Today

• Concurrency – mostly in Java

• Some good practices and design principles for using concurrency features of Java

• Later: language mechanisms that enforce some of these principles
Concurrent programming in Java

• Source: Doug Lea, SUNY Oswego
Concurrency
Concurrency

- Why?
  - availability – minimize response lag, maximize throughput
  - modeling – simulating autonomous objects, animation
  - parallelism – exploit multiprocessors, overlapping I/O
  - protection – isolating activities in threads
Concurrency

• Why?
  • availability – minimize response lag, maximize throughput
  • modeling – simulating autonomous objects, animation
  • parallelism – exploit multiprocessors, overlapping I/O
  • protection – isolating activities in threads

• Why not?
  • complexity – dealing with safety, liveness, composition
  • overhead – higher resource usage
Applications

- I/O bound tasks
- GUIs
- Hosting foreign code
- Server daemons
- Simulations
Concurrent programming

- Concurrency is a **conceptual** property of software

- Concurrent programs **might or might not**
  - operate over multiple CPUs
    - parallel programming – multiple CPUs
  - share access to resources
    - distributed programming – no sharing
Concurrent OO programming

- Always a part of OO
  - Simula67 was concurrent
- But not a factor in rise of OO in 80s
- Reemerged with Java

- Models:
  - active vs. passive objects
Active objects

• Every object has a single thread of control
• Actions are reactive responses to messages
  • may delay handling
• Messages are one-way

• Extensions:
  • asynchronous vs. synchronous messages
  • queueing
  • multicasting
Passive objects

- Objects are passive data
- A thread can invoke a method on an object, method code runs in same thread as the caller
Java

- Dumb active objects
  - `java.lang.Thread`
  - `run()` method

- Passive objects
  - participate in multiple threads
  - can protect themselves from conflicting activities (synchronized)
  - can create and control new threads
Hardware mappings

• Shared memory multiprocessing
  • all objects visible in same machine
  • procedural message passing (method calls)
  • many more threads than CPUs

• Remote message passing
  • object access objects via **remote references** or copying
  • must **marshal** (serialize) messages
Aspects of concurrent OO design

• Policies and protocol
  • system-wide design rules
• Object structures
  • design patterns, microarchitecture
• Coding techniques
  • idioms, workarounds, neat tricks
Forces affecting design

- Forces that must be addressed at each level of design
  - 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
Liveness

• ensure activities make progress
  • good things eventually happen

• deadlock
• livelock
• fairness
System = objects + activities

- Concurrent systems built from:
  - Objects
    - ADTs, monitors, remote objects, components, ...
    - group by structure, role
    - focus on safety
  - Activities
    - messages, call chains, threads, sessions, transactions, workflows, ...
    - group by origin, function
    - focus on liveness
Safe objects

• Perform method actions only when in consistent states

• Impossible to predict consequences of actions attempted when objects are temporarily inconsistent
  • read/write, write/write conflicts
  • invariant failures
  • random-looking externally visible behavior

• Must balance with liveness goals
  • clients want simultaneous access to services
State inconsistency examples

- Draw a figure while moving
  - could draw at old x, new y
  - could draw at location objects was never at
- Withdraw from bank account during a transfer
  - could overdraw account
  - money could disappear
- Read storage location while being written
  - could read some old bytes, some new
  - could get a nonsense value
Live activities

- Every activity should make progress toward completion
  - every called method should eventually execute

- Related to efficiency
  - every called method should execute ASAP

- Activity might not complete if
  - object does not accept message
  - method blocks waiting for condition that never happens
  - insufficient or unfairly shared resources
  - failures
Extremes

- Safety first
  - Ensure each class is safe, then try to improve liveness
  - => slow, deadlock prone code

- Liveness first
  - => buggy code full of races
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
Designing objects for concurrency

- Patterns for safely managing state
- immutability
  - avoid interference by avoiding change
- locking
  - guarantee exclusive access
- state dependence
- containment
Immutability

• Avoid interference by avoiding change
• Immutable objects never change state
• Actions on immutable objects are always safe and live

• Applications:
  • objects representing values
  • objects providing stateless services
  • pure functional programming style

• Methods dealing with immutable aspects of state do not require locking
Stateless server objects

class StatelessAdder {
  int addOne(int a) { return a + 1; } 
}

• No special concurrency concerns
  • no per-instance state => no storage conflicts
  • no rep invariants => no invariant failures
  • any number of instances of addOne can run concurrently => no liveness problems
  • no interaction with other objects
Freezing state on construction

class ImmutableAdder {
    private final int offset;
    ImmutableAdder(int x) { offset = x; }
    int add(int i) { return i + offset; }
}

• No safety or liveness concerns
• Java finals enforce most senses of immutability
• Often uses for values:
  • java.lang.String, java.lang.Integer, java.awt.Color
  • not: java.awt.Point
Locking

• Locking is a simple **message accept** mechanism
  • acquire object lock on entry to method, release on return
• Precludes storage conflicts, invariant failures
  • can guarantee atomicity of methods
• But: introduces potential liveness failures
  • deadlock, lockouts
• Applications:
  • fully synchronized (atomic) objects
  • most other reusable objects with mutable state
Synchronized methods

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

- Java locks are **reentrant**
  - a thread hitting synchronized 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
Storage conflicts

class Even {
    int n = 0;
    public int next() {
        // POST: next is always even
        ++n;
        ++n;
        return n;
    }
}

• Postcondition may fail due to storage conflicts

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read 0</td>
<td></td>
</tr>
<tr>
<td>write 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read 1</td>
</tr>
<tr>
<td></td>
<td>write 2</td>
</tr>
<tr>
<td>read 2</td>
<td>read 2</td>
</tr>
<tr>
<td></td>
<td>write 3</td>
</tr>
<tr>
<td></td>
<td>return 3</td>
</tr>
<tr>
<td>write 3</td>
<td></td>
</tr>
<tr>
<td>return 3</td>
<td></td>
</tr>
</tbody>
</table>
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
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
Fully synchronized objects

- Objects of classes in which **all** methods are synchronized
- Always safe, but not always live or efficient
- Only processes one request at a time
  - all methods locally sequential

- Constraints:
  - **all** methods must be synchronized: unsynchronized methods execute even if another thread holds the lock
  - no public variables or encapsulation violations
  - methods must be suspend or infinite loop
  - consistent state must be maintained on both return and throw

Tuesday, April 27, 2010
Deadlock

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);
    }
}

swap is a transactional method. Can deadlock.
Lock precedence

- Can prevent deadlock in transactional methods via resource-ordering
- Idea: use Java hash codes

```java
void swap(Cell other) {
    if (other == this) return; // alias check
    Cell fst = this; // order via hash codes Cell
    Cell snd = other;
    if (fst.hashCode() > snd.hashCode()) {
        fst = other; snd = this;
    }
    synchronized(fst) {
        synchronized (snd) {
            long t = fst.value;
            fst.value = snd.value;
            snd.value = t;
        }
    }
}
```
Holding locks during downstream calls

```java
class Server {
    double state;
    Helper helper;
    public synchronized void serve() {
        state = illegalValue;
        helper.help();
        state = legalValue;
    }
}
```

- Safety: what happens if help throws exception
- Liveness: what happens if help causes deadlock
- Availability: cannot accept new calls to serve during help

- Rule of thumb (with many variants and exceptions):
  - always lock when updating state
  - never lock when sending a message (calling a function)
  - Redesign methods to avoid holding locks during downstream calls
class Queue {
    int sz;
    synchronized void put(Object o) { ... sz++; }
    synchronized Object take() { ... sz--; }
    int size() { return sz; }
}

• Should size be synchronized?
  • Pro: prevents client from obtaining stale values
  • Pro: ensures transient values never returned
  • Con: what would client ever do with the value?
State dependence

- Two aspects of action control:
  - a **message** from a client
  - the **internal state** of the host

- Design steps:
  - choose policies for dealing with actions that can succeed only if object is in a particular state
  - Design interfaces to reflect policy
  - Ensure objects able to assess state implement policy
Examples

• Operations on collections, streams, databases
  • remove element from empty queue
• Operations on objects maintaining constrained values
  • withdraw money from empty bank account
Balking

- Check state on method entry
  - do not change state while checking it!
- Exit immediately if not in right state
  - exception or special return value
  - let client handle failure

- Simplest policy for fully synchronized objects
Example

class BalkingCounter {
    // INV: MIN <= count <= MAX
    long count = MIN;
    synchronized long value() { return count; }
    synchronized void inc() {
        if (count >= MAX) throw new Fail();
        count++;
    }
    synchronized void dec() {
        if (count <= MIN) throw new Fail();
        count--;
    }
}

Tuesday, April 27, 2010
Example: traversal

• Balking iterators
  • throw ConcurrentModificationException if collection changes between calls to Iterator.next()
  • used in Java collections

• Snapshot iterators
  • make immutable copy of base collection
  • expensive!

• Indexed traversal
  • clients externally synchronized when necessary

• Synchronized aggregates
  • provide apply-to-all methods in collection class
  • deadlock prone
Guarding

- Generalization of locking for state-dependent actions
  - locked: wait until ready
  - guarded: wait until state holds

- Check state upon entry
  - if not in right state, block

- Liveness concerns
  - relies on another thread to update state
  - useless in sequential programs
Guarding via busy wait

```c
void inc() {
    while (count >= MAX) /* spin */ ;
    count++;
}
```

• Unsafe
• Wasteful
• But useful for **latches** (once true, never becomes false)
Guarding via suspension

synchronized void inc() {
    while (count >= MAX) wait();
    count++;
    notifyAll();
}
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
after();

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

enter x.n()
notifyAll();
Containment

- Use containment to guarantee exclusive access to internal objects
- Container controls visibility, provides concurrency control

Applications:
- Wrap unsafe sequential code
- Eliminate locking for internal objects
- Specialized synchronization policies
- Applying different policies to the same mechanisms
class Pixel {
    private final java.awt.Point pt; // Note: mutable!
    Pixel(int x, int y) { pt = new Point(x,y); }
    synchronized void point() {
        return new Point(pt.x, pt.y);
    }
    synchronized void move(int dx, int dy) {
        pt.x += dx; pt.y += dy;
    }
}

Note: reference to Point is immutable (final), but fields of Point are mutable and public, so unsafe without synchronization

Note: return copies of inner objects to avoid encapsulation violations
Implementing containment

- Strict containment creates **islands** of isolated objects
  - allows inner code to run faster (less locking)
  - applies recursively

- Inner code must be **communication-closed**
  - no unprotected calls into or out of the island

- Object objects **must never leak identities** of inner objects

- Outermost objects must **synchronize**
Nested monitors

class Whole {
    class Part {
        synchronized void await() { while (! cond) wait(); }  
        synchronized void signal() { cond = true; notifyAll(); }  
    }
    private final Part part = new Part();
    synchronized void work() { part.await(); }
    synchronized void go() { part.signal(); }
}

If Whole.work called, Part.await releases lock on part, but Whole is still locked.

Whole.go can never be entered to call Part.signal

Nested monitor lockout
Concurrency in Java

- Threads
- One lock per object
  - use synchronized to acquire
- Monitor methods
  - wait, notify