Lab 5 — Multithreading

CS 205

Lab objectives:

- Create multithreaded code.
- Correct synchronization issues within a multithreaded program with locks.

1. You will start by using multiple threads.

   As you progress through these steps, don't delete code from previous steps. Instead, comment previous code out. For example, in step 2b below, you use a `ReentrantLock` that you replace in step 2c. Comment-out the `ReentrantLock` code that you no longer need.

   (a) Download and import the `lab5Starter` project into Eclipse. Take a look at the project's `printPrimes` method. What does the method do? (Note: `n.isProbablePrime(100)` returns true if `n` is a prime, with error probability $< 2^{-100}$. Assume for now that means that `n` is certainly a prime.)

   (b) Take a look at the main method that executes two threads, and run the program. What happens? How can you tell that the two threads ran concurrently? Does the main thread finish executing before the other two threads?

   (c) Now we want to know how many primes are in a given interval instead of printing them to `System.out`. We need to use a `Callable` since a `Runnable` can't return a value. Make a function `countPrimes` that returns a `Callable<Long>`. Make two callables

   ```
   Callable<Long> c1 = countPrimes(new BigInteger("1000000000000000"),
                                   500_000);
   Callable<Long> c2 = countPrimes(new BigInteger("1000000000500000"),
                                   500_000);
   ```

   Submit them both to the same `ExecutorService` as in the preceding example. You'll get two `Future<Long>` values. You can access these values by

   ```
   System.out.println(f1.get());
   System.out.println(f2.get());
   ```

   before calling

   ```
   service.shutdown();
   ```

   (d) Run the program. What does it print? Run it again. Does it print the same thing? The `isProbablePrime` method sounds as if it was guessing, but it is actually perfectly deterministic. For a given `n`, the call `n.isProbablePrime(100)` will always give the same result. It is just that the result may be wrong. The chance for that is 2^{-100} or about 10^{-30}. The probability of you being struck by lightning in a given year is about 10^{-6}. The probability of five of us being struck by lightning in the same year is about

10^{-30}. If that's what keeps you up at night, then you should definitely worry about `n.isProbablePrime(100)` giving you the wrong answer.

(e) Let's find out if running the two tasks in parallel does any good. Add these calls around the calls to `service.submit` and `fn.get`

```
long start = System.currentTimeMillis();

// service.submit and f1.get and f2.get calls

long end = System.currentTimeMillis();
System.out.println("Milliseconds: " + (end - start));
```

Run the program and write down the number of milliseconds. Then change

```
Executors.newFixedThreadPool(2);
```

to

```
Executors.newFixedThreadPool(1);
```

which means that only one thread is available. Run the program again. You should notice a delay between the printouts of the two counts. Did you get the same counts in both the faster and the slower run?

2. Now let's compute the count differently.

(a) We'll increment a shared counter. Add a field

```
private long nonprime = 0;
```

In the `countPrimes` method, increment `nonprime` when a number isn't a prime. After printing the counts of primes, add a call

```
System.out.println(nonprime);
```

Run the program a few times. What results do you get? Which values are the same, and which are different in each run?

(b) As you can see, incrementing a counter from two threads doesn't work reliably — in other words, it doesn't work. Use a `ReentrantLock` to make it work.

(c) Objects contain their own built-in locks. Make a `Counter` class with synchronized methods `increment` and `get`. Remove the `lock` from the previous step and make `nonprime` into an instance of your `Counter` class. Verify that your program works.

3. The preceding program has two tasks that count primes. Now we want to do some work with them.

(a) Create a method that places primes into a queue:

```
public Runnable producePrimes(BigInteger start, long length,
                              BlockingQueue<BigInteger> queue)
```

You will want to use `put`, not `add`, for placing items into the queue so that the thread will wait if the queue is full. Add a method

```
public Runnable consumePrimes(BlockingQueue<BigInteger> queue, ...);
```

that removes primes from the queue and prints those that have at most three distinct digits. You will want to use the method `take` to remove items from the queue so that the thread will wait if there are no items to remove. Here is a method for getting all distinct characters in a string:

```
private String distinct(String s)
{
    StringBuilder result = new StringBuilder();
    int i = 0;
    while (i < s.length())
    {
        int cp = s.codePointAt(i);
        int cc = Character.charCount(cp);
        if (result.indexOf(s.substring(i, i + cc)) == -1)
            result.appendCodePoint(cp);
        i += cc;
    }
    return result.toString();
}
```

In the main method, make an `ArrayBlockingQueue` of capacity `1000`. Change the `newFixedThreadPool` call to have `3` threads. Add the three runnables:

```
... producePrimes(new BigInteger("1000000000000000"), 500_000, queue);
... producePrimes(new BigInteger("1000000000500000"), 500_000, queue);
... consumePrimes(queue, ...);
```

(b) How does the consumer know when it is done? Come up with some mechanism that works.

Add Javadoc comments to your project. Export your lab into a ZIP archive and submit it in Canvas.