java.util.concurrent

CSCI 5828: Foundations of Software Engineering
Lecture 25 — 11/15/2016

Goals

- Explore the services of java.util.concurrent

  - ExecutorService

  - Callable/Future

  - ForkJoinPool and ForkJoinTask

- The examples in this lecture come from the excellent book: <u>Programming Concurrency on the JVM</u> from <u>Pragmatic Programmers</u>

  - Source Code for these examples is available here:

    - <u>https://pragprog.com/book/vspcon/programming-concurrency-on-the-jvm#links</u>

  - I'm not allowed to distribute it myself

ExecutorService (I)

- ExecutorService is a Java interface that defines a common set of services for an abstract *thread pool*;

  - this interface has a variety of concrete implementations that provide a range of concurrent behavior to developers

- What's a thread pool? *(A review)*

  - Thread creation is a slow process

  - Thread pools create a bunch of threads all at once (typically at launch)

  - When a new thread is needed, one is taken from the pool and it starts executing immediately

    - very helpful in situations where, e.g., a server is responding to incoming network requests

ExecutorService (II)

- Static factory methods on the Executors class are used to create instances of the ExecutorService; for instance

  - **CachedThreadPool**: creates threads as needed but will reuse previous ones if they are available

  - **FixedThreadPool**: creates a fixed set of threads

  - **ScheduledThreadPool**: creates a thread pool that can execute tasks after a delay or periodically

  - **SingleThreadExecutor**: creates a thread pool with only a single thread

- You write code that only depends on the *interface* `ExecutorService` and then be free to select the actual threading behavior you get *at run-time* based on external factors
  - you can even *switch* threading behaviors ***on the fly***

ExecutorService (III)

- The <u>API of the ExecutorService</u> allows you to

  - submit a single task for execution

  - submit a collection of tasks for execution

    - where you want all of the tasks results (invokeAll)

    - or where you want just one of the results (invokeAny)

  - shutdown the thread pool when you are done with it

Callable/Future: Making this all work

- In order to give tasks to the thread pool and receive results back, you make use of two additional interfaces

  - `Callable<T> and Future<T>`

- Both make use of Java generics to give flexibility in the return types of the computation

  - For instance, I can promise that my task returns a string

  ```
  Callable<String> callMe = new Callable<String>() {

    public String call() throws Exception { …; return result; }

  }
  ```

  - `callMe` is now a Task that I can hand to an ExecutorService

Callable/Future (II)

- When I give `callMe` to an `ExecutorService`, it is going to hand the task to a thread and ask the thread to execute it

  - At the time, we have no idea how long it will take for the task to complete

    - Thus, the `ExecutorService` gives me an instance of `Future<String>` so I can get the value once the task is complete

      - `Future<String> myString = service.submit(callMe);`

  - This call does **NOT** block, I get a reference to `myString` almost immediately

  - I can then decide to retrieve the string whenever I need it by calling `get()`

    - `String result = myString.get();`

  - This call **MAY** block, if the task is still being executed; otherwise, I get the result right away. I can also call a version of `get()` that accepts a timeout.

Portfolio Calculator (I)

- A simple program to retrieve stock quotes from Yahoo

  - Designed as one abstract superclass `AbstractNAV` (Net Asset Value)

    - methods for

      - reading in stock symbols and number of shares

      - timing how long it takes to calculate the portfolio

  - Two subclasses

    - one sequential => loops over list of symbols and calls Yahoo to get current price

    - one concurrent => creates futures for each stock to retrieve prices; hands them all over to an executor service to execute in parallel

Portfolio Calculator (II)

- Since these tasks are io bound, the program needs to decide how many threads it will need

    - Each task is going to be blocked for most of its life and then it will do a single calculation: `numberOfShares * retrievedPrice`

        - We estimate that waiting for Yahoo to give us the current price is going to take about 90% of the task's life cycle

    - So to estimate the number of threads, we use the following formula

        - `Number of Threads = Number of Cores / (1 - Blocking Coefficient)`

    - My machine has 8 cores, so

        - `8/(1-0.9) = 8/.1 = 80` threads

Portfolio Calculator (III)

- It then creates an array to store all of our Callable objects

  - `final List<Callable<Double>> partitions = new ArrayList<Callable<Double>>();`

- It populates that List by creating one Callable<Double> for each stock symbol
  - It then hands the list over to the executor service which hands back a list of Future<Double> objects

    - `final List<Future<Double>> valueOfStocks = executorPool.invokeAll(partitions, 10000, TimeUnit.SECONDS);`

  - Finally, it loops over each Future object and totals up the final value
    - `for (final Future<Double> valueOfAStock : valueOfStocks) {`
      - `netAssetValue += valueOfAStock.get();`
    - `}`

  - It is this call to get() that can finally block, waiting for the task to complete

Portfolio Calculator (IV)

- Let's see this in action

  - As you will see, the concurrent version of the program is significantly faster

    - Why? => Latency!

  - Each request to Yahoo takes a certain amount of time to create the connection, wait for Yahoo to retrieve the data, and stream the result back

- With the sequential version of the program, we take that latency and add it for each stock request; say the latency was 2 seconds

  - For 80 stocks, we would expect to wait 160 seconds for the whole sequential program to complete

  - In the concurrent program, all tasks contact Yahoo at the same time, the 2 second latency for each task overlaps. As a result, the program takes ~2-3 seconds

Finding Primes

- The reason we saw such an amazing speed-up with the previous program was due to the fact that its tasks were IO-bound.

  - With compute-bound tasks, we have to limit the number of threads to the number of cores

  - Let's just look at the code to see how it creates a bunch of Callable<Integer> tasks that count primes for a given partition

    - The executor service then gives us back a list of Future<Integers> and we call those to total up the number of primes in a given range

Coordinating Threads (I)

- A key challenge in the design of concurrent systems is the coordination of threads

  - We may want to

    - start them

    - wait for them to finish

    - assign tasks to them

    - retrieve results from them

    - allow threads to exchange data

    - etc.

Coordinating Threads (II)

- With the `ExecutorService`, the most typical case now involves

    - submitting a task to a thread pool of type `Callable`

    - receiving a `Future` in response

    - when ready, calling `get()` on the `Future` to retrieve the result

- Let's see this in action with an example called `File Size Calculator`

    - First, let's take a look at the sequential version of this program

- Design is straightforward; recursive function that returns either the size for a single file or, for directories, the combined size of all of its children

Disk Cache

- With programs that target the disk, performance will vary

  - The first time through a particular section of the disk, the time will be slower than subsequent runs on the same section of the disk

- The reason for this is the disk cache

  - The operating system will

    - take the most recently read sections of disk

    - and cache them in memory

    - under the assumption that they will be read again fairly soon

- The difference may not be major but it will be there

  - First run sequential on `/usr`: 36.02 seconds; Second run: 27.3 seconds

First Stab at Concurrency

- Creates a thread pool of 100 threads

- Makes use of recursive function to calculate size of files and directories

  - If its handed a file, return the file size

  - If its handed a directory

    - loop through children

      - submit() a task to the thread pool to calculate the size of the child

        - Each task is a `Callable<Long>`

        - Thread pool returns a `Future<Long>` that gets added to an array

    - loop through array calling get() on each Future to add up subtotals

    - return the result

Result? DEADLOCK!

- This approach to the program has a flaw that appears on "deep directories"

  - Each task adds new tasks to the thread pool and then waits for those tasks to return

  - That means that the calling task is STILL ON THE POOL

    - blocked waiting for its subtasks to complete

  - If your directory has lots of subdirectories (more than 100 in this case)

    - You can get into the situation where each of the 100 threads in the thread pool are blocked waiting for subdirectory calculations to complete

      - when this happens, the program deadlocks

        - or thanks to the timeout that we set, eventually the timeout fires and the program terminates

Discussion

- This problem is unfortunate because

  - the approach is straightforward and understandable

    - you'd likely come up with it on a first pass design

- But, a machine's resources are finite

  - you might be able to make this code work on more directories by upping the number of threads

  - but that approach is not generic

    - eventually you'll run into the limit concerning the number of threads the operating system will allow a single process to create

    - and you'll be stuck

New Approach: Find Directories, Total Later

- To make progress, we need an approach that

  - submits tasks for sub-directories

    - but doesn't require the submitting task to hang around for the results

- New Approach

  - Create a data structure that holds the total size of a directory's files and a list of all of that directory's sub-directories

  - Tasks now calculate the size of files in their assigned directory and create a list of all subdirectories; allowing them to complete and not stick around

  - The main thread takes care of submitting new tasks and totaling results

    - 6 seconds vs. 36 seconds!

Terrific Results But…

- increased complexity!

  - We got great results but the approach we used is not intuitive

    - Creating a class to store partial (immutable) results

    - Creating the function executed by tasks such that it completes quickly

    - Adopting a while loop strategy in main to iterate while there were directories to process

      - and then ensuring that the while loop would not terminate until all directories had been processed

- Let's look at features that java.util.concurrent has that might reduce the complexity of the code

CountDownLatch (I)

- The next approach examines the use of a CountDownLatch

  - plus it relaxes our constraint to avoid shared mutability

- but it achieves the same results with simpler code

  - Simplicity is not to be discounted

    - it has significant impacts on the ability to maintain software systems

CountDownLatch (II)

- What's a `CountDownLatch`?

  - It is a synchronization aid to help coordinate threads

  - It maintains a count and has three primary methods

    - `CountDownLatch(n)` - creates the latch with a specific count

    - `await()` - block the calling thread until the latch's `count == 0`

    - `countDown()` -- decrement the count of the latch

- Typical scenario:

  - create a bunch of threads and `start()` them
  - but don't let them `run()` until some point in the future
    - i.e. have their first line in `run()` call `await()`

New Approach

- Instead of returning subdirectories, we let each task update two shared variables

    - each an instance of AtomicLong (like AtomicInteger but stores long value)

- One AtomicLong stores the total file size

- The second AtomicLong stores the number of "pending file visits"

    - This value gets incremented each time we find a subdirectory to visit

    - It gets decremented each time we are done processing a subdirectory

    - When this value equals zero, we call countDown() on the latch

- The main thread initializes the latch to a value of 1, starts the directory search, and calls await()

Performance

- Comparable performance to previous approach (4.6 seconds vs 6.1 seconds)

  - but with simpler code!

Third Approach: Queue (I)

- We have seen two approaches for exchanging data between threads

  - `Callable/Future` and `Atomic<Type>`

- both techniques ensured that we could pass information between threads

- A third approach is to use a data structure such as a queue to pass information between threads

  - as long as there is space in the queue, producers will not block

  - as long as there are items on the queue, consumers will not block

  - contention will occur only when the queue is full (producers) or when it is empty (consumers)

Third Approach: Queue (II)

- This version of the program creates a blocking queue with 500 slots

- An atomic long is used to keep track of pending file visits

- Tasks traverse the directories as normal, adding file sizes to the queue and updating the atomic long as they submit more tasks to the thread pool

- The main program kicks off the traversal and then sits in a loop

  - it reads items off the queue until there are no more file visits pending and the queue is empty

- Performance:

  - First Run: 4.96 seconds

  - Essentially same performance, just slightly different abstractions, perhaps simpler

    - not by much

Java 7: Fork-Join API

- Java 7 introduced a new type of thread pool and task

    - `ForkJoinPool` and `ForkJoinTask`

- The key benefit of this new thread pool is that threads can steal tasks generated by other active tasks

    - This solves the problem we encountered with the first approach to the concurrent file size calculator

    - When a task generates a bunch of other tasks and blocks, its thread can let it go and work on the other tasks

- With this approach, we get a program very similar to our "naive" approach

    - without the danger for deadlock like we saw before => 5.5 seconds

Summary

- We learned the ins and outs of using the ExecutorService in various ways

  - Saw how Callable and Future work to allow us to pass information between threads

  - Explored various problems that can still occur when using ExecutorService

  - Saw a number of different ways to design the same program

    - Performance was usually the same

      - What was different was the complexity of each design

        - Certain designs provided more simplicity than others

        - If two designs perform the same, prefer the one that is less complex to make it easier to maintain that solution

Coming Up Next

- Lecture 25: Refactoring a poorly designed concurrent program written in Java