CMSC 433 – Programming Language Technologies and Paradigms

Composing Objects
Composing Objects

• To build systems we often need to
  – Create thread safe objects
  – Compose them in ways that meet requirements while maintaining safety
Designing Thread-Safe Classes

• For each class you should know:
  – Which variables make up the object's state
  – What invariants/postconditions apply to the state
  – What policies you will use to manage concurrent access to the object's state
Object State

- Primitive fields
- References and the fields reachable from those references
Object Invariants

• Invariants are logical statements that must be true about an object’s state, e.g.,
  – lowerBound ≤ upperBound
  – List l is sorted in ascending order

• Postconditions capture the expected effect of an operation, e.g.,
  – For list l, after l.add(x) completes l.contains(x)
Synchronization Policy

- Invariants/postconditions must hold under concurrent access
- If operations can violate invariants/postconditions
  - Operation must be atomic
- If invariants involve multiple variables
  - Must fetch and update all variables in an atomic operation
  - All accesses to any of these variables must be guarded by the same lock
public final class Counter {
    // shared mutable state
    private long value = 0;
    // returns current value
    public synchronized long getValue() {
        return value;
    }
    // increments current value by 1
    public synchronized long increment() {
        if (value == Long.MAX_VALUE)
            throw new IllegalStateException("counter overflow");
        return ++value;
    }
}
public class BuggyNumberRange {
    // INVARIANT: lower <= upper
    private volatile int lower = 0;    private volatile int upper = 0;
    public void setLower(int i) {
        if (i > upper)   throw new IllegalArgumentException();
        lower = i;
    }
    public void setUpper(int i) {
        if (i < lower)   throw new IllegalArgumentException();
        upper = i;
    }

    public boolean isInRange(int i) {
        return (i >= lower && i <= upper);
    }
}
public class SimpleNumberRange {
    private int lower = 0;    private int upper = 0;
    public synchronized void setLower(int i) {
        if (i > upper)  throw new IllegalArgumentException();
        lower=i;
    }
    public synchronized void setUpper(int i) {
        if (i < lower)  throw new IllegalArgumentException();
        upper=i;
    }
    public synchronized boolean isInRange(int i) {
        return (i >= lower && i <= upper);
    }
}
State Dependent Actions

• State dependent operations are those that are legal in some states, but not in others

• Examples
  – Operations on collections
    • Can’t remove an element from an empty queue
    • Can’t add an element to a full buffer
  – Operations involving constrained values
    • Can’t withdraw money from empty bank account
  – Operations requiring resources
    • Can’t print to a busy printer
  – Operations requiring particular message orderings
    • Can’t read an unopened file
State Dependent Actions

- Some policies for handling state dependence
  - Balking
  - Guarded Suspension
  - Optimistic Retries
Policies for State Dependent Actions

- There are different ways to handle state dependence
  - Balking – Ignore or throw exception
  - Guarding – Suspend until you can proceed
  - Trying – proceed, but rollback if necessary
    - Retrying – keep trying until you succeed
    - Timing out – try for a fixed period of time
Balking

- Check state upon method entry
  - Must not change state in course of checking it
- Exit immediately if not in right state
  - Throw exception or return special error value
  - Client is responsible for handling failure
public class BalkingBoundedBuffer implements Buffer {
    private List data;
    private final int capacity;

    public BalkingBoundedBuffer(int capacity) {
        data = new ArrayList(capacity);
        this.capacity = capacity;
    }

    ...
}

public synchronized Object take() throws Failure {
    if (data.size() == 0) throw new Failure("Buffer empty");
    Object temp = data.get(0);
    data.remove(0);
    return temp;
}

public synchronized void put(Object obj) throws Failure {
    if (data.size() == capacity) throw new Failure("Buffer full");
    data.add(obj);
}

...
Guarding

- Check state upon entry
  - If not in acceptable state, wait
  - Some other thread must cause a state change that enables waiting thread to resume operation

- Generalization of locking
  - Locked: wait until not engaged in other methods
  - Guarded: wait until arbitrary predicate holds

- Introduces liveness concerns
  - Relies on actions of other threads to make progress
Guarding Mechanisms

• **Busy-waits**
  – Thread continually spins until a condition holds
    • while (!condition) ; // spin
      // use condition
  – Usually to be avoided, but can be useful when conditions latch—i.e., once set true, they never become false

• **Suspension**
  – Thread stops execution until notified that the condition may be true
  – Supported in Java via wait-sets and locks
Guarding Via Suspension

• Waiting for a condition to hold:
  synchronized (obj) {
    while (!condition) {
      try {
        obj.wait();
      } catch (InterruptedException ex) { ... }
      // make use of condition
    }
  }

• Always test a condition in a loop
  • State change may not be what you need
  • Conditions can change more than once before waiting thread resumes operation
Guarding Via Suspension

- Changing a condition:
  
  ```java
  synchronized (obj) {
    condition = true;
    obj.notifyAll(); // or obj.notify()
  }
  ```
Wait-sets and Notification

• Every Java Object has a wait-set
  – Can only manipulate it while holding Object’s lock
    • Otherwise IllegalMonitorStateException is thrown
• Threads enter Object’s wait-set by invoking wait()
  – wait() atomically releases lock and suspends thread
    • Including a lock held multiple times
    • No other held locks are released
  – Optional timed-wait: wait( long millis )
    • No direct indication that a time-out occurred
    • wait() is equivalent to wait(0) — means wait forever
Wait-sets and Notification

- Threads are released from an Object’s wait-set when:
  - `notifyAll()` is invoked on the Object
    - All threads released
  - `notify()` is invoked on the Object
    - One thread selected at ‘random’ for release
  - A specified time-out elapses
  - The Thread has its `interrupt()` method invoked
    - `InterruptedException` thrown
  - A spurious wakeup occurs
- Lock is always reacquired before `wait()` returns
  - Can’t be acquired until a notifying Thread releases it
  - Released thread contends with all other threads for the lock
  - If Lock is acquired, then Lock count is restored
• `notify()` can only be used safely when
  – Only one thread can benefit from the change of state
  – All threads are waiting for the same change of state
    • Or else another `notify()` is done by the released thread
  – These conditions hold in all subclasses
• Any Java Object can be used just for its wait-set and/or lock
public synchronized Object take() throws Failure {
    while (data.size() == 0) {
        try {
            wait();
        } catch (InterruptedException ex) { throw new Failure(); }
    }
    Object temp = data.get(0);
    data.remove(0);
    notifyAll();
    return temp
}
Guarded Bounded Buffer

```java
public synchronized void put(Object obj) throws Failure {
    while (data.size() == capacity)
        try {
            wait();
            wait();
        } catch(InterruptedException ex) { throw new Failure(); }
    data.add(obj);
    notifyAll();
}
```
notify vs. notifyAll()

- Suppose put() and take() used notify() instead of notifyAll()
- Capacity is 1
- Four threads – two just call put() and two just call take()
# Deadlock

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>data.size</th>
<th>Wait Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>take</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>T1</td>
</tr>
<tr>
<td>take</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>T1,T2</td>
</tr>
<tr>
<td>put</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>T2</td>
</tr>
<tr>
<td>put</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>T2,T3</td>
</tr>
<tr>
<td></td>
<td>put</td>
<td></td>
<td></td>
<td>1</td>
<td>T2,T3,T4</td>
</tr>
<tr>
<td>take</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>T3,T4</td>
</tr>
<tr>
<td>take</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>T1,T3,T4</td>
</tr>
<tr>
<td>take</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>T1,T2,T3,T4</td>
</tr>
</tbody>
</table>
Timing Out

- Intermediate points between balking and guarding
  - Can vary timeout parameter from zero to infinity
- Can’t be used for high-precision timing or deadlines
  - Time can elapse between wait and thread resumption
  - Time can elapse after checking the time
- Java implementation constraints
  - wait(ms) does not automatically tell you if it returned because of a notification or because of a 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
    - No way to establish with 100% certainty that timeout occurred
// assume timeout > 0
public synchronized void put(Object obj, long timeout) throws Failure {
    long timeleft = timeout;
    long start = System.currentTimeMillis();

    while (data.size() == capacity) {
        try {
            wait(timeleft);
        } catch (InterruptedException ex) { throw new Failure(); }

        if (data.size() < capacity) // notified, timed-out or spurious?
            break; // condition holds - don't care if we timed out
        else { // maybe a timeout
            long elapsed = System.currentTimeMillis() - start;
            timeleft = timeleft - elapsed;
            if (timeleft <= 0) throw new Failure("Timed-out");
        } // spurious so wait again
    }
    data.add(obj);
    notifyAll();
}
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
  - Record initial state
  - Apply action to new state
- Only commit if
  - Action succeeds and initial state was 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

• May be more efficient than guarded waits when:
  – Conflicts are rare and when running on multiple CPUs

• However, retrying can cause livelock
  – Infinite retries with no progress
  – Should arrange to fail after a certain time or number of attempts
public class OptimisticBoundedCounter {

    ...
Instance Confinement

• Even if an object is not thread-safe, there may still be ways to use it safely, e.g.,
  – Confine its use to a single thread
  – Ensure all accesses to it are guarded by a lock
public class PersonSet {
    private final Set<Person> mySet = new HashSet<Person>();
    // HashSet is not thread-safe
    public synchronized void addPerson(Person p) {
        mySet.add(p);
    }
    public synchronized boolean containsPerson(Person p) {
        return mySet.contains(p);
    }
}
Monitor Pattern

- The PersonSet class uses the Monitor Pattern
  - Object enforces mutually exclusive access to its own state
- Have to be careful when we combine monitors
public static class Statistics {
    public long requests;
    public double avgTime;
    public Statistics(long requests, double avgTime) {
        this.requests = requests;
        this.avgTime = avgTime;
    }
    ...
}

• Fields are public and mutable, so instances can’t be shared
class Container{ …
   private final Statistics stats = new Statistics(0,0.0);
   public synchronized Statistics getStatistics() {
      return new Statistics(stats.requests, stats.avgTime);
   }
   private void someFunc() {
      …
      synchronized(this) {
         double total = stats.avgTime*stats.requests + elapsed;
         stats.avgTime = total / (++stats.requests);
      }
   }
   • Can use it in another class
   • Don’t want to expose mutable state so we make copies of it
• Strict containment creates islands of objects
  – Applies recursively
• Allows inner code to run faster
  – Can be used with legacy sequential code
• Requires inner code to be communication closed
  – No unprotected calls into or out of island
• Requires outer objects to never leak inner references
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 if Whole.rely() is called while cond is false?
• When thread T calls Whole.rely
  – T waits in part
  – While suspended T still holds lock on Whole
  – No other thread will ever unblock T via Whole.set
  • Nested Monitor Lockout
Avoiding Nested Monitors

• Possible fix
  • Let owner object provide lock and wait-set

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

- Parts are not hidden from clients
- Parts use lock provided by common owner
  - Can use either internal or external conventions
Internal Containment Locking

- Parts use their owners as locks
  
  ```java
  class Part {
      protected Container owner_; // Never null
      public Container owner() {return owner_; }
      private void bareAction() { /* ... unsafe ... */ }
      public void m() {
          synchronized (owner()){ bareAction(); }
      }
  }
  ```
  
- Parts don’t deadlock when invoking each other’s methods
- Parts must be aware that they are contained
• Can require callers to provide the lock

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

• If a class is not thread-safe, can create a subclass that adds synchronization

```java
class SafeClass extends UnsafeClass{
    synchronized void foo() {
        super.foo();
    }
}
```

– and instantiate it instead

• Can also use unrelated wrapper classes and delegation