Programming Paradigms for Concurrency
Lecture 5 – Monitors and Blocking Synchronization

Based on
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

Thomas Wies
New York University
What Should you do if you can’t get a lock?

• Keep trying
  – “spin” or “busy-wait”
  – Good if delays are short
• Give up the processor
  – Good if delays are long
  – Always good on uniprocessor
What Should you do if you can’t get a lock?

• Keep trying
  – “spin” or “busy-wait”
  – Good if delays are short

• Give up the processor
  – Good if delays are long
  – Always good on uniprocessor
Producer/consumer based on a FIFO Queue

```java
public produce(Object x) {
    mutex.lock();
    try {
        queue.enq(x);
    } finally {
        mutex.unlock();
    }
}
```
The Need for Modular Synchronization

Suppose queue is bounded:

- `enq` may block until queue has room
- decision whether to block depends on internal state of the queue

Multiple producers/consumers:

- every thread needs to keep track of the lock, the queue state, etc.
The Need for
Modular Synchronization

Suppose queue is bounded:
• enq may block until queue has room
• decision whether to block depends on internal state of the queue

Multiple producers/consumers:
• every thread needs to keep track of the lock, the queue state, etc.
Modular Synchronization

Let queue handle its own synchronization

• **queue has its own lock**
  – acquired by each method call
  – released when the call returns

• **if thread enqueues on a full queue**
  – queue itself detects the problem
  – suspend the caller and resume when the queue has room
Conditions

- a condition object is associated with a lock
- condition objects allow a thread to
  - temporarily release the lock and suspend itself until awoken by another thread
  - awake other threads that are currently suspended
Monitors

The combination of

- an object and its methods
- a mutual exclusion lock
- and the lock’s condition objects

is called a **monitor**

Monitors enable modular synchronization.
Java’s Lock Interface

```java
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    void tryLock();
    void tryLock(long time, TimeUnit unit);
    Condition newCondition();
    void unlock();
}
```
Java’s Condition Interface

```java
class Condition {
    public void await() throws InterruptedException;
    public boolean await(long time, TimeUnit unit)
        throws InterruptedException;
    ...
    public void signal();
    public void signalAll();
}
```
Java's Condition Interface

```java
class Condition {
    public void await() throws InterruptedException;
    public boolean await(long time, TimeUnit unit) throws InterruptedException;
    ...
    public void signal();
    public void signalAll();
}
```

wake up one waiting thread
Java’s Condition Interface

```java
public interface Condition {
    void await() throws InterruptedException;
    boolean await(long time, TimeUnit unit)
        throws InterruptedException;
    ...
    void signal();
    void signalAll();
}
```

wake up all waiting threads
A Typical Monitor Execution

lock

lock()

critical section

waiting room
A Typical Monitor Execution
A Typical Monitor Execution

lock()

lock

critical section

waiting room
A Typical Monitor Execution

lock

critical section

waiting room
A Typical Monitor Execution

lock()

lock

waiting room

critical section
A Typical Monitor Execution

lock()

lock

critical section

unlock()

signalAll()

waiting room
A Typical Monitor Execution

lock()

lock

critical section

unlock()
signalAll()
A Typical Monitor Execution

lock

critical section

waiting room
Using Condition Objects

```java
Condition condition = mutex.newCondition();
...
mutex.lock();
try {
    while (!property)
        condition.await();
} catch (InterruptedException e) {
    ...
}
...
Using Condition Objects

```java
Condition condition = mutex.newCondition();
...
mutex.lock();
try {
    while (!property)
        condition.await();
} catch (InterruptedException e) {
    ...
}
```
Using Condition Objects

```java
Condition condition = mutex.newCondition();
...
mutex.lock();
try {
    while (!property)
        condition.await();
} catch (InterruptedException e) {
    ...
}
...
Using Condition Objects

```java
Condition condition = mutex.newCondition();
...
mutex.lock();
try {
    while (!property)
        condition.await();
} catch (InterruptedException e) {
    ...
}
```

not happy
Using Condition Objects

```java
Condition condition = mutex.newCondition();
...
mutex.lock();
try {
    while (!property)
        condition.await();
} catch (InterruptedException e) {
    ...
}
...
```

release the lock and suspend until notified
Using Condition Objects

```java
Condition condition = mutex.newCondition();
...
mutex.lock();
try {
    while (!property)
        condition.await();
} catch (InterruptedException e) {
    ...
}
application specific response
...
Using Condition Objects

Condition condition = mutex.newCondition();
...
mutex.lock();
try {
    while (!property)
        condition.await();
} catch (InterruptedException e) {
    ...
}
...

happy: property must hold
Example: Blocking Queue

public class BlockingQueue<T> {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    final T[] items;
    int tail, head, count;

    public BlockingQueue(int capacity) {
        items = new T[capacity];
    }
    ...
}
public class BlockingQueue<T> {

    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    final T[] items;
    int tail, head, count;

    public BlockingQueue(int capacity) {
        items = new T[capacity];
    }

    ...

}
Example: Blocking Queue

```java
public class BlockingQueue<T> {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    final T[] items;
    int tail, head, count;

    public BlockingQueue(int capacity) {
        items = new T[capacity];
    }
    ...
}
```

condition to wait on if queue is full
Example: Blocking Queue

```java
public class BlockingQueue<T> {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    final T[] items;
    int tail, head, count;

    public BlockingQueue(int capacity) {
        items = new T[capacity];
    }

    // ... (continues)
}
```

Condition to wait on if queue is empty.
Example: Blocking Queue

```java
public class BlockingQueue<T> {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    final T[] items;
    int tail, head, count;

    public BlockingQueue(int capacity) {
        items = new T[capacity];
    }

    ...}
```
Blocking Queue: enqueue

```java
public void enq(T x) {
    lock.lock();
    try {
        while (count == items.length())
            notFull.await();
        items[tail] = x;
        if (++tail == items.length) tail = 0;
        ++count;
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
```
Blocking Queue: enqueue

```java
public void enq(T x) {
    lock.lock();
    try {
        while (count == items.length())
            notFull.await();
        items[tail] = x;
        if (++tail == items.length()) tail = 0;
        ++count;
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
```

wait until queue has space
public void enq(T x) {
    lock.lock();
    try {
        while (count == items.length())
            notFull.await();
        items[tail] = x;
        if (++tail == items.length()) tail = 0;
        ++count;
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
public void enq(T x) {
    lock.lock();
    try {
        while (count == items.length())
            notFull.await();
        items[tail] = x;
        if (++tail == items.length()) tail = 0;
        ++count;
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
public T deq() {
    lock.lock();
    try {
        while (count == 0)
            notEmpty.await();
        T x = items[head];
        if (++head == items.length) head = 0;
        --count;
        notFull.signal();
        return x;
    } finally {
        lock.unlock();
    }
}
public T deq() {
    lock.lock();
    try {
        while (count == 0)
            notEmpty.await();

        T x = items[head];
        if (++head == items.length) head = 0;
        --count;
        notFull.signal();
        return x;
    } finally {
        lock.unlock();
    }
}
public T deq() {
    lock.lock();
    try {
        while (count == 0)
            notEmpty.await();
        T x = items[head];
        if (++head == items.length) head = 0;
        --count;
        notFull.signal();
        return x;
    } finally {
        lock.unlock();
    }
}
public T dequeue() {
    lock.lock();
    try {
        while (count == 0)
            notEmpty.await();
        T x = items[head];
        if (++head == items.length) head = 0;
        --count;
        notFull.signal();
        return x;
    } finally {
        lock.unlock();
    }
}
Improved enqueue?

```java
public void enq(T x) {
    lock.lock();
    try {
        while (count == items.length())
            notFull.await();
        items[tail] = x;
        if (++tail == items.length) tail = 0;
        ++count;
        if (count == 1) notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
```
The Lost-Wakeup Problem

• Condition variables are inherently vulnerable to lost wakeups
  – one thread waits forever without realizing that its waiting condition has become true

• Programming practices
  – if in doubt, signal all waiting processes
  – specify a timeout when waiting
Reentrant Locks

• same thread can acquire the lock multiple times without blocking
• commonly used in OOP to handle reentrant calls to locked objects
Using Reentrant Locks

```java
public class AtomicArray<T> {
    final Lock lock = new ReentrantLock();
    ...

    public T getAndSet(int i, T v) {
        try {
            lock.lock();
            T old = get(i);
            set(i, v);
            return old;
        } finally {
            lock.unlock();
        }
    }

    public T get() {
        try {
            lock.lock();
            return item[i];
        } finally {
            lock.unlock();
        }
    }

    public void set(int i, T v) { ...
```

Using Reentrant Locks

public class AtomicArray<T> {
    final Lock lock = new ReentrantLock();
    ...

    public T getAndSet(int i, T v) {
        try {
            lock.lock();
            T old = get(i);
            set(i, v);
            return old;
        } finally {
            lock.unlock();
        }
    }

    public T get() {
        try {
            lock.lock();
            return item[i];
        } finally {
            lock.unlock();
        }
    }

    public void set(int i, T v) { ...
}
Our Own Reentrant Lock

```java
public class SimpleReentrantLock implements Lock{
    final Lock lock = new SimpleLock();
    final Condition cond = lock.newCondition();
    int owner, holdCount;

    public SimpleReentrantLock() {
        owner = holdCount = 0;
    }

    ...
}
```
public class SimpleReentrantLock implements Lock{

    final Lock lock = new SimpleLock();
    final Condition cond = lock.newCondition();
    int owner, holdCount;

    public SimpleReentrantLock() {
        owner = holdCount = 0;
    }

    ...
}

nonreentrant lock
Our Own Reentrant Lock

```java
class SimpleReentrantLock implements Lock{
    final Lock lock = new SimpleLock();
    final Condition cond = lock.newCondition();
    int owner, holdCount;

    public SimpleReentrantLock() {
        owner = holdCount = 0;
    }
    ...
}
```

class SimpleLock{
    public boolean tryLock() {
        // implementation
    }
}

class SimpleCondition{
    public boolean tryWait() {
        // implementation
    }
}

condition to wait on if lock is held by other thread
public class SimpleReentrantLock implements Lock{
    final Lock lock = new SimpleLock();
    final Condition cond = lock.newCondition();
    int owner, holdCount;

    public SimpleReentrantLock() {
        owner = holdCount = 0;
    }

    ...
}
public class SimpleReentrantLock implements Lock{
    final Lock lock = new SimpleLock();
    final Condition cond = lock.newCondition();
    int owner, holdCount;

    public SimpleReentrantLock() {
        owner = holdCount = 0;
    }

    ...
public void lock() {
    int me = ThreadID.get();
    lock.lock();
    try {
        if (owner == me) {
            ++holdCount;
            return;
        }
        while (holdCount != 0) condition.await();
        owner = me;
        holdCount = 1;
    } finally { lock.unlock() }
}
public void lock() {
    int me = ThreadID.get();
    lock.lock();
    try {
        if (owner == me) {
            ++holdCount;
            return;
        }
        while (holdCount != 0) condition.await();
        owner = me;
        holdCount = 1;
    } finally {
        lock.unlock();
    }
}

already holding the lock? then just increase counter
Our Own Reentrant Lock

```java
public void lock() {
    int me = ThreadID.get();
    lock.lock();
    try {
        if (owner == me) {
            ++holdCount;
            return;
        }
        while (holdCount != 0)
            condition.await();
        owner = me;
        holdCount = 1;
    } finally {
        lock.unlock();
    }
}
```

otherwise, wait until lock is free and then take ownership
public void unlock() {
    lock.lock();
    try {
        if (holdCount == 0 ||
            owner != ThreadID.get()) {
            throw new IllegalMonitorStateException();
        }
        if (--holdCount == 0) cond.signal();
    } finally {
        lock.unlock();
    }
}
public void unlock() {
    lock.lock();
    try {
        if (holdCount == 0 || owner != ThreadID.get()) {
            throw new IllegalMonitorStateException();
        }
        if (--holdCount == 0) cond.signal();
    } finally {
        lock.unlock();
    }
}

fail, if lock is released too often
public void unlock() {
    lock.lock();
    try {
        if (holdCount == 0 ||
            owner != ThreadID.get()) {
            throw new IllegalMonitorStateException();
        }
        if (--holdCount == 0) cond.signal();
    } finally {
        lock.unlock();
    }
}
Java’s built-in Monitors

- **synchronized** blocks and methods acquire and release an implicit reentrant lock
- access to an implicit condition object is provided via special methods
  - `wait()`
  - `notify()`
  - `notifyAll()`
public synchronized void enq(T x) {
    while (count == items.length())
        wait();
    items[tail] = x;
    if (++tail == items.length()) tail = 0;
    ++count;
    notifyAll();
}
Simplified Blocking Queue: dequeue

```java
public synchronized T deq() {
    while (count == 0)
        wait();
    T x = items[head];
    if (++head == items.length) head = 0;
    --count;
    notifyAll();
    return x;
}
```
Simplified Blocking Queue: dequeue

```java
public synchronized T deq() {
    while (count == 0)
        wait();
    T x = items[head];
    if (++head == items.length)
        head = 0;
    --count;
    notify();
    return x;
}
```

is notify enough?
Lost Wakeup in Simplified Queue with notify()

lock

queue state: []
capacity=1

enq(0) enq(1) enq(2)

waiting room
Lost Wakeup in Simplified Queue with notify()
Lost Wakeup in Simplified Queue with notify()
Lost Wakeup in Simplified Queue with notify()
Lost Wakeup in Simplified Queue with notify()
Lost Wakeup in Simplified Queue with notify()
Lost Wakeup in Simplified Queue with notify()
Lost Wakeup in Simplified Queue with notify()
Lost Wakeup in Simplified Queue with notify()

lock

queue state: [0]
capacity=1

deq()  deq()

enq(1)  enq(2)

waiting room
Lost Wakeup in Simplified Queue with notify()
Lost Wakeup in Simplified Queue with notify()
Lost Wakeup in Simplified Queue with notify()
Lost Wakeup in Simplified Queue with notify()
Lost Wakeup in Simplified Queue with notify()
Lost Wakeup in Simplified Queue with notify()
Lost Wakeup in Simplified Queue with notify()
Lost Wakeup in Simplified Queue with notify()
Lost Wakeup in Simplified Queue with notify()
Lost Wakeup in Simplified Queue with notify()

lock

queue state: [1]
capacity=1

deq()

deq()

waiting room

deq()
Lost Wakeup in Simplified Queue with notify()
Lost Wakeup in Simplified Queue with notify()

lock

queue state: [1]
capacity=1

remaining threads are stuck!
Readers-Writers Lock

• shared objects often have the property that their methods can be partitioned into
  – readers: return information about the object
  – writers: actually modify the object
• no need for readers to synchronize with each other
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
public SimpleReadWriteLock implements ReadWriteLock {
    int readers = 0;
    boolean writer = false;
    Lock lock = new ReentrantLock();
    Condition condition = lock.newCondition();
    Lock readLock = new ReadLock();
    Lock writeLock = new WriteLock();
    Lock readLock() { return readLock; }
    Lock writeLock() { return writeLock; }
    ...
}
public SimpleReadWriteLock implements ReadWriteLock {
    int readers = 0;
    boolean writer = false;
    Lock lock = new ReentrantLock();
    Condition condition = lock.newCondition();
    Lock readLock = new ReadLock();
    Lock writeLock = new WriteLock();
    Lock readLock() { return readLock; }
    Lock writeLock() { return writeLock; }
    ...
}
public SimpleReadWriteLock implements ReadWriteLock {
    int readers = 0;
    boolean writer = false;
    Lock lock = new ReentrantLock();
    Condition condition = lock.newCondition();
    Lock readLock = new ReadLock();
    Lock writeLock = new WriteLock();
    Lock readLock() { return readLock; }
    Lock writeLock() { return writeLock; }
    ...
}
public SimpleReadWriteLock implements ReadWriteLock {
    int readers = 0;
    boolean writer = false;
    Lock lock = new ReentrantLock();
    Condition condition = lock.newCondition();
    Lock readLock = new ReadLock();
    Lock writeLock = new WriteLock();
    Lock readLock() { return readLock; }
    Lock writeLock() { return writeLock; }
    ...
}
public SimpleReadWriteLock implements ReadWriteLock {
    int readers = 0;
    boolean writer = false;
    Lock lock = new ReentrantLock();
    Condition condition = lock.newCondition();
    Lock readLock = new ReadLock();
    Lock writeLock = new WriteLock();
    Lock readLock() { return readLock; }
    Lock writeLock() { return writeLock; }
    ...
}
Readers-Writers Lock

```java
public SimpleReadWriteLock implements ReadWriteLock {
    int readers = 0;
    boolean writer = false;
    Lock lock = new ReentrantLock();
    Condition condition = lock.newCondition();
    Lock readLock = new ReadLock();
    Lock writeLock = new WriteLock();

    Lock readLock() { return readLock; }
    Lock writeLock() { return writeLock; }
    ...
}

the actual read and write locks

(implemented by inner classes)
```
class ReadLock {
  public void lock() {
    lock.lock();
    try {
      while (writer) {
        condition.await();
      }
      readers++;
    } finally { lock.unlock(); }
  }
  ...
}
Inner ReadLock class

class ReadLock {
    public void lock() {
        lock.lock();
        try {
            while (writer) {
                condition.await();
            }
            readers++;
        } finally {
            lock.unlock();
        }
    }
...}
class ReadLock {
    public void lock() {
        lock.lock();
        try {
            while (writer) {
                condition.await();
            }
            readers++;
        } finally {
            lock.unlock();
        }
    }
}

increase the number of readers
Inner ReadLock class

class ReadLock {

    ... 
    
    public void unlock() {
        lock.lock();
        try {
            readers--;
            if (readers == 0)
                condition.signalAll();
        } finally { lock.unlock(); }
    }
}

Inner ReadLock class

class ReadLock {
    ... 
    public void unlock() {
        lock.lock();
        try {
            readers--;  // decrease the number of readers
            if (readers == 0)
                condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
}
Inner ReadLock class

class ReadLock {
    ...
    public void unlock() {
        lock.lock();
        try {
            readers--;
            if (readers == 0)
                condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
}
Inner WriteLock class

class WriteLock {
    public void lock() {
        lock.lock();
        try {
            while (readers > 0 || writer) {
                condition.await();
            }
            writer = true;
        } finally { lock.unlock(); }
    }
    ...
}
class WriteLock {
    public void lock() {
        lock.lock();
        try {
            while (readers > 0 || writer) {
                condition.await();
            }
            writer = true;
        } finally {
            lock.unlock();
        }
    }
    ...
}
Inner WriteLock class

class WriteLock {
    public void lock() {
        lock.lock();
        try {
            while (readers > 0 || writer) {
                condition.await();
            }
            writer = true;
        } finally {
            lock.unlock();
        }
    }
    ...
}
class WriteLock {
    ...

    public void unlock() {
        lock.lock();
        try {
            writer = false;
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
}
class WriteLock {
    ...
    public void unlock() {
        lock.lock();
        try {
            writer = false;
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
}
class WriteLock {
    ...
    public void unlock() {
        lock.lock();
        try {
            writer = false;
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
}
Fair Readers-Writers Lock

• Problem with SimpleReadWriteLock
  – usually readers are much more frequent than writers
  – writers may be locked out for a long time
• Idea: give priority to writers
public class FifoReadWriteLock implements ReadWriteLock {
    int readAcquires = 0;
    int readReleases = 0;
    boolean writer = false;
    Lock lock = new ReentrantLock(true);
    Condition condition = lock.newCondition();
    Lock readLock = new ReadLock();
    Lock writeLock = new WriteLock();
    Lock readLock() { return readLock; }
    Lock writeLock() { return writeLock; }
    ...
FIFO Readers-Writers Lock

```java
public class FifoReadWriteLock implements ReadWriteLock {
    int readAcquires = 0;
    int readReleases = 0;
    boolean writer = false;
    Lock lock = new ReentrantLock(true);
    Condition condition = lock.newCondition();
    Lock readLock = new ReadLock();
    Lock writeLock = new WriteLock();
    Lock readLock() { return readLock; }
    Lock writeLock() { return writeLock; }
    ...
}
```

count releases and acquires of readers separately
FIFO Readers-Writers Lock

```java
public FifoReadWriteLock implements ReadWriteLock {
    int readAcquires = 0;
    int readReleases = 0;
    boolean writer = false;
    Lock lock = new ReentrantLock(true);
    Condition condition = lock.newCondition();
    Lock readLock = new ReadLock();
    Lock writeLock = new WriteLock();
    Lock readLock() { return readLock; }
    Lock writeLock() { return writeLock; }
    ...
}
```
Inner ReadLock class

class ReadLock {
    public void lock() {
        lock.lock();
        try {
            while (writer) {
                condition.await();
            }
            readAcquires++;
        } finally {
            lock.unlock();
        }
    }
    ... }

class ReadLock {
    ...  
    public void unlock() {
        lock.lock();
        try {
            readReleases++;
            if (readReleases == ReadAcquires)
                condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
}

Inner ReadLock class
class WriteLock {
    public void lock() {
        lock.lock();
        try {
            while (writer) condition.await();
            writer = true;
            while (readAcquires != readReleases)
                condition.await();
        } finally {
            lock.unlock();
        }
    }
    ...
}
class WriteLock {
    public void lock() {
        lock.lock();
        try {
            while (writer) condition.await();
            writer = true;
            while (readAcquires != readReleases)
                condition.await();
        } finally {
            lock.unlock();
        }
    }
    ...  
}

first wait for writers to release the lock
Inner WriteLock class

class WriteLock {
    public void lock() {
        lock.lock();
        try {
            while (writer) condition.await();
            writer = true;
            while (readAcquires != readReleases)
                condition.await();
        } finally {
            lock.unlock();
        }
    } 

    ... 
}
Inner WriteLock class

class WriteLock {
    public void lock() {
        lock.lock();
        try {
            while (writer) condition.await();
            writer = true;
            while (readAcquires != readReleases)
                condition.await();
        } finally {
            lock.unlock();
        }
    }
    ...
}
Inner WriteLock class

class WriteLock {
    ...
    public void unlock() {
        writer = false;
        condition.signalAll();
    }
}