

Re: Editors

Source: <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2004-03/0404.html>

From: Beth (*BethStone21_at_hotmail.NOSPICEDHAM.com*)

Date: 03/15/04

Date: Mon, 15 Mar 2004 01:46:42 -0000

TS wrote:

> >> > *Hmmm ... Than why do standard PE-files still have fix-up (aka :
> >> > relocation) tables ... :-)*
> >>
> >> *For DLL Data only. (And, i suppose for PE type Drivers as well,
> >> but i really do not know, actually).*
> >
> > *To be *really* sure, I just checked a few files. For some reason
my quick
> > inspection gives you right on Windows own executables, but not on
> > executables I generated myself. Have to look into that one*
>
> *It depends on the linker: Some include relocation info into all
kinds
> of PE files, some strip them on exe files by default. You may also
> strip relocation info from DLLs, but only if you can be 100% sure
that
> there won't be any overlap in linear address space.*

Yeah, there's usually a "switch" on the linker to say whether you want "relocations" or not...with LINK.EXE, the switch is "-fixed" and that drops the relocation information...

The "relocation information" is needed because of the possibility of DLLs being loaded outside their "preferred base address"...every PE is compiled to work at a specific "base address"...under Win32, though, every process - EXE file - will be given its own "address space" so, with EXEs, it basically can always fulfill the "base address" preference (if you're wondering, the "Win32s" stuff for old 16-bit Windows 3.x couldn't guarantee this when it's providing its Win32 interface and, hence, you might need to include "relocation information" for EXE files...so, the option to choose is left with PE EXEs, anyway...plus, of course, you can also have one EXE file load in another EXE file into the "address space" which would also mean that we're not guaranteed to get the address we expect in that address space and the "relocation information" allows the EXE or DLL file to be moved elsewhere to a space that is free :)...

Generally speaking, though, there's usually no point under Win32 for a typical EXE file to contain "relocations"...it's a bit of a waste of time to include it and you should "strip" it of this information (there's bound to be some kind of "switch" for this...with LINK.EXE, it's "-fixed" :)...as every process is given its own "address space" which is "empty" (except for the "system" stuff at the lower end of memory) then, under Win32, an EXE file like this will always get its "preferred base address" and, yup, relocations are completely unnecessary...

DLL files, though, aren't loaded into their own address space but are loaded into the main EXE process's address space...and many DLL files could all be loaded into that space...and their "preferred base addresses" could conflict...so that not all of the DLLs can be loaded into the address space where they want to be loaded...hence, DLL files should generally carry "relocations" to permit them to be shuffled elsewhere in memory, should their compiled "base address" already be occupied by the EXE or some other DLL (or a memory allocation the program made, if the DLL is loaded later rather than at load-time, then this is another possibility that appears :) or whatever...

Also, for better operation, it's advised – and if you look at Microsoft's DLL files, they follow this advice – that you try to magic up a "base address" that isn't likely to conflict with any other known DLL that you'll be using...the idea being that if a DLL can be loaded at its "preferred base address" then that's great, we can skip the "relocation" and it all goes much quicker when loading...the Microsoft system DLLs are basically all designed with different "base addresses" so that they don't conflict with each other and shouldn't ever need to be relocated in practice...if you're also coming up with a program that has a "suite" of DLL files, it's sensible to also take a similar approach of working out a bunch of "base addresses" for each one that doesn't conflict with any other DLL, including the system ones (although, these, if you look, are deliberately stuck high up in memory that this is unlikely to happen :)...)

Note, though, it's still possible that some other DLL sneaks its way into things that it's not completely "safe" once you do this to remove those relocations on DLLs entirely...the problem being that, say, a "hook" DLL latches onto "kernel32" to, say, implement your firewall or anti-virus or some keyboard hook or whatever...this means that in loading "Kernel32.dll" on such a system, other DLLs you might not be expecting could be loaded in too...and these may, indeed, be "conflicting" (more so in the sense that Kernel32 is likely to be one of the first DLLs linked up and the process of loading works "depth first" that anything "kernel32.dll" loads gets loaded first before moving onto another DLL...hence, you might find the space you were expecting is "occupied" :)...plus, of course, there's the point that the "base addresses" of Windows components isn't necessarily guaranteed..."XP 2" might move things all around, introduce some new

"system DLLs" that get automatically loaded by "Kernel32" without asking...and so on and so forth...

When in doubt, you need those "relocations" because, simply, if a DLL (or EXE but, as stated, as EXE processes are all given their own address spaces, this usually never happens and it is "safe" to strip these for most occasions ;) can't get the "base address" it wants and there's no "relocation information" present, the whole thing just bombs out...it simply won't run at all...hence, unless you have a specific reason to think otherwise, the general "rule of thumb" would be "strip EXE of relocations but always leave relocations in DLL files (plus, if you know that the EXE file is going to be loaded into the address space of some other EXE file – something possible but not usual practice under Win32 – then don't strip those either :)"...

It's a simple practical point..."relocation information" allows a file to be moved elsewhere in memory should the pre-compiled "preferred base address" be already occupied by something else..."relocation information" makes the file "moveable" so you include it whenever there's a possibility that you won't get the original "preferred" place in memory you want...you can leave "relocations" on EXEs and strip them from DLLs...whether this is a good idea in a particular case depends on what's happening...for instance, maybe a DLL file is deliberately designed for only one EXE to use in one situation and you've pre-calculated that it won't "conflict" then you could strip the information (mind you, if you're not "re-using" a DLL – which this situation suggests – you've got to wonder why you're making it into a DLL in the first place, anyway...why not just link it directly into the EXE file instead? It's this "logical" thing which creates the "rule of thumb" of "strip EXEs, put relocations in DLLs"...it's possible to do otherwise but you'd need a pretty particular situation for it to make much of any sense...most EXEs don't get loaded later into some other EXE's address space to need the relocation, most DLLs are destined to be "re-used" that they can't guarantee in all those different uses getting the address they expected to get ;)...

> >> A "normal PE" never need relocation, –not considering DLL Data–,
> >> and not considering an irregular default upload Address (provided,
> >> in the PE Header, on purpose, to force the loader to relocate at
> >> the proper Address)
>
> Any default base address below 4MB will force relocation.

Is it 4MB? Well, anyway, I do know that there's "system" stuff reserved at the bottom of any process' "address space" that, yup, locate your EXE or DLL too low in memory and it'll be forced to relocate because the very lowest addresses in an address space are automatically "reserved" by Windows itself for "system" purposes...being part of the "system", this stuff is always there for

every process...

- > *I'm not sure*
- > *if someone also used an executable also as a DLL – I've seen some*
- > *executables which export addresses, but these were still the first*
- PE
- > *to be loaded into address space.*

It can be done (some COM objects are inside EXE files rather than DLLs, for example)...it's just rare and unusual to have a situation where you'd want this to actually happen...so, generally speaking, unless you specifically want this, it's "okay" to strip relocations from EXE files...but Microsoft leave the option NOT to do so because you can treat EXEs like they were DLLs and load them in later...they are all PE files, after all...the difference between them, in fact, is really the "presumption" that the EXE file is the actual "main program" and, thus, is the "process" that owns the "address space" (hence, all the defaults for loading an EXE :)...this can be overridden and an EXE treated much like a DLL by specifically loading it...but, well, if you want a DLL, then write it as a DLL...it's a rare and unusual situation where you'd want an EXE but also want to treat it like a DLL...it can happen so the system has the support for that...but it's a "logical" kind of thing...you'd very, very rarely want to do so for any typical application that we have the "DLL" and "EXE" distinctions in the first place, so to speak...fundamentally, they are both PE files and only a flag or two inside the headers makes them any different (the file extension too, in a manner of speaking...although, the loader doesn't actually pay attention to that...it's a "visual clue" for the user...the loader goes only by what's in the headers...although, that said, on a practical note, trying to convince Explorer to actually run a renamed EXE file with a DLL extension is why that won't tend to work...in "loader" terms, it pays no attention to the extension and goes just by what the headers say :)...

- > *As far as I know there is an entry that get's called whenever the*
- DLL is
- > *loaded or unloaded by an app, as well when a new thread within that*
- app is
- > *created or closed (all with the correct arguments ofcourse, like*
- > *DLL_PROCESS_ATTACH). That gives the DLL to chance to create or*
- release
- > *data-storage for that app or thread.*
- >
- > *It's the normal program entry point, the same as used for starting a*
- n
- > *executable.*

Yes...and no; Yes, it's just the "entry-point" to the DLL...but it is "special" in that the entry-point has specific "parameters" sent to it and is treated as a "stdcall" procedure when it is called...an

EXE's "entry-point", though, is not "special" in that it's just an address that gets called...

The DLL "entry-point" is considered to be a "stdcall" procedure and takes three parameters (one of which is "reserved" and currently serves no purpose :)...the DLL's "HINSTANCE" because the DLL code is the same even when loaded into different places and an "HINSTANCE" tells "instances" apart...it also, conveniently, just so happens to be the "base address" of the DLL when it was loaded (well, that's an automatically "unique" value so it can be used to "dual purpose" as an "ID" for the instance, as well as the "base address" :)...

And it's the second parameter sent to the DLL "entry-point" that actually makes it quite different from an EXE...the EXE "entry-point" is simply called when the EXE is loaded...but the DLL "entry-point" is called for four events and the second parameter specifies which event: `DLL_PROCESS_ATTACH`, `DLL_THREAD_ATTACH`, `DLL_THREAD_DETACH` and `DLL_PROCESS_DETACH`...the reason for this is that an EXE is the "main" code, where it all starts...hence, the entry-point code can just be called...any "clean up" necessary can just be placed at the end of this code just before the "ExitProcess" call to finish up process...DLLs are different in this regard...they get loaded and unloaded but need a chance by which to initialise and clean-up...hence, the DLL "entry-point" is called with `DLL_PROCESS_ATTACH` when the DLL is first loaded into the process address space (a "global" initialisation opportunity), `DLL_THREAD_ATTACH` gives the DLL a chance to initialise for _each and every thread_ that attaches to the DLL (a "thread local" initialisation opportunity...threads may need their own "thread local" data so this call is made when a thread starts up to let the DLL have the opportunity to initialise things on a "per thread" basis), `DLL_THREAD_DETACH` to allow a "per thread" clean-up and `DLL_PROCESS_DETACH` to allow the "global" clean-up of a DLL just before it's removed from the address space completely...

Also, this "stdcall" entry-point should end with "ret" to pass control back to Windows – it is considered to be a _procedure_ 100% – while an EXE's entry-point is just called and it's "ExitProcess" that's used to terminate (does "ret" work? Maybe...but that's "implementation dependent"...you're not _supposed_ to do that and it has one of those "Microsoft reserves the right to totally screw that up by changing the implementation at any point" thing hanging over it ;)...

An EXE's entry-point is called once when the EXE loads...thereafter, the EXE is in control until it does an "ExitProcess"...a DLL's entry-point, on the other hand, is formally a "stdcall" procedure and may be called numerous times...not just once when it loads but also once when it unloads...as well as any number of times for _each and every thread_, both to allow initialising and cleaning-up for each thread...this is technically "indefinite", as a process could constantly start up and stop threads all the time, meaning the DLL

entry-point is called any number of times with DLL_THREAD_ATTACH and DLL_THREAD_DETACH...

It's this fundamentally different "entry-point" that, in a sense, makes a PE into "an EXE" or "a DLL" ...it's *_why_* you need to have the "DLL" flag in the header to let Windows know what type it is so that it can call the different types of entry point in the correct way...

Unlike "WinMain" – which *_IS_*, indeed, a pure "invention" for the sake of C compilers and the actual EXE entry-point has NO parameters and isn't even strictly a "procedure" without the "ret" – the "DllEntryPoint" routine *_does_* exist...even at the ASM level, you need to create it as a compatible "stdcall" routine and read the parameters in the corresponding way and it gets called multiple times for different events...in this instance, it's NOT a figment of a C compiler's imagination, DLLs really *_do_* need to have their entry-point's formatted in this way...so, no, it's not "just an entry-point"...indeed, *_WinMain_*, for sure, *_IS_* just some invented C compiler nonsense...but "DllEntryPoint" – whose actual name doesn't particularly matter because it's specified by the "entry-point address", not by any symbolic name in the headers – really does exist and needs to be formatted appropriately...

> >So, the DLL does not have to be reloaded, just mapped into the callers
> >code-space, and it will allocate what it needs for data-storage in that
> >callers data-space.
>
> Yes, but this also depends on the flags used in the PE sections.
> Basically, duplicating pages only as soon as a write occurs is a good
good
> memory preserving strategy...

Well, "reloaded" is the wrong term, anyway...it doesn't "reload", as in "load again"...it *_loads_* and then "relocates"...not quite the same thing...but let's not be too pedantic about the terminology, eh? So long as we *_know_* what we're talking about that there's no confusion, call it a "banana" for all I care about jargon! ;)

An EXE doesn't usually (though it *_is_* "possible" to make it otherwise, if you really insist ;) need to be "relocated" because it gets a fresh, empty "address space" (only Windows' own "system" stuff at the very bottom of the address space is not "free" for program use :)...hence, as long as it doesn't conflict with that reserved "system" stuff then it'll always get the "preferred base address" it asks for (at least, it does under true Win32...this is NOT actually guaranteed under "Win32s", which was the "Win32 extension" Microsoft came up with for 16-bit Win3.x systems to run some Win32 code...for Win32s, you *_do_* need the "relocations", even for EXE files :)...which is why it's "safe" to strip EXE files of relocations...

DLLs, though, get loaded and unloaded later – even possibly as the process is running and not even at load-time – in which case, it's possible that the "preferred base address" is already occupied by something else in that address space...the EXE, its memory allocations, another DLL, etc....hence, you would usually be sure to `_include_` "relocations" in this instance to allow the DLL file to be moved elsewhere in the "address space" where there's "free" memory (remembering also that the "address space" is "virtual" that you don't need free RAM for this...just free "address space" and it's physically mapped and loaded (and shuffled about if RAM gets used up ;) by the OS's memory manager stuff...

Although, yes, of course, this should probably be more correctly called "mapped" rather than "loaded" because it doesn't necessarily literally "load" unless pages actually get referenced...

Beth :)