CH7 SUMMARY

COS 226 Chapter 7: Spin Locks and Contention - Comprehensive Summary

1. Introduction and Concurrency Issues

What Are Concurrency Issues?

In multiprocessor systems, several fundamental problems arise when multiple threads try to work together:

Memory Contention: Not all processors can access the same memory location simultaneously. When multiple processors try to access the same memory, they must queue up and wait their turn. Think of this like multiple people trying to use the same ATM machine - only one can use it at a time, and others must wait in line.

Communication Medium Contention: If multiple processors want to communicate at the same time (like sending data over a shared bus), some will have to wait. This is similar to a conference call where everyone tries to speak at once - the system can only handle one speaker at a time.

Communication Latency: It takes time for processors to communicate with memory or with each other. This delay can significantly impact performance, especially when processors are waiting for data or responses.

2. New Goals and Focus Areas

Traditional concurrent programming focused primarily on **correctness** (ensuring the program works properly) and **progress** (ensuring threads don't get stuck forever). However, this chapter introduces additional critical concerns:

- **Performance**: Understanding how different lock implementations affect system speed
- **Architecture Awareness**: Recognizing how the underlying hardware design impacts performance
- **Practical Implementation**: Moving from theoretical solutions to real-world, high-performance implementations

3. Lock Acquisition Strategies

What to Do When You Can't Get a Lock

When a thread tries to acquire a lock that's already taken, it has two main options:

Spinning (Busy-Waiting):

- The thread continuously checks if the lock is available
- Good for short delays because the thread stays active and can immediately grab the lock when it becomes free
- Bad for long delays because it wastes CPU cycles

Giving Up the Processor:

- The thread suspends itself and lets the scheduler run other threads
- Good for long delays because it doesn't waste CPU time
- Always good on single-processor systems (no point in spinning when no other processor can release the lock)
- Bad for short delays because of the overhead of suspending and resuming threads

4. Problems with Classical Lock Algorithms

Why Filter and Bakery Locks Don't Work Well

Space Complexity Problem: These algorithms require reading and writing n distinct memory locations, where n is the number of concurrent threads. This means:

- Memory usage grows linearly with the number of threads
- More cache misses and memory traffic
- Poor scalability for systems with many threads

Peterson Lock Issues

The Peterson lock, while theoretically correct, fails in practice due to **memory ordering** problems:

The Core Problem: Modern processors and compilers don't guarantee that memory operations happen in the order they appear in your code. For example:

```
flag[i] = true;     // This might execute
victim = i;         // After this in actual hardware
```

Why This Happens:

- **Compiler Optimizations:** Compilers reorder instructions to improve performance

- **Hardware Optimizations:** Processors may execute instructions out of order or buffer writes

Memory Barriers/Fences: Special instructions that force outstanding operations to complete before proceeding. However, they're expensive and require careful programmer intervention.

5. Test-and-Set Operations

Understanding Test-and-Set (TAS)

What It Does: Test-and-Set is an atomic operation that:

1. Reads the current value of a variable
2. Sets the variable to a new value (usually true)
3. Returns the old value

Implementation in Java:

```
public synchronized boolean getAndSet(boolean newValue) {
    boolean prior = value;
    value = newValue;
    return prior;
}
```

Lock Usage:

- Lock is free when value is false
- Lock is taken when value is true
- To acquire: call getAndSet(true)
  - If it returns false, you got the lock
  - If it returns true, someone else has it

TAS Lock Implementation

```
class TASlock {
    AtomicBoolean state = new AtomicBoolean(false);

    void lock() {
        while (state.getAndSet(true)) {}  // Spin until we get the lock
    }

    void unlock() {
        state.set(false);  // Release the lock
```

```
10        }
11    }
```

6. Understanding Cache Coherence and Bus Architecture

Bus-Based Multiprocessor Systems

Components:

- **Shared Bus:** A communication medium that connects all processors and memory
- **Per-Processor Caches:** Small, fast memory close to each processor (1-2 cycles vs 10s of cycles for main memory)
- **Main Memory:** Larger, slower shared memory

Cache Terminology:

- **Cache Hit:** Finding needed data in the local cache (fast, good)
- **Cache Miss:** Having to fetch data from main memory (slow, bad)

Cache Coherence Problem

When multiple processors cache the same memory location, problems arise:

1. Processor A reads a variable (caches it)
2. Processor B reads the same variable (also caches it)
3. Processor A modifies its cached copy
4. Now there are inconsistent copies of the data

Write-Back Cache Coherence Protocol

Cache Entry States:

- **Invalid:** Contains meaningless data
- **Valid:** Contains correct data, but read-only (others might have copies)
- **Dirty:** Data has been modified locally, and this is the authoritative copy

How It Works:

1. When a processor wants to write to a cached location, it must invalidate all other cached copies
2. Other processors lose their cached copies and must fetch new data from memory

3. This creates significant bus traffic

7. Why TAS Locks Perform Poorly

The Performance Problem

TAS locks perform much worse than expected due to cache coherence overhead:

1. **Constant Bus Traffic:** Every getAndSet() call requires bus access
2. **Cache Invalidation:** Each attempt invalidates all other processors' cached copies
3. **Memory Delays:** Other processors must wait for bus access to reload their caches
4. **Lock Holder Delays:** Even the thread releasing the lock may be delayed by bus congestion from spinners

Result: Performance degrades significantly as more threads compete for the lock, often performing worse than single-threaded code.

8. Test-and-Test-and-Set (TTAS) Locks

The Two-Phase Approach

TTAS locks use a two-stage strategy:

Lurking Stage:

- Repeatedly read the lock variable (using regular read, not getAndSet)
- Wait until the lock "looks" free (reads false)
- This uses cached copies, generating no bus traffic

Pouncing Stage:

- As soon as the lock looks free, call getAndSet to try to acquire it
- If successful, enter critical section
- If unsuccessful, return to lurking stage

TTAS Implementation

```
class TTASlock {
    AtomicBoolean state = new AtomicBoolean(false);

    void lock() {
        while (true) {
```

```
            while (state.get()) {}          // Lurk: wait until lock looks free
            if (!state.getAndSet(true))     // Pounce: try to acquire
                return;                     // Success!
        }
    }

    void unlock() {
        state.set(false);
    }
}
```

Why TTAS Is Better (But Still Has Problems)

Advantages:

- **Local Spinning:** While waiting, threads read from their local cache (no bus traffic)
- **Reduced Contention:** Only generates bus traffic when the lock appears available

Remaining Problem - Thundering Herd:
When the lock is released:

1. All spinners' cached copies are invalidated
2. All threads fetch the new value and see the lock is free
3. All threads simultaneously call getAndSet()
4. This creates a "storm" of bus traffic
5. Only one thread wins, others must start over

9. Exponential Backoff

The Contention Inference Strategy

Key Insight: If a thread sees the lock is free but fails to acquire it, this indicates high contention. The thread should wait before trying again.

Exponential Backoff Strategy:

1. Start with a minimum delay
2. When acquisition fails, wait for a random time
3. Double the maximum delay after each failure (up to a limit)
4. Use random delays to avoid synchronized retries

Backoff Lock Implementation

```
public class Backoff implements Lock {
    public void lock() {
        int delay = MIN_DELAY;
        while (true) {
            while (state.get()) {}              // Wait until lock looks
free
            if (!state.getAndSet(true))         // Try to acquire
                return;                         // Success!
            sleep(random() % delay);            // Backoff randomly
            if (delay < MAX_DELAY)
                delay = 2 * delay;              // Exponential increase
        }
    }
}
```

Backoff Limitations

Good:

- Easy to implement
- Better performance than TTAS
- Works well with proper tuning

Bad:

- **Parameter Sensitivity:** Performance depends heavily on MIN_DELAY and MAX_DELAY values
- **Platform Dependence:** Optimal parameters vary by hardware and number of processors
- **Still Has Cache Coherence Issues:** All threads still spin on the same location
- **Critical Section Underutilization:** Threads may delay longer than necessary

10. Queue-Based Locks

The Core Idea

Instead of having all threads spin on the same location, create a queue where:

- Each thread spins on a different memory location
- Threads are notified in order
- No wasted invalidations
- First-come-first-served fairness

Anderson Queue Lock

Data Structure:

- Array of boolean flags (one per possible thread)
- Atomic counter tracking the tail of the queue
- Each thread gets assigned a unique slot

How It Works:

1. Thread calls getAndIncrement on tail counter to get its slot number
2. Thread spins on its assigned flag in the array
3. When lock holder finishes, it sets the next thread's flag to true
4. Each thread notifies exactly one successor

Anderson Lock Implementation

```
class ALock implements Lock {
    boolean[] flags = {true, false, false, ...};  // First slot starts true
    AtomicInteger tail = new AtomicInteger(0);
    ThreadLocal<Integer> mySlot;

    public void lock() {
        mySlot = tail.getAndIncrement();           // Get my slot number
        while (!flags[mySlot]) {}                  // Spin on my slot
    }

    public void unlock() {
        flags[mySlot] = false;                     // Reset my slot
        flags[(mySlot + 1) % SIZE] = true;         // Signal next thread
    }
}
```

Advantages:

- **Scalable Performance:** Each thread spins on its own cache line
- **True FIFO Fairness:** Threads acquire lock in arrival order
- **Fast Handover:** Direct notification between threads

Disadvantages:

- **Space Inefficiency:** Requires one flag per potential thread
- **Unknown Thread Count:** Hard to size the array appropriately
- **Wasteful for Low Contention:** Most array slots unused when few threads contend

11. CLH (Craig, Landin, Hagersten) Queue Lock

Virtual Linked List Concept

CLH creates an implicit linked list using atomic references:

- Each thread has a node indicating its status
- Nodes form a queue through atomic swapping
- Each thread spins on its predecessor's node

Node States:

- **true:** Thread wants the lock or has acquired it
- **false:** Thread has released the lock

CLH Implementation

```
class CLHLock implements Lock {
    AtomicReference<Qnode> tail = new AtomicReference<>(new Qnode());
    ThreadLocal<Qnode> myNode = new ThreadLocal<Qnode>() {
        protected Qnode initialValue() { return new Qnode(); }
    };

    public void lock() {
        Qnode qnode = myNode.get();
        qnode.locked = true;                        // Indicate I want the lock
        Qnode pred = tail.getAndSet(qnode);         // Add myself to queue
        while (pred.locked) {}                      // Spin on predecessor
    }

    public void unlock() {
        Qnode qnode = myNode.get();
        qnode.locked = false;                       // Release lock
        myNode.set(pred);                           // Recycle predecessor's node
    }
}
```

Advantages:

- **Space Efficient:** Only requires nodes for actual contending threads
- **Local Spinning:** Each thread spins on a different memory location
- **Implicit Queue:** No explicit next pointers needed

12. MCS (Mellor-Crummey, Scott) Queue Lock

Explicit Linked List Approach

Unlike CLH's implicit list, MCS uses explicit next pointers:

- Each node has a "next" field pointing to the successor
- More complex unlock protocol but handles some edge cases better

MCS Implementation

```
class MCSLock implements Lock {
    AtomicReference<Qnode> tail = new AtomicReference<>(null);

    public void lock() {
        Qnode qnode = new Qnode();
        Qnode pred = tail.getAndSet(qnode);        // Add to queue
        if (pred != null) {                        // If queue wasn't
empty
            qnode.locked = true;
            pred.next = qnode;                     // Link predecessor to
me
            while (qnode.locked) {}                // Spin on my own node
        }
    }

    public void unlock() {
        if (qnode.next == null) {                  // No apparent
successor?
            if (tail.compareAndSet(qnode, null))   // Try to reset tail
                return;                            // Successfully removed
from queue
            while (qnode.next == null) {}          // Wait for successor
to link
        }
        qnode.next.locked = false;                 // Notify successor
    }
}
```

Key Differences from CLH:

- **Explicit Links:** Uses next pointers instead of implicit queue
- **Complex Unlock:** Must handle race conditions between checking for successors and tail updates
- **Local Spinning:** Each thread spins on its own node (like CLH)

13. Timeout and Abortable Locks

The Need for Timeouts

Real applications often need the ability to:

- Give up waiting after a maximum time
- Handle transaction aborts
- Implement deadlock recovery

Challenges with Queue Locks

Simple Locks (Backoff): Easy to abort - just return from the lock() method

Queue Locks: More complex because:

- Threads form a chain of dependencies
- Simply leaving breaks the chain
- Successor threads might wait forever (starvation)

Timeout Lock (TOLock) Approach

Strategy: When a thread abandons its position:

1. Mark the node as "abandoned"
2. Let the successor detect this and skip over the abandoned node
3. The successor then waits on the abandoned node's predecessor

Implementation Highlights:

- Uses a special AVAILABLE node to signal a free lock
- Threads spin on predecessor's "prev" field instead of the node itself
- When timing out, threads try to clean up gracefully
- Complex protocol handles race conditions between timeout and normal operation

14. Composite Locks

Combining the Best of Both Worlds

Problem: Different lock types have complementary strengths:

- **Backoff Locks:** Easy timeout, but not scalable
- **Queue Locks:** Scalable and fair, but complex timeout protocols

Solution: Composite locks use a hybrid approach:

1. Maintain a small array of queue nodes
2. Threads randomly select nodes (with backoff if busy)
3. Once a node is acquired, use queue-lock protocol
4. Combine scalability of queuing with timeout simplicity

Composite Lock Architecture

Components:

- **Waiting Array:** Fixed-size array of queue nodes
- **Queue:** Nodes form a queue when active
- **States:** FREE, WAITING, RELEASED, ABORTED

Three-Phase Protocol:

1. **Node Acquisition:** Randomly select and acquire a node from the array
2. **Queue Joining:** Splice the acquired node into the tail of the queue
3. **Waiting:** Wait for predecessor nodes to be released

Node States and Transitions

FREE → WAITING: Thread successfully acquires a node
WAITING → RELEASED: Thread finishes critical section normally
WAITING → ABORTED: Thread times out while waiting
RELEASED/ABORTED → FREE: Node is recycled for reuse

15. Performance Analysis and Trade-offs

Lock Performance Characteristics

TAS Lock:

- Simple implementation
- Terrible scalability due to bus contention
- Performance degrades with more threads

TTAS Lock:

- Better than TAS due to local spinning
- Still suffers from thundering herd problem

- Moderate improvement but not scalable

Backoff Lock:

- Good performance with proper tuning
- Parameter-sensitive and platform-dependent
- Reasonable for moderate contention

Queue Locks (Anderson, CLH, MCS):

- Excellent scalability
- True FIFO fairness
- Consistent performance across thread counts
- More complex implementation

Composite Lock:

- Balances scalability with timeout simplicity
- Good performance across various contention levels
- Most complex implementation

When to Use Each Lock Type

Low Contention: Simple locks like TTAS with backoff may suffice
High Contention: Queue locks provide better scalability and fairness
Timeout Requirements: Backoff or composite locks handle timeouts more easily
Fairness Critical: Queue locks guarantee FIFO ordering
Space Constrained: CLH and MCS use less space than Anderson lock

16. Key Concepts and Takeaways

Memory Architecture Matters

Understanding cache coherence and bus contention is crucial for lock performance. Algorithms that work well in theory may perform poorly due to hardware characteristics.

Local Spinning Principle

Having threads spin on different memory locations (each in their own cache) dramatically reduces bus contention and improves scalability.

Fairness vs Performance Trade-offs

Queue locks provide fairness but with implementation complexity. Sometimes unfair locks perform better under low contention.

Timeout Handling Complexity

Simple locks make timeout handling trivial, while queue locks require complex protocols to handle abandoned nodes gracefully.

No Universal Solution

Different applications and hardware platforms may benefit from different lock implementations. The "best" lock depends on expected contention levels, fairness requirements, timeout needs, and performance characteristics of the target platform.

This comprehensive overview demonstrates the evolution from simple but poorly-performing locks to sophisticated, scalable solutions that handle real-world requirements like timeouts and fairness while maintaining good performance across varying contention levels.