Data Races are Pure Evil: A Simple Java-based Illustration

Ganesh Gopalakrishnan, School of Computing, University of Utah, Salt Lake City

Main Idea: Data races are egregious errors that are as serious as out-of-bounds accesses. The presence of a data race renders a program incorrect. Yet, there are popular parallel programming books that contain examples that have data races in them. The pattern found in these examples is: one thread sets a flag while another thread spin-waits on this flag. The assumed *happens-before ordering* in such programs can be broken by compiler optimizations. This assignment illustrates these ideas in a very concrete manner, showing that these racing "patterns" must really be avoided (they truly are anti-patterns).

The assignment consists of two parts: (1) offering two Java threads where one sets the request bit (shared global), waits for the acknowledge (shared global) to be set, and then reset the request and wait for the acknowledge to be reset, and iterate this N times. (2) if request/acknowledge are not declared `volatile`, the compiler can reorder and this iteration deadlocks. (3) the takeaway is that the happens-before order between threads and within threads does not create the requisite handshake order.

Concepts Covered: The fact that data races arise when a conflicting access (read/write unsynchronized with a write) is driven home. The students understand what *happens-before* (logical order) means. This truly helps them understand modern language-level support for concurrency, including modern C++ that is playing a hugely important role in disciplined parallel programming (as evidenced by the popularity of the Threading Building Blocks – TBB – library). An excellent book based on TBB and patterns becomes approachable.

Targeted Students: Students who know Java can do this exercise, and the experimental component (to discover the number of iterations, N) helps them see how rare failures can be (N has to be high – e.g., 7,000 – and then the code almost surely hangs).

In what contexts used before: This was assigned to an UG class I am teaching now (10 students). The class is small and quite focused, but the students took pleasure in seeing the code hang. I also showed them how to JIT the Java program (of less than 30 lines) and see the assembly code, and the **fence** instruction introduced corresponding to a `volatile` instruction.

What prerequisites assumed: The students may be given a Xerox of Chapter 16 of Brian Goetz's book "Java Concurrency in Practice," and this is a self-contained reading of about 16 pages.

Strengths/Weaknesses: Strong points: hands-on of a slippery topic; seeing the dangers of races first-hand. This topic seldom receives mention in books on Parallel Programming. They can be then taught that the `clang` compiler now supports race-checking via a compilation flag option, encouraging them to use this flag in their work. Weaknesses: not all aspects of the Java Memory Model covered.

Variations of Interest: They can be asked to rewrite the same code using C++11 atomics. They can be asked to explore how it relates to the double-checked locking pattern (also in Goetz, Chapter 16); this pattern is hugely important, and lead to modern concurrency concepts, as explained in Bill Pugh's website. They can be taught to stay away from the widespread misconception that `C` volatiles and Java volatiles are the same (they are not). They can further explore this topic in Scott Meyer's book *Effective Modern C++*.