COMP 322: Fundamentals of Parallel Programming

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Lecture 31: Java executors and synchronizers

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

Acknowledgments for Today's Lecture

- Combined handout for Lectures 27-31 (to be updated)
- "Introduction to Concurrent Programming in Java", Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial slides
  - —Contributing authors: Doug Lea, Brian Goetz
- "Java Concurrency Utilities in Practice", Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial slides
  - —Contributing authors: Doug Lea, Tim Peierls, Brian Goetz
- "Java Concurrency in Practice", Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes and Doug Lea. Addison-Wesley, 2006.
- "Engineering Fine-Grained Parallelism Support for Java 7", Doug Lea, July 2010.

Announcements

- Homework 6 deadline extended to 5pm today
- comp322 queue for SUG@R is now available every day till end of semester
  - qsub -I -N JOBNAME -q interactive -V -l nodes=1:ppn=8 -W group_list=comp322

Key Functional Groups in j.u.c.

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

Summary: Relating j.u.c. libraries to HJ constructs

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

Thread Creation Patterns

- Earlier, we saw two thread creation patterns for the web server
  - Single-threaded
  - Thread-per-task
  - Both have problems
- Single-threaded: doesn't scale, poor throughput and response time
- Thread-per-task: problems with unbounded thread creation
  - Overhead of thread startup/teardown incurred per request
  - Creating too many threads leads to `OutOfMemoryError`
  - Threads compete with each other for resources
- Better approach: use a *thread pool*
  - Set of dedicated task-processing threads feeding off a common work queue
  - Enables effective resource management

java.util.concurrent.Executor interface

- Framework for asynchronous task execution
- A design pattern with a single-method interface
  - interface Executor { void execute(Runnable w); }
- Separate work from workers (what vs how)
  - ex.execute(work), not new Thread(..).start()
- Cancellation and shutdown support
- Usually created via `Executors` factory class
  - Configures flexible `ThreadPoolExecutor`
  - Customize shutdown methods, before/after hooks, saturation policies, queuing
- Normally use group of threads: ExecutorService

Think Tasks, not Threads

- Executor framework provides services for executing tasks in threads
  - `Runnable` is an abstraction for tasks
  - `Executor` is an interface for executing tasks
- Thread pools are specific kinds of executors

```
exec = Executors.newFixedThreadPool(nThreads);

    final Socket sock = server.accept();
    exec.execute(new Runnable() {
        public void run() {
            processRequest(sock);
    }});
```

  - This will create a fixed-sized thread pool
  - When those threads are busy, additional tasks submitted to exec.execute() are queued up

Executor Framework Features

- There are a number of factory methods in `Executors`
  - `newFixedThreadPool(n)`, `newCachedThreadPool()`, `newSingleThreadedExecutor()`
- Can also instantiate `ThreadPoolExecutor` directly
- Can customize the thread creation and teardown behavior
  - Core pool size, maximum pool size, timeouts, thread factory
- Can customize the work queue
  - Bounded vs unbounded
  - FIFO vs priority-ordered
- Can customize the *saturation policy* (queue full, maximum threads)
  - discard-oldest, discard-new, abort, caller-runs
- Execution hooks for subclasses
  - `beforeExecute()`, `afterExecute()`

ExecutorService interface

- `ExecutorService` extends Executor interface with lifecycle management methods e.g.,
  - `shutdown()`

    Graceful shutdown – stop accepting tasks, finish executing already queued tasks, then terminate
  - `shutdownNow()`

    Abrupt shutdown – stop accepting tasks, attempt to cancel running tasks, don't start any new tasks, return unstarted tasks
- An ExeuctorService is a group of thread objects, each running some variant of the following loop
  - *while (…) { get work and run it; }*
- ExecutorService's take responsibility for the threads they create
  - Service owner starts and shuts down `ExecutorService`
  - `ExecutorService` starts and shuts down threads

Multi-Threaded Web Server with Executor (1 of 3)

```
public class PooledWebServer {
    private final ServerSocket server;
    private ExecutorService exec;

    public PooledWebServer(int port) throws IOException {
        server = new ServerSocket(port);
        server.setSoTimeout(5000);
    }
```


Multi-Threaded Web Server with Executor (2 of 3)

```
public synchronized void startServer(int nThreads) {
      if (exec == null) {
         exec = Executors.newFixedThreadPool(nThreads + 1);
         exec.execute(new Runnable() { // nested async's!
            public void run() {
               while (!Thread.interrupted()) {
                  try {
                     final Socket sock = server.accept();
                     exec.execute(new Runnable() {
                        public void run() { processRequest(sock); }
                     });
                  }
                  catch (SocketTimeoutException e) { continue; }
                  catch (IOException ex) { /* log it */ }
               }
            }
         });
      }
   }
```


Multi-Threaded Web Server with Executor (3 of 3)

```
    public synchronized void stopServer()
                                throws InterruptedException {
        if (exec == null)
            throw new IllegalStateException(); // never started

        if (!exec.isTerminated()) {
            exec.shutdown();
            exec.awaitTermination(5L, TimeUnit.SECONDS);
            server.close();
        }
    }
}
```


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

- NOTE: the HJ work-sharing runtime system uses one ThreadPoolExecutor per place to execute async tasks

Key Functional Groups in j.u.c.

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

j.u.c Synchronizers --- common patterns in HJ's phaser construct

- Class library includes several state-dependent synchronizer classes
  - `CountDownLatch` – waits until latch reaches terminal state
  - `Semaphore` – waits until permit is available
  - `CyclicBarrier` – waits until N threads rendezvous
  - `Phaser` – extension of CyclicBarrier with dynamic parallelism
  - `Exchanger` – waits until 2 threads rendezvous
  - `FutureTask` – waits until a computation has completed
- These typically have three main groups of methods
  - Methods that block until the object has reached the right state
    - Timed versions will fail if the timeout expired
    - Many versions can be cancelled via *interruption*
  - Polling methods that allow non-blocking interactions
  - State change methods that may release a blocked method

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 Example

- 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

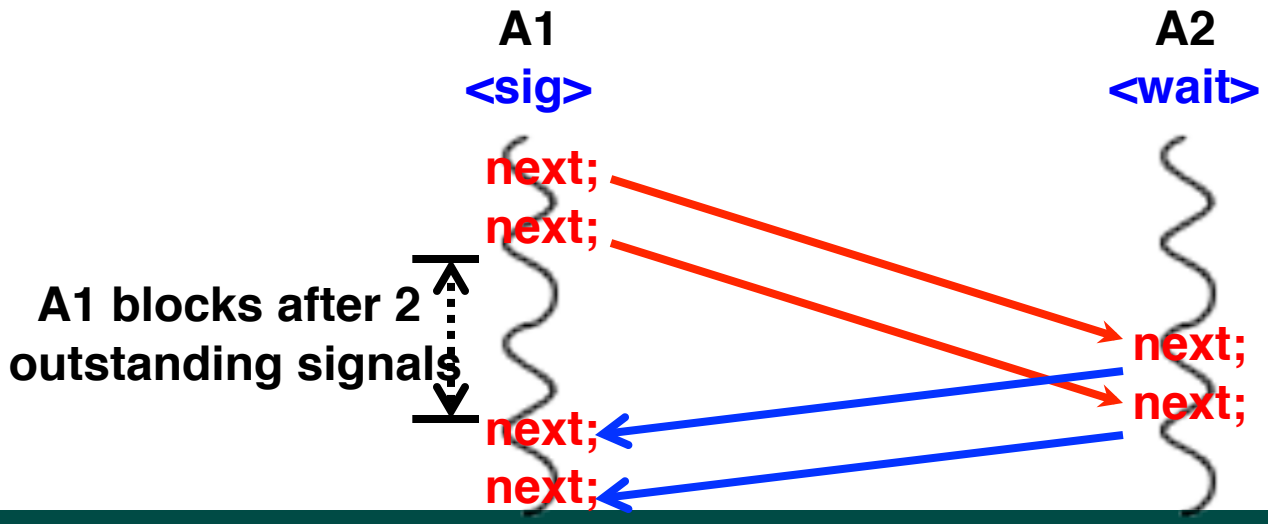
```
public class BoundedBlockingList {
   final int capacity;
   final ConcurrentLinkedList list =
                                  new ConcurrentLinkedList();
   final Semaphore sem;

  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;
   }
   …
}
```


Related work: Extension of HJ's Phasers to "Bounded Phasers"

- Bounded Phasers: Limit maximum phase difference between producers and consumers for a phaser
  - —Add bound_size as a parameter in phaser constructor
  - —A signaling task blocks when it reaches the maximum phase difference (can lead to deadlock)

```
phaser ph = new phaser(phaserMode.SIG_WAIT, 2 /*Bound size*/);
async phased (ph<phaserMode.SIG>) { next; next; ... /*A1*/ }
async phased (ph<phaserMode.WAIT>) { next; next; ... /*A2*/ }
```



Single-Producer Single-Consumer Bounded Buffer using Bounded Phasers

```
finish {
  final phaser ph = new phaser(phaserMode.SIG_WAIT,
                               bound_size);
  async phased (ph<phaserMode.SIG>)
    while (…) { insert(); next; } // producer
  async phased (ph<phaserMode.WAIT>)
    while (…) { next; remove(); } // consumer
}
```

- Can be extended to multiple producers and multiple consumers, assuming synchronous merge in each phase
- Extension to nondeterministic merge is more challenging

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
- `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

Example: using j.u.c.CountDownLatch to implement finish

- Problem: Run a task concurrently in N threads and wait until all are complete
  - Use a CountDownLatch initialized to the number of threads

```
public static void runTask(int numThreads, final Runnable task)
        throws InterruptedException {
  final CountDownLatch done = new CountDownLatch(numThreads);
  for (int i=0; i<numThreads; i++) {
      Thread t = new Thread() {
          public void run() {
              try {
                  task.run();
               } finally {
                  done.countDown(); // I'm done
              }
            }};
       t.start();
  }
  done.await();   // wait for all threads to finish
}
```


Summary: Relating j.u.c. libraries to HJ constructs

- Atomics: java.util.concurrent.atomic

  Can be used as is in HJ programs

- Concurrent Collections

  Can be used as is in HJ programs

- Locks: java.util.concurrent.locks

  Many uses of j.u.c.locks & synchronized can be replaced by HJ isolated

- Synchronizers

  Many uses can be replaced by phasers and data-driven futures

- Executors

  Many uses can be replaced by async, finish, futures, forall

- Queues

  Do not use BlockingQueue in HJ programs, and take care to avoid infinite loops on retrieval operations on non-blocking queues