What's the difference between SHGetMalloc, SHAlloc, CoGetMalloc, and CoTaskMemAlloc

devblogs.microsoft.com/oldnewthing/20040705-00

July 5, 2004

Raymond Chen

Let's get the easy ones out of the way.

First, CoTaskMemAlloc is exactly the same as CoGetMalloc(MEMCTX_TASK) + IMalloc::Alloc, and CoTaskMemFree is the same as CoGetMalloc(MEMCTX_TASK) + IMalloc::Free. CoTaskMemAlloc and CoTaskMemFree (and the less-used CoTaskMemRealloc) are just convenience functions that save you the trouble of having to mess with CoGetMalloc yourself. Consequently, you can safely allocate memory via CoGetMalloc(MEMCTX_TASK) + IMalloc::Alloc, and then free it with CoTaskMemFree, and vice versa. It's all the same allocator.

Similarly, SHAlloc and SHFree are just wrappers around SHGetMalloc which allocate/free the memory via the shell task allocator. Memory you allocated via SHGetMalloc + IMalloc::Alloc can be freed with SHFree.

So far, we have this diagram.

Shell task allocator			OLE task allocator		
SHAlloc/ SHFree	=	SHGetMalloc	??	CoGetMalloc	=	CoTaskMemAlloc/ CoTaskMemFree

Now what about those question marks?

If you read the comments in shlobj.h, you may get a bit of a hint:

```
//=============================================================================
//
// Task allocator API
//
//  All the shell extensions MUST use the task allocator (see OLE 2.0
// programming guild for its definition) when they allocate or free
// memory objects (mostly ITEMIDLIST) that are returned across any
// shell interfaces. There are two ways to access the task allocator
// from a shell extension depending on whether or not it is linked with
// OLE32.DLL or not (purely for efficiency).
//
// (1) A shell extension which calls any OLE API (i.e., linked with
//  OLE32.DLL) should call OLE's task allocator (by retrieving
//  the task allocator by calling CoGetMalloc API).
//
// (2) A shell extension which does not call any OLE API (i.e., not linked
//  with OLE32.DLL) should call the shell task allocator API (defined
//  below), so that the shell can quickly loads it when OLE32.DLL is not
//  loaded by any application at that point.
//
// Notes:
//  In next version of Windowso release, SHGetMalloc will be replaced by
// the following macro.
//
// #define SHGetMalloc(ppmem)   CoGetMalloc(MEMCTX_TASK, ppmem)
//
//=============================================================================
```

(Yes, those typos “guild” and “Windowso” have been there since 1995.)

This discussion strongly hints at what’s going on.

When Windows 95 was being developed, computers typically had just 4MB of memory. (The cool people got 8MB.) But Explorer was also heavily reliant upon COM for its shell extension architecture, and loading OLE32.DLL into memory was a significant kick in the teeth. Under such tight memory conditions, even the loss of 4K of memory was noticeable.

The solution: Play “OLE Chicken”.

The shell, it turns out, didn’t use very much of COM: The only objects it supported were in-process apartment-threaded objects with no marshalling. So the shell team wrote a “mini-COM” that supported only those operations and use it instead of the real thing. (It helped that one of the high-ranking members of the shell team was a COM super-expert.) The shell had its own miniature task allocator, its own miniature binder, its own miniature drag-drop loop, everything it needed **provided** you didn’t run any other programs that used OLE32.

Once some other program that used OLE32 started running, you had a problem: There were now two separate versions of OLE in the system: the real thing and the fake version inside the shell. Unless something was done, you wouldn’t be able to interoperate between real-

COM and fake-shell-COM. For example, you wouldn't be able to drag/drop data between Explorer (using fake-shell-COM) and a window that was using real-COM.

The solution: With the help of other parts of the system, the shell detected that "COM is now in the building" once anybody loaded OLE32.DLL, and it and transferred all the information it had been managing on its own into the world of real COM. Once it did this, all the shell pseudo-COM functions switched to real-COM as well. For example, once OLE32.DLL got loaded, calls to the shell's fake-task-allocator just went to the real task allocator.

But what is "OLE Chicken"? This is another variation of the various "chicken"-type games, perhaps the most famous of which is Schedule Chicken. In "OLE Chicken", each program would avoid loading OLE32.DLL as long as possible, so that it wouldn't be the one blamed for the long pause as OLE32.DLL got itself off the ground and ready for action. (Remember, we're talking 1995-era machines where allocating 32K would bring the wrath of the performance team upon your head.)

Okay, so let's look at that comment block again.

The opening paragraph mentions the possibility that a shell extension does not itself link with OLE32.DLL. Option (1) discusses a shell extension that does use OLE32, in which case it should use the official OLE functions like CoGetMalloc. But Option (2) discusses a shell extension that does not use OLE32. Those shell extensions are directed to use the shell's fake-COM functions like SHGetMalloc, instead of the real-COM functions, so that no new dependency on OLE32 is created. Therefore, if OLE32 is not yet loaded, loading these shell extensions will also not cause OLE32 to be loaded, thereby saving the cost of loading and initializing OLE32.DLL.

So the completion of our diagram for 1995-era programs would be something like this:

Before OLE32.DLL is loaded:

Shell task allocator				OLE task allocator		
SHAlloc/ SHFree	=	SHGetMalloc	≠	CoGetMalloc	=	CoTaskMemAlloc/ CoTaskMemFree

After OLE32.DLL is loaded:

Shell task allocator				OLE task allocator		
SHAlloc/ SHFree	=	SHGetMalloc	=	CoGetMalloc	=	CoTaskMemAlloc/ CoTaskMemFree

The final “Note” hints at the direction the shell intended to go. Eventually, loading OLE32.DLL would not be as painful as it was in Windows 95, and the shell can abandon its fake-COM and just use the real thing. At this point, asking for the shell task allocator would become the same as asking for the COM task allocator.

That time actually arrived a long time ago. The days of 4MB machines are now the stuff of legend. The shell has ditched its fake-COM and now just uses real-COM everywhere.

Therefore, **the diagram today is the one with the equals-sign**. All four functions are interchangeable in Windows XP and beyond.

What if you want to run on older systems? Well, it is always acceptable to use CoTaskMemAlloc/CoTaskMemFree. Why? You can puzzle this out logically. Since those functions are exported from OLE32.DLL, the fact that you are using them means that OLE32.DLL is loaded, at which point the “After” diagram above with the equals sign kicks in, and everything is all one big happy family.

The case where you need to be careful is if your DLL does **not** link to OLE32.DLL. In that case, you don’t know whether you are in the “Before” or “After” case, and you have to play it safe and use the shell task allocator for the things that are documented as using the shell task allocator.

I hope this discussion also provides the historical background of the function SHLoadOLE, which today doesn’t do anything because OLE is already always loaded. But in the old days, this signalled to the shell, “Okay, now is the time to brain-dump your fake-COM into the real-COM.”

Raymond Chen

Follow