Introduction to Locks

Intrinsic Locks

Atomic-looking operations

Resources created for sequential code make certain assumptions, a large one being that a method is an atomic thing.

Consider how \texttt{i++} might appear at a low level:

- Load \texttt{i} into register R1
- Increment register R1
- Store register R1 back to \texttt{i}

What happens if \texttt{i} is a shared value and the above gets swapped out of the CPU in the middle of its execution?
**Programming Invariants**

When implementing classes, it is common to assume that methods are atomic.

A class invariant might not be upheld within a method, but by the time the method completes its work, the invariant is still true.

With concurrent execution, an object’s method could be paused in the middle, leaving that object in an invalid state as far as the rest of the world is concerned.

---

**Safe access of a thread-unsafe resource.**

Consider the Java class `LinkedList<T>` and its `add()` and `poll()` methods.

In sequential programs a `LinkedList` object’s `add()` or `poll()` would always complete before that object could be used to invoke anything else.

In concurrent programs, a shared `LinkedList` object could have multiple threads invoking either of these methods and those calls could be swapped out of the CPU in the middle of their execution.

The results are unpredictable!
**Divide and Conquer**

Imagine taking a range of numbers that you wanted to check for primality and partitioning them among several threads, each of which does the following to its part of the question to add primes to a `LinkedList` object called `listOfPrimes`:

```java
for (int i=rangeStart; i<=rangeEnd; i++) {
    if (isPrime(i)) {
        listOfPrimes.add(i);
    }
}
```

Since that `LinkedList` is not thread-safe, the contents of `listOfPrimes` is unpredictable.

**Locks**

We can associate locks with things such as objects and following a programming paradigm where we always obtain a lock on an object before we invoke and operating that access it.

This would essentially make each method call atomic.

We could do this at a more granular level and restrict our use of locks for when the method might cause an invariant to be untrue in the middle, or when the method is accessing a value whose invariant might be untrue in the middle of some other method call.

There are many locking mechanisms. We will start with intrinsic locks.
**Intrinsic Locks**
Every object in Java has an intrinsic lock (sometimes called a monitor lock) with it.

The syntax contains an implied lock and unlock for it. The body of the synchronized block will not be entered until the lock has been acquired. The lock will be released on exiting the body of that block (even if you do a return from the middle of it).

```java
for (int i=rangeStart; i<=rangeEnd; i++) {
    if (isPrime(i)) {
        synchronized(listOfPrimes) {
            listOfPrimes.add(i);
        }
    }
}
```

**Intrinsic Locks Limitation**
One limitation on this type of lock is that there is no way to simply attempt to acquire a lock. If you ask for a lock and can not acquire it, then you will block until you do.
**Class-Level Locks**

If you wanted to have atomic methods in a class, you could mark each of them with the `synchronized` keyword. This has the same general effect as surrounding the body of the method with `synchronized(this)`.

Note that since intrinsic locks are what is called “reentrant” you could have a synchronized method call another synchronized method and there won’t be a problem with deadlock. Once a thread acquires an intrinsic lock, it can obtain additional holds on it. It will keep track of the number of times it has acquired it and will not release it to the world until it releases its hold on it that many times.

---

**No need to lock immutable objects**

In reality, locking is only an issue with mutable objects (or shared primitives) and not with immutable objects.

With an immutable object, the only time that values change are when the constructor is called. The constructor has to be called when you first create the object. If the object is to be shared, it has to be created before the threads that share it are launched. Therefore, not even the constructor is something that will cause problems if it is swapped out of the CPU in the middle of its execution.