# COMP 322: Fundamentals of Parallel Programming

## Lecture 28: Java Executors and Synchronizers

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

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



## Worksheet #26 solution: use of tryLock()

**Extend the transferFunds() method from Lecture 25 to use library locks with tryLock() instead of synchronized, and to return a boolean value --- true if it succeeds in obtaining in obtaining both locks and performing the transfer, and false otherwise. Sketch your answer below using pseudocode. Can you create a deadlock with multiple calls to transferFunds() in parallel?**

```
public boolean transferFunds(Account from, Account to,
                             int amount) {
  // Assume that each Account object has a lock field of
  // a type/class that implements java.util.concurrent.locks.Lock
  // Assume that no exception can be thrown in this code
  // Calls to this method can never lead to a deadlock
  if (! from.lock.trylock()) return false;
  if (! to.lock.trylock()) return false;
  from.subtractFromBalance(amount); to.addToBalance(amount);
  from.lock.unlock(); to.lock.unlock();
  return true;
}
```

# 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
- "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.
- "A brief intro to: Parallelism, Threads, and Concurrency", Tom Horton, CS 2110 lecture, U. Virginia
  - http://www.cs.virginia.edu/~cs201/slides/cs2110-16-parallelprog.ppt



# Outline

- <u>Java Executors</u>
- Java Synchronizers

# 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 Java threads
  - —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 (as you've already been doing in HJ ...)

- 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 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

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

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

# Multi-Threaded Web Server with Executor Service (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();
     }
   } // stopServer()
 } // 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**
  - **We will learn about "places" later in the course**

# 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`



# 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

```
public static void mergeSortFJRecur(Comparable[] list,
          int first, int last) {
    if (last - first < RECURSE_THRESHOLD) {
        MergeSort.insertionSort(list, first, last);
        return;
    }
    Comparable[] tmpList = new Comparable[list.length];
    threadPool.invoke(
        new SortTask(list, tmpList, first, last));
}
```



# Mergesort's Task-Object Nested Class

```
static class SortTask extends RecursiveAction {
   Comparable[] list;
   Comparable[] tmpList;
   int first, last;
   public SortTask(Comparable[] a, Comparable[] tmp,
                   int lo, int hi) {
         this.list = a;      this.tmpList = tmp;
         this.first = lo;   this.last = hi;
   }
```

# compute() method contains "async" body

```
protected void compute() {
 if (last - first < RECURSE_THRESHOLD)
   MergeSort.insertionSort(list, first, last);
 else {
   int mid = (first + last) / 2;
   SortTask task1 =
            new SortTask(list, tmpList, first, mid);
   SortTask task2 =
            new SortTask(list, tmpList, mid+1, last);
   invokeAll(task1, task2); // Two async's + finish
   MergeSort.merge(list, first, mid, last);
 }
} // compute()
```



# Outline

- Java Executors
- Java Synchronizers

# 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



# 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

# 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

```
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
 }
```

# 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

```
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;
 }
 … } // BoundedBlockingList
```



# Callable Objects can be used to create Future Tasks in Java

- **Any class that implements java.lang.Callable<V> must provide a call() method with return type V**
- **Sequential example with Callable interface**

```
ImageData image1 = imageInfo.downloadImage(1);
ImageData image2 = imageInfo.downloadImage(2);
. . .
renderImage(image1);
renderImage(image2);
```

Listing 5: HTML renderer in Java before decomposition into Callable tasks

```
Callable<ImageData> c1 = new Callable<ImageData>() {
  public ImageData call() {return imageInfo.downloadImage(1);}};
Callable<ImageData> c2 = new Callable<ImageData>() {
  public ImageData call() {return imageInfo.downloadImage(2);}};
. . .
renderImage(c1.call());
renderImage(c2.call());
```

Listing 6: HTML renderer in Java after decomposition into Callable tasks

# 4 steps to create future tasks using Callable objects

1. Create a parameter-less callable closure using a statement like "Callable<Object> c = new Callable<Object>() {public Object call() { return ...; }}; "
2. Encapsulate the closure as a task using a statement like "FutureTask<Object> ft = new FutureTask<Object>(c);"
3. Start executing the task in a new thread by issuing the statement, "new Thread(ft).start();"
4. Wait for the task to complete, and get its result by issuing the statement, "Object o = ft.get();".



# Parallelization of HTML renderer example (Module 2 handout, Chapter 13)

```
Callable<ImageData> c1 = new Callable<ImageData>() {
  public ImageData call() {return imageInfo.downloadImage(1);}};
FutureTask<Object> ft1 = new FutureTask<Object>(c1);
new Thread(ft1).start();
Callable<ImageData> c2 = new Callable<ImageData>() {
  public ImageData call() {return imageInfo.downloadImage(2);}};
FutureTask<Object> ft2 = new FutureTask<Object>(c2);
new Thread(ft2).start();
. . .
renderImage(ft1.get());
renderImage(ft2.get());
```

HTML renderer in Java after parallelization of Callable tasks

```
future<ImageData> ft1 = async<ImageData>{return imageInfo.downloadImage(1);};
future<ImageData> ft2 = async<ImageData>{return imageInfo.downloadImage(2);};
. . .
renderImage(ft1.get());
renderImage(ft2.get());
```

Equivalent HJ code

# Worksheet #28: Relating j.u.c. libraries to HJ constructs

Name 1: ____________________ Name 2: ____________________

For each functional group of j.u.c. libraries included below, indicate one of the following choices: a) can be used in HJ programs, b) can be substituted by equivalent HJ constructs in some cases (give examples), c) cannot be substituted by equivalent HJ constructs in some cases (give examples).

1. Atomic variables

2. Concurrent Collections

3. Locks

4. Executors

5. Synchronizers



# Summary of j.u.c. libraries

- 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
- Executors
  - ExecutorService
  - ScheduledExecutorService
  - Callable
  - Future
  - ScheduledFuture
  - Delayed
  - CompletionService
  - ThreadPoolExecutor
  - ScheduledThreadPoolExecutor
  - AbstractExecutorService
  - FutureTask
  - ExecutorCompletionService
- Synchronizers
  - CountDownLatch
  - Semaphore
  - Exchanger
  - CyclicBarrier