***ЛЕКЦИЯ 4!!!!!!!!!!!!!***

***Структура распределения оперативной памяти.***

**Распределение памяти в ПК (Разделы ОЗУ)**

**RAM устроена довольно сложно, она иерархична (многоэтажна), её разделяют на несколько типов. Деление это обусловлено историческими причинами и тем, что ранее созданные программы должны исполняться и на компьютерах с «современной» 64-разрядной архитектурой. Выделяют 4 вида памяти:**

* **Стандартная (conventional memory area)**
* **Верхняя (upper memory blocks(area))**
* **Дополнительная (expanded memory )**
* **Расширенная (extended memory )**

***Стандартная (conventional memory area)*– первые 640 Кб, также его часто называют lower. В младшие адреса этой памяти загружается программная составляющая ОС и так называемые «драйверы устройств». Оставшуюся свободную часть памяти занимают пользовательские программы. Резидентные программы также остаются в этой памяти.**

***Верхняя (upper memory аrea)*– 640Кб — 1Мб используется для хранения служебной информации: памяти видеоадаптера, BIOS. Специальные драйвера позволяют загружать в свободные участки этой области резидентные программы и драйвера устройств.**

***High memory* – первые 64 Кб после 1Мб. В некоторых ОС в эту часть ОП загружается некоторая часть так называемых «резидентных» программ, освобождая при этом существенную часть базовой памяти для работы прикладных программ. Особенно это полезно для программ, использующих всю ОП. Вся память свыше 1 Мб может быть рассмотрена как дополнительная (expanded) или как расширенная (extended). В ОС менеджер памяти позволяет использовать память и как расширенную и как дополнительную, автоматически обеспечивая тот тип взаимодействия с данными, который нужен прикладным программам. Т.е. пользователю новых современных ПК нет необходимости распределять память «вручную», менеджер выделит память таким образом, как это требует прикладная программа.**

**Дополнительная(expanded) память – постраничная, т.е. ОП разбивается на страницы, каждой странице ставится в соответствие определенный адрес в основной памяти. Расширенная (extended) память построчной организации используется для создания временного логического диска (виртуального диска), как буфер обмена с жестким диском.**

**Распределение ОП в ПК**

|  |  |  |
| --- | --- | --- |
| **1Mб+ 64Кб** | **High** | **Расширенная или дополнительная память** |
| **Резидентные программы и драйверы устройств** |
| **Часть ОС** |
| **1Mб** | **Upper** | **Верхняя память ПЗУ BIOS** |
| **Видеопамять (текстовый буфер)** |
| **Видеопамять (графический буфер)** |
| **640Кб** | **Convertional Memory Area (base)Стандартная (базовая память)** | **Свободная часть транзитная часть** |
| **Свободная часть для программ пользователя** |
| **(резидентная часть)** |
| **Программы OS, драйверы** |
|  |
| **Данные для OS и BIOS, другая служебная информация** |
|  |  |  |
|  |  |  |

Explanation 401.

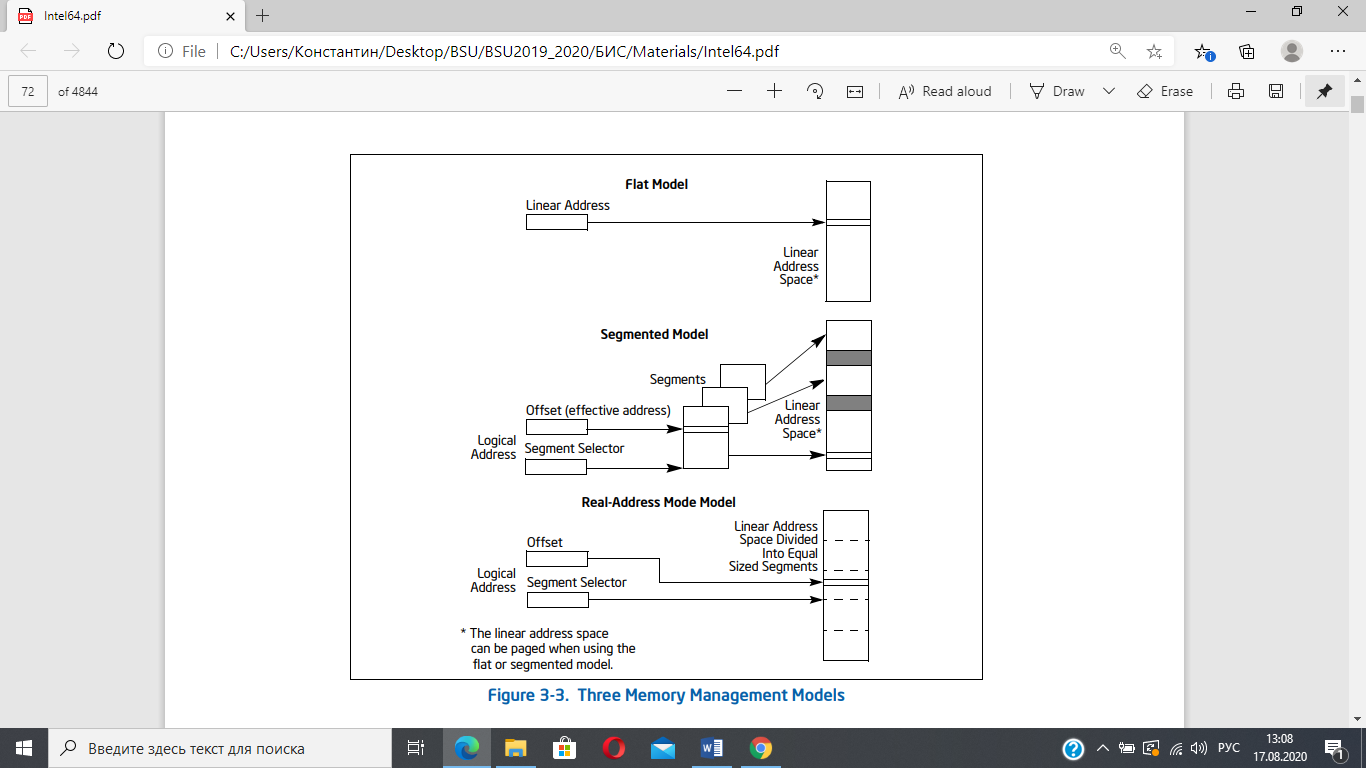
Memory Models.

When employing the processor’s memory management facilities, programs do not directly address physical memory. Instead, they access memory using one of three memory models: flat, segmented, or real address mode:

• Flat memory model — Memory appears to a program as a single, continuous address space. This space is called a linear address space. Code, data, and stacks are all contained in this address space. Linear address space is byte addressable. An address for any byte in linear address space is called a linear address.

• Segmented memory model — Memory appears to a program as a group of independent address spaces called segments. Code, data, and stacks are typically contained in separate segments. To address a byte in a segment, a program issues a logical address. This consists of a segment selector and an offset (logical addresses are often referred to as far pointers). The segment selector identifies the segment to be accessed and the offset identifies a byte in the address space of the segment.. To access a memory location, the processor thus translates each logical address into a linear address. This translation is transparent to the application program. The primary reason for using segmented memory is to increase the reliability of programs and systems. For example, placing a program’s stack in a separate segment prevents the stack from growing into the code or data space and overwriting instructions or data, respectively.

• Real-address mode memory model — This is the memory model for the Intel 8086 processor. It is supported to provide compatibility with existing programs written to run on the Intel 8086 processor. The real-address mode uses a specific implementation of segmented memory in which the linear address space for the program and the operating system/executive consists of an array of segments of up to 64 KBytes in size each.



Упражнение 401. – Как только дойдёте до этого места, ПЕРЕВЕДИТЕ НА РУССКИЙ ЯЗЫК НЕ ИСПОЛЬЗУЯ «ПЕРЕВОДЧИКОВ»!!!!! И ПЕРЕВОД ПРИШЛИТЕ МНЕ НА ПОЧТУ. Отсутствие перевода у меня на почте приведёт к получению МИНУС 32 баллов. Адекватный перевод стоит 16 баллв с плюсом….

• Segmented memory model — Memory appears to a program as a group of independent address spaces called segments. Code, data, and stacks are typically contained in separate segments. To address a byte in a segment, a program issues a logical address. This consists of a segment selector and an offset (logical addresses are often referred to as far pointers). The segment selector identifies the segment to be accessed and the offset identifies a byte in the address space of the segment.. To access a memory location, the processor thus translates each logical address into a linear address. This translation is transparent to the application program. The primary reason for using segmented memory is to increase the reliability of programs and systems. For example, placing a program’s stack in a separate segment prevents the stack from growing into the code or data space and overwriting instructions or data, respectively – и СРАЗУ ЖЕ ПРИШЛИТЕ МНЕ НА ПОЧТУ ПЕРЕВОД!!!!!.

Толкование 402. Главное, что необходимо «понять» из всего вышесказанного, это то, что для того, чтобы прикладная программа могла быть исполнена, ОПЕРАЦИОННАЯ СИСТЕМА (НЕ ОБОЛОЧКА Windows) должна как-то понять, что прикладная программа хочет попасть в оперативную память, точнее в Conventional память, и хочет, чтобы ЕЙ в какой-то момент было передано управление. И, надеюсь, понятно, что ПРОГРАММА – это не то, что написано пером или набрано с помощью некоторого программного средства типа Microsoft Word и хранится в виде текстового файла на внешнем устройстве. То, что написано пером или «набрано с помощью некоторого программного средства типа Microsoft Word и хранится в виде текстового файла на внешнем устройстве» следует назвать «ИСХОДНЫЙ ТЕКСТ НА ЯЗЫКЕ ЗАПИСИ АЛГОРИТМОВ». И это даже (или ТЕМ БОЛЕЕ?) - НЕ ИСПОЛНИМЫЙ МОДУЛЬ, который представляет СОБОЙ ОФОРМЛЕННЫЙ ПО ПРАВИЛАМ ЛИБО ОПЕРАЦИОННОЙ СИСТЕМЫ (ЛИБО ОПЕРАЦИОННОЙ ОБОЛОЧКИ) ФАЙЛ, состоящий ИЗ КОМАНД ЛИБО ОПЕРАЦИОННОЙ СИТСЕМЫ (bat-файлы) ЛИБО из готовых к исполнению машинных команд, которые ПРЕДВАРЯЮТСЯ (либо не предваряются) так называемым ЗАГОЛОВКОМ – например, ИСПОЛНИМЫЕ ФАЙЛЫ, называемые exe-файлы, которые как раз состоят из заголовка, начинающегося с двух латинских букв MZ, имеющего строго определенные РАЗМЕР И СТРУКТУРУ, и НЕПОСРЕДСТВЕННО СОВОКУПНОСТИ МАШИННЫХ КОМАНД, полученных в результате компиляции (трансляции) исходного текста на языке записи алгоритмов и «сборки» (линковки). Если машинные команды в виде файла на внешнем устройстве НЕ ПРЕДВАРЯЮТСЯ ЗАГОЛОВКАМИ, ЭТОТ ФАЙЛ называется ИСПОЛНИМЫМ ФАЙЛОМ ТИПА COM. Для получения подобных файлов необходимо транслировать исходный текст на языке записи алгоритмов низкого уровня (типа ассемблер), осуществить «линчевание» полученного в результате трансляции объектного модуля. Если же речь идёт о языках записи алгоритмов высокого уровня (типа Си), то необходимо компилировать исходный текст на языке записи алгоритмов, а затем «собрать» что-то там с получением исполнимого модуля типа exe.

СТРУКТУРА ИСПОЛНИМЫХ МОДУЛЕЙ.

Упражнение 402. Дочитав до этого места, ответить на вопрос: ЧТО ТАКОЕ MZ.

Упражнение 403. По прочтении текста до конца Лекции УКАЗАТЬ, СКОЛЬКО РАЗ В ЭТОМ ТЕКСТЕ ИСПОЛЬЗУЕТСЯ НЕНОРМАТИВНАЯ ЛЕКСИКА!!!!

Существует множество форматов исполняемых файлов для разных операционных систем, например: COM, EXE (Windows, DOS), COFF, ELF (Unix). Обычно данные, хранящиеся в исполняемом файле, состоят из нескольких частей:

* заголовки (предполагаемые исполнители инструкций, настройки исполнителя инструкций и окружения, формат инструкций). Исполнители инструкций — это аппаратно-программный комплекс, способный исполнить инструкцию (процессоры, микроконтроллеры, интерпретаторы, виртуальные машины);
* инструкции (собственно код программы). Представлены либо машинными инструкциями, либо исходным кодом (для интерпретируемого языка), либо байт-кодом для виртуальной машины;
* дополнительные (данные, используемые для отладки, список библиотек и используемых функций, иконки для ярлыков, тексты, изображения, звуки, видео и др.)

В качестве защищаемых файлов были выбраны наиболее распространенные в наши дни исполняемые PE-файлы для 32-разрядных версий операционной системы Microsoft Windows.

**2.1.1 Структура формата PE**

Формат PE (Portable Executable) — формат исполняемых файлов, динамически подключаемых библиотек dll, драйверов sys и некоторых других, используемый в 32- и 64-разрядных версиях операционной системы Windows. Формат обладает большим количеством полей, поэтому в работе будут описаны только те, которые необходимы для реализации поставленной задачи. В качестве основных источников информации были использованы [10], [11], [12].

В самом общем приближении исполняемый файл в формате PE состоит из набора заголовков и секций. Для того чтобы описать, какие именно данные находятся в секции, используются директории. В одной секции могут находиться данные разных типов. Общая структура PE формата изображена на рисунке 1. Отмечены только самые важные поля.

Рисунок 1. Общая структура PE формата

**IMAGE\_NT\_HEADER**

Signature

**IMAGE\_FILE\_HEADER**

NumberOfSections

SizeOfOptionalHeader

Characteristics

**IMAGE\_OPTIONAL\_HEADER**

Magic

ImageBase

AddressOfEntryPoint

SectionAligment

FileAligment

SizeOfHeaders

**IMAGE\_DATA\_DIRECTORY**

VirtualAddress

Size

**IMAGE\_DOS\_HEADER**

e\_magic

e\_lfanew

**IMAGE\_SECTION\_HEADER**

Name

VirtualSize

VirtualAddress

SizeOfRawData

PointerToRawData

Characteristics

**Секция**

**Секция**

*Рисунок 1. Общая структура PE формата*

**IMAGE\_DOS\_HEADER**

e\_magic: WORD — сигнатура, находящаяся по смещению 0 и равная "MZ" (IMAGE\_DOS\_SIGNATURE);

e\_lfanew: DWORD — смещение от начала файла до заголовка PE.

Поскольку и приложения DOS, и приложения Windows имеют расширение .exe, все исполняемые файлы Windows используют схему двойной загрузки. Она состоит в том, что файл начинается с заголовка DOS, за которым следует заглушка (stub), т. е. небольшой exe-файл формата DOS. При попытке загрузить файл из DOS'а исполняется заглушка, а при загрузке файла из Windows загрузчик анализирует заголовок DOS и извлекает из него смещение до настоящего заголовка исполняемого файла. Стандартное поведение, если запустить программу на dos, то она выведет сообщения вида "This program cannot be run in DOS mode".

**IMAGE\_NT\_HEADER**

Signature: DWORD — сигнатура. Должна быть равна "PE\0\0" (IMAGE\_NT\_SIGNATURE).

PE-заголовок содержит в себе заголовок файла и необязательный заголовок. Несмотря на свое название, IMAGE\_OPTIONAL\_HEADER присутствует в PE-файле всегда (необязательным он является с точки зрения общего формата COFF, поскольку не используется в объектных файлах).

**IMAGE\_FILE\_HEADER**

NumberOfSections: DWORD — количество секций в файле. Секции следуют сразу после заголовка (PE-Header);

SizeOfOptionHeader: WORD — размер опционального заголовка (что следует сразу за текущим);

Characteristics: WORD — характеристики.

По полю Characteristics можно определить, имеем мы дело с "самостоятельным" файлом (установлен бит IMAGE\_FILE\_EXECUTABLE\_IMAGE) или с динамически подключаемой библиотекой (установлен бит IMAGE\_FILE\_DLL).

**IMAGE\_OPTIONAL\_HEADER**

Magic: WORD — сигнатура. Для PE32 равно 0x10b;

ImageBase: DWORD — предпочтительный базовый адрес загрузки в оперативную память. Если файл не может быть загружен по данному адресу, то загрузка происходит по произвольному свободному адресу, после чего применяется информация из таблицы перемещаемых элементов. Это необходимо для того, чтобы величины в коде программы и в некоторых заголовках, которые являются абсолютными адресами, продолжали указывать на верные места в памяти. Абсолютный адрес в виртуальной памяти называется виртуальным адресом (Virtual Address, VA). Относительный виртуальный адрес (Relative Virtual Address, RVA) вычисляется относительно ImageBase:

Начиная с данного поля, все поля, которые указывают на место в памяти, являются на самом деле RVA (за исключением адресов в директории TLS);

AddressOfEntryPoint: DWORD — RVA адрес точки входа;

SectionAligment: DWORD — размер выравнивания (байты) секции при выгрузке в виртуальную память. По умолчанию равно размеру страницы виртуальной памяти для данного процессора;

FileAligment: DWORD — размер выравнивания (байты) секции внутри файла;

SizeOfImage: DWORD — размер файла (в байтах) в памяти, включая все заголовки. Должен быть кратен SectionAligment;

SizeOfHeaders: DWORD — размер всех заголовков (DOS, DOS-Stub, PE, Section), выровненный на FileAligment. Задает смещение от начала файла до данных первой секции;

Также заголовок содержит массив описателей каталогов (директорий) данных. На данный момент количество описателей всегда равно 16 (IMAGE\_NUMBEROF\_DIRECTORY\_ENTRIES). Каждый из каталогов данных содержит определенную служебную информацию. Вид этой информации определяется номером каталога в массиве описателей. Например, константа IMAGE\_DIRECTORY\_ENTRY\_IMPORT равна 1. Следовательно, описатель каталога экспорта можно получить как OptionalHeader.DataDirectory[IMAGE\_DIRECTORY\_ ENTRY\_IMPORT].

**IMAGE\_DATA\_DIRECTORY**

VirtualAddress: DWORD — RVA каталога;

Size: DWORD — размер каталога.

Если каталог отсутствует, то оба поля равны 0.

**IMAGE\_SECTION\_HEADER**

Name: BYTE[IMAGE\_SIZEOF\_SHORT\_NAME] — название секции. На данный момент имеет длину в 8 символов.

VirtualSize: DWORD — размер секции в виртуальной памяти. Если больше, чем SizeOfRawData, то дополняется нулями.

VirtualAddress: DWORD — RVA адрес секции.

SizeOfRawData: DWORD — размер секции в файле. Должен быть кратен FileAligment.

PointerToRawData: DWORD — смещение до начала секции в файле. Должен быть кратен FileAligment. Если секция содержит только неинициализированные данные, то это поле равно нулю.

Characteristics: DWORD — атрибуты доступа к секции и правила для её загрузки в память. Например, атрибут для определения содержимого секции (инициализированные данные, не инициализированные данные, код). Или атрибуты доступа — чтение, запись, исполнение.

**Секция**

Сами секции располагаются в файле после всех заголовков секций. Каждая секция выровнена на границу FileAlignment.

Рассмотрим каталоги, которые должен обрабатывать загрузчик для запуска простых исполняемых файлов.

**Каталог настройки адресов (IMAGE\_DIRECTORY\_ENTRY\_BASERELOC)**

Для того чтобы исполняемый файл можно было загрузить с произвольного адреса, он должен содержать таблицу настроек адресов (таблицу перемещений). Таблица разделена на блоки. Каждый блок содержит настройки для 4 Кб данных и начинается с заголовка IMAGE\_BASE\_RELOCATION. Заголовок содержит начальный RVA и размер блока в байтах с учетом заголовка. Таблица настройки заканчивается блоком, у которого заголовок IMAGE\_BASE\_RELOCATION заполнен нулями. Каждый описатель настройки в четырех старших битах содержит тип настройки, а в 12 младших битах – смещение от начального RVA до настраиваемых данных. На сегодняшний день актуальны два типа настройки из 11:

* IMAGE\_REL\_BASED\_HIGHLOW — к данным, расположенным по настраиваемому адресу, прибавить разность между фактическим адресом, и адресом загрузки из опционального заголовка (ImageBase).
* IMAGE\_REL\_BASED\_ABSOLUTE — нет настройки. Используется для выравнивания блоков по границе двойного слова.

**Каталог экспорта (IMAGE\_DIRECTORY\_ENTRY\_EXPORT)**

Эта таблица всегда присутствует в DLL-файлах, поскольку основным назначением динамических библиотек является экспорт символов, доступных другим DLL- и EXE-файлам. Основная цель таблицы экспорта — связать имена и/или номера экспортируемых функций с их RVA, т. е. с положением в виртуальной памяти процесса.

Таблица экспорта называется IMAGE\_EXPORT\_DIRECTORY. Важными полями являются:

Name: DWORD — RVA ASCIIZ-строки, содержащей имя библиотеки;

Base: DWORD — начальный номер экспортируемых символов.

NumberOfFunctions: DWORD — количество элементов в таблице адресов.

NumberOfNames: DWORD — количество элементов в таблице имен и таблице номеров.

AddressOfFunctions: DWORD — RVA таблицы адресов.

AddressOfNames: DWORD — RVA таблицы имен. Имена отсортированы в алфавитном порядке для ускорения поиска.

AddressOfNameOrdinals: DWORD — RVA таблицы номеров (ординалов).

При поиске экспортируемого символа по его имени сначала производится бинарный поиск этого имени в таблице имен. Если имя найдено и его номер в таблице имен равен N, то извлекается N-й элемент из таблицы номеров. Если этот номер равен K, то элемент таблицы адресов с номером (K - Base) содержит RVA данного символа. Данная схема позволяет приписать одному экспортируемому символу несколько разных имен (рисунок 2).

Рисунок 2. Таблица экспорта

**IMAGE\_EXPORT\_DIRECTORY**

Name = "mydll.dll"

Base = 1

NumberOfNames = 3

NumberOfFunctions=2

AddressOfNames

AddressOfFunctions

AddressOfOrdinals

"func1"

"func2"

"func3"

1

1

0

0x0002B40

0x00CE42

0x00AB02

0x007E4F

0x0091FC

Из этого правила существует исключение: если RVA в таблице адресов находится в пределах таблицы экспорта, то это RVA перенаправления (Forwarder RVA); в противном случае это обычный RVA символа (Export RVA). RVA перенаправления заменяет экспорт данного символа на экспорт некоторого другого символа из другой библиотеки. RVA перенаправления указывает на ASCIIZ-строку. Строка имеет либо вид "MyDll.exportproc" (экспорт по имени), либо вид "MyDll#12" (экспорт по ординалу).

**Каталог импорта (IMAGE\_DIRECTORY\_ENTRY\_IMPORT, IMAGE\_DIRECTORY\_ ENTRY\_BOUND\_IMPORT, IMAGE\_DIRECTORY\_ENTRY\_DELAY\_IMPORT)**

Таблица импорта присутствует практически во всех PE-файлах. Данная таблица используется для разрешения ссылок из файла на динамически подключаемые библиотеки. Существует три режима импорта: стандартный, связывающий (bound import, старая и новая схема) и отложенный (delay import).

Директория импорта IMAGE\_DIRECTORY\_ENTRY\_IMPORT состоит из массива описателей IMAGE\_IMPORT\_DESCRIPTOR. Каждый описатель для одной библиотеки, но для одной библиотеки может быть несколько описателей. Конец массива определяется по описателю, у которого все поля нулевые.

Схема каталога IMAGE\_DIRECTORY\_ENTRY\_IMPORT изображена на рисунке 3.

Рисунок 3. Структура каталога IMAGE\_DIRECTORY\_ENTRY\_IMPORT

**IMAGE\_IMPORT\_DESCRIPTOR**

OriginalFirstThunk (INT)

TimeDateStamp

ForwarderChain

Name

FirstThunk (IAT)

**IMAGE\_THUNK\_DATA**

AddressOfData

**IMAGE\_THUNK\_DATA**

AddressOfData

**IMAGE\_IMPORT\_BY\_NAME**

Hint

Name

OriginalFirstThunk: DWORD — RVA таблицы имен импорта (Import Name Table);

TimeDateStamp: DWORD — 0 в случае стандартного. Дата и время из заголовка динамической библиотеки в случае старой схемы связывающего импорта. -1 в случае новой схемы связывающего импорта. Связывающий импорт будет описан далее;

ForwarderChain: DWORD — индекс первого перенаправленного символа. Используется для старой схемы связного импорта;

Name: DWORD — RVA имени библиотеки;

FirstThunk: DWORD — RVA таблицы адресов импорта (Import Address Table).

Описатель указывает на таблицы IAT и INT (обе являются массивом структур типа IMAGE\_THUNK\_DATA, оканчивающийся структурой, заполненной нулями). Таблица IAT может храниться в отдельном каталоге IMAGE\_DIRECTORY\_ENTRY\_IAT.

Каждая ячейка таблица INT используется для описания символа, адрес которого нужно поместить в соответствующую ячейку таблицы IAT. Во время выполнения программы адрес внешних символов будет использоваться именно из таблицы IAT. Процесс поиска в таблице экспорта загруженной динамической библиотеки необходимого символа и запись его виртуального адреса в таблицу IAT называется динамическим связыванием.

Загрузчик просматривает таблицу INT. Если старший бит поля AddressOfData установлен, то оставшиеся биты задают ординал символа, иначе это поле указывает на описатель символа IMAGE\_IMPORT\_BY\_NAME. Описатель символа состоит из индекса импортируемого символа в таблицы имен библиотеки и указателя на строку с именем. Не все сборщики заносят правильное значение в поле Hint, поэтому имя функции необходимо проверять.

Самый быстрый способ поиска адреса символа в таблице экспорта библиотеки является поиск по ординалу, самый медленный — по имени символа.

Для ускорения связывания процесса импортирования используется связывающий импорт. Этот способ также называют статическое связывание. Суть статического связывания в том, что в таблицу IAT файла прописываются адреса импортируемых символов в предположении, что все динамически загружаемые библиотеки загружаются по прописанным в них базовым адресам. В двух случаях статическое связывание невозможно и приходится прибегать к динамическому связыванию: если библиотека загружена не по ее базовому адресу ImageBase или версия библиотеки отличается от той, с которой было выполнено статическое связывание (проверяется по полю TimeDateStamp в описатели IMAGE\_IMPORT\_DESCRIPTOR и заголовке библиотеки).

Применить старую схему динамического связывания также невозможно в случае перенаправления символа, так как невозможно отследить, соответствуют ли версия библиотеки и дата связывания. Из соображений надежности такие символы всегда связываются заново в процессе загрузки. Поле ForwarderChain содержит индекс первого перенаправленного символа в IAT. По этому индексу хранится индекс следующего перенаправленного символа и т. д. В индексе последнего перенаправленного символа хранится -1.

Новая схема связывающего импорта заносит в поля TimeDateStamp и ForwarderChain значения -1. Информация о такой библиотеке содержится в каталоге IMAGE\_DIRECTORY\_ENTRY\_BOUND\_IMPORT. Каталог имеет структуру, аналогичную каталогу настройки адресов: для каждой библиотеки существует описатель IMAGE\_BOUND\_IMPORT\_DESCRIPTOR, за которым следует таблица описателей IMAGE\_BOUND\_FORWARDER\_REF для каждого перенаправленного импортируемого символа.

Описатель IMAGE\_BOUND\_IMPORT\_DESCRIPTOR содержит имя библиотеки, дату и время ее создания, и количество следом идущих структур IMAGE\_BOUND\_FORWARDER\_REF.

Структура IMAGE\_BOUND\_FORWARDER\_REF состоит из указателя на имя функции и временную отметку создания динамической библиотеки, на которую происходит перенаправление. Благодаря такому устройству, загрузчик может избежать некоторых случаев использования динамического связывания, заменив его статическим связывание, где это возможно.

Отложенная загрузка импорта означает, что библиотека присоединена к исполняемому файлу, но загружается в память не сразу, как при обычном импорте, а только при первом обращении программы к символу, импортируемому из этой библиотеки. Отложенная загрузка импорта не является свойством загрузчика ОС. Она реализуется добавлением к программе дополнительного кода и данных в процессе сборки. Поэтому каталог IMAGE\_DIRECTORY\_ENTRY\_DELAY\_IMPORT в данной работе не рассматривается.