You thought reasoning about signals was bad, reasoning about a total breakdown of normal functioning is even worse

devblogs.microsoft.com/oldnewthing/20091113-00

November 13, 2009

Raymond Chen

A customer came to the Windows team with a question, the sort of question which on its face seems somewhat strange, which is itself a sign that the question is merely the tip of a much more dangerous iceberg.

> Under what circumstances will the `GetEnvironmentVariable` function hang?

This is kind of an open-ended question. I mean, for example, somebody might sneak in and call `SuspendThread` on your thread while `GetEnvironmentVariable` is running, which will look like a hang because the call never completes because the thread is frozen.

But the real question for the customer is, "What sort of problem are you seeing that is manifesting itself in an apparent hang in the `GetEnvironmentVariable` function?"

The customer was kind enough to elaborate.

> We have a global unhandled exception filter in our application so we can log all failures. After we finish logging, we call `ExitProcess`, but we find that the application never actually exits. If we connect a debugger to the stuck application, we see it hung in `GetEnvironmentVariable`.

Your gut response should be, "Holy cow, I'm surprised you even got that far!"

This isn't one of those global unhandled exception filters that got installed because your program plays some really clever game with exceptions, No, this is an "Oh no, my program just crashed and I want to log it" exception handler. In other words, when this exception handler "handles" an exception, it's because your program has encountered some sort of serious internal programming error for which the program did not know how to recover. We saw earlier that you can't do much in a signal handler because you might have interrupted a block of code which was in the middle of updating some data structures, leaving them

momentarily inconsistent. But this exception filter is in an even worse state: Not only is there a good chance that the program is in the middle of updating something and left it in an inconsistent state, you are in fact *guaranteed* that the system is in a corrupted state.

Why is this a guarantee? Because if the system were in a consistent state, you wouldn't have crashed!

Programming is about establishing invariants, perturbing them, and then re-establishing them. It is a game of stepping-stone from one island of consistency to another. But the code that does the perturbing and the re-establishing assumes that it's starting from a consistent state to begin with. For example, a function that removes a node from a doubly-linked list manipulates some backward and forward link pointers (temporarily violating the linked list invariant), and then when it's finished, the linked list is back to a consistent state. But this code assumes that the linked list is not corrupted to begin with!

Let's look again at that call to `ExitProcess`. That's going to detach all the DLLs, calling each DLL's `DllMain` with the `DLL_PROCESS_DETACH` notification. But of course, those `DllMain` are going to assume that the data structures are intact and nothing is corrupted. On the other hand, you know for a fact that these prerequisites are not met—the program crashed precisely because something is corrupted. One DLL might walk a linked list—but you might have crashed because that linked list is corrupted. Another DLL might try to delete a critical section—but you might have crashed because the data structure containing the critical section is corrupted.

Heck, the crash might have been inside somebody's `DLL_PROCESS_DETACH` handler to begin with, for all you know.

"Yeah, but the documentation for `TerminateProcess` says that it does not clean up shared memory."

Well, it depends on what you mean by *clean up*. The reference count on the shared memory is properly decremented when the handle is automatically closed as part of process cleanup, and the shared memory will be properly freed once there are no more references to it. It is not cleaned up in the sense of "corruption is repaired"—but of course the operating system can't do that because it doesn't know what the semantics of your shared memory block are.

But this is hardly anything to get concerned about because *your program doesn't know how to un-corrupt the data either*.

"It also says that DLLs don't receive their `DLL_PROCESS_DETACH` notification."

As we saw before, this is a good thing in the case of a corrupted process, because the code that runs in `DLL_PROCESS_DETACH` assumes that your process has not been corrupted in the first place. There's no point running it *when you know the process is corrupted*. You're just

making a bad situation worse.

“It also says that I/O will be in an indeterminate state.”

Well yeah, but that’s no worse than what you have now, which is that your I/O is in an indeterminate state. You don’t know what buffers your process hasn’t flushed, but since your process is corrupted, you have no way of finding out anyway.

“Are you seriously recommending that I use `TerminateProcess` to exit the last chance exception handler?!?”

Your process is unrecoverably corrupted. (This is a fact, because if there were a way to recover from it, *you would have done it instead of crashing*.) What other options are there?

Quit while you’re behind.

Raymond Chen

Follow