

# Низкоуровневое программирование

Лекция 5

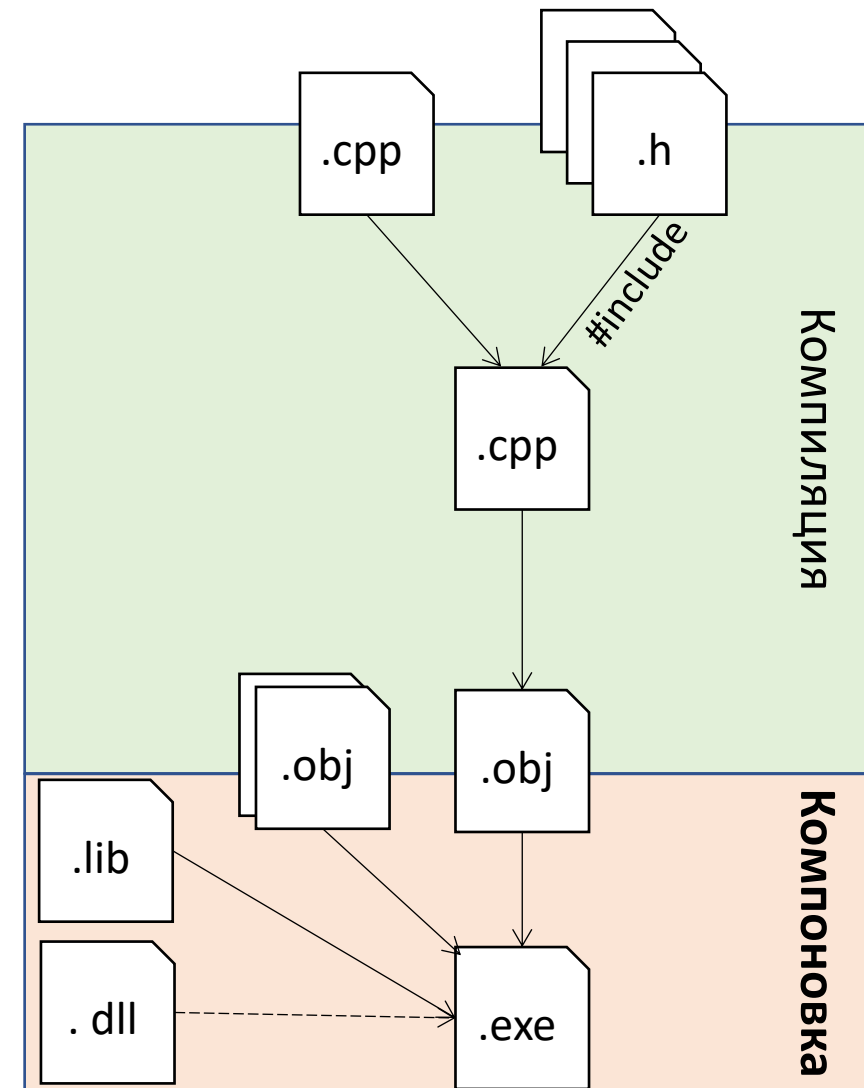
Компиляция

Статическая и динамическая компоновка

# Сборка программ

Процесс создания исполняемой программы из исходного кода называется **сборкой**.

В ходе сборки программы выполняются 2 основных этапа – компиляция и компоновка (linking).



# Компиляция

**Трансляция** - преобразование программы, представленной на одном языке программирования, в программу на другом языке и в определенном смысле равносильную первой [ГОСТ 15971-90].

**Компиляция** – трансляция программы из ее исходного кода на языке программирования высокого уровня в бинарный код машинного языка [ГОСТ Р 71206-2024].

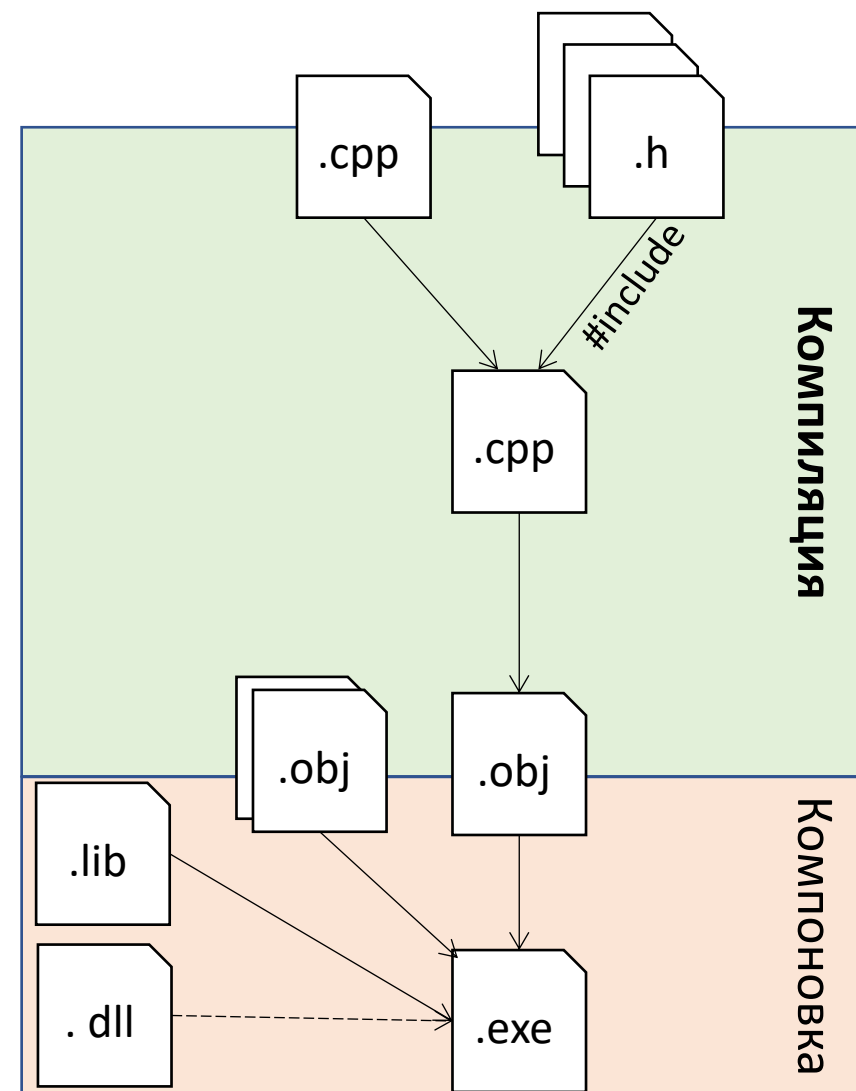
**Компилятор** – программа, выполняющая компиляцию.

Входом компилятора является файл с исходным кодом.

Результатом обработки входного файла препроцессором (обработки макросов и директив #include) является **единица трансляции**, содержащая итоговый исходный код, непосредственно подвергающаяся компиляции.

Выходом компилятора является **объектный файл** – файл с машинным кодом и метаданными.

В метаданных объектного файла находится информация о символах (функциях и переменных), которые используются в объектном файле.



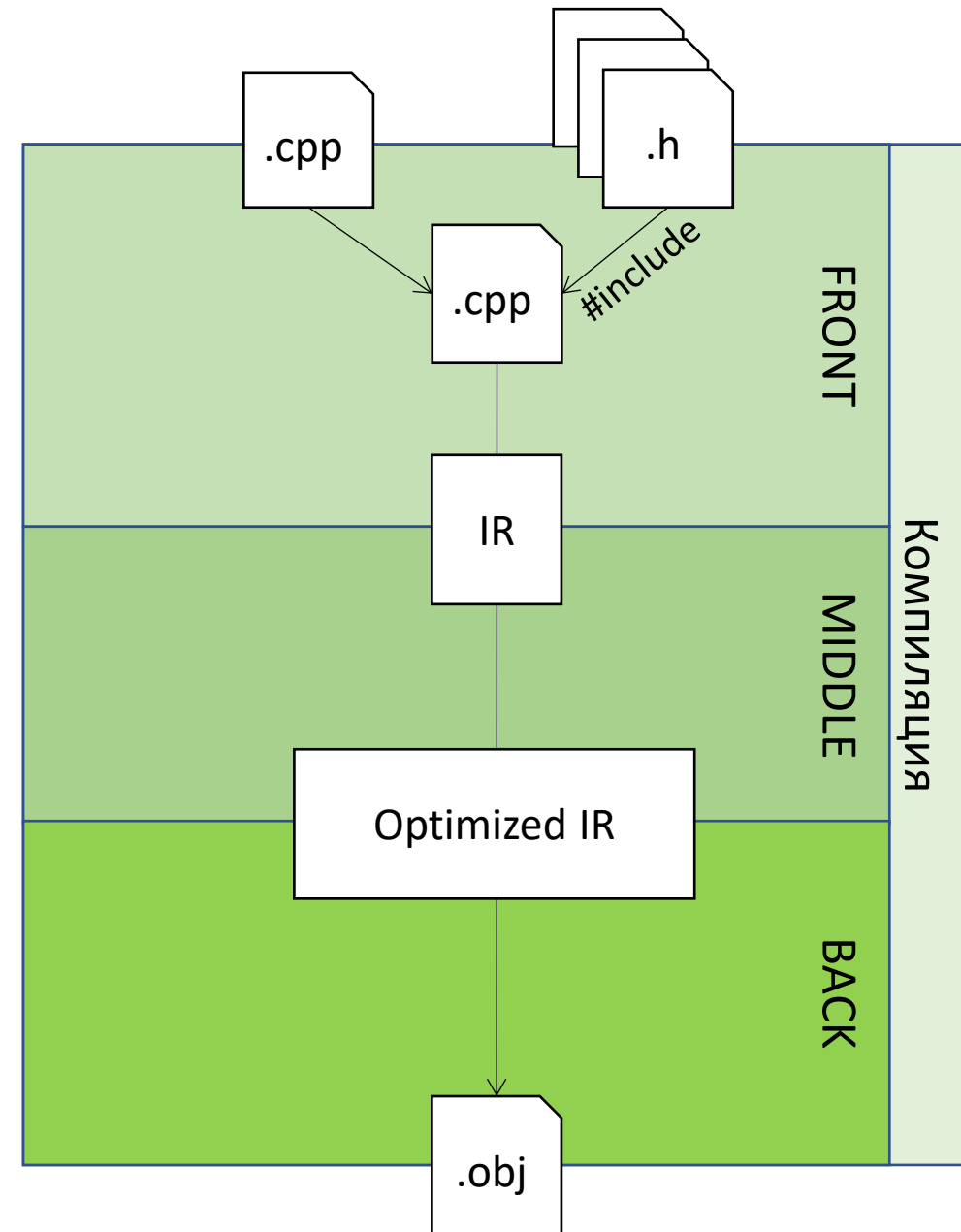
# Трехфазная компиляция

Большинство современных компиляторов состоят из 3 частей: фронтенда, промежуточной части и бэкенда.

**Фронтенд** отвечает за первичное преобразование исходного кода. На этом этапе происходит препроцессинг, построение дерева синтаксического разбора кода и генерация *промежуточного представления кода* (Intermediate Representation, IR).

**Промежуточная часть** отвечает за общую оптимизацию IR-кода (например, замена  $a*2$  на  $a<<1$  или удаление недоступных участков кода).

**Бэкенд** отвечает за оптимизации, специфичные для целевой архитектуры и итоговую трансляцию IR-кода в машинный код.

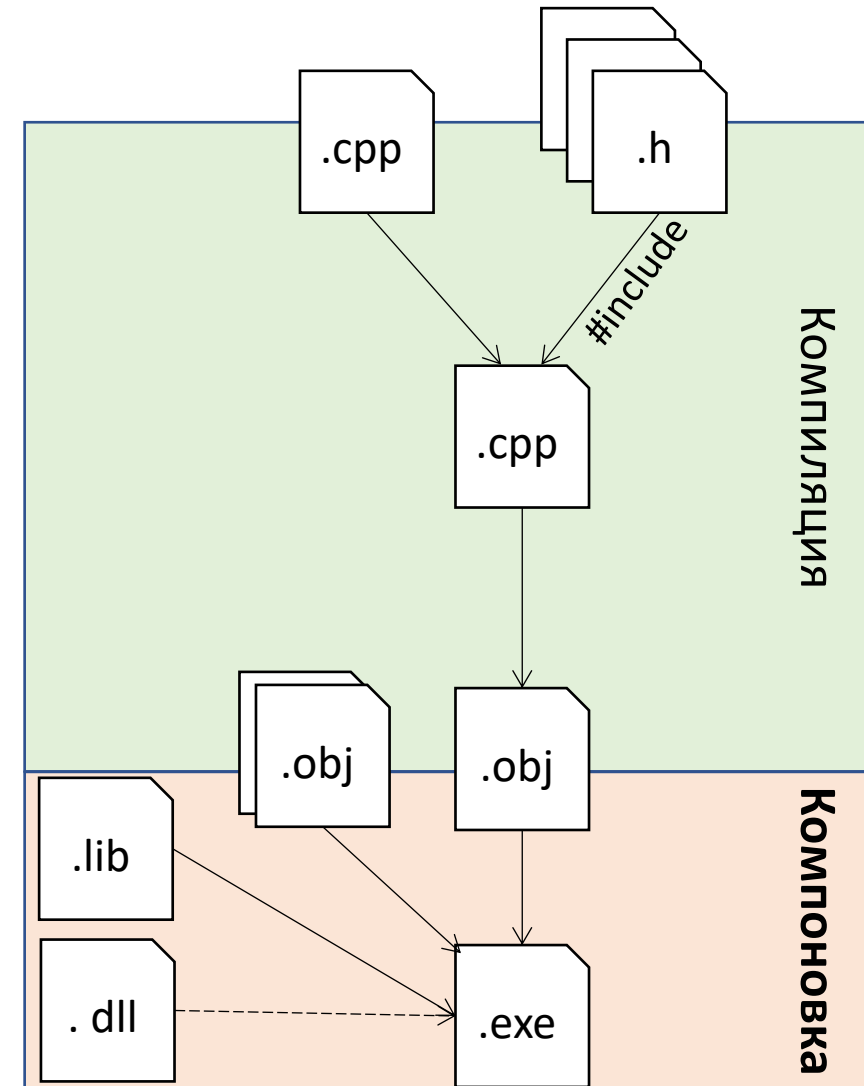


# Компоновка

**Компоновка** – процесс формирования исполняемого файла из набора объектных файлов.

В ходе компоновки производится объединение объектных файлов и определение расположения всех используемых символов.

Помимо объектных файлов, компоновщик может принимать на вход статические и динамические библиотеки.



# Таблица символов

**Символ** – уникальное в пределах единицы трансляции имя, с которым ассоциировано некоторое значение (обычно – адрес или смещение относительно начала секции или файла). Символы, объявленные в модуле, перечислены в **таблице символов**.

Символы обычно имеют тип (функция, данные или др.), область видимости (глобальный или локальный), секцию и значение (обычно – адрес или смещение от начала секции).

Символами являются имена функций и глобальных переменных. Метки языка ассемблера тоже становятся символами. В C/C++ символы всех функций и процедур, которые не объявлены как `static`, являются глобальными (т.е., доступными извне). В NASM для определения глобального символа используется ключевое слово `global`.

```
int x = 5;
static int y = 9;
float f(int a) { /*...*/ }
int g(int a); //функция без определения
extern int h(int a); //внешняя функция
extern int z;      //внешняя переменная
```

Таблица символов				
Имя	Тип	Видимость	Секция	Значение
x	OBJECT	GLOBAL	.data	0x8
y	OBJECT	LOCAL	.data	0xF
z	OBJECT	GLOBAL	???	???
f	FUNC	GLOBAL	.text	0xF0
g	FUNC	GLOBAL	???	???
h	FUNC	GLOBAL	???	???

# Символы и язык С

Поскольку в языке С нет ни пространств имен, ни перегрузок функций, любая функция и любая локальная переменная имеют уникальное в пределах всей программы имя.

Как следствие, имя символа совпадает с именем.

Особым случаем является компилятор MSVC, который при сборке под x86-32 изменяет имя символа в соответствии с используемым соглашением(cdecl или stdcall).

При использовании cdecl к имени функции добавляется слева `_` (для функции справа результатом будет `_f`).

При использовании stdcall к имени функции добавляется слева `_`, а справа - `@` и общий размер аргументов (для функции справа результатом будет `_f@12`)

```
int f(int a, long long b) {/**/}
```



```
f:
    push rbp
    mov rbp, rsp
    ;...
    leave
    ret
```

# Символы и язык C++

В языке C++ из-за наличия перегрузок, классов и пространств имен возникает проблема дубликатов символов.

Для решение данной проблемы используется механизм искажения имен (mangling).

Правила искажения имен могут отличаться от компилятора к компилятору.

```
int f(int a, long long b) —————→ _Z1fix
int f(int a, float b) —————→ _Z1fif
namespace V{
    int f(int a, float b); —————→ _ZN1V1fEif
}
class Cls{
    auto f(int a); —————→ _ZN3Cls6fEi
};
```

```
extern "C" int f(int a, long long b); —————→ f
```



# Внешние символы

**Внешние символы** - символы, которые объявлены, но не определены в текущей единице трансляции. Внешние символы не имеют ассоциированного значения.

Значения внешних символов определяются во время компоновки.

В NASM и в C/C++ для определения внешнего символа используется ключевое слово `extern`.

```
int x = 5;
static int y = 9;
float f(int a) { /*...*/ }
int g(int a); //функция без определения
extern int h(int a); //внешняя функция
extern int z;      //внешняя переменная
```

Таблица символов				
Имя	Тип	Видимость	Секция	Значение
x	OBJECT	GLOBAL	.data	0x8
y	OBJECT	LOCAL	.data	0xF
z	<b>OBJECT</b>	<b>GLOBAL</b>	???	???
f	FUNC	GLOBAL	.text	0xF0
g	<b>FUNC</b>	<b>GLOBAL</b>	???	???
h	<b>FUNC</b>	<b>GLOBAL</b>	???	???

# Компоновка

Во время компоновки все объектные файлы объединяются в один, секции с одинаковыми именами объединяются, добавляются служебные секции.

При этом строится общая таблица символов, неизвестные значения в которой заполняются.

Поскольку при объединении секций могут меняться адреса переменных и функций, производится их корректировка (релокация). При этом корректируется и значение символа, и значение в машинном коде.

Символ должен быть определен либо в одном из объектных файлов, либо дополнительных файлах – зависимостях сборки.

*Если расположение хотя бы одного символа не будет определено или будет найдено более 1 значения, компоновка завершается с ошибкой.*

Таблица символов a.obj	
Символ	Значение
a	0x0F
x	???

Таблица символов b.obj	
Символ	Значение
x	0x0A
y	0x1A
a	???

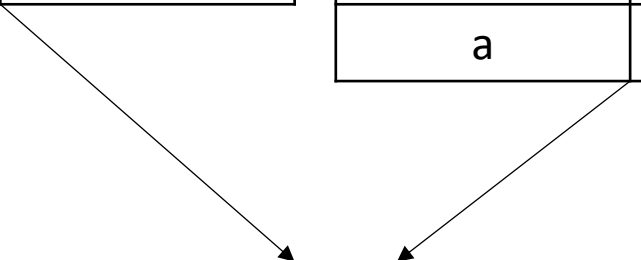


Таблица символов	
Символ	Значение
x	0x0A
y	0x1A
a	0x20

# Статические и динамические библиотеки

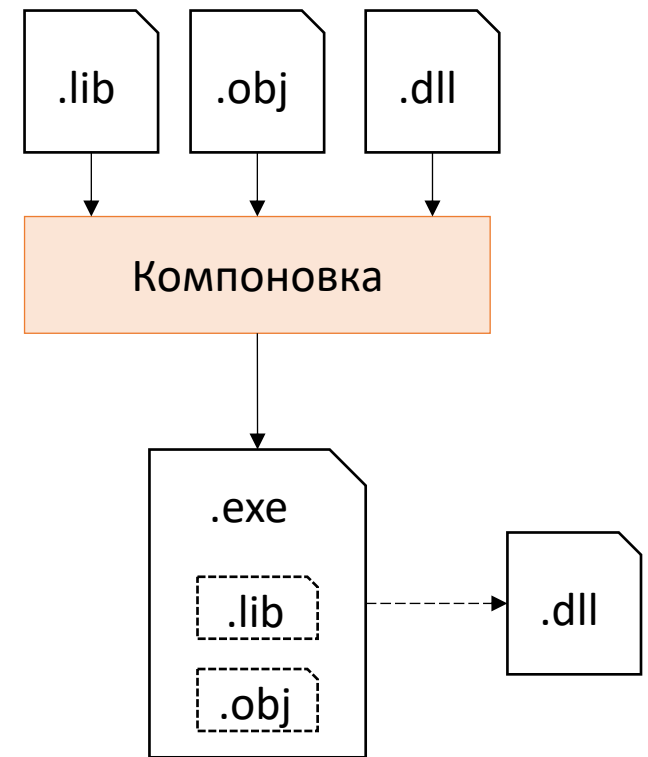
Помимо объектных файлов самой программы, при компоновке можно указать дополнительные зависимости – статические и динамические библиотеки.

**Статическая библиотека (.lib, .a)** – архив, содержащий объектные файлы и информацию о них.

Статическая библиотека становится частью исполняемого файла.

**Динамическая библиотека/разделяемая библиотека (.dll, .so)** – исполняемый файл, загружаемый по требованию в ходе работы программы.

Динамическая библиотека не становится частью исполняемого файла.



# Таблицы импорта

Если символ определен в динамической библиотеке, которая была указана как зависимость при компоновке, то данный символ попадает в **таблицу импорта**.

Библиотеки, упомянутые в таблице импорта, загружаются в ходе процесса **динамической компоновки** после или во время запуска программы.

*В ELF-файлах используются отдельная таблица для импортируемых символов, и отдельная таблица для библиотек, содержащих зависимости => нет жесткой связи между библиотекой и символом.*

*В PE-файлах используется 1 таблица, в которой указываются и символ, и библиотека.*

Таблица импорта a.exe (секция .idata)	
Символ	Файл
malloc	libc.dll
printf	libc.dll

Таблица зависимостей a.elf (секция .dynamic)
libc.so

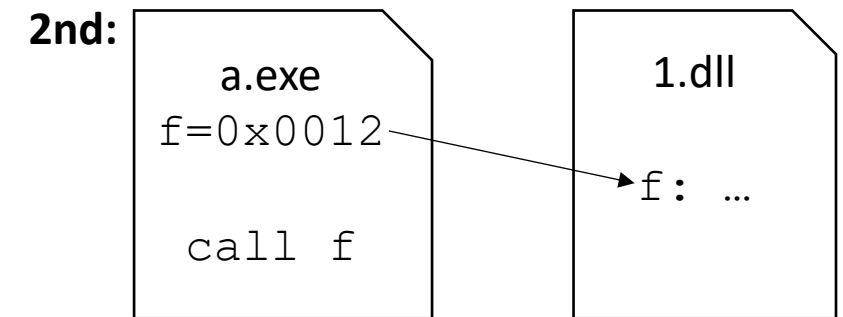
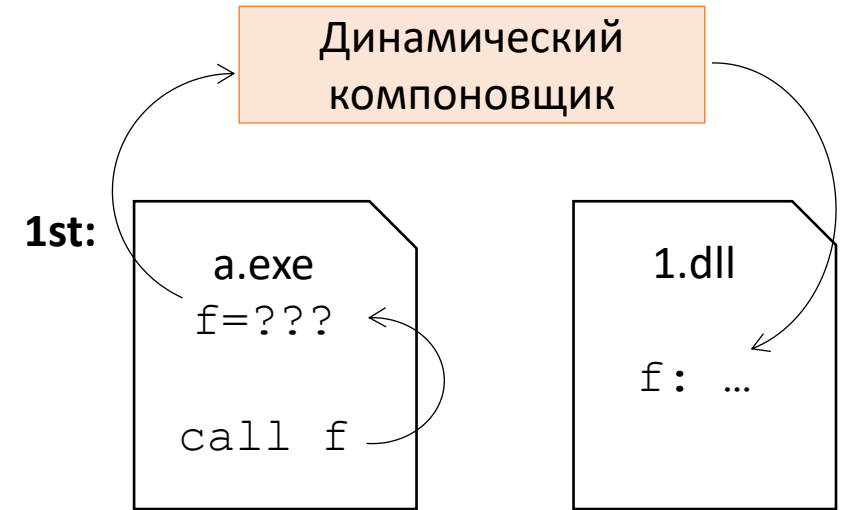
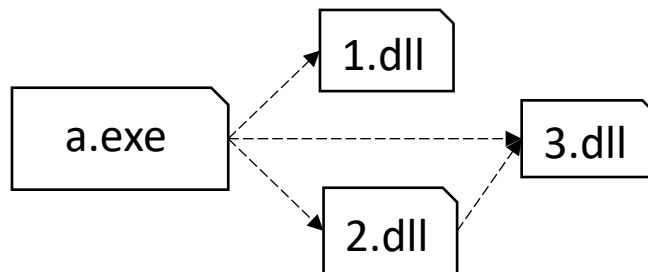
Таблица импорта a.elf (секция .dynsym)
malloc
printf

# Динамическая компоновка

В ходе динамической компоновки :

1. Загружается динамическая библиотека, содержащая символ
2. Определяется адрес символа, адрес записывается в специальную таблицу.
3. При последующих обращениях к символу используется напрямую адрес из таблицы.

Если у динамической библиотеки есть своя динамическая таблица импорта, зависимости из нее загружаются аналогично.



# Исполняемые файлы

**Исполняемый файл** – файл с машинным кодом, который может быть загружен ОС и выполнен.

Всякий исполняемый файл имеет **точку входа** (entry point) – адрес, с которого начинается исполнение файла. Формат исполняемого файла определяется ABI ОС.

К исполняемым файлами *не относятся* объектные файлы и статические библиотеки.

Динамические библиотеки формально являются исполняемыми файлами – они имеют тот же формат файла и могут быть загружены ОС в процессе динамической компоновки, но не имеют точки входа. Динамические библиотеки могут иметь код инициализации/деинициализации, выполняемый при загрузке/выгрузке библиотеки.

Доминирующими форматами исполняемых файлов современных систем являются:

**Windows:** PE (Portable Executable)

**Linux:** ELF (Executable and Linkable Format)

**MacOS:** Mach-O (Mach Object)

# Структура исполняемого файла

Исполняемый файл любого из современных форматов содержит, как минимум, следующие части:

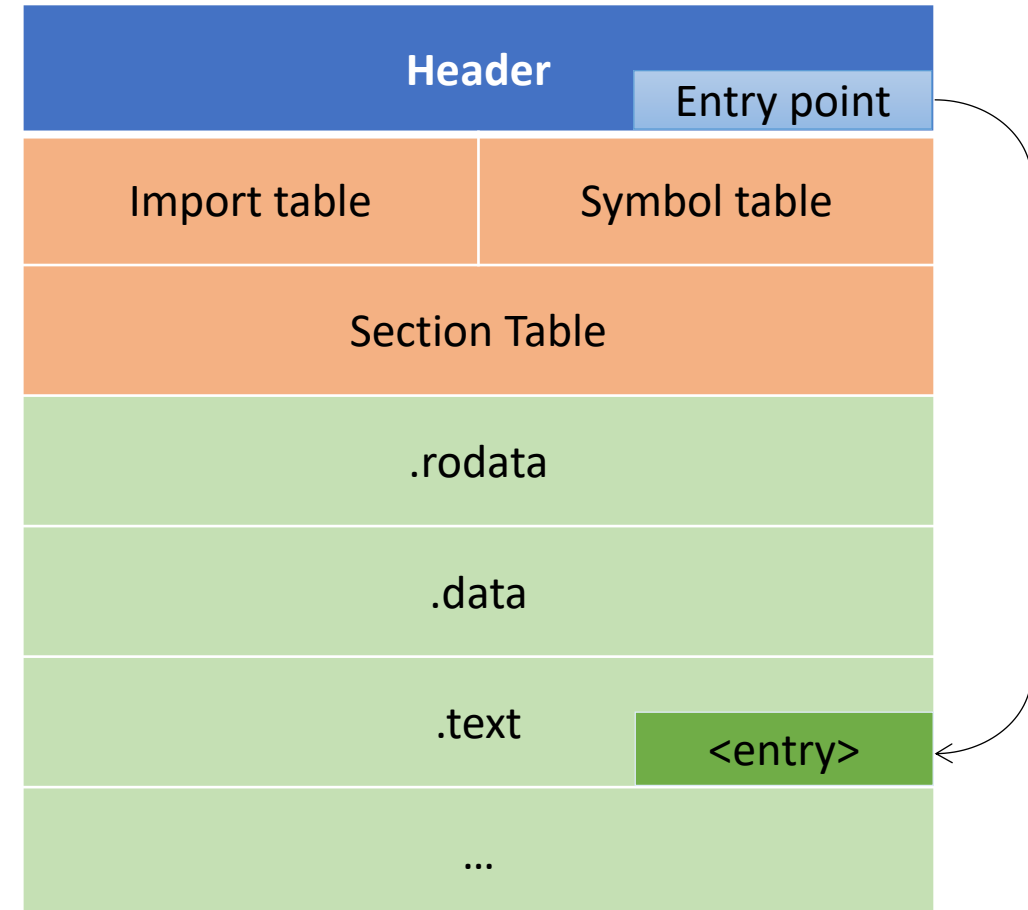
- Заголовок;
- Таблицы символов;
- Таблица секций;
- Таблица релокаций и прочие служебные таблицы;
- Сегменты программы (.text, .data, .rodata, ...).

Header	
Import table	Symbol table
Section Table	
.rodata	
.data	
.text	
...	

# Заголовок исполняемого файла

В заголовке файла перечислена основная информация – тип файла, целевая архитектура, адрес точки входа и пр.

При загрузке файла ОС читает заголовок, загружает необходимые зависимости и передает управление на адрес точки входа, после чего начнется выполнение программы.





# Таблицы символов и секций

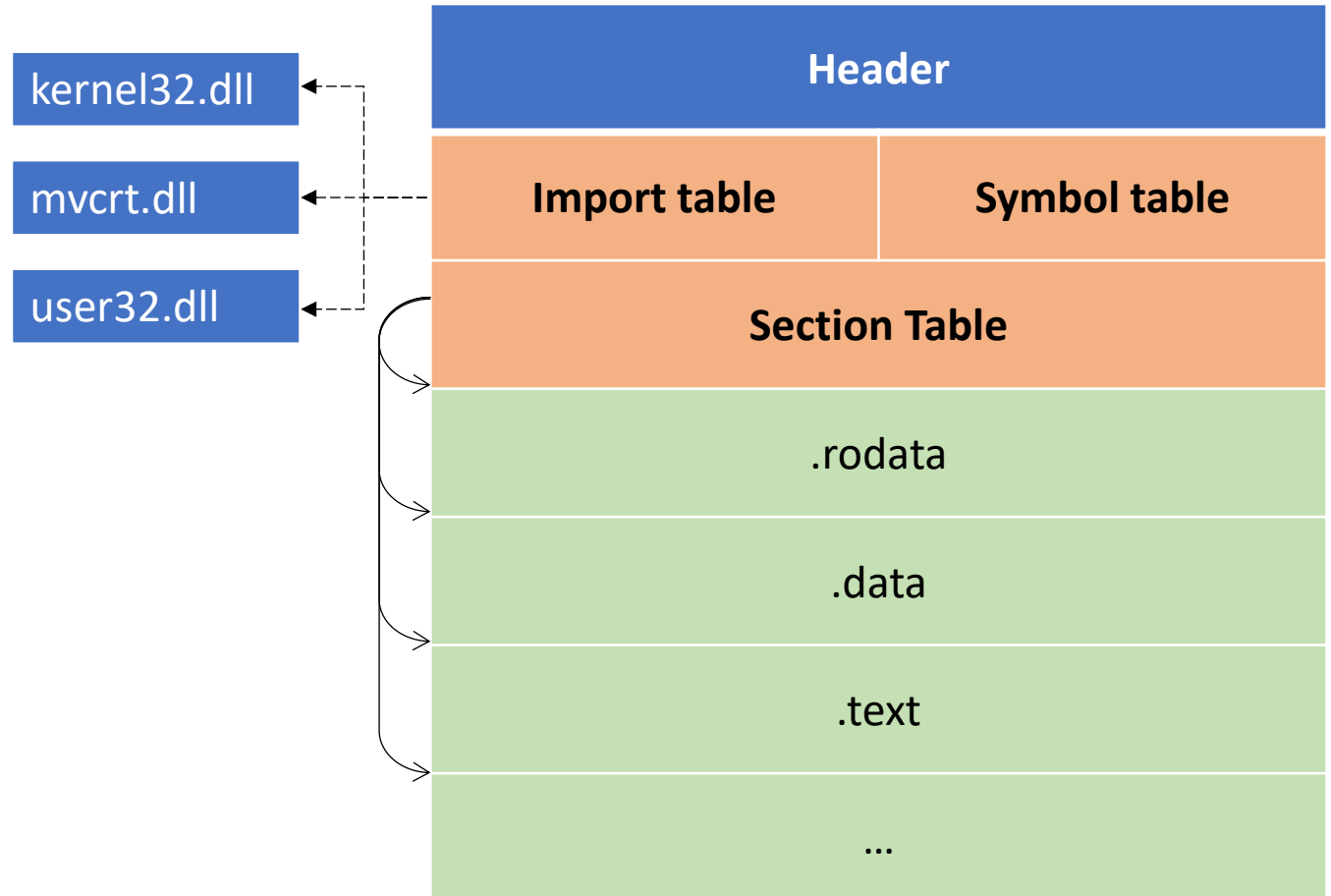
Таблица символов содержит информацию об определенных в этом файле символах.

Таблица импорта содержит информацию о внешних символах, которые должны быть загружены из других исполняемых файлов.

Таблица секций содержит информацию о расположении секций в файле, разрешениях на чтение/запись/выполнение для каждой секции и пр.

Кроме того, для секций может определяться, по какому относительному\* адресу из следует загружать.

*\*относительный адрес отсчитывается от точки загрузки программы.*

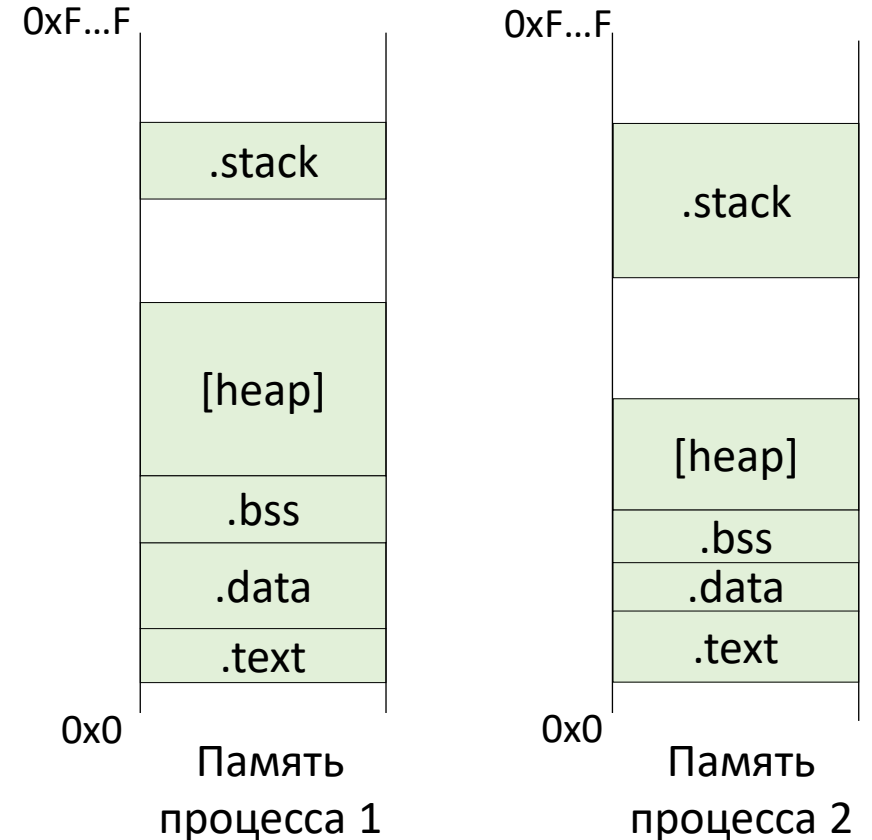


# Процессы и виртуальная память

В момент запуска исполняемого файла ОС создает **процесс** – изолированную абстракцию машины, внутри которой выполняется программа.

Каждый процесс имеет собственное **виртуальное адресное пространство** => программы не могут влиять друг на друга, т.к. каждая из них «живет» в своем пространстве.

Отображение виртуального пространства на физическое будет рассмотрено позднее.



# Загрузка исполняемого файла

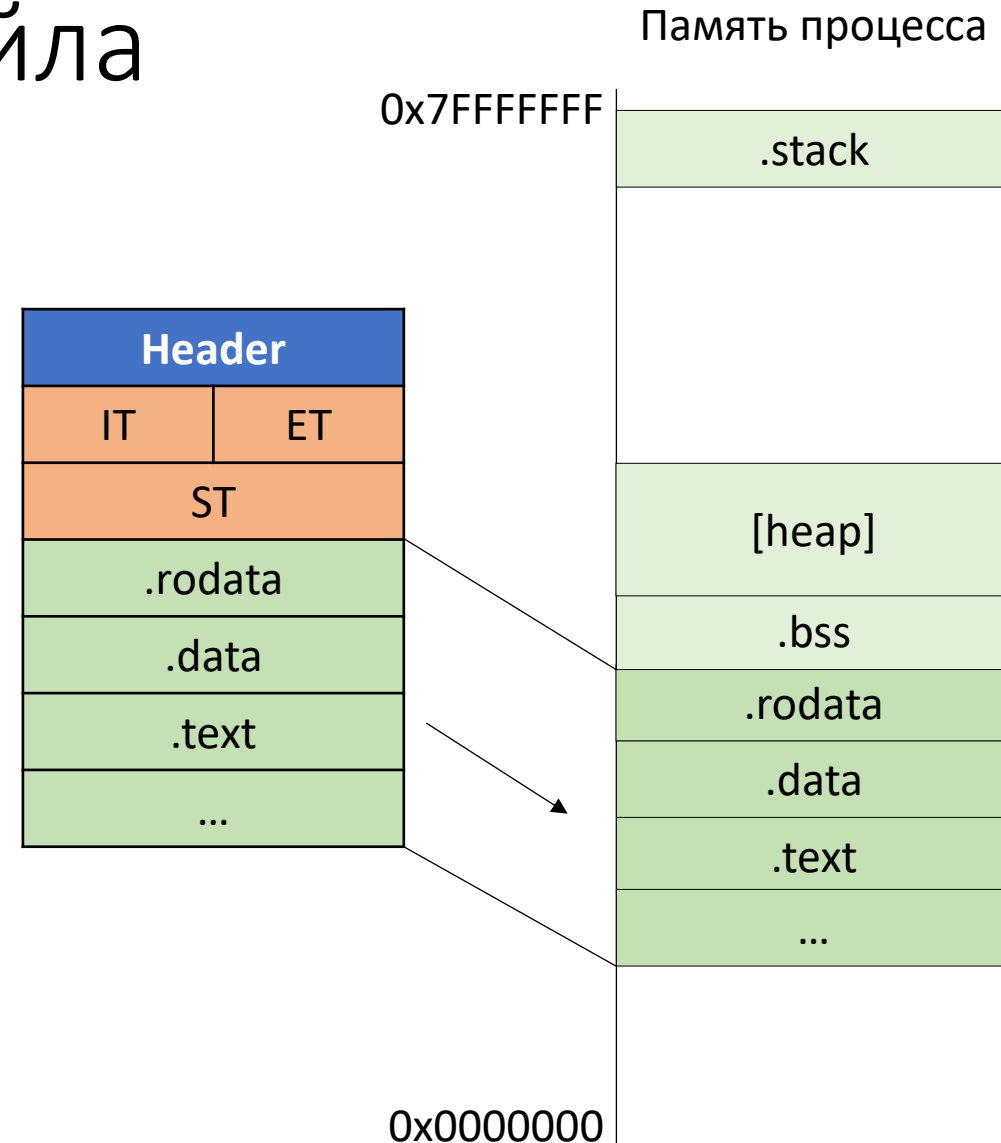
В ходе запуска программы секции программы копируются из исполняемого файла в память процесса. После этого для каждой секции в памяти выставляются соответствующие разрешения на чтение/запись/выполнение.

Затем создаются секции `.stack/.bss`. Размеры этих сегментов задаются в исполняемом файле.

В финальном этапе происходит переход на адрес точки входа – начинается выполнение программы.

*Каждый исполняемый файл и каждая библиотека имеет свои секции `.data`, `.rodata`, `.text` и т.д. Данные секции независимы, не добавляются друг к другу и не пересекаются друг с другом.*

Расположение библиотек в памяти зависит от порядка их загрузки.

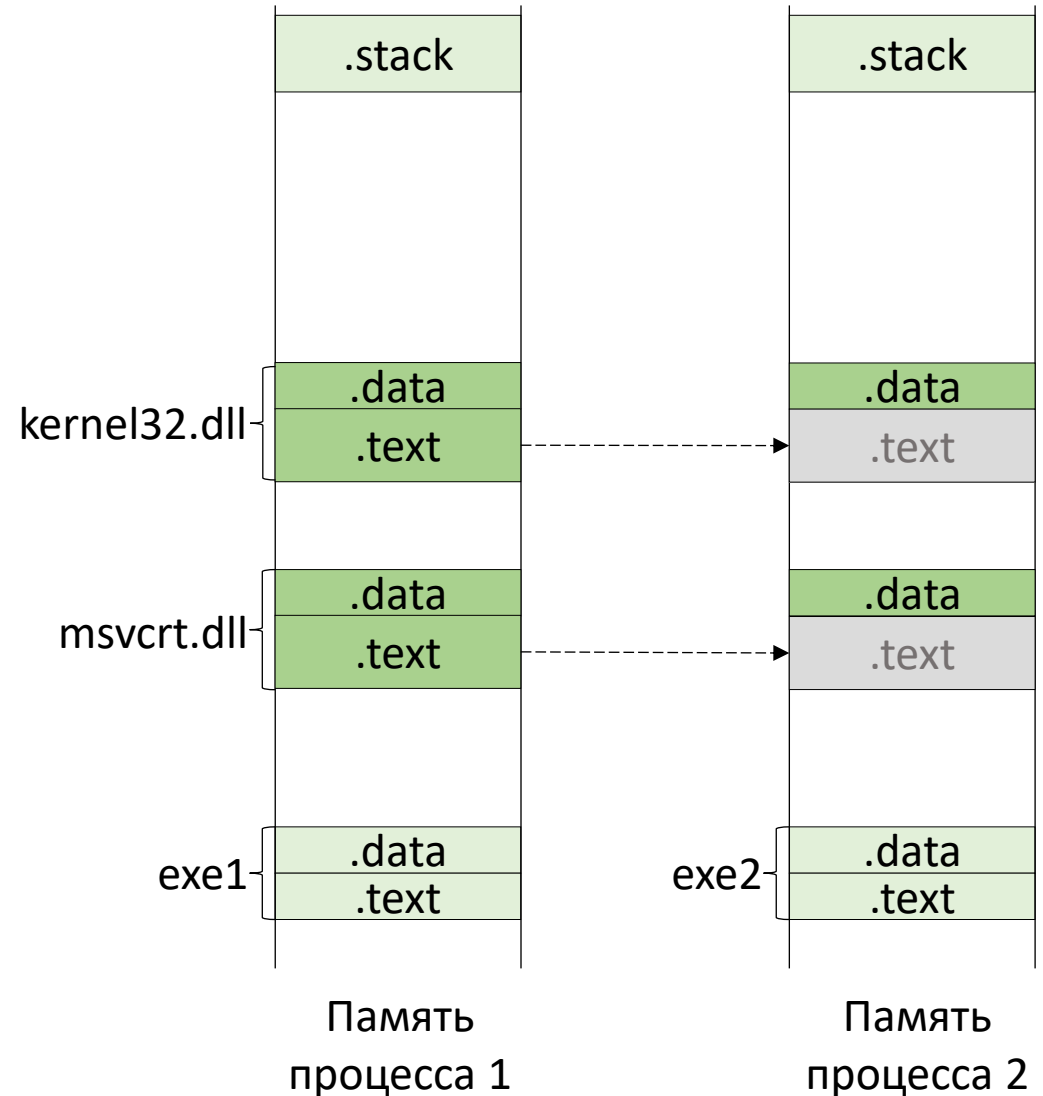


# Загрузка библиотек

Во время загрузки и/или во время исполнения программы наступает этап динамической компоновки.

В ходе динамической компоновки происходит загрузка динамической библиотеки в адресное пространство процесса и обнаружение в ней требуемых символов.

ОС поддерживает список уже загруженных разделяемых библиотек, и, при возможности, просто отображает загруженную библиотеку в адресное пространство процесса (т.е. библиотека занимает одну и ту же область *физической* памяти, но имеет разные *виртуальные* адреса).



# Проблема конфликта адресов

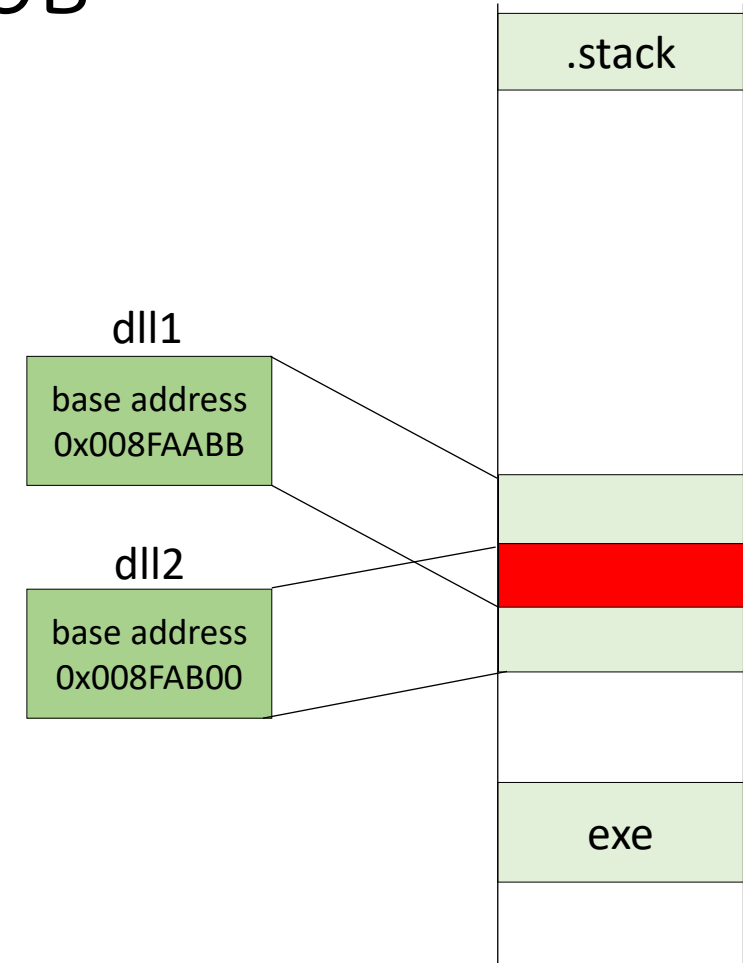
Изначально каждая библиотека загружалась в адресное пространство процесса по фиксированному базовому адресу\*.

Т.к. библиотек было мало, проблем не возникало. С ростом количества библиотек они начали конфликтовать из-за перекрытия.

Если в 16-битную эпоху было заранее известно, что функция F из библиотеки A после загрузки точно находится по определенному адресу, то в настоящее время адрес функции F становится известен только после загрузки.

Обычно для хранения таких адресов используется специальная секция, которая заполняется динамическим компоновщиком.

\* обычные исполняемые файлы, если явно не указать иное, до сих пор загружаются по фиксированному адресу



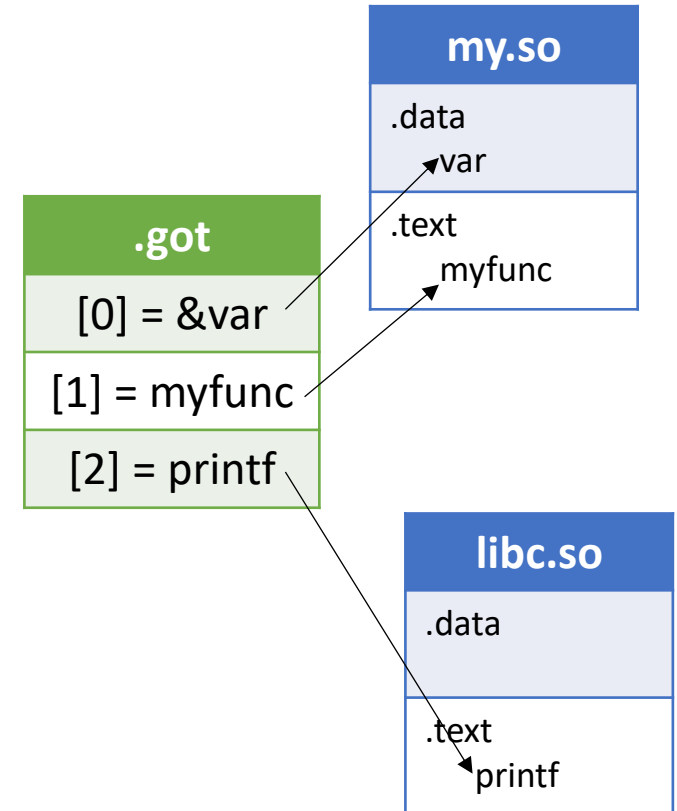
# Global Offset Table (ELF, UNIX-like)

В ELF-файлах для хранения адресов символов используется **глобальная таблица смещений** (GOT, Global Offset Table), хранящаяся в секции `.got` (может делиться на `.got` для адресов переменных и `.got.plt` для адресов функций).

Каждая загруженная библиотека тоже имеет свою GOT, в которой перечислены адреса символов из других библиотек.

В x86-64 GOT используется при обращении только к внешним для текущего модуля символам. В x86 GOT также участвовала в обращениях в внутренним символам.

Д/З: [GOT и GOTOFF \(10.2 x86\)](#), [перевод](#)



# Отложенная и немедленная загрузка (ELF)

Каждому внешнему символу соответствуют 2 записи – запись в GOT и запись в таблице импорта.

Внешние символы, в зависимости от их типа делятся на 2 группы: требующие немедленной загрузки и допускающие отложенную загрузку.

Символы-переменные всегда требуют немедленной загрузки.

Символы-функции допускают отложенную загрузку.

Для символов, требующих немедленной загрузки, библиотеки загружаются вместе с исполняемым файлом.

При компоновке исполняемого файла можно включить принудительную немедленную загрузку всех его зависимостей.

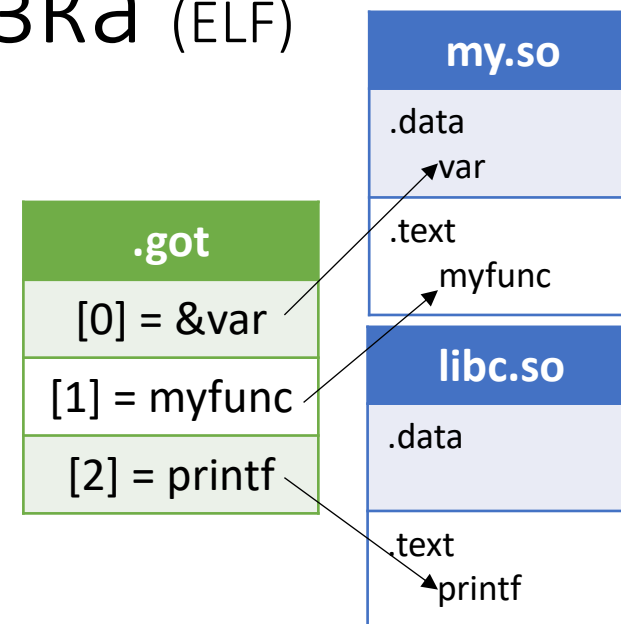


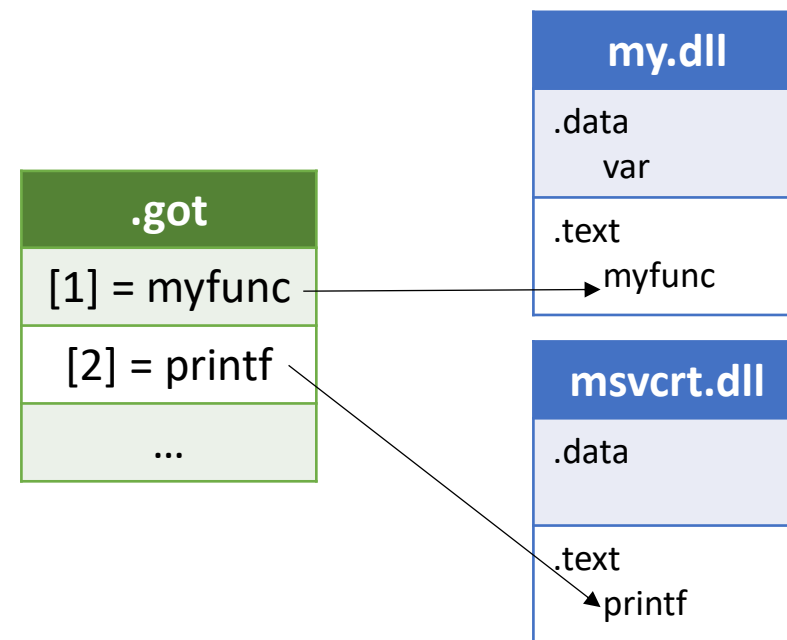
Таблица символов		
Имя	Тип	Значение
var	OBJECT->load now	???
myfunc	FUNC->lazy load	???
printf	FUNC->lazy load	???

# Отложенная загрузка (ELF)

Для отложенной загрузки используется дополнительная таблица **PLT** (Procedure Linkage Table) из секции `.plt` (в настоящее время может делиться на несколько таблиц).

PLT предназначена для хранения трамплинов.

**Трамплин** – это последовательность инструкций, которая завершается инструкцией безусловного перехода `jmp`.



```
printf("%d\n", x);  
↓  
call    printf@PLT  
↓  
section .plt  
printf@PLT: ←  
    jmp [printf@GOT]  
    push <printf index>  
    jmp PLT0 ; call linker
```



# Отложенная загрузка (ELF)

Первой инструкцией трамплина является прыжок по адресу из ячейки в `.got`, в которой должен быть записан адрес процедуры.

По умолчанию, в этой ячейке находится адрес следующей инструкции трамплина.

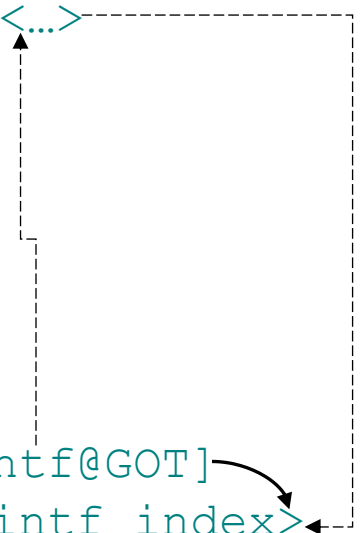
Оставшаяся часть трамплина записывает необходимые данные на стек<sup>1</sup> и вызывает `PLT[0]` – код динамического компоновщика.

Компоновщик загружает библиотеку, меняет GOT и вызывает требуемую функцию.

См. также: [habr](#)

```
section .got
; ...
printf@GOT: dq <...>
; ...

section .plt
printf@PLT:
    jmp [printf@GOT]
    push <printf index>
    jmp PLT0 ; call linker
```



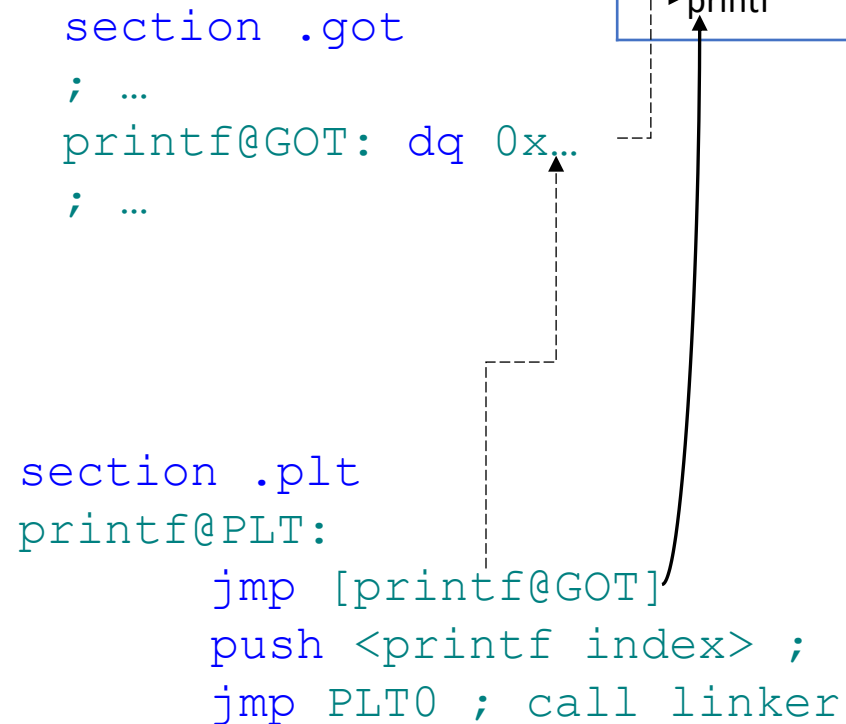
<sup>1</sup> фактически записывается номер записи в таблице релокаций динамически загружаемых символов

# Отложенная загрузка (ELF)

При последующих вызовах первая инструкция трамплина снова прыгнет на адрес из GOT.

Поскольку в GOT находится адрес процедуры, вызовется непосредственно процедура.

Минусом такого подхода является лишняя инструкция перехода. Плюсом – универсальность схемы.



# Адресация внутренних символов

Для доступа к внутреннему символу нужно знать его адрес. Обычно данный адрес рассчитывается компилятором или ассемблером автоматически, исходя из предположения о месте загрузки исполняемого файла и смещения символа относительно начала файла.

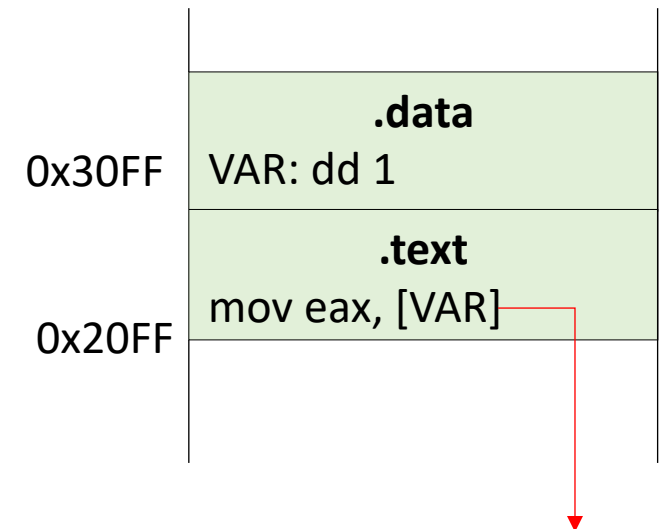
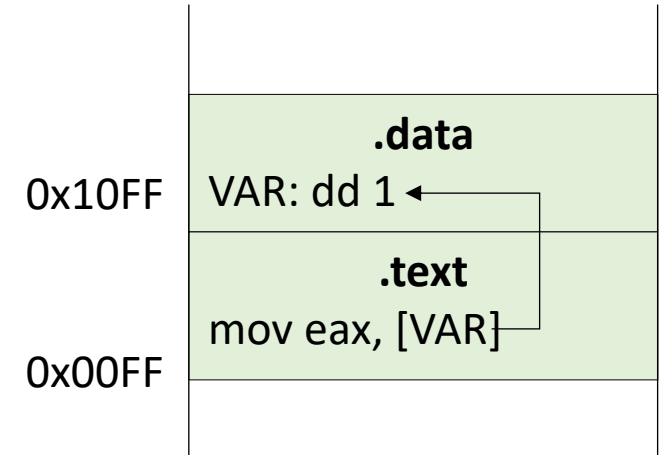
Если исполняемый файл может быть загружен по произвольному адресу, адрес символа может оказаться неверным – фактическое расположение символа будет отличаться.

В x86-32 для адресации внутренних символов приходилось использовать GOT или проводить релокацию.

В случае использования GOT обращение к внутренним символам производится также, как и к внешним символам.

Другим вариантом решения проблемы адресов является релокация.

*Справа:* метка VAR имеет значение 0x10FF, которое верно только в случае загрузки библиотеки по подходящему адресу.



# Релокация

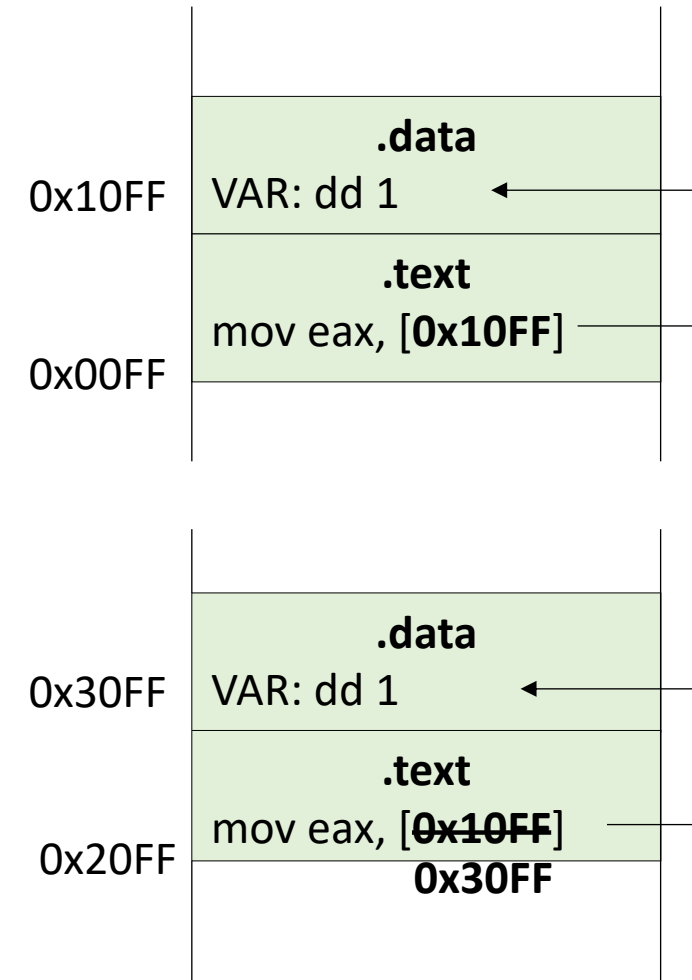
**Релокация** – правка содержимого исполняемого файла с целью адаптации к конкретному месту загрузки.

Специальная таблица релокаций (из заголовка исполняемого файла) для каждого символа хранит места, в которых используется его адрес.

В ходе загрузки компоновщик просматривает таблицу релокаций и в указанных местах исправляет адрес на корректный.

Минусом релокации является изменение содержимого секций кода/констант исполняемого файла – в результате, не получается отобразить соответствующие секции в разные адресные пространства целиком (отображаются только одинаковые части, измененные участки у каждого процесса будут свои). Кроме того, сама релокация увеличивает время загрузки.

Плюсом релокации является скорость работы программы – не требуются лишние обращения к GOT.



# RIP-адресация

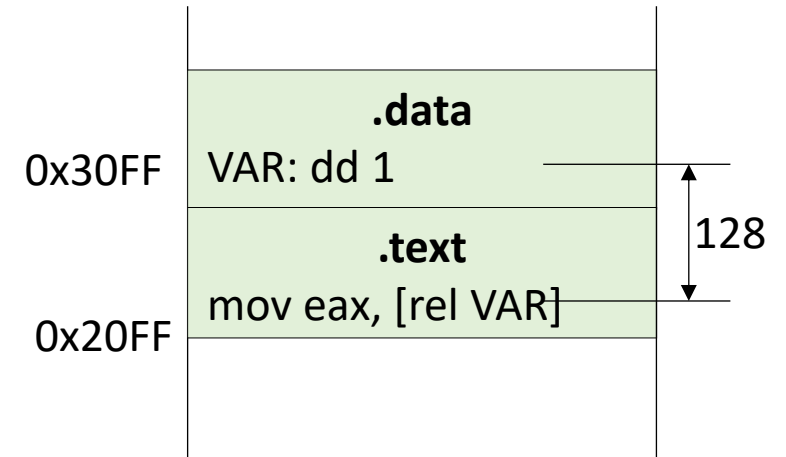
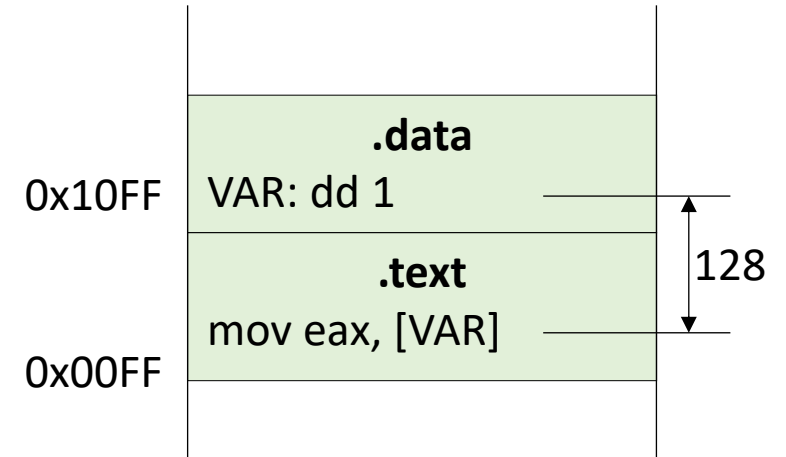
В x86-64 адресация внутренних символов была значительно упрощена.

Пусть в секции `.data` есть переменная `VAR`. Она имеет смещение в 128 байт от некоторой инструкции `mov`, в которой она используется.

После загрузки эта переменная имеет случайный адрес. Но смещение от инструкции `mov` до `VAR` не изменится.

Т.к. адрес следующей инструкции находится в регистре `RIP`, адрес переменной можно тоже рассчитать относительно `RIP`.

По умолчанию RIP-адресация не используется при компиляции исполняемых файлов.



# RIP-адресация в NASM

По умолчанию в NASM используется обычная адресация, что приемлемо для исполняемых файлов, но не для разделяемых библиотек.

Включить RIP-адресацию по умолчанию можно конструкцией `default rel` в начале файла.

Явно использовать RIP-адресацию можно с помощью префикса `rel`.

```
default rel  
; ...  
mov eax, [X]
```

```
mov rax, [rel X]
```

```
mov rax, [X+rip]
```

# Динамическая компоновка в Windows

Динамическая компоновка в Windows выполняется схожим образом, за исключением 3-х моментов:

1. Другие имена секций (в частности, Import Address Table в секции .idata вместо .got).
2. Отложенная загрузка по умолчанию отключена – все библиотеки по умолчанию загружаются сразу.
3. Использование специальных библиотек импорта при компиляции для установки связи между .exe и .dll.
4. Использование релокации вместо механизма GOT для адресации внутренних символов (преимущественно в x86-32).

# Порядок поиска библиотек

В исполняемом файле обычно хранится только имя файла библиотеки и некоторые дополнительные метаданные, но не полный путь к ней.

Поиск библиотеки осуществляется по правилам, установленным в ОС.

Обычно список каталогов поиска включает в себя системный каталог и каталоги, перечисленные в специальной переменной среды (PATH на Windows, LD\_LIBRARY\_PATH на Linux).

Каталог самой программы включен в список поиска на Windows, но не на Linux.

Д/З: [Порядок поиска на Windows](#) , [Порядок поиска на Linux](#)



# Атака подмены библиотеки (пример)

Если путь к разделяемой библиотеке не был задан явно, и атакующий имеет контроль над одним из каталогов поиска (например, каталогом программы), он может поместить туда измененную библиотеку, содержащую нужный ему код – т.е. осуществить **подмену библиотеки** (DLL hijacking).

Т.к. в ELF-файлах нет жесткой связи между импортируемым символом и библиотекой, можно также разместить символ с вредоносным содержимым в библиотеке, которая загружается раньше – при этом этот символ будет считаться загруженным.

Для защиты от таких атак существует ряд мер. Наиболее простым является запись полного пути библиотеки и/или предпочтительных каталогов поиска при сборке приложения, при условии, что атакующий не будет иметь к ним доступа.

Кроме того, на Linux можно указывать библиотеки, которые загружаются в первую очередь в переменное среды LD\_PRELOAD (можно использовать как для атаки, так и для защиты).

Кроме того, меры защиты могут быть предприняты также на уровне ОС, например путем установки более строгого порядка поиска библиотек (см. KnownDlls и SafeDllSearchMode на Windows).