Why are DLLs unloaded in the "wrong" order?

devblogs.microsoft.com/oldnewthing/20050523-05

May 23, 2005

Raymond Chen

When a program starts or when a DLL is loaded, the loader builds a dependency tree of all the DLLs referenced by that program/DLL, that DLL's dependents, and so on. It then determines the correct order in which to initialize those DLLs so that no DLL is initialized until after all the DLLs upon which it is dependent have been initialized. (Of course, if you have a circular dependency, then this falls apart. And as you well know, calling the `LoadLibrary` function or the `LoadLibraryEx` function from inside a DLL's DLL_PROCESS_ATTACH notification also messes up these dependency computations.)

Similarly, when you unload a DLL or when the program terminates, the de-initialization occurs so that a DLL is de-initialized after all its dependents.

But when you load a DLL manually, crucial information is lost: Namely that the DLL that is calling `LoadLibrary` depends on the DLL being loaded. Consequently, if A.DLL manually loads B.DLL, then there is no guarantee that A.DLL will be unloaded before B.DLL. This means, for example, that code like the following is not reliable:

```
HSOMETHING g_hSomething;
typedef HSOMETHING (WINAPI* GETSOMETHING)(void);
typedef void (WINAPI* FREESOMETHING)(HSOMETHING);
GETSOMETHING GetSomething;
FREESOMETHING FreeSomething;
// Ignoring race conditions for expository purposes
void LoadB()
{
 HINSTANCE hinstB = LoadLibrary(TEXT("B.DLL"));
 if (hinstB) {
  GetSomething = (GETSOMETHING)
          GetProcAddress(hinstB, "GetSomething");
  FreeSomething = (FREESOMETHING)
          FreeProcAddress(hinstB, "FreeSomething");
 }
}
// Ignoring race conditions for expository purposes
HSOMETHING CacheSomethingFromB()
{
 if (!g_hSomething &&
     GetSomething && FreeSomething) {
  g_hSomething = GetSomething();
 }
 return g_hSomething;
}
BOOL CALLBACK DllMain(HINSTANCE hinst,
      DWORD dwReason, LPVOID lpReserved)
{
 switch (dwReason) {
 ...
 case DLL_PROCESS_DETACH:
  if (g_hSomething) {
   FreeSomething(g_hSomething); // oops
  }
  break;
 }
 return TRUE;
}
```

At the line marked "oops", there is no guarantee that `B.DLL` is still in memory because `B.DLL` does not appear in the dependency list of `A.DLL`, even though there is a runtime-generated dependency caused by the call to `LoadLibrary`.

Why can't the loader keep track of this dynamic dependency? In other words when `A.DLL` calls `LoadLibrary(TEXT("B.DLL"))`, why can't the loader automatically say "Okay, now A.DLL depends on B.DLL"?

First of all, because as I've noted before, you can't trust the return address.

Second, even if you could trust the return address, you still can't trust the return address. Consider:

```
// A.DLL - same as before except for one line
void LoadB()
{
 HINSTANCE hinstB = MiddleFunction(TEXT("B.DLL"));
 if (hinstB) {
  GetSomething = (GETSOMETHING)
          GetProcAddress(hinstB, "GetSomething");
  FreeSomething = (FREESOMETHING)
          FreeProcAddress(hinstB, "FreeSomething");
 }
}
// MIDDLE.DLL
HINSTANCE MiddleFunction(LPCTSTR pszDll)
{
 return LoadLibrary(pszDll);
}
```

In this scenario, the load of `B.DLL` happens not directly from `A.DLL`, but rather through an intermediary (in this case, `MiddleFunction`). Even if you could trust the return address, the dependency would be assigned to `MIDDLE.DLL` instead of `A.DLL`.

"What sort of crazy person would write a function like `MiddleFunction`?", you ask. This sort of intermediate function is common in helper/wrapper libraries or to provide additional lifetime management functionality (although it doesn't do it any more, though it used to).

Third, there is the case of the `GetModuleHandle` function.

```
void UseBIfAvailable()
{
 HINSTANCE hinstB = GetModuleHandle(TEXT("B"));
 if (hinstB) {
  DOSOMETHING DoSomething = (DOSOMETHING)
          GetProcAddress(hinstB, "DoSomething");
  if (DoSomething) {
   DoSomething();
  }
 }
}
```

Should this call to `GetModuleHandle` create a dependency?

Note also that there are dependencies among DLLs that go beyond just `LoadLibrary`. For example, if you pass a callback function pointer to another DLL, you have created a reverse dependency.

A final note is that this sort of implicit dependency, as hard as it is to see as written above, is even worse once you toss global destructors into the mix.

```
class SomethingHolder
{
public:
 SomethingHolder() : m_hSomething(NULL);
 ~SomethingHolder()
  { if (m_hSomething) FreeSomething(m_hSomething); }
 HSOMETHING m_hSomething;
};
SomethingHolder g_SomethingHolder;
...
```

The DLL dependency is now hidden inside the `SomethingHolder` class, and when `A.DLL` unloads, `g_SomethingHolder`'s destructor will run and try to talk to `B.DLL`. Hilarity ensues.

Raymond Chen

Follow