Языки. Компиляторы. Линковка. Исполняемые модули.

Виртуальная машина

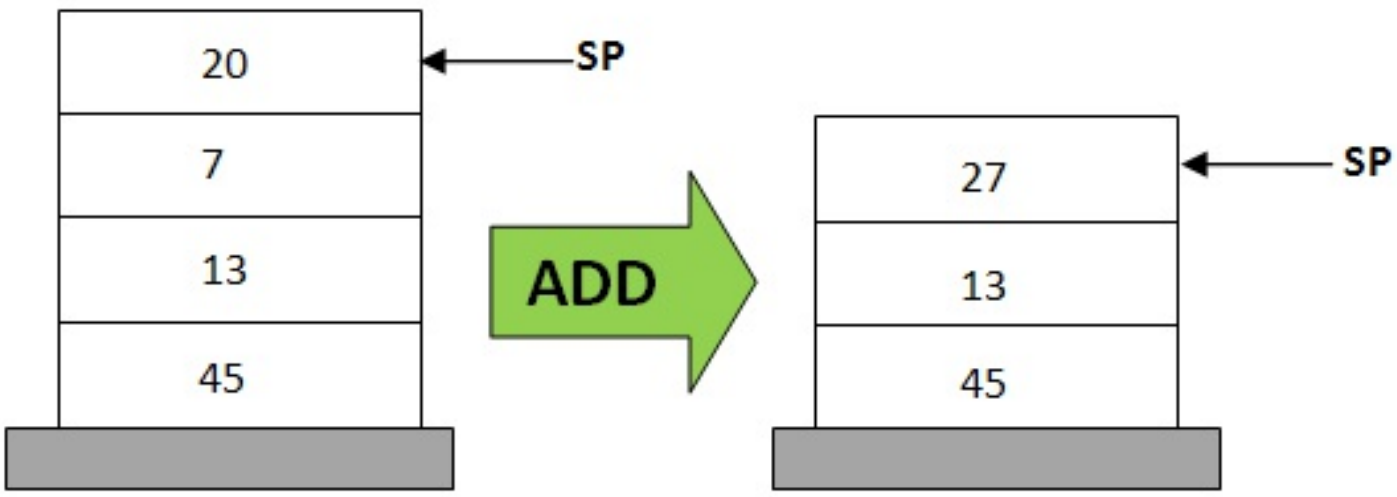
Виртуальная машина - программная и/или аппаратная система, эмулирующая аппаратное обеспечение некоторой платформы (`target` — целевая, или гостевая платформа) и исполняющая программы для `target`-платформы на `host`-платформе (`host` — хост-платформа, платформа-хозяин) или виртуализирующая некоторую платформу и создающая на ней среды, изолирующие друг от друга программы и даже операционные системы; также спецификация некоторой вычислительной среды (например: “виртуальная машина языка программирования Си”).

Виртуальная машина исполняет некоторый машинно-независимый код (например, байт-код, шитый код, p-код) или машинный код реального процессора. Помимо процессора, ВМ может эмулировать работу как отдельных компонентов аппаратного обеспечения, так и целого реального компьютера (включая `BIOS`, оперативную память, жёсткий диск и другие периферийные устройства). В последнем случае в ВМ, как и на реальный компьютер, можно устанавливать операционные системы (например, `Windows` можно запускать в виртуальной машине под `Linux` или наоборот). На одном компьютере может функционировать несколько виртуальных машин (это может использоваться для имитации нескольких серверов на одном реальном сервере с целью оптимизации использования ресурсов сервера).

Материал частично позаимствован с статьи Стековая и регистровая архитектура виртуальной машины и `Dalvik` VM

Стековая виртуальная машина

Стековая виртуальная машина реализует основные, выше описанные свойства виртуальной машины, но в качестве структуры данных, куда помещаются операнды, используется стек. Операции получают данные из стека, обрабатывают их и заносят в стек результат по правилу `LIFO` (последний пришел, первый ушел). В стековой виртуальной машине, операция сложения двух чисел должна выполняться следующим способом (где 20, 7, и “результат” – операнды):



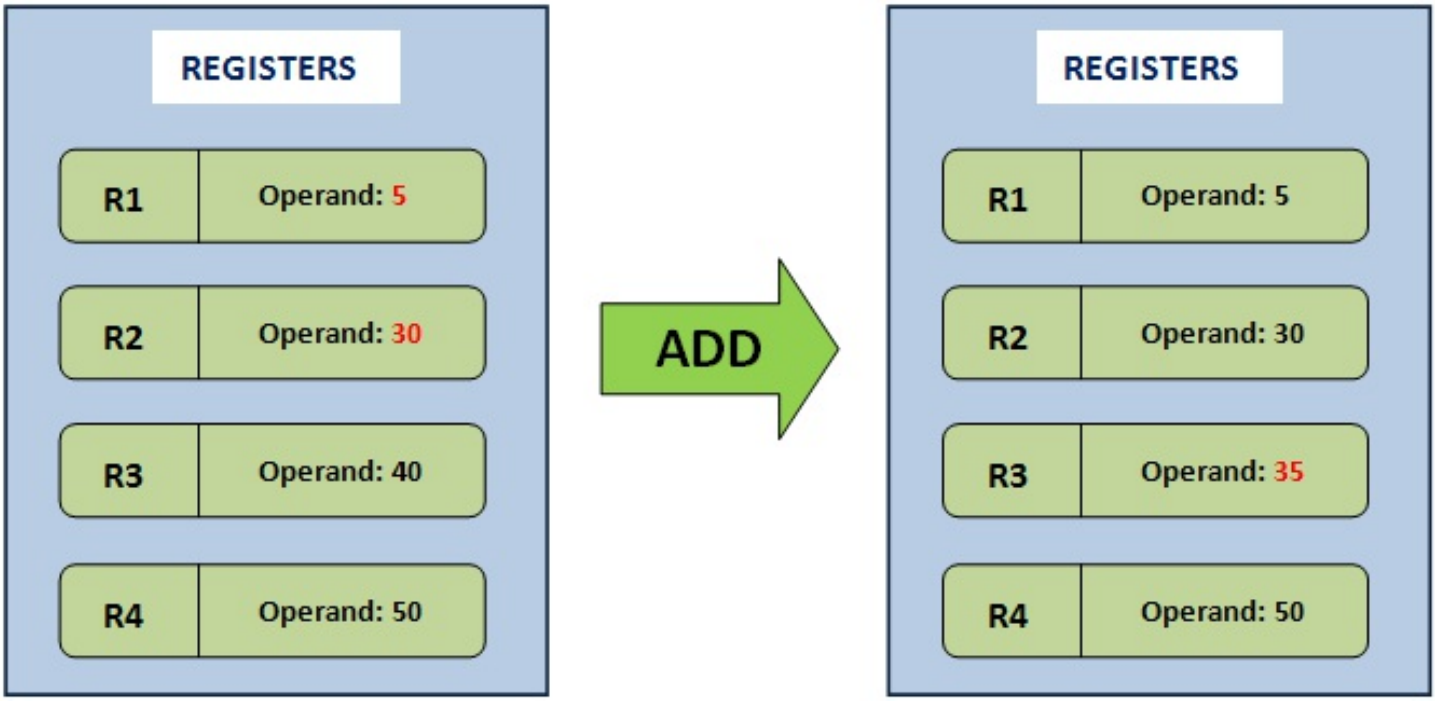

```
POP   20
POP   7
ADD   20, 7, result
PUSH  result
```

Из-за операций `PUSH` и `POP` для операции сложения требуется 4

инструкции. Преимущество стековой модели в том, что операнды задаются неявно указателем стека (на рисунке – `SP`). Это означает, что виртуальной машине не нужно явно указывать адреса операндов, указатель стека указывает на следующий операнд. В стековых виртуальных машинах все арифметические и логические операции выполняются посредством получения операндов и возврата результатов в стек.

Регистровые виртуальные машины

В регистровой реализации виртуальной машины структура данных, в которую помещаются операнды, основана на регистрах процессора. При этом не требуются операции `PUSH` или `POP`, но инструкции должны явно содержать адреса (регистры) в которых содержатся операнды. То есть, операнды для инструкций, в отличии от стековой модели, указываются явно. Например, операция сложения в регистровой виртуальной машине выглядит приблизительно так:



```
ADD R1, R2, R3; # складывает содержимое R1 и R2, результат  з
аносит в R3
```

За счет отсутствия операций `POP` и `PUSH` команды в регистровой виртуальной машине выполняются быстрее аналогичных команд стековой виртуальной машины.

Другое преимущество регистровой модели в том что она позволяет провести оптимизацию, которая не может быть выполнена при стековом подходе. Например несколько раз встречающееся выражение при регистровом подходе может быть вычислено лиш однажды и сохранено в регистре для последующего использования, что экономит время необходимое для пересчета выражения.

Но с другой стороны в среднем инструкция регистровой машины длиннее чем в стековой машине, так как в ней тредуется явное указание операндов.

Нативное приложение

Главное преимущество нативных приложений - то, что они оптимизированы под конкретные операционные системы, а значит работают корректно и быстро. Также они имеют доступ к аппаратной части устройств, то есть могут использовать в своём функционале камеру смартфона, микрофон, акселерометр, геолокацию, адресную книгу, плеер и т.д.

Компиляция, интерпритация.

Компиляция - сборка программы, включающая трансляцию всех модулей программы, написанных на одном или нескольких исходных языках программирования высокого уровня и/или языке ассемблера, в эквивалентные программные модули на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера) или непосредственно на машинном языке или ином двоичнокодовом низкоуровневом командном языке и последующую сборку исполняемой машинной программы. Если компилятор генерирует исполняемую машинную программу на машинном языке, то такая программа непосредственно исполняется физической программируемой машиной (например компьютером). В других случаях исполняемая машинная программа выполняется соотвествующей виртуальной машиной. Входной информацией для компилятора (исходный код) является описание алгоритма или программы на предметно-ориентированном языке, а на выходе компилятора - эквивалентное описание алгоритма на машинно-ориентированном языке (объектный код).

Интерпретация - построчный анализ, обработка и выполнение исходного кода программы или запроса (в отличие от компиляции, где весь текст программы, перед запуском, анализируется и транслируется в машинный или байт-код, без её выполнения). Интерпретатор компилирующего типа - это система из компилятора, переводящего исходный код программы в промежуточное представление, например, в байт-код или p-код, и собственно

интерпретатора, который выполняет полученный промежуточный код (так называемая виртуальная машина). Достоинством таких систем является большее быстродействие выполнения программ (за счёт выноса анализа исходного кода в отдельный, разовый проход, и минимизации этого анализа в интерпретаторе).

Из-за необходимости интерпретации байт-код выполняется значительно медленнее машинного кода сравнимой функциональности, однако он более переносим (не зависит от операционной системы и модели процессора). Чтобы ускорить выполнение байт-кода, используется динамическая компиляция, когда виртуальная машина транслирует псевдокод в машинный код непосредственно перед его первым исполнением (и при повторных обращениях к коду исполняется уже скомпилированный вариант).

Наиболее популярной разновидностью динамической компиляции является `JIT`

Стадии компиляции.

1. Трансляция программы - трансляция всех или только изменённых модулей исходной программы
2. Компановка машинно-ориентированной программы.

Трансляция программы как неотъемлемая составляющая компиляции включает в себя:

1. Лексический анализ. На этом этапе последовательность

символов исходного файла преобразуется в последовательность лексем.
2. Синтаксический (грамматический) анализ. Последовательность лексем преобразуется в дерево разбора.
3. Семантический анализ. Дерево разбора обрабатывается с целью установления его семантики (смысла) - например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т. д. Результат обычно называется “промежуточным представлением/кодом”, и может быть дополненным деревом разбора, новым деревом, абстрактным набором команд или чем-то ещё, удобным для дальнейшей обработки.
4. Оптимизация. Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла. Оптимизация может быть на разных уровнях и этапах - например, над промежуточным кодом или над конечным машинным кодом.
5. Генерация кода. Из промежуточного представления порождается код на целевом машинно-ориентированном языке.

Линковка. Статическая, динамическая.

Линковка - это построение из объектного модуля(модулей) исполняемый модуль.

Для связывания модулей компоновщик использует таблицы символов, созданные компилятором в каждом из объектных модулей. Эти таблицы могут содержать символы следующих типов:

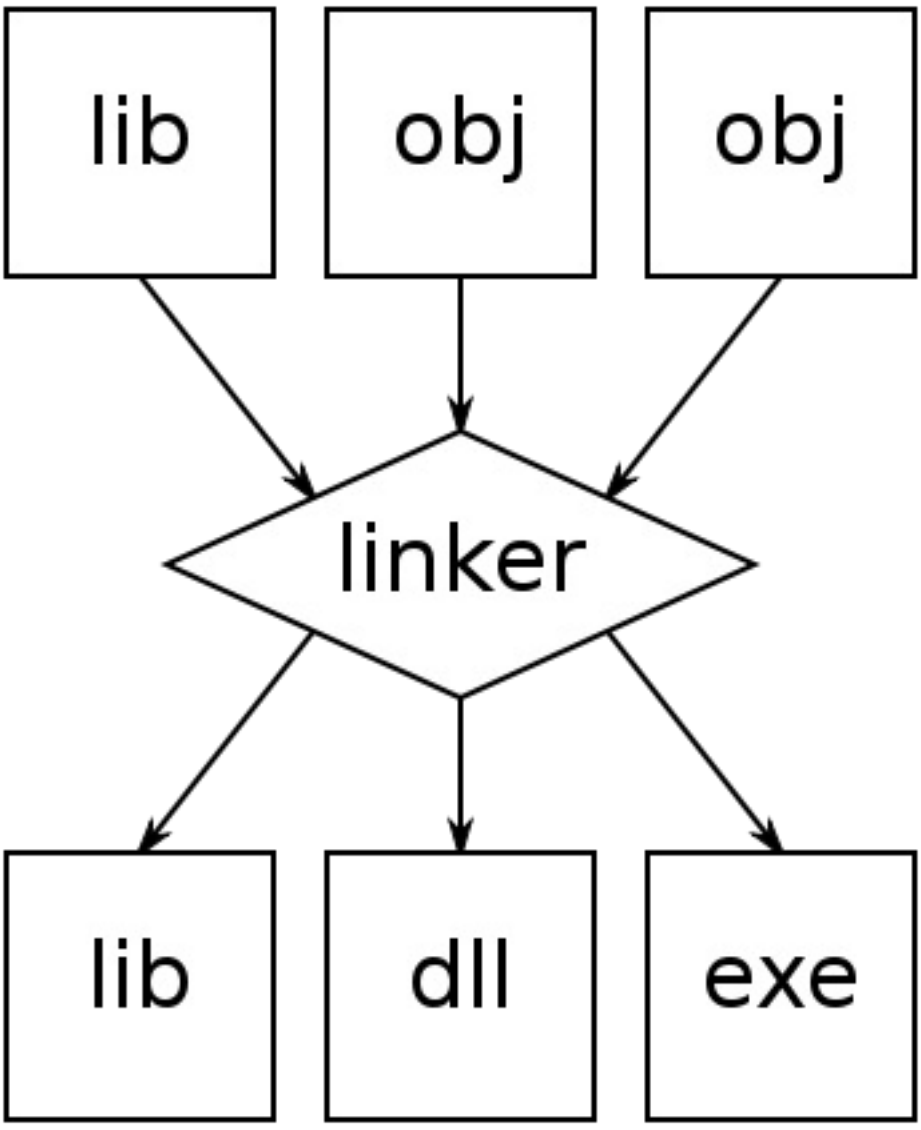
1. Определённые или экспортируемые имена - функции и переменные, определённые в данном модуле и предоставляемые для использования другим модулям;
2. Неопределённые или импортируемые имена - функции и переменные, на которые ссылается модуль, но не определяет их внутри себя;
3. Локальные — могут использоваться внутри объектного файла для упрощения процесса настройки адресов.

Для большинства компиляторов, один объектный файл является результатом компиляции одного файла с исходным кодом. Если программа собирается из нескольких объектных файлов, компоновщик собирает эти файлы в единый исполнимый модуль, вычисляя и подставляя адреса вместо символов, в течение времени компоновки (статическая компоновка) или во время исполнения (динамическая компоновка).

Компоновщик может извлекать объектные файлы из специальных коллекций, называемых библиотеками. Если не все символы, на которые ссылаются пользовательские объектные файлы, определены, то компоновщик ищет их определения в библиотеках, которые пользователь подал ему на вход. Обычно, одна или несколько системных библиотек используются компоновщиком по умолчанию. Когда объектный файл, в котором содержится определение какого-либо искомого символа, найден, компоновщик может включить его (файл) в исполнимый модуль (в случае статической компоновки) или отложить это до момента запуска программы (в случае динамической компоновки).

Работа компоновщика заключается в том, чтобы в каждом модуле определить и связать ссылки на неопределённые имена. Для каждого импортируемого имени находится его определение в других модулях, упоминание имени заменяется на его адрес.

Компоновщик обычно не выполняет проверку типов и количества параметров процедур и функций. Если надо объединить объектные модули программ, написанные на языках со строгой типизацией, то необходимые проверки должны быть выполнены дополнительной утилитой перед запуском редактора связей.



Автор изображения

Динамически разделяемые библиотеки.

`DLL` - в операционных системах `Microsoft Windows` и `IBM OS/2` - динамическая библиотека, позволяющая многократное использование различными программными приложениями. Эти библиотеки обычно имеют расширение `DLL`, `OCX` (для библиотек содержащих `ActiveX`), или `DRV` (для ряда системных драйверов). Формат файлов для `DLL` такой же, как для `EXE` файлов `Windows`, т.е. `Portable Executable` (`PE`) для 32-х битных и 64-х битных `Windows` приложений, и `New Executable` (`NE`) для 16-битных `Windows`-приложений. Так же, как `EXE`, `DLL` могут содержать секции кода, данных и ресурсов. В системах `UNIX` аналогичные функции выполняют так называемые общие объекты (англ. `shared objects`).

Файлы данных с тем же форматом как у `DLL`, но отличающиеся расширением, или содержащие только секцию ресурсов, могут быть названы ресурсными `DLL`. В качестве примера можно назвать библиотеки иконок, иногда имеющие расширение `ICL`, и файлы шрифтов, имеющих расширение `FON` и `FOT`.

Первоначально предполагалось, что введение `DLL` позволит эффективно организовать память и дисковое пространство, используя только один экземпляр библиотечного модуля для различных приложений. Это было особенно важно для ранних версий `Microsoft Windows` с жёсткими ограничениями по памяти.

Далее предполагалось улучшить эффективность разработок и использования системных средств за счёт модульности. Замена `DLL`-программ с одной версии на другую должна была позволить независимо наращивать систему, не затрагивая приложений. Кроме того, динамические библиотеки могли использоваться разнотипными приложениями - например, `Microsoft Office`, `Microsoft Visual Studio` и т. п.

В дальнейшем идея модульности выросла в концепции `Component Object Model` и `System Object Model`.

Фактически полных преимуществ от внедрения динамически подключаемых библиотек получить не удалось по причине явления, называемого `DLL hell`. `DLL hell` возникает, когда несколько приложений требуют одновременно различные, не полностью совместимые версии библиотек, что приводит к сбоям в этих приложениях и к конфликтам типа `DLL hell`, резко снижая общую надёжность операционных систем. Поздние версии `Microsoft Windows` стали разрешать параллельное использование разных версий `DLL` (технология `Side-by-side assembly`), что свело на нет преимущества изначального принципа модульности.

Существует так же ряд утилит, которые позволяют отследить зависимости приложений от подключаемых `DLL`. К примеру `see_dll` из комплекта `Microsoft Visual Studio`.

Исполняемые модули. Секции. Импорт,

экспорт. Типы исполняемых модулей. PE

`Portable Executable` (`PE`, “переносимый исполняемый”) - формат исполняемых файлов, объектного кода и динамических библиотек, используемый в 32- и 64-разрядных версиях операционной системы `Microsoft Windows`. Формат `PE` представляет собой структуру данных, содержащую всю информацию, необходимую `PE`-загрузчику для отображения файла в память. Исполняемый код включает в себя ссылки для связывания динамически загружаемых библиотек, таблицы экспорта и импорта `API` функций, данные для управления ресурсами и данные локальной памяти потока (`TLS`). В операционных системах семейства `Windows NT` формат `PE` используется для `EXE`, `DLL`, `SYS` (драйверов устройств) и других типов исполняемых файлов.

`PE` представляет собой модифицированную версию `COFF` формата файла для `Unix`. `PE/COFF` - альтернативный термин при разработке `Windows`.

На операционных системах семейства `Windows NT` формат `PE` в настоящее время поддерживает следующие архитектуры наборов команд: `IA-32`, `IA-64`, и `x86-64` (`AMD64/Intel64`). До `Windows 2000` `Windows NT` (таким образом, и `PE`) поддерживал `MIPS`, `Alpha`, и `PowerPC`. Поскольку `PE` используется на `Windows CE`, он продолжает поддерживать несколько разновидностей `MIPS`, `ARM` (включая `Thumb`), и `SuperH`.

Сигнатура

Первые 2 байта `PE` файла содержат сигнатуру `0x4D 0x5A` — `MZ` (как наследник `MZ`-формата). Далее двойное слово по смещению `0x3C` содержит адрес `PE`-заголовка. Последний начинается с сигнатуры `0x50 0x45` - `PE`.

Структура

Файл `PE` состоит из нескольких заголовков и секций, которые указывают динамическому компоновщику, как отображать файл в память. Исполняемый образ состоит из нескольких различных областей (секций), каждая из которых требует различных прав доступа к памяти; таким образом, начало каждой секции должно быть выровнено по границе страницы. Например, обычно секция `.text`, которая содержит код программы, отображена как исполняемая/доступная только для чтения, а секция `.data`, содержащая глобальные переменные, отображена как неисполняемая/доступная для чтения и записи. Однако, чтобы не тратить впустую пространство на жёстком диске, различные секции на нём на границу страницы не выровнены. Часть работы динамического компоновщика состоит в том, чтобы отобразить каждую секцию в память отдельно и присвоить корректные права доступа получившимся областям согласно указаниям, содержащимся в заголовках.

Таблица импорта

Одна из известных секций - таблица адресов импорта (`IAT` - `Import Address Table`), которая используется в качестве таблицы поиска, когда приложение вызывает функцию из другого модуля. Это может

быть сделано и в форме импорта по порядковому номеру функции (`ordinal`), и импорта по её имени. Поскольку скомпилированной программе неизвестно расположение библиотек, от которых она зависит, то требуется производить косвенный переход всякий раз, когда происходит вызов `API`-функции. Когда динамический компоновщик загружает модули и объединяет их, он записывает действительные адреса в область `IAT` так, чтобы они указали на ячейки памяти соответствующих библиотечных функций. Хотя это добавляет дополнительный переход внутри модуля, приводящий к потере производительности, это предоставляет ключевое преимущество: количество страниц памяти, которые должны быть скопированы загрузчиком при записи, минимизировано, что приводит к экономии памяти и дискового времени ввода-вывода. Если компилятору будет известно заранее, что вызов будет межмодульным (через атрибут `dllimport`), то он сможет произвести более оптимизированный код, который просто приводит к коду операции косвенного вызова.

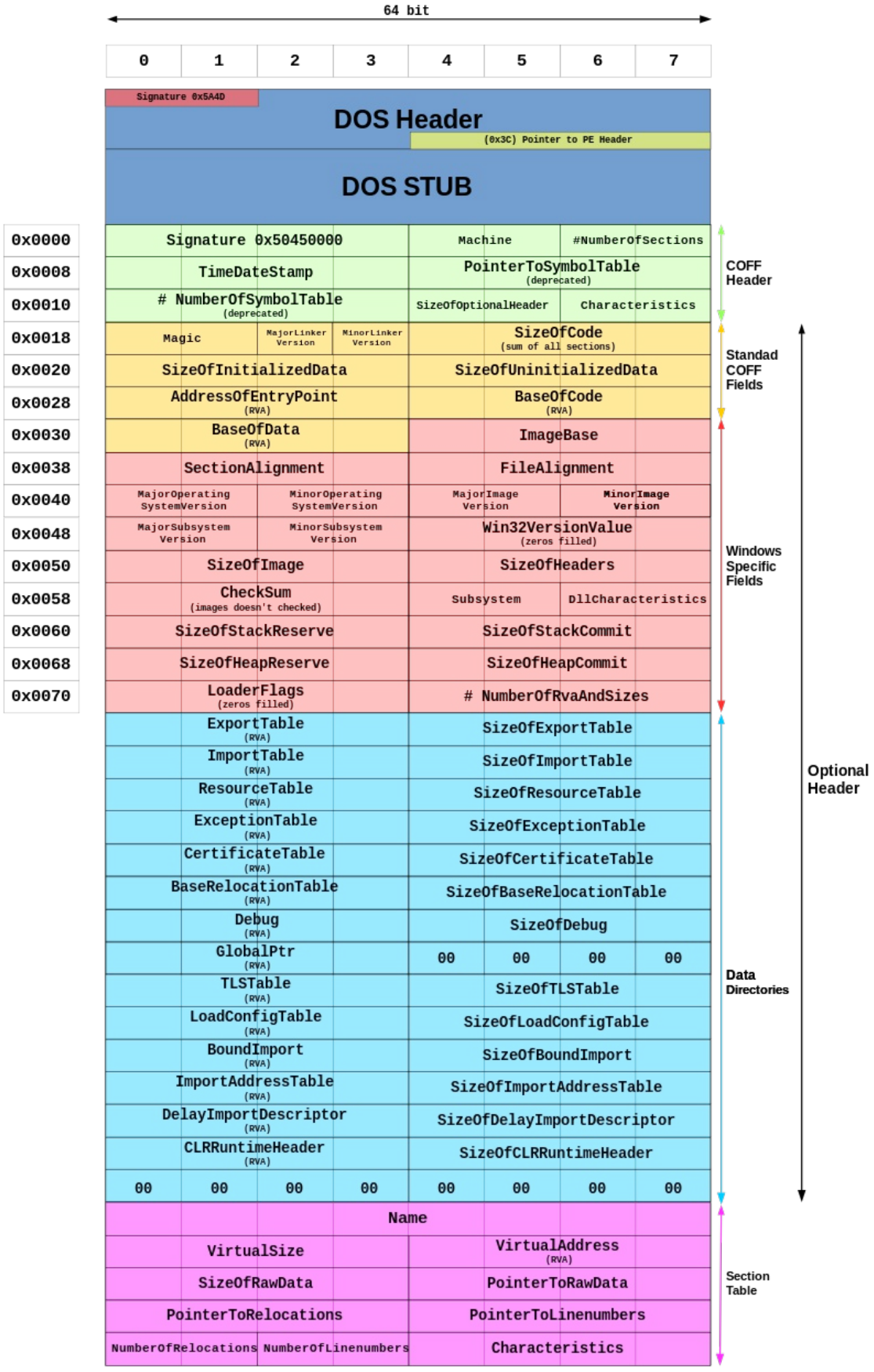
Таблица экспорта

Таблица адресов экспорта (`EAT` - `Export Address Table`) нужна для того, чтобы один модуль (обычно это динамически загружаемая библиотека) мог указать другим модулям, какие функции они могут из него импортировать, и по каким адресам последние расположены.

Таблица перемещений

Файлы `PE` не содержат позиционно-независимого кода. Вместо этого они скомпилированы для предпочтительного базового адреса, и все адреса, генерируемые компилятором/компоновщиком, заранее

фиксированы. Если `PE`-файл не может быть загружен по своему предпочтительному адресу (потому что он уже занят чем-то ещё), операционная система будет перебазировать его. Это включает в себя перевычисление каждого абсолютного адреса и изменение кода для того, чтобы использовать новые значения. Загрузчик делает это, сравнивая предпочтительный и фактический адреса загрузки, и вычисляя значение разности. Тогда для получения нового адреса ячейки памяти эта разность складывается с предпочтительным адресом. Базовые адреса перемещений хранятся в списке и при необходимости добавляются к существующей ячейке памяти. Полученный код является теперь отдельным по отношению к процессу и не является больше разделяемым, так что при таком способе теряются многие из преимуществ экономии памяти динамически загружаемых библиотек. Такой способ также значительно замедляет загрузку модуля. По этой причине следует избегать перебазирования везде, где это возможно; например, библиотеки, поставляемые `Microsoft`, имеют предварительно вычисленные неперекрывающиеся базовые адреса. В случае отсутствия необходимости перебазировании `PE`-файлы имеют преимущество очень эффективного кода, но при наличии перебазирования издержки в использовании памяти могут быть значительными. Это отличает формат `PE` от `ELF`, который использует полностью позиционно-независимый код и глобальную таблицу смещений, которая жертвует временем выполнения в пользу расходования памяти.


64 bit
0
1
2
3
4
5
6
7
Signature 0x5A4D
DOS Header
(0x3C) Pointer to PE Header
DOS STUB
0x0000
0x0008
0x0010
0x0018
0x0020
0x0028
0x0030
0x0038
0x0040
0x0048
0x0050
0x0058
0x0060
0x0068
0x0070
Signature 0x50450000
Machine
#NumberofSections
TimeDateStamp
PointerToSymbolTable
(deprecated)
NumberOfSymbolTable
(deprecated)
SizeOfOptionalHeader
Characteristics
Magic
MajorLinker Version
MinorLinker Version
SizeOfCode
(sum of all sections)
SizeOfInitializedData
SizeOfUninitializedData
AddressOfEntryPoint
(RVA)
BaseOfCode
(RVA)
BaseOfData
(RVA)
ImageBase
SectionAlignment
FileAlignment
MajorOperating SystemVersion
MinorOperating SystemVersion
MajorImage Version
MinorImage Version
MajorSubsystem Version
MinorSubsystem Version
Win32VersionValue
(zeros filled)
SizeOfImage
SizeOfHeaders
CheckSum
(images doesn't checked)
Subsystem
DllCharacteristics
SizeOfStackReserve
SizeOfStackCommit
SizeOfHeapReserve
SizeOfHeapCommit
LoaderFlags
(zeros filled)
NumberOfRvaAndSizes
ExportTable
(RVA)
SizeOfExportTable
ImportTable
(RVA)
SizeOfImportTable
ResourceTable
(RVA)
SizeOfResourceTable
ExceptionTable
(RVA)
SizeOfExceptionTable
CertificateTable
(RVA)
SizeOfCertificateTable
BaseRelocationTable
(RVA)
SizeOfBaseRelocationTable
Debug
(RVA)
SizeOfDebug
GlobalPtr
(RVA)
00 00 00 00
TLSTable
(RVA)
SizeOfTLSTable
LoadConfigTable
(RVA)
SizeOfLoadConfigTable
BoundImport
(RVA)
SizeOfBoundImport
ImportAddressTable
(RVA)
SizeOfImportAddressTable
DelayImportDescriptor
(RVA)
SizeOfDelayImportDescriptor
CLRRuntimeHeader
(RVA)
SizeOfCLRRuntimeHeader
00 00 00 00 00 00 00 00
Name
VirtualSize
VirtualAddress
(RVA)
SizeOfRawData
PointerToRawData
PointerToRelocations
PointerToLinenumbers
NumberOfRelocations
NumberOfLinenumbers
Characteristics
COFF Header
Standad COFF Fields
Windows Specific Fields
Optional Header
Data Directories
Section Table

Автор изображения

Данный материал был взять с `WiKi` Portable Executable

Порядок загрузки исполняемого модуля.

1. Считать данные из запускаемого файла.
2. Если необходимо - загрузить в память недостающие динамические библиотеки.
3. Заменить в коде новой программы относительные адреса и символические ссылки на точные, с учётом текущего размещения в памяти, то есть выполнить связывание адресов (`Relocation`).
4. Создать в памяти образ нового процесса и запланировать его к исполнению.

Базовый адрес.