Concurrent Programming in Java

Doug Lea
State University of New York at Oswego
dl@cs.oswego.edu
http://gee.cs.oswego.edu
Topics

Concurrency

Models, design forces, Java

Designing objects for concurrency

Immutability, locking, state dependence, containment, splitting

Introducing concurrency into applications

Autonomous loops, oneway messages, interactive messages, cancellation

Concurrent application architectures

Flow, parallelism, layering

Libraries

Using, building, and documenting reusable concurrent classes
About These Slides ...

Some slides are based on joint presentations with David Holmes, Macquarie University, Sydney Australia.

More extensive coverage of most topics can be found in the book

*Concurrent Programming in Java*, Addison-Wesley

and the online supplement

http://gee.cs.oswego.edu/dl/cpj

The printed slides contain much more material than can be covered in a tutorial. They include extra background, examples, and extensions. They are not always in presentation order.

Java code examples often omit qualifiers, imports, etc for space reasons. Full versions of most examples are available from the CPJ online supplement.

None of this material should be construed as official Sun information.

Java is a trademark of Sun Microsystems, Inc.
Concurrent Programming in Java

Concurrency

Why?

Availability
Minimize response lag, maximize throughput

Modelling
Simulating autonomous objects, animation

Parallelism
Exploiting multiprocessors, overlapping I/O

Protection
Isolating activities in threads

Why Not?

Complexity
Dealing with safety, liveness, composition

Overhead
Higher resource usage
Common Applications

I/O-bound tasks

- Concurrently access web pages, databases, sockets ...

GUIs

- Concurrently handle events, screen updates

Hosting foreign code

- Concurrently run applets, JavaBeans, ...

Server Daemons

- Concurrently service multiple client requests

Simulations

- Concurrently simulate multiple real objects

Common examples

- Web browsers, web services, database servers, programming development tools, decision support tools
Concurrency is a *conceptual* property of software.

Concurrent programs *might or might not*:

<table>
<thead>
<tr>
<th>Operate across multiple CPUs symmetric multiprocessor (SMPs), clusters, special-purpose architectures, ...</th>
<th>Share access to resources objects, memory, displays, file descriptors, sockets, ...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parallel</strong> programming mainly deals with mapping software to multiple CPUs to improve performance.</td>
<td><strong>Distributed</strong> programming mainly deals with concurrent programs that do <em>NOT</em> share system resources.</td>
</tr>
</tbody>
</table>

*Concurrent programming* mainly deals with concepts and techniques that apply even if not parallel or distributed.

- Threads and related constructs run on any Java platform
- This tutorial doesn’t dwell much on issues *specific* to parallelism and distribution.
Concurrent Object-Oriented Programming

Concurrency has always been a part of OOP (since Simula67)
- Not a factor in wide-scale embrace of OOP (late 1980s)
- Recent re-emergence, partly due to Java

Concurrent OO programming differs from ...

**Sequential OO programming**
- Adds focus on safety and liveness
- But uses and extends common design patterns

**Single-threaded Event-based programming (as in GUIs)**
- Adds potential for multiple events occurring at same time
- But uses and extends common messaging strategies

**Multithreaded systems programming**
- Adds encapsulation, modularity
- But uses and extends efficient implementations
Object Models

Models describe how to think about objects (formally or informally)

Common features
- Classes, state, references, methods, identity, constraints
- Encapsulation
  - Separation between the insides and outsides of objects

Four basic computational operations
- Accept a message
- Update local state
- Send a message
- Create a new object

Models differ in rules for these operations. Two main categories:
- Active vs Passive
- Concurrent models include features of both
- Lead to uniquely concurrent OO design patterns
Every object has a single thread of control (like a process) so can do only one thing at a time.

Most actions are reactive responses to messages from objects

- But actions may also be autonomous
- But need not act on message immediately upon receiving it

All messages are oneway. Other protocols can be layered on.

Many extensions and choices of detailed semantics

- Asynchronous vs synchronous messaging, queueing, pre-emption and internal concurrency, multicast channels, ...
Passive Object Models

In sequential programs, only the single `Program` object is active

- Passive objects serve as the program’s data

In single-threaded Java, `Program` is the JVM (interpreter)

- Sequentially simulates the objects comprising the program
- All internal communication based on procedure calls
Concurrent Object Models

Mixtures of active and passive objects

Normally many fewer threads than passive objects

<table>
<thead>
<tr>
<th>Dumber Active Objects</th>
<th>Smarter Passive Objects</th>
</tr>
</thead>
</table>
| • Can perform only one activity  
  — in Java, `run()`  
| • May simultaneously participate in multiple threads  
| • Share most resources with other threads  
| • Protect themselves from engaging in conflicting activities  
| • Require scheduling in order to coexist  
| • Communicate with objects participating in other threads  
| • Initiate and control new threads  

Hardware Mappings

**Shared memory multiprocessing**

- All objects visible in same (virtual) machine
- Can use procedural message passing
- Usually many more threads than CPUs

**Remote message passing**

- Only access objects via Remote references or copying
- Must marshal (serialize) messages

**Mixed** models including database mediation ("three tier")
Vertical Objects

Most OO systems and applications operate at multiple levels

Objects at each level manipulate, manage, and coordinate lower-level ground objects as resources.

Once considered an arcane systems design principle.
But now applies to most applications

Concurrency

• Thread-objects interpret passive objects

Networking and Distribution

• Server-objects pass around resources

Persistence and Databases

• Database-objects manage states of ground objects

Component Frameworks

• Design tools build applications from JavaBeans, etc

Layered Applications

• Design patterns based on reflection, interpretation, ...
Design Forces

Three main aspects of concurrent OO design

<table>
<thead>
<tr>
<th>Policies &amp; Protocol</th>
<th>Object structures</th>
<th>Coding techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>System-wide design rules</td>
<td>Design patterns, microarchitecture</td>
<td>Idioms, neat tricks, workarounds</td>
</tr>
</tbody>
</table>

Four main kinds of forces that must be addressed at each level

- Safety — Integrity requirements
- Liveness — Progress requirements
- Efficiency — Performance requirements
- Reusability — Compositional requirements
Systems = Objects + Activities

Objects

- ADTs, aggregate components, JavaBeans, monitors, business objects, remote RMI objects, subsystems, ...
- May be grouped according to structure, role, ...
- Usable across multiple activities — focus on SAFETY

Activities

- Messages, call chains, threads, sessions, scenarios, scripts, workflows, use cases, transactions, data flows, mobile computations, ...
- May be grouped according to origin, function, ...
- Span multiple objects — focus on LIVENESS
Safe Objects

Perform method actions **only** when in consistent states

Usually impossible to predict consequences of actions attempted when objects are in temporarily inconsistent states

- Read/write and write/write conflicts
- Invariant failures
- Random-looking externally visible behavior

Must balance with liveness goals

- Clients want simultaneous access to services
State Inconsistency Examples

A figure is drawn while it is in the midst of being moved

- Could draw at new X-value, old Y-value
- Draws at location that figure never was at

Withdraw from bank account while it is the midst of a transfer

- Could overdraw account
- Could lose money

A storage location is read in the midst of being written

- Could result in reading some old bytes and some new bytes
- Normally, a nonsense value
Live Activities

Every activity should progress toward completion

- Every called method should eventually execute

Related to efficiency

- Every called method should execute as soon as possible

An activity might not complete if

- An object does not accept a message
- A method blocks waiting for an event, message or condition that should be, but isn’t produced by another activity
- Insufficient or unfairly scheduled resources
- Failures and errors of various kinds
Design Dualities

Two extreme approaches:

<table>
<thead>
<tr>
<th>Safety-first</th>
<th>Liveness-first</th>
</tr>
</thead>
</table>
| Ensure that each class is safe, then try to improve liveness as optimization measure.  
  • Characteristic of top-down OO Design  
  • Can result in slow, deadlock-prone code | Design live ground-level code, then try to layer on safety features such as locking and guarding.  
  • Characteristic of multithreaded systems programming  
  • Can result in buggy code full of races |

Effective, practical, middle-out approaches combine these.

For example, iteratively improving initial designs to be safe and live across different contexts.
Guaranteeing Safety

“Nothing bad ever happens”

Concurrent safety is an extended sense of type safety

- Adds a temporal dimension
- Not completely enforceable by compilers

<table>
<thead>
<tr>
<th>Low-level view</th>
<th>High-level view</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Bits are never misinterpreted</td>
<td>- Objects are accessible only when in consistent states</td>
</tr>
<tr>
<td>- Protect against storage conflicts on memory cells</td>
<td>- Objects must maintain state and representation invariants</td>
</tr>
<tr>
<td>- read/write and</td>
<td>- Presents subclass obligations</td>
</tr>
<tr>
<td>- write/write conflicts</td>
<td></td>
</tr>
</tbody>
</table>
Guaranteeing Liveness

“Something eventually happens”

Availability

• Avoiding unnecessary blocking

Progress

• Avoiding resource contention among activities
• Avoiding deadlocks and lockouts
• Avoiding unfair scheduling
• Designing for fault tolerance, convergence, stability

Citizenship

• Minimizing computational demands of sets of activities

Protection

• Avoiding contention with other programs
• Preventing denial of service attacks
• Preventing stoppage by external agents
Concurrency and Efficiency

Concurrency can be expensive

- Performance profiles may vary across platforms

Resources

- Threads, Locks, Monitors

Computation

- Construction, finalization overhead for resources
- Synchronization, context switching, scheduling overhead

Communication

- Interaction overhead for threads mapped to different CPUs
- Caching and locality effects

Algorithmic efficiency

- Cannot use some fast but unsafe sequential algorithms

Paying for tunability and extensibility

- Reduces opportunities to optimize for special cases
Concurrent Programming in Java

Concurrency and Reusability

Added Complexity

- More stringent correctness criteria than sequential code
  - Usually not automatically statically checkable
- Nondeterminism impedes debuggability, understandability

Added Context Dependence (coupling)

- Components only safe/live when used in intended contexts
  - Need for documentation
- Can be difficult to extend via subclassing
  - “Inheritance anomalies”
- Can be difficult to compose
  - Clashes among concurrency control techniques
## Reuse and Design Policies

Think locally. Act globally.

### Example design policy domains

<table>
<thead>
<tr>
<th>State-dependence</th>
<th>Service availability</th>
<th>Flow constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>What to do if a request logically cannot be performed</td>
<td>Constraints on concurrent access to methods</td>
<td>Establishing message directionality and layering rules</td>
</tr>
</tbody>
</table>

### Combat complexity

- High-level design rules and architectural constraints avoid inconsistent case-by-case decisions
- Policy choices are rarely “optimal”, but often religiously believed in anyway.

### Maintain openness

- Accommodate any component that obeys a given policy
- Fail but don’t break if they do not obey policy
Three Approaches to Reusability

Patterns  Reusing design knowledge
- Record best practices, refine them to essences
- Analyze for safety, liveness, efficiency, extensibility, etc
- Provide recipes for construction

Frameworks  Reusing policies and protocols
- Create interfaces and classes that establish policy choices for a suite of applications
- Provide utilities and support classes
- Mainly use by creating application-dependent (sub)classes

Libraries  Reusing code
- Create interfaces that apply in many contexts
- Provide high-quality implementations
- Allow others to create alternative implementations
Java Overview

Core Java is a relatively small, boring object-oriented language

Main differences from Smalltalk:

• Static typing
• Support for primitive data types (int, float, etc)
• C-based syntax

Main differences from C++:

• Run-time safety via Virtual Machine
  — No insecure low-level operations
  — Garbage collection
• Entirely class-based: No globals
• Relative simplicity: No multiple inheritance, etc
• Object-based implementations of Array, String, Class, etc
• Large predefined class library: AWT, Applets, net, etc
Java Features

Java solves some software development problems

Packaging: Objects, classes, components, packages
Portability: Bytecodes, unicode, transports
Extensibility: Subclassing, interfaces, class loaders
Safety: Virtual machine, GC, verifiers
Libraries: java.* packages
Ubiquity: Run almost anywhere

But new challenges stem from new aspects of programming:

Concurrency: Threads, locks, ...
Distribution: RMI, CORBA, ...
Persistence: Serialization, JDBC, ...
Security: Security managers, Domains, ...
## Basic Java Constructs

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>Descriptions of object features</td>
</tr>
<tr>
<td>Instance variables</td>
<td>Fields representing object state</td>
</tr>
<tr>
<td>Methods</td>
<td>Encapsulated procedures</td>
</tr>
<tr>
<td>Statics</td>
<td>Per-class variables and methods</td>
</tr>
<tr>
<td>Constructors</td>
<td>Operations performed upon object creation</td>
</tr>
<tr>
<td>Interfaces</td>
<td>Sets of methods implemented by any class</td>
</tr>
<tr>
<td>Subclasses</td>
<td>Single inheritance from class <code>Object</code></td>
</tr>
<tr>
<td>Inner classes</td>
<td>Classes within other classes and methods</td>
</tr>
<tr>
<td>Packages</td>
<td>Namespaces for organizing sets of classes</td>
</tr>
<tr>
<td>Visibility control</td>
<td>private, public, protected, per-package</td>
</tr>
<tr>
<td>Qualifiers</td>
<td>Semantic control: final, abstract, etc</td>
</tr>
<tr>
<td>Statements</td>
<td>Nearly the same as in C/C++</td>
</tr>
<tr>
<td>Exceptions</td>
<td>Throw/catch control upon failure</td>
</tr>
<tr>
<td>Primitive types</td>
<td>byte, short, int, long, float, char, boolean</td>
</tr>
</tbody>
</table>
Particle Applet

```java
import java.awt.*;
import java.applet.*;
public class ParticleApplet extends Applet {
    public void init() {
        add(new ParticleCanvas(10));
    }
}

class ParticleCanvas extends Canvas {
    Particle[] particles;
    ParticleCanvas(int nparticles) {
        setSize(new Dimension(100, 100));
        particles = new Particle[nparticles];
        for (int i = 0; i < particles.length; ++i) {
            particles[i] = new Particle(this);
            new Thread(particles[i]).start();
        }
    }
    public void paint(Graphics g) {
        for (int i = 0; i < particles.length; ++i)
            particles[i].draw(g);
    }
} // (needs lots of embellishing to look nice)
```
public class Particle implements Runnable {
    private int x = 0, y = 0;
    private Canvas canvas;
    public Particle(Canvas host) { canvas = host; }

    synchronized void moveRandomly() {
        x += (int) (((Math.random() - 0.5) * 5);    y += (int) (((Math.random() - 0.5) * 5);
    }

    public void draw(Graphics g) {
        int lx, ly;
        synchronized (this) { lx = x; ly = y; }
        g.drawRect(lx, ly, 10, 10);
    }

    public void run() {
        for(;;) {
            moveRandomly();
            canvas.repaint();
            try { Thread.sleep((int)(Math.random()*10);}      
            catch (InterruptedException e) { return; }
        }
    }
}
Java Concurrency Support

**Thread** class represents state of an independent activity

- Methods to start, sleep, etc
- Very weak guarantees about control and scheduling
- Each **Thread** is a member of a **ThreadGroup** that is used for access control and bookkeeping
- Code executed in threads defined in classes implementing:

```
interface Runnable { public void run(); }
```

**synchronized** methods and blocks control atomicity via locks

- Java automates local read/write atomicity of storage and access of values of type `byte`, `char`, `short`, `int`, `float`, and `Object` references, but not `double` and `long`
- **synchronized** statement also ensures cache flush/reload
- **volatile** keyword controls per-variable flush/reload

**Monitor** methods in class `Object` control suspension and resumption:

- `wait()`, `wait(ms)`, `notify()`, `notifyAll()`
Class Thread

Constructors

```
Thread(Runnable r) constructs so run() calls r.run()
```

— Other versions allow names, ThreadGroup placement

Principal methods

```
start() activates run() then returns to caller

isAlive() returns true if started but not stopped

join() waits for termination (optional timeout)

interrupt() breaks out of wait, sleep, or join

isInterrupted() returns interruption state

getPriority() returns current scheduling priority

setPriority(int priorityFromONEtoTEN) sets it
```

Static methods that can only be applied to current thread

```
currentThread() reveals current thread

sleep(ms) suspends for (at least) ms milliseconds

interrupted() returns and clears interruption status
```
Designing Objects for Concurrency

Patterns for safely representing and managing state

Immutability
  • Avoiding interference by avoiding change

Locking
  • Guaranteeing exclusive access

State dependence
  • What to do when you can’t do anything

Containment
  • Hiding internal objects

Splitting
  • Separating independent aspects of objects and locks
Immutability

Synopsis

• Avoid interference by avoiding change

• Immutable objects never change state

• Actions on immutable objects are always safe and live

Applications

• Objects representing values
  — Closed Abstract Data Types
  — Objects maintaining state representations for others
    — Whenever object identity does not matter

• Objects providing stateless services

• Pure functional programming style
Stateless Service Objects

class StatelessAdder {
    int addOne(int i) { return i + 1; }
    int addTwo(int i) { return i + 2; }
}

There are no special concurrency concerns:

• There is no per-instance state
  → No storage conflicts

• No representational invariants
  → No invariant failures

• Any number of instances of addOne and/or addTwo can safely execute at the same time. There is no need to preclude this.
  → No liveness problems

• The methods do not interact with any other objects.
  → No concurrent protocol design
Freezing State upon Construction

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

Still no safety or liveness concerns

Java (blank) finals enforce most senses of immutablity

• Don’t cover cases where objects eventually latch into values that they never change from

Immutability is often used for closed Abstract Data Types in Java

• java.lang.String
• java.lang.Integer
• java.awt.Color
• But not java.awt.Point or other AWT graphical representation classes (A design error?)
Applications of Immutability

Immutable references to mutable objects

```java
class Relay {
    private final Server delegate;
    Relay(Server s) { delegate = s; }
    void serve() { delegate.serve(); }
}
```

Partial immutability

```java
class FixedList {
    private final FixedList next; // immutable
    FixedList(FixedList nxt) { next = nxt; }
    FixedList successor() { return next; }

    private Object elem = null; // mutable
    synchronized Object get() { return elem; }
    synchronized void set(Object x) { elem = x; }
}
```
Locking

Locking is a simple **message accept** mechanism

- Acquire object lock on entry to method, release on return

Precludes storage conflicts and invariant failures

- Can be used to guarantee atomicity of methods

Introduces potential liveness failures

- Deadlock, lockouts

Applications

- Fully synchronized (atomic) objects
- Most other reusable objects with mutable state
Synchronized Method Example

class Location {

    private double x_, y_

    Location(double x, double y) { x_ = x; y_ = y; }

    synchronized double x() { return x_; }

    double y() {
        synchronized (this) {
            return y_;
        }
    }

    synchronized void moveBy(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 so can only be locked via their enclosing objects

Synchronized can be either method or block qualifier

```java
synchronized void f() { body; } is equivalent to:

void f() { synchronized(this) { body; } }
```

Java locks are reentrant

- A thread hitting `synchronized` passes if the lock is free or it already possesses the lock, else waits
- Released after passing as many }'s as { 's for the lock — cannot forget to release lock

Synchronized also has the side-effect of clearing locally cached values and forcing reloads from main storage
class Even {
    int n = 0;
    public int next(){ // POST?: next is always even
        ++n;
        ++n;
        return n;
    }
}

Postcondition may fail due to storage conflicts. For example, one possible execution trace when \( n \) starts off at 0 is:

Thread 1                  Thread 2
read  0                   read  1
write 1                    write 2
read  2                    read  2
write 3                    write 3
return 3

Declaring \texttt{next} method as \texttt{synchronized} precludes conflicting traces, as long as all other methods accessing \( n \) are also synchronized.
Locking generates messages between threads and memory

- Lock acquisition forces reads from memory to thread 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

- Can lead to unsafe execution
- Can lead to nonsensical execution
Memory Anomalies

Should **acquire lock before use** of any field of any object, and **release after update**

If not, the following are possible:

- Seeing **stale** values that do not reflect recent updates
- Seeing **inconsistent states** due to out-of-order writes during flushes from thread caches
- Seeing **incompletely initialized** new objects

Can declare **volatile** fields to force per-variable load/flush.

- Has very limited utility.
- **volatile** never usefully applies to reference variables
  - The referenced object is not necessarily loaded/flushed, just the reference itself.
  - Instead, should use synchronization-based constructions
Fully Synchronized Objects

Objects of classes in which **all methods are synchronized**

- Always safe, but not always live or efficient

Only process one request at a time

- All methods are locally sequential

Accept new messages only when **ready**

  - No other thread holds lock
  - Not engaged in another activity

- But methods may make self-calls to other methods during same activity without blocking (due to reentrancy)

Constraints

- **All** methods must be synchronized: Java unsynchronized methods execute even when lock held.
- No public variables or other encapsulation violations
- Methods must not suspend or infinitely loop
- Re-establish consistent state after exceptions
Deadlock

```java
class Cell {
    private long value_;

    synchronized long getValue() { return value_;;
    synchronized void setValue(long v) {value_ = v;}

    synchronized void swapValue(Cell other) {
        long t = getValue();
        long v = other.getValue();
        setValue(v);
        other.setValue(t);
    }
}
```

SwapValue is a transactional method. Can deadlock in trace:

thread1

```
enter cell1.swapValue
  t = getValue()
```

thread2

```
enter cell2.swapValue
  t = getValue()
```

```
v = other.getValue()
```

```
v = other.getValue()
```
Lock Precedence

Can prevent deadlock in transactional methods via resource-ordering based on Java hash codes (among other solutions)

```java
class Cell {
    long value;

    void swapValue(Cell other) {
        if (other == this) return; // alias check

        Cell fst = this; // order via hash codes
        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

class Server {
    double state;
    Helper helper;
    public synchronized void svc() {
        state = illegalValue;
        helper.operation();
        state = legalValue;
    }
}

Potential problems with holding locks during downstream calls

Safety: What if helper.operation throws exceptions?

Liveness: What if helper.operation causes deadlock?

Availability: Cannot accept new svc requests during helper op

Rule of Thumb (with many variants and exceptions):

Always lock when updating state

Never lock when sending message

Redesign methods to avoid holding locks during downstream calls, while still preserving safety and consistency
Synchronization of Accessor Methods

class Queue {
    private int sz_ = 0; // number of elements

    public synchronized void put(Object x) {
        // ... increment sz_ ...
    }
    public synchronized Object take() {
        // ... decrement sz_ ...
    }
    public int size() { return sz_; } // synch?
}

Should size() method be synchronized?

Pro:

• Prevents clients from obtaining stale cached values

• Ensures that transient values are never returned
  — For example, if put temporarily set sz_ = -1 as flag

Con:

• What could a client ever do with this value anyway?

Sync always needed for accessors of mutable reference variables
Locking and Singletons

Every Java class object has a lock. Both static and instance methods of Singleton classes should use it.

```java
public class Singleton { // lazy initialization
    private int a;
    private Singleton() { a = 1; }

    private static Class lock = Singleton.class;
    private static Singleton ref = null;

    public static Singleton instance() {
        synchronized(lock) {
            if (ref == null) ref = new Singleton();
            return ref;
        }
    }

    public int getA() {
        synchronized(lock) { return a; }
    }

    public void setA(int v) {
        synchronized(lock) { a = v; }
    }
}
```
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 particular logical state
- Design interfaces and protocols to reflect policy
- Ensure objects able to assess state to implement policy

There is not a separate accept mechanism in Java. So must implement policies in action methods themselves.
Examples of State-Dependent Actions

Operations on collections, streams, databases

- Remove an element from an empty queue

Operations on objects maintaining constrained values

- Withdraw money from an empty bank account

Operations requiring resources

- Print a file

Operations requiring particular message orderings

- Read an unopened file

Operations on external controllers

- Shift to reverse gear in a moving car
## Policies for State Dependent Actions

Some policy choices for dealing with pre- and post- conditions

<table>
<thead>
<tr>
<th>Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blind action</td>
<td>Proceed anyway; no guarantee of outcome</td>
</tr>
<tr>
<td>Inaction</td>
<td>Ignore request if not in right state</td>
</tr>
<tr>
<td>Balking</td>
<td>Fail (throw exception) if not in right state</td>
</tr>
<tr>
<td>Guarding</td>
<td>Suspend until in right state</td>
</tr>
<tr>
<td>Trying</td>
<td>Proceed, check if succeeded; if not, roll back</td>
</tr>
<tr>
<td>Retrying</td>
<td>Keep trying until success</td>
</tr>
<tr>
<td>Timing out</td>
<td>Wait or retry for a while; then fail</td>
</tr>
<tr>
<td>Planning</td>
<td>First initiate activity that will achieve right state</td>
</tr>
</tbody>
</table>
# Interfaces and Policies

Boring running example

```java
interface BoundedCounter {
    static final long MIN = 0;
    static final long MAX = 10;

    long value(); // INV: MIN <= value() <= MAX
                   // INIT: value() == MIN

    void inc();   // PRE: value() < MAX
    void dec();   // PRE: value() > MIN
}
```

Interfaces alone cannot convey policy

- But can suggest policy
  - For example, should `inc` throw exception? What kind?
  - Different methods can support different policies

- But can use manual annotations
  - Declarative constraints form basis for implementation
Balking

Check state upon method entry

- Must not change state in course of checking it
- Relevant state must be explicitly represented, so can be checked upon entry

Exit immediately if not in right state

- Throw exception or return special error value
- Client is responsible for handling failure

The simplest policy for fully synchronized objects

- Usable in both sequential and concurrent contexts
  — Often used in Collection classes (Vector, etc)
- In concurrent contexts, the host must always take responsibility for entire check-act/check-fail sequence
  — Clients cannot preclude state changes between check and act, so host must control
Balking Counter Example

class Failure extends Exception {
}

class BalkingCounter {
    protected long count_ = MIN;
    synchronized long value() { return count_;}

    synchronized void inc() throws Failure {
        if (count_ >= MAX) throw new Failure();
        ++count_
    }

    synchronized void dec() throws Failure {
        if (count_ <= MIN) throw new Failure();
        --count_
    }
}

// ...
void suspiciousUsage(BalkingCounter c) {
    if (c.value() > BalkingCounter.MIN)
        try { c.dec(); } catch (Failure ignore) {} 
}
void betterUsage(BalkingCounter c) {
    try { c.dec(); } catch (Failure ex) { cope();}
}
Collection Class Example

class Vec { // scaled down version of Vector
    protected Object[] data_; protected int size_=0;

    public Vec(int cap) { data_=new Object[cap]; }

    public int size() { return size_; }

    public synchronized Object at(int i)
        throws NoSuchElementException {
        if (i < 0 || i >= size_ )
            throw new NoSuchElementException();
        return data_[i];
    }

    public synchronized void append(Object x) {
        if (size_ >= data_.length) resize();
        data_[size_++] = x;
    }

    public synchronized void removeLast()
        throws NoSuchElementException {
        if (size_ == 0)
            throw new NoSuchElementException();
        data_[--size_] = null;
    }
}
Policies for Collection Traversal

How to apply operation to collection elements without interference

Balking iterators

- Throw exception on access if collection was changed. Implement via version numbers updated on each change
  — Used in JDK1.2 collections
- But can be hard to recover from exceptions

Snapshot iterators

- Make immutable copy of base collection elements. Or conversely, copy-on-write during each update.
  - But can be expensive

Indexed traversal

- Clients externally synchronize when necessary
  - But coupled to particular locking policies

Synchronized aggregate methods

- Support apply-to-all methods in collection class
  - But deadlock-prone
Synchronized Traversal Examples

interface Procedure { void apply(Object obj); }

class XVec extends Vec {
    synchronized void applyToAll(Procedure p) {
        for (int i = 0; i < size_; ++i) p.apply(data_[i]);
    }
}

class App {
    void printAllV1(XVec v) { // aggregate synch
        v.applyToAll(new Procedure() {
            public void apply(Object x) {
                System.out.println(x);
            }
        });
    }

    void printAllV2(XVec v) { // client-side synch
        synchronized (v) {
            for (int i = 0; i < v.size(); ++i)
                System.out.println(v.at(i));
        }
    }
}
Guarding

Generalization of locking for state-dependent actions

- **Locked:** Wait until *ready* (not engaged in other methods)
- **Guarded:** Wait until *an arbitrary state predicate* holds

Check state upon entry

- If not in right state, wait

  - Some other action in some other thread may eventually cause a state change that enables resumption

Introduces liveness concerns

- Relies on actions of other threads to make progress
- Useless in sequential programs
Guarding via Busy Waits

class UnsafeSpinningBoundedCounter { // don’t use
   protected volatile long count_ = MIN;
   long value() { return count_; }

   void inc() {
      while (count_ >= MAX); // spin
      ++count_;
   }
   void dec() {
      while (count_ <= MIN); // spin
      --count_;
   }
}

Unsafe — no protection from read/write conflicts
Wasteful — consumes CPU time

But busy waiting can sometimes be useful; generally when

- The conditions latch
  — once set true, they never become false

- You are sure that threads are running on multiple CPUs
  — Java doesn’t provide a way to determine or control this
Guarding via Suspension

class GuardedBoundedCounter {
    protected long count_ = MIN;

    synchronized long value() { return count_; }

    synchronized void inc() throws InterruptedException {
        while (count_ >= MAX) wait();
        ++count_
        notifyAll();
    }

    synchronized void dec() throws InterruptedException {
        while (count_ <= MIN) wait();
        --count_
        notifyAll();
    }
}

Each wait relies on a balancing notification

- Generates programmer obligations

Must recheck condition upon resumption
Java Monitor Methods

Every Java object has a wait set

- Accessed only via monitor methods, that can only be invoked under synchronization of target

wait()

- Suspends thread
- Thread is placed in wait set for target object
- Synch lock for target is released

notify()

- If one exists, any thread T is chosen from target’s wait set
- T must re-acquire synch lock for target
- T resumes at wait point

notifyAll() is same as notify() except all threads chosen

wait(ms) is same as wait() except thread is automatically notified after ms milliseconds if not already notified

Thread.interrupt causes a wait (also sleep, join) to abort.
- Same as notify except thread resumed at the associated catch
Monitors and Wait Sets

class X {
    synchronized void w() {
        before(); wait(); after();
    }
    synchronized void n() { notifyAll(); }
}

One possible trace for three threads accessing instance x:

T1       T2       T3

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

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

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

acquire lock

after();

acquire lock

after();

notifyAll();
Interactions with Interruption

Effect of `Thread.interrupt()`:

- If thread not waiting, set the `isInterrupted()` bit

- If thread is waiting, force to exit `wait` and throw `InterruptedException` upon resumption

```
<table>
<thead>
<tr>
<th>Acquiring Lock</th>
<th>Running</th>
<th>Waiting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquiring Lock + Interrupted</td>
<td>Running + interrupted</td>
<td></td>
</tr>
</tbody>
</table>
```

Diagram:
- `interrupt` from `Running` to `Running + interrupted`
- `enterAcquire` from `Acquiring Lock` to `Running`
- `exitAcquire` from `Running` to `Acquiring Lock`
- `wait` from `Running` to `Waiting`
- `Thread.interrupted, wait` from `Waiting` to `Running`
- `notify, notifyAll, timeout, interrupt` from `Waiting` to `Running`
Fairness in Java

Fairness is a system-wide progress property:

Each blocked activity will eventually continue when its enabling condition holds. *(Many variants of definition)*

- Threads waiting for lock eventually enter when lock free
- Guarded wait loops eventually unblock when condition true

Usually implemented via First-in-First-Out scheduling policies

- FIFO lock and wait queues
- Sometimes, along with preemptive time-slicing

Java does not guarantee fairness

- Potential starvation
  - A thread never gets a chance to continue because other threads are continually placed before it in queue
- FIFO usually not strictly implementable on SMPs
- But JVM implementations usually approximate fairness
- Manual techniques available to improve fairness properties
Timeouts

Intermediate points between balking and guarding

- Can vary timeout parameter from zero to infinity

Useful for heuristic detection of failures

- Deadlocks, crashes, I/O problems, network disconnects

But cannot be used for high-precision timing or deadlines

- Time can elapse between wait and thread resumption

Java implementation constraints

- \texttt{wait(ms)} does not automatically tell you if it returns because of notification vs timeout

- Must check for both. Order and style of checking can matter, depending on
  
  - If always OK to proceed when condition holds
  
  - If timeouts signify errors
Timeout Example

class TimeOutBoundedCounter {
  protected long TIMEOUT = 5000;
  // ...
  synchronized void inc() throws Failure {

    if (count_ >= MAX) {
      long start = System.currentTimeMillis();
      long waitTime = TIMEOUT;
      for (;;) {
        if (waitTime <= 0) throw new Failure();
        try { wait(waitTime); }
        catch (InterruptedException e) {
          throw new Failure();
        }
        if (count_ < MAX) break;
        long now = System.currentTimeMillis();
        waitTime = TIMEOUT - (now - start);
      }
    }
    ++count_;
    notifyAll();
  }

  synchronized void dec() throws Failure; // similar
}
Buffer Supporting Multiple Policies

```java
class BoundedBuffer {
    Object[] data_;  
    int putPtr_ = 0, takePtr_ = 0, size_ = 0;
    BoundedBuffer(int capacity) {
        data_ = new Object[capacity];
    }

    protected void doPut(Object x) { // mechanics
        data_[putPtr_] = x;
        putPtr_ = (putPtr_ + 1) % data_.length;
        ++size_;  
        notifyAll();
    }

    protected Object doTake() { // mechanics
        Object x = data_[takePtr_];
        data_[takePtr_] = null;
        takePtr_ = (takePtr_ + 1) % data_.length;
        --size_;  
        notifyAll();
        return x;
    }

    boolean isFull(){ return size_ == data_.length; }
    boolean isEmpty(){ return size_ == 0; }
}
```
Buffer (continued)

synchronized void put(Object x)
    throws InterruptedException {
        while (isFull()) wait();
        doPut(x);
    }

synchronized Object take() {
    throws InterruptedException {
        while (isEmpty()) wait();
        return doTake();
    }

synchronized boolean offer(Object x) {
    if (isFull()) return false;
    doPut(x);
    return true;
}

synchronized Object poll() {
    if (isEmpty()) return null;
    return doTake();
}
Buffer (continued)

```java
synchronized boolean offer(Object x, long ms) {
    if (isFull()) {
        if (ms <= 0) return false;
        long start = System.currentTimeMillis();
        long waitTime = ms;
        for (;;) {
            try { wait(waitTime); } catch (InterruptedException e) {
                return false;
            }
            if (!isFull()) break;
            long now = System.currentTimeMillis();
            waitTime = ms - (now - start);
            if (waitTime <= 0) return false;
        }
    }
    return doTake();
}

synchronized Object poll(long ms); // similar
```
Containment

Structurally guarantee exclusive access to internal objects

- Control their visibility
- Provide concurrency control for their methods

Applications

- Wrapping unsafe sequential code
- Eliminating need for locking ground objects and variables
- Applying special synchronization policies
- Applying different policies to the same mechanisms
class Pixel {
    private final java.awt.Point pt_;

    Pixel(int x, int y) { pt_ = new Point(x, y); }

    synchronized Point location() {
        return new Point(pt_.x, pt_.y);
    }

    synchronized void moveBy(int dx, int dy) {
        pt_.x += dx; pt_.y += dy;
    }
}

Pixel provides synchronized access to Point methods

- The reference to Point object is immutable, but its fields are in turn mutable (and public!) so is unsafe without protection

Must make copies of inner objects when revealing state

- This is the most common way to use java.awt.Point, java.awt.Rectangle, etc
Implementing Containment

Strict containment creates islands of isolated objects

- Applies recursively
- Allows inner code to run faster

Inner code must be communication-closed

- No unprotected calls in to or out from island

Outer objects must never leak identities of inner objects

- Can be difficult to enforce and check

Outermost objects must synchronize access

- Otherwise, possible thread-caching problems

Seen in concurrent versions of many delegation-based patterns

- Adapters, decorators, proxies
Hierarchical Locking

Applies when logically contained parts are not hidden from clients

Avoids deadlocks that could occur if parts fully synchronized

Can eliminate this potential deadlock if all locking in all methods in all parts relies on the common owner’s lock.

Extreme case: one Giant Lock for entire subsystem

Can use either internal or external conventions
Internal Hierarchical Locking

Visible components protect themselves using their owners’ locks:

class Part {
    protected Container owner_; // never null
    public Container owner() { return owner_; }
    void bareAction() { /* ... unsafe ... */ }
    public void m() {
        synchronized(owner()) { bareAction(); }
    }
}

Or implement using inner classes — Owner is outer class:

class Container {
    class Part {
        public void m() {
            synchronized(Container.this) {
                bareAction();
            }
        }
    }
}

Can extend to frameworks based on shared Lock objects, transaction locks, etc rather than synchronized blocks
External Hierarchical Locking

Rely on callers to provide the locking

class Client {
    void f(Part p) {
        synchronized (p.owner()) { p.bareAction(); }
    }
}

Used in AWT

- java.awt.Component.getTreeLock()

Can sometimes avoid more locking overhead, at price of fragility

- Can manually minimize use of synchronized
- Requires that all callers obey conventions
- Effectiveness is context dependent
  - Breaks encapsulation
  - Doesn’t work with fancier schemes that do not directly rely on synchronized blocks or methods for locking
class Part {
    protected boolean cond_ = false;

    synchronized void await() {
        while (!cond_)
            try { wait(); }
            catch(InterruptedException ex) {}  
    }

    synchronized void signal(boolean c) {
        cond_ = c; notifyAll();
    }
}

class Whole {
    final Part part_ = new Part();

    synchronized void rely() { part_.await(); }

    synchronized void set(boolean c){
        part_.signal(c);
    }
}

What happens when Whole.rely() called?
Nested Monitors

If thread $T$ calls `whole.rely`

- It waits within `part`
- The lock to `whole` is retained while $T$ is suspended
- No other thread will ever unblock it via `whole.set`

→ **Nested Monitor Lockout**

Policy clash between guarding by `Part` and containment by `Whole`

Never wait on a hidden contained object in Java while holding lock
Avoiding Nested Monitors

Adapt internal hierarchical locking pattern

Can use inner classes, where Part waits in Whole’s monitor

class Whole { // ...
    class Part { // ...    
        public void await() {    
            synchronized (Whole.this) {
                while (...) Whole.this.wait() // ...
            } } } }

Create special Condition objects

• Condition methods are never invoked while holding locks

• Some concurrent languages build in special support for Condition objects

    – But generally only deal with one-level nesting

• Can build Condition class library in Java
Splitting Objects and Locks

Synopsis

- Isolate independent aspects of state and/or behavior of a host object into helper objects
- The host object delegates to helpers
- The host may change which helpers it uses dynamically

Applications

- Atomic state updates
  - Conservative and optimistic techniques
- Avoiding deadlocks
  - Offloading locks used for status indicators, etc
- Improving concurrency
  - Reducing lock contention for host object
- Reducing granularity
  - Enabling fine-grained concurrency control
Isolating Dependent Representations

Does `Location` provide strong enough semantic guarantees?

```java
class Location { // repeated
    private double x_, y_;  
synchronized double x() { return x_; }  
synchronized double y() { return y_; }
    synchronized void moveBy(double dx, double dy) {
        x_ += dx; y_ += dy;
    }
}
```

No protection from interleaving problems such as:

Thread 1: `x=loc.x();` ...............; `y=loc.y();`

Thread 2: .............; `loc.moveBy(1,6);` .............;

Thread 1 can have incorrect view (old `x`, new `y`)

Avoid by splitting out dependent representations in separate class
Conservative Representation Updates

class XY { // immutable
    private final double x_, y_;  
    XY(double x, double y) { x_ = x; y_ = y; }  
    double x() { return x_; }  
    double y() { return y_; }  
}

class LocationV2 {
    private XY xy_;  
    LocationV2(double x, double y) {  
        xy_ = new XY(x, y);  
    }
    synchronized XY xy() { return xy_; }  
    synchronized void moveBy(double dx, double dy) {  
        xy_ = new XY(xy_.x() + dx, xy_.y() + dy);  
    }
}

Locking moveBy() ensures that the two accesses of xy_ do not get different points

Locking xy() avoids thread-cache problems by clients
Optimistic Representation Updates

class LocationV3 {
    private XY xy_;

    private synchronized boolean commit(XY oldp, XY newp) {
        boolean success = (xy_ == oldp);
        if (success) xy_ = newp;
        return success;
    }

    LocationV3(double x, double y) { xy_ = new XY(x, y); }

    synchronized XY xy() { return xy_; }

    void moveBy(double dx, double dy) {
        while (!Thread.interrupted()) {
            XY oldp = xy();
            XY newp = new XY(oldp.x() + dx, oldp.y() + dy);
            if (commit(oldp, newp)) break;
            Thread.yield();
        }
    }
}
Optimistic Update Techniques

Every public state update method has four parts:

→ **Record current version**
  
  Easiest to use reference to immutable representation
  
  — Or can assign version numbers, transaction IDs, or time stamps to mutable representations

→ **Build new version, without any irreversible side effects**
  
  All actions before `commit` must be reversible
  
  — Ensures that failures are clean (no side effects)
  
  — No I/O or thread construction unless safely cancellable
  
  — All internally called methods must also be reversible

→ **Commit to new version if no other thread changed version**
  
  Isolation of state updates to single atomic `commit` method can avoid potential deadlocks

→ **Otherwise fail or retry**
  
  Retries can **livelock** unless proven **wait-free** in given context
Optimistic State-Dependent Policies

As with optimistic updates, isolate state into versions, and isolate state changes to `commit` method.

In each method:

- Record current version
- Build new version
- Commit to version if success and no one changed version
- Otherwise fail or retry

Retry policy is a tamed busy wait. Can be more efficient than guarded waits if

- Conflicts are rare
- Guard conditions usually hold
- Running on multiple CPUs
class OptimisticBoundedCounter {
    private Long count_ = new Long(MIN);
    long value() { return count().longValue(); }
    synchronized Long count() { return count_; }
    private synchronized boolean commit(Long oldc, Long newc) {
        boolean success = (count_ == oldc);
        if (success) count_ = newc;
        return success;
    }

    public void inc() throws InterruptedException {
        for (;;) { // retry-based
            if (Thread.interrupted())
                throw new InterruptedException();
            Long c = count();
            long v = c.longValue();
            if (v < MAX && commit(c, new Long(v+1)))
                break;
            Thread.yield();
        }
    }
    public void dec() // symmetrical
}
Splitting Locks and Behavior

Associate a helper object with an independent subset of state and functionality.

Delegate actions to helper via pass-through method

```java
class Shape {
  // Assumes size & dimension are independent

  int height_ = 0;
  int width_ = 0;

  synchronized void grow() { ++height_; ++width_; }

  Location l = new Location(0,0); // fully synched

  void shift() { l.moveBy(1, 1); } // Use l’s synch
}
```

grow and shift can execute simultaneously

When there is no existing object to delegate independent actions:

- Use an arbitrary object as a lock, and protect associated methods using synchronized block on that lock
  - Useful for concurrent data structures
Concurrent Queue

class TwoLockQueue {
    final static class Node {
        Object value; Node next = null;
        Node(Object x) { value = x; }
    }
    private Node head_ = new Node(null); // dummy hdr
    private Node last_ = head_;
    private Object lastLock_ = new Object();

    void put(Object x) {
        synchronized (lastLock_) {
            last_ = last_.next = new Node(x);
        }
    }

    synchronized Object poll() { // null if empty
        Object x = null;
        Node first = head_.next; // only contention pt
        if (first != null) {
            x = first.value; first.value = null;
            head_ = first; // old first becomes header
        }
        return x;
    }
}

88
puts and polls can run concurrently

- The data structure is crafted to avoid contending access
  - Rely on Java atomicity guarantees at only potential contention point
- But multiple puts and multiple polls disallowed

Weakens semantics

- poll may return null if another thread is in midst of put
- Balking policy for poll is nearly forced here
  - But can layer on blocking version
Introducing Concurrency into Applications

Three sets of patterns

Each associated with a reason to introduce concurrency

Autonomous Loops

Establishing independent cyclic behavior

Oneway messages

Sending messages without waiting for reply or termination
  • Improves availability of sender object

Interactive messages

Requests that later result in reply or callback messages
  • Allows client to proceed concurrently for a while

Most design ideas and semantics stem from active object models.
Autonomous Loops

Simple non-reactive active objects contain a `run` loop of form:

```java
public void run() {
    while (!Thread.interrupted())
        doSomething();
}
```

Normally established with a constructor containing:

```java
new Thread(this).start();
```

Perhaps also setting priority and daemon status

Normally also support other methods called from other threads

Requires standard safety measures

Common Applications

- Animations
- Simulations
- Message buffer Consumers
- Polling daemons that periodically sense state of world
public class Particle implements Runnable {
    private int x = 0, y = 0;
    private Canvas canvas;
    public Particle(Canvas host) { canvas = host; }

    synchronized void moveRandomly() {
        x += (int) (((Math.random() - 0.5) * 5));
        y += (int) (((Math.random() - 0.5) * 5));
    }

    public void draw(Graphics g) {
        int lx, ly;
        synchronized (this) { lx = x; ly = y; }
        g.drawRect(lx, ly, 10, 10);
    }

    public void run() {
        for(;;) {
            moveRandomly();
            canvas.repaint();
            try { Thread.sleep((int)(Math.random() * 10)); }
            catch (InterruptedException e) { return; }
        }
    }
}
import java.awt.*;
import java.applet.*;
public class ParticleApplet extends Applet {
    public void init() {
        add(new ParticleCanvas(10));
    }
}

class ParticleCanvas extends Canvas {
    Particle[] particles;
    ParticleCanvas(int nparticles) {
        setSize(new Dimension(100, 100));
        particles = new Particle[nparticles];
        for (int i = 0; i < particles.length; ++i) {
            particles[i] = new Particle(this);
            new Thread(particles[i]).start();
        }
    }

    public void paint(Graphics g) {
        for (int i = 0; i < particles.length; ++i)
            particles[i].draw(g);
    }
} // (needs lots of embellishing to look nice)
Oneway Messages

Conceptually oneway messages are sent with

- No need for replies
- No concern about failure (exceptions)
- No dependence on termination of called method
- No dependence on order that messages are received

But may sometimes want to cancel messages or resulting activities
## Oneway Message Styles

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events</td>
<td>Mouse clicks, etc</td>
</tr>
<tr>
<td>Notifications</td>
<td>Status change alerts, etc</td>
</tr>
<tr>
<td>Postings</td>
<td>Mail messages, stock quotes, etc</td>
</tr>
<tr>
<td>Activations</td>
<td>Applet creation, etc</td>
</tr>
<tr>
<td>Commands</td>
<td>Print requests, repaint requests, etc</td>
</tr>
<tr>
<td>Relays</td>
<td>Chain of responsibility designs, etc</td>
</tr>
</tbody>
</table>

### Some semantics choices

- **Asynchronous**: Entire message send is independent
  - By far, most common style in reactive applications
- **Synchronous**: Caller must wait until message is *accepted*
  - Basis for *rendezvous* protocols
- **Multicast**: Message is sent to *group* of recipients
  - The group might not even have any members
Messages in Java

Direct method invocations

- Rely on standard call/return mechanics

Command strings

- Recipient parses then dispatches to underlying method
- Widely used in client/server systems including HTTP

EventObjects and service codes

- Recipient dispatches
- Widely used in GUIs, including AWT

Request objects, asking to perform encoded operation

- Used in distributed object systems — RMI and CORBA

Class objects (normally via .class files)

- Recipient creates instance of class
- Used in Java Applet framework

Runnable commands

- Basis for thread instantiation, mobile code systems
Design Goals for Oneway Messages

Object-based forces

Safety
- Local state changes should be atomic (normally, locked)
  - Typical need for locking leads to main differences vs single-threaded Event systems
- Safe guarding and failure policies, when applicable

Availability
- Minimize delay until host can accept another message

Activity-based forces

Flow
- The activity should progress with minimal contention

Performance
- Minimize overhead and resource usage
Design Patterns for Oneway Messages

Thread-per-Message

- `client` → `host` → `handler`
  - Start new thread

Thread-per-Activity via Pass-throughs

- `client` → `host` → `handler`
  - Same thread

Thread-per-Object via Worker Threads (variants: Pools, Listeners)

- `client` → `host` → `channel` → `handler`
  - Worker thread
  - `put` and `take` operations
Reactive Methods

Code scaffolding for illustrating patterns:

```java
class Host {
    // ...
    private long localState_; // Or any state vars
    private Handler handler_; // Message target

    public void react(...) {
        updateState(...);
        sendMessage(...);
    }

    private synchronized void updateState(...) {
        // Assign to localState_
    }

    private void sendMessage(...) {
        // Issue handler.process(...)
    }
}
```

`react()` may be called directly from client, or indirectly after decoding command, event, etc
Thread-per-Message

class Host { //...
    public void react(...) {
        updateState(...);
        sendMessage(...);
    }

    synchronized void sendMessage(...) {

        Runnable command = new Runnable() { // wrap
            final Handler dest = handler_;  
            public void run() {
                dest.process(...);
            }
        };

        new Thread(command).start();  // run
    }
}

Runnable is the standard Java interface describing argumentless, resultless command methods (aka closures, thunks)

Synchronization of sendMessage desirable if handler_ or process() arguments not fixed/final

Variants: Thread-per-connection (sockets)
Thread-per-Message Protocol

client ➔ host

... updateState...

handler ➔ command

start/run

process
Multicasts can either

- Generate one thread per message, or
- Use a single thread for all messages

Depends on whether OK to wait each one out before sending next one
TPM Socket-based Server

class Server implements Runnable {
    public void run() {
        try {
            ServerSocket socket = new ServerSocket(PORT);
            for (;;) {
                final Socket connection = socket.accept();
                new Thread(new Runnable() {
                    public void run() {
                        new Handler().process(connection);
                    }
                }).start();
            }
        } catch (Exception e) { /* cleanup; exit */ }
    }
}

class Handler {
    void process(Socket s) {
        InputStream i = s.getInputStream();
        OutputStream o = s.getOutputStream();
        // decode and service request, handle errors
        s.close();
    }
}
Thread Attributes and Scheduling

Each Thread has an integer priority

- From `Thread.MIN_PRIORITY` to `Thread.MAX_PRIORITY` (currently 1 to 10)
- Initial priority is same as that of the creating thread
- Can be changed at any time via `setPriority`
- `ThreadGroup.setMaxPriority` establishes a ceiling for all threads in the group

JVM schedulers give preference to threads with higher priority

- But preference is left vague, implementation-dependent
- No guarantees about fairness for equal-priority threads
  — Time-slicing is permitted but not required
- No guarantees whether highest-priority or longest-waiting threads acquire locks or receive notifications before others

Priorities can only be used heuristically

- Build custom Queues to control order of sequential tasks
- Build custom Conditions to control locking and notification
Adding Thread Attributes

Thread objects can hold non-public Thread-Specific contextual attributes for all methods/objects running in that thread

- Normally preferable to static variables

Useful for variables that apply per-activity, not per-object

- Timeout values, transaction IDs, Principals, current directories, default parameters

Useful as tool to eliminate need for locking

- Used internally in JVMs to optimize memory allocation, locks, etc via per-thread caches
Implementing Thread-Specific Storage

class GameThread extends Thread { // ...
    private long movementDelay_ = 3;
    static GameThread currentGameThread() {
        return (GameThread)(Thread.currentThread());
    }
    static long getDelay() {
        return currentGameThread().movementDelay_;
    }
    static long setDelay(long t) {
        currentGameThread().movementDelay_ = t;
    }
}
class Ball { // ...
    void move() { // ...
        Thread.sleep(GameThread.getDelay());
    }
}
class Main { ... new GameThread(new Game()) ... } 

Define contextual attributes in special Thread subclasses

- Can be accessed without locking if all accesses are always via Thread.currentThread()
- Enforce via static methods in Thread subclass
Using ThreadLocal

java.lang.ThreadLocal available in JDK1.2

- An alternative to defining special Thread subclasses

Uses internal hash table to associate data with threads

- Avoids need to make special Thread subclasses when adding per-thread data

  — Trade off flexibility vs strong typing and performance

```java
class Ball {
    static ThreadLocal delay = new ThreadLocal();
    void move() { // ...
        if (delay.get() == null) delay.set(new Long(3));
        long d = ((Long)(delay.get())).longValue();
        Thread.sleep(d);
    }
}
```

Can extend to implement inherited Thread contexts

Where new threads by default use attributes of the parent thread that constructed them
Other Scoping Options

Choices for maintaining context information

per Application
per Principal
per Session
per Thread
per Method
per Block
per Class
per Object
per Role
per Group
per Site
per Domain
per Aggregate
per System
per Version
Choosing among Scoping Options

Reusability heuristics

- Responsibility-driven design
- Factor commonalities, isolate variation
- Simplify Programmability
  - Avoid long parameter lists
  - Avoid awkward programming constructions
  - Avoid opportunities for errors due to policy conflicts
  - Automate propagation of bindings

Conflict analysis

Example: Changing per-object bindings via tuning interfaces can lead to conflicts when objects support multiple roles

- Settings made by one client impact others
- Common error with Proxy objects
- Replace with per-method, per-role, per-thread
Thread-per-Activity via Pass-Throughs

class Host { //...

    void reactV1(...) { // no synch
        updateState();    // isolate in synched method
        sendMessage(...);
    }
    void sendMessage(...) { // no synch
        handler_.process(...);    // direct call
    }
}

A kind of forwarding — conceptually removing host from call chain

Callers of react must wait for handler.process to terminate, or generate their own threads

Host can respond to another react call from another thread immediately after updating state
Using Pass-Throughs

Common approach to writing AWT Event handlers, JavaBeans methods, and other event-based components.

But somewhat fragile:

- There is no “opposite” to synchronized
  - Avoid self calls to react from synchronized methods
- Need care in accessing representations at call-point
  - If handler_variable or process arguments not fixed, copy values to locals while under synchronization
- Callers must be sure to create thread around call if they cannot afford to wait or would lock up

Variants

Bounded Thread-per-Message

- Keep track of how many threads have been created. If too many, fall back to pass-through.

Mediated

- Register handlers in a common mediator structured as a pass-through.
Multicast Pass-Throughs

class Host { //...
    CopyOnWriteSet handlers_;

    synchronized void addHandler(Handler h) {
        handlers_.add(h); // copy
    }

    void sendMessage(...) {
        Iterator e = handlers_.iterator();
        while (e.hasNext())
            ((Handler)(e.next())).process(...);
    }
}

Normally use **copy-on-write** to implement target collections

- Additions are much less common than traversals

AWT uses java.awt.AWTEventMulticaster class

- Employs variant of **FixedList** class design

- But coupled to AWT Listener framework, so cannot be used in other contexts
Thread-Per-Object via Worker Threads

Establish a producer-consumer chain

Producer

Reactive method just places message in a channel

Channel might be a buffer, queue, stream, etc

Message might be a Runnable command, event, etc

Consumer

Host contains an autonomous loop thread of form:

```java
while (!Thread.interrupted()) {
    m = channel.take();
    process(m);
}
```

Common variants

Pools

Use more than one worker thread

Listeners

Separate producer and consumer in different objects
Worker Thread Example

interface Channel { // buffer, queue, stream, etc
    void put(Object x);
    Object take();
}

class Host { //...
    Channel channel_ = ...;
    void sendMessage(...) {
        channel_.put(new Runnable() { // enqueue
            public void run(){
                handler_.process(...);
            }
        });
    }
}

Host() { // Set up worker thread in constructor
    // ...
    new Thread(new Runnable() {
        public void run() {
            while (!Thread.interrupted())
                ((Runnable)(channel_.take())).run();
        }
    }).start();
}
Worker Thread Protocol

- Client
- Host
- Handler
- Command
- Channel

- react
- put
- take
- !empty
- run
- process
Channel Options

Unbounded queues
- Can exhaust resources if clients faster than handlers

Bounded buffers
- Can cause clients to block when full

Synchronous channels
- Force client to wait for handler to complete previous task

Leaky bounded buffers
- For example, drop oldest if full

Priority queues
- Run more important tasks first

Streams or sockets
- Enable persistence, remote execution

Non-blocking channels
- Must take evasive action if put or take fail or time out
Example: The AWT Event Queue Thread

AWT uses one thread and a single java.awt.EventQueue

- Single thread makes visual updates appear more coherent
- Browsers may add per-Applet threads and queues

Events implement java.util.EventObject

- Include both “Low-level” and “Semantic” events

Event dequeuing performed by AWT thread

repaint() places drawing request event in queue.

- The request may be optimized away if one already there
- update/paint is called when request dequeued
  - Drawing is done by AWT thread, not your threads
class MyApplet extends Applet
    implements ActionListener {

    Button button = new Button("Push me");
    boolean onOff = false;

    public void init() {
        button.addActionListener(this); // attach
        add(button);                // add to layout
    }

    public void ActionPerformed(ActionEvent evt) {
        if (evt.getSource() == button) // dispatch
            toggle(); // update state
        repaint(); // issue event(not necessary here)
    }

    synchronized void toggle() {
        onOff = !onOff;
    }
}
Using AWT in Concurrent Programs

Most conservative policy is to perform all GUI-related state updates in event handling methods

- Define and generate new `EventObjects` if necessary
- Consider splitting GUI-related state into separate classes
- Do not rely on thread-safety of GUI components

Define drawing and event handling methods in reactive form

- Do not hold locks when sending messages
- Do not block or delay caller thread (the AWT thread)
- Generate threads to arrange GUI-unrelated processing
  - Explicitly set their `ThreadGroups`
- Generate events to arrange GUI-related async processing
  - `Swing` includes some utility classes to make this easier
Thread Pools

Use a collection of worker threads, not just one

- Can limit maximum number and priorities of threads

Often faster than thread-per-message

- But slower than single thread working off a multislot buffer unless handler actions permit parallelism
- Often works well for I/O-bound actions
Listeners

House worker thread in a different object

- Even in a different process, connected via socket

But full support for **remote** listeners requires frameworks for

- Naming remote acquaintances (via registries, jndi etc)
- Failure, reliability, fault tolerance
- Security, protocol conformance, ...

Can make more transparent via **Proxies**

- Channels/Listeners that duplicate interface of Handler, but wrap each message as queued command for later execution
Remote Worker Threads

class Host { // ...
    ObjectOutputStream c; // connected to a Socket
    void sendMessage(...) {
        c.writeObject(new SerializableRunnable() {
            public void run(){
                new Handler().process(...);
            }
        });
    }
}

class Listener { // instantiate on remote machine
    ObjectInputStream c; // connected to a Socket
    Listener() {
        c = new ...
        Thread me = new Thread(new Runnable() {
            public void run() {
                for (; ;) {
                    ((Runnable)(c.readObject())).run();
                }
            }
        });
        me.start();
    }
}
Synchronous Channels

Synchronous oneway messages same as asynchronous, except:

• Caller must wait at least until message is **accepted**

Simplest option is to use synchronized methods

• Caller must wait out **all** downstream processing

Increase concurrency via synchronous channel to worker thread

• Every put must wait for take
• Every take must wait for put

Basis for synchronous message passing frameworks (CSP etc)

• Enables more precise, deterministic, analyzable, but expensive flow control measures.
• Relied on in part because CSP-inspired systems did not allow dynamic construction of new threads, so required more careful management of existing ones.

Variants

• **Barrier**: Threads wait but do not exchange information
• **Rendezvous**: Bidirectional message exchange at wait
Synchronous Channel Example

class SynchronousChannel {
    Object item_ = null;
    boolean putting_ = false; //disable multiple puts

    synchronized void put(Object e) {
        if (e == null) return;
        while (putting_) try { wait(); } catch ...
        putting_ = true;
        item_ = e;
        notifyAll();
        while (item_ != null) try { wait(); } catch ...
        putting_ = false;
        notifyAll();
    }

    synchronized Object take() {
        while (item_ == null) try { wait(); } catch ...
        Object e = item_;
        item_ = null;
        notifyAll();
        return e;
    }
}

## Some Pattern Trade-Offs

<table>
<thead>
<tr>
<th></th>
<th>Thread-per-Message</th>
<th>Pass-Through</th>
<th>Worker Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread-per-Message</strong></td>
<td>+ Simple semantics: When in doubt, make a new thread</td>
<td>+ Low overhead</td>
<td>+ Tunable semantics and structure</td>
</tr>
<tr>
<td></td>
<td>- Can be hard to limit resource usage</td>
<td>- Fragile</td>
<td>+ Can bound resource usage</td>
</tr>
<tr>
<td></td>
<td>- Thread start-up overhead</td>
<td>- No within-activity concurrency</td>
<td>- Higher overhead</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Can waste threads</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- May block caller (if buffer full etc)</td>
</tr>
</tbody>
</table>
Interactive Messages

Synopsis

- Client activates Server with a oneway message

  ![Diagram showing client activating server with a oneway message](image)

- Server later invokes a callback method on client

  ![Diagram showing callback method](image)

  Callback can be either oneway or procedural

  Callback can instead be sent to a helper object of client

  Degenerate case: inform only of task completion

Applications

- Observer designs

- Completion indications from file and network I/O

- Threads performing computations that yield results
Observer Designs

The oneway calls are change notifications
The callbacks are state queries
Examples
  • Screen updates
  • Constraint frameworks
  • Publish/subscribe
  • Hand-built variants of wait and notifyAll
Notifications must use oneway design pattern
Otherwise:
  \[ thread1 \] changeNotification
  \[ val == v \]
  \[ can deadlock against: \]
  \[ thread2 \] currentValue
  \[ cache == val \]
class Subject {
    protected double val_ = 0.0; // modeled state
    public synchronized double getValue(){
        return val_;}
    protected synchronized void setValue(double d){
        val_ = d;}

    protected CopyOnWriteSet obs_ = new COWImpl();
    public void attach(Observer o) { obs_.add(o); }

    public void changeValue(double newstate) {
        setValue(newstate);
        Iterator it = obs_.iterator();
        while (it.hasNext()){
            final Observer o = (Observer)(it.next());
            new Thread(new Runnable() {
                public void run() {
                    o.changeNotification(this);        
                }
            }).start();
        }    // More common to use pass-through calls instead of threads}
Observer Example (Continued)

class Observer {
    protected double cachedState_; // last known state
    protected Subject subj_; // only one here

    Observer(Subject s) {
        subj_ = s; cachedState_ = s.getValue();
        display();
    }

    synchronized void changeNotification(Subject s) {
        if (s != subj_) return; // only one subject

        double oldState = cachedState_;
        cachedState_ = subj_.getValue(); // probe

        if (oldState != cachedState_) display();
    }

    synchronized void display() { // default version
        System.out.println(cachedState_);
    }
}
Completion Callbacks

The async messages are service activations

The callbacks are **continuation** calls that transmit results

- May contain a message ID or completion token to tell client which task has completed

Typically two kinds of callbacks

**Success** – analog of `return`

**Failure** – analog of `throw`

Client readiness to accept callbacks may be state-dependent

- For example, if client can only process callbacks in a certain order
Completion Callback Example

Callback interface

```java
interface FileReaderClient {
    void readCompleted(String filename);
    void readFailed(String filename, IOException ex);
}
```

Sample Client

```java
class FileReaderApp implements FileReaderClient {
    private byte[] data_

    void readCompleted(String filenm) {
        // ... use data ...
    }

    void readFailed(String fn, IOException e) {
        // ... deal with failure ...
    }

    void app() {
        new Thread(new FileReader("file",
                        data_, this)).start();
    }
}
```
Completion Callbacks (continued)

Sample Server

class FileReader implements Runnable {
    final String nm_;  
    final byte[] d_;  
    final FileReaderClient client_; // allow null

    public FileReader(String name, byte[] data,
                       FileReaderClient c) {
        nm_ = name; d_ = data; client_ = c;
    }

    void run() {
        try {
            // ... read...
            if (client_ != null)
                client_.readCompleted(nm_);
        }
        catch (IOException ex) {
            if (client_ != null)
                client_.readFailed(nm_, ex);
        }
    }
}
Threads and I/O

Java I/O calls generally block

- `Thread.interrupt` causes them to unblock
  - (This is broken in many Java implementations)
- Time-outs are available for some Socket operations
  - `Socket.setSoTimeOut`
- Can manually set up classes to arrange time-out interrupts for other kinds of I/O

Common variants of I/O completion callbacks

- Issue callback whenever there is enough data to process, rather than all at once
- Send a `Runnable` completion action instead of callback
- Use thread pools for either I/O or completion actions

Alternatives

- Place the I/O and the subsequent actions all in same method, run in same thread.
- Read into a buffer serviced by a worker thread
Rerouting Exceptions

Callbacks can be used instead of exceptions in any asynchronous messaging context, not just those directly constructing threads.

Variants seen in Adaptors that call methods throwing exceptions that clients do not know how to handle:

```java
interface Server { void svc() throws SException; }
interface EHandler { void handle(Exception e); }

class SvcAdapter {
    Server server = new ServerImpl();
    EHandler handler;
    void attachHandler(EHandler h) { handler = h; }
    public void svc() { // no throw clause
        try { server.svc(); }
        catch (SException e) {
            if (handler != null) handler.handle(e);
        }
    }
}
```

Pluggable Handlers can do anything that a normal catch clause can
- Including cancelling all remaining processing in any thread
- But are less structured and sometimes more error-prone
Joining Threads

Thread.join() may be used instead of callbacks when

- Server does not need to call back client with results
- But client cannot continue until service completion

Usually the easiest way to express termination dependence

- No need to define callback interface or send client ref as argument
- No need for server to explicitly notify or call client
- Internally implemented in java by
  - t.join() calls t.wait()
  - terminating threads call notifyAll()

Can use to simulate futures and deferred calls found in other concurrent OO languages

- But no syntactic support for futures
public class PictureDisplay {
    private final PictureRenderer myRenderer_;    // ...

    public void show(final byte[] rawPic) {
        class Waiter implements Runnable {
            Picture result = null;
            public void run() {
                result = myRenderer_.render(rawPic);
            }
        }
        Waiter waiter = new Waiter();
        Thread t = new Thread(waiter);
        t.start();

        displayBorders(); // do other things
        displayCaption(); // while rendering

        try { t.join(); }  
        catch(InterruptedException e) { return; }

        displayPicture(waiter.result);
    }
}
Join Protocol

... other actions...

displayPicture(result)
Futures

Encapsulate waits for results of operations performed in threads

- Futures are “data” types that wait until results ready
  - Normally requires use of interfaces for types

Clients wait only upon trying to use results

```java
interface Pic { byte[] getImage(); }
interface Renderer { Pic render(byte[] raw); }

class AsynchRenderer implements Renderer {
    static class FuturePic implements Pic {
        byte[] img_ = null;
        synchronized void setImage(byte[] img) {
            img_ = img;
            notifyAll();
        }
        public synchronized byte[] getImage() {
            while (img_ == null)
                try { wait(); } catch (InterruptedException e) { ... }
            return img_;
        }
    }
    public synchronized byte[] getImage() {
        while (img_ == null)
            try { wait(); } catch (InterruptedException e) { ... }
        return img_;
    }
} // continued
```
Futures (continued)

// class AsynchRender, continued

public Pic render(final byte[] raw) {
    final FuturePic p = new FuturePic();
    new Thread(new Runnable() {
        public void run() {
            p.setImage(doRender(raw));
        }
    }).start();
    return p;
}

private Pic doRender(byte[] r); // ...
}

class App { // sample usage
    void app(byte[] r) {
        Pic p = new AsynchRenderer().render(r);
        doSomethingElse();
        display(p.getImage()); // wait if not yet ready
    }
}

Could alternatively write join-based version.
Cancellation

Threads normally terminate after completing their run methods

May need to cancel asynchronous activities before completion

• Applet.stop() called
• User hits a CANCEL button
• Threads performing computations that are not needed
• I/O or network-driven activites that encounter failures

Options

Asynchronous cancellation: Thread.stop

Polling and exceptions: Thread.interrupt

Terminating program: System.exit

Minimizing contention: setPriority(MIN_PRIORITY)

Revoking permissions: SecurityManager methods

Unlinking resources known to cause failure exceptions
Asynchronous Cancellation

Thread.stop stops thread by throwing ThreadDeath exception

Deprecated in JDK1.2 because it can corrupt object state:

class C {
    private int v;          // invariant: v >= 0
    synchronized void f() {
        v = -1;              // temporarily set to illegal value
        compute();          // call some other method
        v = 1;              // set to legal value
    }
    synchronized void g() { // depend on invariant
        while (v != 0) { --v; something(); } }
}

What happens if stop occurs during compute()?

In principle, could catch(ThreadDeath)

• But this would only work well if done after just about every line of code in just about every Java class. Impractical.

• Most other thread systems (including POSIX) either do not support or severely restrict asynchronous cancellation
Safety can be maintained by each object checking cancellation status only when in an appropriate state to do so, relying on:

**thread.isInterrupted**
- Returns current interruption status.

*(static) Thread.interrupted*
- *Clears* status for current thread, returning previous status.

**thread.interrupt**
- Sets interrupted status, and also causes applicable methods to throw `InterruptedException`
- Threads that are blocked waiting for synchronized method or block entry are NOT awakened by `interrupt` exception

**InterruptedException**
- Thrown by `Thread.sleep`, `Thread.join`, `Object.wait` if blocked during interruption, *also clearing status*
- Blocking IO methods in the `java.io` package respond to `interrupt` by throwing `InterruptedException`
Implementing a Cancellation Policy

Best-supported policy is:

```java
Thread.isInterrupted() means cancelled
```

Any method sensing interruption should

- Assume current task is cancelled.
- Exit as quickly and cleanly as possible.
- Ensure that callers are aware of cancellation. Options:

```java
Thread.currentThread().interrupt()

throw new InterruptedException()
```

Alternatives

- Local recovery and continuation
- Centralized error recovery objects
- Always ignoring/resetting status
Detecting Cancellation

Cancellation can be checked as a precondition for any method

```java
    if (Thread.currentThread().isInterrupted())
        cancellationCode();
```

- Also in loop headers of looping methods, etc

Can be caught, thrown, or rethrown as an exception

```java
    try {
        somethingThrowingInterruptedException();
    } catch (InterruptedException ex) {
        cancellationCode();
    }
```

- Or as a subclass of a general failure exception, as in `InterruptedException` or a subclass of it,

Placement, style, and poll frequency require engineering tradeoffs

- How important is it to stop now?
- How hard is it to stop now?
- Will another object detect and deal with at a better time?
- Is it too late to stop an irreversible action?
- Does it really matter if the thread is stopped?
Responses to Cancellation

Early return

- Clean up and exit without producing or signalling errors — May require rollback or recovery
- Callers can poll status if necessary to find out why action was not carried out.
- Reset (if necessary) interruption status before return:
  `Thread.currentThread().interrupt()`

Continuation (ignoring cancellation status)

- When it is too dangerous to stop
- When partial actions cannot be backed out
- When it doesn’t matter (but consider lowering priority)

Throwing `InterruptedException`

- When callers must be alerted on method return

Throwing a general failure Exception

- When interruption is one of many reasons method can fail
Multiphase Cancellation

Foreign code running in thread might not respond to cancellation.

Dealing with this forms part of any security framework. Example:

```java
static boolean terminate(Thread t) {
    if (!t.isAlive()) return true;  // already dead
    // phase 1 -- graceful cancellation
    t.interrupt();
    try { t.join(maxWaitToDie); }  
    catch(InterruptedException e){}  // ignore
    if (!t.isAlive()) return true;  // success
    // phase 2 -- trap all security checks
    theSecurityMgr.denyAllChecksFor(t); // made-up
    try { t.join(maxWaitToDie); }  
    catch(InterruptedException ex) {}  
    if (!t.isAlive()) return true;
    // phase 3 -- minimize damage
    t.setPriority(Thread.MIN_PRIORITY);  
    // or even unsafe last-resort t.stop()
    return false;
}
```
Shutting Down Applets

Applets can create threads
— usually in `Applet.start`
and terminate them
— usually in `Applet.stop`

These threads should be cancellable
• Otherwise, it is impossible to predict lifecycle
• No guarantees about when browsers will destroy, or whether threads automatically killed when unloading

Guidelines
• Explicitly cancel threads (normally in `Applet.stop`)
• Ensure that activities check cancellation often enough
• Consider last-resort `Thread.stop` in `Applet.destroy`
Concurrent Application Architectures

Establishing application- (or subsystem-) wide Policies

- Communication directionality, synchronization
- Avoid inconsistent case-by-case decisions

Samplings from three styles

Flow systems
- Wiring together processing stages
  - Illustrated with Push Flow designs

Parallel execution
- Partitioning independent tasks
  - Illustrated with Group-based designs

Layered services
- Synchronization and control of ground objects
  - Illustrated with Before/After designs
Push Flow Systems

Systems in which (nearly) all activities are performed by objects issuing oneway messages along paths from sources to sinks

• Each message transfers information and/or objects

Examples

Control systems
Assembly systems
Workflow systems
Event processing
Chain of command
Pipeline algorithms

Requires common directionality and locality constraints

• Precludes many safety and liveness problems
• Success relies on adherence to design rules — potentially formally checkable

The simplest and sometimes best open systems protocol
Stages in Flow Systems

Every stage is a producer and/or consumer.

Stages implement common interface with method of form

\[
\text{void } \text{put}(\text{Object item})
\]

May have multiple successors

Outgoing elements may be

- multicasted or
- routed

May have multiple predecessors

Incoming elements may be

- combined or
- collected

Normally require explicit linkages
  — only one stage per connection

Each stage can define `put` using any appropriate oneway message implementation pattern — may differ across stages.
Exclusive Ownership of Resources

Elements in most flow systems act like physical resources in that

- If you have one, then you can do something (with it) that you couldn’t do otherwise.
- If you have one, then no one else has it.
- If you give it to someone else, then you no longer have it.
- If you destroy it, then no one will ever have it.

Examples

- Invoices
- Network packets
- File and socket handles
- Tokens
- Mail messages
- Money
How should stages manage resource objects?

class Stage {
    Resource res;
    void put(Resource r) { /* ... */ }
}

Both reference-passing “shared memory” and copy-based “message passing” policies can encounter problems:

- **Shared memory**
  - Synchronize access to resource

- **Message passing**
  - Deal with identity differences
Ownership Transfer

Transfer policy

At most one stage refers to any resource at any time

Require each owner to forget about each resource after revealing it to any other owner as message argument or return value

- Implement by nulling out instance variables referring to resources after hand-off
  - Or avoiding such variables
- Resource Pools can be used to hold unused resources
  - Or just let them be garbage collected
Assembly Line Example

Boxes are flow elements
  - Have adjustable dimension and color
  - Can clone and draw themselves

Sources produce continuous stream of BasicBoxes

Boxes are pushed through stages
  - Stages paint, transform, combine into composite boxes

A viewer applet serves as the sink

See CPJ p233-248 for most code omitted here
  - Some code here differs in minor ways for sake of illustration
Interfaces

interface PushSource { void start(); }

interface PushStage { void putA(Box p); }

interface DualInputPushStage extends PushStage {
    public void putB(Box p);
}
Adapters

class DualInputAdapter implements PushStage {
    protected final DualInputPushStage stage_; 
    DualInputAdapter(DualInputPushStage stage) {
        stage_ = stage;
    }
    void putA(Box p) { stage_.putB(p); }
}

Allows all other stages to issue putA

- Use adapter when necessary to convert to putB
- Simplifies composition

Alternatively, could have used a single put(command) interface

- Would require each stage to decode type/sense of command
Connections

```java
class SingleOutputPushStage {
    protected PushStage next1_ = null;
    void attach1(PushStage s) { next1_ = s; }
}

class DualOutputPushStage extends SingleOutputPushStage {
    protected PushStage next2_ = null;
    void attach2(PushStage s) { next2_ = s; }
}
```

Alternatively, could have used a collection (Vector etc) of nexts

We assume/require all attaches to be performed before any puts
class Painter extends SingleOutputPushStage
    implements PushStage {
    protected final Color color_; 

    public Painter(Color c) {
        super();
        color_ = c;
    }

    public void putA(Box p) {
        p.color(color_);
        next1_.putA(p);
    }
}

Painter is immutable after initialization
public class Collector
    extends SingleOutputPushStage
    implements DualInputPushStage {

    public synchronized void putA(Box p) {
        next1_.putA(p);
    }

    public synchronized void putB(Box p) {
        next1_.putA(p);
    }

}

Synchronization used here to illustrate flow control, not safety
class Joiner extends SingleOutputPushStage  
Implement DualInputPushStage {  
protected Box a_ = null;  // incoming from putA  
protected Box b_ = null;  // incoming from putB  
protected abstract Box join(Box p, Box q);  
protected synchronized Box joinFromA(Box p) {  
    while (a_ != null)  // wait until last consumed  
        try { wait(); }  
        catch (InterruptedException e) {return null;}  
    a_ = p;  
    return tryJoin();  
}  
protected synchronized Box tryJoin() {  
    if (a_ == null || b_ == null) return null;  
    Box joined = join(a_, b_);  // make combined box  
    a_ = b_ = null;  // forget old boxes  
    notifyAll();  // allow new puts  
    return joined;  
}  
void putA(Box p) {  
    Box j = joinFromA(p);  
    if (j != null) next1_.putA(j);  
}  
}  // (mechanics for putB are symmetrical)
Dual Output Stages

```java
class Cloner extends DualOutputPushStage
    implements PushStage {

    protected synchronized Box dup(Box p) {
        return p.duplicate();
    }

    public void putA(final Box p) {
        Box p2 = dup(p); // synched update (not nec.)
        Runnable r = new Runnable() {
            public void run() { next1_.putA(p); }
        };
        new Thread(r).start(); // use new thread for A
        next2_.putA(p2); // current thread for B
    }
}

Using second thread for second output maintains liveness
```
Configuration

All setup code is of form

```java
Stage aStage = new Stage();
aStage.attach(anotherStage);
```

Would be nicer with a visual scripting tool
Parallel Execution

Classic parallel programming deals with

- Tightly coupled, fine-grained multiprocessors
- Large scientific and engineering problems

Speed-ups from parallelism are possible in less exotic settings

- SMPs, Overlapped I/O

Key to speed-up is independence of tasks

- Minimize thread communication and synchronization
- Minimize sharing of resource objects

Rely on groups of thread-based objects

- Worker thread designs
- Scatter/gather designs
Interacting with Groups

Group Proxies encapsulate a group of workers and protocol

```
class GroupProxy implements Service {
    public Result serve(Data data) {
        split the data into parts;

        for each part p
            start up a thread to process p;

        for each thread t {
            collect results from t; // via callback or join
            if (have enough results) // one, all, or some
                return aggegrate result;
        }
    }
}
```
public class GroupPictureRenderer {

    public Picture[] render(final byte[][] data)
    throws InterruptedException {

        int n = data.length;
        Thread threads[] = new Thread[n];
        final Picture results[] = new Picture[n];

        for (int k = 0; k < n; k++) {
            final int i = k; // inner vars must be final
            threads[i] = new Thread(new Runnable() {
                public void run() {
                    PictureRenderer r = new PictureRenderer();
                    results[i] = r.render(data[i]);
                }
            });
            threads[i].start();
        }

        // block until all are finished
        for (int k = 0; k < n; k++) threads[k].join();

        return results;
    }
}
Iteration using Cyclic Barriers

CyclicBarrier is a synchronization tool for iterative group algorithms.

- Initialize count with number of members.
- `synch()` waits for zero, then resets to initial count.

```java
class PictureProcessor {
    public void processPicture(final byte[][] data) {
        final CyclicBarrier barrier =
            new CyclicBarrier(NWORKERS);
        for (int ii = 0; ii < NWORKERS; ++ii) {
            final int i = ii;
            Runnable worker = new Runnable() {
                public void run() {
                    while (!done()) {
                        transform(data[i]);
                        try {
                            barrier.barrier();
                        } catch (InterruptedException e) { return; }
                        combine(data[i], data[(i+1)%NWORKERS]);
                    }
                    new Thread(worker).start();
                }
            };
            new Thread(worker).start();
        }
    }
}
```
Implementing Cyclic Barriers

class CyclicBarrier {

    private int count_;  
    private int initial_;  
    private int resets_ = 0;  

    CyclicBarrier(int c) { count_ = initial_ = c; }  

    synchronized boolean barrier() throws InterruptedException{
        if (--count_ > 0) {  // not yet tripped
            int r = resets_;  // wait until next reset
            do { wait(); } while (resets_ == r);
            return false;
        }
        else {
            count_ = initial_;  
            ++resets_;  
            notifyAll();  
            return true;  // return true if caller tripped
        }
    }
}
Layered Services

Providing concurrency control for methods of **internal** objects

- Applying special synchronization policies
- Applying different policies to the same mechanisms

Requires visibility control (containment)

- Inner code must be **communication-closed**
- No unprotected calls in to or out from island
- Outer objects **must never leak identities** of inner objects
- Can be difficult to enforce and check

Usually based on before/after methods
Three-Layered Application Designs

Interaction with external world
- generating threads

Concurrent Control
- locking, waiting, failing

Basic mechanisms

Common across many concurrent applications
Generally easy to design and implement
Maintain directionality of control and locking
Before/After Control

Control access to contained object/action via a method of the form

```java
void controlled() {
    pre();
    try { action(); }
    finally { post(); }
}
```

Used by built-in Java `synchronized(obj) { action(); }

- Pre: `{‘ obtains lock ... Post: ‘}’ releases lock

Control code must be separable from ground action code

- Control code deals only with execution state
- Ground code deals only with intrinsic state

Basis for many delegation-based designs
Subclassing is one way to implement before/after containment designs

- Superclass instance variables and methods are “contained” in subclass instances

Template methods

- Isolate ground code and control code in overridable protected methods
- Public methods call control and ground code in an established fashion
- Can provide default versions in abstract classes
- Can override the control code and/or the ground code in subclasses
Apply when

- Methods of ground class can be separated into readers (accessors) vs writers (mutators)
  - For example, controlling access to data repository
- Any number of reader threads can run simultaneously, but writers require exclusive access

Many policy variants possible

- Mainly surrounding precedence of waiting threads
  - Readers first? Writers first? FIFO?
public abstract class RW {
    protected int activeReaders_ = 0; // exec state
    protected int activeWriters_ = 0;
    protected int waitingReaders_ = 0;
    protected int waitingWriters_ = 0;

    public void read() {
        beforeRead();
        try { doRead(); } finally { afterRead(); }  
    }
    public void write(){
        beforeWrite();
        try { doWrite(); } finally { afterWrite(); }  
    }

    protected boolean allowReader() {
        return waitingWriters_ == 0 &&
            activeWriters_  == 0;
    }
    protected boolean allowWriter() {    
        return activeReaders_ == 0 &&
            activeWriters_ == 0;
    }
}
Readers & Writers (continued)

protected synchronized void beforeRead() {
    ++waitingReaders_;
    while (!allowReader())
        try { wait(); }  
        catch (InterruptedException ex) { ... }
    --waitingReaders_;    
    ++activeReaders_;  
}

protected abstract void doRead();

protected synchronized void afterRead()  {
    --activeReaders_;    
    notifyAll();
}

protected synchronized void beforeWrite() {
    ++waitingWriters_;
    while (!allowWriter())
        try { wait(); }
        catch (InterruptedException ex) { ... }
    --waitingWriters_;    ++activeWriters_;  }

protected abstract void doWrite();

protected synchronized void afterWrite() {
    --activeWriters_;    notifyAll();
}
Using Concurrency Libraries

Library classes can help separate responsibilities for

Choosing a policy; for example

— Exclusive versus shared access
— Waiting versus failing
— Use of privileged resources

Applying a policy in the course of a service or transaction

— These decisions can occur many times within a method

Standard libraries can encapsulate intricate synchronization code

But can add programming obligations

• Correctness relies on all objects obeying usage policy
• Cannot automatically enforce

Examples

• Synchronization, Channels, Transactions
Interfaces

Sync encompasses many concurrency control policies

public interface Sync {
    // Serve as a gate, fail only if interrupted
    void acquire() throws InterruptedException;
    // Possibly allow other threads to pass the gate
    void release();
    // Try to pass for at most timeout msecs,
    // return false if fail
    boolean attempt(long timeOut);
}

Sync
void acquire()
void release()
boolean attempt(long timeOut)

Service
service(...) {
    cond.acquire();
    try {
        action();
    } finally {
        cond.release();
    }
}

ConcreteSync

implementations
Synchronization Libraries

Semaphores
• Maintain count of the number of threads allowed to pass

Latches
• Boolean conditions that are set once, ever

Barriers
• Counters that cause all threads to wait until all have finished

Reentrant Locks
• Java-style locks allowing multiple acquisition by same thread, but that may be acquired and released as needed

Mutexes
• Non-reentrant locks

Read/Write Locks
• Pairs of conditions in which the readLock may be shared, but the writeLock is exclusive
Semaphores

Conceptually serve as permit holders

- Construct with an initial number of permits (usually 0)
- require waits for a permit to be available, then takes one
- release adds a permit

But in normal implementations, no actual permits change hands.

- The semaphore just maintains the current count.
- Enables very efficient implementation

Applications

- Isolating wait sets in buffers, resource controllers
- Designs that would otherwise encounter missed signals
  - Where one thread signals before the other has even started waiting
  - Semaphores ‘remember’ how many times they were signalled
Counter Using Semaphores

class BoundedCounterUsingSemaphores {

    long count_ = MIN;

    Sync decPermits_= new Semaphore(0);  
    Sync incPermits_= new Semaphore(MAX-MIN);

    synchronized long value() { return count_; }

    void inc() throws InterruptedException {
        incPermits_.acquire();
        synchronized(this) { ++count_; }
        decPermits_.release();
    }

    void dec() throws InterruptedException {
        decPermits_.acquire();
        synchronized(this) { --count_; }
        incPermits_.release();
    }
}

This uses native synch for update protection, but only inside permit blocks. This avoids nested monitor lockouts
Semaphore Synchronous Channel

class SynchronousChannelVS {
    Object item = null;

    Semaphore putPermit = new Semaphore(1);
    Semaphore takePermit = new Semaphore(0);

    Semaphore ack = new Semaphore(0);

    void put(Object x) throws InterruptedException {
        putPermit.acquire();
        item = x;
        takePermit.release();
        ack.acquire();
    }

    Object take() throws InterruptedException {
        takePermit.acquire();
        Object x = item;
        putPermit.release();
        ack.release();
        return x;
    }
}
Using Latches

Conditions starting out false, but once set true, remain true forever

- Initialization flags
- End-of-stream conditions
- Thread termination
- Event occurrences

```java
class Worker implements Runnable {
    Latch go;
    Worker(Latch l) { go = l; }
    public void run() {
        go.acquire();
        doWork();
    }
}
class Driver { // ...
    void main() {
        Latch go = new Latch();
        for (int i = 0; i < N; ++i) // make threads
            new Thread(new Worker(go)).start();
        doSomethingElse(); // don’t let run yet
        go.release(); // let all threads proceed
    }
}
```
Using Barrier Conditions

Count-based latches

- Initialize with a fixed count
- Each release monotonically decrements count
- All acquires pass when count reaches zero

```java
class Worker implements Runnable {
    Barrier done;
    Worker(Barrier d) { done = d; }
    public void run() {
        doWork();
        done.release();
    }
}

class Driver {
    // ...
    void main() {
        Barrier done = new Barrier(N);
        for (int i = 0; i < N; ++i)
            new Thread(new Worker(done)).start();
        doSomethingElse();
        done.acquire(); // wait for all to finish
    }
}
```
Using Lock Classes

class HandSynched {
    private double state_ = 0.0;
    private Sync lock_;

    HandSynched(Sync l) { lock_ = l; }

    void changeState(double d) {
        try {
            lock_.acquire();
            try {
                state_ = d;
            } finally {
                lock_.release();
            }
        } catch(InterruptedException ex) { }
    }

    double getState() {
        double d = 0.0;
        try {
            lock_.acquire();
            try {
                d = state_;
            } finally {
                lock_.release();
            }
        } catch(InterruptedException ex) {} return d;
    }
}
Wrapper Classes

Standardize common client usages of custom locks using wrappers

- Wrapper class supports `perform` method that takes care of all before/after control surrounding a `Runnable` command sent as a parameter

- Can also standardize failure control by accepting `Runnable` action to be performed on acquire failure

Alternative `perform` methods can accept blocks that return results and/or throw exceptions

- But need to create new interface type for each kind of block

Similar to macros in other languages

- But implement more safely via inner classes
  - Wrappers are composable

Adds noticeable overhead for simple usages

- Most useful for controlling “heavy” actions
Before/After Wrapper Example

class WithLock {
    Sync cond;
    public WithLock(Sync c) { cond = c; }

    public void perform(Runnable command)
        throws InterruptedException {
        cond.acquire();
        try {
            command.run();
        } finally {
            cond.release();
        }
    }

    public void perform(Runnable command,
                        Runnable onInterrupt) {
        try {
            perform(command);
        } catch (InterruptedException ex) {
            if (onInterrupt != null)
                onInterrupt.run();
            else // default
                Thread.currentThread().interrupt();
        }
    }
}
Using Wrappers

class HandSynchedV2 {  // ...
    private double state_ = 0.0;
    private WithLock withlock_;

    HandSynchedV2(Sync l) {
        withlock_ = new WithLock(l);
    }

    void changeState(double d) {
        withlock_.perform(
            new Runnable() {
                public void run() { state_ = d; } },
            null); // use default interrupt action
    }

    double getState() {
        // (need to define interface & perform version)
        try {
            return withLock_.perform(new DoubleAction(){
                public void run() { return state_; } });
        }
        catch(InterruptedException ex){return 0.0;}
    }
}
Using Conditional Locks

`Sync.attempt` can be used in conditional locking idioms

Back-offs

- Escape out if a lock not available
- Can either retry or fail

Reorderings

- Retry lock sequence in different order if first attempt fails

Heuristic deadlock detection

- Back off on time-out

Precise deadlock detection

- Implement `Sync` via lock manager that can detect cycles
class Cell {
    long value;
    Sync lock = new SyncImpl();
    void swapValue(Cell other) {
        for (;;) {
            try {
                lock.acquire();
                try {
                    try {
                        if (other.lock.attempt(100)) {
                            try {
                                long t = value; value = other.value;
                                other.value = t;
                                return;
                            }
                            finally { other.lock.release(); }
                        }
                    } finally { lock.release(); }
                }
                catch (InterruptedException ex) { return; }
            }
            finally { lock.release(); }
            try {
            }
            finally { other.lock.release(); }
        }
        catch (InterruptedException ex) { return; }
    }
}
class Cell {
    long value;
    Sync lock = new SyncImpl();
    private static boolean trySwap(Cell a, Cell b) {
        a.lock.acquire();
        try {
            if (!b.lock.attempt(0)) return false;
            try {
                long t = a.value;
                a.value = b.value;
                b.value = t;
                return true;
            } finally { other.lock.release(); }
        } finally { lock.release(); }
        return false;
    }
    void swapValue(Cell other) {
        while (!trySwap(this, other) &&
            !trySwap(other, this)) Thread.yield();
    }
}
Using Read/Write Locks

```java
public interface ReadWriteLock {
    Sync readLock();
    Sync writeLock();
}
```

Sample usage using wrapper

```java
class WithRWLock {
    ReadWriteLock rw;

    public WithRWLock(ReadWriteLock l) { rw = l; }

    public void performRead(Runnable readCommand)
        throws InterruptedException {
        rw.readLock().acquire();
        try {
            readCommand.run();
        } finally {
            rw.readLock().release();
        }
    }

    public void performWrite(...) // similar
}
```
Transaction Locks

Associate keys with locks

- Each key corresponds to a different transaction.
  - `Thread.currentThread()` serves as key in reentrant Java synchronization locks
- Supply keys as arguments to gating methods
- Security frameworks can use similar interfaces, adding mechanisms and protocols so keys serve as capabilities

Sample interface

```java
interface TransactionLock {

    void begin(Object key); // bind key with lock

    void end(Object key);   // get rid of key

    void acquire(Object key)
        throws InterruptedException;

    void release(Object key);
}
```
Transactional Classes

Implement a common transaction control interface, for example:

```java
interface Transactor {

    // enter a new transaction
    void join(Object key) throws Failure;

    // return true if transaction can be committed
    boolean canCommit(Object key);

    // update state to reflect current transaction
    void commit(Object key) throws Failure;

    // roll back state
    void abort(Object key);
}
```

Transactors must ensure that all objects they communicate with are also Transactors

- Control arguments must be propagated to all participants
Per-Method Transaction Control

Add transaction control argument to each method.

For example:

```java
interface TransBankAccount extends Transactor {
    long balance(Object key)
            throws InterruptedException;

    void deposit(Object key, long amount)
            throws InsufficientFunds,
                    InterruptedException;

    void withdraw(Object key, long amount)
            throws InsufficientFunds,
                    InterruptedException;
}
```

The same interfaces can apply to optimistic transactions

- Use interference detection rather than locking.
- They are generally interoperable
Per-ThreadGroup Transaction Control

Assumes each transaction established in own ThreadGroup

class Context {  // ...
    Object   get(Object name);
    void     bind(Object name, Object val);
}

class XTG extends ThreadGroup {  // ...
    Context getContext();
}

class Account extends Transactor {
    private long balance_;  
    private TransactionLock tlock_;  
    // ...
    void deposit(long amount) throws ... {
        tlock_.acquire(((XTG)
            (Thread.currentThread().getThreadGroup()))
            .getContext().get("TransactionID");
        synchronized (this) {
            if (amount >= 0) balance_ += amount; else ...
        }
    }
}
Integrating Control Policies

Dealing with multiple contextual domains, including

- **Security**: Principal identities, keys, groups, etc
- **Synchronization**: Locks, conditions, transactions, ...
- **Scheduling**: Priorities, timing, checkpointing, etc
- **Environment**: Location, computational resources

Dealing with multiple outcomes

- Block, fail, proceed, save state, commit state, notify, ...

Encapsulating associated policy control information

- For example access control lists, lock dependencies

Introducing new policies in sub-actions

- New threads, conditions, rights-transfers, subtransactions
- Avoiding policy conflicts: policy compatibility matrices, ...

Avoiding excessive programming obligations for developers

Tool-based code generation, layered virtual machines
Using Integrated Control

Methods invoke helpers to make control decisions as needed

class Account { // ...

    void deposit(long amount, ...) {
        authenticator.authenticate(clientID);
        accessController.checkAccess(clientID, acl);
        logger.logDeposit(clientID, transID, amount);
        replicate.shadowDeposit(...);
        db.checkpoint(this);
        lock.acquire();

        balance += amount;

        lock.release();
        db.commit(balance, ...);
        UIObservers.notifyOfChange(this);
    }
}

Not much fun to program.
Implementing Library Classes

Classes based on Java monitor methods can be slow

- Involve context switch, locking, and scheduling overhead
- Relative performance varies across platforms

Some performance enhancements

State tracking

- Only notify when state changes known to unblock waits

Isolating wait sets

- Only wake up threads waiting for a particular state change

Single notifications

- Only wake up a single thread rather than all waiting threads

Avoiding locks

- Don’t lock if can be sure won’t wait

Can lead to significantly faster, but more complex and fragile code
Tracking State in Guarded Methods

Partition action control state into categories with same enabling properties

<table>
<thead>
<tr>
<th>State</th>
<th>Condition</th>
<th>inc</th>
<th>dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>value == MAX</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>middle</td>
<td>MIN &lt; value &lt; MAX</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>bottom</td>
<td>value == MIN</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

Only provide notifications when making a state transition that can ever unblock another thread

- Here, on exit from top or bottom
  - When count goes up from MIN or down from MAX
- Still need `notifyAll` unless add instrumentation

State tracking leads to faster but more fragile code

- Usually many fewer notification calls
- Harder to change guard conditions
- Harder to add subclasses with different conditions
class FasterGuardedBoundedCounter {
    protected long count_ = MIN;
    synchronized long value() { return count_; }

    synchronized void inc()
        throws InterruptedException {
        while (count_ == MAX) wait();
        if (count_++ == MIN) notifyAll();
    }

    synchronized void dec()
        throws InterruptedException {
        while (count_ == MIN) wait();
        if (count_-- == MAX) notifyAll();
    }
}
Buffer with State Tracking

class BoundedBufferVST {
    Object[] data_;  
    int putPtr_ = 0, takePtr_ = 0, size_ = 0;

    protected void doPut(Object x) {
        data_[putPtr_] = x;
        putPtr_ = (putPtr_ + 1) % data_.length;
        if (size_++ == 0) notifyAll();
    }

    protected Object doTake() {
        Object x = data_[takePtr_];
        data_[takePtr_] = null;
        takePtr_ = (takePtr_ + 1) % data_.length;
        if (size_-- == data_.length) notifyAll();
        return x;
    }

    synchronized void put(Object x) throws InterruptedException {
        while (isFull()) wait();
        doPut(x);
    }
    // ...
}
Inheritance Anomaly Example

class XBuffer extends BoundedBufferVST {
    synchronized void putPair(Object x, Object y) throws InterruptedException {
        put(x);
        put(y);
    }
}

PutPair does not guarantee that the pair is inserted contiguously.
To ensure contiguity, try adding guard:

    while (size_ > data_.length - 2) wait();

But doTake only performs notifyAll when the buffer transitions from full to not full:

- The wait may block indefinitely even when space available
- So must rewrite doTake to change notification condition

Would have been better to factor out the notification conditions in a separate overridable method:

- Most inheritance anomalies can be avoided by fine-grained (often tedious) factoring of methods and classes
Isolating Waits and Notifications

Mixed condition problems

- Threads that wait in different methods in the same object may be blocked for different reasons — for example, *not Empty vs not Full* for buffer
- `notifyAll` wakes up all threads, even those waiting for conditions that could not possibly hold

Can isolate waits and notifications for different conditions in different objects — an application of *splitting*

Thundering herd problems

- `notifyAll` may wake up many threads
- Often, at most one of them will be able to continue

Can solve by using `notify` instead of `notifyAll` only when

→ All threads wait on same condition
→ At most one thread could continue anyway

- That is, when it doesn’t matter which one is woken, and it doesn’t matter that others aren’t woken
Implementing Reentrant Locks

```java
final class ReentrantLock implements Sync {
    private Thread owner_ = null;
    private int    holds_ = 0;

    synchronized void acquire() throws Interruption {
        Thread caller = Thread.currentThread();
        if (caller == owner_) ++holds_;
        else {
            try { while (owner_ != null) wait(); }
            catch (InterruptedException e) {
                notify(); throw e;
            }
            owner_ = caller; holds_ = 1;
        }
    }

    synchronized void release() {
        Thread caller = Thread.currentThread();
        if (caller != owner_ || holds_ <= 0)
            throw new Error("Illegal Lock usage");
        if (--holds_ == 0) {
            owner_ = null;
            notify();
        }
    }
}
```
final class Semaphore implements Sync {
    int permits_; int waits_ = 0;

    Semaphore(int p) { permits_ = p; }

    synchronized void acquire() throws InterruptedException {
        if (permits_ <= waits_) {
            ++waits_; 
            try {
                do { wait(); } while (permits_ == 0);
            } catch(InterruptedException ex) {
                --waits_; notify(); throw ex;
            }
            --waits_;
        }
        --permits_;
    }

    synchronized void release() {
        ++permits_; 
        notify();
    }
}
Implementing Latches

Exploit set-once property to avoid locking using double-check:

Check status without even locking

- If set, exit — no possibility of conflict or stale read
- Otherwise, enter standard locked wait

But can have surprising effects if callers expect locking for sake of memory consistency.

```java
final class Latch implements Sync {
    private boolean latched_ = false;

    void acquire() throws InterruptedException {
        if (!latched_)
            synchronized(this) {
                while (!latched_) wait();
            }
    }

    synchronized void release() {
        latched_ = true;
        notifyAll();
    }
}
```
Implementing Barrier Conditions

Double-check can be used for any **monotonic** variable that is tested only for a threshold value.

CountDown Barriers monotonically decrement counts

- Tests against zero cannot encounter conflict or staleness

(This technique does not apply to Cyclic Barriers)

class CountDown implements Sync {
    private int count_;  
    CountDown(int initialc) { count_ = initialc; }  
    void acquire() throws InterruptedException {
        if (count_ > 0)  synchronized(this) {
            while (count_ > 0) wait();  
        }  
    }  
    synchronized void release() {  
        if (--count_ == 0) notifyAll();  
    }  
}
Implementing Read/Write Locks

class SemReadWriteLock implements ReadWriteLock {

    // Provide fair access to active slot
    Sync active_ = new Semaphore(1);

    // Control slot sharing by readers
    class ReaderGate implements Sync {
        int readers_ = 0;

        synchronized void acquire() throws InterruptedException {
            // readers pile up on lock until first passes
            if (readers_++ == 0) active_.acquire();
        }

        synchronized void release() {
            if (--readers_ == 0) active_.release();
        }
    }
    Sync rGate_ = new ReaderGate();

    public Sync writeLock() { return active_; }
    public Sync readLock() { return rGate_; }
}
Documenting Concurrency

Make code understandable

- To developers who use components
- To developers who maintain and extend components
- To developers who review and test components

Avoid need for extensive documentation by adopting:

- Standard policies, protocols, and interfaces
- Standard design patterns, libraries, and frameworks
- Standard coding idioms and conventions

Document decisions

- Use javadoc to link to more detailed descriptions
- Use naming and signature conventions as shorthand clues
- Explain deviations from standards, usage limitations, etc
- Describe necessary data invariants etc

Use checklists to ensure minimal sanity
Sample Documentation Techniques

Patlet references

/** ... Uses
 * &lt;a href="tpm.html">Thread-per-Message</a> **/
void handleRequest(...);

Default naming and signature conventions

Sample rule: Unless specified otherwise, methods that can
block have signature

... throws InterruptedException

Intentional limitations, and how to work around them

/** ... NOT Threadsafe, but can be used with
 * @see XAdapter to make lockable version. **/

Decisions impacting potential subclassers

/** ... Always maintains a legal value,
 * so accessor method is unsynchronized **/
protected int bufferSize;

Certification

/** Passed safety review checklist 11Nov97 **/
Semiformal Annotations

**PRE**  — Precondition (normally unchecked)

    /** PRE:  Caller holds synch lock ... 

**WHEN**  — Guard condition (always checked)

    /** WHEN not empty return oldest ... 

**POST**  — Postcondition (normally unchecked)

    /** POST: Resource r is released... 

**OUT**  — Guaranteed message send (relays, callbacks, etc)

    /** OUT: c.process(buff) called after read... 

**RELY**  — Required property of other objects/methods

    /** RELY: Must be awakened by x.signal()...

**INV**  — Object constraint true at start/end of every activity

    /** INV:  x,y are valid screen coordinates... 

**INIT**  — Object constraint that must hold upon construction

    /** INIT: bufferCapacity greater than zero...
Safety Problem Checklist

Storage conflicts

- Failure to ensure exclusive access; race conditions

Atomicity errors

- Breaking locks in the midst of logically atomic operations

Representation inconsistencies

- Allowing dependent representations to vary independently

Invariant failures

- Failing to re-establish invariants within atomic methods for example failing to clean up after exceptions

Semantic conflicts

- Executing actions when they are logically prohibited

Slipped Conditions

- A condition stops holding in the midst of an action requiring it to hold

Memory ordering and visibility

- Using stale cached values
Liveness Problems

Lockout

- A called method never becomes available

Deadlock

- Two or more activities endlessly wait for each other

Livelock

- A retried action never succeeds

Missed signals

- A thread starts waiting after it has already been signalled

Starvation

- A thread is continually crowded out from passing gate

Failure

- A thread that others are waiting for stops

Resource exhaustion

- Exceeding memory, bandwidth, CPU limitations
Efficiency Problems

Too much locking

- Cost of using synchronized
- Cost of blocking waiting for locks
- Cost of thread cache flushes and reloads

Too many threads

- Cost of starting up new threads
- Cost of context switching and scheduling
- Cost of inter-CPU communication, cache misses

Too much coordination

- Cost of guarded waits and notification messages
- Cost of layered concurrency control

Too many objects

- Cost of using objects to represent state, messages, etc
Reusability Problems

Context dependence

- Components that are not safe/live outside original context

Policy breakdown

- Components that vary from system-wide policies

Inflexibility

- Hardwiring control, premature optimization

Policy clashes

- Components with incompatible concurrency control strategies

Inheritance anomalies

- Classes that are difficult or impossible to subclass

Programmer-hostile components

Components imposing awkward, implicit, and/or error-prone programming obligations