Avoiding deadlocks when cancelling a thread pool callback, part 2: Referring back to the containing object

devblogs.microsoft.com/oldnewthing/20180504-00

May 4, 2018

Raymond Chen

Last time, we looked at the case where the context for the callback is some data that isn't part of the containing object. However, most of the time, the context for the callback is the object that created the callback.

You might naïvely decide to follow the preceding pattern, using the container object as the reference data. However, this doesn't work because that would create a circular reference. Once you put the strong reference to the containing object in the reference data, you have a circular reference, and the object will never be destroyed.[1] Instead, you have to use a weak reference and try to promote it to a strong reference in the callback.

```
class ObjectWithTimer :
  public RuntimeClass<...> // WRL-specific code
{
public:
 ObjectWithTimer();
 void StartTimer();
 void StopTimer();

private:
 static void CALLBACK TimerCallback(
  PTP_CALLBACK_INSTANCE instance,
  void* context, PTP_TIMER timer);

 WRL::WeakRef weakThis; // WRL-specific code
 std::unique_ptr<TP_TIMER, TpTimerDeleter> timer;
};

ObjectWithTimer::ObjectWithTimer()
{
 // Error checking elided for expository purposes.
 WRL::AsWeak(this, &weakThis); // WRL-specific code
}
```

For convenience, we capture the weak reference at construction and just hang onto it for the lifetime of the object. This saves us the trouble of having to create the weak reference each time we start the timer. It also brings us a step closer to making `StartTimer` have no error path. (The last step would be to front-load the `CreateThreadpoolTimer` and leave the `PTP_TIMER` valid for the lifetime of the `ObjectWithTimer`. I leave this as an exercise.)

```
void ObjectWithTimer::StartTimer()
{
  // Error checking elided for expository purposes.
  timer = CreateThreadpoolTimer(
    TimerCallback,
    weakThis.Get(), // WRL-specific code
    nullptr);

  // Start the timer
  SetThreadpoolTimer(timer, ...);
}

void ObjectWithTimer::StopTimer()
{
  timer.reset();
}
```

These methods are basically the same as before, except that we don't clean up the `weakThis` when stopping the timer, because we want to leave it ready for the next `StartTimer`.

```
void ObjectWithTimer::TimerCallback(
  PTP_CALLBACK_INSTANCE instance,
  void* context, PTP_TIMER timer)
{
 // Try to promote the weak reference to a strong reference.
 WRL::ComPtr<ObjectWithTimer> strongThis;
 WRL::WeakRef(reinterpret_cast<IWeakReference*>(context))
    .As(&strongThis); // WRL-specific code

 context = nullptr;

 // If the weak reference failed to resolve, then our container is
 // destructing.
 if (!self) return;

 DisassociateCurrentThreadFromCallback(instance);

 ... do stuff with strongThis ...
}
```

In the version from last time, we promoted the raw COM pointer to a strong reference, with the knowledge that the raw COM pointer was valid. However, it's possible that the promotion of the `WRL::WeakRef` to a strong reference may fail. How? We'll discuss that later.

Here's a translation of the pattern into `std::weak_ref`:

```
class ObjectWithTimer :
  // weak_ptr-specific code
  public std::enable_shared_from_this<ObjectWithTimer>
{
public:
 ObjectWithTimer();
 void StartTimer();
 void StopTimer();

private:
 static void CALLBACK TimerCallback(
  PTP_CALLBACK_INSTANCE instance,
  void* context, PTP_TIMER timer);

 std::weak_ref<ObjectWithTimer> weakThis; // weak_ptr-specific code
 std::unique_ptr<TP_TIMER, TpTimerDeleter> timer;
};

ObjectWithTimer::ObjectWithTimer()
 : weakThis(weak_from_this()) // weak_ptr-specific code
{
}
```

In the case of `weak_ptr`, we can initialize `weakThis` via a member initializer.

```
void ObjectWithTimer::StartTimer()
{
  // Error checking elided for expository purposes.
  timer = CreateThreadpoolTimer(
    TimerCallback,
    std::addressof(weakThis), // weak_ptr-specific code
    nullptr);

  SetThreadpoolTimer(timer, ...);
}

void ObjectWithTimer::StopTimer()
{
  timer.reset();
}

void ObjectWithTimer::TimerCallback(
  PTP_CALLBACK_INSTANCE instance,
  void* context, PTP_TIMER timer)
{
 // Try to promote the weak reference to a strong reference.
 // weak_ptr-specific code
 auto strongThis =
  reinterpret_cast<
   std::weak_ref<ObjectWithTimer>*>(context))
    ->lock();

 context = nullptr;

 // If the weak reference failed to resolve, then our container is
 // destructing.
 if (!strongThis) return;

 DisassociateCurrentThreadFromCallback(instance);

 ... do stuff with strongThis ...
}
```

The subtlety in both of the cases is that the promotion of the weak reference to a strong reference may fail. You might think, "How is that possible? When we shut down the timer, we always wait until the callback has reached the `DisassociateCurrentThreadFromCallback`, and since we're waiting, that means that the `ObjectWithTimer` is still valid. Therefore, the conversion of the weak reference to a strong reference should always succeed."

But it doesn't if the call to `WaitForThreadpoolXxxCallbacks` is running *as part of object destruction*. There is a race window between the start of destruction (when the last strong reference goes away) and the time the callback starts running. To close this window, weak references can no longer be promoted to strong references once an object starts destructing, (If they could, then it would mean that an object would finish running its destructors and find that there's still a strong reference to it. This is clearly a bad state of affairs, and since you can't "undestruct" an object, the system must prevent code from being able to "resurrect" a destructing object via a weak-to-strong conversion.)

Note that if you follow this pattern, then the `ObjectWithTimer` must be a heap-allocated object so that you can create a weak pointer to it and allow the callback to extend the object's lifetime after its owner has released its last reference.

Epilogue: A reminder that this additional complexity is needed only to address the scenario where a callback deadlocks with its main thread. If your callback does not require its main thread to be in any particular state (doesn't use any locks or other exclusive resources that the main thread may be holding while waiting for the callback to complete, doesn't communicate with the main thread), then you can use the simpler life-time management technique described at the start of this mini-series.

Bonus reading: Threadpool articles by Hari Pulapaka.

[1] Then again, maybe that's what you want, in case this is a feature, not a bug. For example, you might want the timer to continue running until some condition is met. The object-with-timer is a fire-and-forget timer that turns itself off when it decides that its job is done.

Raymond Chen

Follow