The debugger lied to you because the CPU was still juggling data in the air

devblogs.microsoft.com/oldnewthing/20121130-00

November 30, 2012

Raymond Chen

A colleague was studying a very strange failure, which I've simplified for expository purpose.

The component in question has the following basic shape, ignoring error checking:

```
// This is a multithreaded object
class Foo
{
public:
 void BeginUpdate();
 void EndUpdate();
 // These methods can be called at any time
 int GetSomething(int x);
 // These methods can be called only between
 // BeginUpdate/EndUpdate.
 void UpdateSomething(int x);
private:
 Foo() : m_cUpdateClients(0), m_pUpdater(nullptr) { ... }
 LONG m_cUpdateClients;
 Updater *m_pUpdater;
};
```

There are two parts of the `Foo` object. One part that is essential to the object's task, and another part that is needed only when updating. The parts related to updating are expensive, so the `Foo` object sets them up only when an update is active. You indicate that an update is active by calling `BeginUpdate`, and you indicate that you are finished updating by calling `EndUpdate`.

```
// Code in italics is wrong
void Foo::BeginUpdate()
{
 LONG cClients = InterlockedIncrement(&m_cUpdateClients);
 if (cClients == 1) {
  // remember, error checking has been elided
  m_pUpdater = new Updater();
 }
 // else, we are already initialized for updating,
 // so nothing to do
}
void Foo::EndUpdate()
{
 LONG cClients = InterlockedDecrement(&m_cUpdateClients);
 if (cClients == 0) {
  // last update client has disconnected
  delete m_pUpdater;
  m_pUpdater = nullptr;
 }
}
```

There are a few race conditions here, and one of them manifested itself in a crash. (If two threads call `BeginUpdate` at the same time, one of them will increment the client count to 1 and the other will increment it to 2. The one which increments it to 1 will get to work initializing `m_pUpdater`, whereas the second one will run ahead on the assumption that the updater is fully-initialized.)

What we saw in the crash dump was that `UpdateSomething` tries to use `m_pUpdater` and crashed on a null pointer. What made the crash dump strange was that if you actually looked at the `Foo` object in memory, the `m_pUpdater` was non-null!

```
    mov ecx, [esi+8] // load m_pUpdater
    mov eax, [ecx]   // load vtable -- crash here
```

If you actually looked at the memory pointed-to by `ESI+8`, the value there was not null, yet in the register dump, `ECX` was zero.

Was the CPU hallucinating? The value in memory is nonzero. The CPU loaded a value from memory. But the value it read was zero.

The CPU wasn't hallucinating. The value it read from memory was in fact zero. The reason why you saw the nonzero value in memory was that in the time it took the null pointer exception to be raised, then caught by the debugger, the other thread managed to finish calling `new Updater()`, store the result back into memory, and then return back to its caller and proceed as if everything were just fine. Thus, when the debugger went to capture the memory dump, it captured a non-zero value in the dump, and the code which updated `m_pUpdater` was long gone.

This type of race condition is more likely to manifest on multi-core machines, because on those types of machines, the two CPUs can have different views of memory. The thread doing the initialization can update `m_pUpdater` in memory, and other CPUs may not find out about it until some time later. The updated value was still in flight when the crash occurred. Before the debugger can get around to capturing the `m_pUpdater` member in the crash dump, the in-flight value lands, and what you see in the crash dump does not match what the crashing CPU saw.

Raymond Chen

Follow