Lecture 12
Task Execution

JCIP Chap. 6.1-6.2
and parts of 6.3
Tasks and Concurrent Applications

• So far: mechanics of concurrency in Java
  – Threads
  – Locking
  – Concurrency control
  – Visibility
  – Etc.

• Tasks have more to do with *concurrent-application design*
  – Tasks are units of work to be done
  – Task-oriented applications work by executing tasks as they are available
  – Example: web server
    • Tasks correspond to requests to the server
    • Server handles requests as they arrive
Design Considerations for Tasks

• Independence
  – Tasks should not interact with one another, if possible
  – This facilitates concurrency

• Size
  – “Smaller is better”
  – Gives more flexibility to scheduling, etc.

• The above considerations are sometimes referred to as determining task boundaries

• Task boundaries + execution policy for tasks helps determine application
  – Throughput: how many tasks are being completed per unit time?
  – Responsiveness: how long for individual tasks to complete?
  – Graceful degradation: how does system behave as it becomes overloaded?
Implementing Tasks

• Tasks: design-level artifacts
• Threads: run-time (implementation) artifacts
• Question: how you map tasks to threads?
  – One idea: sequentially
    • One thread used to execute all the tasks, one right after the other
    • Simple, but …
      – A big task can delay completion of other tasks
      – A crashing task can bring down the whole system!
  – Another idea: one thread / task
    • Solves problems with sequential implementation
    • But it introduces others
      – Task-handling code must be thread-safe
      – There is overhead associated with task creation
      – There are limits on how many tasks can be created
public class SingleThreadWebServer {
  public static void main(String[] args) throws IOException {
    ServerSocket socket = new ServerSocket(80);
    while (true) {
      Socket connection = socket.accept();
      handleRequest(connection);
    }
  }
}

• An instance of sequential task processing
  – Each task (connection) is handled by main thread
  – handleRequest() implemented elsewhere
• If handleRequest() crashes, so does webserver!
ThreadPerTaskWebServer (JCIP p. 115)

```java
public 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();
        }
    }
}
```

- Each task given a separate thread
- Under light-to-moderate load, this improves throughput, responsiveness
- Heavy load: too many threads!
Executors

• A middle ground between sequential task processing and thread-per-task processing
  – Executors contain a *thread pool of worker threads*
  – When a task comes in, and a thread is available, executor gives task to an idle thread
  – If no thread is available, executor queues the result for future execution

• Based on producer / consumer pattern
  – Producers: generators of tasks
  – Consumers: threads that execute tasks

• Decouples task submission from task execution

• Interface
  
  ```java
  public interface Executor {
    void execute(Runnable command);
  }
  ```
TaskExecutionWebServer (JCIP p. 118)

```java
public class TaskExecutionWebServer {
    private static final int NTHREADS = 100; // Fixed number of threads
    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);
        }
    }

    private static void handleRequest(Socket connection) {
        // request-handling logic here
    }
}
```
Execution Policies

- Executor implementation enables different execution policies to be defined
- An execution policy specifies how tasks get executed
  - Which thread?
  - What order (FIFO, LIFO, priority order)?
  - How many concurrent tasks?
  - How many tasks may be queued pending execution?
  - Overload policy? (Which task to kill, and how)
  - Pre- / post-task actions to perform, if any?
- Execution policies are a resource-management tool
  Permit management of concurrency vis a vis number of processors, other resources
Thread Pools

- Contains collection of homogeneous **worker threads**
- Is tightly bound to a work queue holding tasks to be executed
- Worker threads:
  - Request next task from work queue
  - Execute it
  - Return to waiting for next task
- Advantages (vs. creating new thread)
  - No need to wait for creation of new task
  - No overhead associated with task creation, elimination
- **Executors** class contains factory methods for creating thread pools, e.g.
  - `static ExecutorService newFixedThreadPool(int nThreads)`
    Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.
  - `static ExecutorService newCachedThreadPool()`
    Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.
  - `static ExecutorService newSingleThreadExecutor()`
    Creates an Executor that uses a single worker thread operating off an unbounded queue.
  - `static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`
    Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.
ExecutorService?

• The factory methods in Executors return objects in ExecutorService

• ExecutorService
  – Is an interface extending Executor
  – The new methods include mechanisms for shutting down an executor
    • JVM cannot terminate until all non-daemon threads shut down
    • Worker threads are non-daemon threads
    • Shutting down an executor requires shutting down these threads and dealing with any queued tasks
Executor Life Cycle

- An executor can be in one of three states
  - **Running**: executor is executing tasks, accepting new tasks
  - **Shutdown**: executor has stopped accepting new tasks, may or may not be finishing already accepted tasks
  - **Terminated**: executor has terminated all worker threads and is done

- ExecutorService interface includes methods corresponding to these states
  - `void shutdown()`
    - Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
  - `List<Runnable> shutdownNow()`
    - Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.
  - `boolean isShutdown()`
    - Returns true if this executor has been shut down.
  - `boolean isTerminated()`
    - Returns true if all tasks have completed following shut down.
  - `boolean awaitTermination(long timeout, TimeUnit unit)`
    - Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.
What About Tasks Submitted After Shutdown?

They are handled by the *rejected execution handler*

- Could just swallow the tasks
- Could throw `RejectedExecutionException`
- Depends on implementation!

• More details in later part of book
public class LifecycleWebServer {
    private final ExecutorService exec = Executors.newCachedThreadPool();

    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);
    }

    ...
Parallelization Recipe

• Given a computational task, how can you implement a parallel algorithm for it?

• Four steps

  1. Pick a concurrency strategy
  2. Code up units of work
  3. Identify and correct the concurrency hazards of executing the units of work in parallel
  4. Choose and apply a task execution policy
1. Pick a concurrency strategy

- How to break down the entire task into smaller units of work?
  - Depends on the kind of parallelism. Main kinds: *Data parallel, Task parallel, Hybrid, and Unstructured*

- Will need to anticipate some of the coding you'll do next
Kinds of parallelism

• **Data parallelism**
  – The same task run on different data in parallel

• **Task parallelism**
  – Different tasks running on the same data

• **Hybrid data/task parallelism**
  – A parallel pipeline of tasks, each of which might be data parallel

• **Unstructured**
  – Ad hoc combination of threads with no obvious top-level structure
Data parallelism

• Example: convert all characters in an array to upper-case
  – Can divide parts of the data between different tasks and perform the tasks in parallel
  – Key: no dependencies between the tasks that cause their results to be ordered
Task parallelism

• Example
  – Several functions on the same data: average, minimum, binary or, geometric mean
  – No dependencies between the tasks, so all can run in parallel
Pipeline parallelism

• Output of one task is the input to the next
  – Each task can run in parallel
  – Throughput impacted by the longest-latency element in the pipeline
Pipeline load balancing

• Assign more than one computational process to each task
  – Combines data- and pipeline- parallelism
2. Code up units of work

• Write down the parameters of that work
  – e.g., what varies for the chunk done by thread 1 vs. thread 2 vs. thread 3 etc.

• Write down functions that do the work, given the parameters

• Test those functions in a single-threaded scenario
  – e.g., make a loop that calls your functions N times to compute the total result
3. Correct Concurrency Hazards

• Run your units of work in parallel, ignoring synchronization etc.
  – Or do it mentally
• Look for concurrent accesses to shared data structures
  – What could go wrong?
• Fix hazards you find
  – e.g., use synchronization, make copies of immutable structures, etc.
  – Also, enforce an ordering to avoid parallel access if needed
4. Choose a task execution policy

• Choose how you will execute your tasks
  – e.g., thread pool, work-stealing scheduler, etc.

• Choose the number of threads you will want
  – May need several queues and executors (each having several threads) if there are dependencies between the tasks

• Do performance testing
  – to see the effects of different concurrency strategies, sync mechanisms, etc.
Example: WordCount

• See the WordCount.java example as the starting point, and the iterations through this recipe to produce the end versions