COMP 422, Lecture 11: Java Concurrency

Vivek Sarkar

Department of Computer Science
Rice University

vsarkar@rice.edu

Upcoming Class Schedule

- 2/12/08 --- Lecture 11: Java Concurrency
- 2/14/08 --- Lecture 12: X10 (x10.sf.net)
- 2/19/08 --- Lecture 13: Microsoft Task Parallel Library & PLINQ
- 2/21/08 --- Lecture 14: High Performance Fortran (Chuck Koelbel)
- 2/26/08 --- Lecture 15: Midterm Review
- 2/28/08 --- In-class midterm exam
- MIDTERM RECESS
- 3/11/08 --- Lecture 16: Message-Passing Parallelism
- . . .

Acknowledgment: OOPSLA 2007 tutorial
by Joe Bowbeer and David Holmes

http://www.oopsla.org/oopsla2007/index.php?page=sub/&id=69

Java™ Concurrency Utilities in Practice

Joe Bowbeer
Java ME Specialist
Mobile App Consulting
jozart@alum.mit.edu

David Holmes
Senior Java Technologist
Sun Microsystems Australia
David.Holmes@sun.com

Contributing authors: **Doug Lea**
State University of New York, Oswego
dl@cs.oswego.edu

Tim Peierls
BoxPop.biz
Tim@peierls.net

Brian Goetz
Sun Microsystem Inc.
Brian.Goetz@sun.com

About these slides

- Java™ is a trademark of Sun Microsystems, Inc.
- Material presented is based on latest information available for Java™ Platform Standard Edition, as implemented in JDK™ 6.0
- Code fragments elide
  - Exception handling for simplicity
  - Access modifiers unless relevant
- More extensive coverage of most topics can be found in the book
  - *Java Concurrency in Practice, by Brian Goetz et al, Addison-Wesley (JCiP)*
- See also
  - *Concurrent Programming in Java, by Doug Lea, Addison-Wesley (CPJ)*

Review: Java Threading Model

- The Java virtual machine (JVM)
  - —Creates the initial thread which executes the main method of the class passed to the JVM
  - —Creates internal JVM helper threads
    - Garbage collection, finalization, signal dispatching …
- The code executed by the ‘main’ thread can create other threads
  - —Either explicitly; or
  - —Implicitly via libraries:
    - AWT/Swing, Applets
    - Servlets, web services
    - RMI
    - image loading
    - …

Review: Java Thread Creation

- Concurrency is introduced through objects of the class `Thread`
  - Provides a ‘handle’ to an underlying thread of control
- There is always a ‘current’ thread running:
  - Static method `Thread.currentThread()`
- The `start()` method
  - Creates a new thread of control to execute the `Thread` object’s `run()` method
- Two ways to provide a `run()` method:
  - Subclass `Thread` and override `run()`
  - Define a class that implements the `Runnable` interface and get the `Thread` object to run it

```
new Thread(aRunnable).start();
```

- `Runnable` defines the abstraction of work
- `Thread` defines the abstraction of a worker

Review: Thread Interaction

- `void start()`
  - Creates a new thread of control to execute the `run()` method of the `Thread` object
  - Can only be invoked once per `Thread` object
- `void join()`
  - Waits for a thread to terminate
  - t1.join(); // blocks current thread until t1 terminates
- `static void sleep(long ms)throws InterruptedException`
  - Blocks current thread for approximately at least the specified time
- `static void yield()`
  - Allows the scheduler to select another thread to run

Review: Java Synchronization

- Every Java object has an associated lock acquired via:
  - `synchronized` statements
    - ```
      synchronized( foo ){
        // execute code while holding foo's lock
      }
      ```
  - `synchronized` methods
    - ```
      public synchronized void op1(){
        // execute op1 while holding 'this' lock
      }
      ```
- Only one thread can hold a lock at a time
  - If the lock is unavailable the thread is blocked
  - Locks are granted per-thread: reentrant or recursive locks
- Locking and unlocking are automatic
  - Can't forget to release a lock
  - Locks are released when a block goes out of scope
    - By normal means or when an exception is thrown

Review: Use of `wait/notify`

- **Waiting for a condition to hold:**

```
    synchronized (obj) {  // obj protects the mutable state
        while (!condition) {
            try { obj.wait(); }
            catch (InterruptedException ex) { ... }
        }
        // make use of condition while obj still locked
    }
```

- **Changing a condition:**

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

- **Golden rule: Always test a condition in a loop**
  - Change of state may not be what you need
  - Condition may have changed again
    - No built-in protection from ‘barging’
  - Spurious wakeups are permitted – and can occur

`java.util.concurrent`

- **General purpose toolkit for developing concurrent applications**
  - **—No more "reinventing the wheel"!**
- **Goals: "Something for Everyone!"**
  - **—Make some problems trivial to solve by everyone**
    - **Develop thread-safe classes, such as servlets, built on concurrent building blocks like `ConcurrentHashMap`**
  - **—Make some problems easier to solve by concurrent programmers**
    - **Develop concurrent applications using thread pools, barriers, latches, and blocking queues**
  - **—Make some problems possible to solve by concurrency experts**
    - **Develop custom locking classes, lock-free algorithms**

Overview of `j.u.c`

- Executors
  - `Executor`
  - `ExecutorService`
  - `ScheduledExecutorService`
  - `Callable`
  - `Future`
  - `ScheduledFuture`
  - `Delayed`
  - `CompletionService`
  - `ThreadPoolExecutor`
  - `ScheduledThreadPoolExecutor`
  - `AbstractExecutorService`
  - `Executors`
  - `FutureTask`
  - `ExecutorCompletionService`
- Queues
  - `BlockingQueue`
  - `ConcurrentLinkedQueue`
  - `LinkedBlockingQueue`
  - `ArrayBlockingQueue`
  - `SynchronousQueue`
  - `PriorityBlockingQueue`
  - `DelayQueue`
- Concurrent Collections
  - `ConcurrentMap`
  - `ConcurrentHashMap`
  - `CopyOnWriteArray{List,Set}`
- Synchronizers
  - `CountDownLatch`
  - `Semaphore`
  - `Exchanger`
  - `CyclicBarrier`
- Locks: `java.util.concurrent.locks`
  - `Lock`
  - `Condition`
  - `ReadWriteLock`
  - `AbstractQueuedSynchronizer`
  - `LockSupport`
  - `ReentrantLock`
  - `ReentrantReadWriteLock`
- Atomics: `java.util.concurrent.atomic`
  - `Atomic[Type]`
  - `Atomic[Type]Array`
  - `Atomic[Type]FieldUpdater`
  - `Atomic{Markable,Stampable}Reference`

Key Functional Groups in j.u.c.

- **Executors, Thread pools and Futures**
  - —Execution frameworks for asynchronous tasking
- **Concurrent Collections:**
  - —Queues, blocking queues, concurrent hash map, …
  - —Data structures designed for concurrent environments
- **Locks and Conditions**
  - —More flexible synchronization control
  - —Read/write locks
- **Synchronizers: Semaphore, Latch, Barrier, Exchanger**
  - —Ready made tools for thread coordination
- **Atomic variables**
  - —The key to writing lock-free algorithms

The Executor Framework

- Framework for asynchronous task execution
- Standardize asynchronous invocation
  - Framework to execute `Runnable` and `Callable` tasks

    `Runnable: void run()`

    `Callable<V>: V call() throws Exception`
- Separate submission from execution policy
  - Use `anExecutor.execute(aRunnable)`
  - Not `new Thread(aRunnable).start()`
- Cancellation and shutdown support
- Usually created via `Executors` factory class
  - Configures flexible `ThreadPoolExecutor`
  - Customize shutdown methods, before/after hooks, saturation policies, queuing

Creating Executors

- Sample `ExecutorService` implementations from `Executors`
  - `newSingleThreadExecutor`

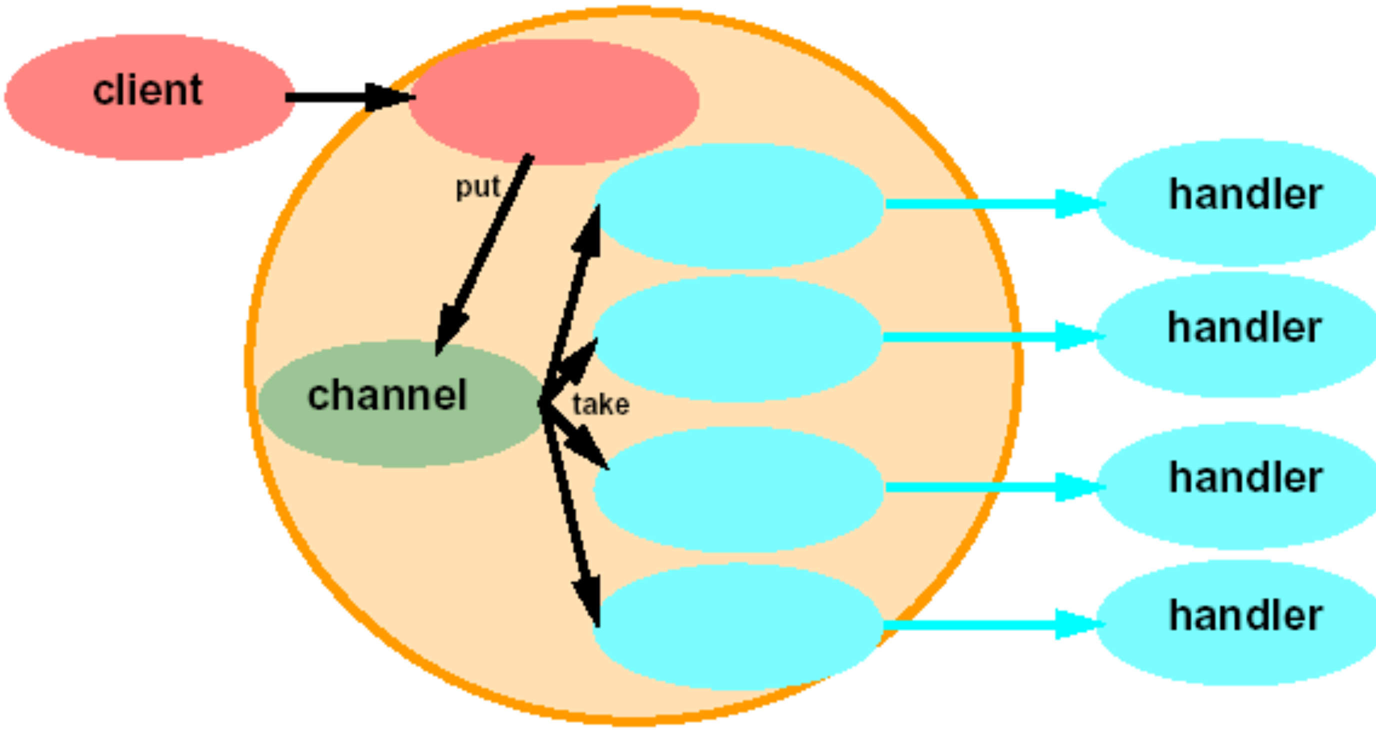
    A pool of one, working from an unbounded queue
  - `newFixedThreadPool(int N)`

    A fixed pool of N, working from an unbounded queue
  - `newCachedThreadPool`

    A variable size pool that grows as needed and shrinks when idle
  - `newScheduledThreadPool(int N)`

    Pool for executing tasks after a given delay, or periodically

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

`ThreadPoolExecutor`

- **Sophisticated `ExecutorService` implementation with numerous tuning parameters**
  - **Core and maximum pool size**
    - **Thread created on task submission until core size reached**
    - **Additional tasks queued until queue is full**
    - **Thread created if queue full until maximum size reached**
    - **Note: unbounded queue means the pool won't grow above core size**
  - **Keep-alive time**
    - **Threads above the core size terminate if idle for more than the keep-alive time**
    - **In JDK 6 core threads can also terminate if idle**
  - **Pre-starting of core threads, or else on demand**

Working with `ThreadPoolExecutor`

- `ThreadFactory` used to create new threads
  - —Default: `Executors.defaultThreadFactory`
- Queuing strategies: must be a `BlockingQueue<Runnable>`
  - —Direct hand-off via `SynchronousQueue`: zero capacity; hands-off to waiting thread, else creates new one if allowed, else task rejected
  - —Bounded queue: enforces resource constraints, when full permits pool to grow to maximum, then tasks rejected
  - —Unbounded queue: potential for resource exhaustion but otherwise never rejects tasks
- Queue is used internally
  - —Use `remove` or `purge` to clear out cancelled tasks
  - —You should not directly place tasks in the queue

    Might work, but you need to rely on internal details
- Subclass customization hooks: `beforeExecute` and `afterExecute`

Futures

- **Encapsulates waiting for the result of an asynchronous computation launched in another thread**
  - —The callback is encapsulated by the Future object
- **Usage pattern**
  - —Client initiates asynchronous computation via oneway message
  - —Client receives a “handle” to the result: a Future
  - —Client performs additional tasks prior to using result
  - —Client requests result from Future, blocking if necessary until result is available
  - —Client uses result
- **Assumes truly concurrent execution between client and task**
  - —Otherwise no point performing an asynchronous computation
- **Assumes client doesn’t need result immediately**
  - —Otherwise it may as well perform the task directly

`Future<V>` Interface

- `V get()`
  - —Retrieves the result held in this `Future` object, blocking if necessary until the result is available
  - —Timed version throws `TimeoutException`
  - —If cancelled then `CancelledException` thrown
  - —If computation fails throws `ExecutionException`
- `boolean isDone()`
  - —Queries if the computation has completed—whether successful, cancelled or threw an exception
- `boolean isCancelled()`
  - —Returns true if the computation was cancelled before it completed

Simple Future Example

- **Asynchronous rendering in a graphics application**

```
interface Pic      { byte[] getImage(); }
interface Renderer { Pic render(byte[] raw); }

class App { // sample usage
    void app(final byte[] raw) throws ... {
        final Renderer r = …;
        FutureTask<Pic> p = new FutureTask<Pic>(
            new Callable<Pic>() {
               Pic call() {
                  return r.render(raw);
               }
            });
        new Thread(p).start();
        doSomethingElse();
        display(p.get()); // wait if not yet ready
    }
    // ...
}
```

Key Functional Groups in j.u.c.

- **Executors, Thread pools and Futures**
  - —Execution frameworks for asynchronous tasking
- **Concurrent Collections:**
  - —Queues, blocking queues, concurrent hash map, …
  - —Data structures designed for concurrent environments
- **Locks and Conditions**
  - —More flexible synchronization control
  - —Read/write locks
- **Synchronizers: Semaphore, Latch, Barrier, Exchanger**
  - —Ready made tools for thread coordination
- **Atomic variables**
  - —The key to writing lock-free algorithms

Concurrent Collections

Concurrent vs. Synchronized

- Pre Java™ 5 platform: *Thread-safe but not concurrent classes*
- Thread-safe synchronized collections
  - **`Hashtable, Vector, Collections.synchronizedMap`**
  - Monitor is source of contention under concurrent access
  - Often require locking during iteration
- Concurrent collections
  - Allow multiple operations to overlap each other
    - Big performance advantage
    - At the cost of some slight differences in semantics
  - Might not support atomic operations

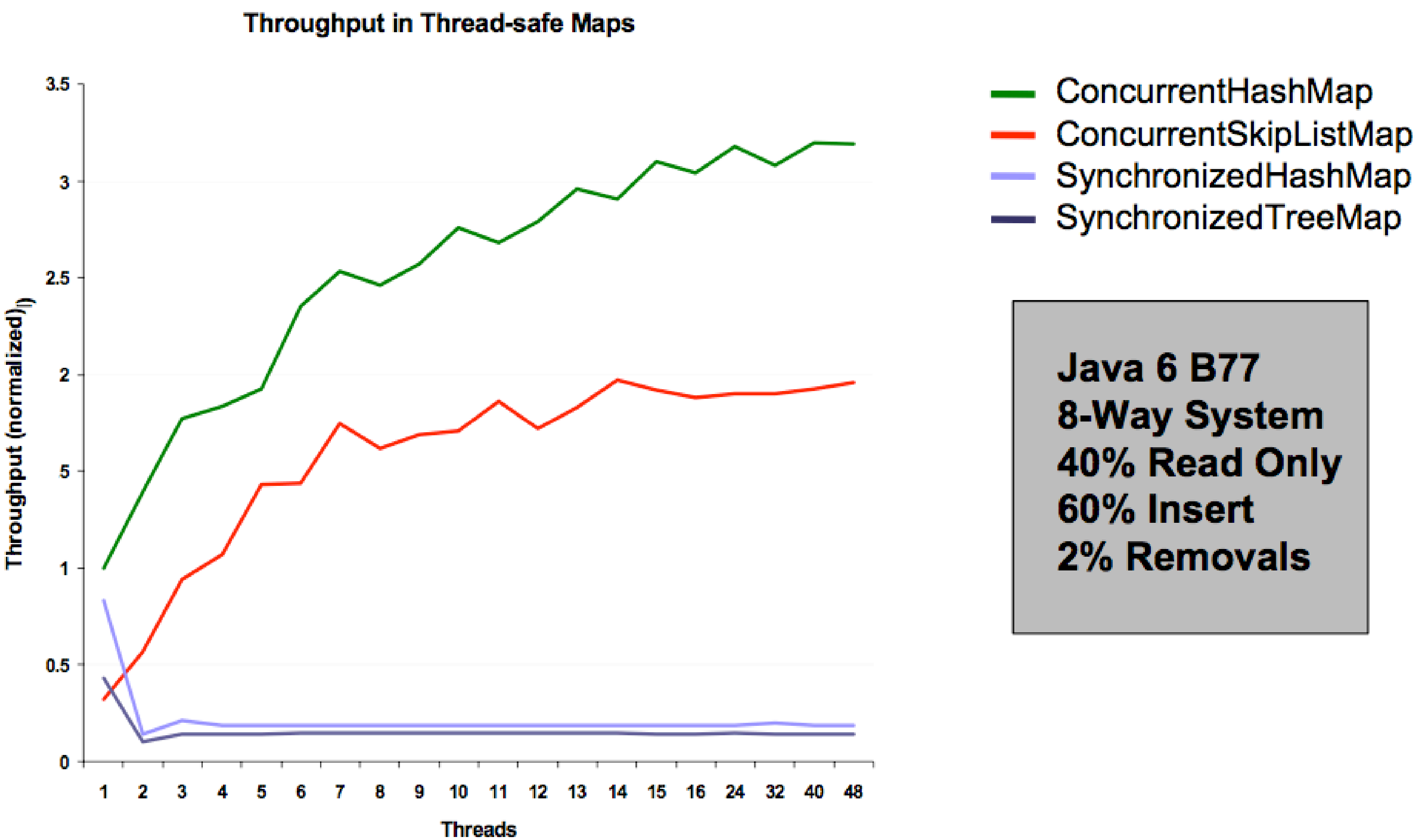
Concurrent Collections

- `ConcurrentHashMap`
  - Concurrent (scalable) replacement for **`Hashtable`** or **`Collections.synchronizedMap`**
  - Allows reads to overlap each other
  - Allows reads to overlap writes
  - Allows up to 16 writes to overlap
  - Iterators don't throw **`ConcurrentModificationException`**
- `CopyOnWriteArrayList`
  - Optimized for case where iteration is much more frequent than insertion or removal
  - Ideal for event listeners

Iteration Semantics

- Synchronized collection iteration broken by concurrent changes in another thread
  - Throws **`ConcurrentModificationException`**
  - Locking a collection during iteration hurts scalability
- Concurrent collections can be modified concurrently during iteration
  - Without locking the whole collection
  - Without **`ConcurrentModificationException`**
  - But changes may not be seen

Concurrent Collection Performance



Java 6 B77
8-Way System
40% Read Only
60% Insert
2% Removals

ConcurrentMap

- **Atomic get-and-maybe-set methods for maps**

```
interface ConcurrentMap<K,V> extends Map<K,V> {
  V putIfAbsent(K key, V value);
  V replace(K key, V value);
  boolean replace(K key, V oldValue, V newValue);
  boolean remove(K key, V value);
}
```

Key Functional Groups in j.u.c.

- **Executors, Thread pools and Futures**
  - —Execution frameworks for asynchronous tasking
- **Concurrent Collections:**
  - —Queues, blocking queues, concurrent hash map, …
  - —Data structures designed for concurrent environments
- **Locks and Conditions**
  - —More flexible synchronization control
  - —Read/write locks
- **Synchronizers: Semaphore, Latch, Barrier, Exchanger**
  - —Ready made tools for thread coordination
- **Atomic variables**
  - —The key to writing lock-free algorithms

Locks

- Use of monitor synchronization is just fine for most applications, but it has some shortcomings
  - Single wait-set per lock
  - No way to interrupt or time-out when waiting for a lock
  - Locking must be block-structured
    - Inconvenient to acquire a variable number of locks at once
    - Advanced techniques, such as hand-over-hand locking, are not possible
- Lock objects address these limitations
  - But harder to use: Need **`finally`** block to ensure release
  - So if you don't need them, stick with **`synchronized`**

Lock / ReentrantLock

```
interface Lock {
  void lock();
  void lockInterruptibly() throws InterruptedException;
  boolean tryLock();
  boolean tryLock(long timeout, TimeUnit unit)
                              throws InterruptedException;
  void unlock();
  Condition newCondition();
}
```

- Additional flexibility
  - Interruptible, try-lock, not block-structured, multiple conditions
  - Advanced uses: e.g. Hand-over-hand or chained locking
- `ReentrantLock`: mutual-exclusion `Lock` implementation
  - Same basic semantics as synchronized
    - Reentrant, must hold lock before using condition, …
  - Supports fair and non-fair behavior
    - Fair lock granted to waiting threads ahead of new requests
  - High performance under contention

Simple lock example

- Used extensively within `java.util.concurrent`

```
final Lock lock = new ReentrantLock();
...
lock.lock();
try {
  // perform operations protected by lock
}
catch(Exception ex) {
  // restore invariants & rethrow
}
finally {
  lock.unlock();
}
```

- **Must manually ensure lock is released**

Key Functional Groups in j.u.c.

- **Executors, Thread pools and Futures**
  - —Execution frameworks for asynchronous tasking
- **Concurrent Collections:**
  - —Queues, blocking queues, concurrent hash map, …
  - —Data structures designed for concurrent environments
- **Locks and Conditions**
  - —More flexible synchronization control
  - —Read/write locks
- **Synchronizers: Semaphore, Latch, Barrier, Exchanger**
  - —Ready made tools for thread coordination
- **Atomic variables**
  - —The key to writing lock-free algorithms

Synchronizers

Utility Classes for Coordinating Access and Control

- `Semaphore`—Dijkstra counting semaphore, managing a specified number of permits
- `CountDownLatch`—Allows one or more threads to wait for a set of threads to complete an action
- `CyclicBarrier`—Allows a set of threads to wait until they all reach a specified barrier point
- `Exchanger`—Allows two threads to rendezvous and exchange data
  - Such as exchanging an empty buffer for a full one

CountDownLatch

- A counter that releases waiting threads when it reaches zero
  - Allows one or more threads to wait for one or more events
  - Initial value of 1 gives a simple gate or latch

```
CountDownLatch(int initialValue)
```

- `await`: wait (if needed) until the counter is zero
  - Timeout version returns false on timeout
  - Interruptible
- `countDown`: decrement the counter if > 0
- Query: `getCount()`
- Very simple but widely useful:
  - Replaces error-prone constructions ensuring that a group of threads all wait for a common signal

Semaphores

- Conceptually serve as permit holders
  - —Construct with an initial number of permits
  - – `acquire`: waits for permit to be available, then “takes” one
  - – `release`: “returns” a permit
  - —But no actual permits change hands
    - The semaphore just maintains the current count
    - No need to acquire a permit before you release it
- “fair” variant hands out permits in FIFO order
- Supports balking and timed versions of `acquire`
- Applications:
  - —Resource controllers
  - —Designs that otherwise encounter missed signals
    - Semaphores ‘remember’ how often they were signalled

Bounded Blocking Concurrent List

- **Concurrent list with fixed capacity**
  - —**Insertion blocks until space is available**
- **Tracking free space, or available items, can be done using a Semaphore**
- **Demonstrates composition of data structures with library synchronizers**
  - —**Much, much easier than modifying implementation of concurrent list directly**

Bounded Blocking Concurrent List

```
public class BoundedBlockingList {
   final int capacity;
   final ConcurrentLinkedList list =
                              new ConcurrentLinkedList();
   final Semaphore sem;

  public BoundedBlockingList(int capacity) {
     this.capacity = capacity;
     sem = new Semaphore(capacity);
  }
   public void addFirst(Object x) throws
                                   InterruptedException {
     sem.acquire();
     try { list.addFirst(x); }
     catch (Throwable t){ sem.release(); rethrow(t); }
   }
   public boolean remove(Object x) {
     if (list.remove(x)) {
       sem.release(); return true;
     }
     return false;
   }
   …
}
```

Key Functional Groups in j.u.c.

- Executors, Thread pools and Futures
  - Execution frameworks for asynchronous tasking
- Concurrent Collections:
  - Queues, blocking queues, concurrent hash map, …
  - Data structures designed for concurrent environments
- Locks and Conditions
  - More flexible synchronization control
  - Read/write locks
- Synchronizers: Semaphore, Latch, Barrier, Exchanger
  - Ready made tools for thread coordination
- Atomic variables
  - The key to writing lock-free algorithms

Atomic Variables

- **Holder classes for scalars, references and fields**
  - `java.util.concurrent.atomic`
- **Support atomic operations**
  - **Compare-and-set (CAS)**
    - `boolean compareAndSet(T expected, T update)`
    - **Atomically sets value to `update` if currently `expected`**
    - **Returns true on successful update**
  - **Get, set and arithmetic operations (where applicable)**
    - **Increment, decrement operations**
- **Nine main classes:**
  - **{ int, long, reference } X { value, field, array }**
  - **E.g. `AtomicInteger` useful for counters, sequence numbers, statistics gathering**

AtomicInteger Example

Construction Counter for Monitoring/Management

- Replace this:

```
class Service {
   static int services;
   public Service() {
     synchronized(Service.class) {
       services++;
     }
  } // ...
}
```

- With this:

```
class Service {
  static AtomicInteger services =
     new AtomicInteger();
  public Service() {
    services.getAndIncrement();
  }
  // ...
}
```

Case Study: Memoizer

- Implement a class for memorizing function results
- Memo Function:
  - —A function that memorizes its previous results
    - Optimization for recursive functions, etc.
  - —Invented by Prof. Donald Michie, Univ. of Edinburgh
- Goal: Implement `Memoizer`
  - —Function wrapper
  - —Provide concurrent access
  - —Compute each result at most once
- Tools:
  - – `ConcurrentHashMap`
  - – `FutureTask`

Memoizer: Generic Computation

- **Generic computation**

```
interface Computable<A, V> {
  V compute(A arg) throws Exception;
}
```

- **Representative example**

```
class ComplexFunction
    implements Computable<String, BigInteger> {

  public BigInteger compute(String arg) {
    // after deep thought...
    return new BigInteger("2");
  }
}
```

Memoizer: Usage

- Current use of `ComplexFunction` requires local caching of result (or expensive re-compute)
  - Computable<String, BigInteger> f =
  - new ComplexFunction();
  - BigInteger result = f.compute("1+1");
  - // cache result for future use
- `Memoizer` encapsulates its own caching
  - Computable<String, BigInteger> f =
  - new ComplexFunction();
  - f = new Memoizer<String, BigInteger>(f);
  - BigInteger result = f.compute("1+1");
  - // call f.compute whenever we need to

Synchronized Memoizer

- **Safe but not concurrent**

```
class SyncMemoizer<A,V> implements Computable<A,V> {

   final Map<A, V> cache = new HashMap<A, V>();
   final Computable<A, V> func;

   SyncMemoizer(Computable<A, V> func) {
     this.func = func;
   }

  public synchronized V compute(A arg)throws
Exception{
     if (!cache.containsKey(arg))
        cache.put(arg, func.compute(arg));
     return cache.get(arg);
   }
}
```

Non-atomic Concurrent Memoizer

- **Safe, concurrent (no sync) but computes may overlap**

```
class NonAtomicMemoizer<A,V> implements
Computable<A,V> {

  final Map<A, V> cache = new ConcurrentHashMap<A,
V>();
  final Computable<A, V> func;

  NonAtomicMemoizer(Computable<A, V> func) {
    this.func = func;
  }

  public V compute(A arg) throws Exception {
    if (!cache.containsKey(arg))
       cache.put(arg, func.compute(arg));
    return cache.get(arg);
  }
}
```

Concurrent Memoizer Using Future

- **Safe, concurrent and exactly one compute per argument**

```
class ConcurrentMemoizer<A, V>
                        implements Computable<A, V> {

  final ConcurrentMap<A, Future<V>> cache =
                new ConcurrentHashMap<A,
Future<V>>();

  final Computable<A, V> func;

  ConcurrentMemoizer(Computable<A, V> func) {
    this.func = func;
  }

  ...
```

Concurrent Memoizer Using Future (2)

```
public V compute(final A arg) throws Exception{
  Future<V> f = cache.get(arg);
  if (f == null) {
    Callable<V> eval = new Callable<V>() {
      public V call() throws Exception {
        return func.compute(arg);
      }
    };
    FutureTask<V> ft = new FutureTask<V>(eval);
    f = cache.putIfAbsent(arg, ft);
    if (f == null) {
      f = ft;
      ft.run();
    }
  }
  return f.get();
}
```

Case Study: Concurrent Linked List

- Goal: Implement a concurrent linked-list
  - —Demonstrate "chained-locking"
- Tools:
  - `ReentrantLock`
- Goal: Implement a "blocking bounded list"
  - —Demonstrate composition: data structure + synchronizer
- Tools:
  - `Semaphore`

Concurrent Linked List – Locking Strategy

- Design goal: fine-grained concurrent access
- Solution: lock-per-node
- Basic principle: all accesses traverse from the head in-order
  - —To access a node it must be locked
  - —To add a new node the node before must be locked
  - —To remove a node both the node and the node before must be locked
- Hand-over-hand Locking:
  - —Lock n1, lock n2, unlock n1, lock n3, unlock n2, lock n4, …
  - —Order in which threads acquire the first lock is maintained

    No overtaking once traversal starts
- Full version would implement `java.util.List`

Concurrent Linked List #1

```
public class ConcurrentLinkedList {
  /**
   * Holds one item in a singly-linked list.
   * It's convenient here to subclass ReentrantLock
   * rather than add one as a field.
   */
  private static class Node extends ReentrantLock {
      Object item;
      Node next;
      Node(Object item, Node next) {
          this.item = item;
          this.next = next;
      }
  }
  /**
   * Sentinel node. This node's next field points to
   * the first node in the list.
   */
  private final Node sentinel = new Node(null, null);
```

Concurrent Linked List #2

```
public void addFirst(Object x) {
  Node p = sentinel;
  p.lock();     // acquire first lock
  try {
    p.next = new Node(x, p.next); // Attach new node
  } finally {
    p.unlock();
  }
}
```

- **Locking considerations**
  - —What needs to be unlocked in the normal case?
  - —What needs to be unlocked if an exception occurs?
    - Will the list still be in a consistent state?
    - Note: can’t protect against asynchronous exceptions
  - —Simple in this case: only one lock held, only one failure mode
- **Note:** `Lock.lock()` **could throw exception e.g.** `OutOfMemoryError`

Concurrent Linked List #3

```
public void addLast(Object x) {
   Node p = sentinel;
   p.lock();    // Acquire first lock
   try {        // Find tail, using hand-over-hand locking
     while (p.next != null) {
       // p is always locked here
       Node prevp = p;
       p.next.lock();  // Acquire next lock
       p = p.next;
       prevp.unlock(); // Release previous lock
     }
     // only p is still locked here
     p.next = new Node(x, null); // Attach new node
   } finally {
     p.unlock();  // Release final lock
   }
 }
```

- Again exception handling is easy to do – but harder to reason about!
- Note: `NullPointerException` and `IllegalMonitorStateException` only possible if list code is broken

Concurrent Linked List #4

```
public boolean contains(Object x) {
  Node p = sentinel;
  p.lock();   // Acquire first lock
  try {       // Find item, using hand-over-hand
locking
    while (p.next != null) {
      // p is always locked here
      Node prevp = p;
      p.next.lock();  // Acquire next lock
      p = p.next;
      prevp.unlock(); // Release previous lock
      // found it?
      if (x == p.item || x != null && x.equals(p.item))
        return true;
    }
    // only p is still locked now
    return false;
  } finally {
    p.unlock();  // Release final lock
  }
}
```

Concurrent Linked List #5

```
public boolean remove(Object x) {
   Node p = sentinel;
   p.lock();     // Acquire first lock
   try {         // Find item, using hand-over-hand locking
     while (p.next != null) {
       Node prevp = p;
       p.next.lock();  // Acquire next lock
       p = p.next;
       // can't unlock prevp yet as removal of p
       // requires update of prevp.next
       try {
         if (x==p.item || x!=null && x.equals(p.item)) {
           prevp.next = p.next; // remove node p
           return true;
         }
       } finally {
          prevp.unlock(); // Release previous lock
       }
     }
     return false;
   } finally {
     p.unlock();  // Release final lock
   }
 }
```