TD MCP – session 4 – Thread Pool

March 19, 2021

Learning objective: explore executor implementations, how to submit tasks, how to divide work into multiple tasks and how fork-join pools help implementing divide-and-conquer algorithms.

The exercises until the seizing are essential for the corresponding practical session.

**Exercise 1: direct executor**

Implement an executor \((\text{void} \ \text{execute}(\text{Runnable} \ r))\) that launches directly the execution of each runnable synchronously.

```java
class DirectExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    }
}
```

**Exercise 2: one thread per task**

Implement an executor that launches the execution of each runnable asynchronously in a new thread.

```java
class DirectExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    }
}
```

**Exercise 3: serialized executions**

Implement an executor that launches the execution of each runnable in sequence in a thread. Task submissions should be asynchronous.

A first solution where multiple threads may be used but executions are correctly serialized (the trick is to create a new runnable executing the initial runnable with an additional call to a scheduling method):

```java
class SerialExecutor implements Executor {
    List<Runnable> tasks = new ArrayList<>();
    Runnable active;
    ```
public synchronized void execute(Runnable r) {
    tasks.add(() -> {
        r.run();
        scheduleNext();
    });
    if (active == null)
        scheduleNext();
}

synchronized void scheduleNext() {
    if (!tasks.isEmpty()) {
        active = tasks.remove(0);
        new Thread(active).start();
    } else
        active = null;
}

The method **execute** must be protected with a **synchronized** block because of the list modification. The second method must be synchronized to avoid the following scenario:

1. initially, the list is empty and an active thread is executing a task;
2. when the active thread executes **scheduleNext**, it pauses just after the emptiness test and before changing **active**;
3. a new task is submitted to the executor;
4. since **active** is not null yet, **scheduleNext** is not called;
5. the active thread resumes its activity, put **active** to null and completes;
6. no new active threads will be created for the waiting task until a new task is submitted.

Alternatively, it is possible to use a thread that executes all tasks given in a collection in sequence, waiting for new runnables to be submitted when the collection is empty):

class SerialExecutor implements Executor {
    List<Runnable> tasks = new ArrayList<>();

    SerialExecutor() {
        new Thread(() -> {
            Runnable active = null;
            synchronized(tasks) {
                if (!tasks.isEmpty())
                    active = tasks.remove(0);
            }
            if (active != null)
                active.run();
        }).start();
    }

    public void execute(Runnable r) {
        synchronized(tasks) {
            tasks.add(r);
        }
    }
}
This last solution could be improved to make the thread wait with a `wait/notify` mechanism instead of an active loop. Moreover, instead of using a `List`, a queue would be more relevant for a FIFO mechanism. Finally, instead of relying on a `synchronized` construct, a concurrent data structure could be used such as using `synchronizedList` from `Collections`, or even better a `ConcurrentLinkedDeque`.

**Exercise 4: seizing a thread pool**

We consider a large number of tasks that each consists of a compute time $C$ and a wait time $W$. We assume the overlap of the waiting times is as low as possible. If the number of threads in the pool is too big, the threads end up competing for scarce cores and memory resources, wasting their time performing context switching. What is the minimum size for a thread pool to fully use all $n$ cores?

If tasks spend half their time waiting, then it is necessary to have $2n$ threads to have $n$ active ones in the OS. If tasks spend three quarter of their time waiting, then it is $4n$. The general minimum number of threads is $n(1 + W/C)$.

**Exercise 5: Fibonacci on an executor service**

Write an implementation that computes a given number of the Fibonacci sequence using an executor service. Explain which executor service must be chosen.

```java
class FibTask implements Callable<Integer> {
    ExecutorService exec = Executors.newCachedThreadPool();
    int arg;
    public FibTask(int n) {
        arg = n;
    }
    public Integer call() {
        if (arg > 2) {
            Future<Integer> left = exec.submit(new FibTask(arg - 1));
            Future<Integer> right = exec.submit(new FibTask(arg - 2));
            return left.get() + right.get();
        } else {
            return 1;
        }
    }
}
```

To avoid a deadlock, the executor must create a new thread for each call, otherwise the threads of the pool will all be busy executing tasks that are blocking to get the results from the futures. The following exercise proposes a partial solution to this issue.

**Exercise 6: sum of square values on an executor service**

Implement an algorithm that sums the square of all values in an array using an executor service.
class SumOfSquaresTask implements Callable<Integer> {
    static ExecutorService exec = Executors.newFixedThreadPool(4);
    int[] array; int lo, hi;
    SumOfSquaresTask(int[] array, int lo, int hi) {
        this.array = array; this.lo = lo; this hi = hi;
    }
    public Integer call() {
        if (hi != lo) {
            int mid = (lo + hi) >>> 1;
            Future<Integer> left = exec.submit(
                new SumOfSquaresTask(array, lo, mid));
            Integer right =
                new SumOfSquaresTask(array, mid + 1, hi).call();
            return left.get() + right;
        } else {
            return array[lo] * array[lo];
        }
    }
}

Note that this solution avoids copying the main data in the array. The separation code consists
in adapting the index bounds.

This solution is heavy because it requires handling explicitly the executor. Moreover, even though
this solution incurs less blocking than in the previous exercise, it still ends up in a deadlock for
the same reason. We will see that fork-join pools may be used to have more abstract code and to
solve this deadlock issue.