Tolga Toker 32639

PA2 – Report

Report: Implementing a Thread-Safe MLFQ Mutex in C++

This report details the design and implementation of a thread-safe Multi-Level Feedback Queue (MLFQ) mutex in C++. The project consists of three C++ header files:

- **queue.h:** Implements a thread-safe concurrent queue using the atomic_queue::AtomicQueueB class.
- **park.h:** Provides mechanisms for parking and unparking threads using a Garage class with park and unpark methods.
- **MLFQmutex.h:** Contains the MLFQMutex class, which implements the core mutex functionality with lock, unlock, and print methods.

Concurrent Queue Implementation (queue.h)

The queue.h file utilizes the atomic_queue::AtomicQueueB class, which offers a lock-free, concurrent queue implementation. This class ensures thread-safety by employing atomic operations and memory ordering constraints to avoid data races and maintain consistency. Key features include:

- **Lock-free design:** Avoids the performance overhead of traditional locking mechanisms like mutexes.
- **Atomic operations:** Guarantees atomicity of enqueue and dequeue operations, preventing data corruption.
- **Memory ordering:** Ensures proper ordering of memory accesses to maintain data consistency across threads.

The provided Queue class wraps the AtomicQueueB class and exposes a simplified interface with enqueue, dequeue, isEmpty, and print methods. This allows for easy integration with the MLFQ mutex implementation.

Thread Parking and Unparking (park.h)

The park.h file defines the Garage class, which facilitates the parking and unparking of threads. This class uses an unordered_map to store an atomic boolean flag for each thread. The park method sets the flag for the current thread to false and waits until it becomes true. The unpark method sets the flag for the specified thread to true and notifies the waiting thread. This mechanism enables efficient waiting without busy-waiting, as threads are put into a blocked state until they are needed.

MLFQ Mutex Implementation (MLFQmutex.h)

```
class MLFQMutex {
private:
    int numLevels;
    double quantum;
    std::atomic<bool> locked;
    std::vector<Queue<pthread_t>> queues;
    std::unordered_map<pthread_t, int> threadLevels;
    std::unordered_map<pthread_t, std::chrono::time_point<std::chrono::system_clock>> threadStartTimes;
    Garage garage;
    pthread_mutex_t mtx;
```

- **Private members:**
    - numLevels: Stores the number of priority levels in the MLFQ.
    - quantum: Represents the time slice (quantum) for each priority level.
    - locked: An atomic boolean indicating whether the mutex is currently locked.
    - queues: A vector of Queue objects, one for each priority level, to store waiting threads.
    - threadLevels: An unordered map that tracks the current priority level of each thread.
    - threadStartTimes: An unordered map that stores the starting time of each thread's critical section.
    - garage: An instance of the Garage class used for parking and unparking threads.
    - mtx: A pthread_mutex_t used for internal synchronization within the lock and unlock methods.

```
public:
    MLFQMutex(int numLevels, double quantum) : numLevels(numLevels), quantum(quantum), locked(false) {
        queues.resize(numLevels);
        pthread_mutex_init(&mtx, NULL);
    }
```

- **Constructor:**
    - Initializes the numLevels, quantum, and locked members with the provided values.
    - Resizes the queues vector to accommodate the specified number of priority levels.
    - Initializes the mtx mutex using pthread_mutex_init.

```
void lock() {
        pthread_t self = pthread_self();
        int currentLevel = threadLevels.find(self) != threadLevels.end() ? threadLevels[self] : 0;
        bool expected = false;
```

- **lock method:**
  - Obtains the current thread's ID using pthread_self().
  - Retrieves the current priority level of the thread from the threadLevels map, defaulting to 0 if the thread is not found.
  - Sets expected to false, indicating the expectation that the mutex is currently unlocked.

```
// Fast path: try to acquire lock directly
        if (locked.compare_exchange_strong(expected, true)) {
            printf("Thread with ID: %lu acquired lock directly\n", self);
            fflush(stdout);
            threadStartTimes[self] = std::chrono::high_resolution_clock::now();
            return;
        }
```

- **Fast path:**
  - Attempts to atomically compare and exchange the value of locked with true. If successful, the lock is acquired, and the thread's start time is recorded in threadStartTimes.
  - Prints a message indicating successful lock acquisition.

```
printf("Adding thread with ID: %lu to level %d\n", self, currentLevel);
        fflush(stdout);
        // Slow path: enqueue and park
        pthread_mutex_lock(&mtx);
        queues[currentLevel].enqueue(self);
        garage.setPark();
        pthread_mutex_unlock(&mtx);
        garage.park();
        threadStartTimes[self] = std::chrono::high_resolution_clock::now();
    }
```

- **Slow path:**
  - If the fast path fails, the thread is added to the appropriate queue in the queues vector based on its currentLevel.
  - The mtx mutex is locked to ensure exclusive access to the queues and the garage.
  - The thread sets its park flag in the garage and then unlocks the mtx.

- The thread calls garage.park() to be put into a blocked state until it is unparked.
- Once unparked and owning the lock, the thread records its start time in threadStartTimes.

```
void unlock() {
        pthread_t self = pthread_self();
        // Stop timer and measure critical section execution time
        auto endTime = std::chrono::high_resolution_clock::now();
        auto startTime = threadStartTimes[self];
        auto duration = std::chrono::duration_cast<std::chrono::seconds>(endTime - startTime).count();
```

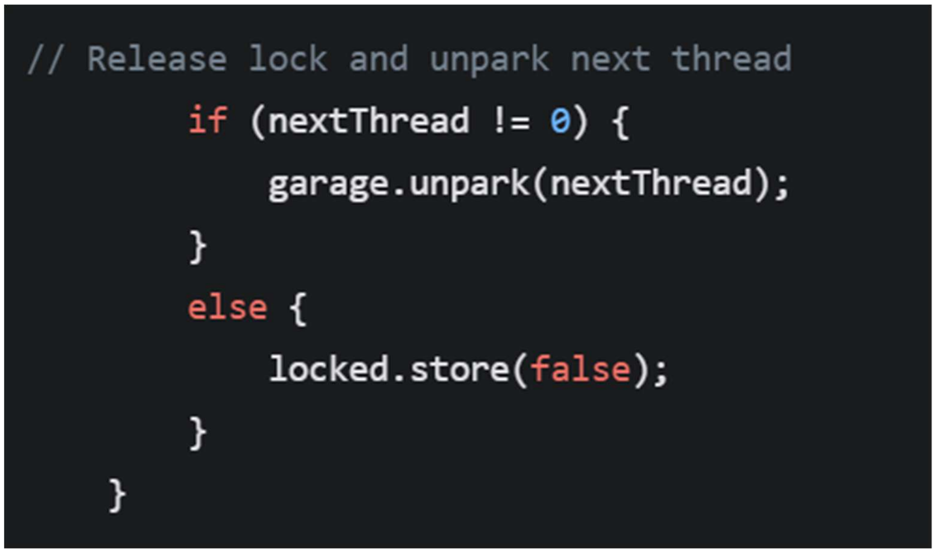
This code block starts the unlock method. It first retrieves the current thread's ID using pthread_self(). Then, it calculates the duration of the critical section by subtracting the start time (recorded during lock) from the current time and converting it to seconds.

```
// Update thread's priority level
        int levelChange = static_cast<int>(duration / quantum);
        int newLevel = std::min(threadLevels[self] + levelChange, numLevels - 1);
        threadLevels[self] = newLevel;
```

Here, the code determines the change in priority level based on the critical section duration and the quantum value. The new level is calculated by adding the level change to the current level, ensuring it doesn't exceed the maximum number of levels. Finally, the thread's level is updated in the threadLevels map.

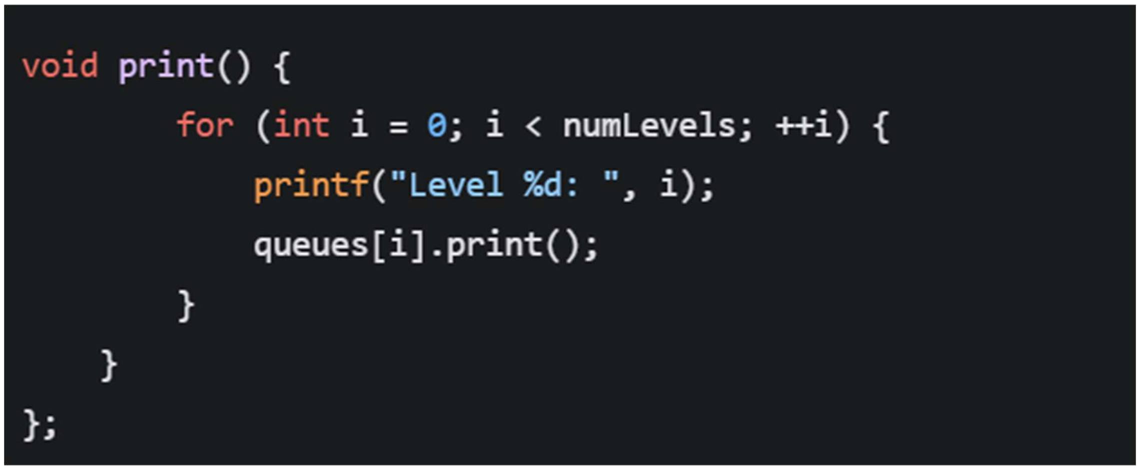
```
// Find next thread to run
        pthread_mutex_lock(&mtx);
        pthread_t nextThread = 0;
        for (int i = 0; i <= numLevels; ++i) {
            if (!queues[i].isEmpty()) {
                nextThread = queues[i].dequeue();
                break;
            }
        }
        pthread_mutex_unlock(&mtx);
```

This section acquires the internal mutex (mtx) to ensure thread-safe access to the queues. It iterates through the queues from the highest priority level to the lowest, searching for a non-empty queue. If found, the first thread in that queue is dequeued and stored as nextThread. The mutex is then released.



```
// Release lock and unpark next thread
        if (nextThread != 0) {
            garage.unpark(nextThread);
        }
        else {
            locked.store(false);
        }
    }
```

Finally, the code checks if a next thread was found. If so, it is unparked using the garage.unpark method. If no threads are waiting, the locked atomic boolean is set to false to indicate that the mutex is now free.



```
void print() {
        for (int i = 0; i < numLevels; ++i) {
            printf("Level %d: ", i);
            queues[i].print();
        }
    }
};
```

The print method simply iterates through each priority level and prints its contents by calling the print method of the corresponding queue.

The MLFQmutex.h file defines the MLFQMutex class, which implements the MLFQ mutex with the following features:

- **Multiple priority levels:** Threads are assigned to different priority levels based on their critical section execution time.
- **Quantum-based scheduling:** Threads are given a time slice (quantum) to execute their critical section before being moved to a lower priority level.
- **Fairness:** The MLFQ algorithm ensures that threads with shorter critical sections are given higher priority, leading to fairer scheduling.

- **Performance:** The implementation avoids busy-waiting by parking threads when the mutex is locked, improving efficiency.

The lock method attempts to acquire the lock directly. If unsuccessful, the thread is added to the appropriate queue based on its current priority level and parked. The unlock method stops the timer, measures the critical section execution time, updates the thread's priority level, finds the next thread to run using the MLFQ algorithm, and unparks that thread. A print method is also provided to display the contents of each priority queue.

Thread Safety and Synchronization

The implementation ensures thread safety through the following mechanisms:

- **Concurrent queue:** The lock-free atomic_queue::AtomicQueueB class guarantees safe concurrent access to the queues.
- **Atomic operations:** Atomic variables and operations are used to prevent data races when accessing shared data structures.
- **Mutex for internal operations:** A pthread_mutex_t is used to synchronize access to critical sections within the lock and unlock methods, ensuring consistent state updates.

Conclusion

This implementation demonstrates a thread-safe and efficient MLFQ mutex in C++. The use of a lock-free concurrent queue, atomic operations, and a parking mechanism contributes to fairness and performance. By combining these elements, the MLFQ mutex provides a robust solution for synchronizing access to shared resources in multi-threaded applications.