Synchronicity:
Atomicity, Visibility, Ordering

Issues leading to shared state problems...

Atomicity
Lock it down! When you want to access something in shared state in any way, obtain a mutual exclusion lock on it. Basically, decide which instructions must have indivisible effects. This is the most straightforward approach but also leads to issues with efficiency and the potential for deadlock.

Visibility
Under what conditions are the effects of one thread visible to another? Basically, if there are writes, are they seen via reads. You might want to make sure all changes are seen by all threads as soon as they occur.

Ordering
In what possible orders could a group of statements be executed. What might the Java compiler do to optimize things in terms of reads and writes in sequences of assignment statements.
**Atomicity Recap and Extension**

We’ve discussed some ways to ensure atomicity in Java with intrinsic locks and *synchronized*.

This can be useful if concurrent writes are possible or concurrent reads and even just one write.

However, in Java using the *synchronized* keyword around a block of code does something in addition to creating a mutual exclusion lock. Releasing the intrinsic lock also essentially forces any pending writes back to memory to be performed. Acquiring the intrinsic lock also essentially forces a reloading of any fields used in the block from memory.

**Visibility**

If you surround all blocks of code that read and write a value with the appropriate intrinsic lock, then visibility is not an issue due to the forced reads from and writes to memory mentioned on the previous slide.

However, if you just want to make sure that the most recent value is being read and used, guaranteeing visibility of writes is all you need.
Volatile

Java provides us with the ability to mark variables as volatile. This means that any writes to it will go out to shared memory right away, and any reads from it will be actual reads from shared memory. Compiler optimizations will not be allowed to work only in a local copy of the variable.

Consider the following lines of code inside of a loop:

\[ A = B; \quad B = A + 1; \]

Should there be any way for \( A > B \) to ever be true, even to another thread?

Let’s try it on the linuxlab.csic.umd.edu machines…

Torn Reads

Note that with volatility issues, we might be dealing with a stale value, but it would always be a value that had “existed” at some point in our program’s logic.

I’ve hinted at the notion that a lack of synchronization could lead to values being used that never “existed” as far as our program was concerned. One very practical example of this is the use of 64-bit values on a 32-bit architecture with the current Java memory model because while it requires smaller reads and writes to be atomic, it allows reads and writes of 64-bit values to be split at the 32-bit level.

This means you could get the high bits from one value and the low bits from another, creating a new value that never “existed” in your program before that point. This means that even if you don’t mind a stale value, you still need to do something to prevent this situation, and setting such a variable as volatile will do that.
**Limitation #1**

One example of a limitation to this is arrays.

An array reference could be set to `volatile` so that if a new array is allocated and the reference set to point at it, that would be visible to other threads right away.

However, there is no way to set the *elements* of the array as volatile.

More generally, any time that an object reference is set to be `volatile` that is just handling the visibility of the reference, not of the contents of the object itself.

---

**Limitation #2**

Also, the use of `volatile` does not preserve any atomicity of blocks of code or prevent changes to the order in which code is executed.

We can look at another example on one of the `linuxlab` machines that shows how unguarded execution gives the wrong results, `synchronized` execution provides the correct results, but marking the shared variables as `volatile` doesn’t help at all.

So, ensuring visibility of changes can be of use, but it is not going to solve mutual exclusion issues.
**When to use?**

Due to the overhead, you don’t want to make everything volatile in your program.

Since it doesn’t deal with data races, you don’t want to use it in those situations.

Sometimes it is a useful thing if locking isn’t needed to read the variable and it’s also true that:

- the writes don’t need to read the current value.
- you **know** that only one thread is ever writing to the variable.

---

**Ordering issues...**

These can be divided into two sub-cases; within-thread and between.

Within-thread ordering should essentially be the same as if things were being done on a single-threaded system, but an optimizing compiler might re-order lines of code for which it sees no dependencies. This might be fine and valid if it were done in a sequential program, but could lead to unusual interactions in a multithreaded system. We’ve seen examples of this earlier.

Between-thread ordering is unpredictable except for places where explicit relative orderings are set up by the use of synchronization techniques such as intrinsic locks or other blocks. For this reason it will be useful to explore scenarios where we know that certain program events happen before other specific program events.