Containment and Monitor Methods

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() is 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
  - One or the other should be changed
Avoiding Nested Monitors

- Adapt internal containment locking pattern

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

- Owner object provides lock and wait-set
- Invert locking order so that outer lock is released before wait
- Requires special steps to maintain atomicity
- Create special condition objects eg. Semaphores, Events
- Condition methods are never invoked while holding locks

Optimistic Policies: Trying

- Isolate state into versions
  - E.g. by grouping into a helper class
- Isolate state changes to atomic commit method that swaps in new state
- On method entry
  - Save/record current state
  - Apply action to new state
- Only commit if
  - Action succeeds and current state version is unchanged
- If can't commit: fail or retry
  - Failures are clean (no side effects)
  - Retry policy is variation of a busy-wait
- Only applicable if actions fully reversible
  - No I/O or thread construction unless safely cancellable
  - All internally called methods must be undoable
Optimistic Techniques

- Variations for recording versions of mutable data:
  - Immutable helper classes
  - Version numbers
  - Transaction Ids
  - Time-stamps
- May be more efficient than guarded waits when:
  - Conflicts are rare and when running on multiple CPUs
- Retries can livelock unless proven wait-free
  - Analog of deadlock in guarded waits
  - Should arrange to fail after a certain time or number of attempts

Optimistic Bounded Counter

```java
public class OptimisticBoundedCounter {
    private final long MIN, MAX;
    private Long count; // MIN <= count <= MAX

    public OptimisticBoundedCounter(long min, long max) {
        MIN = min; MAX = max;
        count = new Long(MIN);
    }

    public long value() {
        return count().longValue();
    }

    public 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(); // record current state
            long v = c.longValue();
            if (v < MAX && commit(c, new Long(v+1)))
                break;
            Thread.yield(); // a good idea in spin loops
        }
    }

    public void dec() { /* symmetrical */}
}
```
Specifying Policies

- Some policies are per-type
  - Optimistic approaches require all methods to conform
- Some policies can be specified per-call
  - Balking vs. Guarding vs. Guarding with time-out
- Options for specifying per-call policy:
  - Extra parameters
    - `void put(Object x, long timeout )`
    - `void put(Object x, boolean balk )`
  - Different name for balking or guarding
    - `boolean tryPut( Object x ) // balking`
    - `void put( Object x ) // guarding`
  - May need different exception signatures

Thread Creation Patterns

- Three general sets of patterns for introducing concurrency:
  - Autonomous loops
    - Establishing independent cyclic behaviour
  - Oneway messages
    - Sending messages without waiting for reply or termination
      - Improves availability of sender object
  - Interactive messages (not covered—see CPJ)
    - 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();
  ```
  - Or by a specific `start` method
  - 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
  - This is the basic approach of our web server so far

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
- Once oneway message has been sent, `host` is ready to `accept` the next message
Oneway Message Styles

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Events</td>
<td>Mouse clicks, etc</td>
</tr>
<tr>
<td>Notes</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 semantic 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

Design Goals for Oneway Messages

- **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

- **Flow**
  - The activity should progress with minimal contention

- **Performance**
  - Minimize overhead and resource usage
Thread Patterns for Oneway Messages

Thread-per-Message

client → host → handler

Thread-per-Object via Worker Threads or Pools

client → host → handler

Threads-Per-Message Web Server

Return to one-shot version of `startServer` but pass each accepted connection to a new thread for processing:

```java
// WebServer14.java
Thread serverThread;

public synchronized void startServer() throws ... {
    if (serverThread != null)
        throw new IllegalStateException("Already started");
    serverThread = new Thread(new ConnectionHandler());
    serverThread.start();
}

private class ConnectionHandler implements Runnable {
    public void run() {
        // ...
        try {
            while (!Thread.interrupted()) {
                RequestHandler r =
                    new RequestHandler(server.accept());
                new Thread(r, "worker-thread").start();
            }
        } catch (InterruptedException ex) { /* ignore */ }
        catch (IOException ex) { /* report */ }
    }
```
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

Web Server Using Worker Thread

```java
public interface Channel { // buffer, queue, stream etc
  Object take() throws InterruptedException;
  void put(Object obj) throws InterruptedException;
  int size();
}

// WebServer15.java
private Channel channel = new BoundedBuffer(); // synchronized

private class ConnectionHandler implements Runnable {
  public void run() {
    RequestHandler r = null;
    try {
      while (!Thread.interrupted()) {
        r = new RequestHandler(server.accept());
        channel.put(r);
      }
    } // ... interrupt and exception handling - more complex
  }
}

private class ChannelConsumer extends Thread {
  // Exception handling elided for simplicity
  // Also for simplicity, assumes channel has only one consumer
  public void run() {
    boolean stopProcessing = Thread.interrupted();
    while (!stopProcessing || channel.size() > 0) {
      ((Runnable) channel.take()).run();
      if (!stopProcessing)
        stopProcessing = Thread.interrupted();
    }
  }
}
```

© 2003 David Holmes and Doug Lea
Channel Options

- Unbounded queues
  - Can exhaust resources if clients faster than handlers
- Bounded buffers
  - Can cause clients to block when full
- Leaky bounded buffers
  - For example, drop oldest if full
- Priority queues (run more important tasks first)
  - Must ensure fairness
- Non-blocking channels
  - Must take evasive action if put or take fail or time out

Thread Pools

- Use a collection of worker threads, not just one
  - Can limit maximum number and priorities of threads
  - Dynamic worker thread management
    - Sophisticated policy controls
  - Often faster than thread-per-message for I/O bound actions
Web Server Using Thread Pool

import EDU.oswego.cs.dl.util.concurrent.PooledExecutor;

private PooledExecutor pool;  // WebServer16
public synchronized void startServer() throws ...
// ... as before
pool = new PooledExecutor();
serverThread = new Thread(new ConnectionHandler());
serverThread.start();

private class ConnectionHandler implements Runnable {
  public void run() {
    RequestHandler r = null;
    try {
      while (!Thread.interrupted()) {
        r = new RequestHandler(server.accept());
        pool.execute(r);
      }
    }
    // ... as before
  }
}

public void shutdownServer() throws ...
// ... as before
serverThread.interrupt();
serverThread.join();
pool.interruptAll();
server.close();

Policies and Parameters for Thread Pools

- The kind of channel used as task queue
  - Unbounded queue, bounded queue, synchronous hand-off, priority queue, ordering by task dependencies, stream, socket

- Bounding resources
  - Maximum number of threads
  - Minimum number of threads
  - "Warm" versus on-demand threads
  - Keepalive interval until idle threads die
    - Later replaced by new threads if necessary

- Saturation policy
  - Block, drop, producer-runs, etc

- These policies and parameters can interact in subtle ways!
Pools in Connection-Based Designs

- For systems with many open connections (sockets), but relatively few active at any given time
- Multiplex the delegations to worker threads via polling
  - Requires underlying support for select/poll and nonblocking I/O
  - Supported in JDK1.4 `java.nio`

```java
Main
serve() {
    accept connection
    add to polled set
}

poll() {
    for each connection
    if input available
    generate task
}
```

© 2003 David Holmes and Doug Lea