Assignments for week 1

Anders Degn Lapiki, Jacob Kjærulff Furberg, Jonas Ishøj Nielsen

03/09/2020

Contents

1 1.1

1.1 1.1.1 - green - Result of running main, when increment is not synchronized

First attempt gave: "Count is 10052047 and should be 20000000".
Second attempt gave: "10319684 and should be 20000000".
Third attempt gave: "Count is 10319684 and should be 20000000".

1.2 1.1.2 - green - Counts value from 10 million to 100

- First attempt gave: "Count is 187 and should be 200".
- Second attempt gave: "Count is 200 and should be 200".
- Third attempt gave: "Count is 189 and should be 200".

Reducing the count reduces the chance that the result is wrong due to unlucky timing, as there are less calls that they can occur on.
It is not thread safe as it clearly doesn't always give the correct result.

1.3 1.1.3 - green - Use count += 1 or count++

Running the changed code doesn't affect the result as all three operations are non-atomic Read-modify-write operations and are thus still susceptible to race conditions.

1.4 1.1.4 - yellow - Use javap -c TestLongCounterExperiments to compare the 3 increments

All 3 produced byte-codes appear to be identical, indicating that all 3 operations are interpreted the same way, which verifies the explanation from section 1.3.

1.5 1.1.5 - yellow - Adding decrement function

Without 'synchronized' the count field will still experience race conditions. When one is synchronized, that function has a lesser effect on the final result, as it can only change the value by 1, but the other will also overwrite any of the synchronized function's changes that happens between its read and write. Because of that both functions needs to be synchronized. When both functions are synchronized running main gives the expected result of "Count is 0 and should be 0".

1.6 1.1.6 - red - Experiment

Running with neither increment or decrement synchronize gave the result:

- First attempt gave: "Count is -9837290 and should be 0".
- Second attempt gave: "Count is -8841014 and should be 0".
- Third attempt gave: "Count is 7184273 and should be 0".

Running with only decrement synchronized gave the result:

- First attempt gave: "Count is 150334 and should be 0".
- Second attempt gave: "Count is 136986 and should be 0".
- Third attempt gave: "Count is 168400 and should be 0".

Running with only increment synchronized gave the result:

- First attempt gave: "Count is -186723 and should be 0".
- Second attempt gave: "Count is -341993 and should be 0".
- Third attempt gave: "Count is -126350 and should be 0".

Running with both being synchronized gives the result "Count is 0 and should be 0".
The values are not unsurprising, since, as mentioned in section 1.5, the non-synchronized function(s) will overwrite the write operations done by other functions. Therefore when only one isn't synchronized, that function will have a bigger impact on the value of count, and make it skewed int hat direction.

2 1.2 - yellow

Running with a range set to 20_000_000 gives the following results:

- Parallel10000 result: 1270607 in: 7550

- Parallel2000 result: 1270607 in: 6659
- Parallel1000 result: 1270607 in: 6620
- Parallel500 result: 1270607 in: 6613
- Parallel250 result: 1270607 in: 6645
- Parallel20 result: 1270607 in: 6831
- Parallel12 result: 1270607 in: 7014
- Parallel6 result: 1270607 in: 8601
- Parallel2 result: 1270607 in: 21783

Which seems to indicate that around 500-1000 threads are maximum it can utilize for this task.
Running with a range set to 100_000_000 gives the following results:

- Parallel10000 result: 5761455 in: 69735
- Parallel2000 result: 5761455 in: 68458
- Parallel1000 result: 5761455 in: 67794
- Parallel500 result: 5761455 in: 67983
- Parallel250 result: 5761455 in: 68804
- Parallel20 result: 5761455 in: 69165
- Parallel12 result: 5761455 in: 71441
- Parallel6 result: 5761455 in: 89708
- Parallel2 result: 5761455 in: 220444

Which seems to indicate that around 1000 threads is maximum it can utilize for this task.

3 1.3

3.1 1.3.1 - green

Code can be seen below:

```
public static void main(String[] args) {
    Printer p = new Printer();
    Thread t1 = new Thread(() -> { while(true) p.print(); });
    Thread t2 = new Thread(() -> { while(true) p.print(); });
    t1.start(); t2.start();
}
```

3.2 1.3.2 - green

The weaving faults occur when a thread A prints — and before A prints the dash, thread B prints its —.

3.3 1.3.3 - green

The weaving faults no longer happens, as the Printer p is used as lock and only one thread at a time can call the print function. If each thread had used its own printer object, making it synchronized wouldn't have changed anything.

3.4 1.3.4 - green

Below is a print statement using a synchronized block.

```
public void print() {
  synchronized (this){
    System.out.print("-");
    try { Thread.sleep(50); } catch (InterruptedException exn) { }
    System.out.print("|");
  }
}
```

3.5 1.3.5 - yellow

Below is a synchronized static print function.

```
public class PrinterThread {
  public static void main(String[] args) {
    Printer p = new Printer();
    Thread t1 = new Thread(() -> { while(true) Printer.print(); });
    Thread t2 = new Thread(() -> { while(true) Printer.print(); });
    t1.start(); t2.start();
  }
}
class Printer {
  public static void print() {
    synchronized (Printer.class){
      System.out.print("-");
      try { Thread.sleep(50); } catch (InterruptedException exn) { }
      System.out.print("|");
    }
  }
}
```

3.6 1.3.6 - red

In figure 1 the ordering of the operations are shown without synchronization at top, and with below.

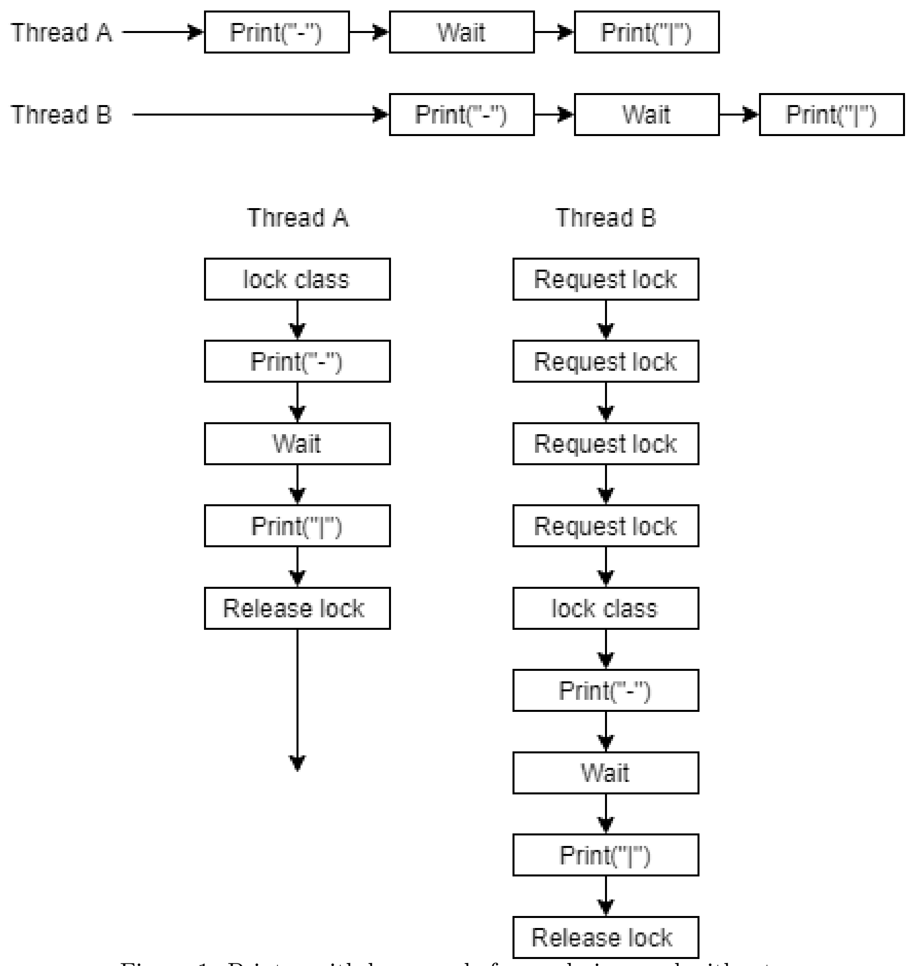

Figure 1: Printer with happens-before ordering, and without

4 1.4

4.1 1.4 - green

There aren't any systems included in Gotz and not in Kristen and vice versa. Resource utilization and exploitation of multiprocessors centers around efficiency of letting another program run if waiting for external operations. Both fairness and concealed parallelism gives the goal of letting programs share machines resources. Convenience and Intrinsic parallelism both covers that parallelism makes it easier to model and write programs for the real world. Convenience and exploitation of multiprocessors also both centers around how multitasking is desirable for efficiency of splitting up a task.

4.2 1.4 - yellow

Exploitation of multiprocessors
Any program with a front-end not freezing while the back-end is handling the user's requests.

Concealed parallelism
Analysing splitting branches of a large tree structure, with a shared variable to tell other threads if one threat succeeded in finding the target element, thus drastically decreasing search time.

Intrinsic parallelism
When awaiting a potential respond from an api/connected servers.

4.3 1.4 - red

With poor exploitation of multiprocessors writing programs efficiently using the other categories would be difficult, and for intrinsic parallelism it would be less effective.
Concealed parallelism is almost a necessity for the other categories to be actualised. The reason being that shared resources is the easiest way for threads to communicate and it is needed for many tasks that threads are used for.
Without intrinsic parallelism, best guess would be that exploiting simultaneous access to a number of physical computers would be impossible.

5 1.5 - yellow

Running the program 5 times gave the results:

- Sum is 1641816.000000 and should be 2000000.000000
- Sum is 1826000.000000 and should be 2000000.000000
- Sum is 1658258.000000 and should be 2000000.000000

- Sum is 1684161.000000 and should be 2000000.000000
- Sum is 1776511.000000 and should be 2000000.000000

The reason that Mystery isn't thread-safe is, that the method is guarded by the object instance m, while the lock for the parameter is held by the Mystery class.
The class can become thread-safe by making each access to sum be guarded by the same object, which is done by replacing line 27 with:

```
public void addInstance(double x) {
   synchronized (Mystery.class) {sum += x;}
}
```

6 1.6 - red

Because increment and increment4 is implemented in different classes, the synchronized keyword doesn't assign the same class to guard each function.
To fix this race-condition change line 37 to:

```
public static void increment4() {
    synchronized (MysteryA.class){count += 4;}
}
```

7 1.7 - red

The bytecode for the two functions can be seen below:

```
public synchronized void increment();
    Code:
       0: aload_0
       1: dup
       2: getfield      #7                  // Field l:J
       5: lconst_1
       6: ladd
       7: putfield      #7                  // Field l:J
      10: return

  public void decrement();
    Code:
       0: aload_0
       1: dup
       2: astore_1
       3: monitorenter
       4: aload_0
```

```
 5: dup
 6: getfield       #7                    // Field l:J
 9: lconst_1
10: lsub
11: putfield       #7                    // Field l:J
14: aload_1
15: monitorexit
16: goto           24
19: astore_2
20: aload_1
21: monitorexit
22: aload_2
23: athrow
24: return
```

The difference happens after the second dup where increment getfield and decrement before that gets the mutex from the monitor center. The bytecode between the getfield and the putfield has also changed. Whereas afterwards the increment returns decrement, and loads the key and tells monitor center to release it.
This seems to indicate that a synchronized function handles the synchronization before entering the function body.