Modern Java Concurrency
Concurrency – 223

Alastair F. Donaldson

November/December 2016

Note: I'd be grateful for feedback on these lecture notes and associated materials, including any typos that you find, or any parts where you find the notes lacking in clarify. Also if you'd like to discuss Java concurrency issues further then I'd be very happy to chat with you. Drop me a line to report issues or follow up: `afd@imperial.ac.uk`.

Acknowledgment: I found Benjamin Winterberg's tutorials on Java 8 concurrency to be a really useful resource when preparing these lectures. I thoroughly recommend reading his three blog posts on the topic, starting with this one:

`http://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/`.

1 Introduction

The aim of these lectures is to give you a brief tour of some of the more recent developments in Java related to concurrency. Compared with the traditional approach to concurrency—the `Thread` class and `Runnable` interface, and the use of `synchronized` methods and blocks—the more recent language and library features support both higher- and lower-level programming, for developers who in the first case wish to be relieved of some of the pain of concurrent programming, or in the second case wish to have an even more detailed degree of control over concurrent execution compared with what the traditional language mechanisms provide.

The lectures will be delivered in a live programming style, and these notes are intended for you to read afterwards to remind yourself what we covered.

Code The code associated with these lectures is available here:

`http://www.doc.ic.ac.uk/~afd/teaching/Concurrency`

This includes both the skeleton code that I will start with during the lectures as I demonstrate each concept, and the final code that we produce during the classes. In what follows, when I refer to packages and class I am referring to the final version of the code, not the skeleton.

2 Running example: a hash set

To illustrate some ideas related to concurrency, we will develop a simple class representing a hash set. In practice, thread-safe collections ship with the Java libraries, so you shouldn't have to write your own, but the hash set example is nevertheless good for teaching purposes.

The hash set class and methods Package `basichashset` includes a class, `MyHashSet`, that implements a basic generic hash set with a few operations:

- `boolean contains(T item);` – returns *true* if and only if an object `.equals()` to `item` is in the set
- `boolean add(T item);` – adds `item` to the set unless an object `.equals()` to `item` was already there, returning true if and only if `item` was added
- `boolean remove(T item);` – removes an object `.equals()` to `item` if one exists, returning true if and only if something was removed
- `int size();` – returns the number of elements currently in the set

Sequential use of the hash set Class `Main` illustrates a basic usage of the hash set in a sequential context. The intention is that the hash set should be fine for sequential use (though of course it is not an optimized implementation).

Thread-safety The hash set class is *not* thread-safe. This is demonstrated by class `BadThreads`, where multiple threads try to add elements to a hash set concurrently. On my laptop, this leads to a null pointer exception inside the `LinkedList` class that I have used to implement the buckets of the hash set. The issue is that `LinkedList` is not thread-safe, so bad things can be expected to happen when multiple threads modify a linked list concurrently.

Aside: use of lambda expressions to write threads compactly Notice that in `BadThreads` I have written:

```
Runnable addElements = () -> {
  for(int i = 0; i < 1000; i++) {
    hs.add("word" + i);
  }
};
```

to create an instance of a `Runnable` in a very compact fashion. This is possible because `Runnable` is a *functional interface*—an interface with just one method. In the case of `Runnable` the single method is the method with signature `void run()`. As a result, we can provide an anonymous instance of `Runnable` using a lambda expression of the form `() -> { ... }`, which describes a method that takes no arguments and causes the statements inside the braces to be executed.

This is much more compact than the alternative, equivalent, approach of using an anonymous class, which looks like this:

```
Runnable addElements = new Runnable () {
  @Override
    public void run() {
      for(int i = 0; i < 1000; i++) {
        hs.add("word" + i);
      }
    }
};
```

For more information about Java 8 lambdas, check out my lectures on the topic: `http://www.doc.ic.ac.uk/~afd/teaching/Java8/Java8-Lambdas-Slides.pdf`. There are many good tutorials online about this as well.

3 Using synchronized in an attempt to make the hash set thread-safe

Let us now turn our attention to making the hash set thread-safe.

A trivial solution: synchronized methods We could trivially make the hash set thread-safe by marking all its methods as `synchronized`. This would lead to poor performance, though, because it would *sequentialize* access to the hash set. It would be impossible for updates to distinct buckets in the hash set to be processed concurrently.

A finer-grained approach: synchronized blocks Instead, we can apply `synchronized` at a finer level of granularity. In `contains` we can wrap the check for whether the element is inside the relevant bucket in a synchronized block, synchronizing on `bucket`:

```
public boolean contains(T item) {
  List<T> bucket = getBucket(item);
  if(bucket == null) {
    return false;
  }
  synchronized(bucket) {
    return bucket.contains(item);
  }
}
```

We can use `synchronized` similarly in `add` and `remove` to ensure exclusive access to a bucket when reading or modifying it.

A challenge is how to create a new bucket in `add` in a thread-safe manner when no bucket yet exists.

Here is the beginning of the original code for `add`:

```
public boolean add(T item) {
  List<T> bucket = getBucket(item);
  if(bucket == null) {
    bucket = new LinkedList<>();
    buckets.set(getBucketIndex(item), bucket);
  }
  ...
}
```

To attempt to make this code thread-safe, we employ the following strategy. If a thread finds that `bucket` is null, the thread will proceed with creating a bucket. However, it is possible that two threads might both find the same bucket to be null, and both try to create it. To avoid this, the threads must synchronize on some common object. They cannot synchronize on the bucket, because it does not yet exist! Instead, they can synchronize on the `this` reference to the hash set itself. This suggests code along the following lines:

```
List<T> bucket = getBucket(item);
if(bucket == null) {
  synchronized(this) {
    bucket = new LinkedList<>();
    buckets.set(getBucketIndex(item), bucket);
  }
}
```

However, this is not quite right: it is possible for both threads to find that `bucket` is null, and then for *both* threads to (one after another) create a new bucket. Each thread will then have a separate bucket for items that should hash to the *same* bucket. Only the bucket for the second thread to pass through the synchronized block will remain in the hash set; the other bucket will be garbage-collected and the item added by the first thread will get lost as a result.

To overcome this, a thread can check whether the bucket is still null after it enters the synchronized block:

```
List<T> bucket = getBucket(item);
if(bucket == null) {
  synchronized(this) {
    bucket = getBucket(item);
    if(bucket == null) { // Another thread might have made the bucket in the meantime
      bucket = new LinkedList<>();
      buckets.set(getBucketIndex(item), bucket);
    }
  }
}
```

This *double-checked locking* idiom tries to ensure that a thread only creates a new bucket if the bucket still does not exist after the thread acquires a lock on the hash set.

Unfortunately, the relaxed Java memory model means that double-checked locking as shown here may fail, as a result of compiler optimizations and/or relaxed memory effects. For more details of this, see *The "Double-Checked Locking is Broken" Declaration*, signed by a multitude of Java experts: `http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html`. In Section 4 we will illustrate the relaxed nature of the Java memory model and in Section 5 we will show how *atomic references* can be used to implement double-checked locking correctly.

Lack of synchronization on `size` A less subtle shortcoming of the thread-safe hash set is that `size` is not protected: `size` is incremented in `add` and decremented in `remove`, and can be accessed concurrently by threads updating different buckets. Because the threads only synchronize on the bucket they are working on, access to `size` is not synchronized. As a result, `size` can get out of sync. The crux of the problem here is that incrementing `size` via `size++` is not *atomic*: this statement involves loading the current value of `size` into a register, incrementing that register, then storing the new value back to memory. If two threads execute `size++` concurrently when `size` is 42, they might both load 42 in to registers, both increment 42 to 43, then both store 43 back to memory, so that in effect `size` only gets incremented once.

The package `synchronizedhashset` contains the semi-thread-safe version of the hash set. Class `LessBad Threads` illustrates that the code works more reliably: on my PC I no longer find that exceptions are thrown. However, occasionally I see that the size of the hash set becomes nonsensical due to the issues related to `size` being updated.

We could resolve the problem with updating `size` by making every access to `size` be synchronized on the hash set. This would be a shame as it would cause `add` and `remove` operations on the hash set to be basically sequentialized. In Section 5 will see how to use an atomic integer to solve this problem.

4 Relaxed memory

Let us take a brief detour to illustrate the relaxed nature of the Java memory model. In the process, we shall introduce the notion of an *executor service*, a *future*, and a *multi-catch* block.

Store buffering Suppose that `x` and `y` are `int` variables, both accessible to threads T1 and T2. Suppose that T1 executes this method:

```
int thread1() {
  x = 1;
  return y;
}
```

and T2 executes this method:

```
int thread2() {
  y = 1;
  return x;
}
```

If both threads are launched, with `x` and `y` initialized to 0, what are the possible values the threads can return?

Under a *sequentially consistent* memory model, where a concurrent execution can be explained as some interleaving of instructions by individual threads, it would be possible to have T1 and T2 return, respectively, 0 and 1, 1 and 0, or 1 and 1. It would *not* be possible for both threads to return 0: there is no interleaving where both threads execute their `return` statements but where neither `x` nor `y` has been updated.

However, the Java memory model *does* allow this situation. This is because the memory model is *relaxed*, to cater for properties of modern hardware, such as the *total store order* memory model that x86 processors exhibit. In particular, x86 processors implement *store buffering*, where stores to memory locations are queued in buffers. Under such a model, it is possible for T1 to issue its write to `x` and T2 to issue its write to `y`, but for these writes to be queued in store buffers waiting to be committed to memory. If the `return` statements are issued before the writes reach memory, the original values of `x` and `y` will be returned, so that both threads return 0. A relaxed memory model also enables certain compiler optimizations that would not be possible in general under an assumption of sequential consistency.

Coding up the store buffering test in Java Take a look at class `Main` in the `relaxedmemory` package. The class declares two static fields, `x` and `y`, that are both integers.

Instead of launching threads via the `Thread` class, the example uses a more modern approach: an *executor service*. The statement:

```
ExecutorService executor = Executors.newFixedThreadPool(2);
```

creates a service for running threads that will use a fixed-size pool of two threads.

The behaviour of T1 is described as a `Callable` instance:

```
Callable<Integer> t1 = () -> {
  x = 1;
  return y;
};
```

`Callable<T>` is a generic interface that declares a single method with signature:

```
public T call() throws Exception;
```

An implementation of `Callable`, for a given `T`, describes the behaviour of a thread that will perform some task and return a result of type `T`, possibly throwing an exception in the process. This is in contrast to a `Runnable`, which cannot return a result, and cannot throw a (non-`RuntimeException`) exception.

Because `Callable<T>` is a *functional interface*, it can be instantiated using a lambda expression, as shown above. This is equivalent to writing the more verbose:

```
Callable<Integer> t1 = new Callable<Integer>() {
  @Override
  public Integer call() throws Exception {
    x = 1;
    return y;
  }
};
```

T2 is described similarly using a lambda:

```
Callable<Integer> t2 = () -> {
  Thread.yield();
  y = 1;
  return x;
};
```

To look for store buffering effects, we can loop a number of times submitting these callables to the executor service. Each time we submit a callable, we get back a *future*. This is an object that will, once the callable has terminated, allow us to get the result that the callable returned:

```
for(int i = 0; i < MAX; i++) {

  x = 0; // Reset the static
  y = 0; // variables

  Future<Integer> r1 = executor.submit(t1); // Submit the threads
  Future<Integer> r2 = executor.submit(t2); // to the executor

  try {
    // The calls to 'get' block until the value associated with the future is ready
    if(r1.get() == 0 && r2.get() == 0) {
      System.out.println("r1 == 0 and r2 == 0");
    }
  } catch (ExecutionException|InterruptedException e) {
    break;
  }
}
```

Futures Notice that we get a `Future<Integer>` for each callable that we submit to the executor service. Each of these is a promise to return an integer at some point in the future. The calls `r1.get()` and `r2.get()` block until the promised integer is ready. Of course, because our threads are so simple, the associated integers will be available almost instantaneously, and very likely before control reaches the `get()` calls. A future is a useful construct in the case that we wish a thread to run in the background computing a chunky piece of work; as you can see from the use of `get()` above, code that uses the future value looks very natural.

Multi-catch blocks Notice the use of a *multi-catch* block to catch either an exception of type `ExecutionException` (representing the exception that a callable might throw) or an `InterruptedException` (which a thread might throw in general. The `|` operator avoids the need to write multiple catch blocks if we wish to handle several exceptions in the same manner.

Shutting down the executor service It is necessary to use:

```
executor.shutdown();
```

to shut down the executor service at the end of program execution. If this command is not issued, the program will hang.

Do we observe store buffering? On my PC, I saw 2 instances of store buffering behaviour over 100,000 runs. On adding `Thread.yield()` at the start of each callable's `call()` method (which tells the scheduler to schedule a different thread), I saw instances of store buffering 47 times over 100,000 runs.

This "store buffering" could be due to x86 store buffering behaviour, or might be the result of compiler optimizations; looking at the compiled bytecode would be one step towards investigating this, though one would have to look at the machine code that is just-in-time compiled by the Java Virtual Machine to understand whether the compiler is reordering memory operations or not.

Using `volatile` to restore sequential consistency If we add the `volatile` qualifier to each of `x` and `y`, we no longer see any relaxed behaviour. This is because a memory operation issued to a `volatile` variable is guaranteed to be ordered before a memory operation that is issued later.

To summarize our detour into relaxed memory: the Java memory model is not sequentially consistent, and the effects of this can occasionally be seen. This can lead to faulty reasoning about protocols such as *double-checked locking* where one typically assumes sequential consistency when reasoning.

5 Overcoming the double-checked locking and size-tracking issues with *atomic references* and an *atomic integer*

Recall the two deficiencies with the almost-thread-safe hash set discussed in Section 3: the vulnerability of the double-checked locking pattern (due to Java's memory model) and the lack of synchronization on the `size` field. We show how these can be addressed using *atomic references* and an *atomic integer*, dealing with the latter first. The version of the hash set with these features is in the `superchargedhashset` package.

Counting with an `AtomicInteger` We can replace:

```
private int size;
```

with:

```
private AtomicInteger size;
```

to use the special `AtomicInteger` class to represent the size of the hash set. This class supports a number of integer operations, with the guarantee that they will appear to take place *atomically* to a client of the class.

the two operations relevant to us are:

- `int incrementAndGet();` – increments the value of the integer and returns its new value
- `int decrementAndGet();` – decrements the value of the integer and returns its new value

Calling `size.incrementAndGet()` and `size.decrementAndGet()` are the atomic analogues of executing `size++` and `size--`. These methods map to special "read-modify-write" hardware instructions that avoid the problems associated with the non-atomic `size++` and `size--` statements. At most one thread can perform a read-modify-write operation on one memory location at a single time: if multiple threads attempt such an operation on a common memory location, the execution of these operations is sequentialized.

It might seem like this sequentialization in the face of contention is just as bad as using a synchronized method on the hash set to increment and decrement size. However, the overhead of issuing a single atomic increment or decrement is lower than that of acquiring a lock. Still, contention can be an issue. Later, we will see how a *long adder* can be used to reduce contention.

To fully replace `size` with an atomic integer, we must use:

```
size = new AtomicInteger(0);
```

to make the value zero initially, and implement the `size()` method as follows:

```
public int size() {
  return size.get(); // the .get() call retrieves an int value from the counter
}
```

Using `AtomicReference`s for double-checked locking The issue with double-checked locking as presented in Section 3 is that operations on references in Java may not exhibit sequentially consistent behaviours. One solution to this involves making the references involved in double-checked locking `volatile`. We will look at a different (and in my view, cleaner) solution, using *atomic references*.

The `AtomicReference<T>` class is a generic class that is used to wrap references to objects of type `T`. An atomic reference is simply a reference to an object, but it has the special property that the reference can be set and retrieved in an atomic fashion. It is thus safe to use in fine-grained concurrent contexts, such as that of double-checked locking.

To handle the hash set buckets using atomic references, we can replace the `buckets` field, originally:

```
private final List<List<T>> buckets;
```

with:

```
private final List<AtomicReference<List<T>>> buckets;
```

Instead of setting each bucket to `null` initially, we initialize each atomic reference so that it does not hold a reference (i.e., each atomic reference holds `null`):

```
for(int i = 0; i < numBuckets; i++) {
  buckets.add(new AtomicReference<>());
}
```

We can now implement double-checked locking in `add` in a manner similar to that shown in Section 3, but using atomic references to ensure correct concurrent behaviour:

```
public boolean add(T item) {
  AtomicReference<List<T>> bucket = getBucket(item);
  if(bucket.get() == null) {
    synchronized(this) {
      bucket = getBucket(item);
      if(bucket.get() == null) {
        bucket.set(new LinkedList<>());
        buckets.set(getBucketIndex(item), bucket);
      }
    }
  }
  ...
}
```

The key point here is that all manipulation of bucket references goes through `AtomicReference` objects, which use low-level hardware instructions to ensure that the reference updates really are atomic. In particular, the situation where thread T1 observes a bucket to be `null` despite the fact that T2 has already created a bucket and issued an instruction to write the address of the new bucket to the `buckets` array is eliminated.

If you study the code in `superchargedhashset` you will see that this use of atomic references comes at a readability cost: some relatively intrusive changes to the original code are required to deal with the fact that each bucket is wrapped in an atomic reference, and it can be easy to get confused between a bucket and an atomic reference that refers to a bucket, especially when testing references against `null`.

6 Reader/writer locks

The code in `readerswritershashset` shows how to implement a version of the hash set with lower contention, using *reader/writer locks* and a *long adder*.

Reader/writer locks allow multiple threads to acquire a read lock, if no thread holds a write lock, and at most one thread to acquire a write lock, so long as no other thread holds any lock. This allows multiple concurrent readers, which is good for scenarios where reading is more common than writing.

A long adder provides much of the functionality of an atomic integer, for the case where one wishes to add to a quantity multiple times, but only read the quantity occasionally. The long adder uses additional memory to avoid threads having to always synchronise when making an update to the quantity being added to, by storing partial sums. When a thread reads the quantity, the partial sums need to be added together to give the required result.