CMSC 433 Fall 2014
Michael Hicks
(Some slides due to Rance Cleaveland)

UNIVERSITY OF MARYLAND

Lecture 13
Applying Thread Pools

JCIP Chap. 6.2.3 – 6.3.8
and Chap. 8.0-8.4

Task Submission

- ExecutorService objects manage thread pools
- They also include methods for task submission
  - `void execute(Runnable command)`
    Executes the given command at some time in the future.
  - `<T> Future<T> submit(Callable<T> task)`
    Submits a value-returning task for execution and returns a Future representing the pending results of the task.
  - `Future<?> submit(Runnable task)`
    Submits a Runnable task for execution and returns a Future representing that task.
- ???
  - Purpose of `submit()` is to permit determination of status of task, collect return value
  - Tasks have four phases:
    - Created
    - Submitted
    - Started
    - Completed
  - `Future` includes `get()`, which can be used to collect return value / check completion
  - Other methods in `Future` include `boolean isDone()`

Callable vs. Runnable

- Runnable
  - Can be fed to `Thread` constructor
  - Cannot return value
  - Cannot throw checked exceptions
- Callable
  - Cannot be fed to Thread constructor
  - Can return value
  - Can throw checked exceptions (these are wrapped inside an `ExecutionException`)

Defining Task Boundaries

- Recall: tasks are "logical chunks of independent computation"
- Identifying good task boundaries allows for more concurrency
- Some applications (e.g. the web-server examples) have a natural notion of task (e.g. request)
- In other cases you may need to work some!

Defining Task Boundaries: An Example

- Example comes from JCIP pp. 124ff: page renderer
  - Page renderer is responsible for converting HTML code into something viewable in a web browser
  - Tasks include formatting text, downloading images
- What are good tasks for rendering?

Page Renderer(1): Sequential

```
public class SingleThreadRenderer {
  void renderPage(CharSequence source) {
    renderText(source);
    List<ImageData> imageData = new ArrayList<ImageData>();
    for (ImageInfo imageInfo : scanForImageInfo(source))
      imageData.add(imageInfo.downloadImage());
    for (ImageData data : imageData)
      renderImage(data);
  }
}
```

- Design decision: one task!
  - Text is rendered
  - Then images are downloaded, one-by-one
- Generally, this would yield poor responsiveness
  - Downloading images requires accessing network
  - Rendering text can be done locally
  - So image-processing would dominate!

Page Renderer(2): Two Tasks

```
public class FutureRenderer {
    private final ExecutorService executor = Executors.newCachedThreadPool();
    void renderPage(CharSequence source) {
        final List<ImageInfo> imageInfos = scanForImageInfo(source);
        Callable<List<ImageData>> task =
                new Callable<List<ImageData>>() {
                    public List<ImageData> call() {
                        List<ImageData> result = new ArrayList<ImageData>();
                        for (ImageInfo imageInfo : imageInfos)
                            result.add(imageInfo.downloadImage());
                        return result;
                    }
                };
        Future<List<ImageData>> future = executor.submit(task);
        renderText(source);
        try {
            List<ImageData> imageData = future.get();
            for (ImageData data : imageData)
                renderImage(data);
        } catch (InterruptedException e) {
            …
        } catch (ExecutionException e) { … }
    }
}
```

Page Renderer(2): Observations

- There is some parallelism

  Text rendering, image downloading done in parallel
- Will this yield a big speed-up?

  Not for pages with lots of images!
  - Downloading of images is still done sequentially
  - The image downloading task could take much longer than text rendering

Page Renderer(3): More Tasks

- Each image can be downloaded independently!
- We can exploit this to refine task boundaries
  - One task for text
  - One task for each image
  - When each image download finishes, image can be rendered
- How can we wait for all the downloads?
  - One approach: loop
    - Iterate for the number of images
    - Perform a `get()` on each `Future`
    - But what if one image takes a lot longer to download
  - Better approach: `CompletionService`

`CompletionService`

- Extends `ExecutorService` with a *blocking completion queue*
  - When a task that has been submitted finishes, a `Future` for it is put in completion queue
  - A user of the completion service can extract next finished computation by performing `take()` on completion service
- This permits processing of task results in order that they were completed

Page Renderer(3)

```
public class Renderer {
    private final ExecutorService executor;
    Renderer(ExecutorService executor) {        this.executor = executor;    }
    void renderPage(CharSequence source) {
        final List<ImageInfo> info = scanForImageInfo(source);
        CompletionService<ImageData> completionService =
                new ExecutorCompletionService<ImageData>(executor);
        for (final ImageInfo imageInfo : info)
            completionService.submit(new Callable<ImageData>() {
                public ImageData call() {
                    return imageInfo.downloadImage();
                }
            });
        renderText(source);
        try {
            for (int t = 0, n = info.size(); t < n; t++) {
                Future<ImageData> f = completionService.take();
                ImageData imageData = f.get();
                renderImage(imageData);
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } catch (ExecutionException e) {
            throw launderThrowable(e.getCause());
        }
    }
}
```

Parallelization with Futures

- We followed a kind of recipe for converting a sequential method call z = m.foo(x,y) into one that happens in parallel. There are three basic steps:
  1. Create a separate class that implements Callable<T> where T is the type of the value returned from the method call
  2. Replace the call site invocation with one the creates an instance of this new class and passes it to an ExecutorService for execution. This will return a Future<T> for a Callable<T>
  3. Replace any place that z was used in the old program, expecting it to be of type T, with a call to the Future.get() method

Step 1

- Starting with

```
T z = m.foo(x,y);
...
z.bar();
```

- We define

```
public class FooCall implements Callable<T> {
  private T1 x;
  private T2 y;
  private T3 o;
  public FooCall(T1 x, T2 y, T3 o) {
    this.x = x;  this.y = y;  this.o = o;
  }
  T call() { return o.foo(x,y); }
}
```

Step 2

- Replace

```
T z = m.foo(x,y);
```

- with

```
Future<T> z = executor.submit(new
FooCall(x,y,o));
```

- where

```
ExecutorService executor =
Executors.newFixedThreadPool(4);
```

- Appears earlier in the program

Step 3

- Replace

```
z.bar();
```

- with

```
z.get().bar()
```

- (and do similarly for all occurrences of z)

Designing Thread Pools

- Considerations
  - How big?
  - What execution policy?
- Decisions about these considerations are influenced by several factors
  - Task dependencies
    - Some tasks are independent
    - Some require results of other tasks
    - Some tasks will even spawn other tasks whose results they need
  - Task thread-confinement assumptions
    - Some tasks assume thread-confinement
      - Legacy single-threaded code
      - Efficiency
    - Such tasks should run in a single-threaded thread pool
  - Variability in task execution times, responsiveness requirements
    - Some tasks may run much longer than others
    - Other tasks may need quick turnarounds
  - Tasks that assume thread-specific knowledge
    - Some tasks may make assumptions about the specific thread on which they are running (e.g. if there is a `ThreadLocal` variable)
    - Such tasks must be handled carefully in thread-pool setting

Thread Starvation Deadlock

- An issue affecting pool sizing
- Suppose you have a fixed-size pool (say, 10)
  - Suppose 10 tasks are running, so no free threads
  - Suppose further that each of these tasks submits a task to the pool and then blocks awaiting the result
- Deadlock!
  - Each of 10 task-threads is blocking
  - There are no threads to handle new tasks on which they are blocking
  - No thread can make progress

Thread-Starvation Deadlock Example (JCIP p. 169)

```
public class ThreadDeadlock {
    ExecutorService exec = Executors.newSingleThreadExecutor();
    public class LoadFileTask implements Callable<String> { … }
    public class RenderPageTask implements Callable<String> {
        public String call() throws Exception {
            Future<String> header, footer;
            header = exec.submit(new LoadFileTask("header.html"));
            footer = exec.submit(new LoadFileTask("footer.html"));
            String page = renderBody();
            // Will deadlock -- task waiting for result of subtask
            return header.get() + page + footer.get();
        }
        private String renderBody() {
            // Here's where we would actually render the page
            return "";
        }
    }
}
```

- Thread pool in this case has one thread
- RenderPageTask spawns off two other tasks: one for page header, one for footer
- Deadlock!

Dealing with Thread-Starvation Deadlock

- Thread-starvation deadlock happens when
  - Pool-size is bounded
  - There are task dependencies: tasks can block waiting for results of other tasks
- If an application has these features, either:
  - Make pool size unbounded
  - Make pool large enough to handle anticipated dependencies (risky!)
  - **DOCUMENT REASONS FOR DECISION!**

Sizing Thread Pools

- Want to avoid thread pools that are "too big" or "too small"
  - Too big: contention among threads for memory, other resources
  - Too small: bad throughput
- We have already seen one consideration for sizing thread pools: thread-deadlock starvation
- Other considerations
  - Are tasks compute or I/O intensive?
  - How many processors on system?
  - How much memory do tasks need?
  - What other possibly scarce resources (e.g. JDBC connections) are needed?
- Note
  - Sometimes you have different classes of tasks that must be run, with different profiles
  - You can use multiple thread pools and tune each independently!

Determining Thread-Pool Sizes

- Some variables
  - N_{CPU}: number of CPUs
  - U_{CPU}: desired utilization ($0 \leq U_{CPU} \leq 1$)
  - W/C: ratio of wait time to compute time
  - $N_{threads}$: number of threads
- For compute-intensive applications (i.e. W/C is low), good rule is $N_{threads} = N_{CPU} + 1$
  - Every task blocks for some reason or another, usually (page fault, etc.)
  - Having one more thread than CPU ensures efficiency
- In general, if cycles are important resource, and threads are homogeneous, independent, then $N_{threads} = N_{CPU} * U_{CPU} * (1 + W/C)$
- Example
  - Suppose
    - $N_{CPU} = 8$ (8-core machine)
    - $U_{CPU} = 0.5$ (machine is free ½ of time to deal with other applications)
    - W/C = 2 (so threads wait on average 2/3 of time they are running)
  - Then $N_{threads} = 8*0.5*(1+2) = 12$.
- Resources besides cycles can be dealt with similarly

Other Size Considerations

- If some tasks are long-running, and others are not, and you want to use one thread pool, then:
  - Ensure number of threads is larger than number of long-running tasks
  - Otherwise, all threads eventually run long-running tasks
  - Bad for throughput, responsiveness of shorter tasks
- In this case, if you know which tasks are long-running, separate thread pools for longer, shorter tasks makes sense

Thread-Pool Execution Policies

- Executors include thread-pool execution policy
- Executors returned by `Executors.newXXXThreadPool()`, etc. include built-in execution policies
- These methods all use a base implementation given in class `ThreadPoolExecutor`
  - To customize execution policy, you can call the ThreadPoolExecutor constructor yourself
  - The parameters to the constructor allow you to modify the execution policy in a variety of ways

Using ThreadPoolExecutor

- General constructor for this class has following form

```
ThreadPoolExecutor (
      int corePoolSize,
      int maximumPoolSize,
      long keepAliveTime,
      TimeUnit unit,
      BlockingQueue<Runnable> workQueue,
      ThreadFactory threadFactory,
      RejectedExecutionHandler handler )
```

- Some of parameters are easy to describe
  - **corePoolsize**
    Target number of threads to keep in pool, even when there are no tasks
  - **maximumPoolSize**
    Maximum number of threads that can be active at one time
  - **keepAliveTime**
    Thread that is idle for this amount of time can be "reaped" (i.e. killed) if number of threads is bigger than corePoolSize
  - **unit**
    Time unit for interpreting keepAliveTime (TimeUnit is an enum data type)

ThreadPoolExecutor: workQueue

- Work queue stores tasks that are awaiting a thread from the thread pool
- Default for `Executors.newFixedThreadPool()`, `Executors.singleThreadExecutor()`: **`LinkedBlockingQueue`**
  - Unbounded, so no task ever "turned away"
  - Blocks when empty, so threads idle by blocking when there are no tasks
  - Queues are FIFO, meaning tasks executed in order in which they arrive
- Default for `Executors.newCachedThreadPool()`: **`SynchronousQueue`**
  - The executors returned by this method use an unbounded number of threads
  - SynchronousQueue has capacity 0!
    - When a new task arrives, synchronous queue hands it off immediately to a thread in the thread pool
    - The executor creates a new worker thread if necessary in this case
- For more control over execution order, can use **`PriorityQueue`** for work queue
  - Tasks executed in priority, rather than arrival, order
- To bound number of waiting tasks, can uses a bounded queue (e.g. **`ArrayBlockingQueue`**)
  - In this case, must decide what to do if queue is full!
  - This decision becomes the *saturation policy* (what to do when work queue is saturated)
  - *Note: if there are inter-task dependencies, and either thread pool or work queue is bounded, then thread-starvation deadlock is possible*

`ThreadPoolExecutor`: **`handler`**

- If work queue is bounded, the saturation policy determines what to do when queue is full and a new task arrives
- This is the purpose of the **`handler`** parameter to the `ThreadPoolExecutor` constructor
  - `handler` has type `RejectedExecutionHandler`
  - It is also called when executor has been shutdown and a new task arrives
  - It can also be set after executor is constructed by calling `setRejectedExecutionHandler()`

Saturation Policies (cont.)

- `ThreadPoolExecutor` implements several saturation policies as (static) classes matching `RejectedExecutionHandler` interface
  - **`AbortPolicy`** (this is the default)

    `execute()` throws RejectedExecutionException if queue is full
  - **`DiscardPolicy`**

    `execute()` silently discards newest task
  - **`DiscardOldestPolicy`**
    - `execute()` discards task at head of work queue (i.e. next one up for execution) and tries to resubmit current task
    - Beware if work queue is a priority queue!
  - **`CallerRunsPolicy`**
    - `execute()` runs the task in the thread calling execute()
    - This helps give worker threads time to catch up, since new invocations of execute will be blocked from that thread!

ThreadPoolExecutor: threadFactory

- Executors need to create new threads from time to time
- The `threadFactory` parameter to `ThreadPoolExecutor` constructor determines how this is done
  - There is a default
  - Customizing `threadFactory` allows you to do common start-up / tear-down actions, assign common names, etc.
- `threadFactory` must implement interface:

```
public interface ThreadFactory {
  Thread newThread(Runnable r);
}
```

- How executor uses thread factory
  - When a new worker thread is needed, executor calls `threadFactory` with a private `Runnable`
  - This `Runnable` is typically an infinite loop that takes tasks (also `Runnables`!) from the work queue and invokes their `run()` methods
  - Note that worker threads are not pass tasks methods directly when they are created!

Customizing Thread Factory (1) – JCIP p. 177

```
public class MyThreadFactory implements
ThreadFactory {
    private final String poolName;
    public MyThreadFactory(String poolName) {
        this.poolName = poolName;
    }
    public Thread newThread(Runnable runnable) {
        return new MyAppThread(runnable, poolName);
    }
}
```

Customizing Thread Factory (2) – JCIP p. 178

```
public class MyAppThread extends Thread {
    public static final String DEFAULT_NAME = "MyAppThread";
    private static final AtomicInteger created = new AtomicInteger();
    private static final AtomicInteger alive = new AtomicInteger();
    public MyAppThread(Runnable r) {        this(r, DEFAULT_NAME);    }
    public MyAppThread(Runnable runnable, String name) {
        super(runnable, name + "-" + created.incrementAndGet());
        setUncaughtExceptionHandler( … );
    }
    public void run() {
        …
        try {
            alive.incrementAndGet();
            super.run();
        } finally {  alive.decrementAndGet(); }
    }
    public static int getThreadsCreated() {   return created.get();  }
    public static int getThreadsAlive() {   return alive.get();  }
    public static boolean getDebug() {   return debugLifecycle;  }
    public static void setDebug(boolean b) {   debugLifecycle = b;  }
}
```

Customizing `ThreadPoolExecutor` at Run-Time

- Parameters passed in during construction of `ThreadPoolExecutor` can also be inspected, modified using getters, setters
- This can be dangerous!
- `Executors` class includes factory method, `unconfigurableExecutorService()`, that removes access to getters, setters