COMP 322: Fundamentals of Parallel Programming

Lecture 31: Java executors and synchronizers

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Acknowledgments for Today’s Lecture

• “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


• “A brief intro to: Parallelism, Threads, and Concurrency”, Tom Horton, CS 2110 lecture, U. Virginia
Key Functional Groups in java.util.concurrent

• **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
Thread Creation Patterns

- Thus far, we have studied two thread creation patterns for a web server example
  - Single-threaded (all requests are executed on a single thread)
  - Thread-per-task (a new thread is created for each new request)
  - 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 threads feeding off a common work queue
  - This is what the HJ runtime does (with different queue data structures used by different scheduling algorithms)
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

```java
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 **ExecutorService** 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
1. public class PooledWebServer {
2.   private final ServerSocket server;
3.   private ExecutorService exec;

5.   public PooledWebServer(int port) throws IOException {
6.     server = new ServerSocket(port);
7.     server.setSoTimeout(5000);
8.   }
9. }
Multi-Threaded Web Server with Executor Service (2 of 3)

10. public synchronized void startServer(int nThreads) {
11.     if (exec == null) {
12.         exec = Executors.newFixedThreadPool(nThreads + 1);
13.         exec.execute(new Runnable() { // outer “async” listens to socket
14.             public void run() {
15.                 while (!Thread.interrupted()) {
16.                     try {
17.                         final Socket sock = server.accept();
18.                         exec.execute(new Runnable(){// inner “async” processes request
19.                             public void run() { processRequest(sock); } }
20.                     });
21.                 }
22.             } catch (SocketTimeoutException e) { continue; }
23.         } catch (IOException ex) { /* log it */ }
24.     }
25. }
26. });
27. }
28. }
29. public synchronized void stopServer()
30. throws InterruptedException {
31.     if (exec == null)
32.         throw new IllegalStateException(); // never started
33.     if (!exec.isTerminated()) {
34.         exec.shutdown();
35.         exec.awaitTermination(5L, TimeUnit.SECONDS);
36.         server.close();
37.     }
38. } // stopServer()
39. } // class PooledWebServer
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
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`
New Java ForkJoin Framework

• Designed to support a common need
  – Recursive divide and conquer pattern
  – For small problems (below cutoff threshold), execute sequentially
  – For larger problems
    • Define a task for each subproblem
    • Library provides
      – a Thread manager, called a ForkJoinPool
      – Methods to send your subtask objects to the pool to be run, and your call waits until they are done
      – The pool handles the multithreading well

• The “thread manager”
  – Used when calls are made to RecursiveTask’s methods fork(), invokeAll(), etc.
  – Supports limited form of “work-stealing”
Using ForkJoinPool

- ForkJoinPool implements the ExecutorService interface
- Create a ForkJoinPool “thread-manager” object
- Create a task object that extends RecursiveTask
  - Create a task-object for entire problem and call invoke(task) on your ForkJoinPool
- Your task class' compute() is like Thread.run()
  - It has the code to do the divide and conquer
  - First, it must check if small problem - don't use parallelism, solve without it
  - Then, divide and create >1 new task-objects. Run them:
    - Either with invokeAll(task1, task2, ...). Waits for all to complete.
    - Or calling fork() on first, then compute() on second, then join()
Using ForkJoin framework vs. Thread class

To use the ForkJoin Framework:

Don't subclass Thread  Do subclass RecursiveTask<V>
Don't override run  Do override compute
Don't call start  Do call invoke, invokeAll, fork
Don't just call join  Do call join which returns answer
Do call invokeAll on multiple tasks
Mergesort Example

- Top-level call. Create “main” task and submit

1. public static void mergeSortFJRecur(Comparable[] list,
2. int first, int last) {
3.     if (last - first < RECURSE_THRESHOLD) {
4.         MergeSort.insertionSort(list, first, last);
5.         return;
6.     }
7.     Comparable[] tmpList = new Comparable[list.length];
8.     threadPool.invoke(
9.         new SortTask(list, tmpList, first, last));
10. }
Mergesort’s Task-Object Nested Class

11. static class SortTask extends RecursiveAction {
12.    Comparable[] list;
13.    Comparable[] tmpList;
14.    int first, last;
15.    public SortTask(Comparable[] a, Comparable[] tmp,
16.                int lo, int hi) {
17.        this.list = a;    this.tmpList = tmp;
18.        this.first = lo; this.last = hi;
19.    }
compute() method contains “async” body

20. protected void compute() {
21.     if (last - first < RECURSE_THRESHOLD)
22.         MergeSort.insertionSort(list, first, last);
23.     else {
24.         int mid = (first + last) / 2;
25.         SortTask task1 =
26.             new SortTask(list, tmpList, first, mid);
27.         SortTask task2 =
28.             new SortTask(list, tmpList, mid + 1, last);
29.         invokeAll(task1, task2); // Two async’s + finish
30.         MergeSort.merge(list, first, mid, last);
31.     }
32. } // compute()
Key Functional Groups in java.util.concurrent

• 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
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

- Semaphore, CountDownLatch, CyclicBarrier, Phaser in Java are special cases of HJ phasers
- Phaser is an extension of CyclicBarrier with dynamic parallelism
- Exchanger is an implementation of Ada's rendezvous construct
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
  — Easier than modifying implementation of concurrent list directly
Bounded Blocking Concurrent List

1. public class BoundedBlockingList {
2.   final int capacity;
3.   final ConcurrentLinkedList list = new ConcurrentLinkedList();
4.   final Semaphore sem;
5.   public BoundedBlockingList(int capacity) {
6.     this.capacity = capacity;
7.     sem = new Semaphore(capacity);
8.   }
9.   public void addFirst(Object x) throws InterruptedException {
10.      sem.acquire();
11.      try { list.addFirst(x); }
12.      catch (Throwable t){ sem.release(); rethrow(t); }
13.   }
14.   public boolean remove(Object x) {
15.      if (list.remove(x)) { sem.release(); return true; }
16.      return false;
17.   }
18.   ... } // BoundedBlockingList
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 N tasks concurrently in N threads and wait until all are complete

  - Use a `CountDownLatch` initialized to the number of threads

1. public static void runTask(int numThreads, final Runnable task)
2.     throws InterruptedException {
3.         final CountDownLatch done = new CountDownLatch(numThreads);
4.         for (int i=0; i<numThreads; i++) {
5.             Thread t = new Thread() {
6.                 public void run() {
7.                     try {
8.                         task.run();
9.                     } finally {
10.                        done.countDown(); // I'm done
11.                     }
12.                 }
13.             t.start();
14.         }
15.         done.await(); // wait for all threads to finish
16.     }
<table>
<thead>
<tr>
<th>Summary: Relating j.u.c. libraries to HJ constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Atomics:</strong> java.util.concurrent.atomic</td>
</tr>
<tr>
<td>Can be used as is in HJ programs</td>
</tr>
<tr>
<td><strong>Executors</strong></td>
</tr>
<tr>
<td>Many uses can be replaced by async, finish, futures, forall</td>
</tr>
<tr>
<td><strong>Concurrent Collections</strong></td>
</tr>
<tr>
<td>Many uses of j.u.c.locks &amp; synchronized can be replaced by HJ isolated</td>
</tr>
<tr>
<td><strong>Locks:</strong> java.util.concurrent.locks</td>
</tr>
<tr>
<td><strong>Queues</strong></td>
</tr>
<tr>
<td>Do not use BlockingQueue in HJ programs, and take care to avoid infinite loops on retrieval operations on non-blocking queues</td>
</tr>
<tr>
<td><strong>Synchronizers</strong></td>
</tr>
<tr>
<td>Many uses can be replaced by finish, phasers, and data-driven futures</td>
</tr>
</tbody>
</table>