acmqueue

You Don't Know Jack about Shared Variables or Memory Models

Data races are evil

Sarita V. Adve, University of Illinois at Urbana-Champaign
Hans-J. Boehm, HP Laboratories

A Google search for "Threads are evil" generates 18,000 hits, but threads—evil or not—are ubiquitous. Almost all of the processes running on a modern Windows PC use them. Software threads are typically how programmers get machines with multiple cores to work together to solve problems faster. And often they are what allow user interfaces to remain responsive while the application performs a background calculation.

Threads are multiple programs running at the same time but sharing variables. Typically, every thread can access all of the application's memory. Shared variables are either the core strength of threads or the root of their evil, depending on your perspective. They allow threads to communicate easily and quickly, but they also make it possible for threads to get in each other's way.

Although shared variables are at the core of most programs, even experts are often confused about the rules for using them. Consider the following simple example.

INCREMENTING A COUNTER

To implement a function `incr` that increments a counter `x`, your first attempt might be

```
void incr()
{
   x++;
}
```

Many would immediately object that this isn't guaranteed to produce the correct answer when called by multiple threads. The statement `x++` is equivalent to `x=x+1`, which amounts to three steps: (1) getting the value of `x`; (2) adding one; (3) writing the result back to `x`. In the unlikely case that two threads coincidentally perform these in lockstep, they will both read the same value, both add one to it, and then both write the same value, incrementing `x` by only one instead of two. A call to `incr()` doesn't behave *atomically*; it is visible to the user that it is composed of different steps. (*Atomicity* means different things to different communities; our use is called *isolation* by database folks.)

We might address the problem by using a *mutex*, which can be locked by only one thread at a time:

```
void incr()
{
mtx.lock();
```

```
x++;
mtx.unlock();
}

In Java, this might look like

void incr()
{
    synchronized(mtx) {
        x++;
    }
}

or perhaps just

synchronized void incr()
{
    x++;
}
```

Those would all work correctly, but mutex calls can be slow, so the result may run slower than desired.

What if we're concerned only about getting an approximate count? What if we just leave off the mutex, and settle for some inaccuracy? What could go wrong?

To begin with, we observed that some actual code incrementing such a counter in two threads without a mutex routinely missed about half the counts, probably a result of unfortunate timing caused by communication between the processors' caches. It could be worse. A thread could do nothing but call `incr()` once, loading the value zero from `x` at the beginning, get suspended for a long time, and then write back one just before the program terminates. This would result in a final count of one, no matter what the other threads did.

Those are the cases that are less surprising and easier to explain. The final count can also be too high. Consider a case in which the count is bigger than a machine word. To avoid dealing with binary numbers, assume we have a decimal machine in which each word holds three digits, and the counter `x` can hold six digits. The compiler translates `x++` to something like

```
tmp_hi = x_hi;
tmp_lo = x_lo;
(tmp_hi, tmp_lo)++;
x_hi = tmp_hi;
x_lo = tmp_lo;
```

where `tmp_lo` and `tmp_hi` are machine registers, and the increment operation in the middle would really involve several machine instructions. Now assume that `x` is 999 (i.e., `x_hi = 0`, and `x_lo =`

999), and two threads, a blue and a red one, each increment x as follows (remember that each thread has its own copy of the machine registers tmp_hi and tmp_lo):

```
tmp_hi = x_hi;
tmp_lo = x_lo;
(tmp_hi, tmp_lo)++;    tmp_hi = 1, tmp_lo = 0
x_hi = tmp_hi;         x_hi = 1, x_lo = 999, x = 1999
          x++;         red runs all steps
                       x_hi = 2, x_lo = 0, x = 2000
x_lo = tmp_lo;         x_hi = 2, x_lo = 0
```

This shows how execution of the two threads interleave: first the blue thread runs almost to completion; then the red thread runs all at once to completion; finally the blue thread runs its last step. The result is that we incremented 999 twice to get 2000. This is difficult to explain to a programmer who doesn't understand precisely how the code is being compiled.

The fundamental problem is that multiple threads were accessing x at the same time, without proper locking or other synchronization to make sure that one occurred after the other. This situation is called a *data race*—which really is evil! We'll get back to avoiding data races without locks later.

ANOTHER RACY EXAMPLE

We've only begun to see the problems caused by data races. Here's an example commonly tried in real code. One thread initializes a piece of data (say, x) and sets a flag (call it done) when it finishes. Any thread that later reads x first waits for the done flag, as in figure 1. What could possibly go wrong?

This code may work reliably with a "dumb" compiler, but any "clever" optimizing compiler is likely to break it. When the compiler sees the loop, it is likely to observe that done is not modified in the loop (i.e., it is "loop-invariant"). Thus, it gets to assume that done does not change in the loop.

Of course, this assumption isn't actually correct for our example, but the compiler gets to make it anyway, for two reasons: (1) compilers were traditionally designed to compile sequential, not multithreaded code; and (2) because, as we'll see, even modern multithreaded languages continue to allow this, for good reason.

FIGURE 1

Waiting on a Flag

blue thread	other threads
`x = …;` `done = true;`	`while (!done) {}` `… = x;`

Thus, the loop is likely to be transformed to

```
tmp = done; while (!tmp) {}
```

or maybe even

```
tmp = done; if (!tmp) while (true) {}
```

In either case, if `done` is not already set when the red thread starts, the red thread is guaranteed to enter an infinite loop.

Assume we have a "dumb" compiler that doesn't perform such transformations and compiles the code exactly as written. Depending on the hardware, this code can still fail.

The problem this time is that the *hardware* may optimize the blue thread. Nearly all processor architectures allow stores to memory to be saved in a buffer visible only to that processor core before writing them to memory visible to other processor cores.[2] Some, such as the ARM chip that's probably in your smartphone, allow the stores to become visible to other processor cores in a different order. On such a processor the blue thread's write to `done` may become visible to the red thread, running on another core, *before* the blue thread's write to `x`. Thus, the red thread may see `done` set to `true`, and the loop may terminate before it can retrieve the proper value of `x`. Thus, when the red thread accesses `x`, it may still get the uninitialized value.

Unlike the original problem of reading `done` once outside the loop, this problem will occur very rarely, and may well be missed during testing.

Again the core problem here is that although the `done` flag is intended to prevent simultaneous accesses to `x`, it can itself be simultaneously accessed by both threads. *And data races are evil!*

BITS AND BYTES

So far, we've talked only about data races in which two threads access exactly the same variable, or object field, at the same time. That has not always been the only concern. According to some older standards, when you declare two small fields `b1` and `b2` next to each other, for example, then updating `b1` could be implemented with the following steps:

1. Load the machine word containing both `b1` and `b2` into a machine register.
2. Update the `b1` piece in the machine register.
3. Store the register back to the location from which it was loaded.

Unfortunately, if another thread updates `b2` just before the last step, then that update is overwritten by the last step and effectively lost. If both fields were initially zero, and one thread executed `b1 = 1`, while the other executed `b2 = 1`, `b2` could still be zero when they both finished. Although the original program was well behaved and had no data races, the compiler added an implicit update to `b2` that initiated a data race.

This kind of data-race insertion has been clearly disallowed in Java for a long time. Both the recently published C++11 standard and the almost finished C1X standard disallow it. We know of no Java implementations with such problems, nor do modern C and C++ compilers generally exhibit precisely this problem. Unfortunately, many do introduce data races under certain obscure, unlikely, and unpredictable conditions. This problem will disappear as C++11 and C1X become widely

supported.

For C and C++, the story for bit-fields is slightly more complicated. We'll discuss that more, later.

AND THE REAL RULES ARE ...

The simplest view of threads, and the one we started with, is that a multithreaded program is executed by interleaving steps from each thread. Logically the computer executes a step from one thread, then picks another thread, or possibly the same one, executes its next step, and so on. This is a *sequentially consistent* execution.

As already shown, real machines and compilers sometimes result in non-sequentially-consistent executions: for example, when the assignment to a variable and a `done` flag are made visible to other threads out of order. Sequential consistency, however, is critical in understanding the behavior of real shared variables, for two reasons:

• Essentially all modern languages (Java, C++11, C1X) do in fact promise sequential consistency *for programs without data races*. This guarantee is normally violated by a few low-level language features—notably, Java's `lazySet()` and C++11 and C1X's explicit `memory_order...` specifications, which are easy to avoid (with the possible exception of OpenMP's atomic directive) and which we'll mostly ignore here. Most programmers will also want to ignore these features.

• So far we've been a bit imprecise about what constitutes a data race. Since this has now become a critical part of our programming rules, we can make it more precise as follows: two memory operations *conflict* if they access the same memory location and at least one of the accesses is a write. For our purposes, a *memory location* is a unit of memory that is separately updatable. Normally every scalar (unstructured) variable or field occupies its own memory location; each can be independently updated. Contiguous sequences of C or C++ bit fields, however, normally share a single location; updating one potentially interferes with the others.

Two conflicting data operations form a *data race* if they are from different threads and can be executed "at the same time." But when is this possible? Clearly that depends on how shared variables behave, which we're trying to define.

We break this circularity by considering only *sequentially consistent* executions: two conflicting operations in a sequentially consistent execution execute at the same time, if one executes immediately after the other in that execution's interleaving. Now we can say that a program is *data-race-free* if none of its sequentially consistent executions has a data race.

Here we've defined a *data* race in terms of *data* operations explicitly to exclude *synchronization* operations such as locking and unlocking a mutex. Two operations on the same mutex don't introduce a data race if they appear next to each other in the interleaving. Indeed, they couldn't usefully control simultaneous accesses if concurrent accesses to the mutexes were disallowed.

Thus, the basic programming model is:

• Write code such that data races are impossible, assuming that the implementation follows sequential consistency rules.

• The implementation then guarantees sequential consistency for such code (assuming that the low-level features previously mentioned are avoided).

This is very different from promising full sequential consistency; our earlier examples are not guaranteed to work as expected, since they all have data races. Nonetheless, when writing a program, there is no need to think explicitly about compiler or hardware memory reordering; we

can still reason entirely in terms of sequential consistency, as long as we follow the rules and avoid data races.

This has some consequences that often surprise programmers. Consider the program in figure 2, where x and y are initially false. When reasoning about whether this has a data race, we observe that there is no sequentially consistent execution (i.e., no interleaving of thread steps) in which either assignment is executed. Thus, there are no pairs of conflicting operations, and hence certainly no data races.

WORK AT A HIGHER LEVEL

So far, our programming model still has us thinking of interleaving thread execution at the memory-access or instruction level. Data races are defined in terms of accesses to memory locations, and sequential consistency is defined in terms of interleaving indivisible steps, which are effectively machine instructions. This is an entirely new complication. A programmer writing sequential code doesn't need to know about the granularity of machine instructions and whether memory is accessed a byte or a word at a time.

Fortunately, once we insist on data-race-free programs, this issue disappears. A very useful side effect of our model is that a thread's synchronization-free regions appear indivisible or atomic. Thus, although our model is defined in terms of memory locations and individual steps, there is really no way to tell what those steps and memory locations are without introducing data races.

More generally, data-race-free programs always behave as though they were interleaved only at synchronization operations, such as mutex lock/unlock operations. If this were not the case, synchronization-free code sections from different threads would appear to interleave as in figures 3 and 4.

In the first case (figure 3), no such interleaved code sections contain conflicting operations, and each section effectively operates on its own separate set of memory locations. The instruction interleaving is entirely equivalent to one in which these code sections execute one after the other as shown in the figure, with the only visible interleaving at synchronization operations.

In the second case (figure 4), two code sections contain conflicting operations on the same memory location. In this case there is an alternate interleaving in which the conflicting operations appear next to each other, and a data race is effectively exhibited, as shown. Thus, this cannot happen for data-race-free programs.

FIGURE 2

Is There a Data Race if Initially x = y = False?

blue thread	red thread
`if (x)y = true`	`if (y)x = true`

This means that any section of code containing no synchronization operations behaves as though it executes atomically (i.e., all at once) without being affected by other threads and without another thread being able to see any variable values occurring in the middle of that code section. Thus, insisting on data-race-free programs has some pleasant consequences:

- We no longer care whether memory is updated a byte or a word at a time. Properly written code can't tell any more than it could for sequential code.
- Library calls that don't use internal synchronization behave as if they execute in a single step. The intermediate states can't be seen by another thread. Thus, such libraries can continue to specify only the overall effect of making a call, not which intermediate values might be taken by variables. Of course, that's what we've been doing all along, but it really makes sense only with data-race freedom.
- Reasoning about multithreaded programs is still hard, but without data races, it's not as hard as people often claim. In particular, we don't have to care about all possible ways of interleaving threads' instructions. At most, we care about the interleavings of synchronization-free regions.

Of course, all of these properties require that the program be data-race-free. Today, detecting and avoiding data-race bugs can be far from easy. Later we discuss recent progress toward making it easier.

In particular, to ensure data-race-freedom, it suffices to ensure that synchronization-free code sections that run at the same time don't both write, or read and write, the same variables. Thus, we can prune a significant number of instruction-level interleavings that need to be explored for this purpose.

FIGURE 3

Conflict-Free Interleaving Is Not Observable

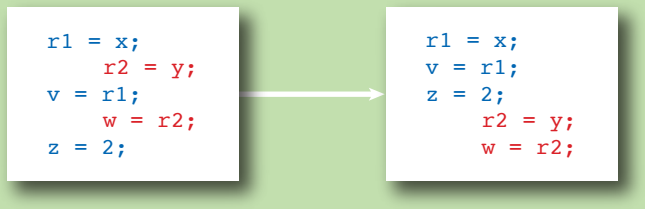
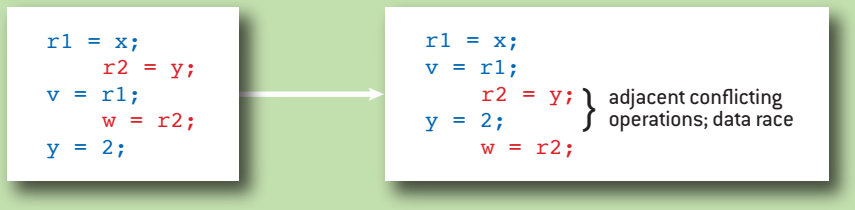


FIGURE 4

Interleaving with Conflict Is Data Race



Libraries can be (and generally are) designed so that the data-race detection problem can be partitioned cleanly between library and client code and, again, the programmer does not need to be concerned with low-level instruction interleaving. In the client code, we reason about data races at the level of logical objects, not memory locations. When deciding whether it is safe to call two library routines simultaneously, we need to make sure only that they don't both access the same *object*, or if they do, that neither access modifies the *object*. It is the library's responsibility to make sure that accesses to logically distinct objects don't introduce a data race as a result of unprotected accesses to some internal hidden memory locations. Similarly, it is the library's responsibility to make sure that reading an object doesn't introduce an internal write to the object that can create a data race.

With the data-race-free approach, library-implemented container data types can behave as built-in integers or pointers; the programmer doesn't need to be concerned with what goes on inside. As long as two different threads don't access the same *container* at the same time, or they're both read accesses, the implementation remains hidden.

Again, while all of these properties simplify reasoning about parallel code, they assume that the library writer and the client are responsible for obeying the prescribed disciplines.

BUT WHAT IF LOCKS ARE TOO SLOW?

The most common way to avoid data races is to use mutexes to ensure mutual exclusion between code sections accessing the same variable. In certain contexts, other synchronization mechanisms such as OpenMP's barriers are more appropriate. Experience has shown, however, that such mechanisms are insufficient in a few cases. Mutexes don't work well with signal or interrupt handlers, and they often involve significant overhead, even if they have started to get faster on recent processors.

Unfortunately, many environments, such as Posix threads, have not provided any real alternatives—so people cheat. Pthread code commonly contains data races, which are typically claimed to be "benign." Some of these are outright bugs, in that the code, as currently compiled, will fail with small probability. The rest often risk getting "miscompiled" by compilers that either outright assume there are no data races[4] and are hence misled by bad assumptions or that just produce some of the surprising effects previously discussed.

To escape this dilemma, most modern programming languages provide a way to declare *synchronization* variables. These behave as ordinary variables, but since accesses are considered to be *synchronization* operations, not *data* operations, synchronization variables can be safely accessed from multiple threads without creating a data race. In Java, a `volatile int` is an integer that can be accessed concurrently from multiple threads. In C++11, you would write `atomic<int>` instead (`volatile` means something subtly different in C and C++).

Compilers treat synchronization variables specially, so our basic programming model is preserved. If there are no data races, threads still behave as though they execute in an interleaved fashion. Accessing a synchronization variable is a synchronization operation, however; code sequences extending across such accesses no longer appear indivisible.

Synchronization variables are sometimes the right tool for very simple shared data, such as the `done` flag in figure 1. The only data race here is on the `done` flag, so simply declaring that as a synchronization variable fixes the problem.

Remember, however, that synchronization variables are difficult to use for complex data structures, since there is no easy way to make multiple updates to a data structure in one atomic operation. Synchronization variables are not replacements for mutexes.

In cases such as that shown in figure 1, synchronization variables often avoid most of the locking overhead. Since they are still too expensive, both C++11 and Java provide some explicit experts-only mechanisms that allow you to relax the interleaving-based model, as mentioned before. Unlike programming with data races, it is possible to write correct code that uses these mechanisms, but our experience is that few people actually get this right. Our hope is that future hardware will reduce the need for it—and hardware is already getting better at this.

REAL LANGUAGES

Most real languages fit our basic model. C++11 and the upcoming C1X standard provide exactly this model. Data races have "undefined behavior"; they are errors in the same sense as an out-of-bounds array access. This is often referred to as *catch-fire* semantics for data races (though we don't know of any cases in which machines have actually caught fire as the result of a data race).

Although catch-fire semantics are sometimes still controversial, they are hardly new. The Ada 83 and 1995 Posix thread specifications are less precise, but took basically the same position.

C++11 and C1X provide synchronization variables as `atomic<t>` and `_Atomic(t)`, respectively. In addition to reading and writing these variables, they support some simple indivisible compound operations; for example, incrementing a synchronization (atomic) variable with the **"++"** operator is an indivisible operation.

The situation for managed languages is more complex, mostly because of the security requirements they add to support untrusted code. Java fully supports our programming model, but it also, with only limited success, attempts to provide some guarantees for programs with data races. Although data races are not officially errors, it is now clear that we can't precisely define what programs with data races actually mean.[8] Data races remain evil.

TOWARD A FUTURE WITHOUT EVIL?

We have discussed how the absence of data races leads to a simple programming model supported by common languages. There simply does not appear to be any other reasonable alternative.[1] Unfortunately, one sticky problem remains: *guaranteeing* data-race-freedom is still difficult. Large programs almost always contain bugs, and often those bugs are data races. Today's popular languages do not provide any usable semantics to such programs, making debugging difficult.

Looking forward, it is imperative that we develop automated techniques that detect or eliminate data races. Indeed, there is significant recent progress on several fronts: dynamic precise detection of data races;[5,6] hardware support to raise an exception on a data race;[7] and language-based annotations to eliminate data races from programs by design.[3] These techniques guarantee that the considered execution or program has *no* data race (allowing the use of the simple model), but they still require more research to be commercially viable. Commercial products that detect data races have begun to appear (e.g., Intel Inspector), and although they do not guarantee data-race-freedom, they are a big step in the right direction. We are optimistic that one way or another, we will (we must!) conquer evil (data races) in the near future.

REFERENCES

1. Adve, S. V., Boehm, H.-J. 2010. Memory models: a case for rethinking parallel languages and hardware. *Communications of the ACM* 53(8): 90–101.
2. Adve, S. V., Gharachorloo, K. 1996. Shared memory consistency models: a tutorial. *IEEE Computer* 29(12): 66–76.
3. Bocchinoetal, R. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.
4. Boehm, H.-J. 2011. How to miscompile programs with "benign" data races. In *HotPar (Hot Topics in Parallelism)*.
5. Elmas, T., Qadeer, S., Tasiran, S. 2007. Goldilocks: a race and transaction-aware Java runtime. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*: 245–255.
6. Flanagan, C., Freund, S. 2009. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the Conference on Programming Language Design and Implementation*.
7. Lucia, B., Ceze, L., Strauss, K., Qadeer, S., Boehm, H.-J. 2010. Conflict exceptions: providing simple concurrent language semantics with precise hardware exceptions. In *Proceedings of the International Symposium on Computer Architecture*.
8. Sevcik, J., Aspinall, D. 2008. On validity of program transformations in the Java memory model. In *European Conference on Object-oriented Programming*: 27–51.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

SARITA V. ADVE is a professor in the department of computer science at the University of Illinois at Urbana-Champaign. Her research interests are in computer architecture and systems, parallel computing, and power and reliability-aware systems. Among her contributions, she has co-developed the memory models for the C++ and Java programming languages, based on her early work on data-race-free models. Adve is an ACM Fellow. She received Ph.D. and M.S. degrees in computer science from the University of Wisconsin, Madison, and a B.Tech. degree in electrical engineering from the Indian Institute of Technology.

HANS-J. BOEHM is a research manager at HP Labs. He is probably best known as the primary author of a commonly used garbage collection library. Experiences with threads in that project eventually led him to initiate the effort to properly define threads and shared variables in C++11. He is an ACM Distinguished Scientist. He was awarded the PLDI 2003 most influential paper award and the SIGPLAN 2006 Distinguished Service Award. He holds a B.S. degree from the University of Washington, and M.S. and Ph.D. degrees from Cornell University.