JSR166 (Concurrency Utilities) Initial Aims and Scope

Doug Lea
State University of New York at Oswego
dl@cs.oswego.edu
http://gee.cs.oswego.edu

Goals

- Standardize medium-level constructs
  - Simplify application programming
  - Avoid incompatibilies, reinventions
  - Improve quality and efficiency of user code
  - Keep small: Support only very common APIs
  - Learn from 3+ years of experience with EDU.oswego.cs.dl.util.concurrent
- Add minimal “native” lower-level support
  - Only constructs “easy” to add to common JVMs
  - Avoid gratuitous incompatibilities with POSIX pthreads and RTSJ
  - Overcome existing small design problems

Disclaimers

- Don't take any of the sample APIs seriously!
  - They are intended to give flavor of the APIs, but all details are sure to change.
  - Even though targeted for JDK1.5, they ignore genericity and other likely 1.5 features.
- I don't speak for other expert group members
- Even though the main targets are APIs useful to application programmers, this talk will start with lower-level concerns.

Scope

- Atomic variables
- Nanosecond timing
- Lock classes
- Attributes for locks and threads
- Condition variables
- Additional functionality for builtin locks
- Queues
- Barriers
- Executors
- Concurrent Collections
- Thread class API

Atomic Variables

- Classes representing scalars that can be atomically updated using compareAndSwap.
  - Essential for writing efficient code on modern multiprocessors.
  - Main target audience is people implementing higher-level functionality.
- Must define APIs so that they can be implemented using locks if necessary.
  - Can also use Pugh's idiom-recognition to replace locks with CAS elsewhere.
  - But AtomicX classes clearly express intent and don't require sophisticated optimization.

Atomic Integer


```
class AtomicInteger {
    AtomicInteger(int initialValue);
    int get();
    void set(int newValue);
    int getAndSet(int newValue);
    int compareAndSwap(int oldv, int newv);
    int increment();
    int decrement();
    int add(int x);
  }

// AtomicRef, AtomicFloat, etc are similar
```

DCAS

- Two-variable CAS is also very useful!
  - See Sun Labs papers and reports.
- Normally must be emulated, but still more efficient and expressive of intent.
- Current API approach is to use Pair classes
  - Compromise across overhead, flexibility, need for JVM trickery

Atomic Integer Pair


```
class AtomicIntegerPair {

  AtomicIntegerPair(AtomicInteger first,
                    AtomicInteger second);

  AtomicIntegerPair(int initialFirst,
                    int initialSecond);

  AtomicInteger first();
  AtomicInteger second();

  boolean compareAndSwap(int expectFirst,
                         int updateFirst,
                         int expectSecond,
                         int updateSecond);

}
```

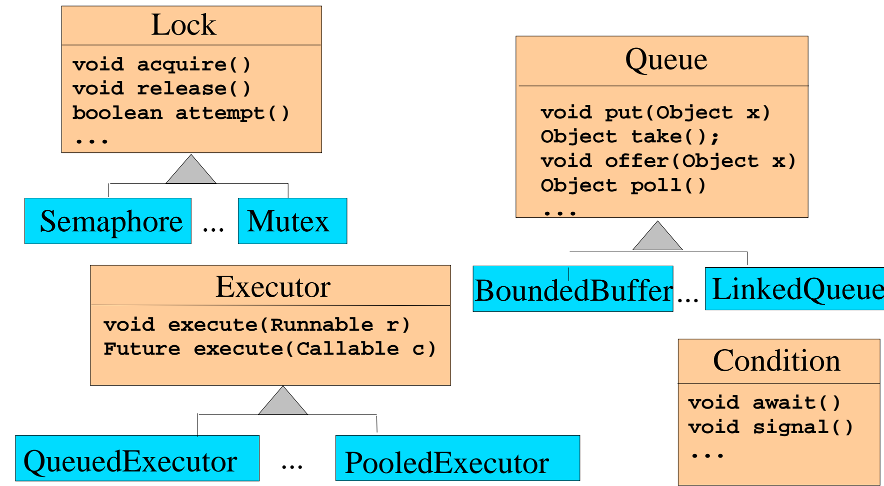
http://gee.cs.oswego.edu

Nanosecond timing

- Problems
  - Milliseconds are sometimes too coarse
  - Fix current API flaw: Object.wait etc have nanosec versions, but there is no way to determine time in nanosecs!
- Options
  - Use RTSJ HighResolutionTime classes
  - Define NanoTimer class with getTime method
- Upcoming examples assume second approach

http://gee.cs.oswego.edu

Main Interfaces



http://gee.cs.oswego.edu

Locks

```
try {
   lock.acquire();
   try {
     action();
   }
   finally {
     lock.release();
   }
 }
 catch (InterruptedException ie) { ... }
```

- More flexibility at expense of verbosity
- Can use locks with custom semantics
  - Reentrancy, Semaphore-style, Priority
- Overcome limitations of synchronized blocks
  - Assuring interruptibility, hand-over-hand locking

http://gee.cs.oswego.edu

Lock Interface

```
interface Lock {
  void acquire() throws IE;
  void acquireUninterruptibly();

  boolean attempt();
  boolean attemptWithinMillis(long ms)
          throws IE;
  boolean attemptWithinNanos(long ns)
          throws IE;

  void release();

  SynchronizationAttributes
     getSynchronizationAttributes();
  Condition createCondition();
 }
 // IE == InterruptedException
```

http://gee.cs.oswego.edu

Exclusion Example

```
class ParticleUsingMutex {
    private int x;  int y;
    private final Random rng = new Random();
    private final Mutex mutex = new Mutex();

    public void move() throws InterruptedException {
      mutex.acquire();
      try {
        x += rng.nextInt(2)-1;
        y += rng.nextInt(2)-1;
      }
      finally { mutex.release(); }
    }

    public void draw(Graphics g) {
      int lx, ly;
      mutex.acquireUninterruptibly();
      try {
        lx = x; ly = y;
      }
      finally { mutex.release(); }
      g.drawRect(lx, ly, 10, 10);
    }
  }
```

http://gee.cs.oswego.edu

ReadWrite Locks

```
interface ReadWriteLock {
   Lock readLock();
   Lock writeLock();
 }
```

- Manage a pair of locks
  - Each used in the same way as ordinary locks
- Implementation classes vary in lock policy
  - WriterPreference, ReentrantWriterPreference, ReaderPreference, FIFO

http://gee.cs.oswego.edu

Synchronization Attributes

- Provide meta-information about policies
  - Fairness, priority, reentrancy, precedences
  - Similar to POSIX approach
  - Can find out policy, but cannot in general change it.

```
class SynchronizationAttributes {
   boolean isFIFO();
   boolean isPrioritized();
   boolean isReentrant();
   // etc
 }
```

http://gee.cs.oswego.edu

Back-Port to Builtin Locks

```
class Locks {
  static boolean // trylocks
    performIfLockAvailable(Object lock,
                           Runnable r);
  static boolean // timeouts
    performIfLockAvailableWithinMillis(
      Object lock, long msecs, Runnable r);
  static boolean
    performIfLockAvailableWithinNanos(
      Object lock, long nsecs, Runnable r);

  static Condition
    createConditionFor(Object lock);

  static SynchronizationAttributes
    getSynchronizationAttributesFor(Object l);
  }
```

Conditions

http://gee.cs.oswego.edu

- New interface to represent monitor condition variables
- Removes limitation of one monitor per object
  - But cannot retrofit builtin single monitor
- Removes need for intricate or inefficient solutions to common concurrency problems
- Removes incompatibility with POSIX
  - Most JVMs layer on POSIX condvars anyway
- Fixes minor annoyances
  - Timeouts
- Allows custom Conditions for custom Locks.

Condition interface

http://gee.cs.oswego.edu

```
interface Condition {
  void await() throws IE;
  void awaitUninterruptibly();
  boolean awaitWithinMillis(long ms)
          throws IE;
  boolean awaitWithinNanos(long ns)
          throws IE;

  void signal();
  void signalAll();

  SynchronizationAttributes
    getSynchronizationAttributes();
}
```

Condition Example

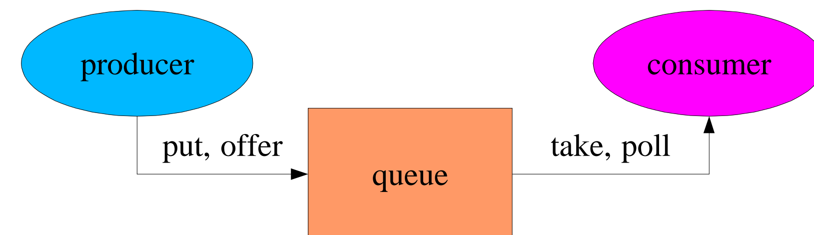
http://gee.cs.oswego.edu

```
class BoundedBuffer {
  final Condition notFull = Locks.createConditionFor(this);
  final Condition notEmpty = Locks.createConditionFor(this);
  Object[] items = new Object[100];
  int putptr, takeptr, count;

  public synchronized void put(Object x) throws IE {
    while (count == items.length)
      notFull.await();
    items[putptr] = x;
    if (++putptr == items.length) putptr = 0;
    ++count;
    notEmpty.signal();
  }

  public synchronized Object take() throws IE {
    while (count == 0)
      notEmpty.await();
    Object x = items[takeptr];
    if (++takeptr == items.length) takeptr = 0;
    --count;
    notFull.signal();
    return x;
  } }
```

Queues

http://gee.cs.oswego.edu

- Needed in most concurrent programs



- Would like a single interface to cover common implementations
  - Bounded and unbounded
  - LinkedQueue, BoundedBuffer, PriorityQueue, Handoff (CSP style), ...

http://gee.cs.oswego.edu

Queue interface

```
interface Queue {
  void put(Object x) throws IE;
  boolean offer(Object x);
  boolean offerWithinMillis(Object x, long ms)
          throws IE;
  boolean offerWithinNanos(Object x, long ns)
          throws IE;

  Object take() throws IE;
  Object poll();
  Object pollWithinMillis(long ms) throws IE;
  Object pollWithinNanos(long ns) throws IE;

  Object peek();
  boolean isEmpty();
  int size();
  long capacity(); // ??
}
```

http://gee.cs.oswego.edu

Queue Example

```
class LoggedService { // ...
  final Queue msgQ = new LinkedQueue();

  public void serve() throws InterruptedException {
    String status = doService();
    msgQ.put(status);
  }

  public LoggedService() { // start background thread
    Runnable logger = new Runnable() {
      public void run() {
        try {
          for(;;)
            System.out.println(msqQ.take());
        }
        catch(InterruptedException ie) {} }};
    new Thread(logger).start();
  }
}
```

http://gee.cs.oswego.edu

CyclicBarriers

- Basic multiparty synchronization tool
  - Each thread must wait for all others to hit barrier
  - Very common in parallel programming
  - Amenable to platform-specific optimization

```
public class CyclicBarrier {
  CyclicBarrier(int parties);
  CyclicBarrier(int parties,
                Runnable barrierAction);
  int sync() throws InterruptedException,
                    BrokenBarrierException;
  int parties();
  boolean isBroken();
  void reset();
}
```

http://gee.cs.oswego.edu

Barrier Example

```
class Solver { // Code sketch
  void solve(final Problem p, int nThreads) {

    final CyclicBarrier barrier = new CyclicBarrier(nThreads
      new Runnable() {
        public void run() { p.checkConvergence(); }}
    );

    for (int i = 0; i < nThreads; ++i) {
      final int id = i;
      Runnable worker = new Runnable() {
        final Segment segment = p.createSegment(id);
        public void run() {
          try {
            while (!p.converged()) {
              segment.update();
              barrier.sync();
            }
          }
          catch(Exception e) { return; }
      };
      new Thread(worker).start();
   }
}
```

Exchangers

http://gee.cs.oswego.edu

- Two-party barrier with data exchange
  - Common in pipeline designs
  - For example, buffer-exchange

```
public class Exchanger {
  Exchanger();
  Object exchange(Object x) throws
                     InterruptedException,
                     BrokenBarrierException;
  boolean isBroken();
  void reset();
}
```

Executor

http://gee.cs.oswego.edu

- Standardize asynchronous invocation
  - Avoid use of “new Thread”
- Two styles supported:
  - Actions: Runnables
  - Functions: Callables

```
interface Executor {
 void execute(Runnable action);
 Future execute(Callable function,
                Object argument);
}
```

- Implement with thread pools, thread factories, lightweight execution frameworks

Executor Example

http://gee.cs.oswego.edu

```
class WebService {
   public static void main(String[] args) {
     Executor pool = new ...
     try {
       ServerSocket socket = new ServerSocket(9999);
       for (;;) {
         final Socket connection = socket.accept();
         pool.execute(new Runnable() {
           public void run() {
             new Handler().process(connection);
           }});
       }
     }
     catch(Exception e) { } // die
   }
}

class Handler { void process(Socket s); }
```

Thread Pools

http://gee.cs.oswego.edu

- Need controls for:
  - The kind of task queue
  - Maximum number of threads
  - Minimum number of threads
  - "Warm" versus on-demand threads
  - Scheduling policies?
  - Shutdown policy
    - immediate, wait for current tasks
  - Keep-alive interval until idle threads die
    - to be later replaced by new ones if necessary
  - Saturation policy
    - block, drop, producer-runs, etc

Futures and Callables

- Callable is functional analog of Runnable

```
interface Callable {
   Object call(Object arg) throws Exception;
 }
```

- Future holds result of async call

```
interface Future {
   Object get() throws InterruptedException,
                       InvocationTargetException;
   Object getWithinMillis(long ms) throws...
   Object getWithinNanos(long ns) throws...
   boolean isAvailable();

   void set(Object result);
   void setException(Exception thrown);
}
```

http://gee.cs.oswego.edu

Future Example

```
class ImageRenderer { Image render(byte[] raw); }

 class App { // ...
    Executor executor = ...;         // any executor
    ImageRenderer renderer = new ImageRenderer();

    public void display(byte[] rawimage) {
      try {
        Future img = executor.execute(new Callable(){
          public Object call() {
            return renderer.render(rawImage);
          }});

        drawBorders(); // do other things while executing
        drawCaption();

        drawImage((Image)(img.get())); // use future
      }
      catch (Exception ex) {
        cleanup();
        return;
      }
    }
 }
```

http://gee.cs.oswego.edu

Concurrent Collections

- Even those java.util.Collections that are thread-safe are not designed for heavily multithreaded use.
- The new lower-level features, plus JMM (JSR-133) revisions allow creation of high performance concurrent Maps, Lists, etc
  - Order-of-magnitude speedups possible for applications with heavy contention
- Other performance-sensitive classes in JDK could also be reworked.

http://gee.cs.oswego.edu

Thread Class

- Revisit interruption mechanics?
  - Can anything be done to minimize inconsistent use and misuse?
  - Is it possible to more closely align with RTSJ and/or POSIX?
- Can/should anything be done to encourage use of Executor instead of Thread for asynchronous invocation?

http://gee.cs.oswego.edu

Next Steps

- Argue about APIs
- Con someone into prototyping JVM support
  - Atomic variables
  - Conditions
  - Locks.performIfLockAvailable
- Write reference implementation
- Write documentation
- Create examples and tutorials
- Make TCK