Parallelism Candidate Recap: Servers (I)

Many approaches... You could have each service handled by a server machine running in its own thread:

```java
class SingleThreadWebServer { //JCIP Listing 6.1
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            handleRequest(connection);
        }
    }
}
```

It works, but it would only handle one request at a time. For a busy web server, this would not be desirable or practical.

Even with only a single processor, a web request might use the network connection, the hard drive(s), and the processor (any of which could block, but which also provides opportunities for parallelism.
Parallelism Candidate Recap: Servers (II)

Many approaches... You could have each request handled by a server machine running in its own thread:

```java
class ThreadPerTaskWebServer {//JCIP Listing 6.2
    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();
        }
    }
}
```

It works, would have good responsiveness, could give better throughput if there are multiple processors or blocking on IO, but it would not scale well or tend to exhibit graceful degradation. Thread creation and termination is costly in terms of time and resources. This could easily consume all available stack space. For a busy web server, this would not be desirable or practical.

The Executor Framework

Think of a task as a unit of work. Think of a thread as a way for a task or set of tasks to run asynchronously from others.

Single-threaded approaches does not support either responsiveness or throughput.

Thread-per-task does not manage resources well.

We’ve discussed the notion of a pool as well as a bounded pool. What if we apply these ideas to threads?

Java provides the Executor framework (based on the producer/consumer pattern) as a way to manage a pool of worker threads.
Parallelism Candidate: Servers (III)

Many approaches... You could have a pool of threads with a queue of tasks lined up to utilize those threads:

class TaskExecutionWebServer { //JCIP Listing 6.4
  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);
      //NOTE: execute and submit are options if you declare exec as an ExecutorService rather than Executor
    }
  }
}

We now have a certain level of responsiveness (the socket connection is accepted) but the number of threads is easily limited to avoid having the system overwhelmed. If the thread pool is all busy, the execute requests will add the tasks to a queue but will not block itself.

Sculpting the Executor framework...

It provides a continuum between single-thread and thread-per-task that is flexible (you can also do either extreme using this).

The previous example used a fixed-sized thread pool created by the method newFixedThreadPool. The pool creates threads as tasks are submitted, but only up to the stated maximum pool size – after that it basically maintains a constant pool size (it will add new threads if some threads die). You can pick that maximum size and control the resources used.

You can achieve thread-per-task by using the newCachedThreadPool method which provides a pool that will send off each task in its own thread but also tries to re-use idle threads. However, it will add new threads as needed, without an upper bound other than what the system can sustain without crashing...

You can achieve single-thread behavior via the newSingleThreadExecutor method which provides a “pool” that actually only spawns a single worker thread (though it will be nice in that it will spawn a new one if the existing one dies).
**Time delayed or periodic execution**

While we’re looking at the topic, there’s a 4th option which is the use of `newScheduledThreadPool` method of the `Executors` class and provides another fixed-size thread pool. This one, however, supports both delayed and periodic task execution.

If you’ve used Visual Basic you might have encountered this with the Timer there (Java has a Timer too though `Timer` in Java has one possibly interesting ability – it can be set up to base things on system clock time rather than relative time so if the system clock is altered, Timer-based things can acknowledge those changes).

**Some advantages of the Executor framework...**

In addition to control over the number of threads, the `Executor` framework provides a way to both specify and modify policies on how to manage the tasks and threads. JCIP Section 6.2.2 lists:
- In what thread will tasks be executed?
- In what order should tasks be executed (FIFO, LIFO, priority order)?
- How many tasks may execute concurrently?
- How many tasks may be queued pending execution?
- If a task has to be rejected because the system is overloaded, which task should be selected as the victim, and how should the application be notified?
- What actions should be taken before or after executing a task?

*(more on these later…)*
The lifecycle of ExecutorService objects...

When you use one of the newXYZ methods, you actually get back an object that implements the ExecutorService interface (which extends Executor).

Once created, the service is either running, in the process of shutting down, or already shut down (terminated).

If it is in the process of shutting down (for example, if shutdown() was called – you might find the blocking awaitTermination method useful after calling shutdown()), it will no longer accept new tasks but might still be in the process of completing tasks that were already submitted. However, it could also be in the process of trying to stop the running tasks and refusing to start the ones that are waiting.

If shutdownNow() is called it will return a list of the tasks that were still waiting to execute and attempts to kill any tasks that were running (typically by throwing an ExecutionException for each, though that could be ignored by the task).

Parallelism Candidate: Servers (IV)

class LifecycleWebServer { //JCIP Listing 6.8
    private final ExecutorService exec = ...; //pick your approach
    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();} //let other code stop server
    void handleRequest(Socket connection) {
        Request req = readRequest(connection);
        if (isShutdownRequest(req))
            stop(); //support external shutdown request
        else dispatchRequest(req);
    }
}
execute versus submit

The execute method is established by the Executor interface. The ExecutorService interface adds three more required submit methods to allow you to check on the status of a task:

<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<null> submit(Runnable task)
Submits a Runnable task for execution and returns a Future representing that task has completed but no actual value.

<T> Future<T> submit(Runnable task, T result)
Submits a Runnable task for execution and returns a Future representing that task that will upon completion return the given result.

The CompletionService interface

Something that implements the CompletionService interface provides a take() method that allows you to take Future objects that represent the “next completed” task if there are any ready and which blocks until there is one if not.

The CompletionService can basically be seen as a wrapper for a BlockingQueue and an Executor.
```java
ExecutorService myThreadPool = Executors.newFixedThreadPool(5);
ExecutorCompletionService<Integer> myCompletionService = new ExecutorCompletionService<Integer>(myThreadPool);

myCompletionService.submit(new Callable<Integer>() {
    public Integer call() {
        try {Thread.sleep(10000);} catch (InterruptedException e) {} return 1;
    }
});

myCompletionService.submit(new Callable<Integer>() {
    public Integer call() {
        try {Thread.sleep(5000);} catch (InterruptedException e) {} return 2;
    }
});

myCompletionService.submit(new Callable<Integer>() {
    public Integer call() {
        try {Thread.sleep(1000);} catch (InterruptedException e) {} return 3;
    }
});

Future<Integer> v1 = myCompletionService.take();
System.out.println(v1.get());

Future<Integer> v2 = myCompletionService.take();
System.out.println(v2.get());

Future<Integer> v3 = myCompletionService.take();
System.out.println(v3.get());
```

**Some things to think about...**

If you are using classes with `ThreadLocal` objects, remember that they are local to a `thread`, not to a task. Threads that get reused will also be using the same `ThreadLocal` object throughout.

Make sure your tasks are not dependant on each other or you could get a deadlock if task A depends on task B and task B gets into the pool before task A and the pool fills up with other such dependant tasks.

– This might be avoided if you can work out how large the pool needs to be to make sure that there’s enough room for the worst-case dependency scenario.

If you have a way to estimate the length of tasks, the pool should be larger than the average number of long tasks so that other tasks get a chance to pass through.
Connecting pool size to resources

If you are designing a program that is CPU-intensive, you might want to tie your pool size to the number of CPUs. Java allows you to find out how many processor units are available via a call to `Runtime.getRuntime().availableProcessors()` so that value plus a small constant (1 or 2 for wiggle room) might be a good choice.

If you expect your program will be IO-intense (storage, network, etc.) with a ratio of IOwork/CPUwork and you wanted to keep the CPUs at full capacity, you’d need to have the number of threads set as the number of processors * (IO/CPU) – this would probably be too many though, since you don’t really want to spike your CPUs…

Defining your tasks...

Using these things well depends in great part in defining the “task” granularity. How do you define the boundaries for the task?

A web server is an easy example – there is a clear task boundary. What about a web client? A single page is a poor boundary even in a tabbed browser. Defining each load “unit” as a task might be better (the html as the first task and then each item that the html requires the browser to fetch as its own task).

What about MergeSort? What about something like the energy grid with permeable walls? What about our ‘max finder’ problem? Coffee shop?

Can you think of other specific problems that might have several choices for “chunks” of work that are independent of each other?
Tweaking or checking on your pool...

The base class of the Executor and ExecutorService classes is ThreadPoolExecutor. You can access some methods from that based by casting the higher object to it. Casting might look like ((ThreadPoolExecutor) myThreadPool).

For example...
- To see how many things are queued up: getQueue().size()
- To pre-start a thread (assuming room exists): prestartCoreThread()
- To pre-start the full capacity of threads: prestartAllCoreThreads()
- To specify that threads that have been idle for a long time should be killed off: setKeepAliveTime()
- If you want to have idle threads trimmed off but have a minimum number of threads you want around: setCorePoolSize();

Creating your own ThreadPoolExecutor

If you want even more control, don’t use any of the factory methods provided but rather build your own ThreadPoolExecutor object. As part of this you can even provide your own factory for spawning the threads.

You get to choose the type of work queue to use to store the tasks that are waiting for a thread to become available and (if for example you provide a bounded queue).

You also get to set a handler that determines how to deal with tasks that aren’t accepted (either to the work queue being full or the pool being shut down). The ThreadPoolExecutor class provides a few useful pre-made static RejectedExecutionHandler classes for you to use (AbortPolicy, DiscardPolicy, DiscardOldestPolicy, CallerRunsPolicy). The CallerRunsPolicy is interesting in that it makes the thread that asked the task to run to run the task.
Extending ThreadPoolExecutor

If you create a new class that extends ThreadPoolExecutor you can also specify your own beforeExecute method and/or your own afterExecute method to run before/after each tasks is run in one of the pool’s threads.

You can also add a terminated method that gets called when the executor itself is terminated.