CMSC 433 – Programming Language Technologies and Paradigms

Thread Creation Patterns
Thread Creation Patterns

• Autonomous loops
  – Establishing independent cyclic behaviour

• One-way 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
Autonomous Loops

• What’s going on when you execute
  – new Thread(aRunnable).start();

• Task view
  – Asynchronous method invocation
  – Common Applications: compute servers

• Actor view
  – Start an autonomous process
  – Common Applications: Animations, Simulations, Message Consumers
• Thread specifically created to do some discrete amount of work
  – Typically done for performance-related reasons
  – e.g., the MsgHandler class in the ThreadPerConnectionLoggingServer application
Simple non-reactive active objects that usually contain a run loop of form:

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

- Our WebServer examples do this
• Oneway messages are “fire-and-forget”
• There is no concern for:
  – Replies, failure status, termination of called method, order in which messages are received by handler
• Once a oneway message has been sent, the host is ready to accept the next message
Oneway Message Styles

- 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

<table>
<thead>
<tr>
<th>Events</th>
<th>Mouse clicks, etc</th>
</tr>
</thead>
<tbody>
<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>
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

- **Safety**
  - Host state changes should be atomic

- **Availability**
  - Minimize delay until host can accept another message

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

- **Performance**
  - Minimize overhead and resource usage
Implementation Strategies

Thread-per-Message

client → host → handler
new thread

Thread-per-Object via Worker Threads or Pools

client → host → handler
channel
worker thread
Threads-Per-Message
Web Server

Thread serverThread;
    public synchronized void startServer() {
        serverThread = new Thread(new ConnectionHandler());
        serverThread.start();
    }

private class ConnectionHandler implements Runnable {
    public void run() {
        // ...
        try {
            while (!Thread.interrupted()) {
                (new Thread(new RequestHandler(server.accept()))).start();
            }
        } catch (...) { /* report */ }
    }
}
private class RequestHandler implements Runnable {
    private final Socket sock;
    public RequestHandler(Socket sock) {
        this.sock = sock;
    }
    public void run() {
        try {
            processRequest(sock);
        } catch (...) { /* report */ }
    }
    ...
}
Using 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:
    – while (!Thread.interrupted()) {
      m = channel.take();
      process(m);
    }
• Common variants
  – Pools
    • Use more than one worker thread
  – Listeners
    • Notify consumer when messages are ready
private Channel channel = new BoundedBuffer(); // synchronized

private class ConnectionHandler implements Runnable {
    public void run() {
        try {
            while (!Thread.interrupted()) {
                channel.put(new RequestHandler(server.accept()));
            }
        }
    }
}

private class ChannelConsumer extends Thread {
    // 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();
        }
    }
}
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
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
Policies & 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/Minimum number of threads
  – “Warm-started” versus on-demand threads
  – Keepalive interval until idle threads die

• Saturation policy
  – Block, drop, etc
Interactive Messages

- Client sends oneway message to Server

  ![Diagram showing client sending message to server]

- Server later invokes callback method on client
  - Callback can be either oneway or procedural

  ![Diagram showing callback method call between client and server]
Interactive Messages

• Applications
  – Observer/Listener designs
  – Completion indications from file and network I/O
  – Threads performing computations that yield results
Observer Pattern

- **Problem**
  - Dependent must be consistent with master’s state

- **Solution structure: Four kinds of objects**
  - Abstract subject (master)
    - Maintains list of dependents
  - Abstract observer (dependents)
    - Defines protocol for updating dependents
  - Concrete subject
    - Manages data for dependents; notifies them when master changes
  - Concrete observers
    - Gets new subject state upon receiving update message
Observer Pattern

- **Subject**
  - Attach(Observer)
  - Detach(Observer)
  - Notify()
  - observers

- **Observer**
  - Update()

- **ConcreteObserver**
  - Update()
  - observerState

- **ConcreteSubject**
  - GetState()
  - SetState()
  - subjectState

- for all o in observers {
  - o→Update()
- }

- observerState = subject→GetState()
Use of Observer Pattern
class Observable{
    protected double val = 0.0;
    public synchronized double getValue(){ return val; }
    protected synchronized void setValue( double d ){ val = d; }
    protected CopyOnWriteList<Observer> obs =
        CopyOnWriteList<Observer>();
    public void attach( Observer o ) { obs.add(o); }
}
public void changeValue(double newstate) {
    setValue(newstate);
    for (Observer o : obs) {
        new Thread(new Runnable() {
            public void run() {
                o.changeNotification(this);
            }
        }).start();
    }
    ...
}
class Observer {
    protected double cachedState; // last known state
    protected Subject subj;
    Observer(Subject s) { subj = s; }
    synchronized void changeNotification(Subject s) {
        cachedState = subj.getValue();
        display();
    }
    synchronized void display() {
        System.out.println(cachedState);
    }
}