CH4 SUMMARY

COS 226: Foundations of Shared Memory - Complete Guide

1. Introduction and Core Concepts

The Church-Turing Thesis Connection

The slides begin by establishing a parallel to sequential computing. Just as the Church-Turing Thesis states that "anything that can be computed, can be computed by a Turing Machine," we need similar foundational principles for concurrent (parallel) computing.

Sequential Computing Model:

- Single Turing Machine
- Reads and writes to infinite tape
- Finite state controller
- Deterministic execution

Concurrent Shared-Memory Computing Model:

- Multiple threads (each running sequential programs)
- Communication through shared memory objects
- Non-deterministic execution due to timing

What Are Threads?

Threads are the basic units of execution in concurrent programming. Key characteristics:

- **Asynchronous**: Threads run independently at different speeds
- **Unpredictable timing**: Can be halted for unpredictable durations
- **Shared memory communication**: Unlike message passing, threads communicate by reading and writing shared memory locations

Example scenario: Imagine two people (threads) working on a shared whiteboard (memory). One person might write something, get interrupted for coffee, while the other person reads what was written and adds their own notes.

2. Fundamental Questions About Shared Memory

The slides pose three critical questions that drive the entire discussion:

1. **What is the weakest useful form of shared memory?**
2. **What can it do?**
3. **What can't it do?**

These questions are essential because understanding the minimal building blocks helps us:

- Design efficient systems
- Understand fundamental limitations
- Build more complex structures from simple ones

3. Registers: The Basic Building Block

What is a Register?

A register is the simplest form of shared memory - essentially a memory location that can hold a binary value. The name is historical (from hardware registers), but conceptually it's just a storage location.

Basic Operations:

```
public interface Register<T> {
    public T read();         // Get the current value
    public void write(T v); // Set a new value
}
```

Types of Registers by Access Pattern

Single-Reader/Single-Writer (SRSW):

- Only one thread can read
- Only one thread can write
- Simplest case, like a private notebook

Multi-Reader/Single-Writer (MRSW):

- Multiple threads can read
- Only one thread can write
- Like a bulletin board where one person posts, many can read

Multi-Reader/Multi-Writer (MRMW):

- Multiple threads can read

- Multiple threads can write
- Most complex, like a shared document everyone can edit

The Challenge of Concurrent Access

When multiple threads access registers simultaneously, we need to define what happens when operations overlap. This is where consistency models become crucial.

4. Consistency Models: Safe, Regular, and Atomic

Safe Registers

Definition: A single-writer, multi-reader register is safe if:

- Non-overlapping reads return the last written value
- Overlapping reads can return ANY value in the register's range

Example: If a register normally holds 0 or 1, but a read overlaps with a write, it might return any random value like 42, "hello", or garbage data.

Why "Safe"? It's the weakest guarantee - you won't crash the system, but you might get nonsensical data during concurrent operations.

Regular Registers

Definition: A single-writer, multi-reader register is regular if:

- Non-overlapping reads return the last written value
- Overlapping reads return either the OLD or NEW value (no garbage)

Key insight: Regular registers can "flicker" between old and new values during writes, but won't return random garbage.

Example scenario:

```
Timeline: write(0) → write(1)
          read(1)  → read(0)
```

This is regular because each read returns either the old (0) or new (1) value, even though they seem out of order.

Atomic Registers

Definition: The strongest consistency model - atomic registers are linearizable implementations where:

- Each read returns the value from the most recent write
- Operations appear to happen instantaneously at some point during their execution
- Maintains the illusion of sequential execution

Key difference from regular: Atomic registers prevent reads from appearing "out of order" - if one read returns a newer value, subsequent reads cannot return older values.

5. The Register Construction Hierarchy

The slides demonstrate how to build stronger consistency guarantees from weaker ones:

```
SRSW Safe Boolean → MRSW Safe Boolean → MRSW Regular Boolean
→ MRSW Regular M-valued → MRSW Atomic → MRMW Atomic
```

Building MRSW Safe from SRSW Safe

The Array Approach:

```java
public class SafeBoolMRSWRegister implements Register<Boolean> {
    boolean[] s_table; // Array of SRSW registers

    public boolean read() {
        return s_table[ThreadID.get()]; // Each thread reads its own
    }

    public void write(boolean x) {
        for (int i = 0; i < s_table.length; i++)
            s_table[i] = x; // Write to all positions
    }
}
```

How it works:

- Give each reader thread its own SRSW register
- Writer updates all registers sequentially
- Each reader only reads its dedicated register
- No concurrent access to individual SRSW registers

From Safe to Regular Boolean MRSW

The key insight: Safe and Regular only differ when the new value equals the old value.

Problem with safe: If writing 1 when register already contains 1, a safe register might return 0 during the write.

Regular solution: Only write when the value actually changes.

```java
public class RegBoolMRSWRegister implements Register<Boolean> {
    threadLocal boolean old; // Last value this thread wrote
    private SafeBoolMRSWRegister value;

    public void write(boolean x) {
        if (old != x) {    // Only write if different
            value.write(x);
            old = x;
        }
    }
}
```

Building Multi-Valued Registers

The Unary Representation Technique:
Instead of storing a number directly, use an array of Boolean registers where position `i` being `true` means the value is `i`.

Example: Value 5 in an 8-bit system:

```
Position: [0][1][2][3][4][5][6][7]
Value:    [F][F][F][F][F][T][F][F]
```

Write algorithm:

1. Set position `x` to true
2. Set all positions below `x` to false

Read algorithm:

1. Scan from position 0 upward
2. Return the first position that's true

This ensures that during a write operation, readers see either the old value or the new value, never garbage.

6. The Timestamp Solution for Atomicity

The Problem with Regular Registers

Regular registers can violate atomicity when reads appear out of temporal order:

```
Timeline: Initially 1234
Writer:   write(5678) ────────────────
Reader A:                read(5678)
Reader B:                          read(1234) ← Problem!
```

Reader B executed after Reader A but got an older value.

Timestamped Values Solution

Core idea: Attach timestamps to values and use them to enforce ordering.

```java
public class StampedValue<T> {
    public long stamp;     // Timestamp
    public T value;        // Actual value
}
```

Writer behavior:

- Increment timestamp with each write
- Write (timestamp, value) pair

Reader behavior:

- Remember the highest timestamp seen
- If a read returns a lower timestamp, ignore it and return the previous highest

SRSW Atomic Implementation

```java
public class AtomicSRSWRegister<T> implements Register<T> {
    StampedValue<T> lastRead;  // Highest timestamp/value seen
    StampedValue<T> value;     // Current register contents

    public T read() {
        StampedValue<T> result = StampedValue.max(value, lastRead);
        lastRead = result;  // Update what we've seen
        return result.value;
    }

    public void write(T v) {
        long stamp = lastStamp + 1;
        value = new StampedValue(stamp, v);
```

```
            lastStamp = stamp;
        }
    }
```

7. Building Multi-Reader Atomic Registers

The Challenge

Simply using an array of SRSW atomic registers (like we did for safe registers) doesn't work for atomicity. Later readers might miss updates that earlier readers saw.

The Matrix Solution

Data structure: n×n matrix where:

- Rows represent readers
- Columns represent readers
- Each cell contains a stamped value
- Writers write only to diagonal elements

Read algorithm:

1. Find maximum timestamped value in your column
2. Write this maximum to all cells in your row (helping other readers)
3. Return the value

Write algorithm:

1. Create new timestamped value
2. Write to all diagonal positions

Why it works: The matrix ensures that information flows between readers. If Reader A sees a new value, it writes that information to its row, making it available to future readers in their columns.

8. Multi-Writer Atomic Registers

The Final Construction

Data structure: Array of atomic MRSW registers, one per writer.

Write algorithm:

1. Read all registers to find maximum timestamp
2. Write new value with timestamp = max + 1 to your register

Read algorithm:

1. Read all registers
2. Return value with highest timestamp

```java
public class AtomicMRMWRegister implements Register<T> {
    StampedValue<T>[] a_table; // One per writer

    public void write(T value) {
        int me = ThreadID.get();
        StampedValue<T> max = findMaxTimestamp();
        a_table[me] = new StampedValue(max.stamp + 1, value);
    }

    public T read() {
        return findMaxTimestamp().value;
    }
}
```

9. Key Principles and Wait-Free Implementation

Wait-Free Property

All the constructions shown are **wait-free**: every method call completes in a finite number of steps, regardless of what other threads are doing.

Why important:

- No mutual exclusion needed
- Guarantees independent progress
- No deadlocks or priority inversion
- Fault tolerance

The Construction Hierarchy Summary

1. **SRSW Safe Boolean** (given primitive)
2. **MRSW Safe Boolean** (array of SRSW)
3. **MRSW Regular Boolean** (conditional writing)
4. **MRSW Regular M-valued** (unary representation)
5. **SRSW Atomic** (timestamps)

6. **MRSW Atomic** (matrix + timestamps)
7. **MRMW Atomic** (array of MRSW atomic)

10. Theoretical Significance

Universal Construction Principle

This hierarchy proves that you can build arbitrarily complex concurrent data structures starting from the simplest possible building block: a single-reader, single-writer safe Boolean register.

Practical Implications

- **Hardware design**: Shows minimum requirements for concurrent systems
- **Software engineering**: Provides building blocks for lock-free algorithms
- **Theoretical foundation**: Establishes what's computationally possible in concurrent systems

The Remarkable Result

The slides conclude with a powerful statement: "One can construct a wait-free MRMW atomic register from SRSW Safe Boolean registers."

This means that even with the weakest possible shared memory primitive, we can build strong consistency guarantees through clever algorithmic techniques, without requiring hardware support like atomic compare-and-swap operations.

11. Real-World Connections

Modern Multiprocessors

These concepts directly apply to:

- **CPU cache coherence protocols**
- **Memory consistency models** (sequential consistency, release consistency)
- **Lock-free data structures** (queues, stacks, hash tables)
- **Database transaction isolation levels**

Performance Considerations

While these constructions are theoretically elegant, they often have high overhead:

- Multiple memory accesses per operation
- Space overhead (arrays, matrices)
- In practice, hardware often provides stronger primitives (compare-and-swap, load-linked/store-conditional)

The value lies in understanding the theoretical foundations and minimum requirements for concurrent computation.