Lecture 07: Even More Java Threads

Computer Science Department, University of Crete

Parallel Programming

Based on slides by J. Foster, M. Hicks, D. Holmes, and D. Lea

Designing Objects for Concurrency

- Isolation
  - Avoid interference by not sharing
- Immutability
  - Avoid inteference by avoiding change
- Locking
  - Dynamically guarantee exclusive access
- Splitting Objects
  - Changing representation to facilitate concurrency control
- Containment
  - Guarantee exclusive control of internal components
  - Manage ownership
  - Protect unhidden components
- Alternatives to Synchronization
  - `volatile` variables and the Java Memory Model

Isolation

- Objects that are not shared cannot suffer interference
  - Heap objects accessible only from current thread
  - Parameters and local variables
    - Applies to *references*, not the objects to which they refer
  - `java.lang.ThreadLocal`
    - Simplifies access from other objects running in the same thread
  - No need for *any* synchronization
- Objects can be accessed by multiple threads as long as they are isolated to one thread at any given time
  - Transfer of ownership protocols
    - Thread 1 uses the object, hands off to Thread 2 and then never accesses the object again
  - Transfer still requires synchronization

Thread Local Data

- Suppose you want to run multiple web servers, each on one thread, each using a different document directory
- Could define a `documentRoot` field in the `WebServer` class
- Or, define the document root as a variable tied to each `Thread` object
  - The easiest way: use `java.lang.ThreadLocal`
  - Equivalent to adding instance variables to all `Thread` objects
  - No need to define subclasses or control thread creation
- All methods running can access thread local data when needed
  - Frequent use: package accessible statistics
- No interference when *all* accesses happen within the same thread

Example: ThreadLocal

```
public class WebServer {
  static final ThreadLocal documentRoot = new ThreadLocal();
   ...
  public WebServer(int port, File root) throws IOException {
     ...
    documentRoot.set(root);
  }

  private void processRequest(Socket sock) throws IOException {
    File  root = (File) documentRoot.get();
     ...
  }
   ...
}
...
```

When to use ThreadLocal

- Variables that apply *per activity*, not *per object*
  - E.g., timeout value, transaction ID, current dirctory, default parameters
- Replacement for `static` variables
  - When different threads should use different values
- Tools to eliminate the need for synchronization
  - Used internally in JVM to optimize memory allocation, lock implementations, etc.
  - E.g., per-thread caches, slabs

Stateless Objects

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

- There are no special concurrency concerns
  - No per-instance state, therefore no storage conflicts
  - No data representation, therefore no representation invariants
  - Multiple concurrent executions, therefore no liveness problems
  - No interaction with other objects, therefore no requirement for synchronization protocol
- Example: `java.lang.Math`

Immutable Objects

```
class ImmutableAdder {
  private final int offset;
  ImmutableAdder(int offset) { this.offset = offset; }
  int add(int i) { return i + offset; }
}
```

- Object state frozen upon initialization
  - Still no safety or liveness concerns
  - No interference as per-instance state never changes
  - Java `final` fields enforce most senses of immutability
- Immutability often suitable for closed Abstract Data Types
  - E.g., `String`, `Integer`, etc.

Containment

- Strict containment creates islands of objects
  - Applies recursively
- Allows code of “inner” objects to run faster
  - Works 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
  - Or uses ownership transfer protocol
- By convention, can be difficult to enforce and check

Example: Containment (1)

```
class Statistics { // Mutable!
  public long requests;
  public double avgTime;
  public Statistics(long requests, double avgTime) {
    this.requests = requests;
    this.avgTime = avgTime;
  }
}
```

- Fields are *public* and *mutable*
  - Therefore, instances cannot be shared
- Can be safely contained within a `WebServer` instance

Example: Containment (2)

```
class WebServer {
  ...
  private final Statistics stats = new Statistics(0, 0.0);
  public synchronized Statistics getStatistics() {
    return new Statistics(stats.requests, stats.avgTime);
  }
  private void processRequest(Socket sock) throws IOException {
    synchronized(this) {
      double total = stats.avgTime * stats.requests + elapsed;
      stats.avgTime = total / (++stats.requests);
    }
  }
}
```

- Cannot expose mutable state
  - Instead, make copies

Hierarchical Containment Locking

- Applies when logically contained parts are not hidden from clients
- Avoids deadlocks that could occur if parts were fully synchronized
- All parts use lock provided by the common owner
- Can use either internal or external conventions

Internal Containment Locking (1)

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

- Visible components protect themselves using their owner's locks
  - Parts do not deadlock when invoking each other's methods
  - Parts must be aware that they are contained

Internal Containment Locking (2)

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

- Implemented using inner classes
- Do not require `synchronized` blocks synchronization
  - Shared `Lock` objects
  - Transaction locks
  - etc.

External Containment Locking

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

- External: rely on clients to provide locking (client-side)
- Used in AWT
  - `java.awt.Component.getTreeLock()`
- Can sometimes avoid more locking overhead
- ... at price of fragility
  - Can manually minimize use of `synchronized`
  - Requires all callers to obey convention
  - Effectiveness depends on context
    - Breaks encapsulation
    - Does not work with fancy schemes that do not rely on `synchronized` blocks or similar methods of locking

Subclassing Unsafe Code (1)

- Assume a method written in native code

```
class HandlerHelper {
  native void mountFileSystem();
}
```

- Suppose our method `processRequest` invokes `mountFileSystem();`

Subclassing Unsafe Code (2)

- We do not trust this class to be thread-safe
  - Wrap calls in `synchronized` blocks (i.e., containment)
  - Or, create a simple subclass that adds synchronization and instantiate that class instead

```
class SafeHandlerHelper extends HandlerHelper {
  synchronized void mountFileSystem() {
    super.mountFileSystem();
  }
}
```

  - Localizes synchronization control where it is required

- Subclassing is usually the most convenient way to do that
  - Can also use unrelated wrapper classes and delegate
  - Can generalize to “template method” schemes (later)

State Dependent Actions

- State Dependence
- Balking
- Guarded Suspension
- Optimistic Retries
- Specifying Policies

Examples of State Dependent Actions

- Operations on collections, streams, databases
  - Remove an element from an empty queue
  - Add an element to a full buffer
- 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

- Policy choices for dealing with preconditions and postconditions
  - **Blind action**: Proceed anyway, no guarantee of outcome
  - **Inaction**: Ignore request if not in the right state
  - **Balking**: Fail via exception if not in the right state
  - **Guarding**: Suspend until in the right state
  - **Trying**: Proceed, check if successful, roll back if not
  - **Retrying**: Keep trying until successful
  - **Timeout**: Wait or retry for a while, then fail
  - **Planning**: First initiate activity that will achieve the right state
- How to convey policy in code?

Interfaces and Policies

```
public interface Buffer {
  int capacity();      // Inv: capacity() > 0
  int size();          // Inv: 0 ≤ size() ≤ capacity()
                       // Init: size() == 0
  void put(Object x); // Pre: size() < capacity()
  Object take();       // Pre: size() > 0
}
```

- Interfaces alone cannot convey policy
- Can suggest policy
  - E.g., should `take()` throw exception? What kind?
  - Different methods can support different policies for same base actions
- Can use manual annotations
  - Declarative constraints form the basis of the implementation

Balking

- Check state upon method entry
  - Must not change state in course of checking state
  - Relevant state must be explicitly represented
    - So it can be checked on entry
- Exit immediately if not in the right state
  - Throw exception or return special value
    - In these examples, throw `Failure`
  - Client is responsible for handling failure
- The simplest policy for synchronized objects
  - Useable in both sequential and concurrent contexts
    - Often used in `Collection` classes, e.g., `Vector`
  - 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

Example: Balking Bounded Buffer

```
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 o) throws Failure {
    if  (data.size() == capacity) throw new Failure("Buffer Full");
    data.add(o);
  }
  public synchronized int size() { return data.size(); }
  public int capacity() { return capacity; }
}
```

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
  - Client must ensure correct state before calling

Guarding Mechanisms: Busy wait

- Thread continually spins until a condition holds

```
while(!condition) ; // spin
//  use condition
```

  - Requires multiple CPUs or timeslicing
    - No way to determin this until Java 1.4

```
int nCPUs = Runtime.availableProcessors();
```

  - But busy waiting can sometimes be useful
    - When the conditions *latch*: once true, they never become false

Guarding Mechanisms: Suspension (1)

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

```
synchronized (obj) {
  while (!condition) {
    try { obj.wait(); }
    catch (InterruptedException e) { ... }
  }
  // use condition
}
```

Guarding Mechanisms: Suspension (2)

- Changing a condition

```
synchronized (obj) {
  condition = true;
  obj.notifyAll();   // or obj.notify()
}
```

  - Or after Java 1.5, using `Lock` and `Condition`
- Golden rule: always test a condition in a loop
  - Change of state may not be what you need
  - Condition may have changed again
  - Break the rule only after *proving* it's safe

Wait sets and Notification (1)

- Every Java `Object` has a wait set
  - Can only be manipulated while the object lock is held
  - Otherwise, `IllegalMonitorStateException`
- Threads enter the wait set by calling `wait()`
  - `wait()` atomically releases the lock and suspends the thread
    - Including re-entrant locks held multiple times
    - *No other* held locks are released
  - Timed waiting via `wait(long milliseconds)`
    - No direct indication that a time-out occured
    - `wait()` and `wait(0)` mean wait forever
    - Nanosecond version too
- Similar for explicit `Lock` objects after Java 1.5
  - Differences in versatility: interruption, timeout notification, separate acquire – release, etc.

Wait sets and Notification (2)

- Threads are released from the wait set when
  - `notifyAll()` invoked on the object (`signalAll()` invoked on the condition)
    - ★ Releases all threads
  - `notify()` invoked on the object (`signal()` invoked on the condition)
    - ★ Releases one thread selected at "random"
  - The specified timeout has elapsed
  - `interrupt()` method called for current thread, causes `InterruptedException`
  - Spurious wakeup occurs when:
    - ★ Inherited property of underlying synchronization mechanisms: POSIX threads, Windows threads, Hardware threads, etc.
- Lock is always reacquired before `wait()` returns
  - Restored lock count for re-entrant locks
  - Cannot be acquired until notifying thread releases it
  - All released threads contend for the lock

Wait sets and Notification (3)

- Avoid `notify()` (and `signal()`), only use for optimization when *all* the following hold:
  - 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
  - And these conditions also hold for *all subclasses*!
- Conditional notification is another optimization
  - When you *know* for what state changes the other threads wait
  - Warning: subclasses may invalidate your "knowledge"
- Use of `wait()`, `notifyAll()`, `notify()` are similar to
  - Condition queues of classic Monitors
  - Condition variables of POSIX threads
  - But, with only one queue per object
    - ★ May complicate some designs and lead to *nested monitor lockouts*
- Any Java object can be used just for its wait set and lock
  - After 1.5, use `Lock` objects

Example: Guarded Bounded Buffer

```
public class GuardedBoundedBuffer implements Buffer {
  private List data;
  private final int capacity;

  public GuardedBoundedBuffer(int capacity) {
    data = new ArrayList(capacity);
    this.capacity = capacity;
  }
  public synchronized Object take() throws Failure {
    while (data.size() == 0)
      try { wait(); }
      catch (InterruptedException e) { throw new Failure(); }
    Object temp = data.get(0);
    data.remove(0);
    notifyAll();
    return temp;
  }
  public synchronized void put(Object obj) throws Failure {
    while (data.size() == capacity)
      try { wait(); }
      catch (InterruptedException e) { throw new Failure(); }
    data.add(obj);
    notifyAll();
  }
  public synchronized int size() { return data.size(); }
  public int capacity() { return capacity; }
}
```

Timeout

- 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 disconnections
- But cannot 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 returs because of notification or timeout
    - `await(ms)` does

Optimistic Techniques

- Variations for recording versions of mutable data
  - Immutable helper classes
  - Version numbers
  - Transaction IDs
  - Time stamps
- May be more efficient than guarded waiting
  - When conflicts are rare and running on multiple CPUs
- Retrying can livelock unless *proven* wait-free
  - Analogous to deadlock in guarded waiting
  - Should arrange to fail after a certain time or number of attempts

Example: Optimistic Bounded Counter

```
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();
      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
    - Balking: `void tryPut(Object x)`
    - Guarding: `void put(Object x)`
  - May need different exception signatures