What is this thing you call "thread safe"?

Article • 10/19/2009 • 6 minutes to read

Caveat: I am not an expert on multi-threading programming. In fact, I wouldn't even say that I am *competent* at it. My whole career, I've needed to write code to spin up a secondary worker thread probably less than half a dozen times. So take everything I say on the subject with some skepticism.

A question I'm frequently asked: "*is this code **thread safe**?* " To answer the question, clearly we need to know what "thread safe" means.

But before we get into that, there's something I want to clear up first. A question I am less frequently asked is "*Eric, why does Michelle Pfeiffer always look so good in photographs?* " To help answer this pressing question, I consulted Wikipedia :

> "*A **photogenic** subject is a subject that usually appears physically attractive or striking in photographs.*"

Why does Michelle Pfeiffer always look so good in photographs? ***Because she's photogenic***. Obviously.

Well, I'm glad we've cleared up that mystery, but I seem to have wandered somehwat from the subject at hand. Wikipedia is just as helpful in defining thread safety :

> "*A piece of code is **thread-safe** if it functions correctly during simultaneous execution by multiple threads.*"

As with photogenicity, this is obvious question-begging. When we ask "is this code *thread safe*?" all we are really asking is "is this code *correct when called in a particular manner*?" So how do we determine if the code is correct? **We haven't actually explained anything here.**

Wikipedia goes on:

> "*In particular, it must satisfy the need for multiple threads to access the same shared data, ...*"

This seems fair; this scenario is almost always what people mean when they talk about thread safety. But then:

> "*...and the need for a shared piece of data to be accessed by only one thread at any given time.*"

Now we're talking about techniques for *creating* thread safety, not *defining* what thread safety means. Locking data so that it can only be accessed by one thread at a time is just one possible technique for creating thread safety; it is not itself the definition of thread safety.

My point is not that the definition is *wrong*; as informal definitions of thread safety go, this one is not terrible. Rather, my point is that the definition indicates that the concept itself is *completely vague* and essentially means nothing more than "behaves correctly in some situations". Therefore, when I'm asked "is this code thread safe?" I always have to push back and ask "what are the *exact threading scenarios* you are concerned about?" and "exactly what is *correct behaviour* of the object in every one of those scenarios?"

Communication problems arise when people with different answers to those questions try to communicate about thread safety. For example, suppose I told you that I have a "threadsafe mutable queue" that you can use in your program. You then cheerfully write the following code that runs on one thread while another thread is busy adding and removing items from the mutable queue:

if (!queue.IsEmpty) Console.WriteLine(queue.Peek());

Your code then crashes when the Peek throws a QueueEmptyException. What is going on here? I said this thing was thread safe, and yet your code is crashing in a multi-threaded scenario.

When I said "the queue is threadsafe" I meant that the queue maintains its internal state consistently no matter what sequence of *individual* operations are happening on other threads. But I did not mean that you can use my queue in any scenario that requires *logical consistency maintained across multiple operations in a sequence*. In short, my opinion of "correct behaviour" and your opinion of the same differed because what we thought of as the relevant scenario was completely different. I care only about not crashing, but you care about being able to reason logically about the information returned from each method call.

In this example, you and I are probably talking about different kinds of thread safety. Thread safety of mutable data structures is usually all about ensuring that the operations on the shared data always operate on the **most up-to-date** state of the shared data as it mutates, even if that means that a particular combination of operations appears to be **logically inconsistent**, as in our example above. Thread safety of immutable data structures is all about ensuring that use of the data across all operations is **logically consistent**, at the expense of the fact that you're looking at an immutable snapshot that might be **out-of-date**.

The problem here is that the choice about whether to access the first element or not is based on "stale" data. Designing a truly thread-safe mutable data structure in a world where *nothing is allowed to be stale* can be very difficult. Consider what you'd have to do in order to make the "Peek" operation above actually threadsafe. You'd need a new method:

if (!queue.Peek(out first)) Console.WriteLine(first);

Is this "thread safe"? It certainly seems better. But what if after the Peek, a different thread dequeues the queue? Now you're not crashing, but you've changed the behaviour of the previous program considerably. In the previous program, if, after the test there was a dequeue on another thread that changed what the first element was, then you'd either crash or print out the up-to-date first element in the queue. Now you're printing out a *stale* first element. Is that *correct*? Not if we *always* want to operate on up-to-date data!

But wait a moment -- actually, the *previous* version of the code had this problem as well. What if the dequeue on the other thread happened *after* the call to Peek succeeded but *before* the Console.WriteLine call executed? Again, you could be printing out stale data.

What if you want to ensure that you are always printing out up-to-date data? What you really need to make this threadsafe is:

queue.DoSomethingToHead(first=>{Console.WriteLine(first);});

Now the queue author and the queue user agree on what the relevant scenarios are, so this is truly threadsafe. Right?

Except... there could be something super-complicated in that delegate. What if whatever is in the delegate happens to cause an event that triggers code to run on another thread, which in turn causes some queue operation to run, which in turn blocks in such a manner that we've produced a deadlock? Is a deadlock "correct behaviour"? And if not, is this method truly "safe"?

Yuck.

By now you take my point I'm sure. As I pointed out earlier, it is unhelpful to say that a building or a hunk of code is "secure" without somehow communicating **which** threats the utilized security mechanism are and are not proof against. Similarly, **it is unhelpful to say that code is "thread safe" without somehow communicating what undesirable behaviors the utilized thread safety mechanisms do and do not prevent.** "Thread safety" is nothing more nor less than a code contract, like any other code contract. You agree to talk to an object in a particular manner, and it agrees to give you correct results if you do so; working out exactly what that manner is, and what the correct responses are, is a potentially tough problem.

****

(*) Yes, I'm aware that if I think something on Wikipedia is wrong, I can change it. There are two reasons why I should not do so. First, as I've already stated I'm not an expert in this area; I leave it to the experts to sort out amongst themselves what the right thing to say here is. And second, my point is not that the Wikipedia page is wrong, but rather that it illustrates that the term itself is vague by nature.

Comments

- **Steve Bjorg**

October 19, 2009

It can get even more complicated. In our implementation of the work-stealing queue (based on the excellent work of Danny Hendler, et al), we have thread-safety in one method only, namely TrySteal(). The other methods, Push() and TryPop() are by-design always called from the same thread. Calling them from different threads would be a violation of the contract. However, TrySteal() can be called by a number of different threads and not corrupt the state of the queue. What's even more amazing is that the implementation achieves this without any locks or loops!

For those curious, the C# implementation of LockFreeWorkStealingDeque is available under Apache 2 in MindTouch Dream.

- **Vish**

October 19, 2009

Wouldn't putting access to data structures that mutate in a critical section(lock) be better? Thread safe (simultaneous access by many threads) should only be for code that dont handle shared data structures...

- **Aaargh!**

October 19, 2009

In this particular example, the queue is perfectly thread safe, the code using the queue isn't. The queue is only responsible for itself, the fact that you're doing multi-threaded

operations on the queue implies that any information you get from the queue can be out of date a few moments later.
Your example code does not take into account the fact that the queue can be modified at any time by other threads, and your code is thus not thread safe. If you need multiple calls to the queue to be consistent, you have an unguarded critical path in _your code_ not in the queue's, and it's your responsibility to guard it. In this case it could be solved by synchronizing on the queue.

- **Aaron Friel**

October 19, 2009
When you reference Wikipedia, especially by link as you have done, it's especially important to use the "Permanent Link" button in the bottom left. That allows you to permanently link to the same version of the article that you saw.
Otherwise what you're linking to could be vandalized (popular articles frequently are, though they are corrected that much more quickly,) and your users could see something other than what you intended, or problems that you point out could be corrected, and your point is moot. :)

- **Frank Booth**

October 19, 2009
> As with photogenicity, this is obvious question-begging.
How is this question-begging?
Question begging does not mean "to avoid providing the answer to a question" it means that a new question needs to be answered as a result of the answer to the original question.
That is all.
No, it doesn't mean that. Let me give you another example of question begging. Suppose I asked "why is diamond hard but butter is soft?" and you answered "diamond and butter are both made out of atoms; the atoms of diamonds are hard and the atoms of butter are soft." You would have begged the question; your answer to my question "why are some things hard and some things soft" is "because some things are made out of stuff that is hard and some things are made out of stuff that is soft" -- that is, you've avoided answering the question by providing an "explanation" that itself cannot be understood without answering the original question -- namely, why is some stuff hard and some stuff soft? This pseudo-explanation has no predictive power; it doesn't tell us anything new, it just circles back on itself.
A non-question-begging answer would be "diamond and butter are both made of atoms; the atoms of diamond are all identical and arranged in a stable, rigid lattice where every point in the lattice is reinforced by a strong bond to four other points. The atoms of butter are a disorganized collection of many different atoms that hold weakly

to each other. It takes only a small force to disrupt the loose arrangement of butter atoms but a very large force to disrupt the strong arrangement of diamond atoms."
Now, this explanation does *raise* more questions. It raises questions like "why are some lattices strong and some weak?" and "why are some objects composed of many different kinds of atoms, and some composed of just one atom?" Question-begging is not the act of raising more questions. Every explanation raises more questions. This particular explanation is testable, and has predictive power; we can investigate the hardness or softness of other substances, and make predictions about what sorts of atomic structures they will have -- or, vice versa, we can look at an atomic structure and try to figure out from it how hard the substance will be.
My point here is that "because she's photogenic" is question-begging. Why does she look so good? Because she's photogenic. Why is she photogenic? Because she looks so good. We have learned nothing about photogenicity (or Ms. Pfeiffer).
Similarly, if you ask "why is this code thread-safe?" and the answer is "because it can be correctly called on multiple threads", you've begged the question. Why is it thread-safe? Because it's correct. Why is it correct? Because it's thread-safe. Again, we have learned nothing about the nature of thread safety.
-- Eric

- **Aaargh!**

October 19, 2009
> Question begging does not mean "to avoid providing the answer to a question"
Correct, it does not mean that.
> it means that a new question needs to be answered as a result of the answer to the original question.
It doesn't mean that either.
The way it's used in the article is correct, see for more information:
http://en.wikipedia.org/wiki/Begging_the_question

- **Dan Diplo**

October 19, 2009
This raises the bigger question - is Michelle Pfeiffer thread safe?

- **Anonymous**

October 19, 2009
The comment has been removed

- **Nym**

October 19, 2009
Hey throw some shared memory in there for yet more fun.

- **Anonymous**

October 19, 2009

The comment has been removed

- **Anonymous**

October 19, 2009

The comment has been removed

- **Anonymous**

October 19, 2009

The comment has been removed

- **Anonymous**

October 19, 2009

The comment has been removed

- **JonB**

October 19, 2009

"Thread safe" is a bit of a C throw-back, where you can determine if a particular function is thread safe if it isn't dependent on any stored state. However an object could only be deemed thread safe in the same manner if it didn't maintain any state, which wouldn't be much of an object.

- **Alun Harford**

October 19, 2009

"Thread-safe" is not at all vague.

It's a statement that if each thread's rely condition is satisfied then that thread satisfies its guarantee condition and each thread's guarantee condition implies the others' rely conditions.

It's only vague when we choose to rely on intuition rather than mathematics when we describe a system.

- **Joe**

October 19, 2009

If something is thread safe, say an API, it can be used by several threads at the same time without the caller having to think about it. This is as old as computers. It is not something you can wave your hand at and say is vague and then ignore. If something is not thread safe, you should only use it from a single thread, or it is likely to crash, hang, or just silently go wrong. If you want to develop in a thread safe manner, and you do, you must think defensively, and at very least do a global lock so in your code, things are only happening single threaded. Later on, carefully break stuff up into separate locks, then maybe use read/write locks. Yes it's harder, and there are many traps, but there is many off the shelf tools and techniques, plus most of computer history, to help you.

That or develop in a language that does it for you, but that will always be more limited than a manual environment. Just like with cars and gears, all racing cars are manual and you can't get the same miles per gallon without manual. Your fooling yourself if you think automatic is as good.

- **Carl Daniel**

October 20, 2009

I have to agree with Eric on this one. While there are good definitions of "thread safe", there is no universal agreement on which of the many good definitions is "correct". Correctness, of course, depends on the circumstances. As a result, the term "thread safe" with no further qualification is vague at best, dangerously misleading at worst.
Thread safety is somewhat analogous to exception safety. The C++ community has settled on a multi-tier defintion of "exception safe" - I would propse that a similar family of thread safety guarantees would be a useful addition to the dialog. The mathematical defintion that Alun provided above sounds like a good candiate for being "the strong guarantee". At another extreme, a class that's documented as providing a single method that can be invoked by less than 3 threads under specific circumstances would be an example of a very weak guarantee.
The problem with the very stong guarantee that Alun provided - just like the stong exception safety guarantee in C++ - is that most cases don't require a guarantee that strong, and generally speaking, providing such a strong guarantee is more difficult and less efficient (naturally, there are always exceptions).

- **Adam V**

October 20, 2009

I'm hoping (fingers crossed!) that the inclusion of the "Michelle Pfeiffer" tag means that Ms. Pfeiffer will make additional appearances in future examples. I for one welcome our new photogenic example overlords.

- **Anonymous**

October 20, 2009

The comment has been removed

- **eff Five**

October 21, 2009

Eric:
If I said the that "this code is thread un-safe" would it be less vague than "is this code thread safe"?

- **Larry**

October 22, 2009

I don't think the assertion of thread safety tells us nothing, or even almost nothing. I've always interpreted "thread-safe" to assert the *minimal* contract of thread-safety: instance separation (no mutable static state) and method/property atomicity; operations on a shared instance in multiple threads will behave the same way and leave the instance in a the same state at sequence points as the same operations called in an unpredictable, arbitrary order on one thread. In other words, if one thread calls obj.A () and another thread calls obj.B (), then those calls will be equivalent to calling obj.A (); obj.B (); or obj.B (); obj.A () in one thread. I never have to lock an thread-safe object to do a single call; I have to lock it (or do something special) only when I want to be sure of the *order* of calls.
As you note, this is not a very strong contract, and I'm not at all surprised that some people might believe "thread safety" asserts a stronger contract. But minimal thread-safety is still a non-trivial contract.

- **Grant Husbands**

October 22, 2009
I agree with Larry, and I'll add that a lack of thread-safety doesn't just affect ordering. Code that isn't thread-safe being called from multiple threads easily results in state corruption, seemingly arbitrary exceptions and memory corruption (not in all languages). When you call code that's simply "thread-safe", you can expect behaviour approximately matching that of the contract.
If there's a thread-safe queue with TryPop, I know I can use that from multiple threads without breaking anything and that each call will get a uniquely-added entry (or false,null). However, Peek will obviously always be meaningless in such circumstances.
This does all make me think, though; there may be some good mileage in having thread-safe libraries go out of their way to induce failure in multi-threaded callers, in an appropriate debug mode. Many queue methods, for example, could delay a little, in hope of returning an unexpected null to a caller or returning the same result simultaneously to two callers (where Peek is involved).
Hopefully, more people will start using multi-threading idioms other than shared-state-with-locks, and much of the problem will be reduced.

- **Filini**

October 22, 2009
@Dan: "This raises the bigger question - is Michelle Pfeiffer thread safe? "
Michelle Pfeiffer can do whatever she wants to my threads :)
(sorry for the OT, I couldn't resist)

- **cpun**

October 23, 2009
I would define a method to be thread-safe if atleast one of these is true:

1. it is a pure function
2. For functions that read and/or write mutable shared state, if the class invariants can NOT be violated as a consequence of the number, timing and/or interleaving of threads executing this very same method concurrently.

Eric, I would say that you are taking on a whole different animal in your example by talking about thread-safety at the level of conducting multiple operations on an object whose state can be modified by different threads. Thread-safe constructs as they are used now do NOT compose (EVER). That is to say in:
if (!queue.IsEmpty) Console.WriteLine(queue.Peek());
I might take and release a lock in IsEmpty and do the same in Peek(), but the outcome of composing these into the above statement is NOT thread safe. This is, I would argue, not a knock on the thread-safety of the member functions of Queue at all. You could easily have a TryPeekAndPrint() call on Queue that does the above operation in a thread-safe manner by composing all the other operations.
This is pretty much where Software Transactional Memory comes in since you want to make sure that the outcome of doing IsEmpty and Peek is not influenced by other threads or transient behavior.

- **jerome**

October 30, 2009
There's exactly the same remark in Java Concurrency in Practice, section 2.1...

- **Anonymous**

November 4, 2009
The comment has been removed

- **Anonymous**

November 16, 2009
The comment has been removed

- **0x60de**

December 1, 2009
I think nobody should ever attest to an "object" being thread-safe. you may have stateful behaviour or may have stateless method calls (each method requires all parameters to be operate on and does not maintain state).
The most we could attest to is an "operation" is thread-safe. Once you follow this definition, you may alternatively attest that "each" operations in Queue class is thread-safe.

- **Anonymous**

December 4, 2009
The comment has been removed

- **Shawn**

July 14, 2011

If you updated wikipedia you'ed need to re-write your opening analogy. You wouldn't want to be referenceing stale data.