Operating Systems

System calls and app loading

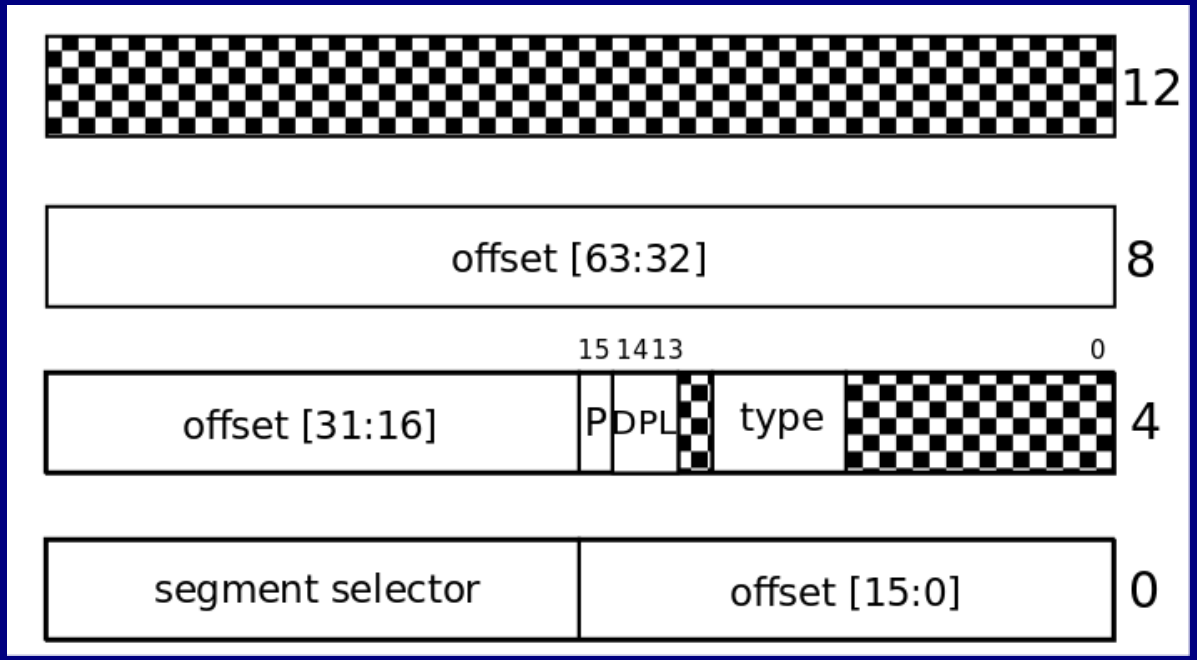
Me

December 9, 2016

Системные вызовы

- Системные вызовы - это механизм взаимодействия непривелигерованного кода (пользовательских приложений) и привелигированного кода (ядра)
  - open/close, read/write, fork, exec и др. требуют обращения к ядру ОС.
- Системный вызов осуществляет контролируемое повышение привелегий:
  - нельзя дать произвольному коду выполняться в привелигированном режиме;
  - при системном вызове вызывается определенная функция ядра ОС;
  - функция ОС должна параноидально проверять все аргументы системного вызова на корректность.

Организация системных вызовов



- Типичный способ организации системного вызова - прерывания:
  - например, в x86 можно использовать запись IDT с DPL равным 3;
  - на практике используют специальные инструкции syscall/sysenter потому что они работают быстрее.

Передача аргументов и возврат значений

- При системном вызове типичным способом передачи аргументов и возврата значений являются регистры:
  - например, в Linux Kernel используется следующая конвенция:
    - регистр RAX соедржит номер системного вызова;
    - регистры RDI, RSI, RDX, R10, R8, R9 содержат аргументы;
    - результат возвращается в регистре RAX.

Пример: linux hello world

```
1         .text
2         .global main
3 main:
4         /* write to stdout */
5         movq $1, %rax              // rax — syscall number
6         movq $1, %rdi              // rdi — file descriptor
7         movq $msg_begin, %rsi      // rsi — string pointer
8         movq msg_len, %rdx         // rdx — size
9         syscall
10
11        /* exit */
12        movq $60, %rax
13        movq $0, %rdi
14        syscall
15
16        .data
17 msg_begin:
18        .ascii "Hello, World\n"
19 msg_len:
20        .quad msg_len - msg_begin
```

Стек системного вызова

- Ядро оперирует данными всех приложений - не хочется, чтобы эти данные "утекали" в пространство пользователя:
  - если обработчик системного вызова будет использовать стек приложения после возврата данные могут "утеч" в пространство пользователя;
  - соответсвенно обработчик должен использовать другой стек, либо стек нужно "занулять" на выходе из системного вызова.

Стек системного вызова

- Ядро не знает размер стека пользовательского приложения:
  - приложение могло использовать почти весь свой стек, и не оставить ничего ядру - мы не знаем, что приложение делало до системного вызова;
  - т. е. для обработчика системного вызова, в конечном итоге, требуется свой стек, который выделяется ядром.

Task-State Segment (TSS)

- Task-State Segment - область памяти, в которой хранится информация о состоянии потока исполнения:
  - в 32 битном x86 могла использоваться для аппаратного переключения контекста;
  - в 64 битном x86 хранит указатель стека привилегированного режима
    - при системном вызове/прерывании CPU загружает в RSP значения записанные в TSS перед вызовом обработчика.
  - структура TSS описана в разделе 7.7 Task Management in 64-bit Mode документации Intel.

Использование TSS

- Для использования TSS нужно выполнить две вещи:
  - завести дескриптор в GDT описывающий TSS (формат дескриптора приведен в разделе 7.2.3 TSS Descriptor in 64-bit mode документации Intel);
  - загрузить селектор этой записи в Task Register используя инструкцию LTR (раздел 7.2.4 Task Register документации Intel).

Соотношение TSS и потоков

- Похоже, изначально предполагалось использовать свою TSS для каждого потока
  - гораздо проще завести по одной TSS на ядро, загрузить Task Register один раз для каждого ядра и при переключении потоков изменять непосредственно TSS;
  - таким образом получается, что у каждого потока есть два стека: стек пространства ядра и стек пространства пользователя (если поток вообще работает в пространстве пользователя).

Исполняемые файлы

- Ядро ОС само по себе не очень полезно - пользователи имеют дело с прикладными приложениями
  - соответственно, для полноценной ОС желательно уметь загружать и запускать программы.
- Программы представляются как один или несколько исполняемых файлов:
  - скрипты с аттрибутом +x и # в начала файла;
  - ELF файлы (опять же с аттрибутом +x);
  - PE/COM файлы (это в Windows);
  - Mach-O (это в Mac OS).

Бинарные исполняемые файлы

- Бинарный исполняемый файл содержит:
  - код и данные необходимые для выполнения;
  - ссылки на другие бинарные файлы (разделяемые библиотеки);
  - может содержать отладочную информацию.
- Бинарный исполняемый файл определяет
  - где в памяти должны располагаться данные и код;
  - где находится точка входа в программу (условно, адрес main).

Формат ELF

```
1  struct elf_hdr {
2      uint8_t e_ident[
           ↪ ELF_NIDENT];
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));
```

- ELF файл начинается с заголовка с общей информацией;
- тип, версия, архитектура - ОС должна проверить файл на валидность;
- адрес точки входа так же хранится здесь - *e_entry*.

Program Header

- Program Header, упрощенно, описывает участок памяти который должна подготовить ОС
  - все Program Header-ы хранятся в таблице в ELF файле;
  - при загрузке ELF файла, ОС должна прочитать эту таблицу и подготовить участки памяти.
- Таблица Program Header-ов:
  - смещение таблицы в файле хранится в поле *e_phoff*;
  - количество записей в таблице хранится в поле *e_phnum*;
  - размер каждой записи хранится в поле *e_phentsize*.

Program Header для x86

- *p_type* - тип заголовка (на интересует только *PT_LOAD* == 1);
- *p_vaddr* и *p_memsz* - адрес и размер в памяти;
- *p_offset* и *p_filesz* - смещение и размер в файле данных, которые нужно загрузить в память;
- *p_filesz* может быть меньше *p_memsz*, хвост должен быть заполнен нулями.

```
1  struct elf_phdr {
2      uint32_t p_type;
3      uint32_t p_flags;
4      uint64_t p_offset;
5      uint64_t p_vaddr;
6      uint64_t p_paddr;
7      uint64_t p_filesz;
8      uint64_t p_memsz;
9      uint64_t p_align;
10 } __attribute__((packed));
```

Загрузка ELF файла

- Таким образом загрузка ELF файла в простом случае состоит из следующих действий:
  - проверяем заголовок ELF файла - что это ELF файл для нужной архитектуры;
  - читаем таблицу Program Header-ов аллоцируем память и подготавливаем таблицу страниц для всех Program Header-ов с типом *PT_LOAD*;
  - передаем управление точке входа с переключением процессора в непривелигированный режим работы.

Библиотеки

- Библиотеки позволяют переиспользовать код (зачастую без перекомпиляции):
  - библиотеки можно организовывать разными способами, например:
    - статические библиотеки - просто наборы объектных файлов, которые можно слинковать с вашей программой;
    - динамические библиотеки подключаются во время работы программы.
- Преимущества динамических библиотек:
  - не нужно хранить один и тот же код в нескольких экземплярах (в отличие от статических библиотек);
  - можно избежать дублирования одного и того же кода в памяти (загрузить динамическую библиотеку в единственном экземпляре на все программы).

Динамические библиотеки

- Динамическая библиотека может быть загружена по любому адресу:
  - почему мы не можем зафиксировать адрес?
  - исполняемый файл не может заранее знать адреса функций и данных из библиотеки;
  - сама библиотека не может знать адреса своих функций и данных заранее.

Position Independent Code (PIC)

- В момент сборки мы можем не знать абсолютные адреса функций/переменных, но мы можем знать относительные:
  - функция которую мы хотим вызвать находится на x байт выше/ниже текущей инструкции (значения регистра RIP);
  - аналогично для данных.
- PIC решает проблему обращения к функциям и переменным внутри динамической библиотеки
  - если библиотека зависит от другой библиотеки - у нас все еще проблемы;
  - программам/библиотекам зависящим от нас это не помогает.

Динамический компоновщик

- Динамический компоновщик решает ту же задачу, что и обычнй, но в других условиях
  - в ELF файле может быть специальный Program Header с типом $PT\_INTERP == 3$ - он указывает на путь к динамическому компоновщику;
  - вместе с имполняемым файлом ОС загружает динамический компоновщик *и передает ему управление*.
- Динамический компоновщик загружает нужные динамические библиотеки
  - информация о необходимых библиотеках хранится в месте, на которое указывает Program Header $PT\_DYNAMIC$;
  - т. е. в момент сборки мы должны знать обо всех библиотеках, от которых мы зависим.

Редактирование связей

- Динамический компоновщик нашел и загрузил все библиотеки в память, что дальше?
  - теперь ему известны адреса всех нужных функций и переменных;
  - он может подредактировать память и проставить в нужных местах ссылки.
- Нужное место - Global Offset Table (GOT):
  - по сути, таблица адресов всех объектов, к которым нам нужно обращаться;
  - динамический компоновщик зная адреса просто заполняет GOT.

Пример

```
1  int bar;
2
3  void foo(void)
4  { ++bar; }
```

```
1  foo:
2     push   %rbp
3     mov    %rsp,%rbp
4
5     mov    0x20093d(%rip),%rax // 0x200fd8
6     mov    (%rax),%eax
7     lea    0x1(%rax),%edx
8     mov    0x200931(%rip),%rax
9     mov    %edx,(%rax)
10
11    nop
12    pop    %rbp
13    retq
```

Пример

- Куда указывает *0x200fd8*? Посмотрим с помощью *readelf -S*:

```
1  [Nr] Name Type      Address  Offset  Size  EntSize  Flags Link Info
       ↪ Align
2  ...
3  [19] .got PROGBITS  200fd0   fd0     30    8        WA    0    0    8
4  ...
```

  - динамический компоновщик после загрузки libfoo.so запишет в нужное место .got адрес переменной bar;
  - а адрес .got находится с помощью относительной адресации.

Пример

```
1  main:
2     push   %rbp
3     mov    %rsp,%rbp
4
5     mov    0x200930(%rip),%eax
6     mov    %eax,%esi
7     mov    $0x4007d4,%edi
8     mov    $0x0,%eax
9     callq  4005d0 <printf@plt>
10
11    callq  4005f0 <foo@plt>
12
13    mov    0x200914(%rip),%eax
14    mov    %eax,%esi
15    mov    $0x4007e6,%edi
16    mov    $0x0,%eax
17    callq  4005d0 <printf@plt>
18
19    mov    $0x0,%eax
20    pop    %rbp
21    retq
```

```
1  int main(void)
2  {
3     printf("bar = %d\n", bar);
4     foo();
5     printf("bar = %d\n", bar);
6     return 0;
7  }
```

Пример

- Что за функция *foo@plt*?

```
1  foo@plt:
2    jmpq   *0x200a32(%rip) // 0x601028
3    pushq  $0x2
4    jmpq   4005c0 <_init+0x20>
```

  - адрес *0x601028* просто ссылается на определенное место в got;
  - т. е. foo@plt просто передает управление по адресу записанному в got;
  - в конечном итоге, там должен оказаться адрес функции foo.

Q&A