Exercises week 1
Friday 29 August 2014

Goal of the exercises

The goal of this week's exercises is to make sure that you can use Java inner classes, Java threads, `synchronized` methods, and the `synchronized` statement on small examples.

The following abbreviations are used in the exercise sheets:

- "Goetz" means Goetz et al.: *Java Concurrency in Practice*, Addison-Wesley 2006.
- "Bloch" means Bloch: *Effective Java*. Second edition, Addison-Wesley 2008.
- "Herlihy" means Herlihy and Shavit: *The Art of Multiprocessor Programming*. Revised reprint, Morgan Kaufmann 2012.

The exercises let you try yourself the ideas and concepts that were introduced in the lectures. Some exercises may be challenging, but they are not supposed to require days of work.

If you get stuck with an exercise outside the exercise sessions, you may use the News Forum for the course in LearnIT https://learnit.itu.dk/mod/forum/view.php?id=32825 to ask for help. This is better than emailing the teaching assistants individually.

Exercises may be solved and solutions handed in in groups of 1 or 2 students.

Exercise solutions should be **handed in through LearnIT** no later than 23:55 on the Thursday following the exercise date.

How to hand in

You should make hand-ins as simple as possible for you and for the teaching assistants. For instance, hand in a zip-file containing the Java source files written to answer the programming questions. Use Java comments to clearly indicate which part of the code relates to which exercise.

You may also use Java comments in the source files to reply to the text questions of the exercises, and to present output from experiments. Alternatively use simple text files for this purpose, but then name the files to make it completely clear what files contain solutions to what questions. In general, do not waste your time formatting everything beautifully with LaTeX or MS Word, unless this is actually faster for you.

Do **not** submit code in the form of screenshots. Do **not** hand in rar files and other exotic archive formats. Do **not** hand in zip-files of complete Eclipse workspaces and similar; they contain extraneous junk.

Do this first

Make sure you have a recent version of the Java Development Kit installed: Java 8 is clearly preferable, but Java 7 will work for most of the course. Type `java -version` in a console on Windows, MacOS or Linux to see what version you have. From inside Eclipse you may instead inspect `Preferences > Java > Installed JREs`.

You may want to install a recent version of an integrated development environment such as Eclipse Luna (4.4).

Get and unpack this week's example code in zip file pcpp-week01.zip on the course homepage.

Exercise 1.1 Consider the lecture's LongCounter example found in file TestLongCounterExperiments.java, and **remove** the `synchronized` keyword from method `increment` so you get this class:

```
class LongCounter {
  private long count = 0;
  public void increment() {
    count = count + 1;
  }
  public synchronized long get() {
    return count;
  }
}
```

1. The `main` method creates a LongCounter object. Then it creates and starts two threads that run concurrently, and each increments the `count` field 10 million times by calling method `increment`.

   What kind of final values do you get when the increment method is **not** synchronized?

2. Reduce the `counts` value from 10 million to 100, recompile, and rerun the code. It is now likely that you get the correct result (200) in every run. Explain how this could be. Would you consider this software correct, in the sense that you would guarantee that it always gives 200?

3. The `increment` method in LongCounter uses the assignment

   ```
   count = count + 1;
   ```

   to add one to `count`. This could be expressed also as `count += 1` or as `count++`.

   Do you think it would make any difference to use one of these forms instead? Why? Change the code and run it, do you see any difference in the results for any of these alternatives?

4. Extend the LongCounter class with a `decrement()` method which subtracts 1 from the `count` field.

   Change the code in `main` so that `t1` calls `decrement` 10 million times, and `t2` calls `increment` 10 million times, on a LongCounter instance. In particular, initialize `main`'s `counts` variable to 10 million as before.

   What should the final value be, after both threads have completed?

   Note that `decrement` is called only from one thread, and `increment` is called only from another thread. So do the methods have to be `synchronized` for the example to produce the expected final value? Explain why (or why not).

5. Make four experiments: (i) Run the example without `synchronized` on any of the methods; (ii) with only `decrement` being synchronized; (iii) with only `increment` being synchronized; and (iv) with both being synchronized. List some of the final values you get in each case. Explain how they could arise.

Exercise 1.2 This exercise concerns anonymous inner classes and has nothing to do with concurrency. Consider a method `doTwice(r)` that takes a Runnable instance `r` and executes it twice:

```
public static void doTwice(Runnable r) {
  r.run();
  r.run();
}
```

1. Write a call to `doTwice` that uses an anonymous inner class (one that implements Runnable) to print the same string twice, like this:

   ```
   Hello, World!
   Hello, World!
   ```

2. Define a static method `doNTimes(r, n)` that takes a Runnable instance `r` and executes it `n` times:

```
public static void doNTimes(Runnable r, int n) {
  ... some code ...
}
```

3. Write a call to `doNTimes` that prints the same string 14 times.
4. Define a static method `write14Times(s)` that uses `doNTimes` to print given string `s` 14 times.
5. Optional exercise: Write the above calls to `doTwice` and `doNTimes` using Java 8 lambdas, or anonymous functions, instead of anonymous inner classes that implement Runnable.

Exercise 1.3 Consider this class, whose `print` method prints a dash "`-`", waits for 50 milliseconds, and then prints a vertical bar "`|`":

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

1. Write a program that creates a Printer object `p`, and then creates and starts two threads. Each thread must call `p.print()` forever. You will observe that most of the time the dash and bar symbols alternate neatly as in `-|-|-|-|-|-|-|`.

   But occasionally two bars are printed in a row, or two dashes are printed in a row, creating small "weaving faults" like those shown below:

   ```
   |-|-|-|-|-|-||--|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
   |-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
   |-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
   |-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
   |-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-||--
   |-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
   |-|-|-|-|-|-|-|-|-|-|-|-||--|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
   |-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
   |-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
   ```

   Since each thread always prints a dash after printing a bar, and vice versa, this phenomenon can be caused only by one thread printing a bar and then the other thread printing a bar before the first one gets to print its dash.

   Describe a scenario involving the two threads where this happens.

2. Making method `print` synchronized should prevent this from happening. Explain why. Compile and run the improved program to see whether it works.
3. Rewrite `print` to use a `synchronized` statement in its body instead of the method being synchronized.
4. Make the `print` method static, and change the `synchronized` statement inside it to lock on the Print class's reflective Class object instead.

   For beauty, you should also change the threads to call static method `Print.print()` instead of instance method `p.print()`.

Exercise 1.1

Question 1

If the increment() method is not synchronized then you cannot be sure of what the final value of count will be. Two threads will be able to call the method simultaneously where one of the calls overwrites the other and thus causes thread interference.

Question 2

The lower the amount of increments the less is the risk of thread interference. The software will still be wrong as it will have the risk of performing an inaccurate number of increments of count.

Question 3

No, the different ways of incrementing the count results in the same error. The statements are all compound statements (check-modify-write sequences) that must execute atomatically to be thread-safe.

Question 4

The final count is expected to be 0. If not both methods are synchronized and share the same intrinsic lock of the LongCounter object, the operations could get interleved. Though the two threads call different methods, they still act on the same mutable shared state.

Question 4

I think the explanation for the incorrect final values of each of the four examples is given above. i) The methods will be executed simultaneously and thus will cause thread interferences. 15280, 20441, -3364 ii) If only one method is synchronized it is not possible to control the lock and thus the count value is not predictable. 4701, 38383, 25257 iii) Same as above. iv) The synchonization makes them share the object's intrinsic lock which ensures atomic executions.

Exercise 1.3

Question 1

1. t1.print()
2. t2.print()
3. t1 sleeps after printing '-'
4. t2 print '-' while t1 sleeps
5. t1 print '|' after catching an interruptedException
6. t2 print '|' after sleep or when catching an interruptedException This will print ||--

Question 2

Making print() synchronized prevents thread interference so the print method will be atomic. The intrinsic lock of the Printer object can only be held by one thread and thus only one thread can execute the method at a given time. Thus the thread will still have the lock while sleeping.

Exercises week 2
Mandatory handin 1
Friday 5 September 2014

Goal of the exercises

The goal of this week's exercises is to make sure that you have an initial understanding of using multiple threads for better performance, a good understanding of visibility of field updates between threads, and the advantages of immutability. You should be able to use locking (`synchronized`) and the `volatile` field modifier to ensure visibility between threads and use the `final` modifier to properly create and publish immutable objects.

Do this first

Get and unpack this week's example code in zip file pcpp-week02.zip on the course homepage.

Exercise 2.1 Consider the lecture's example in file TestMutableInteger.java, which contains this definition of class MutableInteger:

```
class MutableInteger {       // WARNING: USELESS IN THIS FORM
  private int value = 0;
  public void set(int value) {
    this.value = value;
  }
  public int get() {
    return value;
  }
}
```

As said in the Goetz book and the lecture, this cannot be used to reliably communicate an integer from one thread to another, as attempted here:

```
final MutableInteger mi = new MutableInteger();
Thread t = new Thread(new Runnable() { public void run() {
      while (mi.get() == 0) { }
      System.out.println("I completed, mi = " + mi.get());
  }});
t.start();
System.out.println("Press Enter to set mi to 42:");
System.in.read();                   // Wait for enter key
mi.set(42);
System.out.println("mi set to 42, waiting for thread ...");
try { t.join(); } catch (InterruptedException exn) { }
System.out.println("Thread t completed, and so does main");
```

1. Compile and run the example as is. Do you observe the same problem as in the lecture, where the `"main"` thread's write to `mi.value` remains invisible to the `t` thread, so that it loops forever?

2. Now declare both the `get` and `set` methods `synchronized`, compile and run. Does thread `t` terminate as expected now?

3. Now remove the `synchronized` modifier from the `get` methods. Does thread `t` terminate as expected now? If it does, is that something one should rely on? Why is `synchronized` needed on **both** methods for the reliable communication between the threads?

4. Remove both `synchronized` declarations and instead declare field `value` to be `volatile`. Does thread `t` terminate as expected now? Why should it be sufficient to use `volatile` and not `synchronized` in class MutableInteger?

Exercise 2.2 Consider the lecture's example in file TestCountPrimes.java.

1. Run the sequential version on your computer and measure its execution time. From a Linux or MacOS shell you can time it with `time java TestCountPrimes`; within Windows Powershell you can probably use `Measure-Command java TestCountPrimes`; from a Windows Command Prompt you probably need to use your wristwatch or your cellphone's timer.

2. Now run the 10-thread version and measure its execution time; is it faster or slower than the sequential version?

3. Try to remove the synchronization from the `increment()` method and run the 2-thread version. Does it still produce the correct result (664,579)?

4. In this particular use of LongCounter, does it matter in practice whether the `get` method is synchronized? Does it matter in theory? Why or why not?

Exercise 2.3 Consider the potentially computation-intensive problem of counting the number of prime number factors of an integer. This Java method from file TestCountFactors.java finds the number of prime factors of `p`:

```
public static int countFactors(int p) {
  if (p < 2)
    return 0;
  int factorCount = 1, k = 2;
  while (p >= k * k) {
    if (p % k == 0) {
      factorCount++;
      p /= k;
    } else
      k++;
  }
  return factorCount;
}
```

How this method works is not important, only that it may take some time to compute the number of prime factors. Actually the time is bounded by a function proportional to the square root of `p`, in other words $O(\sqrt{\mathtt{p}})$.

1. Write a sequential program to compute the total number of prime factors of the integers in range 0 to 4,999,999. The result should be 18,703,729. How much time does this take?

2. For use in the next subquestion you will need a MyAtomicInteger class that represents a thread-safe integer. It must have a method `int addAndGet(int amount)` that atomically adds `amount` to the integer and returns its new value, and a `int get()` method that returns the current value.

   Write such a MyAtomicInteger class.

3. Write a parallel program that uses 10 threads to count the total number of prime factors of the integers in range 0 to 4,999,999. Divide the work so that the first thread processes the numbers 0–499,999, the second thread processes the numbers 500,000–999,999, the third thread processes the numbers 1,000,000–1,499,999, and so on, using your MyAtomicInteger class. Do you still get the correct answer? How much time does this take?

4. Could one implement MyAtomicInteger without synchronization, just using a volatile field? Why or why not?

5. Solve the same problem but use the AtomicInteger class from the java.util.concurrent.atomic package instead of MyAtomicInteger. Is there any noticeable difference in speed or result? Should the AtomicInteger field be declared `final`?

Exercise 2.4 Consider the lecture's versions of Goetz's factorization examples in file TestFactorizer.java.

1. In the VolatileCachingFactorizer class, why is it important that the `cache` field is declared volatile?

2. In the OneValueCache class, why is it important that both fields are declared `final`?

Exercise 2.1

Question 1

Yes

Question 2

Yes

Question 3

No, and you shouldn't rely on it. The changes to the variable is not made visible to the other thread. By making both methods synchronized, we rely on the visibility guarantees given by locking the object.

Question 4

The thread still terminates as expected. By using a lock we get two guarantees: visibility and atomicity. Using the keyword `volatile` we are only gauranteed visibility. In this case we only need the visibility guarantee, so we can simply use `volatile` instead of locking.

Exercise 2.2

Question 1

```
Sequential result:     664579

real    0m6.968s
user    0m6.967s
sys     0m0.037s
```

Question 2

The 10 thread version executes faster:

```
Parallel10 result:     664579

real    0m1.906s
user    0m12.045s
sys     0m0.052s
```

If we look at the real execution time the code is now 3.7 times faster. But if we consider the time spent by all the threads in total, the execution time almost doubled.

Question 3

No, in this particular case we only got 663,733, thereby missing 846 primes:

```
Parallel2  result:     663733

real    0m4.467s
user    0m7.119s
sys     0m0.035s
```

When `increment` isn't synchronized we risk getting race conditions, as the incrementation isn't atomic anymore.

Question 4

In this particular case it doesn't matter. `get` is not called while more than one thread is running. All the incrementations are done before the call to `get`, so we don't risk getting race conditions here.

Exercise 2.3

Question 1

```
Total number of factors is  18703729

real    0m7.345s
user    0m7.340s
sys     0m0.041s
```

Question 3

Yes:

```
Total number of factors is  18703729

real    0m2.283s
user    0m14.700s
sys     0m0.104s
```

Question 4

No, we need `addAndGet` to be executed atomically.

Question 5

There is a slight increase in performance:

```
Total number of factors is  18703729

real    0m2.145s
user    0m13.886s
sys     0m0.062s
```

We do not need to declare the `AtomicInteger` `final`, since the class is already thread-safe, but it is good practice to do it, as it makes it easy to argue about.

Exercise 2.4

Question 1

It is important to make the `cache` variable volatile to ensure that all threads have the same object, i.e. the most current version of the cache. Leaving out the `volatile` keyword will not produce an incorrect answer in this case, but it will ruin the original intention of the cache.

Question 2

Both of the fields in `OneValueCache` needs to be final in order to make the object immutable. This is also ensured by not making a setter for the fields.

Exercises week 3
Friday 12 September 2014

Goal of the exercises

The goal of this week's exercises is to make sure that can build a threadsafe class in Java, make effective use of Java's concurrent collection classes in package `java.util.concurrent`, and use the future concept.

Do this first

Get and unpack this week's example code in zip file pcpp-week03.zip on the course homepage.

Exercise 3.1 A histogram is a collection of buckets, each of which is an integer count. The span of the histogram is the number of buckets. In the problems below a span of 30 will be sufficient; in that case the buckets are numbered 0...29.

Consider this Histogram interface for creating histograms:

```
interface Histogram {
  public void increment(int bucket);
  public int getCount(int bucket);
  public int getSpan();
}
```

Method call `increment(7)` will add one to bucket 7; method call `getCount(7)` will return the current count in bucket 7; method `getSpan()` will return the number of buckets.

There is a non-threadsafe implementation Histogram1 in file SimpleHistogram.java. You may assume that the `dump` method given there is called only when no other thread manipulates the histogram and therefore does not require locking, and that the span is fixed (immutable) for any given Histogram object.

1. Make a thread-safe implementation, class Histogram2, of interface Histogram by adding suitable modifiers (`final` and `synchronized`) to a copy of the Histogram1 class. Which fields and methods need which modifiers? Why? Does the `getSpan` method need to be synchronized?

2. Now consider again counting the number of prime factors in a number `p`, as in Exercise 2.3 and file TestCountFactors.java. Use the Histogram2 class to write a parallel program that counts how many numbers in the range 0...4 999 999 have 0 prime factors, how many have 1 prime factor, how many have 2 prime factors, and so on. You may draw inspiration from the TestCountPrimes.java example.

   The correct result should look like this:

   ```
     0:          2
     1:     348513
     2:     979274
     3:    1232881
     4:    1015979
     5:     660254
     6:     374791
     7:     197039
     8:      98949
     9:      48400
   ... and so on
   ```

   showing that 348 513 numbers in 0...4 999 999 have 1 prime factor (those are the prime numbers), 979 274 numbers have 2 prime factors, and so on. (The 2 numbers that have 0 prime factors are 0 and 1). And of course the numbers in the second column should add up to 5 000 000.

3. Define a thread-safe class Histogram3 that uses an array of java.util.concurrent.atomic.AtomicInteger objects instead of an array of integers to hold the counts.

   In principle this solution might perform better, because there is no need to lock the entire histogram object when two threads update distinct buckets. Only when two threads call `increment(7)` at the same time do they need to make sure the increments of bucket 7 are atomic.

   Can you now remove `synchronized` from all methods? Why? Run your prime factor counter and check that the results are correct.

4. Define a thread-safe class Histogram4 that uses a java.util.concurrent.atomic.AtomicIntegerArray object to hold the counts. Run your prime factor counter and check that the results are correct.

5. Now extend the Histogram interface with a method `getBuckets` that returns an array of the bucket counts:

   ```
   public int[] getBuckets();
   ```

   Show how you would implement this method for each of the classes Histogram2, Histogram3 and Histogram4 so that they remain thread-safe. Explain for each implementation whether it gives a fixed snapshot or a live view of the bucket counts, possibly affected by subsequent `increment` calls.

   Note in particular that for instance in the case of Histogram2 it would not be thread-safe to just return a reference to the internal array of integers, since a client who receives that reference could mess with the histogram's bucket counts without any synchronization.

6. (Optional). In Java 8 there is class java.util.concurrent.atomic.LongAdder that potentially offers even better scalability across multiple threads than AtomicInteger and AtomicLong; see the Java class library documentation. Create a Histogram5 class that uses an array of LongAdder objects for the buckets, and use it to solve the same problem as before.

Exercise 3.2 File TestCache.java contains a version of the prime factorization server example that implements the Computable interface and therefore can be wrapped in a memoizer, as developed in the lecture.

In this exercise you must write a program that creates and starts 16 threads numbered $t = 0 \ldots 15$, each of which computes the factors of 40 000 numbers, and such that their work partially overlaps (to demonstrate that the cache works):

- Every thread t must compute the factors of the 20 000 numbers from 10 000 000 000 to 10 000 019 999.
- Thread t must further compute the factors of the 20 000 numbers from $10\ 000\ 020\ 000 + t \cdot 5\ 000$ to $10\ 000\ 039\ 999 + t \cdot 5\ 000$.

In total the numbers in the range from 10 000 000 000 to $10\ 000\ 039\ 999 + 15 \cdot 5\ 000 = 10\ 000\ 114\ 999$ will be factorized, that is, 115 000 distinct numbers.

With a view to next week's (mandatory) exercises it is advisable to implement this scheme in terms of two parameters `start` and `range`:

```
final long start = 10_000_000_000L, range = 20_000L;
```

Then thread t considers the two ranges `from1...to1` and `from2...to2`, startpoint included and endpoint excluded, where `from1 = start`, `to1 = from1+range`, `from2 = start+range+t*range/4`, and `to2 = from2+range`.

1. Write a method `exerciseFactorizer` that takes as argument a thread-safe caching factorizer and calls it from 16 threads as specified above. The method outline may be something like this:

   ```
   private static void exerciseFactorizer(Computable<Long, long[]> f) {
     final int threadCount = 16;
     final long start = 10_000_000_000L, range = 20_000L;
     System.out.println(f.getClass());
     ...
   }
   ```

where the purpose of printing `f.getClass()` is just to show which of the cache classes is currently being used.

2. Wrap the given Factorizer in the Memoizer1 class and run the above program on this cached factorizer, then print the number of calls to the underlying Factorizer. You might use code such as this:

```
Factorizer f = new Factorizer();
exerciseFactorizer(new Memoizer0<Long,long[]>(f));
System.out.println(f.getCount());
```

   The number of calls to the factorizer should be 115 000. Is it?

   If your platform allows it, measure and note the execution time for this activity, using eg. `time java TestCache` on MacOS or Linux. In that case, note both the "real time" which is the wall-clock time, the "user time" which is the total CPU time spent by your code, and the "system time" which is the total CPU time spent in the operating system kernel.

3. Repeat this experiment with Memoizer2. How many times is the factorizer called? How long does the whole process take? Explain both results.

4. Repeat this experiment with Memoizer3. How many times is the factorizer called? How long does it take? Explain both results.

5. Repeat this experiment with Memoizer4. How many times is the factorizer called? How long does it take? Explain both results.

6. (Optional, requires Java 8) Repeat this experiment with Memoizer5. How many times is the factorizer called? How long does it take? Explain both results.

7. (Optional, requires Java 8) Write a caching class Memoizer0 that uses ConcurrentHashMap and its `computeIfAbsent` method to simply compute the given work `c.compute(arg)`, and using no FutureTasks or other fancy features. This can be done in 10 lines of code or less, and the correctness and thread-safety should be obvious. Repeat the above experiment with your Memoizer0. How many times is the factorizer called? How long does it take? Explain both results.

   Note: This is vastly simpler than Goetz's development, yet performs quite well in the present application even though it may violate the advice given in the Java 8 class library documentation for `computeIfAbsent`: "Some attempted update operations on this map by other threads may be blocked while computation is in progress, so the computation should be short and simple". Or maybe prime factorization *is* a "short and simple" computation, whereas for instance an HTTP request to a webserver would not be — such a request might block for many seconds. Hence in general the fancy Memoizer5 cache is probably still much preferable to the simpler Memoizer0.

Exercise 3.1

Question 1

- The `counts` field must be final to ensure that it will not get modified.
- The `increment` method must be synchronized to ensure atomicity.
- The `getCount` method must be synchronized to ensure that we do not read stale values.
- The `getSpan` method does not need to be synchronized as the `counts` field is final and thus cannot be modified ensuring that no stale lenght value is ever read.

Question 3

We can remove synchronized from the `increment` and `getCount` methods as the AtomicInteger class ensures that no threads will read stale values.

Question 5

Histogram2: We make the `getBuckets` method synchronized and thus retrieves a fixed snapshot of the histogram as no other operations can be done concurrently.

Histogram3: It is not possible to give a fixed snapshot without locking the operations in the `increment` method.

Histogram4: While copying the AtomicIntegerArray we use the lock of the AtomicIntegerArray to ensure that we return a fixed snapshot.

Exercises week 4
Mandatory handin 2
Friday 19 September 2014

Goal of the exercises

The goal of this week's exercises is to make sure that you can conduct meaningful performance measurements of Java programs, know that measurements can vary widely between apparently similar platforms, and can discuss observed strangenesses in timing results.

Do this first

The exercises build on the lecture note *Microbenchmarks in Java and C#* and the accompanying example code. Carefully study the hints and warnings in section 7 of that note before you measure anything.

Download and unpack the Java example code from file benchmarks-java.zip as indicated in the *Microbenchmarks* note, section 12.

Get and unpack this week's example code in zip file pcpp-week04.zip on the course homepage.

Exercise 4.1 In this exercise you must perform, on your own hardware, some of the single-threaded measurements done in *Microbenchmarks* note.

1. Run the `Mark1` through `Mark6` measurements yourself, and save results to text files. Use the `SystemInfo` method to record basic system identification, and supplement with whatever other information you can find about the execution platform. On Linux you may use `cat /proc/cpuinfo`; on MacOS you may use Apple > About this Mac; on Windows you may use Start > Control panel > System and security > System > View amount of RAM and processor speed.

   Include the results in your hand-in, and reflect and comment on them: Are they plausible? Any surprises? Mention any cases where they deviate significantly from those shown in *Microbenchmarks*.

2. Use `Mark7` to measure the execution time for the mathematical functions `pow`, `exp`, and so on, as in *Microbenchmarks* section 4.2. Record the results in a text file along with appropriate system identification. Preferably do this on at least two different platforms, eg. your own computer and a fellow student's, or some computer at the university.

   Include the results in your hand-in, and reflect and comment on them: Are they plausible? Any surprises? Mention any cases where they deviate significantly from those shown in *Microbenchmarks*.

Exercise 4.2 In this exercise you must perform, on your own hardware, the measurement performed in the lecture using the example code in file TestTimeThreads.java.

1. First compile and run the timing code as is, using `Mark6`, to get a feeling for the variation and robustness of the results. Do not hand in the results but discuss any strangenesses, such as large variation in the time measurements for each case.

2. Now change all the measurement to use `Mark7`, which reports only the final result. Record the results in a text file along with appropriate system identification.

   Include the results in your hand-in, and reflect and comment on them: Are they plausible? Any surprises? Mention any cases where they deviate significantly from those shown in the lecture.

Exercise 4.3 In this exercise you must use the benchmarking infrastructure to measure the performance of the prime counting example given in file TestCountPrimesThreads.java.

1. Measure the performance of the prime counting example on your own hardware, as a function of the number of threads used to determine whether a given number is a prime. Record system information as well as the measurement results for 1. . . 32 threads in a text file. If the measurements take excessively long time on your computer, you may measure just for 1. . . 16 threads instead.

2. Use Excel or gnuplot or Google Docs online or some other charting package to make graphs of the execution time as function of the number of threads.

3. Reflect and comment on the results; are they plausible? Is there any reasonable relation between the number of threads that gave best performance, and the number of cores in the computer you ran the benchmarks on? Any surprises?

4. Now instead of the LongCounter class, use the java.util.concurrent.atomic.AtomicLong class for the counts. Perform the measurements again as indicated above. Discuss the results: is the performance of AtomicLong better or worse than that of LongCounter? Should one in general use adequate built-in classes and methods when they exist?

5. Now change the worker thread code in the Runnable's `run()` method to work like a very performance-conscious developer might have written it. Instead of calling `lc.increment()` on a shared thread-safe variable `lc` from all the threads, create a local variable `long count = 0` inside the `run()` method, and increment that variable in the for-loop. This local variable is thread-confined and needs no synchronization. After the for-loop, add the local variable's value to a shared AtomicLong, and at the end of the `countParallelN` method return the value of the AtomicLong.

   This reduces the number of synchronizations from several hundred thousands to at most `threadCount`, which is at most 32. In theory this might make the code faster. Measure whether this is the case on your hardware. Is it? (It is not faster on my Intel-based MacOS laptop).

   (Optional) Can you think of any possible explanations for the few-synchronizations code not being faster than the original many-synchronizations code?

Exercise 4.4 Consider again the cache implementations in TestCache.java from week 3, applied to the prime factorization problem in exercise 3.2. Use the same thread count 16 and threads numbered $t = 0 \ldots 15$ as in that exercise. To make the measurements faster, you may reduce the amount of work that each thread must perform, so that thread t computes the factors of 4 000 numbers, namely the factors of the 2 000 numbers from 10 000 000 000 to 10 000 001 999 and also of the 2 000 numbers from 10 000 002 000 $+ t \cdot 500$ to 10 000 003 999 $+ t \cdot 500$.

1. Use the `Mark7` function to measure and report the execution time when wrapping the Factorizer class as a Memoizer1 instance.

2. Similarly, measure and report the execution time when wrapping the Factorizer class as a Memoizer2 instance.

3. Similarly, measure and report the execution time when wrapping the Factorizer class as a Memoizer3 instance.

4. Similarly, measure and report the execution time when wrapping the Factorizer class as a Memoizer4 instance.

5. (Optional) Similarly, measure and report the execution time when wrapping the Factorizer class as a Memoizer5 instance.

6. (Optional) Similarly, measure and report the execution time when wrapping the Factorizer class as a Memoizer0 instance.

7. Reflect on the results of the measurements you made. Which cache implementation performs best in this particular application: factorization of relatively large numbers, 16 threads working on partially overlapping ranges of such numbers? Does this result agree with the lecture's and Goetz's development of the cache classes?

8. What experiment would you set up to compare the scalability of the different cache implementations? You do not have to actually make that experiment, just describe it.

Exercise 4.1

Question 1

The results of the measurements of the mark* methods are placed in the 'benchmarks' folder together with the system info.

Our benchmarks are almost identical to the ones made by Peter, so we conclude that they are very plausible.

In the Ex4_1_6 benchmarks we experienced a large standard deviation which might be because of concurrent programs running on the computer and maybe because of the garbage collector as it is similar to Peter's results where it does the same.

Question 2

Our results are stored in the benchmarks folder. Our benchmarks of mark7 shows that the stronger computer (Ex4_1_2_1) is faster than (Ex4_1_2_2) as expected. It is a little strange that the deviation of Ex4_1_2_2 is consistently larger than the other. This could be because the computer running Ex4_1_2_2 having more programs running and thus each of the benchmark runs doesn't have the same basis.

Exercise 4.2

Question 1

We think that there are quite a few outbursts in standard deviation during the different benchmarking runs. It is definitely not a steady move towards a robust result.

Question 2

The means of our benchmarking results are almost the same as in the lecture notes with just a very small constant factor larger (~10% slower). However, the standard deviations on the last results are very large compared to those of the lectures notes.

Exercise 4.3

Question 1

Question 2

The visual graphs of the result is saved as Ex4_3_2.png.

Question 3

Our benchmarking results shows that the 8-thread benchmark is the fastest. This makes sense as the computer used have 4 cores with hyperthreading, thus, 8 processors.

Question 4

There is no significant difference in the performance between AtomicLong and LongCounter - if anything, AtmoicLong is a little bit faster. When adequate built-in classes are available one should use them. They are most likely optimized and thoroughly tested.

The graph of the implementation using Atomic long is saved in Ex4_3_4.png

Question 5

We experienced that our modified version was slower than the original version.

We cannot argument for this strange behaviour.

Exercise 4.4

Question 1-6

```
Memoizer1                    1871977212,6 ns     29854323,13          2
Memoizer2                    1405574646,1 ns     52687573,52          2
Memoizer3                     999874912,2 ns     22564365,87          2
Memoizer4                     993271221,6 ns     26108494,08          2
Memoizer5                     977745474,8 ns     32062326,46          2
Memoizer                      986815814,1 ns     24626683,76          2
```

Question 7

We expected the first memoizer to be the slowest, as it blocks on the factorizer call. Though the second one doesn't block during the computation, it has a large risk of doing the same computation more than once, thereby spending time on unneccessary computations. The third version avoids the duplicate computations, but might create unnecessary `FutureTask` objects, which, as the second version, spends time on an unneccessary computation (the object construction), though the computation here is faster. The fourth and fifth version avoids these extra object constructions completely and blocks minimally.

As expected we see a larger improvement from the first cache version to the second, and again from the second to the third. The last three versions are almost identically, as the only thing removed, is the extra construction of small unneccessary objects.

The benchmarks correspond well with our expectations and with the literature.

Exercises week 5
Friday 26 September 2014

Goal of the exercises

The goal of this week's exercises is to make sure that you can use tasks and the Java executor framework; and that you can modify and extend a simple processing pipeline using a given blocking queue implementation as well as one from the Java class library. Note that all of Exercise 5.2 may be skipped; the other exercises are more interesting.

Do this first

Get and unpack this week's example code in zip file pcpp-week05.zip on the course homepage.

Exercise 5.1 Use the lecture's example in file TestCountPrimesTasks.java to count prime numbers using tasks and the executor framework instead of explicitly creating threads for each concurrent activity.

1. Using a CachedThreadPool as executor, measure the time consumption as a function of the number of tasks used to determine whether a given number is a prime. The Mark7 method is relevant.
2. (Optional) If you have Java 8, use a WorkStealingPool and repeat the experiments.
3. Use Excel or gnuplot or Google Docs online or some other charting package to make graphs of the execution time as function of the number of tasks. Do this for the executors you have tried.
4. Reflect and comment on the results; are they plausible? How do they compare with the performance results obtained using explicit threads in last week's exercise? Is there any reasonable relation between the number of threads and the number of cores in the computer you ran the benchmarks on? Any surprises?
5. (Optional) Java 8 has a class java.util.concurrent.atomic.LongAdder that is a kind of AtomicLong. It may perform much better than LongCounter and AtomicLong when many threads add to it often and its actual value is inspected only rarely. Also, it should perform well in case locking is especially slow on the hardware being used. Replace the use of LongCounter with LongAdder, and rerun the measurements.

Exercise 5.2 **(Optional)** Use the week 4 benchmarking techniques to measure the time to create, start, and finish a simple task. You may draw inspiration from TestTimeThreads.java used in Exercise 4.2.

For your experiments in questions 5.2.3 and 5.2.4, use a CachedThreadPool, or if you have Java 8, a WorkStealingPool. Using a FixedThreadPool would create many threads up front and therefore allocate a huge amount of memory, and that would distort subsequent measurements.

NB: An Executor and the associated thread pool is a complicated and memory-consuming object, so *create only one Executor to run the measurements in each of those two exercise questions.* If you create a new Executor inside the IntToDouble instance then you will just measure the time spent creating and managing the Executors, and most likely your Java process will run out of memory.

To create a separate executor for each of the exercise questions, use a code pattern like this:

```
{
  ExecutorService executor = Executors.newWorkStealingPool();
  Mark6("Task create submit cancel",
        ...
  );
  executor.shutdownNow();
}
```

This will also shut down the executor after performing the experiments.

1. Measure the time to run a for-loop that increments an AtomicInteger 1000 times.
2. Measure the time to create (but not submit) a Runnable task that increments an AtomicInteger 1000 times.

3. Measure the time to create and submit a Runnable task (that increments an AtomicInteger 1000 times) to get a Future, and then immediately cancel the Future. The cancellation is necessary, otherwise all the tasks submitted by benchmarking will accumulate on the executor, causing very large memory usage.

4. Measure the time to create and submit a Runnable task (that increments an AtomicInteger 1000 times) to get a Future, and then immediately wait for the Future to run by calling `get()` on it.

5. Reflect and comment on the results; are they plausible? How do they compare with the cost of creating, starting and completing threads in last week's Exercise 4.2? Any surprises?

Exercise 5.3 This exercise is about fetching a bunch of web pages. This activity is heavy on input-output and the latency (delay) involved in requests and responses sent over a network. In contrast to the previous exercises, fetching a webpage does not involve much computation; it is an input-output bound activity rather than a CPU-bound activity.

File TestDownload.java contains a declaration of a method `getPage(url, maxLines)` that fetches at most `maxLines` lines of the body of the webpage at `url`, or throws an exception.

1. First, run that code to fetch and print the first 10 lines of `www.wikipedia.org` to see that the code and net connection works.

2. Now write a sequential method `getPages(urls, maxLines)` that given an array `urls` of webpage URLs fetches up to `maxLines` lines of each of those webpages, and returns the result as a map from URL to the text retrieved. Concretely, the result may be an immutable collection of type Map<String,String>. You should use neither tasks nor threads at this point.

   File TestDownload.java contains such a list of URLs; you may remove unresponsive ones and add any others you fancy.

   Call `getPages` on a list of URLs to get at most 200 lines from each, and for each result print the URL and the number of characters in the corresponding body text — the latter is a sanity check to see that something was fetched at all.

3. Use the Timer class — **not** the Mark6 or Mark7 methods — for simple wall-clock measurement (described in the *Microbenchmarks* lecture note from week 4) to measure and print the time it takes to fetch these URLs sequentially. Do not include the time it takes to print the length of the webpages.

   Perform this measurement five times and report the numbers. Expect the times to vary a lot due to fluctuations in network traffic, webserver loads, and many other factors. In particular, the very first run may be slow because the DNS nameserver needs to map names to IP numbers.

   (In principle, you could use Mark6 or Mark7 from the *Microbenchmarks* note to more accurately measure the time to load webpages, but this is probably a bad idea. It would run the same web requests many times, and this might be regarded as a denial-of-service attack by the websites, which could then block requests from your network).

4. Now create a new parallel version `getPagesParallel` of the `getPages` method that creates a separate task (not thread) for each page to fetch. It should submit the tasks to an executor and then wait for all of them to complete, then return the result as a Map<String,String> as before.

   The advantage of this approach is that many webpage fetches can proceed in parallel, even on a single-core computer, because the webservers out there work in parallel. In the old sequential version, the first request will have to fully complete before the second request is even initiated, and so on; meanwhile the CPU and the network sits mostly idle, wasting much time.

   As executor, use a WorkStealingPool if you have Java 8, otherwise a CachedThreadPool. Make sure that the executor is allocated only once, for instance as a static field in the class; do not allocate a fresh executor for each call to `getPagesParallel`.

   Call it as in the previous question, measure the wall-clock time it takes to complete, and print that and the list of page lengths. Repeat the measurement five times and compare the results with the sequential version. Discuss results, in particular, why is fetching 23 webpages in parallel not 23 times faster than fetching them one by one?

Exercise 5.4 Consider the pipeline built in the lecture's file TestPipeline.java to gather and print webpage links.

1. Run it as is, to see that it works on your machine.

   (As you will notice, you need to manually terminate the program by pressing control-C or similar when it has not printed any new results for a while. This is because the "consumers" PageGetter, LinkScanner and LinkPrinter wait forever for more input arriving through their input queue. Fixing this deficiency in a general way seems to complicate the example considerably, so we will live with it).

2. If a webpage contains several occurrences of the same link to another webpage then LinkScanner will "produce" that link multiple times, and LinkPrinter will therefore print it multiple times; this is not desirable. Instead of changing LinkScanner or LinkPrinter, write a new class Uniquifier<T> that consumes items of type T and produces items of type T, but only once each. The class should implement Runnable and its constructor should take as argument an input queue and an output queue, each of type Queue<T>.

   The uniquifier's `run` method may maintain a HashSet<T> containing the items already seen. When an item received from the input queue is already in the hashset, it is ignored; otherwise it is added to the hashset and also sent to the output queue.

   Insert a Uniquifier<Link> stage, and suitable queues, between the LinkScanner and the LinkPrinter, so that a link `(from,to)` from one webpage to another is printed only once.

3. Change the implementation to submit the UrlProducer, PageGetter, LinkScanner, Uniquifier and LinkPrinter as tasks on an executor service rather create threads from them. If you have Java 8, use a WorkStealingPool, otherwise a CachedThreadPool, as executor service.

   The results should be the same as when using threads. Are they?

4. Now use a FixedThreadPool of size 6 to run the tasks. This should work as before.

5. Now use a FixedThreadPool of size 3 to run the tasks. This probably does not work. What so you observe? Can you explain why?

6. Probably the pipeline's most time-consuming stage is the PageGetter. To improve overall throughput, one may simply create two PageGetter objects (as threads or tasks), both taking input from the BlockingQueue<String> queue and sending output to the BlockingQueue<Webpage> queue.

   Implement this idea. You should get the same results as before, though possibly in a different order. Do you? Why?

7. (Optional) Implement your own bounded blocking queue class BoundedQueue<T>. It must implement our BlockingQueue<T> interface and be able to hold n elements, where n is given as a parameter when the BoundedQueue<T> object is created.

   The queue may use an ArrayList<T> as a cyclic buffer to hold the items. The queue constructor should create it with capacity n and add `null` to it n times; all subsequent use should be through indexing `items.set(i, x)` and `items.get(j)` where `i` and `j` are suitably computed indexes. (It would be more natural to store the items in a standard array `T[]` of size n but due to Java's weak generic types, this causes problems that have nothing to do with concurrency).

   As before, the `put` method should block when the buffer is full, and the `take` method should block when the buffer is empty.

   Explain why your bounded blocking queue works and why it is thread-safe.

Exercise 5.1

Question 1

```
# OS:   Mac OS X; 10.9.5; x86_64
# JVM:  Oracle Corporation; 1.8.0_20
# CPU:  2,3 GHz Intel Core i7
# RAM:  16 GB 1600 MHz DDR3
# Date: 2014-09-26T10:20:13+0200
countSequential                   12682,3 us     574,48         32
9592.0
countParTask1     32               5685,7 us     733,98         64
9592.0
countParTask2     32               5015,9 us     239,13         64
9592.0
countParTask3     32               4339,1 us     247,44         64
9592.0
```

Question 2

```
# OS:   Mac OS X; 10.9.5; x86_64
# JVM:  Oracle Corporation; 1.8.0_20
# CPU:  2,3 GHz Intel Core i7
# RAM:  16 GB 1600 MHz DDR3
# Date: 2014-09-26T10:18:00+0200
countSequential                   12536,9 us     797,52         32
9592.0
countParTask1     32               3887,8 us      89,64        128
9592.0
countParTask2     32               4243,8 us     293,69         64
9592.0
countParTask3     32               4520,2 us     164,92         64
9592.0
```

Question 3

Interactive graphs:

- CachedThreadPool - countParTask1
- CachedThreadPool - countParTask2
- WorkStealingPool - countParTask1
- WorkStealingPool - countParTask2

(Static images and data points can be found in the `benchmarks` folder)

Question 4

The execution times are almost the same on our machine. It seems the thread version runs the fastest with a count up to 8 (the machine has 8 cores). After 8 it seems like the numbers are almost the same. The small variations are too small to give a proper answer. Comparison

Question 5

Exercise 5.2

Question 1

Question 2

Question 3

Question 4

Question 5

Exercise 5.3

Question 1

Runs: 1. 12.933446358 2. 11.08861516 3. 8.806998804 4. 11.117534792 5. 7.94627111

Question 4

A parallel run of tasks is only as past as the slowest executed task. This means if 22 of the urls take 1 second to execute and the last takes 3 seconds, then the total execution time will be 3 seconds.

Runs: 1. 2.886404326 2. 1.68489287 3. 1.764366912 4. 1.797240872 5. 1.731433001

Exercise 5.4

Question 1

Question 2

Question 3

We did not manage to get this to work..

Question 4

Question 5

PCPP **IT University, E2014**

Exercises week 6
Mandatory handin 3
Friday 3 October 2014

Goal of the exercises

The goal of this week's exercises is to make sure that you can write deadlock-free synchronization code, diagnose deadlocks using the `jvisualvm` tool, and check `@GuardedBy` annotations with the ThreadSafe tool. Note that Exercise 6.3 is optional and need not be answered.

Do this first

Get and unpack this week's example code in zip file pcpp-week06.zip on the course homepage. Also download the ThreadSafe tool, either

- Eclipse plugin, from http://download.contemplateltd.com/threadsafe/threadsafe-eclipse-1.3.3.zip, or
- Command line interface, from http://download.contemplateltd.com/threadsafe/threadsafe-cli-1.3.3.zip

Unpack and install as indicated in the guide at http://www.contemplateltd.com/threadsafe-solo-quick-start.

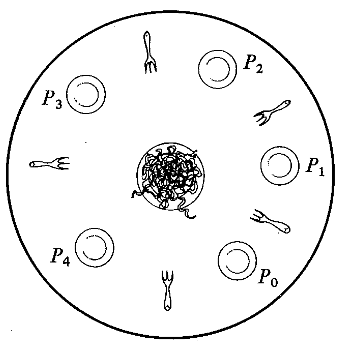
ThreadSafe is commercial software and you must replace the file `threadsafe.properties` with the one containing PCPP's license key; you find that in LearnIT under week 6. Do not share the license key with people outside the IT University.

Exercise 6.1 In this exercise you must experiment with and modify run the lecture's accounts transfer example.

1. Run TestAccountDeadlock.java on your computer. Does it deadlock? If not, how could that be?
2. Modify TestAccountLockOrder.java to use the `transferE` and `balanceSumE` methods, both of which use hashcodes to determine locking order. As said in the lecture and the code comments, this may still deadlock in the rare case two distinct Accounts get the same hashcode. Run it a couple of times on your computer. Does it actually deadlock? (Probably not).
3. Now make `transferE` and `balanceSumE` guaranteed deadlock-free by implementing the Goetz idea (section 10.1.2, code on page 209) that deals with identical hashcodes by taking a third lock that is used only for this purpose. Compile and run it. Does it still work and not deadlock?
4. Would it be safe and deadlock-free to just ignore the hashcodes and *always* use the last `else`-branch in the Goetz page 209 code, taking all three locks whenever a transfer is made? Discuss. What is the reason for not just doing that?

Exercise 6.2 In this exercise you should use the `jvisualvm` tool (distributed with the Java Software Development Kit) to investigate the famous Dining Philosopher's problem, due to E. W. Dijkstra.

Five philosophers (threads) sit at a round table on which there are five forks (shared resources), placed between the philosophers. A philosopher alternatingly thinks and eats spaghetti. To eat, the philosopher needs exclusive use of the two forks placed to his left and right, so he tries to lock them.
Both the places and the forks are numbered 0 to 5. The fork to the left of place `p` has number `p`, and the fork to the right has number `(p+1)%5`.
(Drawing from Ben-Ari: *Principles of Concurrent Programming*, 1982).



1. Consider the Dining Philosophers program in file TestPhilosophers.java. Explain why it may deadlock.
2. Compile the program, and run it until it deadlocks. Do this a few times. Does the time to deadlock vary much?
3. Again, run the program till it deadlocks and leave it there. Start `jvisualvm`, attach it to the TestPhilosophers Java process, and find what it says about the reason for the deadlock. Copy the relevant message to your answer and explain in your own words what it says.
4. Rewrite the philosopher program to avoid deadlock. The solution (as in the lecture) is to impose an ordering on the locks (forks) and then every philosopher (thread) should take the locks in that order. For instance, when a philosopher needs to take locks numbered `i` and `j`, always take the lowest-numbered one first. Implement this small change, and run the program for as long as you care. It should not deadlock.
5. Rewrite the philosopher program to use ReentrantLock on the five forks, and so that every philosopher first attempts to pick up the left fork, then the right one, leaving both on the table if any one of them is in use. You can simply make class Fork a subclass of java.util.concurrent.locks.ReentrantLock and call `tryLock()` on the Fork, as in the lecture's TestAccountTryLock.java example. Try to run the program. Does *any* philosopher get to eat at all?
6. Now is there any *fairness*, that is, at every point does a philosopher who is trying to eat eventually get to do it (also expressed as, does every philosopher get to eat infinitely often)? Use an array of threadsafe counters, for instance AtomicIntegers, to count how many times each philosopher has eaten, and make a for-loop on the "main" thread that prints these numbers every 10,000 milliseconds. What do you observe?

   It is quite possible that the philosophers (threads) get to eat roughly equally often, but this is by now means guaranteed, and the correct functioning of a program should not depend on the thread scheduler's fairness. It may vary between Java versions and operating systems, and be different on Sunday than Monday.

Exercise 6.3 (**Optional**) In this exercise you must apply the ThreadSafe tool to week 3's FirstBadListHelper and SecondBadListHelper classes in file TestListHelper.java.

1. First run ThreadSafe on the example in file ts/guardedby/TestGuardedBy.java.
2. Run ThreadSafe on week 3's FirstBadListHelper and SecondBadListHelper classes. Which of the thread-safety problems does ThreadSafe discover, and which ones does it overlook? Show the messages from ThreadSafe, explain them in your own words, and say whether you agree with them.
3. Now, add a `@GuardedBy("list")` annotation SecondBadListHelper's list field. Does this help? What does ThreadSafe say, and do you agree with the message? Explain.

Exercise 6.4 In this exercise you must apply the ThreadSafe tool to your threadsafe `int[]`-based Histogram2 class from week 3.

1. Add relevant `@GuardedBy` annotations to your threadsafe Histogram2 class from week 3. Compile it. Then use ThreadSafe to check it. Does it pass? Now delete synchronized from one of the public `increment` and `getCount` methods. What does ThreadSafe say? Does ThreadSafe expect the `getSpan()` method to be synchronized?
2. Add a new method `void addAll(Histogram hist)` to the Histogram interface and Histogram2 class. The new method should throw a RuntimeException if this histogram and `hist` have different spans. Otherwise it should add the counts of `hist` to this histogram. What does the method need to lock on? Explain. (Hint: There should be no risk of deadlocks in this question). Compile and run ThreadSafe on the code. Does it agree with you?
3. Now pretend that you are Mort Madcap, who cannot be bothered with interfaces, encapsulation, and other object-oriented dogmas. So despite what the exercise says, he has implemented `addAll` so that (1) it takes an argument of type Histogram2, not Histogram, and (2) it accesses the `hist.counts` array directly instead of using the `getCount` method. Does ThreadSafe help Mort spot any errors he may have made? Explain in your own words what ThreadSafe tries to say, and whether there is any reason to worry.

Exercise 6.1

Question 1

`TestAccountDeadlock` run into a deadlock as `clerk1` and `clerk2` are running simultaneously locking account 1 and 2 respectively while waiting for the lock of the other account to be released. Thus, none of them are able to acquire the other lock making them both wait indefinitely.

Question 2

No, the code did not enter a deadlock. It is possible for the code to enter a deadlock as two objects could have the same identity hashcode even though they are not the same object. If we call the method with two identity hashcode equal objects, they will enter the same branch in the if sentence, but due to ordering, they might not lock on the same object first, introducing a deadlock.

If waiting long enough, the code could enter a deadlock. We tried making a loop that ran until two identical hashcodes appeared, but we did not actually manage to hit a hash collision.

Question 3

We used the Goetz idea by adding a special case for identity hashcode collisions. In case we get a collision, we acquire a special shared tie lock, which must be acquired before any locking on the account objects happen. In the case that we try to transfer in both directions between two identity hashcode equal objects, we will only be able to acquire the account locks, if we have the shared tie lock, thereby preventing a deadlock.

If we should happen to call the same method with two other identity hashcode equal objects, they must wait for the same shared tie lock used by the others. This is not really a problem, as the identity hashcode collisions should happen rarely.

The code works and it still does not enter a deadlock.

Question 4

It would be safe and deadlock-free to ignore the hashcodes and using a tie lock instead, but it would not be scalable as all transfers must acquire the tie lock even if they do not transfer between the same accounts.

This would become a concurrency bottleneck.

Exercise 6.2

Question 1

The program might deadlock if each philosopher grabs the fork to their left simultaneously and then try to grab the fork to their right. All the forks will be taken leaving all philosophers waiting for a fork to free up, thereby introducing a deadlock.

Question 2

The program enters a deadlock after running various lengths of time.

The program enters a deadlock only if all the philosophers have one fork at the same time.

The fewer philosophers you have it is higher risk of each philosopher having one fork each. It seems like the more cores you have, the easier it is to enter a deadlock.

Question 3

The following output was generated by a thread dump:

```
Found one Java-level deadlock:
=============================
"Thread-3":
  waiting to lock monitor 0x00007fc8ac0b9758 (object 0x00000007957bc330, a Fork),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x00007fc8ac06b958 (object 0x00000007957bc340, a Fork),
  which is held by "Thread-1"
"Thread-1":
  waiting to lock monitor 0x00007fc8ac06b8a8 (object 0x00000007957bc350, a Fork),
  which is held by "Thread-2"
"Thread-2":
  waiting to lock monitor 0x00007fc8ac0b9808 (object 0x00000007957bc360, a Fork),
  which is held by "Thread-3"
```

This shows that each thread holds a lock to a fork and are waiting for another the right hand fork lock to be released.

Question 4

This version of the program does not enter a deadlock as each philosopher will always take the fork with the lowest index first. Thus, the last philosopher will first try to acquire the lock for fork 0 and then it will try to acquire the lock for its own fork. In this way all the philiosophers will never wait for a fork lock to be released and it will never enter a deadlock.

Question 5

Yes, it does not enter a deadlock at any time. This will never happen as one philisopher will never wait for another lock.

A possible problem with the program is that livelock can occur. This occurs when all philosophers repeatedly tries to take the locks without succeeding, thereby releasing their locks starting over the attempt of eating. This is not very likely and if it does occur, the risk of multiple failed attempts in a row is low as we use sleep with a random time.

Question 6

There is nothing to ensure that all of them each eats the same number of times. A philosopher could grab the same fork two times in a row.

Exercise 6.3

Question 1

Question 2

Question 3

Exercise 6.4

Question 1

Yes, it passes initially when everything is synchronized.

After removing synchronized on `increment`, ThreadSafe reports "unsynchronized read/write" on the method.

The `getSpan` method is not reported by ThreadSafe as it is accessing a final value.

Question 2

The `addAll` method needs to lock while adding the buckets to the histogram as it needs the increment (+=) operations to be atomic.

ThreadSafe agrees as it does not report anything. When removing the lock on this it reports that we have unsynchronized read and writes during the increment as expected.

Question 3

In the `madAll` we only synchronize on `this`. ThreadSafe reports that the access to `that.counts` is only sometimes synchronized, as we do not hold the lock on `that` while reading. Since we do not write back to `that`, it should not be a problem. The only drawback here, will be that the reading of the whole histogram will not be atomic.

Exercises week 7
Friday 10 October 2014

Goal of the exercises

The goal of this week's exercises is to make sure that you can achieve good performance and scalability of lock-based concurrent software, using finer-grained locks, lock striping, the Java class library's atomically updatable numbers, immutability and the visibility effects of volatiles and atomics.

Due to the fall break, the handin deadline for these exercises is Thursday 23 October 2014.

Do this first

Get and unpack this week's example code in zip file pcpp-week07.zip on the course homepage.

File TestStripedMap.java contains implementations of several thread-safe hash map classes:

(A) A complete implementation of SynchronizedMap<K,V> which follows the Java monitor pattern: all mutable fields are private, all public methods are synchronized, and no internal data structures escape.

(B) A partial implementation of StripedMap<K,V> which does not follow the Java monitor pattern, but divides the buckets table into stripes, locking each stripe both on read and write accesses. This is the subject of Exercise 7.1.

(C) A partial implementation of StripedWriteMap<K,V> which also divides the buckets table into stripes, but locks each stripe only on write accesses. Read accesses do not take locks at all, but their correctness is assured — we hope — by (1) working on immutable item nodes, and (2) ensuring visibility of writes by careful use of atomics and volatiles. This is the subject of Exercise 7.2.

(D) A simple wrapper WrapConcurrentHashMap<K,V> around Java's ConcurrentHashMap<K,V>, for comparison.

Exercise 7.1 The SynchronizedMap<K,V> implementation scales (and therefore performs) poorly on a multicore computer because of all the locking: only one thread at a time can read or write the hash map.

The lecture showed that scalability can be considerably improved by *lock striping*. Instead of locking on the entire table of buckets, one divides it into a number of stripes (here 32), and locks only the single stripe that is going to be read or updated.

This is the idea in the StripedMap<K,V> class, whose implementation in file TestStripedMap.java contain only methods `containsKey` and `put` and some auxiliary methods.

Your task below is to implement the remaining public methods, as described by interface OurMap<K,V>. They are very similar to the method implementations in class SynchronizedMap<K,V>, except that they do not lock the entire hash map. only the relevant stripe.

1. Implement method `V get(K k)` using lock striping. It is similar to `containsKey`, but returns the value associated with key `k` if it is in the map, otherwise `null`. It should use the `ItemNode.search` auxiliary method.

2. Implement method `int size()` using lock striping; it should return the total number of entries in the hash map. The size of stripe `s` is maintained in `sizes[s]`, so the `size()` method should simply compute the sum of these values, locking each stripe in turn before accessing its value.

   Explain why it is important to lock stripe `s` when reading its size from `sizes[s]`?

3. Implement method `V putIfAbsent(K k, V v)` using lock striping. It is very similar to `putIfAbsent` in class SynchronizedMap<K,V> but should of course only lock on the stripe that will hold key `k`. It should use the `ItemNode.search` auxiliary method. Remember to increment the relevant `sizes[stripe]` count if any entry was added.

4. Implement method `V remove(K k)` using lock striping. Again very similar to SynchronizedMap<K,V>. Remember to decrement the relevant `sizes[stripe]` count if any entry was removed.

5. Implement method void forEach(Consumer<K,V> consumer). This may be implemented in two ways: either (1) iterate through the buckets as in the SynchronizedMap<K,V> implementation, locking the corresponding stripe before accessing the bucket; or (2) iterate over the stripes, and for each stripe iterate over the buckets that belong to that stripe. The latter takes each stripe lock only once, instead of potentially thousands of time. Explain your implementation.

   In both cases, since forEach reads the volatile buckets field several times but locks only stripe-wise, it must first obtain a reference theBuckets and then use that in the rest of the method, like this:

   ```
   public void forEach1(Consumer<K,V> consumer) {
     final ItemNode<K,V>[] bs = buckets;
     for (int hash=0; hash<bs.length; hash++) {
       synchronized (locks[hash % lockCount]) {
         ItemNode<K,V> node = bs[hash];
         ...
       }
   }
   ```

   Otherwise another thread may call reallocateBuckets and hence replace the buckets array with one of a different size between observing the length of the array and accessing its elements.

6. You may use method testMap(map) for very basic single-threaded functional testing while making the above method implementations. See how to call it in method testAllMaps. To actually enable the assert statements, run with the -ea option:

   ```
   java -ea TestStripedMap
   ```

7. Measure the performance of SynchronizedMap<K,V> and StripedMap<K,V> by timing calls to method exerciseMap. Report the results from your hardware and discuss whether they are as expected.

8. What advantages are there to using a small number (say 32 or 16) of stripes instead of simply having a stripe for each entry in the buckets table? Discuss.

9. Why can using 32 stripes improve performance even if one never runs more than, say, 16 threads? Discuss.

10. (Subtle, but answered in the source code) Why is it important for thread-safety that the number of buckets is a multiple of the number of stripes?

Note that method reallocateBuckets has been provided for you. Its auxiliary method lockAllAndThen uses recursion to take all the stripe lock; this is the only way in Java to take a variable number of intrinsic locks.

Exercise 7.2 The striped hash map in class StripedMap<K,V> scales better with more threads than the SynchronizedMap<K,V> hash map. However, it can be further improved by locking a stripe only when writing, not when reading, so that many reads can proceed concurrently without locking. This idea is outlined in class StripedWriteMap<K,V>, which is a somewhat subtle undertaking, based on several ideas that are different from both SynchronizedMap<K,V> and StripedMap<K,V>.

First, the item nodes are made immutable, all fields of class ItemNode<K,V> are final. That means that as soon as a read access (containsKey, get or forEach) has obtained a reference to a list of item nodes in a bucket, it need not be concerned with atomicity or visibility: nothing it accesses can be affected by other threads.

Second, the slice sizes will now be represented by an AtomicIntegerArray so that no locking is needed when incrementing and decrementing the stripe sizes. It also ensures that a thread executing the size() method can see the increments and decrements made by threads that put, putIfAbsent and remove entries.

Third, the writes to and reads from the sizes array are (ab)used to ensure visibility of updates to the buckets array. After any write to an element of buckets, sizes is written also, and before any read of an element of buckets, sizes is read. This ensures that containsKey, get, forEach will see any writes performed by put, putIfAbsent and remove.

Fourth, making class ItemNode<K,V> immutable means that put and remove may need to copy part of the list of item nodes in a bucket, but those lists should in any case hold at most a few items (otherwise the hash map is slow), the allocation of a new item node is fast, the cost appears to be outweighed but the time saved on not locking, and parts of the code become much neater this way.

Some ideas in StripedWriteMap<K,V> are inspired by the implementation of Java's ConcurrentHashMap, which however uses many more sophisticated techniques.

1. Implement method `int size()`. This is very straightforward: simply compute the sum of the stripe sizes. Since these are represented in an AtomicIntegerArray, all writes are visible to this method's reads; no locking is needed.

2. Implement method `V putIfAbsent(K k, V v)`. You must lock on the relevant stripe. Use auxiliary method `ItemNode.search(bl, k, old)` to determine whether `k` is already in the hash map, where `bl` is the bucket list reference obtained from `buckets[hash]`. If yes, then do nothing; else create a new item node from `k`, `v` and `bl`, and update the `buckets` table with that. Remember to update the stripe size if an entry was added.

   Why do you not need to write to the stripe size if nothing was added?

3. Implement method `V remove(K k)`. Lock on the relevant stripe. Use `ItemNode.delete(bl, k, old)` to delete the entry with key `k`, if any, from bucket list `bl`, and update the buckets table with the result. Remember to update the stripe size if an entry was removed.

4. Implement method `void forEach(Consumer<K,V> consumer)`. Same comments apply as Exercise 7.1.5, but additionally you must read the stripe's size before iterating over its buckets, for visibility of writes.

5. Measure the performance of SynchronizedMap<K,V>, StripedMap<K,V> StripedWriteMap<K,V> and WrapConcurrentHashMap<K,V> using method `exerciseAllMaps`. Report the results and discuss whether they are as expected.

6. (Optional, only really interesting if you have access to a computer with many cores) Measure the scalability of the four hash map implementations by running method `timeAllMaps`. Report the results, in tabular or graphical form, and discuss the results.

Exercise 7.3 File TestLongAdders.java contains several implementations of a long (64-bit) integer with thread-safe `add` and `get` operations: (a) Java's AtomicLong; (b) Java 8's LongAdder; (c) a simple `long` field with synchronized operations; (d) a number represented as the sum of multiple "stripes" densely allocated in an AtomicLongArray; and (e) a number represented as the sum of multiple "stripes" allocated as scattered AtomicLong objects.

If you do not have Java 8, delete or comment out the code (b) that uses class LongAdder.

1. Compile the file and run the code to measure, on your own hardware, the performance of the various atomic long implementations. Report the numbers and discuss whether they are plausible, eg. relative to the number of cores in your machine and the number of threads trying to access the thread-safe long integer.

2. Create a new class NewLongAdderLessPadded as a variant of the NewLongAdderPadded class, where you remove the strange `new Object()` creations in the constructor. Create a suitable version of the method `exerciseNewLongAdderPadded` where you measure the performance of this new class along with the others.

   Do those `new Object()` allocations make any difference, positive or negative, on your hardware?

Exercises week 8
Mandatory handin 4
Friday 24 October 2014

Goal of the exercises

The goal of this week's exercises is for you to show that you can achieve good performance and scalability of lock-based concurrent software, using finer-grained locks, lock striping, the Java class library's atomically updatable numbers, immutability and the visibility effects of volatiles and atomics.

Also, to make sure that you can write responsive user interfaces using threads and make them work correctly.

Do this first

Get and unpack this week's example code in zip file pcpp-week08.zip on the course homepage.

Exercise 8.1 Do Exercise 7.1 and hand it in. (Yes, that's true: everybody who already did this exercise can relax, just hand it in).

Exercise 8.2 Do Exercise 7.2 and hand it in. (Yes, more slacking).

Exercise 8.3 File TestFetchWebGui.java contains a simple Java Swing user interface to initiate the fetching of some web pages and then report their sizes.

As implemented, the program uses a single SwingWorker subclass instance to fetch all the web pages sequentially, which is slow because each download has to complete before the next one starts. In this exercise you must change it so that it initiates multiple downloads at the same time, and prints the results as they become available.

1. Implement concurrent download. You can ignore the cancellation button and progress bar for now. There seems to be two ways to implement concurrent download of N webpages. Either (1) create N SwingWorker subclass instances that each downloads a single webpage; or (2) create a single SwingWorker subclass instance that itself uses Java's executor framework to download the N web pages concurrently. Approach (1) seems more elegant because it uses the SwingWorker executor framework only, instead of using two executor frameworks. Also, approach (2) seems dubious unless it is clear that a SwingWorker's `publish` method can be safely called on multiple threads; what does the Java class library documentation say about this? Implement and explain the correctness of your solution for concurrent download.
2. Make the cancellation button work also with concurrent download.
3. Make the progress bar work also with concurrent download. One way to do this is to create an AtomicInteger that all the download operations update as they complete, and let them all call `setProgress` with a suitable value.

Exercise 8.4 (**Optional.**) File TestLiftGui.java contains an implementation of a lift simulator, corresponding to the north end of the IT University's atrium: two lifts, both serving seven floors, from basement (floor number -1) to floor 5.

1. Explain why the whole simulation and its graphical user interface is thread-safe, in spite of the Swing GUI toolkit components not being thread-safe.
2. Apply the ThreadSafe tool to the simulation program. Does it report any potential problems?
3. Change the lift simulator and GUI to work for a hotel with four lifts, all of which serve floors -2 through 10, and still with a single lift controller.
4. In the current implementation, each lift has a thread whose `run` method uses the `Thread.sleep` method to sleep most of the time. An alternative design is to use the Java executor framework, for instance, a scheduled thread pool, to periodically update each lift's state. The `scheduleAtFixedRate` method of the ScheduledThreadPoolExecutor class in package java.util.concurrent seems relevant. In this design, each lift is represented by a Runnable whose `run` method gets called, say, 16 times a second. The main work in

this rewriting probably is to introduce extra fields in the Lift object so that the lift "knows" which state it is in: going nowhere (direction `None`), going up (direction `Up`), going down (direction `Down`), opening doors, or closing doors, and so that the `run` method can act accordingly. There should be **no** calls to `sleep` left in the Lift methods.

5. Modify the user interface so that a lift's inside buttons show which floors the lift will eventually stop at. For instance, you may set the foreground (text) color of the button for a given floor to `Color.RED` when the lift will stop there, otherwise the (default) `Color.BLACK`.

Exercise 7.1

Question 1

We have implemented `V get(K k)` similarly to SynchronizedMap where we only lock on the relevant stripe.

Question 2

We iterate through each stripe to get its count. We lock the specific stripe when adding its count to the total count. We do this to ensure visibility.

Question 3

This is implemented in the same way as `put` just without replacing any previous values.

Question 4

This is implemented as in `SynchronizedMap`, but where we only lock the relevant stripe.

Question 5

We have implemented the `forEach` by first making a local reference to the array stored in `buckets` to avoid iterating over the same `ItemNode` several times if `reallocateBuckets` is called simultaneously. Then we iterate through the stripes and lock each of them and iterate its buckets. This will acquire each lock only once.

Question 6

By using the tests we discovered an error in our `remove` method where instead of picking the first `ItemNode` in the chain and iterating, we just picked the `ItemNode` we wanted to remove and thereby we were not able to change the references to exclude it.

Question 7

Our performance measurements are documented in the 7_1_7.txt and 7_1_7.png in the 'benchmarks' folder. The graph is also available at: http://goo.gl/ogw5Un.

This results is partially as expected. The `SynchronizedMap` is fastest when using just a single thread. When using more than one thread, it is twice as slow no matter the amount of threads available. This was surprising to us, and we do not have a reasonable idea of why this happens. Our best guess is that using the synchronized methods introduces an extra overhead, when using more than one thread, increasing the execution time.

The two maps are equally fast when using a single thread, but after that `StripedMap` is remarkable faster. The execution time decreases for the first 6-8 threads (on an 8 core machine), but after that the performance slows down as more threads are fighting to acquire the same locks and more threads are scheduled on the same core. This is to be expected.

Question 8

If we had a lock for each bucket, we would need to create new locks, as the number of buckets increased. The current resizing ensures that a key is always associated with the same stripe, and thereby the same lock. As the `size` array contains the count for each stripe/lock, we would have to recalculate the count for each bucket when resizing. This would take an unreasonably long time.

Question 9

We want to lower the probability of a lock being unavailable. We need at least as many locks as threads, so each thread ideally could hold a lock simultaneously. Having more locks increases the chance of a lock being available when a thread tries to take it.

Question 10

Having the number of buckets a multiple of the number of stripes ensures that each bucket always will be within the same stripe. ???? ???? ???? ???? ???? //Not true-> If not, there is a risk that an interviening call to `reallocateBuckets` could allocate the needed entry to a different stripe which could make the thread lose the lock for the specific entry leading to the risk of losing updates to the entry.

Exercise 7.2

Question 1

We implemented the size method, as described in the assignment text, by iterating through the `sizes` array and summing up its sizes. This is done without the use of locks as the array is an AtomicIntegerArray.

Question 2

If nothing was added in the call to `putIfAbsent` we don't need to write to the stripe `sizes` array in order to ensure visibility.

Question 3

We have implemented the `remove` method just as described in the assignment text.

Question 4

In our implementation of `forEach` we do stripe-wise locking. To ensure proper visibility we read the stripe size before iterating the buckets in the stripe. This ensures that we get the latest updates of the specific stripe. If updates are made after the size is read, we won't be able to see them. This is not a problem since the `forEach` only will guarantee a rolling snapshot.

Question 5

The result of our performance measurement is placed in the file '7_2_5.txt' in the 'benchmark' folder.

The results of our measurements are as expected. `SynchonizedMap` is by far the slowest with `StripedMap` being a good improvement. The non-locking reads in the `StripedWriteMap` gives us an extra performance boost, which is similar to, but not as fast as, the `WrapConcHashMap`.

Question 6

Exercise 8.3

Question 1

We chose to implement method 1 described in the assignment. We create N DownloadWorker instances each downloading a singe webpage. All the DownloadWorkers are executed in parallel. Their result is appended to the textarea through the done method invoked on the event dispatch thread when the task is completed.

Question 2

For cancellation we check before starting the download whether or not the SwingWorker subclass has already been cancelled. If the cancel button is pressed, an ActionListener on the cancel button calls the cancel method on each worker which throws an InterruptedException in the get method.

Question 3

We have made a shared AtomicInteger as a counter keeping track on the running tasks. This is decremented every time a DownloadWorkers `done` method is called and afterwards the progress bar is updated using the `setProgress` method. We do not need the atomicity of the counter, since all the calls on it are made by the event thread. It is just to allow all the workers to work of the same counter.

Exercises week 9
Friday 31 October 2014

Goal of the exercises

The goal of this week's exercises is to make sure that you understand the challenges of testing concurrent software, that you can nevertheless write a reasonable test suite for such software using recommended techniques, and use mutation to judge the quality of the test suite.

Do this first

There are no specific additional files for this week's exercises. Instead they build on your own completion of the TestStripedMap.java file from week 7.

Exercise 9.1 In this exercise you must conduct a functional test of the StripedWriteMap<K,V> implementation of a concurrent hash map presented in week 7's lecture, and completed by you in week 8's mandatory exercise. Since it was designed for this course by the lecturer, and the implementation was completed by you, nobody knows whether it is correct, so we need to test it.

1. First, consider a functional test of the hash map's sequential correctness as attempted in method `testMap` in the file. Does the implementation pass this simple test? Describe any inadequacies in the test, such as lack of method coverage or statement coverage in the hash map implementation. Extend the test to address the deficiencies. Does the implementation still pass the sequential test?

2. Now turn to testing of the hash map's functional correctness in a concurrent context, where multiple threads read and modify the hash map at the same time.

   You may draw inspiration from Goetz at al. section 12.1 which shows how to test a blocking queue:

   - Create a single StripedWriteMap<Integer,String> concurrent hash map instance to test, with Integers as keys and Strings as values. To increase the chance that multiple threads will manipulate the same bucket, and the same stripe, at the same time, you should create the map with few stripes, maybe 7, and with few buckets, maybe 77 — remember that the number of buckets must be a multiple of the number of stripes. Also, you should run with a rather small range of random keys to insert into the table, maybe $0 \ldots 99$, to increase the chance of the same key being added or removed at the same time.
   - Create multiple testing threads to manipulate the concurrent hash map. There should be more threads than cores, but not unreasonably many, so 16 testing threads would be a good choice on most current hardware.
   - Each testing thread performs `containsKey`, `put`, `putIfAbsent` and `remove` on the concurrent hash map, on randomly chosen keys.
   - Each testing thread should have its own random number generator. Using a shared random number generator might affect the thread scheduling and hence impair the thread interleaving coverage of the test.
   - Each testing thread maintains the sum of all new keys it puts into the hash map, minus the keys it removes. Note that neither `put(k,v)` nor `putIfAbsent(k,v)` adds a new key if `k` is already present.
   - After all testing threads have completed, the sum of the keys in the hash map should equal the sum of the sums from the testing threads.
   - Use a CyclicBarrier from package java.util.concurrent to make sure that the testing threads run only when all of them are ready; this minimizes the risk that they will run sequentially.

   Implement such a functional test. Does it find defects in the hash map implementation? To what degree does the test convince you that the StripedWriteMap implementation is correct? In particular, does it tell you anything about the correctness of `containsKey`?

3. Run the functional test also on a WrapConcurrentHashMap<Integer,String> instance; here it hopefully finds no defects. In general, if your test finds a defect in the StripedWriteMap implementation, run the test also on WrapConcurrentHashMap to see whether the deficiency is in the StripedWriteMap or in your test (or both).

4. The functional test as proposed above checks only that the hash map contains the expected keys. To check also that the associated values are correct (or at least plausible) you may number the testing threads `t` = $0 \ldots (N-1)$ and let thread `t` insert a String value of form `"t:k"` for key `k`.

   Then check, when all testing threads have completed, that every entry in the hash map has the form `(k, "t:k")` for some thread number `t`.

5. You can further let each testing thread keep, in an array `int[] counts = new int[`N`]`, a net count of the number of entries "belonging" to thread `t`. That is, if thread `t` adds a new entry `(k,"t:k")` then it increments `counts[t]`. Similarly, if it removes an entry `(k,"u:k")` made by thread `u`, then it decrements `counts[u]`. Note that the latter may happen both as a consequence of `remove(k)` and as a consequence of `put(k,"t:k")`.

   After all testing threads have completed, you should compute the sum of the `counts` arrays and traverse the hash map to check that each thread `t` has precisely as many values in the table as the sum indicates. For instance, if the sum of the N threads' `count[7]` fields is 426, then there should be 426 entries in the table of the form `(7, "7:k")`.

6. Suggest further ways to improve the test of the concurrent hash map implementations.

Why don't we just compare the results of operations on StripedWriteMap<K,V> with the results of doing the same operations on WrapConcurrentHashMap<K,V>, which presumably is a good reference implementation? The reason is two-fold: (1) While we may control the generation of pseudo-random numbers, we do not control the thread scheduler and hence the interleaving of the threads' method calls, so we cannot expect to make two identical test runs, one on StripedWriteMap<K,V> and another on WrapConcurrentHashMap<K,V>. (2) Then we could manipulate both implementations in the same test run, thus exposing them to the exact same sequence of operations. But that would cause any synchronization internally in the reference implementation to interfere with the test thread scheduling, which would make the test much less effective. Probably this is not a big concern in the case of WrapConcurrentHashMap<K,V> which does little internal locking, but often the "reference implementation" will be a fully locking, basically sequential, implementation and that would completely invalidate the test of a more concurrency-friendly new implementation.

Exercise 9.2 If your functional test in Exercise 9.1 finds no defects in the hash map implementation, you may investigate how good the test is by *mutation testing*, by *injecting faults* in the hash map implementation and running the functional test again. For instance, you may:

1. Remove `synchronized` around one or more blocks of code to see whether the functional test "discovers" the lack of synchronization.

2. Change a single occurrence of `synchronized (locks[stripe])` so that it locks on the wrong object, for instance by replacing it with `synchronized (locks[0])` or `synchronized (this)`, to see whether the functional test "discovers" the improper synchronization.

3. Change the representation of the `sizes` array from AtomicIntegerArray to plain `int[]` and the `get` and `getAndAdd` method calls to plain array reads `sized[stripe]` and increments `sized[stripe]++`, to see whether the functional test "discovers" that the sizes are not correctly updated.

4. Remove some of the reads from `sizes[stripe]` to see whether the absence of these atomic reads affects visibility of writes to reads.

   It is probably unlikely that the functional test will discover this particular fault, although it undermines the visibility of writes to subsequent reads. Also, it is not obvious how to devise a test that would reliably reveal this lack of visibility.

5. What other ways might there be of injecting faults so as to investigate how good the functional test is? Discuss, and if possible, suggest and try out other faults that may be injected.

Exercise 9.1

Question 1

The `forEach` and `reallocate` methods are not covered by the tests in the `testMap` method.

To cover the `forEach` method we made a test ensuring that the sum of iterations is equal to the amount of buckets. Also, we tested if the inserted ItemNodes are outputtet by the method

To cover the `reallocate` method we made a test ensuring that the content is the same after execution. Unfortunately. it is not possible to test that the size of the buckets array has increased or if the ItemNodes is still in the same stripe after reallocation.

After running the tests, the implementation seems to be correct.

Question 2

Question 3

Question 4

Question 5

Question 6

Exercise 9.2

Question 1

Question 2

Question 3

Question 4

Question 5

Exercises week 10
Mandatory handin 5
Friday 7 November 2014

Goal of the exercises

The goal of this week's exercises is to make sure that you can write concurrent programs using the transactional memory approach to mutable shared state, and that you can assess the advantages and pitfalls of (optimistic) transactional concurrency compared to the lock-based pessimistic concurrency.

Do this first

Get and unpack this week's example code in zip file pcpp-week10.zip on the course homepage.

Download the Multiverse library `multiverse-core-0.7.0.jar` from the location shown on the public course homepage, and put it in a suitable place such as `~/lib/multiverse-core-0.7.0.jar`.

To compile and run a Java file such as `TestAccounts.java` with Multiverse use the following commands:

```
javac -cp ~/lib/multiverse-core-0.7.0.jar TestAccounts.java
java -cp ~/lib/multiverse-core-0.7.0.jar:. TestAccounts
```

Exercise 10.1 (Optional) Test and time the queue implementation in TestStmQueues.java.

1. This week's lecture presented a bounded queue class StmBoundedQueue implemented using transactional memory. Test it using the test suite developed in week 9's lecture; see file TestBoundedQueueTest.java. Does the new queue pass those tests?

2. Measure the overall time to run the above-mentioned test, on week 9's lock-based SemaphoreBoundedQueue implementation as well as on this week's StmBoundedQueue implementation. Use the simple Timer class directly to measure the time from right after passing the start CyclicBarrier to right after passing the end CyclicBarrier; you do not need the Mark6 or Mark7 timing infrastructure. How well does the transactional queue implementation perform compared to the lock-based one from week 9?

Exercise 10.2 Implement a histogram class `StmHistogram` using transactional memory. You should use the Multiverse library for Java and *not* `synchronized`, locks, or Java atomics. Build on the partial solution in file stm/TestStmHistogram.java.

The basic histogram functionality is as in the week 3 and 6 exercises, but the interface now is slightly different:

```
interface Histogram {
  void increment(int bin);
  int getCount(int bin);
  int getSpan();
  int[] getBins();
  int getAndClear(int bin);
  void transferBins(Histogram hist);
}
```

Method `getAndClear(bin)` should atomically return the count in bin `bin` of the histogram and also reset that count to zero.

Method `transferBins(hist)` should transfer all the counts from `hist` to this histogram, by adding each bin count in `hist` to this histogram and atomically also setting that count to zero. Thus if `this.getCount(7)` is 20 and `hist.getCount(7)` is 30 before the call, and there are no other calls going on, then after the call `this.getCount(7)` is 50 and `hist.getCount(7)` is 0.

(The `transferBins` operation is more meaningful and useful than the `addAll` implemented in previous exercises. In an application where multiple threads update their own histograms, `transferBins` can be used to periodically aggregate the thread-local histograms into a common global histogram, without losing or duplicating any counts.)

1. Implement the basic methods `increment`, `getCount` and `getSpan` in your StmHistogram class. The bins of the histogram should be held as transactional integer variables, that is:

```
class StmHistogram implements Histogram {
  private final TxnInteger[] counts;
  ...
}
```

2. The file stm/TestStmHistogram.java contains code to run 10 threads in parallel to count the number of prime factors in all the numbers in the range 0. . . 3 999 999. It uses your transactional histogram implementation to maintain the counts.

   The correct result should look like this:

```
0:          2
1:     283146
2:     790986
3:     988651
4:     810386
5:     524171
6:     296702
7:     155475
8:      78002
9:      38069
... and so on
```

   showing that 283 146 numbers in 0. . . 3 999 999 have 1 prime factor (those are the prime numbers), 790 986 numbers have 2 prime factors, and so on. (The 2 numbers that have 0 prime factors are 0 and 1). And of course the numbers in the second column should add up to 4 000 000. Run this code. Does it produce the correct result with your histogram implementation?

3. Implement method `getBins` so that it returns an array of the counts in the bins of the histogram.

4. Implement method `getAndClear(bin)` so that it atomically returns the count in bin `bin` of the histogram and also resets that count to zero.

5. Implement method `transferBins(hist)` so that it transfers all the counts from `hist` to this histogram, by adding each bin count in `hist` to this histogram and atomically also setting that count to zero. The `getAndClear` method should be useful in doing this.

   Note that `transferBins(hist)` can be implemented in at least two ways: (a) use one large transaction that transfers all the bins; or (b) use a transaction for each bin that atomically transfers that bin. Since (b) gives shorter transactions and therefore less likelihood that the transaction will fail and be retried, it is probably much preferable to (a). With (b), if a concurrent thread increments the `hist` counts, there may be no point in time at which all bins of `hist` are actually zero. This is acceptable so long as no counts are lost and no counts are duplicated during the transfer.

6. Now extend the code from subquestion 2 so that the main thread creates a new StmHistogram instance called `total` and occasionally calls `total.transferBins(histogram)` where `histogram` is the one the prime counting threads write to.

   The main thread may do this 200 times, say every 30 milliseconds by calling `Thread.sleep(30)` in between the calls to `transferBins`. It should start doing this only after the prime counting threads have been started. When all the threads have terminated, the main thread should call `dump(total)` to print the `total` histogram.

   At the end `histogram`'s counts should be all zero, and `total`'s counts should be what `histogram`'s used to be, regardless when and how many times `transferBins` has been called. Is this the case?

7. What effect would you expect `total.transferBins(total)` to have? What effect does it have in your implementation? Explain.

Exercise 10.3 Implement a concurrent hash map `StmMap` using transactional memory. Start from the sketch in file stm/TestStmMap.java. You should use the Multiverse library for Java and *not* `synchronized`, locks, or Java atomics. Thanks to the visibility effects of the Multiverse `atomic` transactions, there is no need for subtle `volatile` tricks or similar.

The basic map functionality should be as in the week 7, 8 and 9 exercises, but you can ignore the internal `reallocateBuckets` method — although it is quite easy to implement it *correctly* using transactional memory, it is not clear how to implement it *efficiently*.

Use immutable ItemNode<K,V> nodes as in the StripedWriteMap implementation; you can reuse the item node class exactly as it is.

The entries in the `buckets` array are mutable and updates to them must be under transactional control, so each entry must have type TxnRef<ItemNode<K,V>>. Moreover, the `buckets` field itself is mutable and must be a TxnRef<. . .>, so in total `buckets` should have this somewhat impressive declaration:

```
class StmMap<K,V> implements OurMap<K,V> {
  private final TxnRef<TxnRef<ItemNode<K,V>>[]> buckets;
  ...
}
```

That is, a transactional reference to an array of transactional references to ItemNode<K,V> objects.

1. Implement the `get` method. You can either (a) enclose the entire method body in `atomic`, or (b) use `atomic` only around the code that accesses the `buckets` array reference and indexes into the array, or, since nothing gets updated in these methods, presumably you can (c) use the `atomicGet` method on the TxnRefs. Doing (a) seems simplest, but (b) better separates the transactional `buckets` accesses from the subsequent readonly search of the immutable item node lists and so makes the transaction shorter.

2. Implement the `forEach(consumer)` method. Use approach (b) or (c) outlined in the previous subquestion to keep the transaction short. The call to `consumer(node.k, node.v)` should clearly not be inside a transaction, because the transaction may fail and be restarted, in which case `consumer` may be called an arbitrary number of times for each entry in the hash map — very unlikely to be what is expected.

3. Implement the `put`, `putIfAbsent` and `remove` methods. Do not worry about updating the size count for now. Briefly explain why you believe your implementations are correct.

4. Implement the `size` method. The simplest approach is to use a single TxnInteger to hold the total number of entries in the hash map, and update that field inside the `put`, `putIfAbsent`, and `remove` transactions. That will probably be a concurrency bottleneck and lead to poor scalability on a manycore machine, but this is acceptable here.

5. Discuss the problems involved in implementing `reallocateBuckets` efficiently using transactions and optimistic concurrency. There seems to be at least two problems: (1) It makes no sense to transfer only half the hash table buckets from the old `buckets` to the new one, so the reallocation should be in a transaction. But that might be a very long transaction, with a high likelihood that some concurrent `put`, `putIfAbsent` or `remove` transaction causes the reallocate transaction to abort and then restart, again and again, wasting much computation. Moreover (2) by reallocating optimistically, many threads could start overlapping reallocations, and in the end only one of those transactions will succeed, again wasting all the computation performed in the failing transactions.

   So it seems that one needs a protocol by which all other updating (`put`, `putIfAbsent`, `remove`) threads block when one thread has started a reallocation. Could one have a transactional field `newBuckets` alongside `buckets`, with the convention that `newBuckets` is non-`null` exactly while one thread is reallocating, and all other mutating threads wait for `newBuckets` to be `null`? How does a thread block using transactions? (Hint: see the lecture's bounded queue implementation).

   Discuss this idea in approximately 15 lines of text; you do not need to implement it.

Exercise 10.2

Question 1

We started by creating the constructor, which simply creates an array of `TxnInteger` and use the factory method `newTxnInteger` to individually creating each of them. This is not done within an `atomic` block, since the constructor is not accessed concurrently.

The `increment` and `getCount` simply wrap the code in an `atomic` block from the Multiverse API. As the value in `getSpan` is final, we simply return this value.

Question 2

Yes, we got the correct solution. We double checked the values with our earlier implementation of `Histogram` from Exercise 3.

Question 3

For `getBins` we create a `final` array with the same span as the histogram. We then atomically retrieve each bin count individually in a loop. This gives us a lot of short transactions, instead of getting all the values in one long transaction, thus making retries less costly.

We have used this method in a new `dump2` method, to ensure that it works as intended. The method is used at the end of `countPrimeFactorsWithStmHistogram`.

Question 4

The `getAndClear` method simply caches the previous value before setting it to zero and returning the cached value. This is all wrapped in an `atomic` block.

It seems the TxnInteger supports this atomic action already, but we found the documentation to be too insufficient and even conflicting to be sure that we got the correct results.

Question 5

The `transferBins` method works on each bin seperately as in `getBins`. We use the `getAndClear` method to get and reset the bin value, and atomically increment the bin value in the current histogram.

Question 6

We introduced a new histogram `total`, which is updated every 30 ms. using `transferBins`. We use the `getNumberWaiting` method on the `stopBarrier` to see when all the threads are done, so we can stop the loop. At the end we dump both histograms to ensure `total` contains all counts and that `histogram` is empty.

Question 7

This shouldn't have any effect other than probably a slight decrease in performance. Each value will be atomically read and cleared, and then the retrieved value will be added to the current value, thereby restoring the difference. We therefore do not lose any values.

This has no effect on our implementation.

Exercise 10.3

Question 1

We have chosen to implement `get` by using two calls to `atomicWeakGet` on `buckets` and `bs[bucket]`. In this way we don't have to enclose everything in the method in an `atomic` block.

Question 2

We have implemented `forEach` similarly to `get` by using `atomicWeakGet` to get a reference to the array of `TxnRef`s. We use this reference when we retrieve each bucket. Again, we use `atomicWeakGet` to retrieve the first node before even calling the `consumer`. We iterate the immutable chain and call the `consumer` on each node. This ensures that `consumer` is only called once for each node.

Question 3

We were not able to rely only on the predefined methods, such as `setAndGet`, on `TxnRef` when implementing `put`, `putIfAbsent` and `remove`. This was because retrieve the node and either delete it or insert a new node in the same atomic operation. We need the return value to know what to put back in the bucket, so an atomic operation would not be possible. Instead we wrap the critical part in an `atomic` block, and do the necessary manipulations there.

We currently do not expect the value of `buckets` to ever change, as we do not reallocate it anymore, so we don't wrap the retrieval of it's value in any atomic blocks.

Question 4

We instantiated a variable `size` as a `TxnInteger` containing the total number of entries. This is called in all of the methods created in question 3 where we decrement or incement it respectively. Using this `TnxInteger` could cause a bottleneck as it is frequently called.

Question 5

We have drafted some code showing how we think a blocking mechanism could work. The code is found in a section labeled `CONCEPT CODE FOR 10.3.5` in the source file `TestStmMap.java`.

The idea is to keep a boolean transaction variable called `isReallocating` which is set to true only when some thread is reallocating the `buckets` array.

Methods that need to block while the reallocation is happening, must start out by checking if any reallocation is currently happening. An example can be seen in the `blocksWhileReallocating` method. We make an atomic block and start out by checking the value of `isReallocating`. If the value is true, we call `retry()` in order to block and wait for the value to change again, so we can do our work. This call is followed by the method's actual code. Should it be the case that reallocation has started when reaching the end of the transaction, the transaction will be aborted and restarted and the `retry` method will be called (provided that the value still hasn't changed) thereby explicitly blocking until the reallocation is done.

The `reallocateBuckets` method, which is responsible for resizing the buckets array, has to block all other methods that try to write until it is done. It starts out by checking if `isReallocating` is true. If this is the case, another thread is already resizing, therefore we simply return to avoid doing the same work twice. Otherwise, it sets the variable to true and exits the atomic block to ensure the value is visible to the other threads immediately. This will cause writing methods to block from now on. We then start to do the reallocation and overwrite the `buckets` variable atomically with the new array using the `atomicSet`. Finally we atomically set the value of `isReallocating` to false, so the blocking threads can continue their work.

Exercises week 11
Friday 14 November 2014

Goal of the exercises

The goal of this week's exercises is to make sure that you can use lock-less approaches to mutable shared memory, and that you can use the compare-and-swap primitive to implement simple lock-less data structures.

Do this first

Get and unpack this week's example code in zip file pcpp-week11.zip on the course homepage.

Exercise 11.1 Implement a CasHistogram class in the style of week 10 with this interface:

```
interface Histogram {
  void increment(int bin);
  int getCount(int bin);
  int getSpan();
  int[] getBins();
  int getAndClear(int bin);
  void transferBins(Histogram hist);
}
```

The implementation should use AtomicInteger instead of transactions or locks, and use *only* methods `get` and `compareAndSet`, not the other methods provided on AtomicInteger. This should be quite easy if you take some hints from the lecture.

1. Write a class CasHistogram so that it implements the above interface. Explain why the methods `increment`, `getBins`, `getAndClear` and `transferBins` work.
2. Use your new CasHistogram class for the parallel prime counting example; you can take most of the code from week 10's stm/TestStmHistogram.java example file. Does it produce the right results (see Exercise 10.2.2) when run on this example?
3. Measure the overall time to run the above-mentioned test, on week 10's StmHistogram implementation as well as on this week's CasHistogram implementation. Use the simple Timer class directly to measure the time from right after passing the start CyclicBarrier to right after passing the end CyclicBarrier; you do not need the Mark6 or Mark7 timing infrastructure. Report the measured running times. Which implementations is fastest? Reflect on the possible reasons.
4. (Optional) Measure the overall time to run the above-mentioned test also on week 3's lock-based Histogram2 implementation. As before, use the simple Timer class directly to measure the time from right after passing the start CyclicBarrier to right after passing the end CyclicBarrier; you do not need the Mark6 or Mark7 timing infrastructure. How does the performance of that (coarse) lock-based implementation compare to the CAS-based one in this application?

Exercise 11.2 A read-write lock, in the style of Java's java.util.concurrent.locks.ReentrantReadWriteLock, can be held either by any number of readers, or by a single writer. In this exercise you must implement a simple read-write lock class SimpleRWTryLock that is not reentrant and that does not block. It should have the following four public methods:

```
class SimpleRWTryLock {
  public boolean readerTryLock() { ... }
  public void readerUnlock() { ... }
  public boolean writerTryLock() { ... }
  public void writerUnlock() { ... }
}
```

Method `writerTryLock` is called by a thread that tries to obtain a write lock. It must succeed and return true if the lock is not already held by any thread, and return false if the lock is held by at least one reader or by a writer.

Method `writerUnlock` is called to release the write lock, and must throw an exception if the calling thread does not hold a write lock.

Method `readerTryLock` is called by a thread that tries to obtain a read lock. It must succeed and return true if the lock is held only by readers (or nobody), and return false if the lock is held by a writer.

Method `readerUnlock` is called to release a read lock, and must throw an exception if the calling thread does not hold a read lock.

The class can be implemented using AtomicReference and compare-and-swap, by maintaining a single field `holders` which is an atomic reference of type Holders, an abstract class that has two concrete subclasses:

```
private static abstract class Holders { }

private static class ReaderList extends Holders {
  private final Thread thread;
  private final ReaderList next;
  ...
}

private static class Writer extends Holders {
  public final Thread thread;
  ...
}
```

The ReaderList class is used to represent an immutable linked list of the threads that hold read locks. The Writer class is used to represent a thread that holds the write lock. When `holders` is `null` the lock is unheld.

(Representing the holders of read locks by a linked list is very inefficient, but simple and adequate for illustration. The real Java ReentrantReadWriteLock essential has a shared atomic integer count of the number of locks held, supplemented with a ThreadLocal integer for reentrancy of each thread and for checking that only lock holders unlock anything. But this would complicate the exercise. Incidentally, the design used here allows the read locks to be reentrant, since a thread can be in the reader list multiple times, but this is inefficient too).

1. Implement the `writerTryLock` method. It must check that the lock is currently unheld and then atomically set `holders` to an appropriate Writer object.

2. Implement the `writerUnlock` method. It must check that the lock is currently held and that the holder is the calling thread, and then release the lock by setting `holders` to `null`; or else throw an exception.

3. Implement the `readerTryLock` method. This is marginally more complicated because multiple other threads may be trying (successfully) to lock at the same time, or may be unlocking read locks at the same time. Hence you need to repeatedly read the `holders` field and so long as it is either `null` or a ReaderList attempt to update the field with an extended reader list, containing also the current thread.

4. Implement the `readerUnlock` method. This also requires a loop and for the same reason as above. You should repeatedly read the `holders` field and so long as it is non-`null` and refers to a ReaderList and the calling thread is on the reader list, create a new reader list where the thread has been removed, and try to atomically store that in the `holders` field; if this succeeds, it should return. If `holders` is `null` or does not refer to a ReaderList or the current thread is not on the reader list, then it must throw an exception.

   For the `readerUnlock` method it is useful to implement a couple of auxiliary methods on ReaderList:

   ```
   public boolean contains(Thread t) { ... }
   public ReaderList remove(Thread t) { ... }
   ```

5. Write simple sequential test cases that demonstrate that your read-write lock works with a single thread. For instance, it should not be able to take a read lock while holding a write lock, and vice versa, and should not be allowed to unlock a read lock or write lock that it does not already hold.

6. Write slightly more advanced test cases that use at least two threads to test basic lock functionality.

Exercise 11.3 This exercise concerns the scalability of five different pseudo-random number generators.

1. Run the scalability test in file TestPseudoRandom.java on your own computer and preferably also a different one for comparison. By default the scalability test runs with 1 to 32 threads; if your computer has 2 or 4 cores you may reduce the 32 thread to 16 or 8 threads. Hand in the results in a table or graphical form and reflect on them. Which random number generator is fastest in absolute terms, and which one scales best with more threads?

Exercises week 12
Friday 21 November 2014

Goal of the exercises

The goal of this week's exercises is to make sure that you can work with lock-free data structures, test them, and measure their performance.

Do this first

Get and unpack this week's example code in zip file pcpp-week12.zip on the course homepage.

Exercise 12.1 This exercise is about testing the lock-free Michael-Scott queue class MSQueue presented in the lecture, and implemented in file TestMSQueue.java. That queue is unbounded, so the `enqueue` method will never block, and symmetrically the `dequeue` method is non-blocking: it just returns `null` if the queue is empty, instead of waiting for an item to arrive in the queue.

1. Write a simple sequential test for the Michael-Scott queue implementation. You may adapt the sequential test from week 9's TestBoundedQueueTest.java file.
2. Write a concurrent test for the Michael-Scott queue implementation. You may adapt the concurrent test from the same file mentioned above. In the original test it is enough for each consumer to perform `nTrials` calls to `take` because each call is guaranteed to return an item, but when testing the non-blocking queue a consumer must loop and call `dequeue` until it has obtained `nTrials` actual non-`null` items.

   Does the MSQueue implementation pass the concurrent test?
3. Inject some faults in the MSQueue implementation and see whether the test detects them. Describe the faults and whether the test detects them, and if it does detect them, how it fails.

Exercise 12.2 In this exercise we take a closer look at the Michael-Scott queue implementation described in Michael and Scott's paper, and implemented by class MSQueue in file TestMSQueue.java.

1. The checks performed at source lines E7 and D5 look reasonable enough, but are they really useful? For instance, it seems that right after `(last == tail.get())` was successfully evaluated to true at E7, another thread could modify `tail`. Hence it seems that the check does not substantially contribute to the correctness of the data structure.

   Do you agree with this argument? Think about it, make some drawings of possible scenarios, perform some computer experiments, or anything else you can think of, and report your findings.
2. Run the sequential and concurrent tests from Exercise 12.1 on a version of the MSQueue class in which you have deleted the check at line E7 in the source code. Does it pass the test?
3. Run the sequential and concurrent tests from Exercise 12.1 on a version of the MSQueue class in which you have deleted the check at line D5 in the source code.
4. If the checks at lines E7 and D5 are indeed unnecessary for correctness, what other reasons could there be to include them in the code? How would you test your hypotheses about such reasons?
5. Describe and conduct an experiment to cast some light on the role of one of E7 and D5.

Exercise 12.3 In this exercise you must measure the scalability of some unbounded queue implementations.

1. Measure performance of the Michael-Scott queue implementation class MSQueue that uses an AtomicReference<Node<T>> in each Node<T> object.

   You may plan for moderate contention, for instance use N threads that call `enqueue` and N that call `dequeue`, and perform some computation in the meantime, and for moderate N such as $1 \ldots 4$. For instance, the producers may produce prime numbers and the consumers check they are prime (using the `isPrime` method from several previous examples).

2. Measure performance of the Michael-Scott queue implementation class MSQueueRefl that instead uses a volatile Node<T> field in each Node<T> object, and the AtomicReferenceFieldUpdater to avoid the allocation of an AtomicReference<Node<T>> for each Node<T> object.

   How much does this change improve the performance?

3. Implement a lock-based unbounded queue, still based on the Node<T> class, but using synchronized `enqueue` and `dequeue` methods, none of which blocks. Measure the performance of this queue implementation and compare with the two Michael-Scott queue implementations.

4. Measure also the performance of a version of the Michael-Scott queue where the E7 and D5 checks have been removed, as discussed in Exercise 12.2.

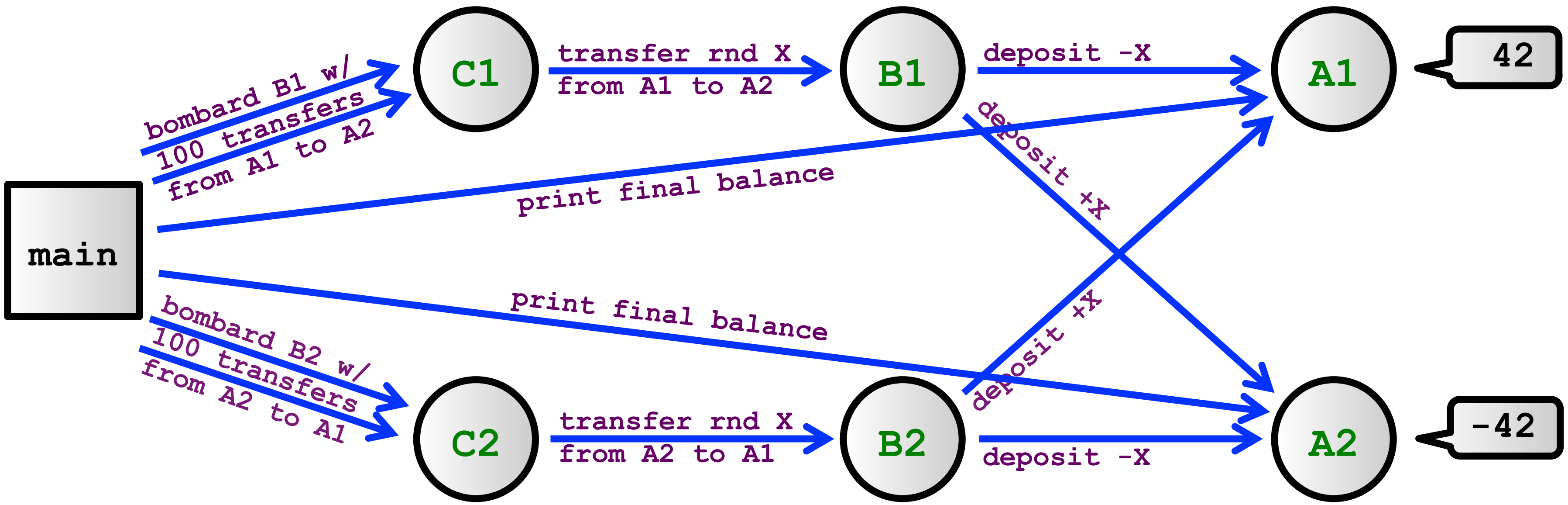
Exercise 12.4 (**Optional:**) File TestUnionFind.java contains three implementations of the union-find data structure, and basic test sequential and concurrent test cases for them.

1. Compile and run the test cases. Do the implementations pass the tests?

2. Now try to mutate the implementations with faults, starting with the coarse-locking CoarseUnionFind implementation. In particular, make them thread-unsafe, for instance by removing synchronization. Can you provoke the concurrent test to fail at all? This may be hard.

3. Make the concurrent test harder. For instance, you may increase the chance that two threads manipulate the same nodes at the same time. Is it possible to make the concurrent test fail on mutated implementations?

5) ABC (Clerk / Bank / Account)



5) ABC.erl

```
-module(helloworld).
-export([start/0,
         account/1,bank/0,clerk/0]).

%% -- BASIC PROCESSING -------------------
n2s(N) -> lists:flatten( %% int2string
   io_lib:format("~p", [N])). %% HACK!

random(N) -> random:uniform(N) div 10.

%% -- ACTORS ----------------------------

account(Balance) ->
   receive
      {deposit,Amount} ->
         account(Balance+Amount) ;
      {printbalance} ->
         io:fwrite(n2s(Balance) ++ "\n")
   end.

bank() ->
   receive
      {transfer,Amount,From,To} ->
         From ! {deposit,-Amount},
         To ! {deposit,+Amount},
         bank()
   end.
```

```
ntransfers(0,_,_,_) -> true;
ntransfers(N,Bank,From,To) ->
   R = random(100),
   Bank ! {transfer,R,From,To},
   ntransfers(N-1,Bank,From,To).

clerk() ->
   receive
      {start,Bank,From,To} ->
         random:seed(now()),
         ntransfers(100,Bank,From,To),
         clerk()
   end.


start() ->
   A1 = spawn(helloworld,account,[0]),
   A2 = spawn(helloworld,account,[0]),
   B1 = spawn(helloworld,bank,[]),
   B2 = spawn(helloworld,bank,[]),
   C1 = spawn(helloworld,clerk,[]),
   C2 = spawn(helloworld,clerk,[]),
   C1 ! {start,B1,A1,A2},
   C2 ! {start,B2,A2,A1},
   timer:sleep(1000),
   A1 ! {printbalance},
   A2 ! {printbalance}.
```

5) ABC.java (Skeleton)

```java
import java.util.Random;   import java.io.*;   import akka.actor.*;

// -- MESSAGES --------------------------------------------------
class StartTransferMessage implements Serializable { /* TODO */ }
class TransferMessage implements Serializable { /* TODO */ }
class DepositMessage implements Serializable { /* TODO */ }
class PrintBalanceMessage implements Serializable { /* TODO */ }

// -- ACTORS --------------------------------------------------
class AccountActor extends UntypedActor { /* TODO */ }
class BankActor extends UntypedActor { /* TODO */ }
class ClerkActor extends UntypedActor { /* TODO */ }

// -- MAIN --------------------------------------------------
public class ABC { // Demo showing how things work:
    public static void main(String[] args) {
        final ActorSystem system = ActorSystem.create("ABCSystem");

        /* TODO (CREATE ACTORS AND SEND START MESSAGES) */

        try {
            System.out.println("Press return to inspect...");
            System.in.read();

            /* TODO (INSPECT FINAL BALANCES) */

            System.out.println("Press return to terminate...");
            System.in.read();
        } catch(IOException e) {
            e.printStackTrace();
        } finally {
            system.shutdown();
        }
    }
}
```

```
*** OUTPUT ***
Press return to inspect...
Press return to terminate...
Balance = 42
Balance = -42
```

MANDATORY HAND-IN!

a) Color ABC.erl
(according to color convention):
send, **receive**, **msgs**
actors, **spawn**, **rest.**
(try 2 B as consistent as possible)

b) Implement ABC.java
(as close to **ABC**.erl as possible)

c) Answer question:
What happens if we replace
{deposit, ±Amount} w/ the msgs?: