

Операционные Системы

Исполняемые файлы

January 5, 2019

Первый процесс

- ▶ Ядро ОС настроило прерывания, аллокаторы, планировщик. Что дальше?
 - ▶ мы должны запустить первый процесс и первое приложение!
 - ▶ например, Linux проверяет файлы: `/sbin/init`, `/etc/init`, `/bin/init` и `/bin/sh`.

Исполняемые файлы

- ▶ Существует множество форматов исполняемых файлов:
 - ▶ ELF, a.out, PE, COM, Mach-O;
 - ▶ скрипты, начинающиеся с `#!` (sha-bang).

Заголовок исполняемого файла

- ▶ Заголовок:
 - ▶ magic number/string - позволяет быстро определить формат файла;
 - ▶ различного рода флаги и параметры:
 - ▶ версия формата исполняемого файла;
 - ▶ архитектура;
 - ▶ ссылки на другие части файла.

Заголовок ELF файла

```
1      struct elf64_hdr {
2          uint8_t e_ident[16];
3          uint16_t e_type;
4          uint16_t e_machine;
5          uint32_t e_version;
6          uint64_t e_entry;
7          uint64_t e_phoff;
8          uint64_t e_shoff;
9          uint32_t e_flags;
10         uint16_t e_ehsize;
11         uint16_t e_phentsize;
12         uint16_t e_phnum;
13         uint16_t e_shentsize;
14         uint16_t e_shnum;
15         uint16_t e_shstrndx;
16     } __attribute__((packed));
```

Точка входа

- ▶ У любой программы есть первая инструкция - точка входа:
 - ▶ формат исполняемого файла явно или не явно указывает адрес первой инструкции;
 - ▶ ОС после загрузки исполняемого файла передает управление первой инструкции;
 - ▶ обычно передача управления сопровождается понижением уровня привилегий кода (переходом в userspace).

Заголовок ELF файла

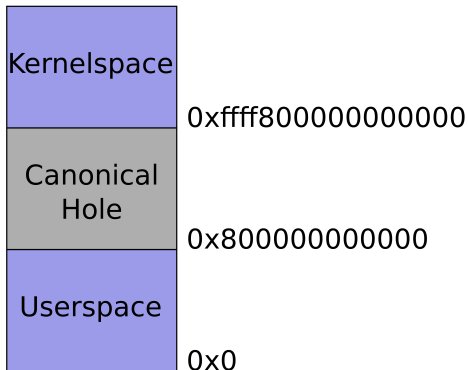
```
1      struct elf64_hdr {
2          uint8_t e_ident[16];
3          uint16_t e_type;
4          uint16_t e_machine;
5          uint32_t e_version;
6
7          /* Logical address of the first instruction */
8          uint64_t e_entry;
9
10         uint64_t e_phoff;
11         uint64_t e_shoff;
12         uint32_t e_flags;
13         uint16_t e_ehsize;
14         uint16_t e_phentsize;
15         uint16_t e_phnum;
16         uint16_t e_shentsize;
17         uint16_t e_shnum;
18         uint16_t e_shstrndx;
19     } __attribute__((packed));
```

Описание адресного пространства

- ▶ Формат исполняемого файла описывает логическое адресное пространство:
 - ▶ какие участки логического адресного пространства нужны и для чего;
 - ▶ где в памяти процесса должны располагаться код и данные;
 - ▶ где в исполняемом файле хранятся код и данные.

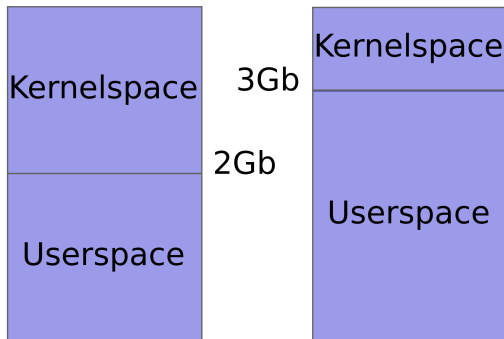
Типичное адресное пространство

x86-64

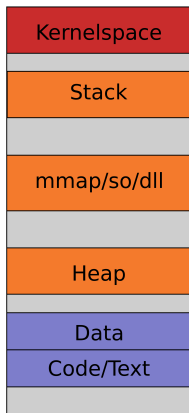


Типичное адресное пространство

x86-32



Типичное адресное пространство



Elf Program Headers

```
1      struct elf64_hdr {
2          uint8_t e_ident[16];
3          uint16_t e_type;
4          uint16_t e_machine;
5          uint32_t e_version;
6          uint64_t e_entry;
7
8          /* Offset of the program header table */
9          uint64_t e_phoff;
10
11         uint64_t e_shoff;
12         uint32_t e_flags;
13         uint16_t e_ehsize;
14
15         /* The size of a program header table entry */
16         uint16_t e_phentsize;
17         /* The number of entries in the program
18            header table */
19         uint16_t e_phnum;
20
21         uint16_t e_shentsize;
22         uint16_t e_shnum;
23         uint16_t e_shstrndx;
24     } __attribute__((packed));
```

Elf Program Headers

```
1      struct elf64_phdr {
2          /* There are different types of segments,
3             we need PT_LOAD == 1 */
4          uint32_t p_type;
5
6          /* Read/Write/Execute */
7          uint32_t p_flags;
8
9          /* Offset of the segment in the file */
10         uint64_t p_off;
11
12         /* The logical address of the segment in memory */
13         uint64_t p_vaddr;
14         uint64_t p_paddr;
15
16         /* The size of the file image of the segment */
17         uint64_t p_filesz;
18
19         /* The size of the memory image of the segment */
20         uint64_t p_memsz;
21
22         uint64_t p_align;
23     } __attribute__((packed));
```

Загрузка исполняемого файла

- ▶ Подготовить адресное пространство согласно описанию в файле
 - ▶ аллоцировать память и настроить таблицы страниц;
 - ▶ скопировать код и данные из файла в память;
 - ▶ возможно создать стек отдельно.
- ▶ "Прыгнуть" в userspace
 - ▶ передать управление точке входа, указанной в файле;
 - ▶ возможно, понизить уровень привилегий.

Библиотеки

- ▶ Виды библиотек:
 - ▶ статические - становятся частью исполняемого файла;
 - ▶ динамические - хранятся отдельно от исполняемого файла
 - ▶ загружаются при запуске приложения или по требованию.

Динамические библиотеки

- ▶ Особенность динамических библиотек - могут быть загружены по разным адресам
 - ▶ как код библиотеки обращается к своим коду и данным?
 - ▶ как код приложения обращается к библиотеке?
 - ▶ как код библиотек обращается к коду и данным других библиотек?

Компоновщик

- ▶ Компоновщик (linker, link editor) - программа, которая "связывает" бинарные файлы вместе и генерирует исполняемый файл
 - ▶ в момент компиляции адреса функций/переменных могут быть не известны;
 - ▶ компилятор просто оставляет "пустое место", а компоновщик записывает в него адрес.

Динамический компоновщик

- ▶ Адреса функций/переменных из динамических библиотек не известны
 - ▶ компилятор/статический компоновщик оставляют "пустые места";
 - ▶ динамический компоновщик должен записать в них адреса, после того как библиотека была загружена.

Загрузка ELF файла с динамическими библиотеками

- ▶ ELF файл загружается как обычно
 - ▶ ищем Program Header-ы с типом PT_LOAD и загружаем их в память.
- ▶ Смотрим в Program Header с типом PT_INTERP
 - ▶ там хранится имя файла динамического компоновщика;
 - ▶ загружаем его в память в дополнение к программе.
- ▶ Передаем управление *динамическому компоновщику*.

Поиск динамических библиотек

- ▶ Исполняемый файл должен хранить информацию о динамических библиотеках
 - ▶ например, ELF Program Header с типом `PT_DYNAMIC` указывает, где в файле хранится эта информация;
 - ▶ динамический компоновщик загружает все требуемые зависимости в память.

Редактирование связей

- ▶ Исполняемый файл и динамические библиотеки хранят список обращений к внешним сущностям
 - ▶ вызовы функций из других (и не только) библиотек;
 - ▶ обращения к переменным (и не только) из других библиотек;
 - ▶ динамический компоновщик находит адреса и записывает их в определенные места в памяти.

GOT

- ▶ ELF формат использует Global Offset Table (GOT)
 - ▶ код, обращающийся к переменной, знает относительный адрес GOT и номер записи в ней, соответствующей этой переменной;
 - ▶ компилятор генерирует код, который берет адрес из GOT;
 - ▶ динамический компоновщик записывает в GOT правильные адреса при загрузке.

PLT

- ▶ ELF формат также использует Procedure Linkage Table (PLT)
 - ▶ код обращающийся к функции знает относительный адрес PLT и номер "заглушки" в ней, соответствующей этой переменной;
 - ▶ компилятор генерирует код, который вызывает "заглушку" из PLT вместо реальной функции;
 - ▶ динамический компоновщик может изменять PLT, а может изменять GOT, к которой "заглушка" из PLT обращается.