Design and Evaluation of a Thread-Safe Priority Queue with Synchronization

Group 17: JiKe Zhang, Yifu Lin, Huanjia Hong

1 Introduction

This report explores synchronization strategies through the design and implementation of thread-safe priority queues. Priority queues are widely used in scheduling and task management, but ensuring correctness and performance under concurrent access is challenging due to their dynamic ordering behavior.

While standard implementations like Java's **PriorityBlockingQueue** provide thread safety through coarse-grained locking, they often fail to scale efficiently under high contention. To address the limitations of Java's built-in priority queue implementations—particularly their reliance on coarse-grained locking—we propose three custom designs, each targeting a different level of synchronization guarantee while optimizing for concurrency and scalability.

We introduce these designs in order of increasing synchronization strength: **Blocking Multi-Heap**, **Lock-Free Skip List**, and **Wait-Free Bucket Queue**.

- **Blocking Multi-Heap**: The primary bottleneck in the built-in JDK design is the use of a single global lock. We aim to replace this large lock with multiple smaller ones by partitioning the heap into several local heaps, each guarded by its own lock, coordinated by a global read-write lock.
- **Lock-Free Skip List**: After addressing blocking limitations, we consider whether blocking can be entirely avoided. Skip lists, which have been studied as lock-free priority queues since the 1980s, suit CAS-based lock-free techniques well due to their node-based sorted structure.
- **Wait-Free Bucket Queue**: Wait-freedom is difficult to achieve with strict ordering. Inspired by thread-slot wait-free queues, we map fixed priorities to dedicated FIFO buckets. This design suits real-time systems such as game frame scheduling, where priorities are bounded.

After presenting each design in detail, we conduct a series of **correctness and performance evaluations** under different concurrency scenarios. Based on these results, we compare the trade-offs among the three approaches and conclude with our reflections and future improvement directions.

Note: While priority queues provide several auxiliary methods such as `peek`, `remove`, and `size`, these are typically composed of or derived from the core operations **offer(insert)** and **poll(remove the minimum)**. As such, this report focuses exclusively on these two fundamental operations when analyzing and designing concurrent implementations.

2 Synchronization Basics

2.1 Context and Motivation

In multithreaded programs, multiple threads often access shared data concurrently. Without proper coordination, this may lead to *race conditions*, where the result depends on the relative timing of operations.

For example, two threads modifying a shared bank balance without synchronization may overwrite each other's updates and cause data inconsistency. **Synchronization ensures correctness and consistency** by enforcing mutual exclusion and memory visibility rules, preventing such conflicts.

2.2 Synchronization Mechanisms Overview

Synchronization can be achieved through a variety of mechanisms. **Lock-based** approaches, such as mutexes, semaphores, and monitors, provide exclusive access to critical sections. These are simple to implement, but can cause blocking and contention under high concurrency. **Atomic**

and lock-free primitives like Compare-And-Swap (CAS) offer finer-grained control and are widely used in high-performance concurrent structures. Higher-level constructs such as barriers, latches, and phasers help coordinate thread progress in structured parallel computations.

Synchronization mechanisms span **multiple abstraction layers** within a computing system. At the lowest level, the hardware provides atomic instructions such as `CAS` and memory fences to enforce operation ordering. Operating systems add mechanisms like interrupt disabling and scheduler-based synchronization. At the software level, programming languages and libraries offer constructs such as locks, semaphores, and concurrent collections.

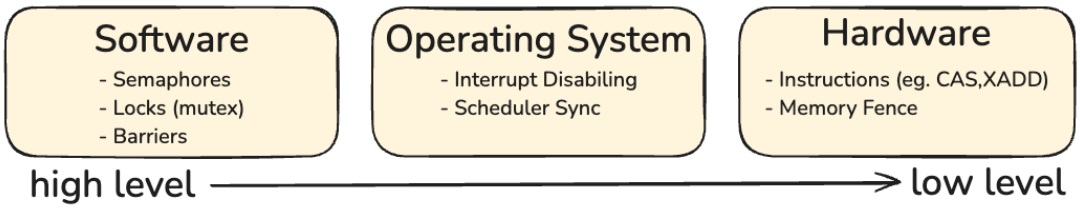

Figure 1: Synchronization mechanisms exist at all levels: from software to hardware.

A clear understanding of these layers is essential for designing scalable and reliable concurrent systems.

2.3 Progress Conditions

Progress conditions describe the guarantees a concurrent system provides about operation completion under contention. They help classify synchronization strategies based on how they ensure forward progress.

Blocking operations rely on locks; a thread may be delayed indefinitely if another thread holds the required resource. While easy to reason about, blocking can lead to issues like **deadlock or starvation** under contention.

Lock-free algorithms ensure that **at least one thread** in the system will complete its operation in a **finite** number of steps, even if others are stalled. This improves system-wide responsiveness but may still allow **individual thread starvation**.

Wait-free algorithms provide the strongest guarantee: every operation by **every thread** is guaranteed to complete in a **bounded** number of steps, regardless of other threads' behavior. This eliminates starvation but is more complex and costly to implement.

Understanding progress conditions is essential when designing scalable concurrent data structures, as they directly impact throughput, latency, and fairness.

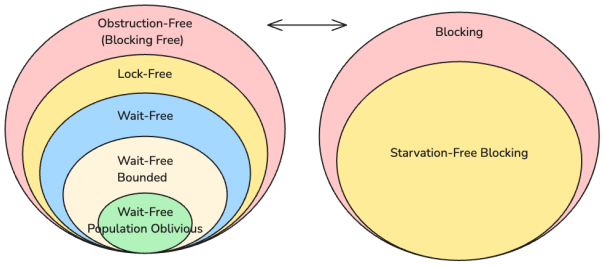

Figure 2: Classification of progress conditions from blocking to non-blocking.

3 Concurrent Priority Queues: Motivation and Fundamentals

3.1 Priority Queues and Their Importance

A priority queue is a data structure that retrieves elements based on priority rather than insertion order, unlike FIFO queues. It is central to many systems such as OS schedulers, event queues, and real-time task dispatchers. In this course alone, several projects rely on priority-based task management, from Dijkstra's algorithm to coroutine scheduling. Internally, most implementations use a binary heap to maintain a **sorted structure**, where operations

like **offer** and **poll** require restructuring through sift-up and sift-down. These operations are relatively costly and become more challenging under concurrent access, motivating the need for efficient, thread-safe designs.

3.2 Thread-Safe Priority Queues

In multithreaded environments, standard priority queues must be extended with synchronization to prevent race conditions that can corrupt the heap and break its ordering invariant. Java's built-in solution is the **PriorityBlockingQueue**, which maintains thread safety by protecting all operations with a single global **ReentrantLock**.

While this implementation is robust and widely used, it relies on **coarse-grained** locking, making all `offer` and `poll` operations mutually exclusive. This design ensures correctness but significantly limits concurrency under contention, motivating the need for more scalable alternatives with finer synchronization granularity or non-blocking designs.

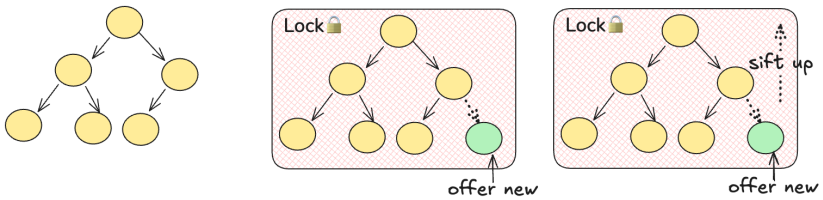

Figure 3: Offer operation with global lock in PriorityBlockingQueue.

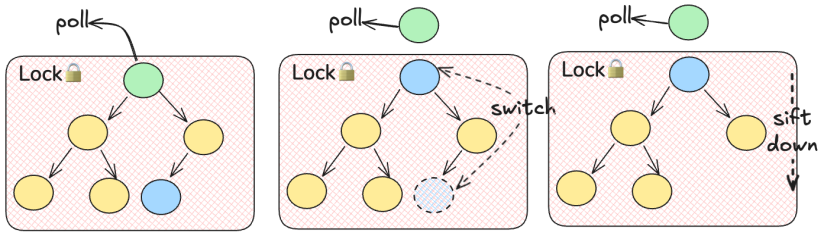

Figure 4: Poll operation with global lock in PriorityBlockingQueue.

4 Blocking Multi-Heap Priority Queue

To reduce contention caused by a single global lock in traditional blocking designs, we propose a multi-heap structure that enables concurrent access to independent segments.

4.1 Design Overview

The core idea is to split the priority queue into k **independent segment heaps**. Each segment is protected by its own `ReentrantLock`, and a global `ReadWriteLock` coordinates overall access. We use three levels of locking:

- **Global Read Lock (RLock)**: Acquired by `offer` operations. Multiple offers can proceed concurrently, but `poll` is blocked because ongoing insertions may affect the global minimum, making it unsafe to determine the true smallest element.
- **Global Write Lock (WLock)**: Temporarily acquired by `poll` to scan all segment roots. It blocks both offers and polls to ensure that the minimum is selected from a consistent snapshot—offers might insert a new minimum, and concurrent polls might remove it.
- **Segment Lock**: Each segment heap has its own `ReentrantLock`, required for insertion or removal within that heap. This prevents race conditions on local heap state and ensures safe structural modifications (e.g., sift-up, sift-down).

To **insert** an element, the thread first acquires the global read lock (RLock), then randomly selects a segment heap and acquires its `ReentrantLock`. It inserts the element, performs a local sift-up to restore heap order, then releases the segment lock followed by the global read lock. Since all offers only require shared read access and operate on separate heaps, multiple insertions can proceed concurrently.

To **remove the minimum** element, the poll operation first acquires the global write lock (`WLock`) to scan the root nodes of all segment heaps and identify the smallest one.To proceed safely, the thread first acquires the global read lock and then the lock for the target segment heap. Only after both locks are held does it release the global write lock, ensuring that the selected minimum remains valid. It then removes the root from the segment, performs a sift-down to restore heap order, and finally releases both the segment lock and the global read lock. This **lock-switching** approach ensures correctness while minimizing blocking, though only one `poll` may proceed at a time.

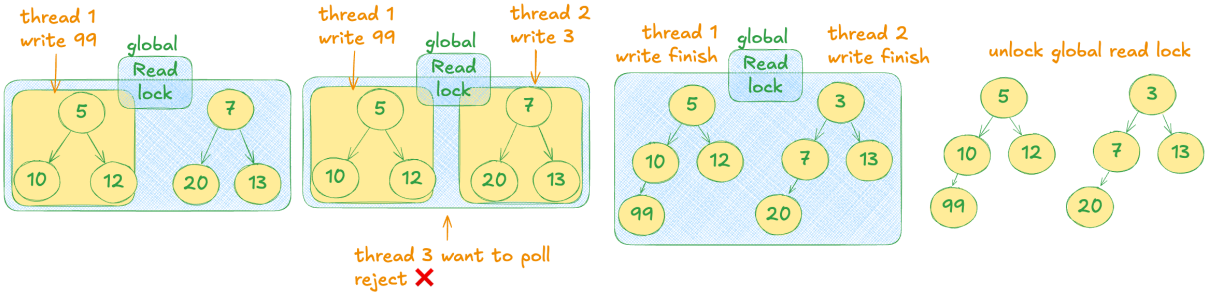

Figure 5: Multi-heap `offer`: shared global RLock with localized segment locks.

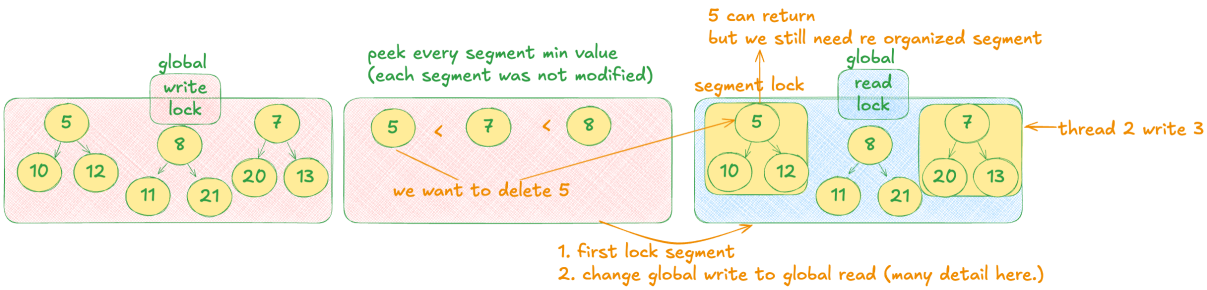

Figure 6: Multi-heap `poll`: WLock for root scan, then RLock and segment lock for removal.

4.2 Design Alternatives and Discussion

An improvement we considered is organizing heaps by key range instead of assigning them randomly. This would make `offer` operations deterministic based on the element's priority. Inspired by **B-tree and B+ tree** structures, segment heaps could dynamically split and merge to maintain a balanced load.

However, this introduces trade-offs. Reads (`poll`) would concentrate on the heap that holds the smallest range, making it a hotspot. In contrast, if keys are uniformly distributed, `offer` and `poll` traffic could naturally spread across segments. **Balancing efficiency and fairness under skewed workloads remains a challenge**.

5 Lock-Free Skip List Priority Queue

5.1 Why Skip List?

As discussed earlier, our goal is to improve concurrency by eliminating blocking. Skip lists are well-suited for this due to their sorted, **node-based layout**, which supports fast traversal and enables lock-free updates using Compare-And-Swap (CAS).

A skip list maintains multiple layers of linked lists. The bottom layer holds all elements in sorted order, while upper layers serve as **shortcuts for faster search**, enabling expected $O(\log n)$ operations. Most importantly, even if one thread fails or is delayed, others can still make progress—making skip lists ideal for high-concurrency environments.

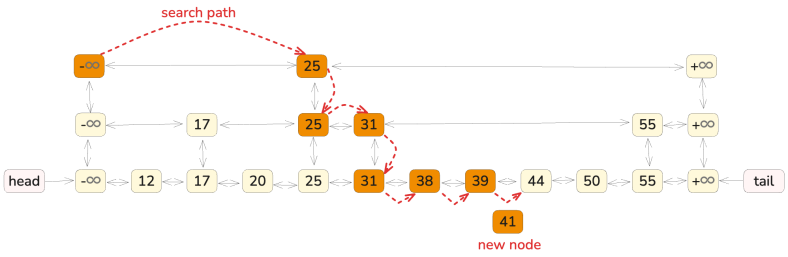

Figure 7: Skip list structure and search path during insertion.

5.2 Design Overview

Our skip list design is inspired by prior work [1]. Each node contains a key, value, and an array of forward pointers for each level it appears in. Nodes are inserted or removed using CAS on the forward pointers. To support safe removal without blocking, we use **logical deletion**: nodes are first marked as deleted, then physically unlinked by helping threads.

To **insert** a new element, a thread traverses from the top level to the bottom to locate the correct position between a predecessor and a successor node. It then uses CAS to link the new node by updating the predecessor's forward pointer. If the CAS fails due to interference, the entire traversal restarts from the head node. This approach ensures correctness but can be expensive under contention. In practice, more efficient strategies such as retrying from a higher level or caching index-layer paths could reduce overhead—offering promising directions for optimization. Nevertheless, the operation remains non-blocking: even if one thread fails or retries repeatedly, others can still make progress.

To retrieve and **remove the minimum** element, the thread scans the bottom level from the head node, skipping logically deleted nodes until it finds a valid one. It then uses CAS to mark the node as deleted and unlinks it logically. Physical unlinking may be assisted by future traversals. This ensures that removal is safe even if multiple threads race to delete the same node.

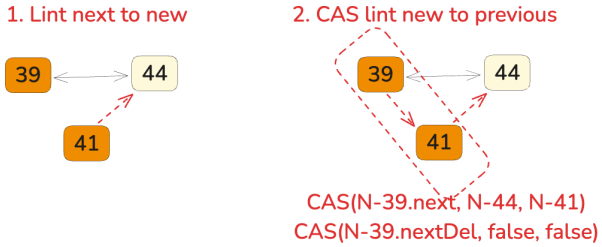

Figure 8: CAS-based insertion in skip list. Offer may fail if concurrent modification occurs.

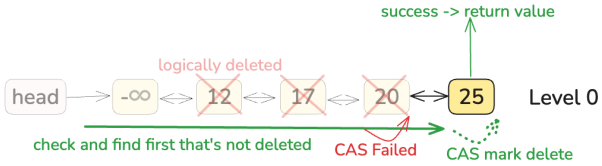

Figure 9: Skip list poll scans for the first non-deleted node. CAS failures may occur during logical deletion.

5.3 Offer-Poll Conflicts

Since `offer` and `poll` may operate on adjacent nodes concurrently, interference can occur. For example, one thread may be inserting a new node just before the node another thread is trying to delete. In such cases, CAS may fail, causing a retry. However, lock-free designs ensure system-wide progress: even if one thread is delayed, others continue to make progress.This coordination via CAS and retry loops enables high-throughput, fine-grained concurrency without global locks.

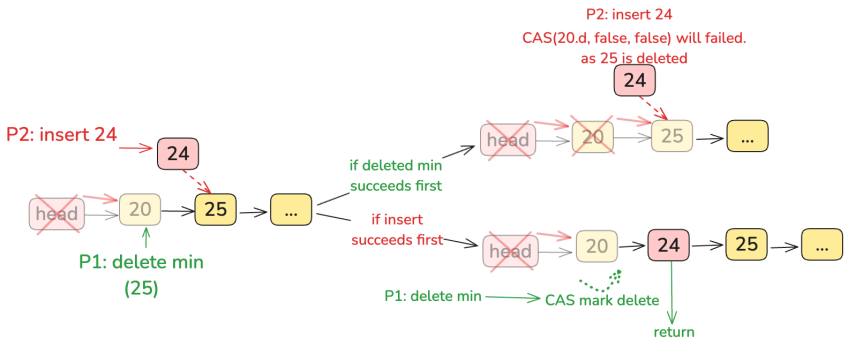

Figure 10: Conflict between `offer` and `poll` in a lock-free skip list.

5.4 Design Discussion

Our current implementation relies heavily on **AtomicReference** wrappers to manage node pointers across multiple levels. While CAS-based updates are lock-free and efficient, **excessive use** of atomic references increases memory overhead and leads to a large number of CAS operations—each of which, though fast individually, incurs noticeable cumulative cost under contention. Optimizing the memory layout or reducing unnecessary indirection could improve performance.

Another critical factor is how the height of each node (i.e., its **index level**) is determined. While basic skip list designs often use uniform random assignment, more effective strategies such as exponential decay-based level selection have shown to significantly improve performance. In our experiments, adopting such strategies improved throughput by 2–4× in typical workloads, particularly under high concurrency.

6 Wait-Free Bucket Priority Queue

6.1 Design Overview

To provide strong progress guarantees, we design a wait-free priority queue using a bucket-based structure. Each bucket represents a **fixed priority level** and maintains its own wait-free list. The number of buckets is bounded and predefined, making this design suitable for real-time or game-like systems where priorities are known and limited. And our wait-free list follows the approach in [2].

To facilitate efficient polling, we maintain a global atomic pointer `currentMin` to track the lowest non-empty bucket. This reduces scanning overhead during `poll` operations.

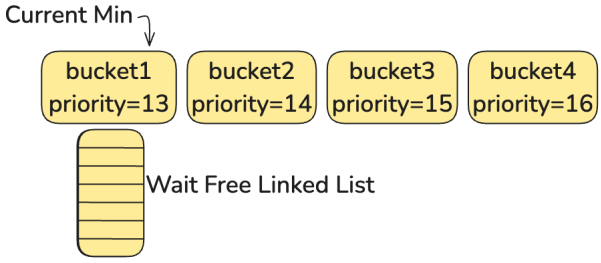

Figure 11: Bucket-based priority queue: fixed-priority queues with wait-free list.

6.2 Offer and Poll

Inserting an element involves computing its priority and inserting it into the corresponding bucket using a wait-free enqueue. This operation is strictly bounded in steps, and the thread does not block or spin on others.

Polling proceeds by scanning from the bucket indicated by `currentMin` upward until a non-empty queue is found. The thread attempts a wait-free dequeue. If successful, `currentMin`

is updated accordingly. Threads may also assist pending operations in the queue they scan, preserving wait-freedom across threads.

6.3 Wait-Free List Internals

Each bucket is implemented as a wait-free FIFO queue based on a linked list. Operations are split into multiple steps so that the **helping mechanism** can be realized: only the owner thread publishes the initial descriptor and declares success, while any other thread may assist in completing those steps.

Threads cooperate via per-thread **operation descriptors** (OpDesc) atomic reference array to guarantee wait-freedom. Each descriptor is assigned a **phase number** at the moment a thread initiates its operation. Phase numbers are drawn from a single global atomic counter, guaranteeing a strict ordering in which every new operation receives a larger phase number than any earlier one.

Whenever a thread begins an operation, it creates and publishes its descriptor into the shared array. Before proceeding with its own work, it scans that array and helps any pending descriptors with smaller phase numbers complete. During execution, CAS operations are used to update pointers, mark logical deletions, and commit results. A CAS failure simply indicates another thread has already performed the change, so it does not block progress. Additionally, versioned pointers are employed to prevent ABA issues.

Note: The wait-free list queue used in each bucket is based on publicly available code; we implemented the outer integration logic.

7 Correctness Evaluation and Benchmark

7.1 Test Design

Our evaluation includes both correctness testing and benchmark analysis. **Correctness** is validated by ensuring that the priority queue maintains its ordering invariant under both **sequential** and **concurrent** `offer` and `poll` operations. This is divided into two parts: `PriorityQueueTest` verifies single-threaded behavior, while `ThreadSafePriorityQueueTest` focuses on correctness under concurrency. The **benchmark** evaluates throughput (operations per millisecond) across different workloads and access patterns.

7.2 Test Setup

Correctness tests are implemented using JUnit, covering both single-threaded and multi-threaded scenarios. Benchmark tests are implemented using JMH (Java Microbenchmark Harness) to evaluate throughput under varying **concurrency levels** and **read/write ratios**. The workload includes both sequential and randomly ordered integer input. We use three testing modes:

- **Read-Only (poll)**: Multiple threads perform `poll` with thread counts from 1 to 128.
- **Write-Only (offer)**: Multiple threads perform `offer` with thread counts from 1 to 128.
- **Mixed Read/Write**: Threads are split to simulate 90% read / 10% write, 70/30, 50/50, and 0/90 access patterns. For example, a 9:1 thread split simulates 90% polling and 10% insertion.

Each benchmark group is repeated across different queue implementations (e.g., Blocking Multi-Heap, Lock-Free Skip List, Wait-Free Bucket Queue), and load levels (e.g., `LIGHT`, `MEDIUM`, `HEAVY`).

7.3 Results Summary

The figure above presents the throughput (ops/ms, log scale) of each queue under four operation patterns: **add, poll, poll90add10, and add90poll10**. The benchmarks were conducted on a MacBook Pro with Apple M4 chip using JMH. In the add test, the queue starts empty and is continuously fed with randomly generated integers. The poll test preloads the queue with 100,000 random integers and repeatedly performs polling operations, refilling the queue once it's

depleted. The poll90add10 and add90poll10 tests simulate mixed workloads, with the former being 90% polling and 10% adding, and the latter the reverse.

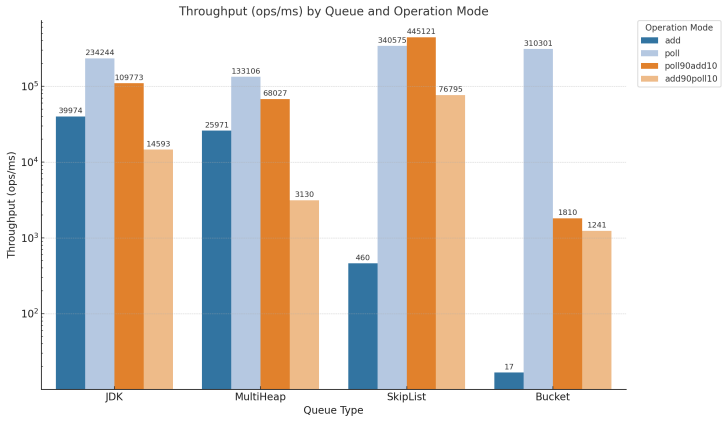

Figure 12: Throughput comparison under varying thread counts and read/write ratios.

While **blocking-based** designs suffer from inherent limitations such as contention and lack of scalability, they offer simple, stable implementations that are well-suited to most general-purpose scenarios. In contrast, **lock-free** structures require significantly more coordination logic to preserve correctness, making them harder to implement and tune. Their performance is also sensitive to contention due to frequent CAS retries. Finally, although **wait-free** designs provide the strongest theoretical guarantees, they are extremely difficult to implement in a generic setting, and their practical performance often lags behind due to structural and coordination overhead.

Appendix: Table of Contributions

Name	Contribution
JiKe Zhang (ajhz632)	Led the overall project design and implementation. Responsible for the Lock-Free Skip List, synchronization background, benchmark framework and evaluation, as well as the overall structure planning of the report and presentation. Also created all diagrams.
Yifu Lin (lyif004)	Implemented the Blocking Multi-Heap priority queue and contributed the related content in the report and slides.
Huanjia Hong (hhon333)	Implemented the Wait-Free Bucket Queue and contributed the corresponding section in the report and presentation.

Table 1: Group member contributions.

Note: Due to space constraints, we omit many low-level implementation details and full data structure listings. Instead, we focus on design ideas and logical reasoning throughout the report. For specific structures and code, please refer to the appendix and our GitHub repository.

References

[1] Håkan Sundell and Philippas Tsigas. A skiplist-based concurrent priority queue with minimal memory contention. In *Euro-Par 2013*, pages 236–247, 2013.

[2] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked lists. *SIGPLAN Notices*, 47(8):309–310, 2012.