Task Execution

Alma Orucevic-Alagic
2013-11-14

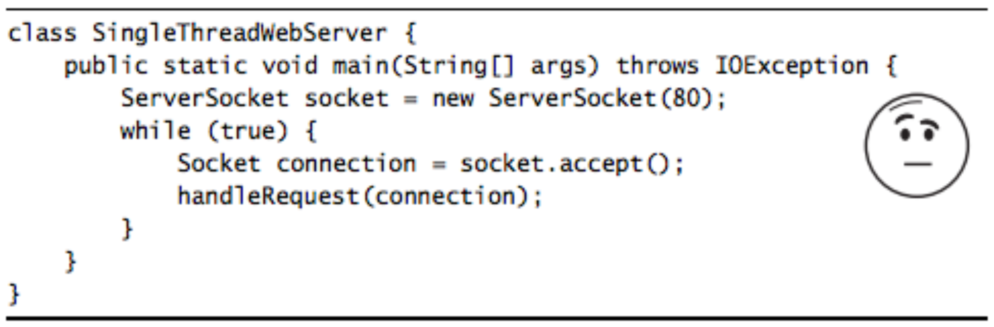
A summary of the chapter 5 of Göetz, Brian, et al. Java Concurrency in Practice. Addison-Wesley, 2006.

Motivation and Concepts

- Motivation:
  - Divide application into discrete units of work (tasks) in order to:
    - Simplify program execution
    - Facilitate error recovery (transaction boundaries)
    - Stimulate setup which promotes work parallelization
- Concepts:
  - Task Boundary
  - Independent Activity
    - NO state, result, or side effects DEPENDANCIES on other tasks.
  - Processing capacity

Single vs. Multithreaded

Single Thread

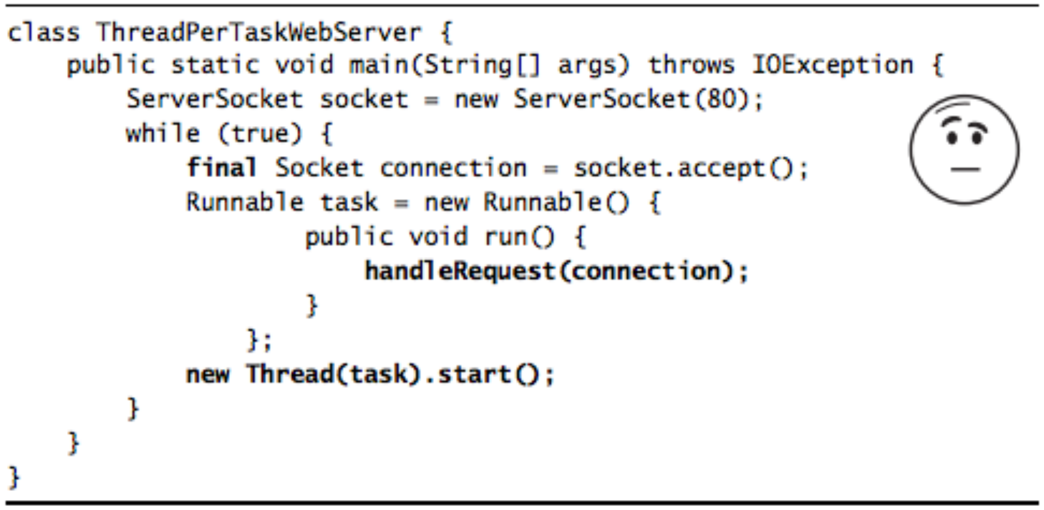


```
class SingleThreadWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            handleRequest(connection);
        }
    }
}
```

LISTING 6.1. Sequential web server.

- Poor Performance
- Low Throughput
- Bad Responsiveness
- GUI EDT exception

Multi Thread



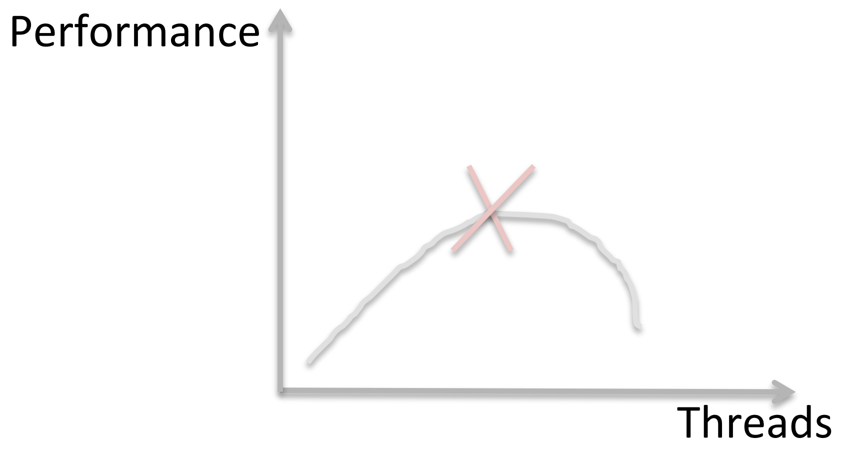
```
class ThreadPerTaskWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                    public void run() {
                        handleRequest(connection);
                    }
                };
            new Thread(task).start();
        }
    }
}
```

LISTING 6.2. Web server that starts a new thread for each request.

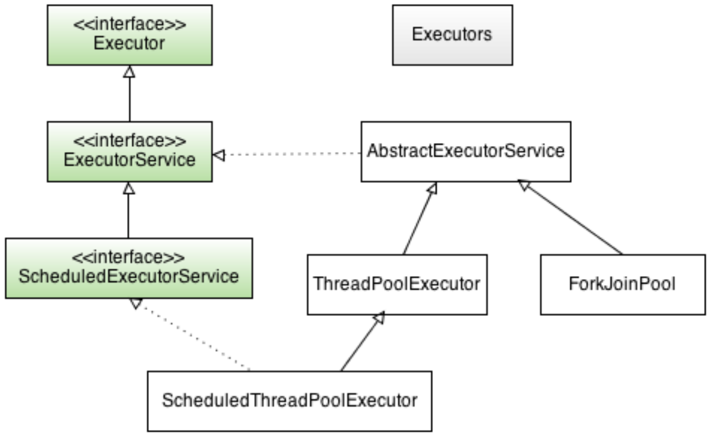
- Greater Responsiveness
- Higher Throughput
- Thread-safe code requirement

The Dark Side of Multithreading

- Thread Lifecycle Overhead
- Resource Consumption
- Stability
- Point of Diminishing Returns



The Executor Framework



<<interface>> Executor
Executors
<<interface>> ExecutorService
AbstractExecutorService
<<interface>> ScheduledExecutorService
ThreadPoolExecutor
ForkJoinPool
ScheduledThreadPoolExecutor

- Motivation:
  - Abstract thread management from application (life cycle, usage, scheduling…)
- Classes:
  - Executors: Utility Class/Factory
  - Executor: Interface with a method “*void execute(Runnable command);*”
  - ExecutorService: Methods to handle life cycle
  - ScheduledExecutorService: Methods related to scheduling
  - Thread Pools: Worker threads request a new task from queue,execute, wait/get next from queue
    - Executors class: Factory method to create thread pools:
      - Executors.newFixedThreadPool(int nThreads)
      - Executors.newCachedThreadPool(ThreadFactory threadFactory)
      - Executors.newSingleThreadExecutor()
      - Executors.newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)
  - Create customized thread pools (ThreadPoolExecutor, ScheduledThreadPoolExecutor, ForkJoinPool)

Task Execution

The Executor Framework …continued

```
class TaskExecutionWebServer {
    private static final int NTHREADS = 100;
    private static final Executor exec
        = Executors.newFixedThreadPool(NTHREADS);

    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            exec.execute(task);
        }
    }
}
```

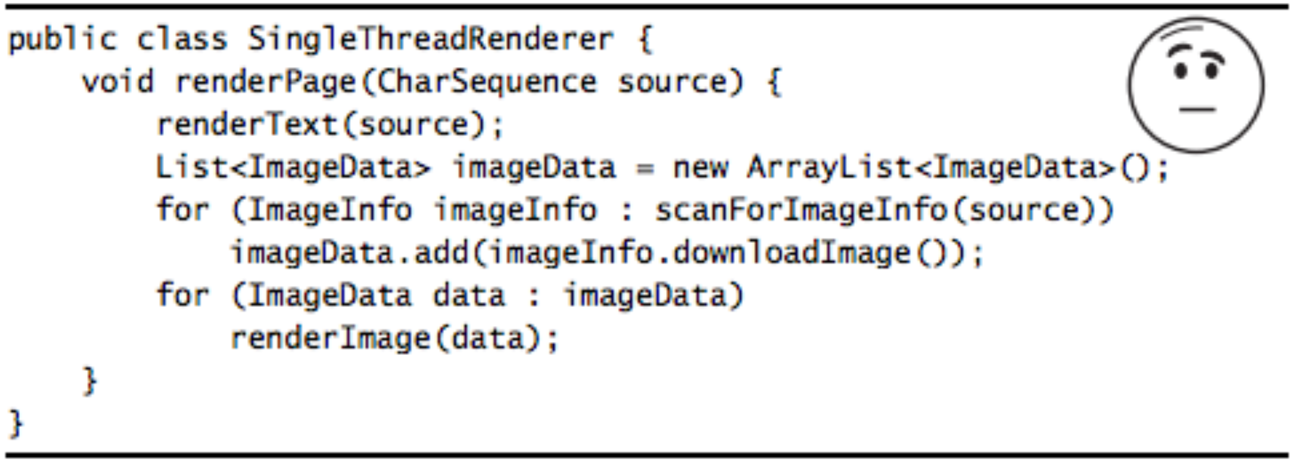
- Task Execution Policy: ”What , where, when, and how”
- Thread Pools: Reuse threads, avoid create/tear down costs, eliminate latency..
- ScheduledThreadPoolExecutor: Manages Execution of Deferred and Periodic Tasks.
- Executor Lifecycle: ExecutorService (also supplies submit method for Callable and Future objects)
- ExecutorService States: Running, Shutting Down, Terminated

Web Server with Shutdown Support

```
class LifecycleWebServer {
    private final ExecutorService exec = ...;

    public void start() throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (!exec.isShutdown()) {
            try {
                final Socket conn = socket.accept();
                exec.execute(new Runnable() {
                    public void run() { handleRequest(conn); }
                });
            } catch (RejectedExecutionException e) {
                if (!exec.isShutdown())
                    log("task submission rejected", e);
            }
        }
    }

    public void stop() { exec.shutdown(); }

    void handleRequest(Socket connection) {
        Request req = readRequest(connection);
        if (isShutdownRequest(req))
            stop();
        else
            dispatchRequest(req);
    }
}
```

Exploitable Parallelism

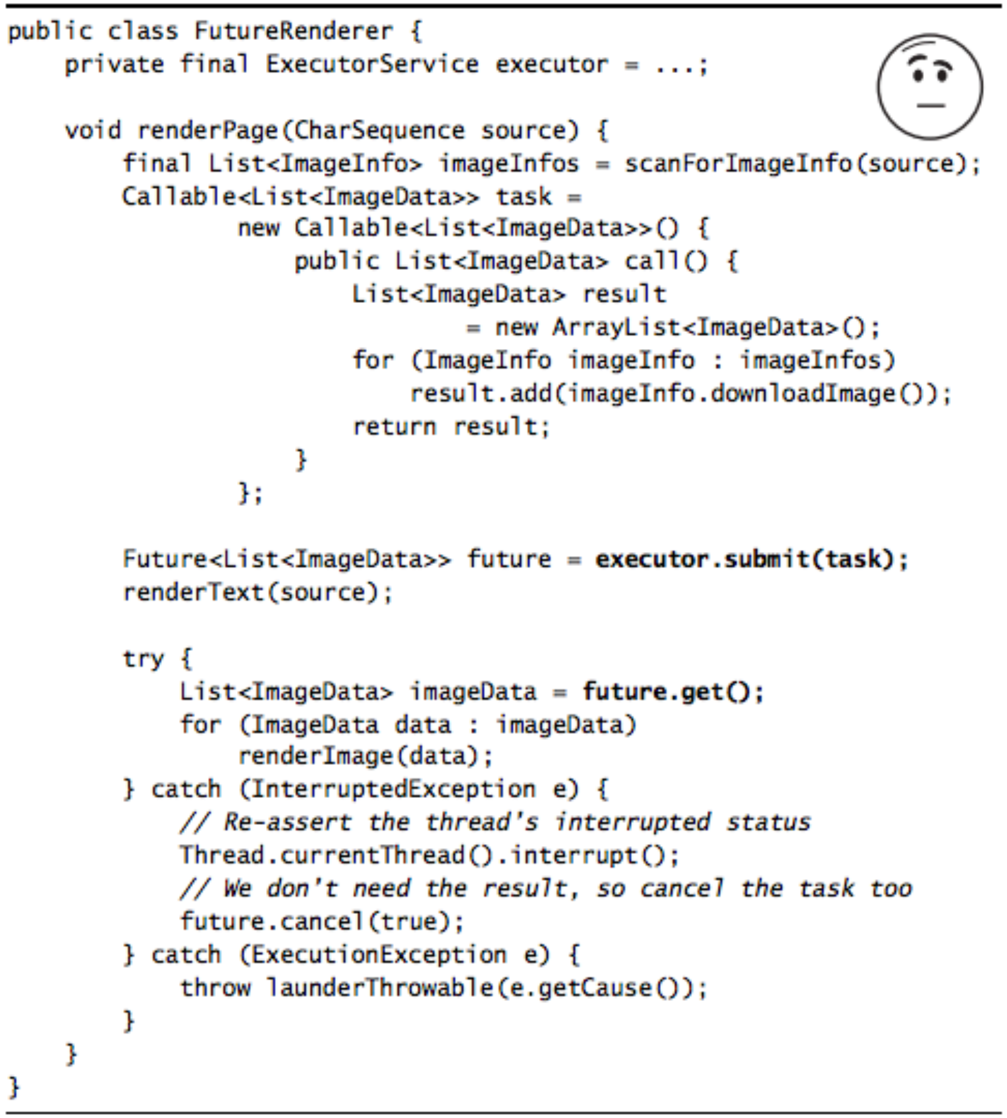
- Task defined as *Runnable*
- Task boundary not always obvious!
- Define independent, homogeneous, parallel exploit capable tasks.



```
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);
    }
}
```

- Result-bearing tasks: Callable and Future

ExecutionService: Future, Callable



```
public class FutureRenderer {
    private final ExecutorService executor = ...;

    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) {
            // Re-assert the thread's interrupted status
            Thread.currentThread().interrupt();
            // We don't need the result, so cancel the task too
            future.cancel(true);
        } catch (ExecutionException e) {
            throw launderThrowable(e.getCause());
        }
    }
}
```

CompletionService

- Intent: Decouple production of new asynchronous tasks from the consumption of results by completed tasks.
- Motivation: Retain results of Callable tasks as they come available –
  - Option 1: V Future.get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException)

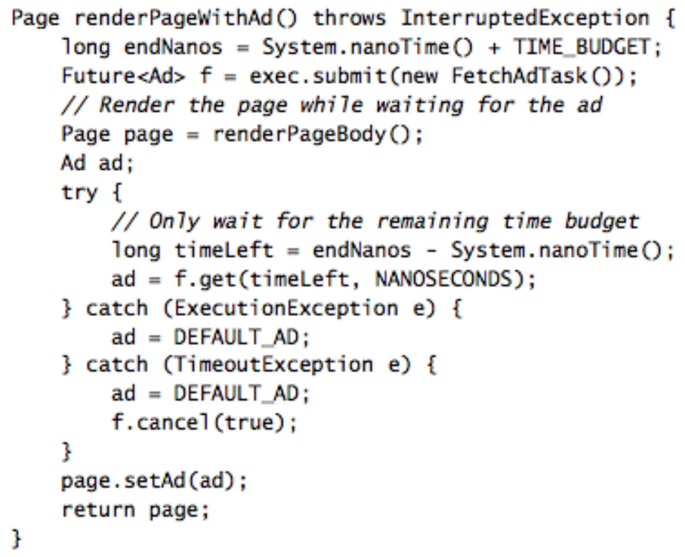
    Waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if available.
  - Option 2: Use ComplitionService to decouple. Producers submit tasks for execution, consumers take completed tasks and process their results in order they complete.
- Example: Manage asynchrous IO (decouple reads from actions on reads)

CompletionService Example

```
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());
        }
    }
}
```

Timed Tasks

- How long should/can be waited on a task to complete?
- What is time budget?
    - Use V Future.get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException)
    - Returns when results are done or throws TimeoutException – act upon the Exception.



```
Page renderPageWithAd() throws InterruptedException {
    long endNanos = System.nanoTime() + TIME_BUDGET;
    Future<Ad> f = exec.submit(new FetchAdTask());
    // Render the page while waiting for the ad
    Page page = renderPageBody();
    Ad ad;
    try {
        // Only wait for the remaining time budget
        long timeLeft = endNanos - System.nanoTime();
        ad = f.get(timeLeft, NANOSECONDS);
    } catch (ExecutionException e) {
        ad = DEFAULT_AD;
    } catch (TimeoutException e) {
        ad = DEFAULT_AD;
        f.cancel(true);
    }
    page.setAd(ad);
    return page;
}
```

Task boundary: One bid

- Executes the given tasks, returning a list of Futures holding their status and results when all complete.
    - <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
- Executes the given tasks, returning a list of Futures holding their status and results when all complete or the timeout expires, whichever happens first.
    - <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)

Exercises

1)

Describe an execution setup where CompletionService would be preferred over ExecutorService. Motivate your answer.

2)

Use Executors to implement execution of tasks that list files in the first level directories under root based on the first letter of directory name (a-z, A – Z).
Results of the file listings' process by other set of tasks responsible for sorting the results of each listing (e.g. all files under directory whose name starts with letter A) and print out names of the first and the last file in the sorted list.
The file listing tasks should be processed/sorted upon completion by its corresponding Future without having to use Future.done() or timed method on Future.get to check if task has completed.