CH10 SUMMARY

COS 226 Chapter 10: Concurrent Queues and the ABA Problem

Comprehensive Summary with Explanations

---

1. Introduction to Pools

What are Pools?

Pools are similar to the Set data structures covered in Chapter 9, but with key differences:

- **No contains() method requirement**: Unlike sets, pools don't necessarily need to tell you if an item exists in them
- **Duplicate items allowed**: The same item can appear multiple times in a pool, whereas sets typically only store unique elements

Think of a pool like a collection bin where you can throw items in and pull items out, without worrying about whether duplicates exist or checking if something specific is already there.

---

2. Queue Classification by Capacity

Bounded Queues

- **Fixed capacity**: The queue has a maximum number of items it can hold
- **Use case**: Useful when system resources (like memory) are limited
- **Example**: Like a checkout line that can only hold 10 customers at a time

Unbounded Queues

- **Unlimited capacity**: Can hold any number of objects (limited only by available memory)
- **Use case**: When you don't want to restrict the number of items
- **Example**: Like an online order queue that accepts orders as long as the server has memory

---

3. Method Behavior Classifications

Total Methods

A method is **total** if it never waits for conditions to become true. It responds immediately, even if unsuccessfully.

Example: When you try to dequeue from an empty queue, instead of waiting, the method immediately throws an exception or returns a failure code saying "sorry, queue is empty."

Partial Methods

A method is **partial** if it may wait for certain conditions to hold before proceeding.

Example: A thread trying to dequeue from an empty queue will block (suspend its execution) and wait until another thread enqueues an item. Only then will it wake up and complete the dequeue operation.

Synchronous Methods

A method is **synchronous** if it waits for another method to overlap with its execution interval.

Example: When you call enqueue, it won't complete until another thread calls dequeue to receive that item. Both operations must happen simultaneously—like a handshake. This is used in languages like CSP and Ada for threads to "rendezvous" and exchange values directly.

---

4. Blocking vs Non-Blocking Behavior

Problem scenarios:

- Removing from an empty pool
- Adding to a full bounded pool

Blocking Approach

- The caller waits (blocks) until the state changes
- Example: A thread trying to enqueue into a full queue will wait until space becomes available

Non-Blocking Approach

- The method throws an exception immediately
- Example: A thread trying to enqueue into a full queue gets an exception right away and can decide what to do

---

5. Concurrency Potential in Bounded Queues

The Key Insight: Queues have natural concurrency potential because enqueue and dequeue operations work at opposite ends:

- **enq()** operates at the **tail** (back of the queue)
- **deq()** operates at the **head** (front of the queue)

This means an enqueue and dequeue call should be able to proceed concurrently without interfering with each other—like two people working on opposite ends of a line of boxes.

However: Multiple concurrent enqueue calls will interfere with each other (they all need to modify the tail), and multiple concurrent dequeue calls will interfere with each other (they all need to modify the head).

The Challenge: What happens when the queue is empty or full? This requires careful synchronization.

---

6. Bounded Lock-Based Queue Implementation

Structure

The queue is implemented as a **linked list** with:

- **head** field: Points to the first node
- **tail** field: Points to the last node
- **Sentinel node**: A placeholder node that doesn't contain actual data but simplifies implementation

Why a sentinel? It ensures that even an empty queue has at least one node, eliminating special cases in the code.

Two-Lock Design

The implementation uses **two distinct locks** instead of one:

1. **deqLock**: Locks the front of the queue
    - Ensures only one thread is dequeuing at a time
    - Prevents multiple dequeuers from interfering
2. **enqLock**: Locks the end of the queue
    - Ensures only one thread is enqueuing at a time
    - Prevents multiple enqueuers from interfering

Why two locks? Using two separate locks means enqueuers and dequeuers can operate **independently** and don't unnecessarily block each other. This increases concurrency and performance.

Tracking Queue State

Since the queue is bounded, we need to track how full it is:

- **size field**: An `AtomicInteger` that tracks the current number of objects in the queue
- **capacity field**: The maximum number of items the queue can hold

Why AtomicInteger? Because both enqueuers and dequeuers need to modify the size, and it needs to be thread-safe without additional locking.

Condition Variables

Each lock has an associated **condition variable** for thread coordination:

1. **notFullCondition** (associated with enqLock):
   - Notifies waiting enqueuers when the queue is no longer full
   - An enqueuer waits on this condition if the queue is at capacity
2. **notEmptyCondition** (associated with deqLock):
   - Notifies waiting dequeuers when the queue is no longer empty
   - A dequeuer waits on this condition if the queue is empty

---

7. Bounded Queue Operations in Detail

Enqueue Operation Flow

1. **Lock the enqLock** to ensure exclusive access to the tail
2. **Check if queue is full**: If `size == capacity`, wait on `notFullCondition`
3. **Create and add new node** to the end of the list
4. **Increment size** using `getAndIncrement()`
5. **If size was 0** (queue was empty), set flag to wake dequeuers
6. **Unlock enqLock**
7. **If flag set**, signal all waiting dequeuers via `notEmptyCondition.signalAll()`

Why signal after unlocking? To avoid holding the lock while waking other threads, which could cause unnecessary contention.

Dequeue Operation Flow

1. **Lock the deqLock** to ensure exclusive access to the head

2. **Check if queue is empty**: If `size == 0`, wait on `notEmptyCondition`
3. **Read value** from the first actual node (head.next)
4. **Make first node the new sentinel** by advancing head pointer
5. **Decrement size** using `getAndDecrement()`
6. **If size was at capacity**, set flag to wake enqueuers
7. **Unlock deqLock**
8. **If flag set**, signal all waiting enqueuers via `notFullCondition.signalAll()`
9. **Return the dequeued value**

---

8. Optimization: Splitting the Counter

The Bottleneck: The shared `AtomicInteger size` is accessed by every enqueue and dequeue operation, creating a potential performance bottleneck.

Solution: Split into two counters:

1. **enqSideSize**: Decremented by deq(), checked by enq()
   - enq() only cares if this value equals capacity (queue full)
2. **deqSideSize**: Incremented by enq(), checked by deq()
   - deq() only cares if this value equals zero (queue empty)

This reduces contention because enqueuers and dequeuers now touch different memory locations most of the time.

---

9. Unbounded Queue Implementation

Simplifications

Since the queue can hold unlimited items:

- **No capacity tracking needed**
- **enq() always succeeds** (never waits for space)
- **deq() throws EmptyException** if queue is empty (no waiting)
- **Same linked-list representation** as bounded queue
- **Still uses two locks** (enqLock and deqLock) for concurrency

Unbounded enq()

```
1. Lock enqLock
2. Create new node
```

```
3. Append to tail
4. Update tail pointer
5. Unlock enqLock
```

Unbounded deq()

```
1. Lock deqLock
2. Check if queue is empty (head.next == null)
3. If empty, throw EmptyException
4. Read value from first node
5. Advance head pointer
6. Unlock deqLock
7. Return value
```

---

10. Lock-Free Queue Introduction

Goal: Prevent starvation by having faster threads help slower threads, without using locks.

Key Mechanism: Use **Compare-And-Set (CAS)** operations instead of locks.

Node Structure for Lock-Free Queue

Each node now has:

- **value**: The data stored in the node
- **next**: An `AtomicReference<Node>` (not just a regular reference)

The `AtomicReference` allows atomic CAS operations on the next pointer, which is essential for lock-free synchronization.

---

11. Lock-Free Enqueue: The Lazy Approach

Enqueue is split into two distinct, **non-atomic** steps:

1. Logical Enqueue

Appends the new node to the linked list by CAS-ing the last node's next field.

2. Physical Enqueue

Updates the queue's tail field to point to the newly added node.

Critical Insight: Between these two steps, a new node is reachable by other threads before tail actually points to it. This means **tail can lag behind**, pointing to either:

- The actual last node, or
- The second-to-last node (penultimate node)

Helping Mechanism

When a thread wants to enqueue:

1. It checks if tail has a successor (if `tail.next != null`)
2. If yes, **another enqueue is in progress**, so this thread "helps" by CAS-ing tail forward
3. Only after tail is correct does it attempt its own enqueue

This "helping" ensures progress even if a thread gets delayed mid-operation.

Detailed Lock-Free enq() Logic

```
1. Create new node
2. Loop until successful:
   a. Read current tail and tail.next
   b. Verify tail hasn't changed
   c. If tail.next is null (tail is correct):
      - Try CAS to append new node to tail.next (logical enqueue)
      - If successful, try CAS to update tail (physical enqueue)
      - Return (don't wait for physical enqueue to succeed)
   d. If tail.next is not null (tail is lagging):
      - Help by CAS-ing tail to tail.next
      - Loop and try again
```

---

12. Lock-Free Dequeue

Key Considerations

- Must check if queue is empty
- Without a counter, queue is empty when **head == tail == sentinel**
- Must handle the case where tail is lagging

Detailed Lock-Free deq() Logic

```
1. Loop until successful:
   a. Read head, tail, and head.next
   b. Verify head hasn't changed
   c. If head == tail (queue appears empty):
      - If head.next is null, queue is truly empty → throw EmptyException
      - If head.next is not null, tail is lagging → help by advancing tail
   d. If head != tail (queue not empty):
      - Read value from head.next
      - Try CAS to advance head to head.next
      - If successful, return value
      - If CAS fails, another thread dequeued first → retry
```

---

13. Memory Reuse and the ABA Problem

The Memory Management Challenge

After dequeuing nodes, what happens to them?

- With garbage collection: Language runtime automatically reclaims memory
- Without garbage collection: We need to manually manage memory

Simple Solution: Free-Lists

Each thread maintains a **private free-list** of unused nodes:

- **On enqueue**: Get a node from the free-list (or allocate a new one if empty)
- **On dequeue**: Add the removed node back to the free-list for reuse

This seems efficient, but introduces a serious problem...

---

14. The Dreaded ABA Problem

Problem Description

When nodes are recycled (removed from queue, then added back later), we get the **ABA problem**:

Scenario:

1. Thread A reads that head points to node X (with value "A")
2. Thread A is about to CAS head from X to X.next (node Y)
3. **Thread A gets delayed** (context switch, interrupt, etc.)
4. Thread B dequeues X (value "A")
5. Thread B dequeues Y
6. Thread B enqueues a new node
7. Thread B **recycles node X** and enqueues it again

8. Now X points to a completely different next node than before
9. **Thread A wakes up** and sees head still points to X (value "A")
10. Thread A's CAS succeeds because the pointer matches!
11. **But** the meaning of that pointer has changed—X.next is different now
12. The queue is now corrupted, with head pointing to a node that's not actually in the queue

Why It's Called ABA

The value at a memory location changes from **A** → **B** → back to **A**, and the observer can't tell that anything changed by just looking at the pointer.

---

15. ABA Problem Solutions

Solution 1: Tagged Pointers

Concept: Tag each pointer with a **counter** that increments with each modification.

How it works:

- Instead of just storing a pointer, store a (pointer, counter) pair
- Every time you modify the pointer, increment the counter
- CAS compares both pointer AND counter
- Even if pointer returns to same value (A → B → A), counter will be different

Implementation: Java provides `AtomicStampedReference` class for this purpose.

Challenges:

- Pointer size vs word size issues on some architectures
- Counter overflow? (Usually not a practical concern, or use bounded tags with careful analysis)

Solution 2: Hazard Pointers (not detailed in slides)

Another approach is to use hazard pointers to indicate which nodes are currently being accessed, preventing premature recycling.

---

16. Dual Data Structures

Motivation: Reduce synchronization overhead in synchronous queues by splitting operations into two stages.

Core Concept

At any given time, the queue contains:

- **Enqueue reservations** (threads waiting to give items), OR
- **Dequeue reservations** (threads waiting to receive items), OR
- Is empty

Never both types at once!

Two-Stage Operation

Every operation (enq or deq) happens in two stages:

1. **Reservation**: Put a placeholder in the queue indicating you're waiting
2. **Fulfillment**: Another thread completes your operation by filling your reservation

---

17. Dual Queue: Dequeue Operation

When Queue is Empty

If a dequeuer tries to remove from an empty queue:

1. **Create a reservation object** containing:
    - A flag to signal completion
    - Space for the item value
2. **Add reservation to queue**
3. **Spin on the flag** waiting for fulfillment
4. When enqueuer fulfills it, flag changes and dequeue completes

When Queue Contains Items

If queue contains actual items (not reservations):

1. Remove the first item normally
2. Return it

---

18. Dual Queue: Enqueue Operation

When Queue is Empty or Contains Items

If queue is empty or contains items (not reservations):

1. Add the item normally to the queue
2. Return

When Queue Contains Dequeue Reservations

If queue contains dequeue reservations:

1. Take the first reservation
2. **Fulfill it** by depositing your item into the reservation object
3. **Set the flag** to notify the waiting dequeuer
4. Return

Alternative: Enqueue Can Also Wait

An enqueue can create its own reservation and wait for a dequeue to fulfill it:

1. Create enqueue reservation with your item
2. Add to queue
3. Spin on flag until dequeuer takes it

---

19. Dual Queue Node Structure

```
enum NodeType { ITEM, RESERVATION }

class Node {
    volatile NodeType type;
    volatile AtomicReference<T> item;
    volatile AtomicReference<Node> next;
}
```

Fields:

- **type**: Is this an actual item or a reservation?
- **item**: The value (or null for reservations, filled on fulfillment)
- **next**: Pointer to next node
- **volatile**: Ensures visibility across threads

---

20. Dual Queue Enqueue Implementation Details

Case 1: Queue Empty or Contains Items

```
1. Create new ITEM node
2. If queue is empty or tail is an ITEM:
   a. Add node to tail (standard enqueue)
   b. Spin until item is taken (item field changes)
   c. Once taken, help remove the node from queue
   d. Return
```

Spin-wait: `while (offer.item.get() == e) {}`

- Continues spinning as long as item field still contains original value
- When dequeuer fulfills reservation, item field changes
- This "busy waiting" is acceptable for short waits in lock-free algorithms

Case 2: Queue Contains Dequeue Reservations

```
1. Get first reservation (head.next)
2. Verify nothing changed
3. CAS reservation's item from null to your value (fulfill it)
4. Advance head to remove fulfilled reservation
5. Return if successful
```

---

21. Dual Queue Benefits

Advantages

1. **Locally cached spinning**: Waiting threads spin on their own reservation's flag, which stays in their cache—much faster than spinning on shared memory locations
2. **Fairness**: Reservations are fulfilled in FIFO order, ensuring no thread waits indefinitely
3. **Linearizable**: Operations appear to take effect atomically at a single point in time, maintaining strong consistency guarantees
4. **Reduced contention**: By having threads wait on different memory locations (their own reservations), we reduce cache coherence traffic

---

22. Key Takeaways

Bounded Queues

- Use two locks for better concurrency
- Require careful coordination with condition variables
- Splitting counters can reduce contention

Unbounded Queues

- Simpler than bounded (no capacity management)
- Still benefit from two-lock design
- Can be implemented lock-free with CAS

Lock-Free Queues

- Provide progress guarantees without locks
- Use helping mechanisms for robustness
- Require careful attention to memory reuse

ABA Problem

- Critical issue when recycling memory in lock-free algorithms
- Solved with tagged pointers or other techniques
- `AtomicStampedReference` provides Java implementation

Dual Data Structures

- Novel approach for synchronous communication
- Reduces synchronization overhead through two-stage operations
- Provides good locality and fairness properties

---

23. Performance Implications

Lock-based approaches:

- Simpler to reason about
- Can have contention bottlenecks
- Good when lock hold times are short

Lock-free approaches:

- More complex implementation
- Better worst-case latency (no waiting for lock holders)
- Helping mechanisms ensure system-wide progress
- Require more careful memory management

Dual structures:

- Excellent for producer-consumer scenarios
- Reduce unnecessary synchronization

- Trade computation (spinning) for reduced coordination overhead