CH1-2 SUMMARY

COS 226 Concurrency: Complete Guide to Multiprocessor Programming

1. Introduction to Multiprocessor Programming

The Multicore Reality

Modern computing has shifted from the traditional approach of making single processors faster to using multiple processor cores working together. This shift happened because:

- **Traditional scaling**: In the past, we could make programs run faster simply by waiting for processors to get faster (Moore's Law)
- **Multicore scaling**: Now we have multiple cores, but utilizing them effectively is not automatic
- **Real-world challenges**: Getting actual speedup requires careful programming - parallelization and synchronization are complex

The Shared Memory Multiprocessor (SMP) Model

In a shared memory multiprocessor system:

- Multiple processors (cores) share the same memory space
- Each processor has its own cache for performance
- All processors communicate through shared memory via a bus system
- Objects and data structures live in the shared memory that all threads can access

Key Characteristics of Our Programming Model

1. **Multiple threads**: Sometimes called processes, these are independent execution paths
2. **Single shared memory**: All threads can access the same memory locations
3. **Objects live in memory**: Data structures are stored in shared memory
4. **Unpredictable asynchronous delays**: We can't predict when threads will be interrupted or how long operations will take

2. The Mutual Exclusion Problem: Alice & Bob's Pond

The Classic Problem Setup

The mutual exclusion problem is illustrated through a story about Alice and Bob who share a pond for their pets:

- Alice and Bob each have a pet
- Both pets want to use the pond
- **The problem**: The pets don't get along and can't be in the pond simultaneously
- **The challenge**: Alice and Bob can't see each other or communicate directly

Formalizing Mutual Exclusion

Mutual exclusion requires two key properties:

1. **Safety Property (Mutual Exclusion)**:
   - Both pets should never be in the pond at the same time
   - This ensures nothing bad happens
2. **Liveness Property (No Deadlock)**:
   - If only one wants in, it gets in
   - If both want in, at least one eventually gets in
   - This ensures something good eventually happens

Failed Solution Attempts

Simple Protocol: "Just Look"

Approach: Look at the pond to see if it's clear
Problem: Trees obscure the view
Lesson: Threads can't "see" what other threads are doing - explicit communication is required

Cell Phone Protocol: Direct Communication

Approach: Bob calls Alice (or vice versa)
Problems:

- Bob might be in the shower
- Alice might be recharging her battery
- Bob might be shopping
  Lesson: Message-passing doesn't work because the recipient might not be listening or available

Can Protocol: Interrupt-Based Communication

Approach: Bob puts cans on Alice's windowsill with strings leading to his house

- When Bob wants to send a message, he knocks over a can
- Alice resets the cans when she sees them

Problems:

- Alice might go on holiday (can't reset cans)
- Cans can't be reused
- Bob eventually runs out of cans

Lesson: Cannot solve mutual exclusion with interrupts - requires infinite number of available bits

3. The Flag Protocol: A Working Solution

Alice's Protocol

1. Raise flag to indicate interest in using the pond
2. Check if Bob's flag is down
3. If Bob's flag is down, release pet
4. When pet returns, lower flag

Bob's Protocol (Refined Version)

1. Raise flag to show interest
2. While Alice's flag is up:
    - Lower own flag
    - Wait for Alice's flag to go down
    - Raise flag again
3. When Alice's flag is down, unleash pet
4. Lower flag when pet returns

Key Insight: Bob defers to Alice, creating a priority system.

The Flag Principle

The fundamental principle that makes this work:

- Each thread raises its flag first

- Then looks at the other's flag
- **Critical insight**: If both raise flags and look, the last to look must see both flags up

Analysis of Flag Protocol

✅ **Mutual Exclusion**: YES - Pets are never in the yard simultaneously

✅ **Deadlock Freedom**: YES - When both pets want to use the yard, Bob defers to Alice

❌ **Starvation Freedom**: NO - Bob's pet might never get a turn if Alice's pet keeps using the pond

❌ **Fairness**: The protocol is unfair to Bob

❌ **Waiting Issues**: If Alice becomes ill while her flag is up, Bob must wait indefinitely

4. Producer-Consumer Problem

The New Scenario

Alice and Bob have divorced but still need to coordinate:

- Bob puts food in the pond (producer)
- Alice releases pets to eat the food (consumer)
- They can't meet due to restraining orders

The Challenge

Avoid two problems:

1. Releasing pets when there's no food
2. Putting out food if uneaten food remains

The Solution: Can Protocol Revisited

1. Bob puts food in pond
2. Bob knocks over can to signal Alice
3. Alice sees the can and releases pets
4. Alice resets can when pets are fed

This creates a **bounded buffer problem**: managing a fixed-size buffer between producer and consumer.

Properties Achieved

- **Mutual Exclusion**: Pets and Bob never in pond together
- **No Starvation**: If Bob is always willing to feed and pets are always hungry, pets eat infinitely often
- **Producer/Consumer Coordination**: Pets never enter pond unless there's food, Bob never provides food if there's unconsumed food

5. Reader/Writer Problem: The Billboard Communication

The Setup

Alice and Bob communicate using large billboards with letter tiles from Scrabble:

- They write messages one letter at a time
- Multiple messages can be written simultaneously
- **Problem**: Messages can get mixed up (e.g., "WASH" and "SELL" become garbled)

The Reader/Writer Challenge

Design a protocol where:

- Writer writes one letter at a time
- Reader reads one letter at a time
- Reader sees either complete old message or complete new message
- **No mixed messages allowed**

This illustrates the general reader/writer synchronization problem in concurrent systems.

6. Performance Motivation: Amdahl's Law

Why Multicore Programming Matters

Simply upgrading from a single processor to an n-way multiprocessor doesn't automatically give n-fold performance increase. The key challenges:

- We want as much code as possible to execute in parallel
- Sequential parts of the program limit overall performance gains

Amdahl's Law Explained

Amdahl's Law quantifies the theoretical speedup limit based on the sequential portion of a program.

Formula:

```
Speedup = 1 / (1 - p + p/n)
```

Where:

- n = number of processors
- p = fraction of task that can be executed in parallel
- (1-p) = fraction that must be executed sequentially

Key Insight: The sequential part of the task will take (1-p) time, while the parallel part will take p/n time.

Practical Examples

Example 1: 10 processors, 60% parallel, 40% sequential

- Speedup = 1 / (0.4 + 0.6/10) = 1 / 0.46 = 2.17
- Only 2.17x speedup instead of 10x!

Example 2: 10 processors, 80% parallel, 20% sequential

- Speedup = 1 / (0.2 + 0.8/10) = 1 / 0.28 = 3.57

Example 3: 10 processors, 90% parallel, 10% sequential

- Speedup = 1 / (0.1 + 0.9/10) = 1 / 0.19 = 5.26

The Moral

To effectively use multiple processors:

1. Find ways to parallelize code effectively
2. Minimize sequential parts
3. Reduce idle time when threads wait

Even small sequential portions dramatically limit speedup potential.

7. Formal Models of Concurrency

Time and Events

To reason about concurrent programs, we need formal models:

Event: An instantaneous occurrence in a thread (e.g., reading a variable, writing a value)
Thread: A sequence of events a_0, a_1, a_2, ...
Notation: $a_0 \rightarrow a_1$ indicates ordering of events

Intervals and Precedence

Interval: Time between two events, $A_0 = (a_0, a_1)$
Precedence: Interval A_0 precedes interval B_0 if A_0 ends before B_0 starts
Notation: $A_0 \rightarrow B_0$ means "A_0 happens before B_0"

Properties of Precedence Ordering

1. **Never reflexive**: $A \not\rightarrow A$ (nothing precedes itself)
2. **Not symmetric**: If $A \rightarrow B$, then $B \not\rightarrow A$
3. **Transitive**: If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$
4. **Can be incomparable**: Both $A \not\rightarrow B$ and $B \not\rightarrow A$ might be false (concurrent intervals)

8. Critical Sections and Locks

The Counter Example Problem

Consider a simple counter implementation:

```
public class Counter {
    private long value;
    public long getAndIncrement() {
        value++;  // This is actually multiple operations!
        return value;
    }
}
```

The `value++` operation actually consists of:

1. Read current value into temporary variable
2. Add 1 to temporary variable
3. Write result back to value
4. Return the value

Problem: With concurrent access, two threads might read the same value, increment it, and write back the same result, losing one increment.

Critical Sections

A **critical section** is a block of code that can be executed by only one thread at a time. It requires **mutual exclusion**.

Lock Interface

```
public interface Lock {
    public void lock();    // acquire lock
    public void unlock();  // release lock
}
```

Using Locks Properly

```
public class Counter {
    private long value;
    private Lock lock;

    public long getAndIncrement() {
        lock.lock();           // acquire lock
        try {
            int temp = value;
            value = value + 1;  // critical section
            return temp;
        } finally {
            lock.unlock();     // always release lock
        }
    }
}
```

The `finally` block ensures the lock is released even if an exception occurs.

9. Properties of Good Lock Algorithms

Three Essential Properties

1. **Mutual Exclusion**: Threads do not access the critical section simultaneously
2. **Deadlock-Free**: If some thread attempts to acquire the lock, some thread will succeed
3. **Starvation-Free**: Every thread that attempts to acquire the lock will eventually succeed

Understanding the Differences

Deadlock-Free vs Starvation-Free:

- **Deadlock-free**: The system as a whole makes progress (at least one thread completes)
- **Starvation-free**: Individual threads make progress (every thread eventually gets the lock)

A system can be deadlock-free but not starvation-free if some threads never get their turn.

10. Two-Thread Lock Solutions

Thread Conventions

For two-thread solutions:

- `i` represents the current thread (0 or 1)
- `j = 1 - i` represents the other thread
- Each thread knows its own ID

LockOne Algorithm

Basic Idea: Thread indicates interest, then checks if other thread is in critical section.

```
class LockOne implements Lock {
    private boolean[] flag = new boolean[2];

    public void lock() {
        flag[i] = true;         // Set my flag
        while (flag[j]) {}      // Wait for other flag to go false
    }

    public void unlock() {
        flag[i] = false;        // Clear my flag
    }
}
```

Analysis:

- ✅ **Mutual Exclusion**: YES - if both flags are true, both threads wait
- ❌ **Deadlock-Free**: NO - if both threads set flags simultaneously, both wait forever
- **Sequential Access**: Works fine when threads don't run concurrently

LockTwo Algorithm

Basic Idea: Thread offers to be the victim that defers to the other thread.

```
class LockTwo implements Lock {
    private int victim;

    public void lock() {
        victim = i;             // Let other go first
        while (victim == i) {} // Wait for permission
    }

    public void unlock() {
        // Nothing to do
    }
}
```

Analysis:

- ✅ **Mutual Exclusion**: YES - only one thread can not be the victim
- ❌ **Sequential Access**: Fails with sequential access (both threads would be victim)
- **Concurrent Access**: Works fine when threads run simultaneously

Peterson's Algorithm: The Complete Solution

Key Insight: Combine LockOne and LockTwo to get benefits of both.

```
class Peterson implements Lock {
    private boolean[] flag = new boolean[2];
    private int victim;

    public void lock() {
        flag[i] = true;           // Announce interest
        victim = i;               // Defer to other
        while (flag[j] && victim == i) {}  // Wait while other interested
AND I'm victim
    }

    public void unlock() {
        flag[i] = false;          // No longer interested
    }
}
```

How it Works:

1. Thread announces interest (`flag[i] = true`)
2. Thread offers to defer (`victim = i`)
3. Thread waits only if other thread is interested AND current thread is the victim
4. If no conflict, thread proceeds immediately
5. If conflict, the thread that set victim last waits

Complete Analysis:

- ✅ **Mutual Exclusion**: Cannot have both threads in critical section
- ✅ **Deadlock-Free**: In any conflict, exactly one thread is not the victim
- ✅ **Starvation-Free**: When thread re-enters, it sets itself as victim, allowing other thread to proceed

11. N-Thread Lock Solutions

Filter Lock: Peterson's Algorithm Extended

Concept: Generalize Peterson's algorithm to work with n threads using n-1 "waiting rooms" (levels).

Key Ideas:

- Threads must traverse levels 1 through n-1 to reach critical section (level n)
- At each level, at least one thread trying to enter that level succeeds
- At least one thread is blocked if many try to enter the same level
- Only one thread can make it through all levels to the critical section

```
class Filter implements Lock {
    int[] level;   // level[i] = current level of thread i
    int[] victim;  // victim[L] = thread that's victim at level L

    public Filter(int n) {
        level = new int[n];
        victim = new int[n];
        for (int i = 0; i < n; i++) {
            level[i] = 0;  // All threads start at level 0
        }
    }

    public void lock() {
        for (int L = 1; L < n; L++) {           // For each level 1 to n-1
            level[i] = L;                       // Announce intention to enter level L
            victim[L] = i;                      // Give priority to others
            while ((∃k ≠ i: level[k] >= L) && victim[L] == i) {
                // Wait while someone else is at same or higher level AND I'm the victim
            }
        }
    }

    public void unlock() {
```

```
            level[i] = 0;  // Return to level 0
        }
    }
```

Analysis:

- ✅ **Mutual Exclusion**: Only one thread can reach level n
- ✅ **Deadlock-Free**: At each level, at least one thread can progress
- ✅ **Starvation-Free**: Uses Peterson's algorithm logic at each level
- ❌ **Fairness**: Threads can be overtaken arbitrary number of times

Bakery Algorithm: First-Come-First-Served

Concept: Like taking a number at a bakery - serve customers in order of their numbers.

How it Works:

1. Thread takes a number (label) when it wants to enter
2. Thread waits until all threads with lower numbers have been served
3. Uses lexicographic ordering to break ties

```
class Bakery implements Lock {
    boolean[] flag;  // flag[i] = true if thread i wants to enter
    int[] label;     // label[i] = thread i's number

    public Bakery(int n) {
        flag = new boolean[n];
        label = new int[n];
        for (int i = 0; i < n; i++) {
            flag[i] = false;
            label[i] = 0;
        }
    }

    public void lock() {
        flag[i] = true;  // I'm interested
        label[i] = max(label[0], ..., label[n-1]) + 1;  // Take next number
        while (∃k: flag[k] && (label[k], k) << (label[i], i)) {
            // Wait while someone is interested with lower (label, thread_id)
        }
    }

    public void unlock() {
        flag[i] = false;  // No longer interested
```

```
24        }
25    }
```

Lexicographic Ordering: `(label[i], i) << (label[j], j)` if and only if:

- `label[i] < label[j]` OR
- `label[i] == label[j] AND i < j`

This handles the case where two threads might read the same maximum and choose the same label.

Complete Analysis:

- ✅ **Mutual Exclusion**: Two threads cannot have the same (label, thread_id) pair
- ✅ **Deadlock-Free**: There's always one thread with the earliest label pair
- ✅ **Starvation-Free**: Labels are monotonically increasing
- ✅ **First-Come-First-Served**: Earlier callers get lower labels

Practical Issues:

- **Scalability**: Need to read n distinct locations (not practical for large n)
- **Bounded Timestamps**: Labels grow without bound, may overflow in long-lived systems

Solutions to Bounded Timestamps

- Replace labels with bounded timestamps
- Ensure timestamps maintain ordering property
- May require sequential timestamp assignment using mutual exclusion

12. Key Takeaways and Practical Implications

Fundamental Insights

1. **Coordination is Hard**: Even simple problems like mutual exclusion require sophisticated solutions
2. **Properties Trade-offs**: Different algorithms optimize for different properties (fairness vs. performance)
3. **Scalability Challenges**: Solutions that work for 2 threads may not scale to n threads effectively

Practical Considerations

1. **Performance**: Algorithms like Bakery are elegant but not practical due to scalability issues
2. **Fairness vs Efficiency**: Fair algorithms often sacrifice performance
3. **Real-world Implementation**: Most practical systems use hardware-supported primitives rather than these software-only solutions

Why Study These Algorithms?

1. **Foundational Understanding**: Provides deep insight into synchronization challenges
2. **Proof Techniques**: Demonstrates how to reason about concurrent systems
3. **Algorithm Design**: Shows progression from simple ideas to complete solutions
4. **Theoretical Limits**: Establishes what's possible with software-only solutions

These algorithms form the theoretical foundation for understanding more practical synchronization primitives like hardware locks, semaphores, and higher-level concurrent data structures used in modern systems.