**Containment and Monitor Methods**

```java
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

© 2003 David Holmes and Doug Lea
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 e.g., 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 */}
}
```

© 2003 David Holmes and Doug Lea
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
    ```java
    void put(Object x, long timeout )
    void put(Object x, boolean balk )
    ```
  - Different name for balking or guarding
    ```java
    boolean tryPut( Object x ) // balking
    void put( Object x ) // guarding
    ```
  - May need different exception signatures

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
Thread Patterns for Oneway Messages

Thread-per-Message

Thread-per-Object via Worker Threads or Pools

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();
            }
        }
    }
}
```
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();
        }
    }
}
```