CH8 SUMMARY

COS 226 Chapter 8: Monitors and Blocking Synchronization - Detailed Summary

Introduction to Monitors

Monitors are a structured programming approach that combines three essential elements into a single modular package:

- Data (the shared information)
- Methods (operations on that data)
- Synchronization mechanisms (controlling access to the data)

Think of a monitor as a protective wrapper around shared data that automatically manages which threads can access it and when. It's like having a librarian who controls access to a special collection—only certain people can enter at certain times, following specific rules.

The Producer-Consumer Problem

The Metaphor

The slides introduce this concept through a story about Alice and Bob:

- Bob puts food in a pond (he's the **producer**)
- Alice releases her pets to feed from the pond (she's the **consumer**)
- They have restraining orders against each other, so they can't coordinate directly
- Problems arise when:
  - Alice releases pets when there's no food (pets go hungry)
  - Bob puts out food when uneaten food already exists (waste)

Real-World Application

In computing, this translates to:

- Two threads (independent execution paths in a program) that need to communicate
- They share a **FIFO queue** (First-In-First-Out, like a line at a store)
- The **producer** generates data and adds it to the queue
- The **consumer** removes data from the queue and processes it

The Core Problem

When the queue becomes full, what happens if the producer tries to add more data? When the queue is empty, what happens if the consumer tries to remove data? These scenarios create synchronization challenges that need careful management.

Initial Approach: Basic Locking

Simple Producer Code

```
mutex.lock()
try {
    queue.enq();
} finally {
    mutex.unlock();
}
```

This basic approach uses a **mutex** (mutual exclusion lock) to ensure only one thread accesses the queue at a time. However, this doesn't solve the full-versus-empty problem.

Problems with Basic Locking

- The producer shouldn't try to add data if the buffer is full
- The consumer shouldn't try to remove data from an empty buffer
- A better solution: Let the queue manage its own synchronization internally

Monitor Locks: The Foundation

How Monitor Locks Work

- **Only one thread at a time** can hold a monitor lock
- A thread **acquires** the lock when it enters a synchronized method
- A thread **releases** the lock when the method returns
- All methods in a monitor automatically acquire the lock on entry and release it on exit

Spinning vs. Blocking

When a thread cannot immediately acquire a lock, it has two options:

Spinning:

- The thread repeatedly checks if the lock is available
- Like constantly refreshing a webpage to see if tickets are available
- Good for **short waits** (microseconds to milliseconds)
- Wastes CPU cycles but avoids overhead of context switching
- **Does not work on single-processor systems** (the spinning thread would prevent the lock holder from running)

Blocking:

- The thread suspends itself and the operating system schedules another thread
- Like setting an alarm and going to sleep instead of staying awake
- Good for **long waits** (milliseconds to seconds)
- Saves CPU resources but has overhead from context switching

In practice: Systems often combine both approaches—spin briefly first, then block if the lock isn't acquired quickly.

Condition Variables: The Key Innovation

The Problem

A thread needs to release a lock temporarily (because a condition isn't met) and then reacquire it later to try again. Basic locks can't do this.

The Solution: Condition Objects

In Java's concurrency package, a **Condition** object provides the ability to release a lock temporarily and wait for a specific condition to become true.

Creating and Using Conditions

```
Condition condition = mutex.newCondition();

mutex.lock();
try {
    while (!property) {
        condition.await();
    } catch (InterruptedException e) {...}
    // ... do work ...
}
```

Breakdown of what happens:

1. Create a Condition object associated with a specific lock
2. Acquire the lock
3. Test if a property/condition holds (e.g., "is queue not full?")
4. If the condition is false, call `await()` which:
   - **Releases the lock** (so other threads can use it)
   - **Suspends the thread** (puts it to sleep)
5. When awakened later, the thread automatically **reacquires the lock** and rechecks the condition
6. The suspension can be interrupted by other threads

Why Use a While Loop?

Notice the `while (!property)` instead of `if (!property)`. This is crucial because when a thread wakes up, there's no guarantee the condition is actually true—another thread might have changed things. Always recheck conditions after waking up.

Complete Producer-Consumer Implementation

The LockedQueue Class Structure

```
class LockedQueue {
    Lock lock = new ReentrantLock();
    Condition isFull = lock.newCondition();
    Condition isEmpty = lock.newCondition();
    T[] items;
    int max;
}
```

Key components:

- One lock for mutual exclusion
- Two conditions: one for "queue is full" and one for "queue is empty"
- An array to store items with a maximum capacity

Producer Method (enq)

```
public void enq(T x) {
    lock.lock();
    try {
        while (items.length == max)
            isFull.await();
        items[tail] = x;
        // ... update tail ...
        isEmpty.signal();
    } finally {
        lock.unlock();
    }
}
```

What happens step by step:

1. Acquire the lock
2. Check if the queue is full
3. If full, wait on the `isFull` condition (releases lock and sleeps)
4. When awakened, reacquire lock and recheck
5. Add the item to the queue

6. **Signal** any waiting consumer that the queue is no longer empty
7. Release the lock

Consumer Method (deq)

```
public T deq() {
    lock.lock();
    try {
        while (items.length == 0)
            isEmpty.await();
        T x = items[head];
        // ... update head ...
        isFull.signal();
    } finally {
        lock.unlock();
    }
}
```

What happens step by step:

1. Acquire the lock
2. Check if the queue is empty
3. If empty, wait on the `isEmpty` condition (releases lock and sleeps)
4. When awakened, reacquire lock and recheck
5. Remove an item from the queue
6. **Signal** any waiting producer that the queue is no longer full
7. Release the lock

Waking Up Threads: Signaling Methods

How does a waiting thread know when to wake up?

1. **Time-based awakening:**
   - `await(long time)` - wait for specified time
   - `awaitUntil(Date)` - wait until specific date/time
   - `awaitNanos(long nanoSec)` - wait for specified nanoseconds
2. **Signal-based awakening:**
   - `signal()` - wakes up **one** waiting thread
   - `signalAll()` - wakes up **all** waiting threads

Important caveat: Even after waking up (whether by timeout or signal), threads must compete for the lock again before proceeding.

Monitor Definition

The combination of methods, mutual exclusion locks, and condition objects working together is called a **monitor**. It's a complete synchronization pattern that encapsulates all the necessary coordination logic.

Lost-Wakeup Problem

What Is It?

A **lost wakeup** occurs when one or more threads wait forever without realizing that the condition they're waiting for has already become true. It's like waiting by the phone for a call that already came while you were in another room.

Example Scenario

```
1   public void enq(T x) {
2       lock.lock();
3       try {
4           while (count == items.length)
5               isFull.await();
6           items[tail] = x;
7           ++count;
8           if (count == 1)
9               isEmpty.signal();
10      } finally {
11          lock.unlock();
12      }
13  }
```

The problem: If there are multiple consumers waiting, only one gets signaled when `count == 1`. If additional items are added before that consumer runs, the other consumers might never wake up even though items are now available.

Prevention Strategies

1. **Signal all necessary threads:** Use `signalAll()` when multiple threads might be waiting
2. **Use timeouts:** Always specify a timeout when waiting so threads periodically recheck conditions
3. **Be careful with conditional signaling:** Don't only signal in specific circumstances unless you're certain it's correct

Readers-Writers Locks

The Problem

Many shared objects have two types of operations:

- **Readers:** Only look at data, don't modify it (e.g., searching a database)
- **Writers:** Modify the data (e.g., updating a record)

Key insight: Multiple readers can safely access data simultaneously since none of them change it, but writers need exclusive access.

Synchronization Requirements

- **Readers:** Can all read simultaneously (no synchronization needed between readers)
- **Writers:** Must have exclusive access (must synchronize with everyone)
- **The challenge:** How do we allow multiple readers while ensuring writers get exclusive access?

The Rules

A readers-writers lock must satisfy:

1. No thread can acquire the **write lock** while any thread holds either the write lock OR the read lock
2. No thread can acquire the **read lock** while any other thread holds the write lock

Interface

```
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

This provides two separate lock objects: one for readers and one for writers.

SimpleReadWriteLock Implementation

Structure

```
public class SimpleReadWriteLock implements ReadWriteLock {
    boolean writer;         // Is there currently a writer?
    int readers;            // How many readers are there?
    Lock lock;              // Internal lock for coordination
    Condition condition;    // For signaling state changes
    Lock readLock, writeLock;  // Separate locks for readers/writers
}
```

Constructor

```
public SimpleReadWriteLock() {
    writer = false;      // No initial writers
```

```
        readers = 0;          // No initial readers
        lock = new ReentrantLock();
        readLock = new ReadLock();
        writeLock = new WriteLock();
        condition = lock.newCondition();
    }
```

ReadLock Implementation

Acquiring the read lock:

```
class ReadLock implements Lock {
    public void lock() {
        lock.lock();
        try {
            while (writer)
                condition.await();  // Wait while there's a writer
            readers++;              // Increment reader count
        } finally {
            lock.unlock();
        }
    }
}
```

What happens:

1. Acquire the internal coordination lock
2. If a writer is active, wait
3. Otherwise, increment the reader count
4. Release the coordination lock (but the read lock is now conceptually held)

Releasing the read lock:

```
public void unlock() {
    lock.lock();
    try {
        readers--;                  // Decrement reader count
        if (readers == 0)
            condition.signalAll();  // Wake waiting writers
    } finally {
        lock.unlock();
    }
}
```

When the last reader leaves (`readers == 0`), signal any waiting writers.

WriteLock Implementation

Acquiring the write lock:

```
class WriteLock implements Lock {
    public void lock() {
        lock.lock();
        try {
            while (readers > 0)
                condition.await();  // Wait while there are readers
            writer = true;          // Mark that a writer is active
        } finally {
            lock.unlock();
        }
    }
}
```

Releasing the write lock:

```
public void unlock() {
    writer = false;
    condition.signalAll();  // Wake all waiting readers and writers
}
```

The Fairness Problem

Issue: The SimpleReadWriteLock is correct but potentially **unfair**. If readers are much more frequent than writers, writers could be **starved**—waiting indefinitely while readers continuously acquire the lock.

Why this happens: As soon as a writer releases the lock, if readers are waiting, they might all acquire the read lock before the next writer gets a chance.

FIFOReadWriteLock: A Fair Solution

The Strategy

Ensure no new readers can acquire the lock after a writer has started waiting. This provides **FIFO fairness**—first come, first served.

Structure

```
public class FIFOReadWriteLock implements ReadWriteLock {
    boolean writer;
    int readAcquires;    // Number of readers that have acquired the lock
    int readReleases;    // Number of readers that have released the lock
    Lock lock;
    Condition condition;
    Lock readLock, writeLock;
```

```
}
```

Key difference: Instead of counting current readers, we track acquisitions and releases separately.

How It Works

ReadLock:

```
public void lock() {
    lock.lock();
    try {
        while (writer)
            condition.await();
        readAcquires++;    // Increment acquisitions
    } finally {
        lock.unlock();
    }
}

public void unlock() {
    lock.lock();
    try {
        readReleases++;    // Increment releases
        if (readAcquires == readReleases)
            condition.signalAll();  // All readers are done
    } finally {
        lock.unlock();
    }
}
```

WriteLock:

```
public void lock() {
    lock.lock();
    try {
        writer = true;     // Block new readers immediately
        while (readAcquires != readReleases)
            condition.await();  // Wait for all current readers to finish
    } finally {
        lock.unlock();
    }
}
```

The fairness mechanism: When a writer tries to acquire the lock, it immediately sets `writer = true`, which prevents any new readers from starting. The writer then waits only for existing readers to finish (when `readAcquires == readReleases`).

Reentrant Locks

The Problem with Non-Reentrant Locks

If a thread tries to acquire a lock it already holds, it will deadlock **with itself**. This happens when:

- A method that acquires a lock calls another method
- That second method tries to acquire the same lock

Example scenario:

```
lock.lock();
try {
    methodA();  // Already holding the lock
    methodB();  // This also tries to lock()... DEADLOCK!
} finally {
    lock.unlock();
}
```

What Is a Reentrant Lock?

A **reentrant lock** (also called a **recursive lock**) can be acquired multiple times by the same thread without deadlocking. Think of it like a building where you can enter and exit multiple rooms using the same key.

Java's Built-In ReentrantLock

The `java.util.concurrent.locks` package provides `ReentrantLock` which:

- Is owned by the last thread that successfully locked it
- Returns immediately if the current thread already holds the lock
- Keeps track of how many times it's been acquired by the owner

Custom ReentrantLock Implementation

Structure:

```
public class SimpleReentrantLock implements Lock {
    Lock lock;
    Condition condition;
    int owner;       // ID of the thread that owns the lock
    int holdCount;   // How many times the owner has acquired it
}
```

Acquiring the lock:

```
public void lock() {
    int me = ThreadID.get();
    lock.lock();

    if (owner == me) {
        holdCount++;    // I already own it, just increment
        return;
    }

    while (holdCount != 0)
        condition.await();  // Someone else owns it, wait

    owner = me;          // Take ownership
    holdCount = 1;       // First acquisition
}
```

What happens:

1. Check if I already own the lock
2. If yes, just increment the hold count and return immediately
3. If no, wait until the lock is free (holdCount == 0)
4. Take ownership and set holdCount to 1

Releasing the lock:

```
public void unlock() {
    lock.lock();
    try {
        if (holdCount == 0 || owner != ThreadID.get())
            throw new IllegalMonitorStateException();

        holdCount--;

        if (holdCount == 0)
            condition.signal();  // Fully released, wake others
    } finally {
        lock.unlock();
    }
}
```

What happens:

1. Verify that I actually own the lock (can't unlock someone else's lock)
2. Decrement the hold count
3. If holdCount reaches 0, the lock is fully released—signal waiting threads

Important: You must call `unlock()` the same number of times you called `lock()` to fully release the lock.

Semaphores

Beyond Mutual Exclusion

A **mutual exclusion lock** allows only **one thread** in the critical section at a time. A **semaphore** is a generalization that allows **up to c threads**, where c is the **capacity**.

Real-world analogy: A mutex is like a single-occupancy bathroom (only one person at a time). A semaphore is like a parking lot with c spaces—multiple cars can park, but once full, others must wait.

Semaphore Structure

```
public class Semaphore {
    Lock lock;
    Condition condition;
    int capacity;    // Maximum number of threads allowed
    int state;       // Current number of threads inside
}
```

Semaphore Operations

Acquiring (entering the critical section):

```
public void lock() {
    lock.lock();
    try {
        while (state == capacity)
            condition.await();  // Wait if at capacity
        state++;                // One more thread enters
    } finally {
        lock.unlock();
    }
}
```

Releasing (leaving the critical section):

```
public void unlock() {
    lock.lock();
    try {
        state--;                // One thread leaves
        condition.signalAll();  // Wake waiting threads
    } finally {
        lock.unlock();
```

```
        }
    }
```

Use Cases

Semaphores are useful when you want to:

- Limit the number of concurrent connections to a resource
- Control access to a pool of resources (e.g., database connections)
- Implement resource throttling (allow only N operations at once)

Example: A web server might use a semaphore with capacity 100 to ensure no more than 100 clients are served simultaneously, preventing resource exhaustion.

Key Takeaways

1. **Monitors** provide a structured way to combine data, methods, and synchronization
2. **Condition variables** allow threads to release locks temporarily while waiting for conditions to become true
3. **Always recheck conditions** after waking up from await()
4. **Signal appropriately** to avoid lost wakeups—use signalAll() when in doubt
5. **Readers-writers locks** optimize performance when reads are more frequent than writes
6. **Fairness matters**—simple implementations can starve certain threads
7. **Reentrant locks** prevent self-deadlock in nested method calls
8. **Semaphores** generalize mutual exclusion to allow multiple concurrent threads

These synchronization primitives form the foundation of concurrent programming, enabling threads to coordinate safely while maximizing parallelism.