If dynamic DLL dependencies were tracked, they'd be all backwards

devblogs.microsoft.com/oldnewthing/20090710-00

July 10, 2009

Raymond Chen

Whenever the issue of DLL dependencies arises, I can count on somebody arguing that these dynamic dependencies should be tracked, even if doing so cannot be proven to be reliable. Even if one could walk the call stack reliably, you would *still* get it wrong. The example I gave originally was the common helper library, where `A.DLL` loads `B.DLL` via an intermediate function in `MIDDLE.DLL`. You want the dependency to be that `A.DLL` depends on `B.DLL`, but instead the dependency gets assigned to `MIDDLE.DLL`. "But so what? Instead of a direct dependency from `A.DLL` to `B.DLL`, we just have two dependencies, one from `A.DLL` to `MIDDLE.DLL`, and another from `MIDDLE.DLL` to `B.DLL`. It all comes out to the same thing in the end." Actually, it doesn't. It comes out much worse. After all, `MIDDLE.DLL` is your common helper library. All of the DLLs in your project depend on it. Therefore, the dependency diagram in reality looks like this:

```
A.DLL  →  B.DLL
   ↓
MIDDLE.DLL
```

`A.DLL` depends on `B.DLL`, and both DLLs depend on `MIDDLE.DLL`. That common DLL really should be called `BOTTOM.DLL` since everybody depends on it. Now you can see why the dependency chain `A.DLL → MIDDLE.DLL → B.DLL` is horribly wrong. Under the incorrect dependency chain, the DLLs would be uninitialized in the order `A.DLL`, `MIDDLE.DLL`, `B.DLL`, even though `B.DLL` depends on `MIDDLE.DLL`. That's because your "invented" dependency *introduces a cycle in the dependency chain*, and a bogus one at that. Once you have cycles in the dependency chain, everything falls apart. You took something that might have worked into something that explodes upon impact. This situation appears much more often than you think. In fact it happens *all the time*. Because in real life, the loader is implemented in the internal library `NTDLL.DLL`, and `KERNEL32.DLL` is just a wrapper function around the real DLL loader. In other words, if your `A.DLL` calls `LoadLibrary("B.DLL")`, you are already using a middle DLL; its name is `KERNEL32.DLL`.

If this “dynamic dependency generation” were followed, then `KERNEL32.DLL` would be listed as *dependent on everything*. When it came time to uninitialize, `KERNEL32.DLL` would uninitialized before all dynamically-loaded DLLs, because it was the one who loaded them, and then all the dynamically-loaded DLLs would find themselves in an interesting world where `KERNEL32.DLL` no longer existed. Besides, the original problem arises when `A.DLL` calls a function in `B.DLL` during its `DLL_PROCESS_DETACH` handler, going against the rule that you shouldn’t call anything outside your DLL from your `DllMain` function (except perhaps a little bit of `KERNEL32` but even then, it’s still not the best idea). It’s one thing to make accommodations so that existing bad programs continue to run, but it’s another to build an entire infrastructure built on unreliable heuristics in order to encourage people to do something they shouldn’t be doing in the first place, and whose guesses end up taking a working situation and breaking it.

You can’t even write programs to take advantage of this new behavior because walking the stack is itself unreliable. You recompile your program with different optimizations, and all of a sudden the stack walking stops working because you enabled tail call elimination. If somebody told you, “Hey, we added this feature that isn’t reliable,” I suspect your reaction would not be “Awesome, let me start depending on it!”

Raymond Chen

Follow