Handy delegate shortcut hides important details: The hidden delegate

devblogs.microsoft.com/oldnewthing/20150821-00

August 21, 2015

Raymond Chen

One of my colleagues was having trouble with a little tool he wrote.

> I installed a low-level keyboard hook following the code in this article, but it crashes randomly. Here's what I know so far:
>
> - I spawn a new STA thread to register the hook, so that it can run a message pump, which is a requirement for low-level hooks.
> - After setting the hook, the program waits on a `ManualResetEvent` with `WaitOne()`. Since this is being called from an STA thread, it will pump messages while waiting, which is what we want.
> - The event is signaled by another part of the program when the hook is no longer needed, at which point the thread unregisters the hook before exiting normally.
>
> The crash happens inside `WaitOne()` immediately after keyboard activity occurs. The debugger tells me that it is crashing trying to dispatch a call into a managed stub via the message pump, but that's all I was able to extract.

I took a look at the article that my colleague referenced and observed that there was a subtlety in the code that not obvious, and which may have been lost in translation. I shared my observation in the form a psychic prediction.

> My psychic powers tell me that you did not prevent the delegate from getting GCd. The next time GC runs, the delegate will get collected, and the next attempt to fire the callback will AV because its calling into memory that has been freed.
>
> The sample code from the blog avoids this problem by putting the delegate in a private static, which makes it a GC root, ineligible for collection.
>
> ```
> private static LowLevelKeyboardProc _proc = HookCallback;
> ```
>
> This is subtle because the private static is decoupled from `SetHook`. If you copied `SetHook` but not the private static, then you inadvertently created a bug because local variables can get optimized out.
>
> Either put it in a static, like the sample does, or explicitly extend the delegates lifetime by calling `GC.KeepAlive()` after you unhook the hook.
>
> ```
> LowLevelKeyboardProc proc = HookCallback;
> IntPtr hookId = SetHook(proc);
> WaitOne();
> RemoveHook(hookId);
> GC.KeepAlive(proc); // keep the proc alive until this line is reached
> ```

My colleague realized that was the problem.

> I'd actually thought of that (mostly). I made my callback method itself a static, thinking that this was enough. What I forgot is that C# wraps that in an instance of the delegate automatically, and it was this hidden delegate that was getting GC'd not the callback function itself. This explains why I could always inspect the callback method and see that it was alive and well, yet we were still jumping into space when invoking the callback.
>
> Explicitly calling out the assignment reminded me of the details of delegates. Thanks!

The classical notation for creating a delegate is

```
DelegateType d = new DelegateType(o.Method);
DelegateType d = new DelegateType(Method); // this.Method
```

C# version 2.0 added *delegate inference* which lets you omit the `new DelegateType` most of the time. The compiler will automatically convert the method name (and optional `this` object) into a delegate.

```
DelegateType d = o.Method;
DelegateType d = Method; // this.Method
```

This shorthand is so old, you may not even remember (or realize) that it is a shorthand for a *hidden delegate*.

In my colleague's program, the line

```
    IntPtr hookId = SetHook(HookCallback);
```

was shorthand for

```
    LowLevelKeyboardProc temp = HookCallback;
    IntPtr hookId = SetHook(temp);
```

Once the delegate was made explicit rather than hidden, the issue became clear: Since there was nothing keeping the delegate alive, the delegate disappeared at the next GC, and the unmanaged function pointer disappeared with it.

And now CLR Week will disappear until next time.

Raymond Chen

Follow