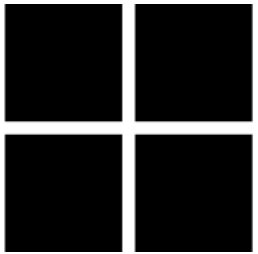
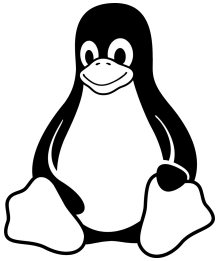
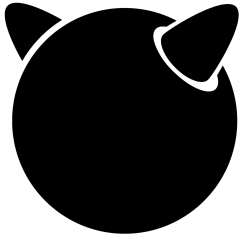
Operating systems





macOS



Исполняемые файлы и библиотеки

Исполняемые файлы

- На рисунке изображён исполняемый файл формата ELF (Unix, Linux, macOS)
- Исполняемый файл состоит из заголовков и секций
- Про секции text, data, bss, rodata мы уже знаем
- Программа пользуется библиотеками, где находятся внешние функции. Информация о том, где искать адреса внешних функций и глобальных данных находится в **таблице символов**

Как ядро запускает программу на примере Linux

Когда вызывается системный вызов exec(), происходит следующее:

- Ядро читает заголовок файла, чтобы определить его формат. В Linux это ELF, но поддерживаются и другие (скрипты, a.out и т.д.)
- Если это ELF-файл (магическое число: 0x7F 'E' 'L' 'F'), ядро парсит заголовок: проверяет архитектуру (x86, ARM и т.д.), тип (exec или shared object) и секции (.text для кода, .data для данных и т.д.)
- Если файл — скрипт (начинается с #!), ядро запускает интерпретатор (например, /bin/bash для .sh-скриптов) и передаёт ему скрипт как аргумент
- Если формат не поддерживается, возвращается ошибка (EACCES или ENOEXEC)

Как ядро запускает программу на примере Linux

- Ядро очищает текущее адресное пространство процесса, но сохраняет PID и другие метаданные
- Используя системный вызов `mmap()`, ядро отображает множество секций ELF-файла в виртуальную память, самые важные из них:
  - `.text` (код): отображается с правами на чтение и исполнение (`PROT_READ | PROT_EXEC`), обычно shared для экономии памяти
  - `.data` и `.rodata`: с правами на чтение/запись (`PROT_READ | PROT_WRITE` для .data)
  - `.bss`: инициализируется нулями (не хранится в файле, но выделяется в памяти)
- Адреса назначаются с учётом ASLR (Address Space Layout Randomization) для безопасности, чтобы адреса не были предсказуемыми
- Если программа динамически скомпонована, ядро загружает динамический загрузчик (linker) `/lib/ld-linux.so`. Этот загрузчик затем загружает библиотеки (например libc.so) также через `mmap()`

Как ядро запускает программу на примере Linux

- Ядро выделяет стек (stack) для процесса в верхней части адресного пространства (0x7fffffff000 на x86-64)

На стек помещаются:

  - Аргументы командной строки (argv)
  - Переменные окружения (envp, вроде PATH, HOME)
  - Auxiliary vector (auxv) – это дополнительные данные, такие как адрес загрузчика, случайные байты для ASLR и т.д.

- Указатели на argv и envp устанавливаются в регистрах (rdi, rsi на x86-64)
- Если есть, настраивается VDSO (Virtual Dynamic Shared Object) — специальная область от ядра для быстрых syscall'ов (`gettimeofday` и т. д.), без перехода в kernel space

Как ядро запускает программу на примере Linux

- После настройки ядро устанавливает точку входа (entry point), то есть адрес поля `e_entry` в ELF-заголовке (обычно `_start` в crt0.o)
- Процесс возвращается в user mode
- Управление передается на `_start()`, который вызывает `__libc_start_main()`, а затем `main()` программы
- Если программа статически скомпонована, загрузчик не нужен — сразу в код

Небольшой FAQ про memory mapping

- Q: Будет ли ядро и загрузчик выделять новую память, чтобы загрузить секции исполняемой программы и библиотек?
- A: Нет, потому что ядро отслеживает открытые программы и библиотеки и использует memory mapping, чтобы отобразить секции единственных загруженных экземпляров программы или библиотек. Так мы экономим память и пользуемся одной из важных возможностей виртуального адресного пространства

- Q: Как быть, если одна из секций (например `data`) экземпляра программы была изменена, но она была общей для всех программ
- A: Ядро пользуется принципом CoW

Статическая компоновка программ

- Compile time: компилируем код, получая объектные файлы, которые предоставляют символы, которые в них определены, и требуют символов, которые в них используются;
- Link time: во время компоновки объектных файлов в исполняемый файл компоновщик разрешает (resolves) символы в их адреса и подставляет эти адреса в машинный код. То есть патчит заглушки, где нужны адреса
- Runtime: образ исполняемого файла требует, чтобы его секции загрузили в память по фиксированным адресам, и рассчитывает на это в своей работе

Динамическая компоновка программ

- Link time: компоновщик оставляет некоторые символы неразрешёнными, но записывает в executable информацию о том, какие динамические библиотеки ему требуются для работы
- Runtime: динамический загрузчик разрешает используемые символы в их адреса, разыскивая символы в загруженных библиотеках

Динамическое разрешение символов

- Ядро этим заниматься не должно
- Программа, которая этим занимается, называется **динамический загрузчик**. Она должна исполняться в адресном пространстве той программы, в которой она загружается

Динамическое разрешение символов

- Когда запускается динамически скомпонованная программа, сначала загружается интерпретатор (interpreter), указанный в ELF-заголовке (обычно это `/lib64/ld-linux-x86-64.so.2`)
- Этот загрузчик (ld.so) анализирует зависимости программы (из секции **.dynamic** ELF-файла) и находит нужные .so-библиотеки (например, libc.so)
- Далее используется memory mapping, чтобы отобразить секции динамической библиотеки в виртуальное адресное пространство процесса

Динамическое разрешение символов

- Для каждой библиотеки ld.so открывает файл (через `open()`)
- Затем вызывается `mmap` несколько раз, чтобы отобразить разные секции ELF-файла в память:
  - .text (код): mmap с PROT_READ | PROT_EXEC, MAP_SHARED для исполняемого кода, который может быть shared между процессами.
  - .data и .bss (данные): mmap с PROT_READ | PROT_WRITE, MAP_PRIVATE для modifiable данных, с copy-on-write
  - .rodata (read-only данные): mmap с PROT_READ, MAP_SHARED
- Это происходит рекурсивно: библиотеки сами могут зависеть от других, так что ld.so загружает всю цепочку

Динамическое разрешение символов

- Теперь каким-то образом нужно **пропатчить** адреса вызовов библиотечных функции, но секция памяти для кода у нас открыта только на чтение (*по соображениям безопасности*)
- В коде программы на местах, где должны быть вызовы из динамических библиотек, будут прописаны адреса в специальную таблицу (PLT)
- Эту таблица является отдельной секцией и к ней есть доступ на чтение, то есть динамический загрузчик может пропатчить адреса в этой таблице

Procedure Linkage Table

- PLT это таблица в исполняемом файле или библиотеке, где каждая запись соответствует вызову внешней функции, создается компоновщиком
- Когда программа вызывает функцию, она передаёт управление в соответствующую запись PLT
- Первоначально запись в PLT указывает на код в динамическом загрузчике, который разрешает (resolves) адрес функции
- После первого вызова адрес функции сохраняется в GOT, и последующие вызовы через PLT используют этот адрес напрямую, минуя загрузчик (ленивое связывание, lazy binding)

Global Offset Table

GOT хранит абсолютные адреса глобальных переменных и функций, чтобы код библиотеки мог к ним обращаться, не зная их точного расположения в памяти на этапе компиляции

На этапе загрузки библиотеки динамический загрузчик (dynamic linker, например, ld.so в Linux) заполняет GOT актуальными адресами.

Как компилировать *динамические* библиотеки

`gcc -o` **`lib<название>.so -shared -fPIC`** `<код.c> <еще-код.c> ...`

- `-shared` компонует динамическую библиотеку вместо исполняемого файла
- `-fPIC` создаёт PLT и GOT сегменты и патчит адреса в этих секциях, чтобы вызовы функций ссылались на пропатченные адреса в таблице

При попытке сделать `-shared` без `-fPIC`, динамический загрузчик ***попробует*** сделать секцию кода доступной для чтения и пропатчить адреса там, не генерируя для этого таблицы. Это может не сработать!

Помимо этого, в каждый процесс, использующий библиотеку без `-fPIC`, будет иметь свою копию кода библиотеки, а не отображение существующей!

Как компилировать *динамические* библиотеки

`gcc -o` **`lib<название>.so -shared -fPIC`** `<код.c> <еще-код.c> ...`

- `-shared` компонует динамическую библиотеку вместо исполняемого файла
- `-fPIC` создаёт PLT и GOT сегменты и патчит адреса в этих секциях, чтобы вызовы функций ссылались на пропатченные адреса в таблице

При попытке сделать `-shared` без `-fPIC`, динамический загрузчик ***попробует*** сделать секцию кода доступной для чтения и пропатчить адреса там, не генерируя для этого таблицы. Это может не сработать!

Помимо этого, в каждый процесс, использующий библиотеку без `-fPIC`, будет иметь свою копию кода библиотеки, а не отображение существующей!

Можно загружать такие библиотеки в runtime

```
#include <dlfcn.h>

typedef float custom_fabsf_func(float x);
custom_fabsf_func custom_fabsf = NULL;

int main() {
    void *library = dlopen(“./libcustom.so”,
                           RTLD_LOCAL | RTLD_LAZY);

    custom_fabsf = dlsym(library, "custom_fabsf");

    float x = custom_fabsf(-3.14159f);
```