Semaphores, Locks & Conditions
Intrinsic vs. Explicit Locks

- Pre Java 5.0 only **intrinsic** mechanisms were available for coordinating access to shared data.
  - synchronized
  - volatile

How do synchronized and volatile differ in providing thread-safe access to shared data?

What are the limitations of using synchronized as a locking mechanism?
Intrinsic vs. Explicit Locks

• Synchronized – creates an intrinsic lock for accessing a section of code
• Volatile – variables declared volatile insure thread safe access by disabling optimizations or caching (memory barrier)

• Limitations of synchronized:
  – not possible to interrupt a thread waiting for a lock
  – thread wait forever attempting to acquire lock
  – lock must be released in the same block of code in which they are acquired
  – Lock an entire object rather than the parts we need.
  – Especially troublesome for collections
  – Inhibits performance
Intrinsic vs. Explicit Locks

• Synchronized – creates an intrinsic lock for accessing a section of code
• Volatile – variables declared volatile insure thread safe access by disabling optimizations or caching (memory barrier)

• Limitations of synchronized:
  – not possible to interrupt a thread waiting for a lock
  – thread wait forever attempting to acquire lock
  – lock must be released in the same block of code in which they are acquired
  – Lock an entire object rather than the parts we need.
  – Especially troublesome for collections
  – Inhibits performance
Semaphores and Locks

- Java 5+ added Semaphores, Locks, and Conditions
  - *Explicit* locking
  - Semaphores and Locks operate like synchronized, but:
    - Need not be nested
    - Can pass a lock from object to object within a thread
  - Conditions - wait for one of many possible states to arise
    - Condition associated with specific lock for atomicity control.
    - Conditions only available via factory in Lock
Semaphore

- Implements a general semaphore.
- Initialize with a number of permits.
- Permits can be acquired and released.
- Block on acquire if no permits remain (until one released).
- Interface abstract:

```java
public class Semaphore {
    public Semaphore( int permits );
    public Semaphore( int permits; boolean fair );
    public void acquire();
    public void acquire( int npermits );
    public void release();
    public void release( int npermits );

    // other methods exists – see java.util.concurrent.Semaphore
}
```
Fixed Resource Control Using Semaphores

class Resource { . . . }

class ResourcePool {
    private final int NR;
    private final Resource pool[];
    private final boolean used[];
    private final Semaphore available;

    public ResourcePool(int nr) {
        NR = nr;
        pool = new Resource[NR];
        used = new boolean[NR];
        available = new Semaphore(NR);
    }

    public Resource get() {
        available.acquire();
        return nextResource();
    }

    public synchronized void put(Resource r) {
        int index = find(r, pool);
        used[index] = false;
        available.release();
    }

    private synchronized Resource nextResource() { . . . }

    private int find(Resource r) { . . . }
}
The Lock Interface

- Timed or polled lock acquisition
- Locks must be released in finally block to prevent deadlock in the case of an exception thrown in guarded code
- Responsive to interruption – locking can be used within cancellable activities.

```java
public interface Lock {
    public void lock();
    public void unlock();
    public Condition newCondition();
    public void lockInterruptibly();
    public boolean tryLock();
    public boolean tryLock(long time, TimeUnit unit);
}
```
The Lock Interface

How does this differ from intrinsic (synchronized) locking?

• Intrinsic locking – deadlock is fatal (witness Dining Philosophers).
• Timed and poll locking offers probabilistic deadlock avoidance.
• Timed locks can cancel an activity early if not complete within a time period
java.util.concurrent.lock

• Interfaces
  – Lock
  – ReadWriteLock
  – Condition

• Provided Classes
  – ReentrantLock (Lock)
  – ReentrantReadWriteLock (ReadWriteLock)
    • ReentrantReadWriteLock . ReadLock (Lock w/o Conditions)
    • ReentrantReadWriteLock . WriteLock (Lock)
  – AbstractQueuedSynchronizer
    • AbstractQueuedSynchronizer . ConditionObject (Condition)
  – LockSupport
class X {
    private final Lock mylock = new ReentrantLock( fair );

    // Other class stuff . . .

    void m() {
        mylock.lock();  // block until lock is acquired
        try {
            // . . . method body
            // . . . method body
        } finally {
            mylock.unlock()
        }
    }
}
ReadWriteLock

• Built-in support for the readers / writers problem:
  - Assume a data structure which is read much more frequently than it is written.
  - No reason to forbid multiple concurrent reads.
  - But cannot overlap reads and writes.
  - Use distinct but related locks

```java
class ReadWriteLock {
    Lock readLock() ;
    Lock writeLock() ;
}
```
public class Example {
    private final ReadWriteLock rwl = new ReentrantReadWriteLock( fair );

    Reader Method Structure
    public void read() {
        rwl.readLock().lock();
        try {
            // read your heart out
            // other threads may be
            // reading as well
        } finally {
            rwl.readLock().unlock();
        }
    }

    Writer Method Structure
    public void write() {
        rwl.writeLock().lock();
        try {
            // Current thread can write
            // but no other thread is
            // reading or writing.
        } finally {
            rwl.writeLock().unlock();
        }
    }
}
Conditions

• Where a Lock replaces the use of synchronized methods and statements
• Condition replaces the use of the wait, notify, and notifyAll methods
• Each Condition is a distinct object on a Lock to give the effect of having multiple wait-sets per Lock object
• Condition instances are always created on a specific Lock and cannot be shared across Locks.
  — Example: `Condition waitOn = myLock.newCondition();`
class BoundedBuffer<E> {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put(E x) throws InterruptedException {  
        lock.lock();  
        try {  
            while (count == items.length)  
                notFull.await();  
            items[putptr] = x;  
            if (++putptr == items.length)  
                putptr = 0;

            ++count;  
            notEmpty.signal();
        } finally {  
            lock.unlock();
        }
    }

    public E take() throws InterruptedException {  
        lock.lock();  
        try {  
            while (count == 0)  
                notEmpty.await();  
            E x = (E) items[takeptr];  
            if (++takeptr == items.length)  
                takeptr = 0;

            --count;  
            notFull.signal();
            return x;
        } finally {  
            lock.unlock();
        }
    }
}
Locks Using Semaphores

class MyLock implements Lock {
    private final Semaphore mutex = new Semaphore(1);

    public void lock() {
        mutex.acquire();
    }

    public void unlock() {
        mutex.release();
    }

    public Condition newCondition() {
        return new MyCondition( this );
    }

    // Other lock methods
}

Conditions Using Semaphores

class MyCondition implements Condition {
    private int nwaiters = 0;
    private final MyLock myLock;
    private final Semaphore myWaitSema = new Semaphore(0);

    public MyCondition(MyLock lock) {
        myLock = lock;
    }

    public void await() {
        nwaiters++;
        myLock.unlock();
        myWaitSema.acquire();
        myLock.lock();
    }

    public void notify() {
        if (nwaiters > 0) {
            nwaiters--;
            myWaitSema.release();
        }
    }

    // Other condition methods
}
Performance & Fairness

- **Fair** locks – threads acquire a lock in order requested
- **Nonfair** locks – permits *barging*, running threads can jump ahead of threads waiting to acquire a lock
- Intrinsic locks (usually) implemented as nonfair
- ReentrantLock offers a constructor option.

**Why not just implement all locks as fair?**
- Fairness imposes a level of overhead that decreases performance – see JCIP p.283
- Requesting (barging) thread is already running and ready to use the lock, whereas thread that was next in line, but suspended, needs to become active again.
Performance & Fairness

• Fairness imposes a level of overhead that decreases performance – see JCIP p.283

• Requesting (barging) thread is already running and ready to use the lock, whereas thread that was next in line, but suspended, needs to become active again.
Intrinsic or Explicit?

• ReentrantLock or synchronized?
• As of Java 6 intrinsic locking performs on par with explicit locking in terms of scalability (number of threads contending for lock)
• Favor Reentrant only when advanced features (timing, polled, interruptible, fairness) is required.
• Favor synchronized for simplicity