WIKIPEDIA

Portable Executable

The **Portable Executable** (PE) format is a file format for executables, object code, DLLs and others used in 32-bit and 64-bit versions of Windows operating systems. The PE format is a data structure that encapsulates the information necessary for the Windows OS loader to manage the wrapped executable code. This includes dynamic library references for linking, API export and import tables, resource management data and thread-local storage (TLS) data. On NT operating systems, the PE format is used for EXE, DLL, SYS (device driver), MUI and other file types. The Unified Extensible Firmware Interface (UEFI) specification states that PE is the standard executable format in EFI environments.[2]

Portable Executable

Filename extension	`.acm, .ax, .cpl, .dll, .drv, .efi, .exe, .mui, .ocx, .scr, .sys, .tsp`
Internet media type	`application/vnd.microsoft.portable-executable`[1]
Developed by	Currently: Microsoft
Type of format	Binary, executable, object, shared libraries
Extended from	DOS MZ executable COFF

On Windows NT operating systems, PE currently supports the x86-32, x86-64 (AMD64/Intel 64), IA-64, ARM and ARM64 instruction set architectures (ISAs). Prior to Windows 2000, Windows NT (and thus PE) supported the MIPS, Alpha, and PowerPC ISAs. Because PE is used on Windows CE, it continues to support several variants of the MIPS, ARM (including Thumb), and SuperH ISAs. [3]

Analogous formats to PE are ELF (used in Linux and most other versions of Unix) and Mach-O (used in macOS and iOS).

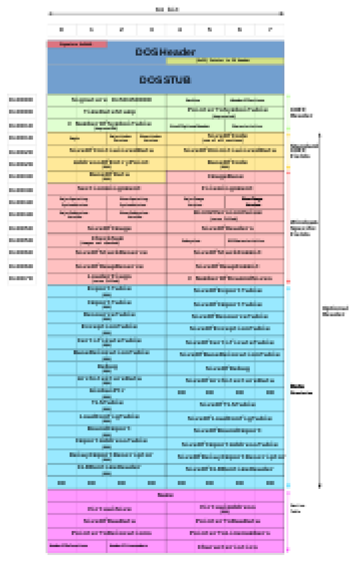
Contents

History

Microsoft migrated to the PE format from the 16-bit NE formats with the introduction of the Windows NT 3.1 operating system. All later versions of Windows, including Windows 95/98/ME and the Win32s addition to Windows 3.1x, support the file structure. The format has retained limited legacy support to bridge the gap between DOS-based and NT systems. For example, PE/COFF headers still include a DOS executable program, which is by default a DOS stub that displays a message like "This program cannot be run in DOS mode" (or similar), though it can be a full-fledged DOS version of the program (a later notable case being the Windows 98 SE installer).[4] This constitutes a form of fat binary. PE also continues to serve the changing Windows platform. Some extensions include the .NET PE format (see below), a 64-bit version called PE32+ (sometimes PE+), and a specification for Windows CE.

Technical details

Layout



Structure of a Portable Executable 32 bit

A PE file consists of a number of headers and sections that tell the dynamic linker how to map the file into memory. An executable image consists of several different regions, each of which require different memory protection; so the start of each section must be aligned to a page boundary.[5] For instance, typically the *.text* section (which holds program code) is mapped as execute/readonly, and the *.data* section (holding global variables) is mapped as no-execute/readwrite. However, to avoid wasting space, the different sections are not page aligned on disk. Part of the job of the dynamic linker is to map each section to memory individually and assign the correct permissions to the resulting regions, according to the instructions found in the headers.[6]

Import table

One section of note is the *import address table* (IAT), which is used as a lookup table when the application is calling a function in a different module. It can be in the form of both import by ordinal and import by name. Because a compiled program cannot know the memory location of the libraries it depends upon, an indirect jump is required whenever an API call is made. As the dynamic linker loads modules and joins them together, it writes actual addresses into the IAT slots, so that they point to the memory locations of the corresponding library functions. Though this adds an extra jump over the cost of an intra-module call resulting in a performance penalty, it provides a key benefit: The number of memory pages that need to be copy-on-write changed by the loader is minimized, saving memory and disk I/O time. If the compiler knows ahead of time that a call will be inter-module (via a dllimport attribute) it can produce more optimized code that simply results in an indirect call opcode.[6]

Relocations

PE files normally do not contain position-independent code. Instead they are compiled to a preferred *base address*, and all addresses emitted by the compiler/linker are fixed ahead of time. If a PE file cannot be loaded at its preferred address (because it's already taken by something else), the operating system will *rebase* it. This involves recalculating every absolute address and modifying the code to use the new values. The loader does this by comparing the preferred and actual load addresses, and calculating a delta value. This is then added to the preferred address to come up with the new address of the memory location. Base relocations are stored in a list and added, as needed, to an existing memory location. The resulting code is now private to the process and no longer shareable, so many of the memory saving benefits of DLLs are lost in this scenario. It also slows down loading of the module significantly. For this reason rebasing is to be avoided wherever possible, and the

DLLs shipped by Microsoft have base addresses pre-computed so as not to overlap. In the no rebase case PE therefore has the advantage of very efficient code, but in the presence of rebasing the memory usage hit can be expensive. This contrasts with ELF which uses fully position-independent code and a global offset table, which trades off execution time in favor of lower memory usage.

.NET, metadata, and the PE format

In a .NET executable, the PE code section contains a stub that invokes the CLR virtual machine startup entry, `_CorExeMain` or `_CorDllMain` in `mscoree.dll`, much like it was in Visual Basic executables. The virtual machine then makes use of .NET metadata present, the root of which, `IMAGE_COR20_HEADER` (also called "CLR header") is pointed to by `IMAGE_DIRECTORY_ENTRY_COMHEADER`[7] entry in the PE header's data directory. `IMAGE_COR20_HEADER` strongly resembles PE's optional header, essentially playing its role for the CLR loader.[3]

The CLR-related data, including the root structure itself, is typically contained in the common code section, `.text`. It is composed of a few directories: metadata, embedded resources, strong names and a few for native-code interoperability. Metadata directory is a set of tables that list all the distinct .NET entities in the assembly, including types, methods, fields, constants, events, as well as references between them and to other assemblies.

Use on other operating systems

The PE format is also used by ReactOS, as ReactOS is intended to be binary-compatible with Windows. It has also historically been used by a number of other operating systems, including SkyOS and BeOS R3. However, both SkyOS and BeOS eventually moved to ELF.

As the Mono development platform intends to be binary compatible with the Microsoft .NET Framework, it uses the same PE format as the Microsoft implementation. The same goes for Microsoft's own cross-platform .NET Core.

On x86(-64) Unix-like operating systems, Windows binaries (in PE format) can be executed with Wine. The HX DOS Extender also uses the PE format for native DOS 32-bit binaries, plus it can, to some degree, execute existing Windows binaries in DOS, thus acting like an equivalent of Wine for DOS.

On IA-32 and x86-64 Linux one can also run Windows' DLLs under loadlibrary.[8]

Mac OS X 10.5 has the ability to load and parse PE files, but is not binary compatible with Windows.[9]

UEFI and EFI firmware use Portable Executable files as well as the Windows ABI x64 calling convention for applications.

See also

- EXE
- Executable and Linkable Format
- Mach-O
- a.out
- Comparison of executable file formats
- Executable compression
- ar (Unix) since all COFF libraries use that same format

- Application virtualization

References

1. Andersson, Henrik (2015-04-23). "application/vnd.microsoft.portable-executable" (https://www.iana.org/assignments/media-types/application/vnd.microsoft.portable-executable). IANA. Retrieved 2017-03-26.
2. "UEFI Specification, version 2.8B" (https://uefi.org/sites/default/files/resources/UEFI%20Spec%202.8B%20May%202020.pdf) (PDF)., a note on p.15, states that "this image type is chosen to enable UEFI images to contain Thumb and Thumb2 instructions while defining the EFI interfaces themselves to be in ARM mode."
3. "PE Format (Windows)" (https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547(v=vs.85).aspx). Retrieved 2017-10-21.
4. E.g. Microsoft's linker has /STUB switch (http://msdn.microsoft.com/en-us/library/7z0585h5.aspx) to attach one
5. "The Portable Executable File From Top to Bottom" (http://www.csn.ul.ie/%7Ecaolan/pub/winresdump/winresdump/doc/pefile2.html). Retrieved 2017-10-21.
6. "Peering Inside the PE: A Tour of the Win32 Portable Executable File" (https://msdn.microsoft.com/en-us/library/ms809762.aspx). Retrieved 2017-10-21.
7. The entry was previously used for COM+ metadata in COM+ applications, hence the name
8. https://github.com/taviso/loadlibrary
9. Chartier, David (2007-11-30). "Uncovered: Evidence that Mac OS X could run Windows apps soon" (https://arstechnica.com/journals/apple.ars/2007/11/30/uncovered-evidence-that-mac-os-x-could-run-windows-apps-soon). *Ars Technica*. Retrieved 2007-12-03. "... Steven Edwards describes the discovery that Leopard apparently contains an undocumented loader for Portable Executables, a type of file used in 32-bit and 64-bit versions of Windows. More poking around revealed that Leopard's own loader tries to find Windows DLL files when attempting to load a Windows binary."

External links

- PE Format (https://docs.microsoft.com/en-us/windows/desktop/Debug/pe-format) (latest online document)
- Microsoft Portable Executable and Common Object File Format Specification (https://web.archive.org/web/20081208121446/http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx) (revision 8.1, OOXML format)
- Microsoft Portable Executable and Common Object File Format Specification (https://web.archive.org/web/20090126141159/http://download.microsoft.com/download/e/b/a/eba1050f-a31d-436b-9281-92cdfeae4b45/pecoff.doc) (revision 6.0, .doc format)
- The original Portable Executable article (http://msdn2.microsoft.com/en-us/library/ms809762.aspx) by Matt Pietrek (MSDN Magazine, March 1994)
- Part I. An In-Depth Look into the Win32 Portable Executable File Format (http://msdn.microsoft.com/en-us/magazine/cc301805.aspx) by Matt Pietrek (MSDN Magazine, February 2002)
- Part II. An In-Depth Look into the Win32 Portable Executable File Format (https://web.archive.org/web/20120915093039/http://msdn.microsoft.com/en-us/magazine/cc301808.aspx) by Matt Pietrek (MSDN Magazine, March 2002)
- The .NET File Format by Daniel Pistelli (https://archive.today/20130130042959/http://www.ntcore.com/files/dotnetformat.htm)
- Ero Carrera's blog describing the PE header and how to walk through (http://blog.dkbza.org/)

- PE Internals provides an easy way to learn the Portable Executable File Format (http://www.andreybazhan.com/pe-internals/)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Portable_Executable&oldid=1004008277"