Sabotaging yourself: Closing a handle and then using it

devblogs.microsoft.com/oldnewthing/20120926-00

September 26, 2012

Raymond Chen

A customer reported a problem with the `WaitForSingleObject` function:

> I have a DLL with an `Initialize()` function and an `Uninitialize()` function. The code goes like this:
>
> ```
> HANDLE FooMutex;
> BOOL Initialize()
> {
>  ... unrelated initialization stuff ...
>  FooMutex = CreateMutex(NULL, FALSE, "FooMutex");
>  ... error checking removed ...
>  return TRUE;
> }
> BOOL Uninitialize()
> {
>  // fail if never initialized
>  if (FooMutex == NULL) return FALSE;
>  // fail if already uninitialized
>  if (WaitForSingleObject(FooMutex, INFINITE) == WAIT_FAILED)
>   return FALSE;
>  ... unrelated cleanup stuff ...
>  ReleaseMutex(FooMutex);
>  CloseHandle(FooMutex);
>  return TRUE;
> }
> ```
>
> Under certain conditions, the `Initialize()` function is called twice, and the `Uninitialize()` function is correspondingly called twice. Under these conditions, if I run the code on a single-processor system with hyperthreading disabled, then everything works fine. But if I enable hypethreading, then the second call to `Uninitialize()` hangs in the `WaitForSingleObject` call. (As you can see, it's waiting for a mutex handle which was closed by the previous call to `Uninitialize()`.)
>
> Why does this happen only on a hyperthreaded machine? Shouldn't the `WaitForSingleObject` return `WAIT_FAILED` because the parameter is invalid? Is this a bug in Windows hyperthreading support?

Remember, don't immediately jump to the conspiracy theory:[1] Hyperthreading may be the trigger for your problem, but it's probably not a bug in hyperthreading.

While it's true that `WaitForSingleObject` returns `WAIT_FAILED` when given an invalid parameter, handle recycling means that any invalid handle can suddenly become valid again (but refer to an unrelated object).

The problem with hyperthreading will probably recur if you run the scenario on a multiprocessor machine. Hyperthreading (and multi-core processing) means that two threads can be executing simultaneously, which means that the opportunity for race conditions grows significantly.

What's probably happening is that between the two calls to `Uninitialize()`, another thread called `CreateThread` or `CreateEvent` or some other function which creates a waitable kernel object. That new kernel object was coincidentally assigned the same numerical handle value that was previously assigned to your `FooMutex`. (This is perfectly legitimate since you closed the handle, thereby making it available for re-use.) Now when you call `WaitForSingleObject(FooMutex)`, you are actually waiting on that other object. And since that other object is not signaled, the wait call waits.

Okay, so how do we fix the problem? The simple fix is to null out `FooMutex` after closing the handle. This assumes however that your DLL design imposes the restriction on clients that all calls to `Initialize()` and `Uninitialize()` take place on the same thread, and that the DLL is uninitialized on the *first* call to `Uninitialize()`.

Oh, and you may have noticed another bug: When `Initialize()` is called the second time, the DLL reinitializes itself. In particular, it re-creates the mutex and overwrites the handle from the previous call to `Initialize()`. Now you have a handle leak. I suspect that's why the call to `CreateMutex` explicitly passes `"FooMutex"` as the mutex name. The previous version passed `NULL`, creating an anonymous mutex, but then the author discovered that the mutex "stopped working" because some code was using the old handle (using the mutex created by the first call to `Initialize()`) and some code was using the new handle (using the mutex created by the second call to `Initialize()`). Using a named mutex "fixes" the problem by forcing the two calls to `Initialize()` to create a handle to the same underlying object.

To fix the handle leak, you can just stick a `if (FooMutex != NULL) return TRUE;` at the top. The DLL has already been initialized; don't need to initialize it again.

The design as I inferred it is somewhat unusual, however. Usually, when a component exposes `Initialize()` and `Uninitialize()` and supports multiple initialization, the convention is that the DLL initialization remains valid until the *last* call to

`Uninitialize()`, not the first one. If that was the design intention of this DLL, then a different fix is called for, one which I leave as an exercise, since we've drifted pretty far from the original question.

[1]The authors of *The Pragmatic Programmer* call this principle 'select' isn't broken.

Raymond Chen

Follow