What is so special about the Application STA?

devblogs.microsoft.com/oldnewthing/20210224-00

February 24, 2021

Raymond Chen

Windows 8 introduced a new COM threading model which is a variation of the single-threaded apartment (STA). It's called the *Application STA*.

The Application STA is a single-threaded apartment, but with the additional restriction that it cannot be re-entered. Consider the following timeline, with time proceeding from top to bottom:

Thread 1		**Thread 2 (normal STA)**		**Thread 3**
		Call thread 2	→	Start operation A
		(waiting)		(working)
Call thread 2	→	Start operation B		

We have three threads, call them Threads 1, 2, and 3. Thread 2 makes a call to Thread 3 and waits for a response. While Thread 2 is waiting, Thread 1 makes a call into Thread 2. This causes Thread 2 to become re-entered, and that's a tricky situation.

If the second call is to perform some operation unrelated to the first call, then you're probably going to be okay. But suppose the second call is to perform an operation on the same object that the first call is working on. In that case, the first call will have its object modified out from under it.

```
void FrobAllWidgets()
{
  for (auto&& widget : m_widgets) {
    widget.Frob(); // calls to thread 3
  }
}

void AddWidget(Widget const& widget)
{
  m_widgets.push_back(widget);
}
```

Suppose Thread 2 is doing a `FrobAllWidgets`, and the call to `widget.Frob();` results in a call to Thread 3.

While one of those calls is in progress, Thread 2 is just sitting around waiting for the call to complete.

And just at that moment, Thread 1 comes in and adds another widget.

This mutates the vector of widgets, causing the `for` loop in `FrobAllWidgets` to go haywire because the vector was resized, causing all the iterators to become invalid.

You might try to fix this by adding a critical section:

```
wil::critical_section m_cs;

void FrobAllWidgets()
{
  auto guard = m_cs.lock();
  for (auto&& widget : m_widgets) {
    widget.Frob(); // calls to thread 3
  }
}

void AddWidget(Widget const& widget)
{
  auto guard = m_cs.lock();
  m_widgets.push_back(widget);
}
```

You think you fixed it, but in fact nothing has changed. Critical sections support recursive acquisition, so what happens is that the re-entrant call to `AddWidget` tries to acquire the critical section, and it *succeeds*, because the owner is the same thread.

Okay, so switch to something that does not support recursive acquisition, like a shared reader-writer lock.

```
wil::srwlock m_srw;

void FrobAllWidgets()
{
  auto guard = m_srw.lock_shared();
  for (auto&& widget : m_widgets) {
    widget.Frob(); // calls to thread 3
  }
}

void AddWidget(Widget const& widget)
{
  auto guard = m_srw.lock_exclusive();
  m_widgets.push_back(widget);
}
```

Well, at least this time there’s no crash. Instead the call hangs, because the re-entrant call to `AddWidget` tries to acquire the exclusive lock, but it cannot because of the existing call to `FrobAllWidgets`. And that existing call cannot complete until `AddWidgets` completes, because `AddWidgets` is running on the same stack.

FrobAllWidgets
↳ Widget::Frob
↳ WaitForFrobToFinish
↳ ReceiveInboundCall
↳ AddWidget

`AddWidget` is running on the same stack, so `FrobAllWidgets` cannot return until the stack unwinds, which means that `AddWidget` needs to return, but it can’t.

The Application STA tries to address this problem by blocking re-entrancy. The diagram now looks like this:

Thread 1		**Thread 2 (application STA)**		**Thread 3**
		Call thread 2	→	Start operation A
		(waiting)		(working)
Call thread 2	⌝			
	↓			

	↓		←	Finished
	↓	Finish		
	↳	Operation B starts		

When Thread 1 makes a call into the Application STA while the Application STA is waiting for a response from Thread 3, the call is not allowed to proceed, because that would result in re-entrancy. The call from Thread 1 is placed on hold until Thread 2 is no longer waiting for an outbound call to complete. Once Thread 2 returns to a normal state, it can receive inbound calls again.

You can detect that you are running in an Application STA by calling `CoGetApartmentType` and checking for an apartment type of STA and an apartment type qualifier of APPLICATION_STA.

Next time, we'll look at another quirk of the Application STA.

Bonus chatter: I forgot to include the Application STA in my list of possible results from `CoGetApartmentType`, so I've retroactively updated it.

Raymond Chen

Follow