COM in-process DLL unloaded while trying to clean up from the destruction of the last object

devblogs.microsoft.com/oldnewthing/20200430-00

April 30, 2020

Raymond Chen

A customer was studying a crash that was related to how their COM in-process DLL cleaned up prior to unloading.

Their DLL cleans up when the last object is destroyed.

```
class MyObject
{
 MyObject() { IncrementObjectCount(); }
 ~MyObject() { DecrementObjectCount(); }
 ...
};

std::mutex lock;
unsigned int objectCount;

void IncrementObjectCount()
{
  std::lock_guard guard(lock);
  ++objectCount;
}

void DecrementObjectCount()
{
  std::lock_guard guard(lock);
  if (--objectCount == 0) CleanUpStuff();
}

HRESULT DllCanUnloadNow()
{
  return objectCount == 0 ? S_OK : S_FALSE;
}
```

Let's start from the bottom.

The `DllCanUnloadNow` function is called by COM to see if it's okay to unload the DLL. What we have here is a very standard implementation that allows the DLL to be unloaded if there are no outstanding objects.

The `objectCount` variable keeps track of how many objects are still in existence. When the object count reaches zero, the DLL cleans up some global resources in anticipation of the possibility of being unloaded. If the next thing that happens is that a new object is created, then those global resources will be recreated on demand.

And the first thing in the code fragment is the object itself, which updates the object count at construction and destruction.

What they found in the crash is that the DLL is unloaded while `CleanUpStuff` is still running. The way this can happen is if `DllCanUnloadNow` is called while the cleanup is still in progress.

When the last object is destructed, the object count reaches zero, and cleanup begins. If `DllCanUnloadNow` is called during this time, the object count is zero, so `DllCanUnload-Now` says, "Sure, go ahead and unload me!" And then you get unloaded while `CleanUp-Stuff` is still running.

There are a few ways to solve this.

One is to make `DllCanUnloadNow` take the lock before inspecting the object count. That way, it cannot catch `DecrementObjectCount` during the danger window between the count reaching zero and the cleanup being complete.

```
HRESULT DllCanUnloadNow()
{
  std::lock_guard guard(lock);
  return objectCount == 0 ? S_OK : S_FALSE;
}
```

You can optimize this by doing a short-circuit check outside the lock, and a full check inside.[1]

```
HRESULT DllCanUnloadNow()
{
  if (objectCount) return S_FALSE;
  std::lock_guard guard(lock);
  return objectCount == 0 ? S_OK : S_FALSE;
}
```

Another solution is to update the object count *after* cleaning up. That way, an object count of zero means "All cleaned up."

```
void DecrementObjectCount()
{
  std::lock_guard guard(lock);
  auto newCount = objectCount - 1;
  if (newCount == 0) {
    CleanUpStuff();
    std::atomic_thread_fence(std::memory_order_release);
  }
  objectCount = newCount;
}
```

Notice that we created a release barrier between cleaning up and updating the object count, so that the effects of the cleanup are visible to other threads before the revised object count becomes visible to other threads. This extra step isn't important in this particular case because the only variable that is used outside the lock is the object count itself, and it therefore cannot race against itself. But if `DllCanUnloadNow` had accessed both `objectCount` and some global variable that was cleaned up by `CleanUpStuff()`, then there would be a risk of tearing. But then again, you probably shouldn't be accessing those variables outside the lock anyway.

The above solutions all have the property that we clean up as soon as the object count hits zero. This can be wasted effort if a new object is about to be created shortly thereafter.

The general guidance from the COM team is to do your cleanup in `DllCanUnloadNow`.

```
std::atomic_int objectCount;

void IncrementObjectCount()
{
  ++objectCount;
}

void DecrementObjectCount()
{
  --objectCount;
}

HRESULT DllCanUnloadNow()
{
  if (objectCount) return S_FALSE;
  CleanUpStuff();
  return S_OK;
}
```

At this point, the locks around `objectCount` are gone. I'm assuming that the `CleanUpStuff` function will take whatever locks it needs to protect the shared globals.

[1] For expository purposes, I'm ignoring data races and assuming that the object count is atomically updated. You can use `std::atomic` to get atomic behavior. Or, since Win32 requires that simple reads and writes to aligned 32-bit values are atomic but not synchronized, you can just read the value directly.

Raymond Chen

Follow