Even in computing, simultaneity is relative

devblogs.microsoft.com/oldnewthing/20040903-00

September 3, 2004

Raymond Chen

Einstein discovered that simultaneity is relative. This is also true of computing.

People will ask, "Is it okay to do X on one thread and Y on another thread simultaneously?" Here are some examples:

- X = "close a handle" and Y = "use that handle".
- X = "call UnregisterWaitForSingleObject on a handle", Y = "call UnregisterWaitForSingleObject on that same handle".

You can answer this question knowing nothing about the internal behavior of those operations. All you need to know are some physics and the answers to much simpler questions about what is valid sequential code.

Let's do a thought experiment with simultaneity.

Since simultaneity is relative, any code that does X and Y simultaneously can be observed to have performed X before Y or Y before X, depending on your frame of reference. That's how the universe works.

So if it were okay to do them simultaneously, then it must also be okay to do them one after the other, since they **do occur one after the other** if you walk past the computer in the correct direction.

Is it okay to use a handle after closing it? Is it okay to unregister a wait event twice?

The answer to both questions is "No," and therefore it isn't okay to do them simultaneously either.

If you don't like using physics to solve this problem, you can also do it from a purely technical perspective.

Invoking a function is not an atomic operation. You prepare the parameters, you call the entry point, the function does some work, it returns. Even if you somehow manage to get both threads to reach the function entry point simultaneously (even though as we know from

physics there is no such thing as true simultaneity), there's always the possibility that one thread will get pre-empted immediately after the "call" instruction has transferred control to the first instruction of the target function, while the other thread continues to completion. After the second thread runs to completion, the pre-empted thread gets scheduled and begins execution of the function body.

Under this situation, you effectively called the two functions one after the other, despite all your efforts to call them simultaneously. Since you can't prevent this scenario from occurring, you have to code with the possibility that it might actually happen.

Hopefully this second explanation will satisfy the people who don't believe in the power of physics. Personally, I prefer using physics.

Raymond Chen

Follow