Resources

• Doug Lea
  • Java Concurrency in Practice
• Maurice Herlihy & Nir Shavit
  • The Art of Multiprocessor Programming
Design forces

- Forces that must be addressed when designing a concurrent program
  - safety – “bad things don’t happen”
  - liveness – “good things eventually happen”
  - efficiency – performance
  - reusability – compositionality
Safety

• Want to prevent interference between concurrent activities
  • bad things do not happen

• Storage conflicts, race conditions
• Transaction violations
class Even {
    int n = 0;
    public int next() {
        // POST: n is always even
        ++n;
        ++n;
        return n;
    }
}

• Postcondition may fail due to storage conflicts
Liveness

- ensure activities make progress
  - good things eventually happen

- deadlock
- livelock
- fairness
Java concurrency support

- **Thread** class represents state of an independent activity
  - methods to start, sleep, etc
  - weak scheduling guarantees
  - each Thread is a member of a ThreadGroup for control and bookkeeping
  - code that runs in a thread defined by Runnable
- **synchronized** methods and blocks control atomicity via locks
- **monitor** methods in Object control suspension and resumption
  - wait, notify
Creating a thread

class MyThread extends Thread {
    public void run() {
        ...  
    }
}

Thread t = new MyThread();
t.start(); // start the thread

t.join(); // block until thread completes
Java locks

• Every Java object possesses one lock
• Manipulated only via `synchronized` keyword
• Class objects contain a lock used to protect statics
• Scalars like int are not Objects; must lock enclosing Object

• `synchronized` can be either a method or block qualifier
  
  ```java
  synchronized void f() { ... }
  ==
  void f() { synchronized (this) { ... } }
  ```
Synchronized methods

class Location {
    private double x, y;
    Location(double x, double y) { this.x = x; this.y = y; }
    synchronized double x() { return x; }
    synchronized double y() { return y; }
    synchronized void move(double dx, double dy) {
        x += dx;
        y += dy;
    }
}
Synchronized methods

class Location {
    private double x, y;
    Location(double x, double y) { this.x = x; this.y = y; }
    synchronized double x() { return x; }
    synchronized double y() { return y; }
    synchronized void move(double dx, double dy) {
        x += dx;
        y += dy;
    }
}
Java locks

- Java locks are **reentrant**
  - a thread hitting a **synchronized** block passes if the lock is free or if it already holds the lock, else blocks

- released after passing as many }s as {s for the lock—cannot forget to release the lock

- Synchronized also has side-effect of clearing locally cached values and forcing reloads from main memory
Locks and caching

• Locking generates traffic between threads and memory
  • Lock acquire forces reads from memory to cache
  • Lock release forces writes of cached updates to memory

• Without locking, there are no promises about if and when caches will be flushed or reloaded
  • without locking, a thread might never observe another thread writes

• can lead to unsafe, nonsensical execution
Locks and caching

• Modern architectures do not support sequential consistency

• cannot assume a write in one thread will be visible in another thread
Memory anomalies

• Should **acquire lock before use** of any field of any objects

• Should **release after update**

• If not, then might:
  • see **stale values** that do not reflect recent writes
  • see **inconsistent states** due to out-of-order writes
  • see **incompletely initialized** objects
class Cell {
    private long value;
    synchronized long get() { return value; }
    synchronized void set(long v) { value = v; }
    synchronized void swap(Cell o) {
        long t = get();
        long v = other.get();
        set(v);
        other.set(t);
    }
}

thread 1   thread 2
enter  
c1.swap

  t = get

enter  
c2.swap

  t = get

  v = c2.get

  v = c1.get

swap is a transactional method.
Can deadlock.
Java monitor methods

• Every Java object has a wait set

• wait()
  • suspends thread
  • adds thread to wait set of target object
  • release lock of target

• notify()
  • choose any thread T in wait set of target
  • let T acquire lock
  • resume T

• notifyAll()
  • same as notify(), but let all threads compete for lock
class X {
    synchronized void w() {
        before(); wait(); after();
    }
    synchronized void n() { notifyAll(); }
}

One possible trace for three threads accessing instance x:
before(); wait(); after();

T1 T2 T3
enter x.w()
before(); wait();
release lock

enter x.w()
before(); wait();
release lock

enter x.n()
notifyAll();

acquire lock

after();

x

waitset
NQueens

- Classic search problem

- Given an NxN chess board, place N queens on the board so that none can attack another in one move
- Can attack if in same row, column, or diagonal
class NQueens {
    public boolean isConsistent(int[] q, int n) {
        for (int i = 0; i < n; i++) {
            if (q[i] == q[n]) return false; // same column
            if (((q[i] - q[n]) == (n - i))) return false; // same major diagonal
            if (((q[n] - q[i]) == (n - i))) return false; // same minor diagonal
        }
        return true;
    }

    public void search(int N) {
        search(new int[N], 0);
    }

    public void search(int[] q, int n) {
        int N = q.length;
        if (n == N) System.out.println("found!");
        else {
            for (int i = 0; i < N; i++) {
                q[n] = i;
                if (isConsistent(q, n)) search(q, n+1);
            }
        }
    }
}

Tuesday, April 27, 2010
public void search(final int[] q, final int n) {
    final int N = q.length;
    if (n == N) System.out.println("found!");
    else {
        Thread[] t = new Thread[N];
        for (int i = 0; i < N; i++) {
            t[i] = new Thread() {
                public void run() {
                    int[] q2 = new int[N];
                    System.arraycopy(q2, 0, q, 0, N);
                    q2[n] = i;
                    if (isConsistent(q2, n)) search(q2, n+1);
                }
            };
            t[i].start();
        }
        for (int i = 0; i < N; i++) t[i].join();
    }
}
class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread() {
            public void run() {
                pong();
            }
        };
        t.run();
        System.out.println("ping");
    }
    static synchronized pong() {
        System.out.println("pong");
    }
}
class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread() {
            public void run() {
                pong();
            }
        };
        t.run();
        System.out.println("ping");
    }
    static synchronized pong() {
        System.out.println("pong");
    }
}
Fixed ping pong

class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread() {
            public void run() {
                pong();
            }
        };
        t.start();
        System.out.println("ping");
    }
    static synchronized pong() {
        System.out.println("pong");
    }
}
class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread() {
            public void run() {
                pong();
            }
        };
        t.start();
        System.out.println("ping");
    }
    static synchronized pong() {
        System.out.println("pong");
    }
}
Concurrency Utilities

• Concurrency building blocks added in Java5
  • library for concurrency analogous to collections for data

• Goal: enhance scalability, performance, readability, thread safety of Java programs
• Goal: beat C performance for server apps
Concurrency utils: JSR 166

- Task scheduling framework
  - Executor interface, replaces direct use of Thread
- Callable and Future
- Synchronizers
  - Semaphore, CyclicBarrier, CountDownLatch
- Concurrent Collections
  - BlockingQueue
- Lock
- Atomic
Executor

- No direct Thread invocation
  
  - myExecutor.execute(aRunnable)

  - **not** new Thread(aRunnable).start()
public interface Executor {
    void execute(Runnable cmd);
}

public interface ExecutorService extends Executor {
    ...
}

public class Executors {
    // factory methods
    static ExecutorService newFixedThreadPool(int n);
}

Executor pool = Executors.newFixedThreadPool(5);
pool.execute(runnable);
class WebService {
    public static void main(String[] a) {
        Executor pool = Executors.newFixedThreadPool(5);
        ServerSocket socket = new ServerSocket(8080);
        while (true) {
            final Socket conn = socket.accept();
            final Runnable task = new Runnable() {
                public void run() {
                    new Handler().process(conn);
                }
            };
            pool.execute(task);
        }
    }
}

class Handler { void process(Socket conn) { ... } }
ScheduledExecutorService

- `schedule()`
  - run once after a fixed delay
- `scheduleAtFixedRate()`
  - repeat at fixed period
- `scheduleWithFixedDelay()`
  - repeat at fixed period with fixed delay between runs

- Operations return `ScheduleFuture`
  - supports `cancel()` operation
Thread stop

• Thread class provides a `stop()` method.

  • *Never ever ever* call it!

• Stops the thread, unlocking all monitors held by the thread, leaving objects in an inconsistent state

• Better: use Executor and `shutdown()` or roll your own
Thread stop

class Th extends Thread {
    private boolean done;
    public void run() {
        while (! done) {
            ...
        }
    }
    public void shutdown() {
        done = true;
    }
}
class Th extends Thread {
    private boolean done;
    public void run() {
        while (true) {
            synchronized (this) {
                if (done) return;
            }
            ...
        }
    }
    public void shutdown() {
        synchronized (this) { done = true; }
    }
}
Shutdown

- ExecutorService supports operations for graceful shutdown
  
- void shutdown();
- List<Runnable> shutdownNow();
- boolean isShutdown();
- boolean isTerminated();
- ...

Tuesday, April 27, 2010
Synchronized

- Control access to a shared resource
- Ex: bank account

```java
synchronized double getBalance() {
    Account acct = verify(name, password);
    return acct.balance;
}
```

```java
synchronized double getBalance() {
    synchronized (this) {
        Account acct = verify(name, password);
        return acct.balance;
    }
}
```

```java
synchronized double getBalance() {
    Account acct = verify(name, password);
    synchronized (acct) {
        return acct.balance;
    }
}
```

Lock held for a long time.

Same as above.

Better. Only acct object is locked, and for a shorter time.
Locks

• Java provides basic locking via synchronized
• Good for many situations, but:
  • single monitor per object
  • not possible to interrupt thread waiting for lock
  • not possible to timeout when waiting for lock
  • block structure
    • difficult to acquire multiple locks

• Solution: Lock interface
interface Lock {
    void lock();
    void lockInterruptably();
    boolean tryLock();
    void unlock();
    Condition newCondition();
}
ReentrantLock

- Simplest concrete implementation of Lock
- Same semantics as synchronized, but with more features
- Generally better performance under contention than synchronized
- Multiple wait-sets supported using Condition interface

**But remember:** Lock is not automatically released
Lock example

Lock lock = new ReentrantLock();

public void accessResource() {
    lock.lock();
    try {
        ...
    } finally {
        lock.unlock();
    }
}
ReadWriteLock

• Has two locks controlling read and write access
• Multiple thread can acquire the read lock if no thread has the write lock
• Only one thread can acquire the write lock
• Better performance for read-mostly data access
ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
Lock rl = rwl.readLock();
Lock wl = rwl.writeLock();
ArrayList<String> data = new ArrayList<String>();

public String getData(int pos) {
    rl.lock();
    try { return data.get(pos); } 
    finally { rl.unlock(); } 
}

public String addData(int pos, String value) {
    wl.lock();
    try { data.add(pos, value); } 
    finally { wl.unlock(); } 
}
Synchronizers

- Semaphore
  - manages fixed size pool of resources
- CountDownLatch
  - one or more threads wait for a set of threads to complete an action
- CyclicBarrier
  - set of threads wait until all reach a specified point
- Exchanger
  - two threads reach a fixed point and exchange data
BlockingQueue

- Provides thread-safe way for multiple threads to manipulate collection

```java
interface BlockingQueue<E> {
    void put(E e);
    boolean offer(E o);
    E take();
    E poll();
    int drainTo(Collection<? super E> c);
    ...
}
```
BlockingQueue implementations

- ArrayBlockingQueue
  - bounded queue, backed by array, FIFO
- LinkedBlockingQueue
  - optionally bounded queue, backed by linked nodes, FIFO
- PriorityBlockingQueue
  - unbounded queue
  - uses comparator or natural ordering (Comparable) to determine the order of the queue
private ArrayBlockingQueue mq;

public Logger(BlockingQueue<String> q) { mq = q; }

public void run() {
    try {
        while (true) {
            String msg = mq.take(); // blocks if queue is empty
            ... // do something
        }
    } catch (InterruptedException e) { }
}
Blocking queue example

```java
private ArrayBlockingQueue<String> mq = new ArrayBlockingQueue<>(10);
Logger logger = new Logger(mq);

public void run() {
    String msg;
    try {
        while (true) {
            ... // do something
            mq.put(msg); // blocks if queue is full
        }
    } catch (InterruptedException e) { }
}
```
Concurrent collections

- ConcurrentMap (interface)
  - extends Map interface with atomic operations
- ConcurrentHashMap
  - fully concurrent retrieval
  - tunable concurrency for updates
    - constructor takes #expected concurrent threads
- ConcurrentLinkedQueue
  - unbounded, thread-safe queue, FIFO
- CopyOnWriteArrayList
  - optimized for frequent iteration, infrequent modification
ConcurrentHashMap vs. synchronized

**Figure 11.3.** Comparing scalability of Map implementations.
Summary

- New concurrency features very powerful
  - not as simple as synchronized, but much better performance
- Take some time to learn them
- Use them