Customers not getting the widgets they paid for if they click too fast -or- In C#, the += operator is not merely not guaranteed to be atomic, it is guaranteed not to be atomic

devblogs.microsoft.com/oldnewthing/20140814-00

August 14, 2014

Raymond Chen

In the C# language, operation/assignment such as `+=` are explicitly *not* atomic. But you already knew this, at least for properties.

Recall that properties are syntactic sugar for method calls. A property declaration

```
string Name { get { ... } set { ... } }
```

is internally converted to the equivalent of

```
string get_Name() { ... }
void set_Name(string value) { ... }
```

Accessing a property is similarly transformed.

```
// o.Name = "fred";
o.put_Name("fred");
// x = o.Name;
x = o.get_Name();
```

Note that the only operations you can provide for properties are `get` and `set`. There is no way of customizing any other operations, like `+=`. Therefore, if you write

```
o.Name += ", Jr.";
```

the compiler has no choice but to convert it to

```
o.put_Name(o.get_Name() + ", Jr.");
```

If all you have is a hammer, everything needs to be converted to a nail.

Since the read and write are explicitly decoupled, there is naturally a race condition here. The underlying property may change value in between the time you read the old value and the time you write the new value.

But there are extra subtleties here. Let's dig in.

The rule for operators like `+=` are spelled out in *Section 14.3.2: Compound Assignment*:

> [T]he operation is evaluated as x = x op y, except that x is evaluated only once.

(There is some discussion of what "evaluated only once" means, but that's not important here.)

The subtleties lurking in that one sentence are revealed when you see how that sentence interacts with other rules in the language.

Now, you might say, "Sure, it's not atomic, but my program is single-threaded, so this should never affect me."

Actually, you can get bitten by this even in single-threaded programs. Let's try it:

```
class Program
{
 static int x = 0;
 static int f()
 {
  x = x + 10;
  return 1;
 }
 public static void Main()
 {
  x += f();
  System.Console.WriteLine(x);
 }
}
```

What does this program print?

You might naïvely think that it prints `11` because `x` is incremented by 1 by `Main` and incremented by 10 in `f`.

But it actually prints 1.

What happened here?

Recall that C# uses strict left-to-right evaluation order. Therefore, the order of operations in the evaluation of `x += f()` is

1. Rewrite as `x = x + f()`.
2. Evaluate both sides of the `=` operator, left to right.
    a. Left hand side of assignment: Find the variable `x`.
    b. Right hand side of assignment:
        i. Evaluate both sides of the `+` operator, left to right.
            1. Evaluate `x`.
            2. Evaluate `f()`.
        ii. Add together the results of steps 2b(i)1 and 2b(i)2.
3. Take the result of step 2b(ii) and assign it to the variable `x` found in step 2a.

The thing to notice is that a lot of things can happen between step 2b(i)1 (evaluating the old value of `x`), and step 3 (assigning the final result to `x`). Specifically, we shoved a whole function call in there: `f()`.

In our case, the function `f()` *also modifies* `x`. That modification takes place after we already captured the value of `x` in step 2b(i)1. When we get around to adding the values in step 2b(ii), we don't realize that the values are out of date.

Let's step through this evaluation in our example.

1. Rewrite as `x = x + f()`.
2. Evaluate both sides of the `=` operator, left to right.
    a. Left hand side of assignment: Find the variable `x`.
    b. Right hand side of assignment:
        i. Evaluate both sides of the `+` operator, left to right.
            1. Evaluate `x`. The result is 0.
            2. Evaluate `f()`. The result is 1. It also happens that `x` is modified as a side-effect.
        ii. Add together the results of steps 2b(i)1 and 2b(i)2. In this case, 0 + 1 = 1.
3. Take the result of step 2b and assign it to the variable `x` found in step 2a. In this case, assign 1 to `x`.

The modification to `x` that took place in `f` was clobbered by the assignment operation that completed the `+=` sequence. And this behavior is not just in some weird "undefined behavior" corner of the language specification. The language specification explicitly *requires* this behavior.

Now, you might say, "Okay, I see your point, but this is clearly an unrealistic example, because nobody would write code this weird."

Maybe you don't intentionally write code this weird, but you can do it accidentally. And this is particularly true if you are using the new `await` keyword, because an `await` means, "Hey, like, put my function on hold and do other stuff for a while. When the thing I'm awaiting is ready, then resume execution of my function." And that "do other stuff for a while" might change `x`.

Suppose that you have a button in your application called *Buy More*. When the user clicks it, they can buy more widgets. Let's assume that the `BuyMoreAsync` function return the number of items bought. (If the user cancels the purchase it returns zero.)

```
// Suppose the user starts with 100 widgets.
async void BuyMoreButton_OnClick()
{
 TotalWidgets += await BuyMoreAsync();
 Inventory.Text = string.Format("You have {0} widgets.",
                               TotalWidgets);
}
async Task<int> BuyMoreAsync()
{
 int quantity = QuickPurchase.IsChecked ? 1
                                        : await GetQuantityAsync();
 if (quantity != 0) {
  if (await DebitAccountAsync(quantity * PricePerWidget)) {
   return quantity;
  }
 }
 return 0; // user bought no items
}
```

You receive a bug report that you track back to the fact that `TotalWidgets` does not match the number of widgets purchased. It affects only people who checked the *quick purchase* box, and only people purchasing from overseas.

Here's what is going on.

The user clicks the *Buy More* button, and they have *Quick Purchase* enabled. The *BuyMore-Async* function tries to debit the account for the price of one widget.

While waiting for the server to process the transaction, the user gets impatient and clicks *Buy More* a second time. This triggers a second task to debit the account for the price of one widget.

Okay, so you now have two tasks running, each processing one of the clicks. In theory, the worst case is that the user accidentally buys two widgets, but in practice…

The first `DebitAccountAsync` task completes, and `BuyMoreAsync` returns 1, which is then added to the value of `TotalWidgets` at the time the button was clicked, as we discussed above. At the time the button was clicked the first time, the number of widgets was 100, so the total number of widgets is now 101.

The second `DebitAccountAsync` task completes, and `BuyMoreAsync` returns 1, which is then added to the value of `TotalWidgets` at the time the button was clicked, as we discussed above. When the button was clicked the second time, the number of widgets was *still 100*. We set the total widget count to `100 + 1 = 101`.

Result: The user paid for two widgets but got only one.

The fix for this is to explicitly move waiting for the purchase to complete outside of the compound assignment.

```
int quantity = await BuyMoreAsync();
TotalWidgets += quantity;
```

Now, the `await` is outside the compound assignment so that the value of `TotalWidgets` is not captured prematurely. When the purchase completes, we update `TotalWidgets` without interruption from any async operations.

(You probably also should fix the program so it disables the *Buy More* button while a transaction is in progress, to avoid the *impatient user ends up making an accidental double purchase* problem. The above fix merely gets rid of the *user pays for two items and gets only one* problem.)

Like closing around the loop control variable, this is the sort of subtle change that should be well-commented so that somebody doesn't "fix" it in a well-intentioned but misguided attempt to remove unnecessary variables. The purpose of the variable is not to break an expression into two but rather to force a particular order of evaluation: You want to to finish the purchase operation before starting to update the widget count.

Raymond Chen

Follow