Avoiding deadlocks when cancelling a thread pool callback, part 1: External callback data

devblogs.microsoft.com/oldnewthing/20180503-00

May 3, 2018

Raymond Chen

We saw last time how to use the thread pool cleanup functions to manage the lifetime of the context data. But if the callback function tries to call into the thread that is calling `WaitForThreadpoolTimerCallbacks`, then you have a deadlock. The callback cannot proceed until `WaitForThreadpoolTimerCallbacks` returns, but `WaitForThreadpoolTimerCallbacks` won't return until the callback completes.

You can find yourself in this situation without realizing it. The callback function might send a message to a window that is owned by the waiting thread. Or it could invokes a method on an apartment-threaded object that belongs to the waiting thread. Or it could attempt to enter a critical section that is held by the waiting thread.[1] In many cases, the waiting thread is cleaning up an object in its destructor, so you don't even control the locks that may be held at that point.

This is where `DisassociateCurrentThreadFromCallback` enters the picture. The `DisassociateCurrentThreadFromCallback` function tells the thread pool, "For the purpose of waiting until all callbacks are complete, consider this callback to be complete even though it's still running." You can think of this as the `ReplyMessage` of the thread pool. This means that functions like `WaitForThreadpoolXxxCallbacks` will return, but other functions like `XxxWhenCallbackReturns` won't be fooled. They will wait for the callback to return for real before setting the event or leaving the critical section or whatever.

Let's figure out how to take advantage of this.

We make the context data for the callback function a thread-safe reference-counted object. Typical examples are a COM pointer to an agile object, and a reference to a `std::shared_ptr`.[2] The first thing the callback function does is to increment the reference count on the object and save it in an RAII object. For a COM pointer, it would be creating a COM smart pointer around it (say, converting it to a `WRL::ComPtr`). For a `std::shared_ptr` it would be copying the `std::shared_ptr` to a local variable.

Once the context data has been safely referenced, you call `DisassociateCurrentThread-FromCallback` to release the waiting thread, if any.

At this point, you can do your work with the captured strong reference (the `WRL::ComPtr` or the local copy of the `std::shared_ptr`) and stop using the inbound context parameter, because it is no longer valid; the waiting thread may have destroyed it.

When your callback function completes, the RAII type will release the reference to the context data, and if that was the last reference, it will destroy the context data.

We start with this helper class.

```
struct TpTimerDeleter
{
 void operator()(PTP_TIMER timer)
 {
  SetThreadpoolTimer(timer, nullptr, 0, 0);
  WaitForThreadpoolTimerCallbacks(timer, true);
  CloseThreadpoolTimer(timer);
 }
};
```

This deleter class performs the standard synchronous shutdown of a thread pool timer, as noted in the documentation.

Here's how we use it to build a thread pool callback that doesn't deadlock at cancellation. We'll start with one based on `WRL::ComPtr`:

```
class ObjectWithTimer
{
public:
 StartTimer();
 StopTimer();

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

 WRL::ComPtr<AgileContextData> contextData; // ComPtr-specific code
 std::unique_ptr<TP_TIMER, TpTimerDeleter> timer;
};
```

Having a `std::unique_ptr` automatically makes the class non-copyable. We'll see soon why it's okay to let the object be movable.

Exercise: Why did I declare the `timer` member after the `contextData` member? (Normally, I don't answer the exercises in the main body of the article, but this is an important detail, so I'll answer it at the end.)

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

 SetThreadpoolTimer(timer, ...);
}
```

The `StartTimer` method assumes that the `contextData` method has been initialized (presumably by methods not shown) and that the timer has not already been started. It creates the timer with the raw COM pointer as the reference data. This is a non-refcounted pointer, so we have to make sure it remains valid for as long as the callback is potentially-callable. Once we create the timer, we start it by calling `SetThreadpoolTimer` and passing the timer parameters (not shown here).

The context parameter passed to the callback is the thing we have to worry about if the `ObjectWithTimer` gets moved. In this case, it's okay to move the `ObjectWithTimer` because the context parameter doesn't point to the `ObjectWithTimer`; it is a raw COM pointer to an object that won't move. Therefore, the `ObjectWithTimer` is movable.

```
void ObjectWithTimer::TimerCallback(PTP_CALLBACK_INSTANCE instance,
  void* context, PTP_TIMER timer)
{
 // ComPtr-specific code
 WRL::ComPtr<AgileContextData> contextData{
    reinterpret_cast<AgileContextData*>(context) };

 context = nullptr;

 DisassociateCurrentThreadFromCallback(instance);

 ... do stuff with contextData ...
}
```

The callback function starts by taking our weak pointer and converting it to a strong pointer. We designed our code so that the raw COM pointer is valid for as long as the callback is potentially-callable, so we know that it is valid here. We put it inside a `WRL::ComPtr` to make it a strong reference.

And then here's the interesting part: We call `DisassociateCurrentThreadFromCallback` to tell the thread pool to release anybody who is waiting for the callback to complete. The callback *is still running*, but anybody who is waiting for it to complete is free to proceed anyway. We can do this because we have captured the information from the `context` parameter, and the main object can free it. (To ensure we don't access the `context` parameter by mistake, we also set `context` to `nullptr`.)

We then perform our callback operation with the strong reference in `contextData`. This can call back into the main thread because the main thread is no longer stuck in `WaitForThreadpoolTimerCallbacks`, It will eventually return to its event dispatch loop, or release its locks, or whatever it is that needs to happen for the thread pool thread to be able to communicate with the main thread.

The other half of the dance comes when we want to stop the timer.

```
void ObjectWithTimer::StopTimer()
{
 timer.reset();
 contextData.Reset(); // ComPtr-specific code
}
```

To stop the timer, we let the `TpTimerDeleter` do the heavy lifting of checking if we have a timer and if so, shutting it down cleanly.

Once that's done, we can safely release our reference to the context data. If the callback is running, it will have its own reference.

For simplicity, this code doesn't try to save the thread pool timer object for future use. One of the features of the thread pool functions is that object creation preallocates all resources. Once you've created the thread pool timer successfully, all other operations will always succeed (assuming they are used correctly, of course). Therefore, what we could've done is allocate the `PTP_TIMER` at construction (throwing if not possible), and the have the `StartTimer` and `StopTimer` methods merely reconfigure the timer and (in the case of `StopTimer`) wait for the callbacks to drain.

Note that in order to avoid the deadlock, we have to accept that the callback may run *after the timer has been stopped*. When you have a deadlock, something has to give, and we choose to break the deadlock by letting the callback complete asynchronously. If you need to know when the callback is definitely finished, you could add an event that gets signaled when the COM object is destructed, so that the caller knows when everything is finally finished.

Adapting the above code to `std::shared_ptr` is not too difficult. The tricky part is that we cannot pass a raw pointer to the context data as our reference data, because raw C++ objects are not reference-counted. Instead, we pass a pointer to the `std::shared_ptr`, which is the thing with the reference count.

```
class ObjectWithTimer
{
public:
 // Make object non-movable
 ObjectWithTimer(ObjectWithTimer&&) = delete;
 ObjectWithTimer operator=(ObjectWithTimer&&) = delete;

 StartTimer();
 StopTimer();

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

 std::shared_ptr<AgileContextData> contextData; // shared_ptr-specific code
 std::unique_ptr<TP_TIMER, TpTimerDeleter> timer;
};

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

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

void ObjectWithTimer::TimerCallback(PTP_CALLBACK_INSTANCE instance,
  void* context, PTP_TIMER timer)
{
 // Capture the context with a strong reference
 // shared_ptr-specific code
 std::shared_ptr<AgileContextData> contextData{
    *reinterpret_cast<
      std::shared_ptr<AgileContextData>*>(context) };

 context = nullptr;

 DisassociateCurrentThreadFromCallback(instance);

 ... do stuff with contextData ...
}

void ObjectWithTimer::StopTimer()
{
  timer.reset();
  contextData.reset(); // shared_ptr-specific code
}
```

The principle is the same here as before, but the implementation is made more complicated by the fact that the `std::shared_ptr` is not necessarily (and in fact usually isn't) the size of a pointer, so it doesn't fit in the context pointer. We have to pass a pointer to the `std::shared_ptr`.

Exercise: We could have written `&contextData` instead of `std::addressof(contextData)`. Why did I use `std::addressof(contextData)`?

The fact that a `std::shared_ptr` doesn't fit in a pointer means that the `std::shared_ptr` cannot move, because it is still being referenced by the context pointer passed to the thread pool callback. We can make the `ObjectWithTimer` object movable by putting the `std::shared_ptr` inside a `std::unique_ptr`, and passing the raw pointer to the `std::shared_ptr`. The `std::unique_ptr` will transfer the pointer to the moved-to object, and the `std::shared_ptr` itself doesn't move.

```
class ObjectWithTimer
{
public:
 // Make object movable again
 // ~~ObjectWithTimer(ObjectWithTimer&&) = delete;~~
 // ~~ObjectWithTimer operator=(ObjectWithTimer&&) = delete;~~

 StartTimer();
 StopTimer();

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

 std::unique_ptr<
   shared_ptr<AgileContextData>> contextData; // unique_ptr-specific code
 std::unique_ptr<TP_TIMER, TpTimerDeleter> timer;
};

void ObjectWithTimer::StartTimer()
{
  // Error checking elided for expository purposes.
  timer = CreateThreadpoolTimer(
    TimerCallback,
    contextData.get(), // unique_ptr-specific code
    nullptr);

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

void ObjectWithTimer::TimerCallback(PTP_CALLBACK_INSTANCE instance,
  void* context, PTP_TIMER timer)
{
 // Capture the context with a strong reference
 // no change here
 std::shared_ptr<AgileContextData> contextData{
    *reinterpret_cast<
      std::shared_ptr<AgileContextData>*>(context) };

 context = nullptr;

 DisassociateCurrentThreadFromCallback(instance);

 ... do stuff with contextData ...
}

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

```
  contextData.reset(); // no change here
}
```

The idea in all the cases is that we keep a reference-counted object in the `contextData` and provide the callback a way to convert its context parameter to a strong reference. Since we are careful always to keep the context parameter valid as long as the callback is potentially-callable, this conversion is straightforward: Just create your own strong reference from the raw pointer.

Often, the context for the callback is the containing object itself, rather than some external data. We'll explore that scenario next time, because this article is too long as it is.

[1] The critical section case is easy to imagine: You might have a critical section that protects access to object state. The waiting thread owns the critical section because it's trying to clean up the object's thread pool timer. The callback is trying to acquire the critical section because it wants to access the object's state as part of the callback operation.

[2] Be careful with the `std::shared_ptr`. Copying a `std::shared_ptr` is thread-safe, but mutating it is not, so you should initialize your `std::shared_ptr` with the context structure and not modify the `std::shared_ptr` until you are sure that no other threads are accessing it.

Raymond Chen

Follow