CS 307 PA2 Report:
Load Balancing and Synchronization Mechanisms for Multi-Core Task Scheduling

Harun Yılmaz

30621

November 21, 2024

Contents

1 Introduction

Efficient scheduling in multi-core systems is crucial to optimize CPU utilization and overall performance. However, achieving this goal presents significant challenges, including maintaining load balance across cores and ensuring thread-safe data structures for concurrent operations.

This report focuses on two key aspects of designing a multi-core scheduler: the implementation of a **concurrent queue algorithm** to support thread-safe operations and the development of a **load balancing strategy** to distribute workloads effectively while minimizing the impact of task migration on cache affinity.

The following sections provide a detailed explanation of the chosen queue algorithm and synchronization mechanisms, as well as the work-stealing approach used to achieve dynamic load balancing, along with formal arguments for their correctness and performance.

2 Concurrent Queue Algorithm and Synchronization

2.1 Two-Lock Concurrent Queue Algorithm

The `WorkBalancerQueue` is implemented as a queue adhering to the FIFO (First-In-First-Out) policy. It utilizes a **doubly linked list** with a **dummy node** at the head to enable the separation of head and tail operations. Each core manages its queue, which stores tasks assigned to it, and can allow other cores to fetch tasks for load balancing.

```
1   typedef struct WorkBalancerQueue {
2       WBQNode* head;
3       WBQNode* tail;
4       pthread_mutex_t head_lock, tail_lock;
5       atomic_int q_size; // Number of tasks
6   } WorkBalancerQueue;
7
8   typedef struct WBQNode {
9       Task* task;
10      WBQNode* next;
11      WBQNode* prev;
12  } WBQNode;
```

Listing 1: `WorkBalancerQueue` and `WBQNode` structs

This design is inspired by the **Michael and Scott's two-lock concurrent queue algorithm**[MS96], which separates the head and tail operations using two distinct mutex locks (`head_lock` and `tail_lock`). Listing 1 shows these two locks in `WorkBalancerQueue` struct.

This separation is allowed by a clever addition of a **"dummy" node** to the beginning of the queue. The dummy node pointed by the head pointer, ensures that enqueue and dequeue operations can proceed independently, reducing contention between threads and improving concurrency compared to single-lock queue implementations.

2.1.1 Queue Operations

```
/*
* Enqueue a task to the tail end
* Only callible by the owner thread
* Increment q_size
*/
void submitTask(WorkBalancerQueue* q, Task* _task) {
    WBQNode* newNode = malloc(sizeof(WBQNode));
    assert(newNode != NULL); // Make sure malloc didn't fail
    newNode->task= _task;
    newNode->next = newNode->prev = NULL;

    pthread_mutex_lock(&q->tail_lock);
    // Link tail with newNode
    newNode->prev = q->tail;
    q->tail->next = newNode;
    q->tail = newNode;  // Update tail to the last node (newNode)
    atomic_fetch_add(&q->q_size, 1);
    pthread_mutex_unlock(&q->tail_lock);
}
```

Listing 2: `submitTask` procedure

```
/*
* Dequeue a task from HEAD end
* Can be only called by the owner thread
* When the WBQ is empty it returns NULL.
* Decrement q_size
*/
Task* fetchTask(WorkBalancerQueue* q) {
    pthread_mutex_lock(&q->head_lock);
    // If WBQ is logically empty
    if (q->head->next == NULL){
        pthread_mutex_unlock(&q->head_lock);
        return NULL;
    }

    WBQNode* dummy = q->head;
    WBQNode* new_head = dummy->next;
    Task* fetchedTask = new_head->task;

    q->head = new_head;                 // Update head
    q->head->prev = NULL;               // Update new dummy's prev
    atomic_fetch_sub(&q->q_size, 1);    // Decrement queue size
    pthread_mutex_unlock(&q->head_lock);

    free(dummy);                        // Free the old dummy
    dummy = NULL;                       // Nullify its pointer
    return fetchedTask;
}
```

Listing 3: `fetchTask` function

```
    /*
    * Dequeue a task from TAIL end
    * This method will not be called by the owner thread
    * When the WBQ is empty it returns NULL.
    */
    Task* fetchTaskFromOthers(WorkBalancerQueue* q) {
        pthread_mutex_lock(&q->tail_lock);
        pthread_mutex_lock(&q->head_lock);
        // If the queue is logically empty
        if(q->head->next == NULL){
            pthread_mutex_unlock(&q->head_lock);
            pthread_mutex_unlock(&q->tail_lock);
            return NULL;
        }

        WBQNode* to_be_fetched= q->tail;
        Task* fetchedTask = to_be_fetched->task;
        q->tail = q->tail->prev;                // Update tail to its
    prev
        q->tail->next = NULL;                   // Update new tail's
    next to NULL

        atomic_fetch_sub(&q->q_size, 1);
        pthread_mutex_unlock(&q->head_lock);
        pthread_mutex_unlock(&q->tail_lock);

        free(to_be_fetched);                    // Free the old dummy
        to_be_fetched = NULL;                   // Nullify itspointer
        return fetchedTask;
    }
```

Listing 4: `fetchTaskFromOthers function`

2.2 Evaluating the Two-Lock Queue

2.2.1 Advantages of Two-Lock Queue

The two-lock algorithm used in the `WorkBalancerQueue` implementation offers a significant advantage over single-lock queue algorithms by reducing contention between threads. In particular, separating enqueue and dequeue operations into distinct critical sections allows concurrent access to the head and tail, enabling higher throughput on multi-threaded systems. As observed by Michael and Scott in their research [MS96], the two-lock algorithm demonstrates better performance compared to single-lock queues, especially in systems with more than five active processors. This makes it a reasonable choice for dedicated multi-core systems without advanced atomic primitives like compare-and-swap (CAS). While the two-lock approach may introduce some contention under heavy loads, it remains an **effective balance between simplicity and performance** for this application.

2.2.2 Disadvantages of Two-Lock Queue

While the two-lock queue improves performance over single-lock queues, it is less efficient than **lock-free algorithms**, which are non-blocking in terms of performance, due to high cost of mutex locks.

- **Higher Latency Under Contention:** When many threads simultaneously access the same `WorkBalancerQueue`, threads may block on the locks, leading to delays.
- **Limited Scalability:** As the number of threads grows, contention on the locks increases, which can degrade performance compared to lock-free approaches.

2.2.3 Why Not Lock-Free?

Lock-free algorithms, such as those based on compare-and-swap (CAS), are highly efficient and scalable because they eliminate blocking. However, I opted for the two-lock approach instead of a lock-free algorithm for the following reasons:

- **Complexity:** Lock-free algorithms introduce significant complexity in implementation and debugging. Managing the atomic operations required for lock-free algorithms is error-prone and challenging, particularly for a doubly linked list structure.
- **Predictability:** The two-lock queue provides predictable performance and is easier to reason about in terms of correctness. Lock-free algorithms may suffer from livelock under high contention, where threads repeatedly fail their CAS operations.
- **Suitability for the Assignment:** While performance is a consideration, the focus of this assignment is on correctness and understanding synchronization mechanisms. The two-lock approach strikes a balance between simplicity and performance, making it a suitable choice.

2.3 Ensuring Synchronization: Two Mutex Locks

To ensure thread safety, `WorkBalancerQueue` uses two mutex locks of type `pthread_mutex_t` from the POSIX thread library `<pthread.h>`:

- `head_lock`**:** Protects access to the head pointer and is used during dequeue operations via `fetchTask`.
- `tail_lock`**:** Protects access to the tail pointer and is used during enqueue operations via `submitTask` and tail-end fetching via `fetchTaskFromOthers`.

This separation ensures that concurrent access to different parts of the queue does not cause data corruption or race conditions.

2.3.1 Task Insertion Synchronization

The `submitTask` 2 function ensures thread-safe insertion of a task at the tail of the queue by employing `tail_lock` to synchronize access to the tail pointer. Specifically, `tail_lock` is acquired before the pointers of the new node are updated, and the tail pointer is safely moved to the new node. The critical section is kept minimal, covering only the pointer updates, which reduces contention.

The `q_size` variable, which tracks the number of tasks, is updated atomically using `atomic_fetch_add` and `atomic_fetch_sub` functions, avoiding the need for an additional lock. This synchronization approach guarantees correctness by preventing race conditions and ensuring that only one thread can modify the tail of the queue at any given time.

2.3.2 Task Fetching Synchronization

The synchronization mechanism for task fetching ensures thread-safe removal of tasks from both the head and tail of the queue.

In `fetchTask` 3, which dequeues tasks from the head, the `head_lock` mutex is used to ensure that only one thread can modify the head pointer at a time. **This prevents race conditions and ensures that for the owner thread of the queue, tasks are removed in FIFO order**. Updates to the `q_size` are performed atomically using `atomic_fetch_add` and `atomic_fetch_sub` to maintain consistent size tracking across concurrent operations. The critical section ends after updating the head pointer and freeing the old dummy node, ensuring minimal contention.

In `fetchTaskFromOthers` 4, **which removes tasks from the tail**, both `tail_lock` and `head_lock` are acquired to synchronize access to the shared structure. This dual-locking ensures consistency between the head and tail, preventing interference with enqueue operations or head-pointer modifications. The tail pointer and its associated links (`prev` and `next`) are safely updated within the critical section. The atomic decrement of `q_size` further ensures consistency in the size, even under concurrent access.

2.3.3 FIFO Order is Preserved

Enqueue operations always append new nodes at the tail, maintaining the order of arrival. Dequeue operations remove nodes from the head performed by the function `fetchTask`, ensuring tasks are processed in the order they were inserted.

One crucial exception is **removing nodes from the tail** which is performed by the function `fetchTaskFromOthers`. This mechanism is a part of the load balancing algorithm, which will explained thoroughly in the next section.

2.3.4 Synchronization Measures Handling Concurrent Accesses

It should be already obvious to you that concurrent reads and writes could lead to race conditions on shared data. There are various issues that can arise from such concurrent access to the same queue by multiple cores. In our case the most crucial issues are: **task loss** or **task duplication** during insertion or deletion.

The synchronization mechanism used in the WorkBalancerQueue API methods ensures that no task is duplicated or lost during concurrent operations by. Let us see this in an possible worst-case scenario:

2.3.5 A Possible Concurrent Access Scenario:

Imagine two threads, T1 and T2, simultaneously accessing the same queue. Thread T1 tries to dequeue a task from the head using `fetchTask`, while thread T2 attempts to fetch a task from the tail using `fetchTaskFromOthers`. Without proper synchronization, the following issues could arise:

- **Race Condition:** Both threads might modify the head or tail pointers at the same time, leading to inconsistent states (e.g., skipping a node or corrupting the queue structure).
- **Duplicate Removal:** Both threads might access the same task and remove it simultaneously, leading to task duplication in their respective queues.

When T1 calls `fetchTask`, it acquires `head_lock`. This ensures that no other thread can modify the head of the queue while T1 is operating. Simultaneously, T2 calls `fetchTaskFromOthers` and acquires `tail_lock` to access the tail. Before modifying the tail, T2 also acquires `head_lock` to ensure consistency between head and tail operations. This double-locking prevents T1 and T2 from making conflicting modifications to the shared structure.

Hence, both operations are **atomic with respect to their critical sections**. For example, if T1 updates the head, it does so fully before releasing `head_lock`, ensuring that any subsequent operation by T2 sees a consistent state.

3 Load Balancing and Performance

3.1 Performance Trade-offs to Consider

In multi-core systems, effective load balancing is crucial for optimizing CPU utilization and achieving high performance. Ideally, tasks should be evenly distributed across cores to avoid idle time and ensure efficient use of the entire CPU. However, a load balance design always involves **significant trade-offs**. After all there is **no free lunch**[Dir24] in computer science.

1. **Cache Affinity and Cold Starts:** Task migrations can degrade performance due to cold starts, where the new core lacks the cached data of the stolen task. By prioritizing stealing from heavily loaded cores and limiting migration frequency, the load balancing algorithm mitigates this issue while still addressing load imbalance.
2. **Dynamic Threshold Updates:** Dynamically adjusting `hw` and `lw` ensures that the strategy adapts to varying workloads, but the overhead of periodic updates must be balanced against performance gains. In the studied load balancing algorithm, the chosen interval for updates, determined empirically, provides a practical trade-off between adaptability and efficiency.
3. **Load Imbalance:** The use of average queue size and deviation ensures a more balanced workload distribution while avoiding excessive task migrations. However, ignoring task durations limits the ability to fully optimize the balance in certain scenarios.
4. **Queue Size Tracking:** Using an atomic integer `q_size` for queue size tracking ensures accurate and thread-safe updates without introducing significant locking overhead. This design simplifies the implementation while maintaining correctness.

3.2 Work-Stealing Approach

The implemented load balancing strategy employs a **work-stealing approach**, where underutilized cores dynamically fetch tasks from overloaded cores to maintain load balance and improve CPU utilization. The design carefully manages performance trade-offs between load balancing and minimizing task migration to preserve cache affinity.

3.2.1 Detecting Overloaded and Underloaded Queues

To identify cores that are overloaded or underutilized, the algorithm uses dynamically updated thresholds `lw` and `hw`:

1. **Low Watermark (`lw`):** If the queue size of a core drops below LW, it is considered underloaded and initiates work stealing.
2. **High Watermark (`hw`):** Cores with queue sizes above HW are considered overloaded and are targeted by underloaded cores for task stealing.

The queue size is tracked using an atomic integer `q_size` in the `WorkBalancerQueue` structure 1. This atomic variable is incremented during enqueue operations (`submitTask`) 2 and decremented during dequeue operations (`fetchTask` 3 and `fetchTaskFromOthers`) 4. The atomic nature of `q_size` ensures thread-safe updates without requiring additional locks, avoiding contention and minimizing overhead during queue size adjustments.

3.2.2 Dynamic Threshold Calculation

The thresholds `hw` and `lw` are updated every n iterations in the main loop of `processJobs`. This interval is controlled by an `iter_count` variable, and the value of n, which is 3, was determined empirically through experiments with various task loads.

The **average queue size** is calculated by `calculateAvgQueueSize` as the total number of tasks divided by the number of cores. The **absolute deviation** of task counts across all cores calculated by `calculateAbsoluteDeviation` is used as an offset to adjust the thresholds:

$$\text{HW} = \text{avg\_queue\_size} + |\text{deviation}|, \quad \text{LW} = \text{avg\_queue\_size} - |\text{deviation}|$$

```
1    double calculateAvgQueueSize() {
2        double total_size = 0;
3        for (int i = 0; i < NUM_CORES; i++) {
4            total_size += atomic_load(&processor_queues[i]->q_size);
5        }
6        return total_size / NUM_CORES;
7    }
8
9    double calculateAbsoluteDeviation(double avg_q_size) {
10       double total_deviation = 0.0;
11       for (int i = 0; i < NUM_CORES; i++) {
12           int queue_size = atomic_load(&processor_queues[i]->q_size);
13           total_deviation += fabs(queue_size - avg_q_size);
14       }
15       return total_deviation / NUM_CORES;
16   }
```

This dynamic calculation ensures thresholds adapt to workload changes, enabling the system to respond efficiently to imbalances.

3.2.3 Role of `fetchTaskFromOthers`

When a core detects that its queue size is below LW, it invokes the `findQueueWithMaxLoad` 5 function to identify the core with the largest queue size. If this identified queue exceeds `hw`, the underloaded core attempts to steal a task using `fetchTaskFromOthers` 4. This function safely removes a task from the tail of the target queue, minimizing disruption to the target core's task order and preserving FIFO processing for its remaining tasks.

3.2.4 Work Stealing Decision

To minimize unnecessary migrations, the stealing core targets the most overloaded queue. This approach prevents random or excessive migrations, reducing the likelihood of cold starts that occur when tasks lose cache affinity. The decision to steal is further gated by the `hw` and `lw` thresholds, ensuring that migrations only occur when there is a meaningful imbalance.

```
1     WorkBalancerQueue* findQueueWithMaxLoad(WorkBalancerQueue* my_queue
    , int my_id) {
2         WorkBalancerQueue* max_load_queue = NULL;
3         int max_size = -1;
4         // Iterate through all queues
5         for (int i = 0; i < NUM_CORES; i++) {
6             if (i == my_id) continue; // Skip own queue
7
8             int queue_size = atomic_load(&processor_queues[i]->q_size);
9             if (queue_size > max_size) {
10                max_size = queue_size;
11                max_load_queue = processor_queues[i];
12            }
13        }
14        return max_load_queue;
15 }
```

Listing 5: `findQueueWithMaxLoad function`

3.2.5 What About Task Duration?

While the current implementation focuses on queue size as the primary metric for determining load, incorporating the total task duration of a queue **could improve the accuracy of load balancing**. A queue with a low task count could still be heavily loaded if its tasks have significantly long durations.

However, including task durations in the calculations would introduce additional computational overhead and **increase the complexity of the load balancing mechanism**. For these reasons, task durations were not included as a work load metric in this design. In more optimized designs, incorporating task durations is recommended to achieve finer-grained load balancing.

3.3 Performance Intuition

The work-stealing design effectively balances the trade-offs between load balancing and task stealing. The dynamic threshold mechanism ensures adaptability, while the stealing strategy used **minimizes unnecessary migrations**, preserving cache affinity as much as possible. The use of **atomic operations for queue size** tracking enhances performance, and while task durations are not currently included, they offer a promising avenue for further optimization in future designs.

4 Conclusion

In this assignment, I implemented a `WorkBalancerQueue` using a doubly linked list with a **two-lock synchronization mechanism**, inspired by Michael and Scott's algorithm. This approach ensures thread safety, prevents task duplication or loss, and maintains FIFO order by using separate locks for the head and tail and atomic operations for queue size tracking.

The load balancing algorithm employs a **dynamic work-stealing strategy**, utilizing adaptive high and low watermarks (`hw` and `lw`) based on average queue size and deviation. This design effectively balances workloads while minimizing unnecessary task migrations to preserve cache affinity. The strategy demonstrated adaptability across varying workloads and maintained system efficiency.

Future improvements include **incorporating task duration as a metric** to refine load balancing decisions and **exploring lock-free queue algorithms** to further enhance performance. Experimentation with alternative migration heuristics could also optimize load balancing in more complex scenarios.

References

[Dir24] Science Direct. *No Free Lunch*. 2024. URL: https://www.sciencedirect.com/topics/computer-science/no-free-lunch.

[MS96] Maged M Michael and Michael L Scott. "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms". In: *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing* (1996), pp. 267–275.