MSDN Article ID: Q90745 - December 17, 1996

This article is from the Microsoft knowledge base regarding DLL's and how global data in handled on Win32 systems. For more detail visit the Microsft web site or check the current MSDN developer documentation.

Dynamic Loading of Win32 DLLs

When usingLoadLibrary() under Win16 or OS/2, the Dynamic Link Library (DLL) is loaded only once. Therefore, the DLL has the same address in all processes. Dynamic loading of DLLs is different under Windows NT.

A DLL is loaded separately for each process because each application has its own address space, unlike Win16 and OS/2. Pages must be mapped into the address space of a process. Therefore, it is possible that the DLL is loaded at different addresses in different processes. The memory manager optimizes the loading of DLLs so that if two processes share the same pages from the same image, they will share the same physical memory.

Each DLL has a preferred base address, specified at link time. If the address space from the base address to the base address plus image size is unavailable, then the DLL is loaded elsewhere and fixups will be applied. There is no way to specify a load address at load time.

To summarize, at load time the system:
1. Examines the image and determines its preferred base address and required size.
2. Finds the address space required and maps the image, copy-on-write, from the file.
3. Applies internal fixups if the image isn't at its preferred base.
4. Fixes up all dynamic link imports by placing the correct address for each imported function in the appropriate entry of the Import Address Table. This table is contiguous with 32-bit addresses, so 1024 imports require dirtying only one page.

The pages containing code are shared, using a copy-on-write scheme. The term copy-on-write means that the pages are read-only; however, when a process writes the page, instead of an access violation, the memory manager makes a private copy of the page and allows the write to proceed. For example, if two processes start from the same .EXE, both initially have all pages mapped from the .EXE copy-on-write. As the two processes proceed to modify pages, they get their own copies of the modified pages. The memory manager is free to optimize unmodified pages and actually map the same physical memory into the address space of both processes. Modified pages are swapped to/from the page file instead of the .EXE file.

There are two kinds of fixups. One is the address of an imported function. All these fixups are localized in what the Portable Executable (PE) specification calls the Import Address Table (IAT). This is an array of 32- bit function pointers, one for each imported API. The IAT is located on its own page(s), because it is always modified. Calling an imported function is actually an indirect call through the appropriate entry

in this array. In case that the image is loaded at the preferred address, the only fixups needed are for imports.

Note that there is an optimization whereby each import library exports a 32- bit number for each API along with any name and ordinal. This serves as a "hint" to speed the fixups performed at load time. If the hints in the program and the DLL do not match, the loader uses a binary search by name.

The other kind of fixup is needed for references to the image's own code or data when the image can't be loaded at its preferred address. When a page must be taken out of memory, the system checks to see whether the page has been modified. If it has not, then the page is still mapped copy-on-write against the EXE and can be discarded from memory. Otherwise, it must be written to the page file before it can be removed from memory, so that the page file is used as the backing store (where the page is recovered from) rather than the executable image file.

The DLL's entry point does not get called for a secondLoadLibrary() call in a process (that is, no second DLL_PROCESS_ATTACH entry). There is one call to DllEntry/DLL_THREAD_ATTACH per thread no matter the number of times a thread callsLoadLibrary(). The same goes for FreeLibrary (), but the DLL_PROCESS_DETACH happens only on the last call (that is, reference count back to zero for the process).

Global instance data for the DLL is on a per process basis (only one set per unique process). If it is necessary to ensure that global instance data is unique for eachLoadLibrary() performed in a single process, consider thread local storage (TLS) as an alternative. This requires multiple threads of execution, but TLS allows unique data for each ThreadID. There is very little overhead on the DLL's part; just create a global TLS index during process initialization. During thread initialization, allocate memory (via HeapAlloc(), GlobalAlloc(), LocalAlloc(), malloc(), and so on) and store a pointer to the memory using the global TLS index value in the function TlsSetValue. Win32 internally stores each thread's pointer by TLS index and ThreadID to achieve the thread specific storage.