CH3 SUMMARY

Complete Guide to Concurrent Objects - COS 226 Chapter 3

Table of Contents

1. Introduction to Objects and Concurrency
2. Sequential vs Concurrent Objects
3. Correctness Properties
4. Progress Properties
5. Implementation Examples
6. Java Memory Model
7. Summary and Best Practices

---

1. Introduction to Objects and Concurrency [#introduction]

What Are Objects?

In object-oriented programming, an **object** is essentially a container that holds data and provides controlled ways to manipulate that data. Think of it like a box with:

- **State**: The current condition or contents of the object (stored in fields/variables)
- **Methods**: The only allowed ways to interact with and modify the state

Example - Queue Object:

- **State**: A sequence of items waiting to be processed (like people in a line)
- **Methods**:
  - `enqueue (enq)` : Add an item to the back of the line
  - `dequeue (deq)` : Remove and return the item at the front of the line

The Challenge: What About Concurrent Objects?

When multiple threads (independent processes) try to use the same object simultaneously, we face new challenges:

1. **How do we describe what the object should do?** (Specification)
2. **How do we build one that works correctly?** (Implementation)
3. **How do we verify it's working properly?** (Verification)

Safety vs Liveness Properties

In concurrent programming, we need to ensure two types of properties:

- **Safety (Correctness)**: "Nothing bad happens"
  - The object behaves correctly even when multiple threads use it
  - Data doesn't get corrupted or lost
- **Liveness (Progress)**: "Something good eventually happens"
  - Threads don't get stuck waiting forever
  - The system makes progress and completes work

Sequential vs Concurrent Objects {#sequential-vs-concurrent}

Sequential Objects - The Simple Case

In single-threaded programs, objects are straightforward to understand and verify:

Pre and Post Conditions:

- **Precondition**: What must be true before calling a method
- **Postcondition**: What will be true after the method completes

Example - Dequeue Method:

```
Scenario 1:
Precondition: Queue is non-empty
Postconditions:
- Returns the first item in queue
- Removes that first item from queue

Scenario 2:
Precondition: Queue is empty
Postconditions:
- Throws EmptyException
- Queue state remains unchanged
```

Why Sequential is Easy:

- Methods are called one at a time
- Object state is meaningful between method calls
- Each method can be described in isolation
- Adding new methods doesn't affect existing descriptions

Concurrent Objects - The Complex Reality

Methods Take Time:
In concurrent systems, method calls aren't instantaneous events - they're **intervals** with:

- **Invocation**: When the method call starts
- **Response**: When the method call completes
- **Pending**: A method that has started but not yet finished

The Overlap Problem:

```
Time ------>
Thread A: |--enq(x)--|
Thread B:    |--deq()--|
Thread C:      |--enq(y)--|
```

Multiple method calls can overlap in time, creating complex interactions.

Key Differences from Sequential:

1. **State Management**: Object might never be "between method calls" since calls overlap
2. **Method Interactions**: Must handle all possible combinations of overlapping calls
3. **Complexity**: Everything can potentially interact with everything else

---

Correctness Properties {#correctness-properties}

When multiple threads access an object simultaneously, how do we define "correct" behavior? Three main approaches have been developed:

1. Quiescent Consistency

Core Principles:

- **Principle 3.3.1**: Method calls should appear to happen in sequential (one-at-a-time) order

- **Principle 3.3.2**: Method calls separated by periods of quiescence should appear in real-time order

What is Quiescence?
An object is **quiescent** when it has no pending method calls - essentially, it's temporarily inactive with no ongoing operations.

Real-World Analogy:
Imagine a busy restaurant kitchen. During peak hours, multiple orders are being prepared simultaneously (concurrent operations). But during quiet periods between rushes (quiescence), any new orders that come in should be processed in the order they arrive.

Example - Shared Counter:

```
Scenario: Two threads concurrently write -3 and 7 to a register
Later: Another thread reads the register
✓ Correct: Returns either -3 or 7
✗ Incorrect: Returns -7 (mixture of values)
```

Key Properties:

- **Compositional**: If each object in a system is quiescent consistent, the entire system is quiescent consistent
- **Flexible**: Doesn't require strict program order for concurrent operations
- **Practical**: Good for high-performance applications with relaxed consistency requirements

2. Sequential Consistency

Core Principles:

- **Principle 3.3.1**: Method calls should appear in sequential order (same as quiescent)
- **Principle 3.4.1**: Method calls should take effect in **program order**

What is Program Order?
The order in which a single thread issues method calls. Operations from different threads have no program order relationship.

Example Problem:

```
Single thread writes 7, then writes -3 to a register
Later the same thread reads and gets 7
This violates sequential consistency because it didn't get the last
written value (-3)
```

Concurrent Example:

```
Thread A: enq(x) then deq(y)
Thread B: enq(y) then deq(x)

Valid sequential orders:
1. A.enq(x) → B.enq(y) → B.deq(x) → A.deq(y)
2. B.enq(y) → A.enq(x) → A.deq(y) → B.deq(x)

Both preserve program order within each thread.
```

Important Limitation:

- **Not Compositional**: A system of sequentially consistent objects may not itself be sequentially consistent

3. Linearizability

Core Principle:

- **Principle 3.5.1**: Each method call should appear to take effect instantaneously at some moment between its invocation and response

The Gold Standard:
Linearizability is the strongest and most intuitive correctness property. Every linearizable execution is also sequentially consistent, but not vice versa.

Linearization Points:
To prove an object is linearizable, you must identify for each method call a specific **linearization point** - the exact moment when the method's effects become visible to other threads.

For Different Implementation Types:

- **Lock-based implementations**: Usually within the critical section
- **Lock-free implementations**: The single atomic step where changes become visible

Why Linearizability Matters:

- **Intuitive**: Matches our expectation that operations happen "atomically"
- **Compositional**: Systems built from linearizable components are linearizable
- **Modular**: Perfect for building large systems from smaller components

Real-World Analogy:
Think of linearizability like a busy bank with multiple tellers. Even though customers are being served simultaneously at different windows, each transaction appears to happen at a

single instant in time. You never see half-completed transactions or inconsistent account balances.

Comparing the Three Properties

Property	Strength	Use Case	Compositional
Quiescent Consistency	Weakest	High-performance systems	Yes
Sequential Consistency	Medium	Hardware memory models	No
Linearizability	Strongest	Component-based systems	Yes

Progress Properties {#progress-properties}

While correctness properties ensure safety, progress properties ensure **liveness** - that the system actually makes forward progress and doesn't get stuck.

Blocking vs Non-Blocking

Blocking Implementations:

- Delay of one thread can delay other threads
- Typically use locks or other mutual exclusion mechanisms
- Risk of deadlock, priority inversion, and convoy effects

Non-Blocking Implementations:

- Delay of one thread cannot delay other threads
- Use atomic operations and lock-free techniques
- Immune to deadlock but may have other complexities

Progress Condition Hierarchy

From Weakest to Strongest:

1. Deadlock-Free

- **Guarantee**: Some thread trying to acquire a lock will eventually succeed
- **Limitation**: Other threads might wait indefinitely (starvation possible)
- **Example**: A system where high-priority threads always get the lock first

2. Starvation-Free

- **Guarantee**: Every thread trying to acquire a lock will eventually succeed
- **Improvement**: No thread waits forever
- **Example**: Fair locks that serve threads in arrival order

3. Lock-Free

- **Guarantee**: Some thread calling a method will eventually complete and return
- **Key Insight**: System-wide progress guaranteed, but individual threads might be delayed
- **Example**: Compare-and-swap based algorithms where one thread always succeeds

4. Wait-Free (Strongest)

- **Guarantee**: Every thread calling a method will complete in a finite number of steps
- **Ultimate Goal**: Per-thread progress guaranteed
- **Bounded Wait-Free**: There's a known upper limit on execution steps

Visual Progress Hierarchy

```
Wait-free ⊆ Lock-free ⊆ Starvation-free ⊆ Deadlock-free

Everyone makes progress ← → Someone makes progress
    (Non-blocking)              (Blocking)
```

Real-World Analogies

Lock-Free (Restaurant Kitchen):
During a busy dinner rush, at least one dish always gets completed and served, even if some orders take longer due to ingredient availability or complexity.

Wait-Free (Emergency Room):
Every patient is guaranteed to be seen within a maximum time limit, regardless of how busy the emergency room gets.

Implementation Examples {#implementation-examples}

Lock-Based Queue Implementation

This is a traditional approach using mutual exclusion to ensure thread safety:

```
class LockBasedQueue<T> {
    int head, tail;           // Array indices
    T[] items;                // Storage array
    Lock lock;                // Single shared lock

    public LockBasedQueue(int capacity) {
        head = 0; tail = 0;   // Initially empty
        lock = new ReentrantLock();
        items = (T[]) new Object[capacity];
    }

    public void enq(T x) throws FullException {
        lock.lock();          // Acquire exclusive access
        try {
            if (tail - head == items.length)
                throw new FullException();
            items[tail] = x;
            tail++;
        } finally {
            lock.unlock();    // Always release lock
        }
    }

    public T deq() throws EmptyException {
        lock.lock();
        try {
            if (tail == head)
                throw new EmptyException();
            T x = items[head];
            head++;
            return x;
        } finally {
            lock.unlock();
        }
    }
}
```

Key Properties:

- **Correctness**: Linearizable (linearization points within critical sections)
- **Progress**: Starvation-free (assuming fair locks)
- **Performance**: Limited by lock contention

Wait-Free Two-Thread Queue

This specialized implementation works without locks for exactly two threads:

```
public class WaitFreeQueue {
    int head = 0, tail = 0;
    items = (T[]) new Object[capacity];

    // Only one thread calls this
    public void enq(Item x) {
        while (tail-head == capacity); // Busy-wait if full
        items[tail % capacity] = x;
        tail++;
    }

    // Only the other thread calls this
    public Item deq() {
        while (tail == head);          // Busy-wait if empty
        Item item = items[head % capacity];
        head++;
        return item;
    }
}
```

Key Properties:

- **Correctness**: Linearizable (with execution-dependent linearization points)
- **Progress**: Wait-free for both threads
- **Limitation**: Only works with exactly two threads (one producer, one consumer)

Linearization Points (Execution-Dependent):

- **Successful deq()**: When `head` is incremented
- **Empty deq()**: When the method throws an exception
- **enq()**: When `tail` is incremented

Java Memory Model {#java-memory-model}

The Java Memory Model defines how threads interact through memory, and understanding it is crucial for writing correct concurrent programs.

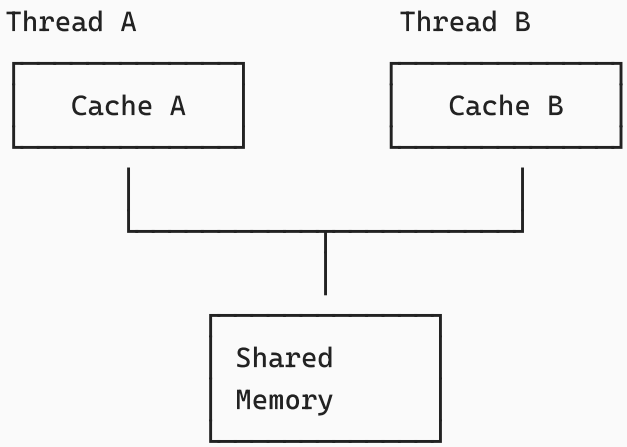
The Problem: Compiler and Hardware Optimizations

Key Issue: Java does not guarantee linearizability for simple field reads and writes due to optimizations.

Why Optimizations Cause Problems:

- Compilers may reorder instructions for performance
- CPUs may execute instructions out of order
- Caches may delay writes to main memory

Thread Memory Architecture



Memory Model Issues:

- Threads cache field values locally
- Changes might not immediately propagate to shared memory
- Reads might return stale cached values instead of current shared values

Classic Example: Double-Checked Locking

The infamous Singleton pattern implementation that seems correct but isn't:

```
public class Singleton {
    private static Singleton instance;

    // BROKEN - Don't use this!
    public static Singleton getInstance() {
        if (instance == null) {              // First check
            synchronized(Singleton.class) {
                if (instance == null)        // Double check
                    instance = new Singleton();
            }
        }
        return instance;
    }
}
```

Why It Breaks:

1. `new Singleton()` involves multiple steps:
   - Allocate memory
   - Initialize object
   - Assign to `instance` field
2. The Java Memory Model allows these steps to be reordered
3. Another thread might see a non-null `instance` that points to uninitialized memory

Synchronization Mechanisms

Java provides several ways to ensure proper memory synchronization:

1. Locks and Synchronized Blocks

Memory Effects:

- **Acquiring a lock**: Invalidates thread's cached values, forces fresh reads from shared memory
- **Releasing a lock**: Writes all changes back to shared memory immediately

Guarantee: If all accesses to a field are protected by the same lock, reads and writes to that field are linearizable.

2. Volatile Fields

```
private volatile boolean flag = false;
```

Memory Effects:

- **Reading volatile field**: Like acquiring a lock - forces fresh read from shared memory
- **Writing volatile field**: Like releasing a lock - immediately writes to shared memory

Limitation: Only individual operations are atomic, not sequences of operations.

3. Final Fields

```
class Example {
    final int x;    // Cannot change after constructor
    int y;          // Regular field

    public Example() {
```

```
6           x = 3;      // Safe initialization
7           y = 4;      // May not be visible to other threads immediately
8       }
9   }
```

Guarantee: Any thread that sees a reference to an object is guaranteed to see the correctly initialized values of all final fields.

Summary of Memory Model Rules

Fields are linearizable if:

1. The field is declared `volatile`, OR
2. The field is protected by a unique lock used by all readers and writers

Without proper synchronization:

- Writes may not be immediately visible to other threads
- Reads may return stale cached values
- Operations may appear to execute out of order

Summary and Best Practices {#summary}

Correctness Properties Quick Reference

Property	When to Use	Key Benefit
Quiescent Consistency	High-performance systems where some reordering is acceptable	Compositional, flexible
Sequential Consistency	Low-level systems, hardware interfaces	Intuitive program order
Linearizability	Component-based systems, general-purpose concurrent objects	Strongest guarantees, compositional

Progress Properties Quick Reference

Property	Guarantee	Best For
Deadlock-free	Some thread makes progress	Basic correctness
Starvation-free	All threads make progress eventually	Fair resource allocation

Property	Guarantee	Best For
Lock-free	System-wide progress guaranteed	High-performance, avoid blocking
Wait-free	Per-thread progress guaranteed	Real-time systems, critical applications

Implementation Strategy Guidelines

Choose Lock-Based When:

- Correctness is more important than maximum performance
- Complex operations that don't map well to atomic primitives
- Existing codebase already uses locks extensively
- Team has more experience with lock-based programming

Choose Lock-Free When:

- Performance is critical
- Want to avoid deadlock and priority inversion
- Have expertise in atomic operations and memory models
- Can accept the complexity of lock-free algorithms

Best Practices

1. Design Phase

- **Start with the specification**: What should the object do?
- **Choose appropriate correctness property**: Usually linearizability for general use
- **Consider progress requirements**: Real-time vs. best-effort systems

2. Implementation Phase

- **Use proven patterns**: Don't invent new synchronization mechanisms
- **Identify linearization points clearly**: Document where operations take effect
- **Handle the Java Memory Model**: Use volatile, synchronized, or explicit locks

3. Testing and Verification

- **Test with multiple threads**: Sequential tests aren't sufficient

- **Use stress testing**: High contention reveals race conditions
- **Consider formal verification**: For critical systems

4. Common Pitfalls to Avoid

- **Double-checked locking**: Usually broken without volatile
- **Assuming atomic operations**: Most operations aren't atomic in Java
- **Ignoring memory model**: Field updates may not be visible across threads
- **Lock-free without expertise**: Lock-free programming is much harder than it appears

Final Thoughts

Concurrent programming is inherently complex, but understanding the fundamental concepts of correctness and progress properties provides a solid foundation. Start with simpler approaches (like lock-based implementations) and move to more advanced techniques (like lock-free algorithms) only when necessary and with proper expertise.

The key is to:

1. **Understand the theory** - Know what correctness and progress mean
2. **Choose appropriate tools** - Match the technique to the requirements
3. **Test thoroughly** - Concurrent bugs are often subtle and timing-dependent
4. **Document assumptions** - Make synchronization strategies explicit

Remember: correctness first, performance second. A fast program that occasionally produces wrong results is worse than a slower program that always works correctly.