C++ scoped static initialization is not thread-safe, on purpose!

devblogs.microsoft.com/oldnewthing/20040308-00

March 8, 2004

Raymond Chen

[**Note**: After this article was written, the C++ standard has been revised. Starting in C++11, scoped static initialization is now thread-safe, but it comes with a cost: Reentrancy now invokes undefined behavior.]

The rule for static variables at block scope (as opposed to static variables with global scope) is that they are initialized the first time execution reaches their declaration.

Find the race condition:

```
int ComputeSomething()
{
  static int cachedResult = ComputeSomethingSlowly();
  return cachedResult;
}
```

The intent of this code is to compute something expensive the first time the function is called, and then cache the result to be returned by future calls to the function.

A variation on this basic technique is is advocated by this web site to avoid the "static initialization order fiasco". (Said fiasco is well-described on that page so I encourage you to read it and understand it.)

The problem is that this code is not thread-safe. Statics with local scope are internally converted by the compiler into something like this:

```
int ComputeSomething()
{
  static bool cachedResult_computed = false;
  static int cachedResult;
  if (!cachedResult_computed) {
    cachedResult_computed = true;
    cachedResult = ComputeSomethingSlowly();
  }
  return cachedResult;
}
```

Now the race condition is easier to see.

Suppose two threads both call this function for the first time. The first thread gets as far as setting cachedResult_computed = true, and then gets pre-empted. The second thread now sees that cachedResult_computed is true and skips over the body of the "if" branch and returns an uninitialized variable.

What you see here is not a compiler bug. This behavior is **required by the C++ standard**.

You can write variations on this theme to create even worse problems:

```
class Something { ... };
int ComputeSomething()
{
  static Something s;
  return s.ComputeIt();
}
```

This gets rewritten internally as (this time, using pseudo-C++):

```
class Something { ... };
int ComputeSomething()
{
  static bool s_constructed = false;
  static uninitialized Something s;
  if (!s_constructed) {
    s_constructed = true;
    new(&s) Something; // construct it
    atexit(DestructS);
  }
  return s.ComputeIt();
}
// Destruct s at process termination
void DestructS()
{
 ComputeSomething::s.~Something();
}
```

Notice that there are multiple race conditions here. As before, it's possible for one thread to run ahead of the other thread and use "s" before it has been constructed.

Even worse, it's possible for the first thread to get pre-empted immediately after testing s_constructed but **before** setting it to "true". In this case, the object s gets **double-constructed** and **double-destructed**.

That can't be good.

But wait, that's not all. Not look at what happens if you have *two* runtime-initialized local statics:

```
class Something { ... };
int ComputeSomething()
{
  static Something s(0);
  static Something t(1);
  return s.ComputeIt() + t.ComputeIt();
}
```

This is converted by the compiler into the following pseudo-C++:

```
class Something { ... };
int ComputeSomething()
{
  static char constructed = 0;
  static uninitialized Something s;
  if (!(constructed & 1)) {
    constructed |= 1;
    new(&s) Something; // construct it
    atexit(DestructS);
  }
  static uninitialized Something t;
  if (!(constructed & 2)) {
    constructed |= 2;
    new(&t) Something; // construct it
    atexit(DestructT);
  }
  return s.ComputeIt() + t.ComputeIt();
}
```

To save space, the compiler placed the two "x_constructed" variables into a bitfield. Now there are multiple **non-interlocked** read-modify-store operations on the variable "constructed".

Now consider what happens if one thread attempts to execute "constructed |= 1" at the same time another thread attempts to execute "constructed |= 2".

On an x86, the statements likely assemble into

```
  or constructed, 1
...
  or constructed, 2
```

without any "lock" prefixes. On multiprocessor machines, it is possible for the two stores both to read the old value and clobber each other with conflicting values.

On ia64 and alpha, this clobbering is much more obvious since they do not have a single read-modify-store instruction; the three steps must be explicitly coded:

```
  ldl t1,0(a0)     ; load
  addl t1,1,t1     ; modify
  stl t1,1,0(a0)   ; store
```

If the thread gets pre-empted between the load and the store, the value stored may no longer agree with the value being overwritten.

So now consider the following insane sequence of execution:

- Thread A tests “constructed” and finds it zero and prepares to set the value to 1, but it gets pre-empted.
- Thread B enters the same function, sees “constructed” is zero and proceeds to construct both “s” and “t”, leaving “constructed” equal to 3.
- Thread A resumes execution and completes its load-modify-store sequence, setting “constructed” to 1, then constructs “s” (a second time).
- Thread A then proceeds to construct “t” as well (a second time) setting “constructed” (finally) to 3.

Now, you might think you can wrap the runtime initialization in a critical section:

```
int ComputeSomething()
{
 EnterCriticalSection(...);
 static int cachedResult = ComputeSomethingSlowly();
 LeaveCriticalSection(...);
 return cachedResult;
}
```

Because now you’ve placed the one-time initialization inside a critical section and made it thread-safe.

But what if the second call comes from within the same thread? (“We’ve traced the call; it’s coming from inside the thread!”) This can happen if ComputeSomethingSlowly() itself calls ComputeSomething(), perhaps indirectly. Since that thread already owns the critical section, the code enter it just fine and you once again end up returning an uninitialized variable.

Conclusion: When you see runtime initialization of a local static variable, be very concerned.

Raymond Chen

Follow