Lock-free algorithms: The one-time initialization

devblogs.microsoft.com/oldnewthing/20110407-00

April 7, 2011

Raymond Chen

A special case of the singleton constructor is simply lazy-initializing a bunch of variables. In a single-threaded application you can do something like this:

```
// suppose that any valid values for a and b stipulate that
// a ≥ 0 and b ≥ a. Therefore, -1 is never a valid value,
// and we use it to mean "not yet initialized".
int a = -1, b = -1;
void LazyInitialize()
{
 if (a != -1) return; // initialized already
 a = calculate_nominal_a();
 b = calculate_nominal_b();
 // Adjust the values to conform to our constraint.
 a = max(0, a);
 b = max(a, b);
}
```

This works fine in a single-threaded program, but if the program is multi-threaded, then two threads might end up trying to lazy-initialize the variables, and there are race conditions which can result in one thread using values before they have been initialized:

Thread 1	Thread 2
`if (a != -1)` [not taken]	
`a = calculate_nominal_a();` // returns 2	
	`if (a != -1) return;` // premature return!

Observe that if the first thread is pre-empted after the value for `a` is initially set, the second thread will think that everything is initialized and may end up using an uninitialized `b`.

“Aha,” you say, “that’s easy to fix. Instead of `a`, I’ll just use `b` to tell if initialization is complete.”

```
void LazyInitialize()
{
 if (b != -1) return; // initialized already (test b, not a)
 a = calculate_nominal_a();
 b = calculate_nominal_b();
 // Adjust the values to conform to our constraint.
 a = max(0, a);
 b = max(a, b);
}
```

This still suffers from a race condition:

Thread 1	Thread 2
`if (b != -1)` [not taken]	
`a = calculate_nominal_a();` // returns 2	
`b = calculate_nominal_b();` // returns 1	
	`if (b != -1) return;` // premature return!

“I can fix that too. I’ll just compute the values of `a` and `b` in local variables, and update the globals only after the final values have been computed. That way, the second thread won’t see partially-calculated values.”

```
void LazyInitialize()
{
 if (b != -1) return; // initialized already
 // perform all calculations in temporary variables first
 int temp_a = calculate_nominal_a();
 int temp_b = calculate_nominal_b();
 // Adjust the values to conform to our constraint.
 temp_a = max(0, temp_a);
 temp_b = max(temp_a, temp_b);
 // make the temporary values permanent
 a = temp_a;
 b = temp_b;
}
```

Nearly there, but there is *still* a race condition:

Thread 1	Thread 2
`if (b != -1)` [not taken]	
`temp_a = calculate_nominal_a();` // returns 2	
`temp_b = calculate_nominal_b();` // returns 1	
`temp_a = max(0, temp_a);` // temp_a = 2	
`temp_b = max(temp_a, temp_b);` // temp_b = 2	
`a = temp_a;` // store issued to memory	
`b = temp_b;` // store issued to memory	
store of `b` completes to memory	
	`if (b != -1) return;` // premature return!
store of `a` completes to memory	

There is no guarantee that the assignment `b = 2` will become visible to other processors after the assignment `a = 2`. Even though the store operations are issued in that order, the memory management circuitry might complete the memory operations in the opposite order, resulting in Thread 2 seeing `a = -1` and `b = 2`.

To prevent this from happening, the store to `b` must be performed with Release semantics, indicating that all prior memory stores must complete before the store to `b` can be made visible to other processors.

```
void LazyInitialize()
{
 if (b != -1) return; // initialized already
 // perform all calculations in temporary variables first
 int temp_a = calculate_nominal_a();
 int temp_b = calculate_nominal_b();
 // Adjust the values to conform to our constraint.
 temp_a = max(0, temp_a);
 temp_b = max(temp_a, temp_b);
 // make the temporary values permanent
 a = temp_a;
 // since we use "b" as our indication that
 // initialization is complete, we must store it last,
 // and we must use release semantics.
 InterlockedCompareExchangeRelease(
    reinterpret_cast<LONG*>&b, temp_b, -1);
}
```

If you look at the final result, you see that this is pretty much a re-derivation of the singleton initialization pattern: Do a bunch of calculations off to the side, then publish the result with a single `InterlockedCompareExchangeRelease` operation.

The general pattern is therefore

```
void LazyInitializePattern()
{
 if (global_signal_variable != sentinel_value) return;
 ... calculate values into local variables ...
 globalvariable1 = temp_variable1;
 globalvariable2 = temp_variable2;
 ...
 globalvariableN = temp_variableN;
 // publish the signal variable last, and with release
 // semantics to ensure earlier values are visible as well
 InterlockedCompareExchangeRelease(
    reinterpret_cast<LONG*>&global_signal_variable,
    temp_signal_variable, sentinel_value);
}
```

If this all is too much for you (and given some of the subtlety of double-check-locking that I messed up the first time through, it's clearly too much for me), you can let the Windows kernel team do the thinking and use the one-time initialization functions, which encapsulate all of this logic. (My pal Doron called out the one-time initialization functions a while back.) Version 4 of the .NET Framework has corresponding functionality in the `Lazy<T>` class.

Exercise: What hidden assumptions are being made about the functions `calculate_nominal_a` and `calculate_nominal_b`?

Exercise: What are the consequences if we use `InterlockedExchange` instead of `InterlockedCompareExchangeRelease`?

Exercise: In the final version of `LazyInitialize`, are the variables `temp_a` and `temp_b` really necessary, or are they just leftovers from previous attempts at fixing the race condition?

Exercise: What changes (if any) are necessary to the above pattern if the global variables are pointers? Floating point variables?

Update: See discussion below between Niall and Anon regarding the need for acquire semantics on the initial read.

Raymond Chen

Follow