CH5 SUMMARY

The Relative Power of Primitive Synchronization Operations

Chapter 5 - COS 226

Introduction and Motivation

When designing a new multiprocessor system, one crucial question arises: **which atomic instructions should be included?** This isn't just an academic question - supporting all possible atomic operations would be inefficient and costly. Therefore, we need to understand which synchronization primitives are truly powerful and necessary.

The core challenge is that in a multiprocessor environment, multiple threads need to coordinate their actions safely. Without proper synchronization, we get race conditions, data corruption, and unpredictable behavior. But different synchronization primitives have different levels of "power" - some can solve more complex problems than others.

Wait-Free Implementation

Wait-free implementation is the gold standard of concurrent programming. It means:

- Every method call completes in a **finite number of steps**
- No thread can be blocked indefinitely by other threads
- **No mutual exclusion is needed** - threads don't have to wait for locks

This is powerful because it eliminates many common problems like deadlock, priority inversion, and thread starvation. However, achieving wait-free implementations is extremely challenging and often requires sophisticated algorithms.

The Register Hierarchy

The slides show a progression from the simplest possible shared memory construct to more complex ones:

Starting Point: Safe Boolean Register

- **Single Reader, Single Writer (SRSW)**
- Only stores Boolean values (0 or 1)

- "Safe" means that if no write is occurring, reads return the correct value
- If a read happens during a write, it can return any valid Boolean value

Progression Through Complexity

1. **SRSW → SRSW M-valued**: From Boolean to multi-valued registers
2. **Safe → Regular → Atomic**: Increasing consistency guarantees
3. **SRSW → MRSW → MRMW**: From single reader/writer to multiple readers/writers

Each step up this hierarchy requires increasingly sophisticated implementation techniques, but eventually leads to **atomic registers** that can be safely accessed by multiple readers and writers simultaneously.

Why Mutual Exclusion is Problematic

The slides highlight several scenarios where traditional mutual exclusion (locks) creates serious problems:

Asynchronous Interrupts

In real systems, threads can be interrupted at any time by the operating system. If a thread holding a lock gets interrupted, all other threads waiting for that lock are stuck until the interrupted thread resumes.

Heterogeneous Processors

Modern systems often have processors with vastly different speeds. A slow processor (like an old Pentium 286) holding a lock can block much faster processors (like modern Pentiums), creating severe performance bottlenecks.

Fault Tolerance

If a thread holding a lock crashes, all other threads waiting for that lock may be permanently blocked, making the entire system fragile.

The Consensus Problem

To evaluate the power of different synchronization primitives, the slides introduce the **consensus problem** as a benchmark:

What is Consensus?

Consensus is a fundamental problem in distributed computing where:

1. Each thread starts with a **private input value**
2. Threads **communicate** using shared objects
3. All threads must **agree on one of the input values**

Formal Definition

A consensus object provides a `decide()` method with these properties:

- **Consistent**: All threads that call `decide()` return the same value
- **Valid**: The returned value must be one of the input values from the participating threads

Example Scenario

```
Thread A inputs: 32
Thread B inputs: 19
Thread C inputs: 21

After communication, all threads decide: 19
(19 was one of the input values, satisfying validity)
```

Consensus Numbers: The Power Hierarchy

The **consensus number** of a synchronization primitive is the maximum number of threads that can solve consensus using only that primitive. This creates a hierarchy:

- **Consensus number 1**: Only works for sequential programs (no real concurrency)
- **Consensus number 2**: Can handle two-thread synchronization
- **Consensus number ∞**: Can handle any number of threads

Atomic Registers: Consensus Number 1

The Impossibility Result

The slides prove that **atomic registers cannot solve consensus for even two threads**. Here's the intuitive explanation:

The Proof Sketch

Consider two threads with different input values (0 and 1):

1. **Univalent executions**: If both threads have the same input, the outcome is predetermined
   - Both input 0 → must decide 0
   - Both input 1 → must decide 1
2. **The problem case**: When inputs differ (Thread A has 0, Thread B has 1)
   - Solo execution by A must decide 0 (to be consistent with the all-0 case)
   - Solo execution by B must decide 1 (to be consistent with the all-1 case)
   - But this means different executions can lead to different decisions, violating consistency

This fundamental limitation means that **atomic registers have consensus number 1** - they can only provide meaningful synchronization in sequential programs.

FIFO Queues: Consensus Number 2

The Algorithm

FIFO queues can solve two-thread consensus using a clever protocol:

1. **Setup**: Initialize a FIFO queue with a red ball followed by a black ball
2. **Protocol**:
   - Each thread writes its proposed value to a shared array
   - Both threads race to dequeue the first item from the queue
   - The thread that gets the **red ball** decides its own value
   - The thread that gets the **black ball** decides the other thread's value

Code Implementation

```
public class QueueConsensus extends ConsensusProtocol {
    private Queue queue;

    public QueueConsensus() {
        queue = new Queue();
        queue.enq(Ball.RED);    // Winner marker
        queue.enq(Ball.BLACK);  // Loser marker
    }

    public Object decide(Object value) {
        propose(value);  // Write value to shared array
        Ball ball = queue.deq();  // Race for the first ball

```

```
            if (ball == Ball.RED)
                return proposed[ThreadID.get()];  // I win - my value
            else
                return proposed[1 - ThreadID.get()];  // I lose - other's
    value
        }
    }
```

Why It Works

The FIFO queue provides a **total ordering** - exactly one thread gets the red ball and exactly one gets the black ball. This breaks the symmetry that made atomic registers fail.

Limitation: Cannot Handle Three Threads

While FIFO queues solve two-thread consensus, they cannot solve three-thread consensus. The proof of this limitation establishes that **FIFO queues have consensus number 2**.

Read-Modify-Write Operations

What are RMW Operations?

Read-Modify-Write (RMW) operations atomically:

1. Read a value from memory
2. Apply a function to transform it
3. Write the result back
4. Return the original value

Common RMW Operations

From `java.util.concurrent`:

- **getAndSet(x)**: Replace current value with x, return old value
- **getAndIncrement()**: Add 1 to current value, return old value
- **getAndAdd(k)**: Add k to current value, return old value
- **get()**: Simply return the current value

The Exception: compareAndSet()

This operation is special:

- Takes an expected value `e` and update value `u`
- If current value equals `e`, replace it with `u`
- Returns boolean indicating success
- This conditional behavior makes it more powerful than other RMW operations

Trivial vs Non-Trivial RMW

- **Trivial**: The function doesn't change the value (e.g., identity function)
- **Non-trivial**: There exists some value v where v ≠ function(v)

Examples:

- `identity(x) = x` is **trivial**
- `increment(x) = x + 1` is **non-trivial**

Non-Trivial RMW: Consensus Number 2

The Algorithm

Any non-trivial RMW operation can solve two-thread consensus:

```
public class RMWConsensus extends ConsensusProtocol {
    private RMWRegister r = new RMWRegister(initialValue);

    public Object decide(Object value) {
        propose(value);  // Store my value in shared array

        if (r.getAndMumble() == initialValue)
            return proposed[ThreadID.get()];  // I was first
        else
            return proposed[otherThreadID];   // Other was first
    }
}
```

Why It Works

- The RMW operation can only return the initial value to **exactly one thread**
- This thread knows it won the race and decides its own value
- The other thread knows it lost and decides the winner's value
- The non-trivial nature ensures the value changes, preventing both threads from thinking they won

Common2 RMW Operations

The slides define a special class of RMW operations called **Common2**, where for any two functions f_i and f_j, either:

They Commute

`f_i(f_j(v)) = f_j(f_i(v))` for all values v

Examples:

- **getAndAdd()**: Addition is commutative
- **getAndIncrement()**: Special case of addition

They Overwrite

`f_i(f_j(v)) = f_i(v)` for all values v

Examples:

- **getAndSet()**: The second operation completely overwrites the first

Consensus Number = 2

Theorem: Any RMW register in Common2 has consensus number exactly 2.

This means these operations can solve two-thread consensus but cannot handle three or more threads.

compareAndSet(): Unlimited Power

The Key Difference

Unlike other RMW operations, `compareAndSet()` is **conditional** - it only updates the value if it matches an expected value. This conditional behavior gives it unlimited power.

Consensus Algorithm for Any Number of Threads

```
1  public class CASConsensus extends ConsensusProtocol {
2      private final int FIRST = -1;
3      private AtomicInteger r = new AtomicInteger(FIRST);
4
```

```
    public Object decide(Object value) {
        propose(value);  // Store my value
        int i = ThreadID.get();

        if (r.compareAndSet(FIRST, i))
            return proposed[i];        // I won
        else
            return proposed[r.get()];  // Someone else won
    }
}
```

Why It Has Infinite Consensus Number

- The register starts with a special "FIRST" value (-1)
- Each thread tries to replace FIRST with its own thread ID
- **Exactly one thread** succeeds in this replacement
- That thread's value becomes the consensus decision
- All other threads can read the winner's ID and return the corresponding value

This algorithm works for any number of threads because `compareAndSet()` provides an atomic "test-and-set" operation that can break symmetry among any number of competing threads.

The Synchronization Hierarchy

The slides establish a clear hierarchy:

1. **Level 1**: Atomic registers - consensus number 1
2. **Level 2**: FIFO queues, non-trivial RMW operations - consensus number 2
3. **Level ∞**: compareAndSet() and similar operations - unlimited consensus number

This hierarchy is fundamental to understanding concurrent programming: to solve n-thread synchronization problems, you need primitives with consensus number at least n.

Practical Implications

For System Designers

- Including `compareAndSet()` in your instruction set gives you universal synchronization power
- Simpler operations like atomic reads/writes are not sufficient for complex synchronization

- The choice of synchronization primitives fundamentally limits what concurrent algorithms are possible

For Programmers

- Understanding consensus numbers helps predict which synchronization problems can be solved with available primitives
- It explains why certain concurrent data structures require specific atomic operations
- It provides a theoretical foundation for reasoning about the impossibility or possibility of wait-free implementations

This theoretical framework, while abstract, has profound practical implications for both hardware design and concurrent software development.