Programming Assignment - 2:
MLFQ Mutex Implementation

Mehmet Altunören

22539

1. Overview

In this report, I describe two key components I developed for PA2:

1. A **thread-safe concurrent queue** (queue.h), and
2. An **MLFQ-based mutex** (MLFQmutex.h).

I explain my design choices, demonstrate how each implementation satisfies the correctness requirements under concurrency, and argue how they achieve fairness and performance goals.

---

2. Concurrent Queue Implementation (queue.h)

2.1 Algorithm Choice

To implement a thread-safe FIFO queue, I adopted the classic **two-lock linked-list queue** pattern (Michael & Scott style). I chose this because:

- It cleanly separates enqueue and dequeue work under distinct locks.
- It avoids heavy contention: enqueuers lock only the tail, while dequeuers lock only the head.
- It is simple to reason about and adapt in C++ with pthread_mutex_t.

2.2 Internal Data Structures

- **Node<T>**: each node stores a T value and a Node* next.
- **Dummy node**: on construction, I allocate one dummy node that both head and tail point to.
- **Pointers**:
    - head always refers to the dummy or the node preceding the first real element.
    - tail refers to the last real node (or the dummy if empty).
- **Mutexes**:
    - head_lock (a pthread_mutex_t) protects operations on the head pointer and its next link.
    - tail_lock protects operations on the tail pointer and its next link.

2.3 API Methods

Constructor & Destructor

- **Constructor**: allocates the dummy node, initializes both locks.
- **Destructor**: iterates from head, deleting each node; destroys both mutexes.

enqueue(const T& item)

1. Allocate a new Node<T> containing item.

2. Lock tail_lock.
3. Link the new node at tail->next and set tail = newNode.
4. Unlock tail_lock.

Because only enqueuers ever touch tail, multiple threads can call enqueue() concurrently without interfering with dequeuers.

dequeue() → T

1. Lock head_lock.
2. Let first = head->next.
   - If first == nullptr, the queue is empty: unlock head_lock and throw std::runtime_error("Queue is empty!").
3. Copy first->value to a local variable.
4. Advance head to first, delete the old dummy node.
5. Unlock head_lock.
6. Return the copied value.

Only one thread at a time can manipulate head, ensuring mutual exclusion for removals.

isEmpty() / print()

- **isEmpty()** locks head_lock, checks head->next == nullptr, unlocks, and returns the result.
- **print()** locks head_lock and iterates from head->next, printing each element (or "Empty" if none), then unlocks.

2.4 Thread-Safety and FIFO Guarantee

- By partitioning access, enqueues and dequeues proceed entirely in parallel when the queue is non-empty and not changing head and tail simultaneously.
- No element can be lost or duplicated: each node is either safely linked under tail_lock, or unlinked under head_lock.
- FIFO order is enforced because nodes are always appended at the tail and removed from the head in arrival order.

2.5 Performance Considerations

- Lock granularity is minimal: only single operations under each lock.
- Contention only occurs when many threads enqueue at once (contending for tail_lock) or dequeue at once (contending for head_lock).
- Average enqueue/dequeue latency is low and scales well under moderate concurrency.

---

3. MLFQ-Based Mutex Implementation (MLFQmutex.h)

3.1 MLFQ Algorithm Recap

I embedded a **user-level Multi-Level Feedback Queue** scheduler into a pthread mutex to ensure fairness among waiting threads:

1. **Initial priority**: each thread's level (threadPriority) is 0 on its first lock().
2. **Quantum**: a fixed time slice Q (in seconds) specified at constructor time.
3. **Priority update**: after exiting the critical section, measure the elapsed time execTime; set newLevel = min(oldLevel + ⌊execTime / Q⌋, levels−1).
4. **Selection**: on unlock(), pick the **lowest non-empty** priority queue (round robin within the level) to unpark next.

3.2 Data Structures

- **thread_local int threadPriority**: tracks each thread's current level across invocations.
- **atomic_flag mutexFlag**: the core lock bit—test_and_set(memory_order_acquire) acquires, clear(memory_order_release) releases.
- **SpinLock internalLock**: a simple atomic_flag-based spin-lock guarding all shared state (queues, bitmask, flag).
- **vector<Queue<pthread_t>*> priorityQueues**: one queue per level to hold parked threads.
- **unsigned activeQueuesMask**: a bitmask where bit i = 1 if priorityQueues[i] is non-empty; used with __builtin_ctz to find the lowest non-empty level in O(1).
- **Garage garage**: provides setPark(), park(), and unpark(pthread_t) to block/wake threads without busy-waiting.
- **Timing fields**: startTime and endTime record the critical-section entry/exit times.

3.3 lock() Design

```
void lock() {
  internalLock.lock();
  if (!mutexFlag.test_and_set(memory_order_acquire)) {
    // Fast path: acquired immediately
    internalLock.unlock();
  } else {
    // Slow path: someone else holds the lock
    enqueueThread();      // enqueues under internalLock, prints & flushes
    garage.setPark();     // prevent missed wakeup
    internalLock.unlock();
    garage.park();        // block until unparked
  }
  // Record entry time for duration accounting
  startTime = high_resolution_clock::now();
```

```
}
```

- **Fast path**: zero-overhead acquired when uncontended.
- **Slow path**: under the guard lock, enqueue self at our current threadPriority, call setPark(), then unlock and park(). This sequence avoids the "lost wake-up" problem.

Printing: enqueueThread() writes "Adding thread with ID: <tid> to level <level>", and flushes immediately to avoid interleaved console output.

3.4 unlock() Design

```
void unlock() {
    internalLock.lock();
    endTime = high_resolution_clock::now();
    auto duration = duration_cast<seconds>(endTime - startTime);
    adjustThreadPriority(duration);          // bump level by floor(duration/Q)
    pthread_t next = dequeueNextThread();    // finds lowest non-empty queue
    if (next != (pthread_t)-1) {
        garage.unpark(next);                 // hand off to a waiting thread
    } else {
        mutexFlag.clear(memory_order_release);// no waiters: fully release
    }
    internalLock.unlock();
}
```

- Compute actual critical-section length, adjust the calling thread's threadPriority.
- dequeueNextThread() uses activeQueuesMask and __builtin_ctz to locate the next thread in minimal time.

3.5 Correctness

- **Mutual exclusion**: only one thread can clear test_and_set and proceed unparked; internalLock serializes all shared-state mutations.
- **Deadlock-freedom**: every unlock() either unparks a waiter or clears the flag, ensuring progress.

3.6 Fairness

- Threads that run short critical sections (I/O-like) remain at high priority (level 0), minimizing their wait time.
- CPU-bound threads sink to lower levels by $\lfloor \text{execTime}/Q \rfloor$ each time, letting short tasks "overtake" them.
- No thread can starve indefinitely because a long CPU-bound thread cannot stay at the top forever.

3.7 Performance

- **Fast path**: test_and_set with no extra locking when uncontended.
- **Thresholded spin-lock**: SpinLock yields to the scheduler after 1000 spins to avoid wasteful busy-waiting.
- **Park/unpark**: threads block in the OS rather than spin, eliminating CPU overhead under contention.

---

4. Use of Provided park.h

I utilized the supplied Garage implementation without modification. It correctly handles park(), setPark(), and unpark() to avoid lost wake-ups.

---

5. Conclusion

- My **concurrent queue** preserves strict FIFO under heavy concurrency while keeping enqueue/dequeue paths separate and low-overhead.
- My **MLFQ mutex** combines a fast, optimistic test-and-set path with a user-level feedback queue scheduler to balance correctness, fairness, and performance.