Lab on
Behavior Models and Verification

Java PathFinder
Memory models
Double locking and memory models

- Weakening requirements as to ordering of instructions due to performance leads to deeply hidden problems
- Usually caused by absence of sequential-ordering semantics
Single-threaded version

class Foo {

    private Helper helper;
    public Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
        }

        return helper;
    }
}

}
// Correct but possibly expensive
// multithreaded version

class Foo {

    private Helper helper;

    public synchronized Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
        }

        return helper;
    }
}

// Broken multithreaded version
// "Double-Checked Locking" idiom

class Foo {
    private Helper helper;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }
}
Why this is broken?

- Intuitively, this algorithm seems like an efficient solution.
- However, if:
  1. Thread A notices that the value is not initialized, so it obtains the lock and begins to initialize the value.
  2. Due to the semantics of some programming languages, the code generated by the compiler is allowed to update the shared variable to point to a partially constructed object before A has finished performing the initialization. For example, in Java if a call to a constructor has been inlined then the shared variable may immediately be updated once the storage has been allocated but before the inlined constructor initializes the object.[5]
  3. Thread B notices that the shared variable has been initialized (or so it appears), and returns its value. Because thread B believes the value is already initialized, it does not acquire the lock. If B uses the object before all of the initialization done by A is seen by B (either because A has not finished initializing it or because some of the initialized values in the object have not yet percolated to the memory B uses (cache coherence)), the program will likely crash.
So...

// Works with acquire/release semantics for volatile in Java 1.5 and later
// Broken under Java 1.4 and earlier semantics for volatile

class Foo {
    private volatile Helper helper;

    public Helper getHelper() {
        Helper result = helper;
        if (result == null) {
            synchronized(this) {
                result = helper;
                if (result == null) {
                    helper = result = new Helper();
                }
            }
        }
        return result;
    }
}
// Correct lazy initialization in Java

class Foo {
    private static class HelperHolder {
        public static final Helper helper = new Helper();
    }

    public static Helper getHelper() {
        return HelperHolder.helper;
    }
}