 181 182 183 184
   185 186 187 188 189 190 84 191 85 192 86 193 87 194
    88 195 89 196 197 198 199 90 91 92 93 94 95 96 97
   98 99
```

Since all the content of the queue cannot be printed in a single line of this document, it contains multiple lines and arbitrary white spaces in the beginning and end of lines. As you can observe from the C++ code, queue contents are printed in a single line.

## 5.2 Sample Mutex Test with One Priority Level

The program in `"sample1Level.cpp"` declares a shared *MLFQMutex* with 1 priority level. Then, four threads concurrently try to acquire this mutex to execute their critical section three times each. We assign program IDs to threads in the range $[0, 3]$ and keep track of their critical sections by printing some lines before and after their ciritical sections. When you run this program, we expect an output like the following:

```
Thread with program ID:0 acquired lock
Adding thread with ID: 139661784073792 to level 0
Thread with program ID:0 releasing lock
Adding thread with ID: 139661775681088 to level 0
Adding thread with ID: 139661767288384 to level 0
Thread with program ID:1 acquired lock
Adding thread with ID: 139661792466496 to level 0
Thread with program ID:1 releasing lock
Adding thread with ID: 139661784073792 to level 0
Thread with program ID:2 acquired lock
Thread with program ID:2 releasing lock
Adding thread with ID: 139661775681088 to level 0
Thread with program ID:3 acquired lock
```

```
Thread with program ID:3 releasing lock
Adding thread with ID: 139661767288384 to level 0
Thread with program ID:0 acquired lock
Thread with program ID:0 releasing lock
Adding thread with ID: 139661792466496 to level 0
Thread with program ID:1 acquired lock
Thread with program ID:1 releasing lock
Adding thread with ID: 139661784073792 to level 0
Thread with program ID:2 acquired lock
Thread with program ID:2 releasing lock
Adding thread with ID: 139661775681088 to level 0
Thread with program ID:3 acquired lock
Thread with program ID:3 releasing lock
Adding thread with ID: 139661767288384 to level 0
Thread with program ID:0 acquired lock
Thread with program ID:0 releasing lock
Thread with program ID:1 acquired lock
Thread with program ID:1 releasing lock
Thread with program ID:2 acquired lock
Thread with program ID:2 releasing lock
Thread with program ID:3 acquired lock
Thread with program ID:3 releasing lock
Threads terminated. Total duration is: 0.00159688
   seconds.
```

Your output might be slightly different than this one. First of all, in which order they will acquire the mutex depends on the scheduler. However, after one iteration of all the threads, they must keep obtaining the mutex in the same Round-Robing order since MLFQ has one priority level and critical sections of threads are longer than $Qval$.

In addition to print statement by the program, there are some lines printed in the `lock` method when a thread calls it while the mutex is already acquired. These lines start with **"Adding thread with ..."** These lines print the OS assigned unique thread ID instead of the program IDs. Places of these lines also depend on the scheduler and cannot be determined beforehand.

However, after one iteration of the threads, next iterations follow the same Round-Robin order.

## 5.3 Sample Mutex Test With Multiple Priority Levels

The program in "sampleMultiLevel.cpp" is a modified version of the previous program. Instead of a single priority level, this one keeps a mutex with 7 priority levels. Instead of four threads, this program runs three concurrent threads trying to acquire the mutex . Critical section lengths of threads are directly proportional with their program IDs. Hence, the thread with higher program ID sinks down to higher level queues faster.

A sample output is provided below:

```
Thread with program ID:Adding thread with ID: 0
   acquired lock139946337879616 to level
0
Adding thread with ID: 139946329486912 to level 0
Thread with program ID:0 releasing lock
Adding thread with ID: 139946346272320 to level 0
Thread with program ID:1 acquired lock
Thread with program ID:1 releasing lock
Adding thread with ID: 139946337879616 to level 2
Thread with program ID:2 acquired lock
Thread with program ID:2 releasing lock
Adding thread with ID: 139946329486912 to level 3
Thread with program ID:0 acquired lock
Thread with program ID:0 releasing lock
Adding thread with ID: 139946346272320 to level 1
Thread with program ID:1 acquired lock
Thread with program ID:1 releasing lock
Adding thread with ID: 139946337879616 to level 4
Thread with program ID:0 acquired lock
Thread with program ID:0 releasing lock
Thread with program ID:2 acquired lock
Thread with program ID:2 releasing lock
Adding thread with ID: 139946329486912 to level 6
Thread with program ID:1 acquired lock
Thread with program ID:1 releasing lock
Thread with program ID:2 acquired lock
Thread with program ID:2 releasing lock
Threads terminated. Total duration is: 18.002 seconds.
```

As can be seen from the output, `lock` method's printed lines might interleave with program's console output even though `cout.flush()` is used

after every `cout` statement. The important point here is that, thread with program ID 2 loses 3 priority levels after each critical section whereas this number is 2 for thread 1 and 1 for thread 0.