Programming Assignment #5 CS671

1st submission due 7/26/2021
2nd submission due 8/2/2021

Programming with threads on the JVM

This assignment illustrates some of the techniques used when writing concurrent programs on the JVM (in Java or in Scala). It focuses on elementary components (e.g., the `Thread` class) and purposely ignores some of the higher-level constructs that are available in Java or Scala (e.g., most of `java.util.concurrent` or the monadic operators of Scala's `Future`). Students are expected to rely on the following constructs:

- `java.lang.Thread` and `java.lang.Runnable`.
- `scala.concurrent.ExecutionContext`, `scala.concurrent.Future` and `scala.concurrent.Promise`.
- `scala.AnyRef.synchronized`.

Furthermore, synchronization among threads must be done properly:

- No busy waiting (spinning).
- No carefully adjusted calls to `Thread.sleep`.

Instead, suitable blocking operations must be used. The only blocking operations needed in this assignment are:

- entry into a `synchronized` method;
- `Semaphore.acquire` (to implement `Dispatcher`);
- `scala.concurrent.Await` (for the `GrepApp` application).

`Thread.sleep` (and even sometimes busy waiting) can only be used when testing and debugging.

Work dispatcher class

The main component of the parallel package is the `Dispatcher` class (Listing 1). A dispatcher is created with a series of work units of type `A`, obtained from an iterator. Worker threads call method `next` to obtain units of work. When all the work units have been dispatched, this method returns `None`. Each work unit is expected to produce a single result of type `B`. For each work unit they process, worker threads should call the method `report` exactly once with the result. This method returns `true` when the last report is received.

```
class Dispatcher[A, B](values: Iterator[A], bound: Option[Int] = None) {
  def next(): Option[A] = ...
  def report(result: B): Boolean = ...
  def future: Future[List[B]] = ...
}
```

Listing 1: `Dispatcher` class.

The results of the work units are only available after all units have been processed (i.e., after `report` returns `true`). They are made available as a `Future[List[B]]`, which is completed when the last report is received, i.e., right before report returns `true`. The list contains the reports in the order in which they were received (which is not necessarily the order in which the work units were dispatched).

A dispatcher may issue a large number of work units and may choose to limit the number of units that are processed concurrently. An optional bound can be passed to the constructor of the class. If present, the bound places a limit on the number of workers actively processing units in parallel. This is implemented by using a semaphore. When a bound is present, threads need to acquire a permit from the semaphore to obtain a work unit. This permit goes back to the semaphore when the report corresponding to the unit is received. When the number of available permits reaches 0, the method `next` becomes blocking (until a permit is released via the method `report`).

The dispatcher is a passive object (it does not create threads) and is *thread-safe*. This means that the methods `next`, `report`, and `future` can be called at any time by any thread.

Worker classes

A dispatcher can be used to represent a certain amount of work to be done in parallel. One way this work can be completed is through class `ThreadWork` (Listing 2). Instances of this class are created with a dispatcher, a work function, a desired number of worker threads and a thread factory. The method `start` creates `workers` threads using the factory, starts them and returns immediately. The threads repeatedly call `next` on the dispatcher to obtain work units. Each work unit is processed by a worker thread using the function `work`. The output of the function is reported back to the dispatcher. When all the work units have been processed, the threads terminate. Note that function `work` is shared among all worker threads and is assumed to be thread-safe (functions tend to be stateless and hence thread-safe).

```
class ThreadWork[A, B](workers: Int, work: A => B,
                       dispatcher: Dispatcher[A, B])
                      (implicit tf: ThreadFactory) {
  def start(): Unit = ...
}
```

Listing 2: `ThreadWork` class.

Class `PoolWork` (Lis. 3) is an alternative to `ThreadWork`. Instead of relying on a factory to create new threads, this class uses existing worker threads from a thread pool. To ensure maximum parallelism, the `start` method creates one task per work unit and submits all the tasks to the thread pool. This is made somewhat trickier by the fact that the total number of units to be produced by a dispatcher is not known *and* the method `next` can be blocking if the dispatcher uses a semaphore. It is important that the method `start` return immediately and not block the calling thread, even if the dispatcher is unable to produce all the tasks right away because of a semaphore. As a consequence, the thread that calls `start` cannot call `next` on the dispatcher directly.

```
class PoolWork[A, B](work: A => B, dispatcher: Dispatcher[A, B])
                    (implicit exec: ExecutionContext) {
  def start(): Unit = ...
}
```

Listing 3: `PoolWork` class.

Fig. 1 shows the output of a simple demo program (provided). This run uses a `PoolWork` with 4 threads to run 5 jobs. The first 4 jobs (of durations 2, 1, 5 and 3) are picked up by the 4 worker threads at time 17. One second later, at time 18, `pool-1-thread-2` finishes its 1-second job and picks up the last job, of duration 4. All the workers finish their jobs one by one. The last to finish, at time 22, are `pool-1-thread-3` (which started the 5-second job at time 17) and `pool-1-thread-2` (which completed the 1-second and 4-second jobs). The results are returned in the order of completion (1, 2, 3, 4, 5 or 1, 2,

3, 5, 4). Note that the calls to `start` on `PoolWork` and to `future` on the dispatcher take no time and are both completed around time 17.

```
> test:runMain Demo 2 1 5 3 4
main at xx:xx:17.305: dispatcher created
main at xx:xx:17.344: workers started
pool-1-thread-1 at xx:xx:17.349: worker gets 2.0
pool-1-thread-2 at xx:xx:17.349: worker gets 1.0
pool-1-thread-4 at xx:xx:17.349: worker gets 3.0
pool-1-thread-3 at xx:xx:17.349: worker gets 5.0
pool-1-thread-2 at xx:xx:18.359: worker gets 4.0
pool-1-thread-2 at xx:xx:22.369: [-1.0, -2.0, -3.0, -5.0, -4.0]
pool-1-thread-2 at xx:xx:22.370: time: 5.018 secs
```

Figure 1: Sample demo run.

This program can be modified for the purpose of experimentation by changing the number of worker threads and by using `ThreadWork` instead of `PoolWork`.[1]

Grep application

Class `GrepApp` (and its companion object) implements a simple application that searches textual sources in parallel. The main method is `find`, which takes as arguments a regular expression and a collection of sources. It creates a dispatcher that produces all the sources, and processes these sources in parallel using an instance of `ThreadWork` or of `PoolWork`. The `find` method blocks while the work is ongoing and returns a list of all matches found in the textual sources, in no particular order.

The class takes a `bound` parameter that can be used to limit the amount of parallelism:

- If a `ThreadWork` object is used, it can create `bound` threads using a simple thread factory. No semaphore is needed.
- If a `PoolWork` object is used, it will rely on `scala.concurrent.ExecutionContext.Implicits.global` for its thread pool (the creation of custom thread pools is beyond the scope of this assignment). The number of threads used in this pool in unknown (and can change dynamically) and may be larger than the specified bound. Accordingly, the dispatcher in this case should use a semaphore with `bound` permits

Figure 2 (on the next page) shows a sample run of the command-line application (provided).

Notes

- Source code is submitted for testing using Git (master branch). The first submission must be tagged 5a and the second submission 5b (even if the first submission is missing). If modifications to the code are needed after tagging a submission, a new tag should be created of the form 5a.1, 5a.2, 5b.1, etc. Do not delete tags!
- In addition to this Git submission, students need to upload on MyCourses a sample of their code for review. Code should be uploaded as a pdf file created from a pretty-printer (e.g., a2ps, pandoc, highlight, IntelliJ, etc.). The sample should be about one page long, and can include

[1] The program is written in such a way that the first two lines are printed by the main thread (`main`) while the last two are printed by one of the workers (`pool-1-thread-2`). This is not important; only the timing values matter here.

specific questions (e.g., “This code is very convoluted; could it be made simpler?” or “This code seems to be very slow and I don’t know why” or “I’m very proud of this code, which follows a pure functional pattern; what do you think?”).

- Students need to implement classes `Dispatcher`, `ThreadWork`, `PoolWork` and `GrepApp`. Class `JavaSem` is given. It implements the `Semaphore` interface and can be used to implement `Dispatcher`.
- In `GrepApp`, each source may produce zero, one or multiple matches. Therefore, the dispatcher cannot be of type `Dispatcher[TextSource,MatchResult]` but must instead use a type that allows work units to return multiple matches, e.g., `Dispatcher[TextSource,List[MatchResult]]`.
- Regular expressions, in Scala, are mostly used through pattern matching. This is convenient to extract groups from complex regular expressions but a bit of an overkill to simply check if a string contains a substring that matches a given regular expression. Method matches can be used, but keep in mind that regular expressions are *anchored* by default in Scala.
- Incorrect synchronization can result in deadlock situations, in which a set of threads get “stuck” waiting for each other. One way to debug these situations is to produce a thread dump that shows where in the source code each thread is blocked. These dumps are obtained by sending a `QUIT` signal to the JVM. This can be achieved from the terminal with `Ctrl-\` or from the IntelliJ IDEA using the “thread dump” button.

```
> runMain parallel.GrepApp file:src/test/resources/alice.txt file:src/test/resources/usconst.txt ch.p 4
Result(file:src/test/resources/usconst.txt,
act shall be selected, and such person shall act accordingly until a President,753)
Result(file:src/test/resources/usconst.txt,
transmits to them a written declaration to the contrary, such powers and duties,833)
Result(file:src/test/resources/alice.txt,
’It was much pleasanter at home,’ thought poor Alice, ’when one wasn’t,735)
Result(file:src/test/resources/alice.txt,
which produced another dead silence.,851)
Result(file:src/test/resources/alice.txt,
’There’s certainly too much pepper in that soup!’ Alice said to herself,,1304)
Result(file:src/test/resources/alice.txt,
’Talking of axes,’ said the Duchess, ’chop off her head!’,1354)
Result(file:src/test/resources/alice.txt,
she went on, very much pleased at having found out a new kind of,2196)
Result(file:src/test/resources/alice.txt,
Just at this moment Alice felt a very curious sensation, which puzzled,2914)
```

Figure 2: Sample `GrepApp` run.