Ian Kenny

February 4, 2022

# 5CCS2OSC Operating Systems and Concurrency

## Java synchronization

# Multithreading

In the previous lecture we saw that we can share data between threads, but that this sharing can lead to issues. We saw that two threads accessing a single variable (a counter) can leave the counter in an indeterminate state because the operation used on the counter is not atomic (the ++ operator). That particular operator consists of three separate operations.

Java provides a number of approaches that enable the programmer to avoid such problems. The solutions usually involve preventing multiple threads accessing a shared object 'at the same time'.

Remember, when we say 'at the same time' we mean 'concurrently' but not necessarily literally at the same time.

# Multithreading

The program on the next slide does the same thing as `MemoryDemoRunnable2` from the last lecture. If you recall, that demo showed *thread interference*. Therefore, `MemoryDemoRunnable3` also shows thread interference.

This is because both threads are using the same `Runnable` object hence the same variable `counter`.

The output will be similar to `MemoryDemoRunnable2` .

# MemoryDemoRunnable3

```
public class MemoryDemoRunnable3 implements Runnable {
   private int counter = 0;
   public void run() {
      for (int i = 0; i < 100_000; i++) {
         incCounter();
      }
      System.out.println(Thread.currentThread().getName()
          + " " + counter);
   }
   private void incCounter() {
      counter++;
   }
   public static void main(String[] args) {
      MemoryDemoRunnable3 runnable1 = new
          MemoryDemoRunnable3();
      Thread thread1 = new Thread(runnable1);
      Thread thread2 = new Thread(runnable1);
      thread1.start();
      thread2.start();
   }
}
```

Listing 1: MemoryDemoRunnable3.java

# Critical sections

The issue here is that this program contains what is known as a **critical section**.

A critical section is a sequence of instructions that only one thread at a time should be allowed to execute, otherwise we get incorrect or unpredictable behaviour.

We protect critical sections with the notion of **mutual exclusion**, which simply means that one thread at a time gets exclusive access to a critical section.

In general we protect such sections with **locks**.

# Java `synchronized`

We can fix the issue with the previous program with one deceptively simple change. We simply declare the method that manipulates the counter (in this example) as `synchronized`.

If we declare a method `synchronized`, the JVM will ensure that *only one thread at a time* can enter that method hence this defines a critical section that consists of the entire method.

This will ensure that a thread can't be interrupted by another thread while it manipulates the counter. In this case, since `counter++` is the only instruction in the method, it makes that instruction effectively atomic (but only in this method).

# SynchronizedMethodDemo2

```
public class SynchronizedMethodDemo2 implements Runnable {
   private int counter = 0;
   public void run() {
      for (int i = 0; i < 100_000; i++) {
         incCounter();
      }
      System.out.println("Final Counter: "+
          Thread.currentThread().getName() + " " +
          counter);
   }
   private synchronized void incCounter() {
      counter++;
   }
   public static void main(String[] args) {
      SynchronizedMethodDemo2 runnable1 = new
          SynchronizedMethodDemo2();
      Thread thread1 = new Thread(runnable1);
      Thread thread2 = new Thread(runnable1);

      thread1.start();
      thread2.start();
   }
}
```

Listing 2: SynchronizedMethodDemo2.java

# Java synchronized

With this program, at least one of the threads is guaranteed to reach the correct total, which is 200,000. The other thread might not.

One of the threads will generally reach the indexing limit in its loop before the counter has reached the final total. This is because, although we now do not have the threads interrupting each other in the middle of an instruction, they do still interrupt each other in terms of progress towards the final result.

We can think of this as the threads *cooperating* to get to the required total, rather than each one attempting to do that.

# Locks

A lock is a construct that enables the programmer to declare a critical section that can only be accessed by one thread at a time.

Just before entering the critical section the thread attempts to *acquire* the lock. If this is successful then the thread will enter the critical section.

When the thread exits the critical section it *releases* the lock.

If Thread A holds the lock for a particular critical section, and Thread B attempts to acquire the lock, Thread B will be blocked until Thread A releases the lock. If Thread A never releases the lock, Thread B will block forever.

# Intrinsic locks

Every Java object has a lock associated with it, called its *intrinsic lock*.

When we define a critical section with `synchronized` we are defining the critical section with respect to a particular object.

With the previous code example, the object that will be protected in its critical section is the object represented by `this`, i.e. the current object.

We do not have to specify that we are locking on `this` since that it is implicit, in this case.

# Intrinsic locks

With a method-level lock, like the one in the example above, we don't need to specify that we are locking on `this` because that is implicit.

When a thread enters the critical section it *acquires* the lock. When it exits the critical section it releases the lock.

Because the lock is obtained on an object (the `this` object in this case), this means that other threads will not be able to enter **any** `synchronized` method in that object. This makes sense because if it locked only the single method, then other methods could be called and *those* could suffer from thread interference.

# Intrinsic locks

A method-level lock locks the whole method, whether that is necessary or not.

Consider the program on the next slide.

Focus on the `addToCounter()` method since the whole program doesn't do anything new.

# SynchronizedMethodDemo3

```
public class SynchronizedMethodDemo3 implements Runnable {
   private int counter = 0;
   public void run() {
      for (int i = 0; i < 100_000; i++) {
         addToCounter(1);
      }
      System.out.println("Final Counter: "+
          Thread.currentThread().getName() + " " +
          counter);
   }
   private synchronized void addToCounter(int amount) {
      if (amount >= 0) {
         counter += amount;
      }
   }
   public static void main(String[] args) {
      SynchronizedMethodDemo3 runnable1 = new
          SynchronizedMethodDemo3();
      Thread thread1 = new Thread(runnable1);
      Thread thread2 = new Thread(runnable1);

      thread1.start();
      thread2.start();
   }
}
```

Listing 3: SynchronizedMethodDemo3.java

# SynchronizedMethodDemo3

The whole `addToCounter()` method is `synchronized`. This means that the `if` statement is also `synchronized` when there is no point to that. The parameter `amount` is local to each thread hence is not shared so doesn't need locking.

We can avoid this scenario by only locking the specific parts of a method that we need to using a `synchronized` *block*. This means we don't define the critical section as consisting of the entire method, but only a part of it.

(This program is intended to demonstrate the idea above. The fact is, in this particular program, the `if` statement would always be true anyway.)

# SynchronizedBlockDemo

```
public class SynchronizedBlockDemo implements Runnable {
   private int counter = 0;
   public void run() {
      for (int i = 0; i < 100_000; i++) {
         addToCounter(1);
      }
      System.out.println("Final Counter: "+
          Thread.currentThread().getName() + " " +
          counter);
   }
   private void addToCounter(int amount) {
      if (amount >= 0) {
         synchronized(this) {
            counter += amount;
         }
      }
   }
   public static void main(String[] args) {
      SynchronizedBlockDemo runnable1 = new
          SynchronizedBlockDemo();
      Thread thread1 = new Thread(runnable1);
      Thread thread2 = new Thread(runnable1);

      thread1.start();
      thread2.start();
   }
}
```

Listing 4: SynchronizedBlockDemo.java

# Intrinsic locks

In this example, the lock will not be acquired on the `this` object until the `if` has turned out to be true.

When using a `synchronized` block we have to specify which object we want to lock on.

In this case we specify `this` but, as we will see next, we can lock on other objects.

# Locking on other objects

It may be the case that we don't want to lock the current object but want to lock *another*, shared object.

Instead of a `synchronized` block synchronizing on `this` it can synchronize on another object.

This other object is likely to be an object that is shared between threads and is itself not synchronized, i.e. is not *thread safe*.

Consider the simple `Counter` class on the next slide.

# Counter

```
public class Counter {
   private int counter;
   public void incCounter() {
      counter++;
   }
   public int getCount() {
      return counter;
   }
}
```

Listing 5: Counter.java

# Thread safety

This class itself is not *thread safe*.

This means that it can't be shared between multiple threads with any guarantee that it will function correctly.

This is for the reason that we have discussed: the ++ operator is not atomic. Each thread could be interrupted while performing that operation leading to unpredictable results.

You might say, "well, just make it thread safe then by synchronizing the instruction `counter++`".

That might be the obvious solution but it may not be possible if we did not create this class and do not control its source code.

The program on the next slide demonstrates how a different class can, nevertheless, synchronize access to the counter.

# SynchronizedObjectDemo

```
public class SynchronizedObjectDemo implements Runnable {
   private Counter counter;
   public SynchronizedObjectDemo(Counter c) {
      counter = c;
   }
   public void run() {
      for (int i = 0; i < 100_000; i++) {
         synchronized(counter) {
            counter.incCounter();
         }
      }
      System.out.println("Final Counter: "+ Thread.currentThread().getName() +
" " + counter.getCount());
   }
   public static void main(String[] args) {
      Counter counter = new Counter();

      SynchronizedObjectDemo runnable1 = new SynchronizedObjectDemo(counter);
      Thread thread1 = new Thread(runnable1);
      SynchronizedObjectDemo runnable2 = new SynchronizedObjectDemo(counter);
      Thread thread2 = new Thread(runnable2);

      thread1.start();
      thread2.start();
   }
}
```

Listing 6: SynchronizedObjectDemo.java

# Thread safety

One potential problem with the approach to synchronizing on the `counter` object in the way shown on the previous slide is that it relies on *other* classes to remember to do the same. If one thread misuses the counter, then all threads could suffer.

Ideally, all classes that might be used to create shared objects that are **mutable**, i.e. changeable, should be designed as thread safe in the first place.

An object that is **immutable** is inherently thread safe since the state of the object can't be changed anyway.

# Visibility

We will now talk about an issue that we have ignored so far: that of when data becomes visible to different threads.

We might assume that if one thread has finished updating some variable then other threads will see that value. However, that is not necessarily the case.

# Memory levels

What we have discussed so far is that processes have a heap and a stack allocated by the OS. Threads share the heap of the process that created them but have their own stacks.

When a processes, hence threads, are running, however, there are other types of memory involved due to the operation of the CPU.

The CPU (or core) has **registers**. These are general purpose memory locations built into the CPU that the CPU uses for computations, etc.

CPU registers are essentially extremely-fast-to-access memory for the CPU.

# Memory levels

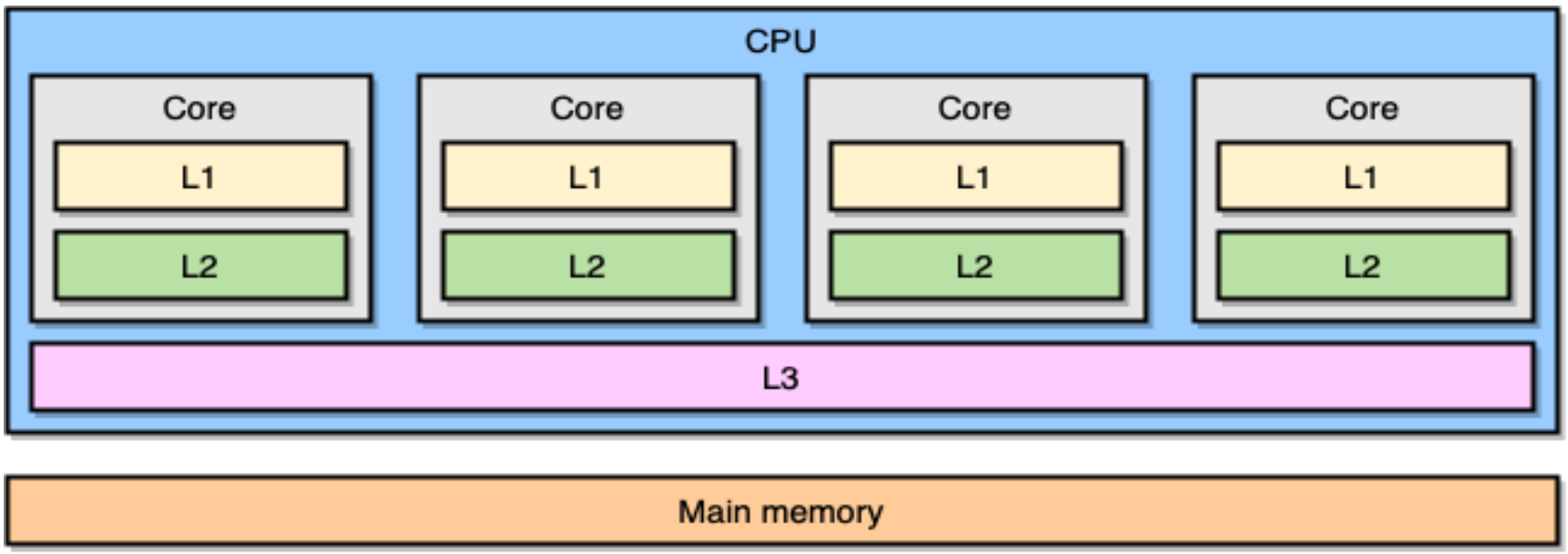
Modern CPUs also have a number of levels of **cache** memory. The number of caches depends on the model. However, there will usually be one to four levels of cache in a CPU. These caches are used to store frequently accessed data and instructions close to the CPU so that it can access them very quickly.

Each core of a CPU has its *own* registers and cache memory.

# Memory levels

Finally, there is the main RAM itself. This is where the heaps and the stacks are stored. In relative terms, accessing main RAM is a very slow operation compared to accessing registers and cache memory.

# Memory levels




source: https://teivah.medium.com/go-and-cpu-caches-af5d32cc5592

# Memory levels

This multi-layer arrangement of memory causes some problems that have had to be addressed otherwise applications simply wouldn't work.

The problem is that this architecture deliberately creates data redundancy and potential inconsistency. For example, a specific variable that has been copied to the registers or cache may have a different (perhaps more up to date) value than the 'original' in the main RAM: same variable, different values.

# Memory levels

Consider this scenario.

Thread A executing on Core1 has copied the value of a variable to its cache.

Thread B executing on Core2 has copied the same variable to its *own* cache.

Thread A copies the variable to Core1's registers, updates the variable and writes the new value back to its cache.

This new value for the variable is in Core1's cache and hence is not visible to Thread B which still has the old value of the variable in its own cache.

# Memory Flushing

This scenario is not an edge case. It is fundamental to the design of this memory architecture.

If we allowed this inconsistent state to persist, applications would very quickly start misbehaving.

CPUs have systems for *flushing* the caches (and registers, where applicable) to main memory.

This process involves making sure that the latest versions of all variables are flushed from the registers and caches through to the RAM.

Only when this has happened can we guarantee that all of the threads accessing some data item are seeing the same value.

# Memory Flushing

This issue becomes more pronounced in the case where threads are sharing an object that has instance variables.

The object is on the heap, hence in the main RAM, so each thread might make its own copies of the instance variables in its cache or registers to speed up its operations on those variables.

Thus, even if an object is shared, it might be in a different state for each thread.

# Visibility

The class on the next slide shows what is sometimes called an *exchanger* class that enables multiple threads to share access to some data, for example. This example also shows the *producer-consumer* pattern that we will return to.

The idea is that one or more threads will call the `addToBuffer()` method to add data to a buffer and one or more other threads will call the `takeFromBuffer()` method to take some data from the buffer.

The type `VisibilityBuffer` is just a buffer, i.e. an array of some kind. The `currentFill` variable contains the index of the buffer element that was last added, and the variable `newDataAvailable` is set to true when new data has been added and set to false when some data has been taken. The workings of this class don't really matter for our purposes here.

# Visibility

```
public class VisibilityBufferDemo {
   private int currentFill;
   private boolean newDataAvailable;
   private VisibilityBuffer buffer;

   public void addToBuffer(byte[] data) {
      buffer.addToBuffer(data, currentFill);
      newDataAvailable = true;
   }
   public byte[] takeFromBuffer(int nBytes) {
      newDataAvailable = false;
      return buffer.takeFromBuffer(nBytes, currentFill);
   }
   public boolean isNewDataAvailable() {
      return newDataAvailable;
   }
}
```

Listing 7: VisibilityBufferDemo.java

# Visibility

Consider the case when there are two threads sharing an object of this class.

As already stated, this means that each thread might make local copies of the three instance variables, so that it can speed up its own access to them. This could potentially lead to a situation in which the two threads have different values for each of the instance variables.

On the face of it, simply declaring critical sections in which access to these variables is restricted to one thread at a time will not help with this problem. This isn't about the threads interfering with each other, but about them caching instance variables locally and holding inconsistent values for them.

# `volatile`

Java provides us with a keyword to use with such variables: i.e. those that might be cached locally by each thread causing inconsistent data problems.

If we mark the variables as `volatile` then this is a signal that they need to be flushed to main memory when changed, and also read back from main memory when accessed.

In fact, no matter how many instance variables there are, if all we are trying to do is to make sure that the latest values are always flushed to the RAM and read from the RAM, then we only need to mark one of the variables as `volatile`.

This is due to Java's *volatile visibility guarantee*: if a variable marked as volatile is flushed to the RAM, then *all* of the variables visible to the thread will also be flushed.

# The visibility guarantee

There is also a visibility guarantee when it comes to `synchronized` blocks.

When a thread enters a `synchronized` block, all of the variables visible to the thread will be refreshed from main memory.

When the thread exits the `synchronized` block, all of the variables will be flushed back to main memory.

In the example on the next slide, the `volatile` modifier has been added to one of the instance variables.

# Visibility

```
public class VisibilityBufferDemo2 {
   private int currentFill;
   private volatile boolean newDataAvailable;
   private VisibilityBuffer buffer;

   public void addToBuffer(byte[] data) {
      buffer.addToBuffer(data, currentFill);
      newDataAvailable = true;
   }
   public byte[] takeFromBuffer(int nBytes) {
      newDataAvailable = false;
      return buffer.takeFromBuffer(nBytes, currentFill);
   }
   public boolean isNewDataAvailable() {
      return newDataAvailable;
   }
}
```

Listing 8: VisibilityBufferDemo2.java

# Instruction reordering

There is one other topic to discuss here before moving on.

The JVM, the Java compiler, the 'HotSpot' compiler and the CPU can all decide to **reorder** the instructions in your programs.

There are multiple reasons why this might happen. For example, one reason might be to take advantage of CPU pipelining to parallelise independent instructions (i.e. instructions that do not affect each other) to increase instruction throughput.

On the next slide we see an amendment to the previous demo that might result from a reordering of the statements in the method `addToBuffer()`. As far as the JVM, compiler, HotSpot and CPU are concerned, the statements in that method are candidates for reordering because they seem to be independent.

# Visibility

```
public class VisibilityBufferDemo3 {
   private int currentFill;
   private volatile boolean newDataAvailable;
   private VisibilityBuffer buffer;

   public void addToBuffer(byte[] data) {
      newDataAvailable = true;
      buffer.addToBuffer(data, currentFill);
   }
   public byte[] takeFromBuffer(int nBytes) {
      newDataAvailable = false;
      return buffer.takeFromBuffer(nBytes, currentFill);
   }
   public boolean isNewDataAvailable() {
      return newDataAvailable;
   }
}
```

Listing 9: VisibilityBufferDemo3.java

# Instruction reordering

Consider what happens when the following statement is executed:

```
newDataAvailable = true;
```

Since `newDataAvailable` is a `volatile` variable this change of value will cause it to be flushed to main memory along with *all the other instance variables*.

This means that new data is apparently available, according to the value of `newDataAvailable`, when in fact that isn't true, at this precise instant, because the data is not updated until the next line.

# Happens before

This brings us to another guarantee offered by Java: the *volatile happens before* guarantee.

This guarantees that if a statement occurs before a change to a volatile variable (in the program as written) then it will remain before. It will not be reordered. In the previous example, this would prevent the reordering.

Indeed, this guarantee also applies to `synchronized` blocks too involving non-volatile variables.

Instructions will not be reordered in such a way as to move a statement out of a `synchronized` block to after the `synchronized` block, and statements that are before the `synchronized` block will not be moved to after the `synchronized` block.