PA2 Report

Name, Surname: Beren Aydoğan
ID: 31938

Part 1: Concurrent Queue Algorithm

In my concurrent queue implementation, I have employed a variant of the Michael & Scott concurrent queue algorithm. This algorithm utilizes fine-grained locking, where separate locks are maintained for the head and the tail of the queue, minimizing lock contention. It uses the smart concept of a dummy node to avoid locking both locks when there is a single node in the list, making sure the queue never becomes empty.

My implementation ensures mutual execution through the use of two mutex locks:

- head_lock: To control access to the head of the queue.
- tail_lock: To control access to the tail of the queue.

By locking the critical sections that try to access the head and tail using these locks, my code prevents multiple threads from simultaneously modifying the state of the queue and avoids race conditions. These critical sections are present in:

- enqueue() method:
    - Creates a new node containing the item to be queued.
    - Locks the tail_lock to ensure exclusive access of a single thread to the tail of the queue.
    - Adds the new node to the tail of the queue.
    - Updates tail pointer to point to this new node.
    - Releases the lock.
- dequeue() method:
    - Locks the head_lock to ensure exclusive access of a single thread to the head of the queue.
    - Checks if the queue is empty by checking if head's next pointer is null.
        - If not empty,

                - Keeps the value of the node to be dequeued to be returned and keeps the new dummy node.
                - Dequeues by updating the head pointer to point to the next node, effectively making it the new dummy node.
                - Deletes older head node.
                - Releases the lock.
                - Returns the value of the dequeued node.
            - If empty,
                - Releases the lock.
                - Throws an exception and informs the user that the queue is empty.

My implementation also includes:

- Default constructor (Queue()):
    - Creates and inserts the dummy node into the queue.
    - Sets head and tail pointer to point to the dummy node.
    - Initializes pthread locks, head_lock and tail_lock, using pthread_mutex_init().
- Destructor (~Queue()):
    - Iterates through the queue to delete all nodes.
    - Destroys pthread locks, head_lock and tail_lock, using pthread_mutex_destroy().
- isEmpty() method:
    - Returns true if head->next == nullptr, false otherwise.
- print() method:
    - If not empty,
        - Iterate through the queue.
        - Print elements of nodes.
    - If empty,
        - Print “Empty”.

Preservation of FIFO Order

The FIFO (First-In, First-Out) order is maintained by the specific implementation of the queue operations:

- The enqueue operation always adds new elements to the end of the queue.
- The dequeue operation always removes elements from the front.

This ordering ensures that elements are processed in the exact sequence in which they were added, complying with the FIFO principle.

Preventing Data Loss and Duplication

The individual locks on enqueue and dequeue operations help in preventing data loss and duplication.

Since enqueue and dequeue operations are mutually exclusive within their respective sections (the head and the tail of the queue), there's no scenario where a concurrent operation could either result in skipping or losing a node (data loss) or the risk of duplicating entries (data duplication). Only one thread may perform an enqueue or dequeue operation at a time holding the respective lock, and the others who want to do these operations are blocked until the running thread leaves the lock.

<u>Part 2: Mutex Implementation</u>

My MLFQMutex class uses several private components:

- unordered_map<pthread_t, int> to map threads to their current priority levels.
- Two atomic_flag variables, flag and guard, to manage access to the mutex.
- A vector of queue pointers, vector<Queue<pthread_t>*> levels where each queue corresponds to a priority level.
- Number of the priority levels in the MLFQ implementation, noOfPriorityLevels.
- Quantum value of the queues, Qval which is same for all queues.
- Variables
    - o chrono::time_point<std::chrono::high_resolution_clock> start;
    - o chrono::time_point<std::chrono::high_resolution_clock> stop;

  to measure the duration for which a thread holds the mutex.

Private Helper Functions

Function Name: void updatePriorityLevel(pthread_t tid, chrono::seconds duration){}

- **What it does:** Adjusts the priority level of a thread based on the duration for which it held the mutex.
- **Purpose:** Ensures that threads holding the mutex for longer periods are assigned a higher priority and their priorities are in between the bound of queues.
- **Method:** Takes the duration a thread held the lock and converts it into seconds (execTime). Retrieves the current priority level of the thread using the thread ID (tid). Calculates the new priority level by adding an integer value determined by the floor division of the execution time by Qval. This value is intended to scale the impact of the hold time on the priority level. Ensures that the new priority level does not exceed the maximum available level (noOfPriorityLevels - 1). Updates the priority level of the thread in the threads map to the new calculated level.

Function Name: void enqueueThread(){}

- **What it does:** Manages the queueing of threads when they cannot acquire the mutex immediately**.**
- **Purpose:** Placing each thread in the correct queue based on its current priority listed in the map.
- **Method:** Determines the ID of the current thread. If the thread is not already in the threads map, it is added with a default priority level of 0. Adds the thread to the queue corresponding to its current priority level.

Function Name: pthread_t highestPriorityThread(){}

- **What it does:** Identifies and dequeues the thread with the highest priority among those waiting in the queues.
- **Purpose:** Scheduling the thread with the highest priority when the mutex is available. Letting the unlock() method know if there are no threads waiting.
- **Method:** Sets the next_thread variable initially to NULL. Iterates through the priority levels from highest to lowest. For each level, if the queue is not empty, it dequeues the first thread from the queue and breaks the loop. Returns the ID of the dequeued thread or NULL if no threads are waiting.

Public Methods

Parametrized Constructor (MLFQMutex(int givenNoOfPriorityLevels, double givenQval))

- Initializes the number of priority levels and the quantum value (Qval)
- Populates "levels" vector with queues, one for each priority level.
- Clears both flag and guard to indicate that the mutex is initially available.

Destructor (~MLFQMutex())

- Deallocates the memory for every queue in "levels".

lock() method

- Acquires the guard using a spinlock which ensures that no other thread can enter the critical section of lock() simultaneously.
- Checks if the flag can be acquired.
    - If successful, the thread obtains the mutex.
    - If the mutex is not available, the current thread is enqueued to its corresponding priority level stored in the map using the private helper function enqueueThread(). After enqueueing, setpark() is called to avoid race conditions that may occur before parking. Then, the thread prints its ID and level, releases the guard, and parks itself, suspending execution until it is unparked.
- Starts the timer when the calling thread acquires the lock to measure how long the thread will hold the mutex.

unlock() method

- Acquires the guard using a spinlock which ensures that no other thread can enter the critical section of unlock() simultaneously.
- Stops the timer when the thread acquires the guard.
- Updates the priority level of the thread based on the mutex hold time using the private helper function updatePriorityLevel(pthread_t tid, chrono::seconds duration).

- Attempts to dequeue the highest priority thread that is waiting by fetching its id using the private helper function highestPriorityThread(). If a thread is found, it is unparked; otherwise, the mutex flag is cleared.
- Clears the guard, allowing other threads to enter the critical sections of lock and unlock.

Correctness, Fairness, and Performance

- Correctness
  - Mutual Exclusion: The use of atomic flags ensures that at any given time, only one thread can enter the critical section.
  - Deadlock Prevention: The algorithm ensures that threads are either executing, waiting in a queue, or parked; a thread holding the mutex will always eventually release it and unpark another if any exists in the queue, preventing deadlocks.
- Fairness
  - Threads are dynamically prioritized based on their mutex hold time. This means that longer waiting times potentially decrease a thread's priority, giving other threads a chance to run.
  - By using a queue system, the algorithms prevents starvation by ensuring that all threads eventually get a chance to execute.
- Performance
  - The use of spinlocks for guard acquisition might degrade performance under high contention, causing a waste of CPU cycles by spinning. However, since lock() and unlock() methods are not long, the overhead they cause will not be as much as regular spinlocks.