Does the CLR really call CoInitializeEx on the first call to unmanaged code, even if you don't deal with COM at all and are just calling native code via p/invoke?

devblogs.microsoft.com/oldnewthing/20150316-00

March 16, 2015

Raymond Chen

Some time ago, I called out this part of the documentation regarding managed and unmanaged threading:

> On the first call to unmanaged code, the runtime calls **CoInitializeEx** to initialize the COM apartment as either an MTA or an STA apartment. You can control the type of apartment created by setting the System.Threading.ApartmentState property on the thread to **MTA**, **STA**, or **Unknown**.

Commenter T asks, "Does it do this even if you don't deal with COM at all and call native code through a P/Invoke?"

Well, the documentation says it does, and we can confirm with an experiment:

```
using System.Runtime.InteropServices;
class Program
{
 public static void Main()
 {
  var thread = new System.Threading.Thread(
    () => {
   System.Console.WriteLine("about to p/invoke");
   GetTickCount();
  });
  thread.Start();
  thread.Join();
 }
 [DllImport("kernel32.dll")]
 extern static uint GetTickCount();
}
```

Run this program with a breakpoint on `CoInitializeEx`.

First breakpoint is hit with this stack:

```
rax=00007ffebc529b70 rbx=00000000007c6100 rcx=0000000000000000
rdx=0000000000000000 rsi=0000000000000001 rdi=0000000000000002
rip=00007ffebc529b70 rsp=000000000056f038 rbp=000000000056f0b0
 r8=0000000000000001  r9=0000000000000000 r10=0000000000000000
r11=0000000000000037 r12=0000000000004000 r13=0000000000000001
r14=0000000000000001 r15=0000000000000001
combase!CoInitializeEx
clr!Thread::SetApartment
clr!SystemDomain::SetThreadAptState
clr!SystemDomain::ExecuteMainMethod
clr!ExecuteEXE
clr!_CorExeMainInternal
clr!CorExeMain
mscoreei!CorExeMain
MSCOREE!CorExeMain_Exported
KERNEL32!BaseThreadInitThunk
ntdll!RtlUserThreadStart
```

This call is initializing the main thread of the process. The flags passed to this first call to `CoInitializeEx` are 0, which means that the default threading model of `COINIT_MULTI-THREADED` is used.

The next time the breakpoint hits is with this stack:

```
rax=00000000ffffffff rbx=00000000007d1180 rcx=0000000000000000
rdx=0000000000000000 rsi=0000000000000001 rdi=00000000007d1180
rip=00007ffebc529b70 rsp=000000001a6af9a8 rbp=000000001a6afa20
 r8=000000001a6af948  r9=0000000000000000 r10=00000000007f0340
r11=00000000007f0328 r12=0000000000004000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
combase!CoInitializeEx
clr!Thread::SetApartment
clr!Thread::DoExtraWorkForFinalizer
clr!WKS::GCHeap::FinalizerThreadWorker
clr!ManagedThreadBase_DispatchInner
clr!ManagedThreadBase_DispatchMiddle
clr!ManagedThreadBase_DispatchOuter
clr!WKS::GCHeap::FinalizerThreadStart
clr!Thread::intermediateThreadProc
KERNEL32!BaseThreadInitThunk
ntdll!RtlUserThreadStart
```

From the name `FinalizerThreadStart`, this is clearly the finalizer thread.[1]

Next.

```
rax=00000000ffffffff rbx=000000000039eb20 rcx=0000000000000000
rdx=0000000000000000 rsi=0000000000000001 rdi=0000000000000000
rip=00007ffebc529b70 rsp=000000001a5af3d8 rbp=000000001a5af450
 r8=0000000000000000  r9=000000001a5af3f0 r10=0000000000000000
r11=0000000000000286 r12=0000000000004000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
combase!CoInitializeEx
clr!Thread::SetApartment
clr!Thread::PrepareApartmentAndContext
clr!Thread::HasStarted
clr!ThreadNative::KickOffThread
clr!Thread::intermediateThreadProc
KERNEL32!BaseThreadInitThunk
ntdll!RtlUserThreadStart
```

Okay, this looks like it's kicking off a new thread. I inferred this from the presence on the stack of the function which is deviously named `KickOffThread`.

And the flags passed to this call to `CoInitializeEx` are 0, which once again means that it defaults to MTA.

There, we have confirmed experimentally that, at least in this case, the implementation matches the documentation.

That the implementation behaves this way is not surprising. After all, the CLR does not have insight into the `GetTickCount` function. It does not know *a priori* whether that function will create any COM objects. After all, we could have been p/invoking to `SHGetDesktop-Folder`, which does use COM. Given that the CLR cannot tell whether a native function is going to use COM or not, it has to initialize COM just in case.

[1] Or somebody who is trying to mislead us into thinking that it is the finalizer thread. I tend to discount this theory because as a general rule, code is not intentionally written to be impossible to understand.

Raymond Chen

Follow