Answers to a customer's questions about memory and DLLs

devblogs.microsoft.com/oldnewthing/20161209-00

December 9, 2016

Raymond Chen

A customer had a few questions about memory and DLLs.

> We have some questions that no one here can answer.
>
> 1. When are DLLs actually loaded (mapped) into memory? Does it happen when the calling program starts, or does it happen when the DLL is actually called?
> 2. Do DLLs have their own heap and code memory, or do they use those of the calling program?

Okay, let's see if we can take these questions apart.

First: When are DLLs actually loaded into memory?

Well, it depends on how you linked to them. If you didn't do anything special, then a module that links to another DLL will list the target DLL in its import table, and the target DLL will be loaded at the same time the module itself is loaded. If that module is the main module, then the target DLL will be loaded at process startup.

On the other hand, if you listed the DLL in your `/DELAYLOAD` option, then the DLL will be loaded the first time any code in your module calls a function in the target DLL.

Next question: Do DLLs have their own heap, or do they use those of the calling program?

It is up to the DLL whether it wants to create its own heap, or whether it wants to use an existing heap. In fact, a DLL doesn't need to be consistent in its decision. It could use its own heap for some things and an existing heap for other things.

If a DLL wants to interoperate with other DLLs, and it wants to be able to allocate and free memory across DLL boundaries, then the two DLLs need to agree on which heap the memory should be allocated from and freed to. One way of doing this is to choose an existing external heap and have both parties agree to use that. You see this, for example, with COM interfaces, which all agree to use the COM task allocator to allocate and free memory across COM object boundaries. Memory returned by a COM method must be allocated from the COM task allocator (usually by calling `CoTaskMemAlloc`), and that memory must then be freed to the COM task allocator (usually by calling `CoTaskMemFree`).¹

Another pattern that is often used is for a DLL to allocate memory, and then provide a custom free function that the other DLL must call in order to free the memory. The most commonly encountered example of this is the function `NetApiBufferFree`

Note that the above rules about agreeing on a heap to use apply only in the case where the memory is allocated by one DLL and freed by another. If the allocation and free happens both in the same DLL, then that DLL can use any policy it likes, as long as every allocation is matched with a compatible free. Internal objects could be allocated with `malloc`, as long as they are freed by `free`.

Note that the `malloc`/`free` pairing must be to the same instance of the C runtime library. If one DLL calls `malloc` and passes it to another DLL, and that other DLL calls `free`, then the two sides of the equation must be using the same C runtime library.

Next part of the question: Do DLLs have their own code memory, or do they use those of the calling program?

I'm not sure what this question is trying to ask. Certainly the code for a DLL comes from the DLL. What would be the point of a DLL that just used code from the calling program? The calling program wouldn't need the DLL at all. It already has the code!

So I'll assume that the question really is "Are DLLs loaded into the same address space as the calling program?" And the answer is yes, all the DLLs loaded by a process share the same address space.

There's another question the customer didn't ask, but which is closely related: Which C runtime does the DLL use?

The answer to this question is a bit circular. The DLL uses the C runtime that it chooses. When you create the DLL, you specify somewhere which C runtime you want to use, and how you want to use it. (You may not be aware that you even made this choice, because most project templates will make a default choice for you.)

One choice is to link the C runtime statically into your module. This means that the code in the C runtime is incorporated into your module. This becomes your private C runtime, and it will not be shared with anyone. This private C runtime initializes when your module is loaded.

Another choice is to link the C runtime dynamically into your module. This means that the code in the C runtime remains in a DLL which will be loaded either when your module is loaded (if linked statically) or when the first function in it is called (if linked with

`/DELAYLOAD`).

In practice, nobody delay-loads the C runtime library.

If you choose this route, then the C runtime library will be shared with any other modules that use the same version of the same C runtime library. The C runtime DLL will be loaded when the first module that uses it is loaded.

If you use `malloc` or `new`, then you are calling the corresponding function in the C runtime library that your module has chosen. In the case where you statically linked the C runtime, this is definitely not the same instance of the C runtime that anybody else is using, so that memory can be freed only by your module. In the case where you dynamically linked the C runtime, then this might or might not be the same instance of the C runtime that another module is using. Only if the other module is indeed using the same version of the same C runtime library will it be able to free the memory with the corresponding `free` or `delete` function.

The last detail is something people often overlook. They will create two modules which do not share the same instance of the C runtime, either because one or both linked the C runtime statically, or because the two both linked to different versions of the C runtime. If you are in either of these cases, then you cannot share C runtime data structures between the two modules because the two modules are using different C runtimes. This means that you cannot pass a `FILE*` between modules, you cannot pass a `std::string` between modules, and you cannot even pass a file descriptor between modules, because even though file descriptors are integers, they are integers that are managed by the corresponding C runtime library.

This is why most coding guidelines specify that each module is responsible for allocating, managing, and freeing its own objects, rather than handing objects to other modules expecting the other module to be able to free them properly. The things you have to line up in order to get cross-module object memory management to work can be easily thrown out of alignment.[2]

Bonus chatter: Note that there is is a confusing overload of the word "static" when applied to DLL linkage. You can choose to link a library statically or dynamically, and if you choose dynamic linking, then your next choice is whether the target DLL is loaded statically or delayed (also called "dynamically"). Let me draw a table, because people like tables.

How was the library linked?	How is the DLL loaded?	When is the code loaded?
Statically	N/A (there is no other DLL)	When the module loads

Dynamically	Statically	When the module loads
	Delayed/dynamic	When the first function is called

[1] There are a few old interfaces that use `HGLOBAL` to share memory (such as `STGMEDIUM`), but those exist for historical reasons, not because they're a good idea.

[2] For example, you might decide to upgrade one module to a newer version of the compiler, in order to take advantage of a new feature, but choose to leave another module with an older version of the compiler because you haven't made any changes to it and don't want to take any risk that the new compiler will expose an issue. (Or because you're building a patch for your program, and you want to minimize the size of your patch by omitting files which didn't change.) Once you do that, you have a C runtime mismatch, and then scary things will happen if the two modules are not in agreement on how memory will be allocated and freed.

Raymond Chen

Follow