

# Лекция 18

# Оператор switch

- В зависимости от количества и группировки значений:
  - Линейная цепочка if
  - Дерево if
  - Таблица переходов

# Таблица переходов

```
int a;  
switch (a) {  
case 1: // block-1  
    break;  
case 2: // block-2  
    break;  
case 3: // block-3  
    break;  
}
```

```
mov a, %eax  
sub $1, %eax  
cmp $2, %eax  
ja out_switch  
jmp *swtab(,%eax,4)  
swtab: .int blk1, blk2, blk3  
blk1: jmp    out_switch  
blk2: jmp    out_switch  
blk3: jmp    out_switch  
out_switch:
```

# Floating point

- Устройство вычислений с плавающей точкой может отсутствовать на простых процессорах (low-end микроконтроллерах)
  - Заменяется фиксированной точкой или программным вычислением
- X86/x64 два разных (!) устройства для вычислений с плавающей точкой

# X86 Legacy - FPU

- 8 регистров размером 80 бит (то есть тип long double), организованных в стек: %st(0), %st(1), ..., %st(7)
- Загрузка из памяти: варианты fld
- Сохранение в память: варианты fst
- Разные вычисления: fadd, fsub, ..., fsin

# X86 calling convention

- Если подпрограмма возвращает результат типа float или double, он возвращается в %st(0)
- Результат должен быть удален из стека FPU, даже если не используется (ответственность вызывающей стороны)
- Регистры %st(0), ..., %st(7) – scratch

# SSE и расширения

- SSE, SSE2, ..., - набор SIMD команд, или векторные расширения
- SIMD (single instruction multiple data) – классификация Флинна обработки данных
  - SIMD (single instruction multiple data)
  - SISD (single instruction single data)
  - ...
- Другой пример SIMD – вычисления на GPU
- Одна инструкция обрабатывает сразу несколько однотипных наборов данных (вектор)

# Регистры SSE

- X86: 8 регистров %xmm0 - %xmm7 – 128 бит
- X64: 16 регистров %xmm0 - %xmm15
- AVX: 256-битные регистры %ymm
- %mxcsr – регистр статуса
- Все регистры – scratch
- На x64 в %xmm передаются параметры, в %xmm0 возвращается результат



# Упаковка/распаковка

- Один регистр %xmm может содержать:
  - 2 числа типа double
  - 4 числа типа float
  - 2 64-битных целых числа
  - 4 32-битных
  - 8 16-битных
  - 16 8-битных
- Операции выполняются над всеми числами за раз (как правило)
- Для %ymm регистров одновременно обрабатывается в два раза больше значений
- Можно использовать для обычных (скалярных) операций
- В AVX поддерживаются трехадресные операции: SRC1, SRC2, DST

# Выравнивание аргументов

- MOVDQA – загружает значение XMM из памяти целиком (128 бит)
- PADDD – складывает два вектора из 4 32-битных значений в XMM регистрах или памяти
- У подобных инструкций адрес в памяти должен быть **выровнен по 16**
- Иначе – Segmentation Fault

# Сравнение floating point

- Инструкция COMISD сравнивает две младших половины (double) регистров или памяти  
`comisd %xmm1, %xmm0 // xmm0 – xmm1`
- Устанавливаются флаги PF (unordered – то есть при операциях с NaN); CF, ZF
- Можно использовать беззнаковые переходы:  
`ja // переход, если %xmm0 > %xmm1`

# Где почитать

- <https://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf>
- <http://x86.renejeschke.de/>
-

# Кодирование инструкций

- Каждая инструкция однозначно представляется в бинарном виде
- На x86/x64 инструкция кодируется последовательностью от 1 до 16 байт
- Требования к кодированию:
  - Однозначность декодирования
  - Компактность

# Кодирование относительно IP

- Инструкция jmp:  
8048098:eb 17 jmp 80480b1
- Занимает 2 байта, адрес после нее: 804809A
- В инструкции кодируется смещение относительно EIP: 17
- $804809a + 17 = 80480b1$
- На x86 так кодируются инструкции call, jmp, jCC
- На x64 существует специальный режим адресации: RIP-relative:  
mov L1(%rip), %rax

# Абсолютное кодирование

- Загрузка адреса в памяти на регистр:  
80480a2: b9 b5 80 04 08 mov \$0x80480b5,%ecx  
80480b5: 48 65 6c 6c 00 .asciz "Hell"
- Загрузка значения глобальной переменной в регистр
- В закодированной инструкции записывается абсолютный адрес в памяти

# Адресация относительно rip (x86\_64)

- 64-битная система команд позволяет использовать регистр rip (содержит адрес инструкции, следующей за текущей) для адресации памяти

```
0000000000401146 <func>:
 401146:      48 8d 3d 06 00 00 00      lea    0x6(%rip),%rdi      # 401153 <str>
 40114d:      e8 de fe ff ff          callq  401030 <puts@plt>
 401152:      c3                    retq
0000000000401153 <str>:
 401153:      73 74 72 69 6e 67 00
```



# Загрузка программы в память

- Программа – единое целое, взаимное расположение кода внутри секций и секций друг относительно друга не изменяется
  - Смещения в относительных переходах и работе с памятью настраиваются компоновщиком и при загрузке в память не изменяются
- При компоновке фиксируется адрес, по которому программа должна размещаться в памяти, абсолютные адреса в программе настраиваются относительно него
  - ELF Linux x86 по умолчанию: 0x804800
  - Можно изменить с помощью -Wl,-Ttext-segment=ADDR

# Позиционная зависимость

- Если программа неработоспособна при загрузке по адресу, отличному от прописанного в исполняемом файле – программа **позиционно зависима**
- **Позиционно-независимый код** (PIC – position independent code) – сохраняет работоспособность при загрузке с любого адреса в памяти
- Полезен:
  - В динамических библиотеках
  - Динамическая кодогенерация

# Исполняемый файл (executable)

- Файл, основное содержимое (payload) которого – программа для процессора
- После загрузки в память и первоначальной настройки исполняется непосредственно процессором, не требует программы-”интерпретатора”
- Тексты программ на python, bash; .jar-файлы (байт-код java); .net MSIL – не будем считать исполняемыми файлами

# Native executable files

- У каждой операционной системы свой формат исполняемых файлов:
  - Windows: PE (portable executable)
  - MacOS, iOS: Mach-O (Mach Object)
  - Unix-like (Linux, \*BSD, Solaris): ELF (executable and linkable format)
- Один и тот же формат может использоваться для разных процессоров
- Исполняемый файл специфичен для пары процессор-ОС (платформа)

# Portability

- Операционные системы могут поддерживать “чужие” исполняемые файлы, если хост-система работает на том же процессоре.
  - FreeBSD поддерживает возможность запуска ELF-файлов для Linux той же процессорной архитектуры
  - Wine – поддержка запуска Windows-приложений на Unix-like системах

# Инструменты работы

- Binutils – набор утилит для работы с исполняемыми файлами:
  - Objdump – отобразить содержимое:
    - objdump -d FILE – дизассемблировать
    - objdump -x FILE – вывести всю служебную инфу
    - objdump -j SECTION -s FILE – вывести содержимое секции SECTION в файле FILE
  - Readelf – анализ ELF-файла
  - Nm - список “символов” (symbols, т.е. именованных сущностей) в файле
- GDB, IDA PRO, OllyDBG - отладчики

# ELF

- ELF – Executable and Linkable Format
- Применяется на многих Unix-like OS (Linux, \*BSD, Solaris, ...)
- Для всех процессоров, на которых эти ОС работают
- Формат: объектных файлов (.o), статических библиотек (.a), динамических библиотек (.so), исполняемых файлов, дампов памяти, модулей ядра ОС (.ko)...

# Структура ELF-файла

- Заголовок ELF-файла (magic bytes и структура `Elf32_Ehdr`)
- Таблица заголовков программы (program headers) – структура `Elf32_Phdr`
- Payload – содержимое файла
- Таблица заголовков секций (section headers) – структура `Elf32_Shdr`



# (почти) минимальный ELF

- Заголовок файла – 52 байта
- 1 запись program header по 32 байта - 32
- Исполняемый код – 7
- Таблица имен секций – 17
- 3 записи секций по 40 байт – 120 байт
- Итог: 228 байт.
- Задание – получить такой ELF-файл

# Program Headers

- Информация о том, как исполняемый файл должен грузиться в память
- LOAD – заданная область файла должна быть загружена в память
- LOAD требует выравнивания (align) в 0x1000 (4096 байт) – это размер страницы x86/x64 (рассмотрим далее)
- LOAD с флагами 'R E' – это сегмент кода
- LOAD с флагами 'RW' – это сегмент данных
- Записи Program Headers могут описывать накладывающиеся адреса

# Section Headers

- Секция – это структурная единица в исполняемом файле
- Секции группируются в сегменты
- Типичные секции:
  - .text, .data, .bss, .rodata
  - .plt – procedure linkage table
  - .got – global offset table
  - .got.plt – global offset table for PLT

# Динамическая компоновка

- Как правило (по умолчанию) программа, загружаемая на выполнение, не содержит в своем файле кода используемых библиотек
- Перед запуском программы требуется подгрузить динамические библиотеки
- Программа-загрузчик (интерпретатор) прописывается в секции `.interp` файла (для x86 - `/lib/ld-linux.so.2`)
- Операционная система загружает исполняемый файл, загружает требуемый загрузчик и запускает его
- Загрузчик выполняет подгрузку библиотек и передает управление программе

# Динамические символы

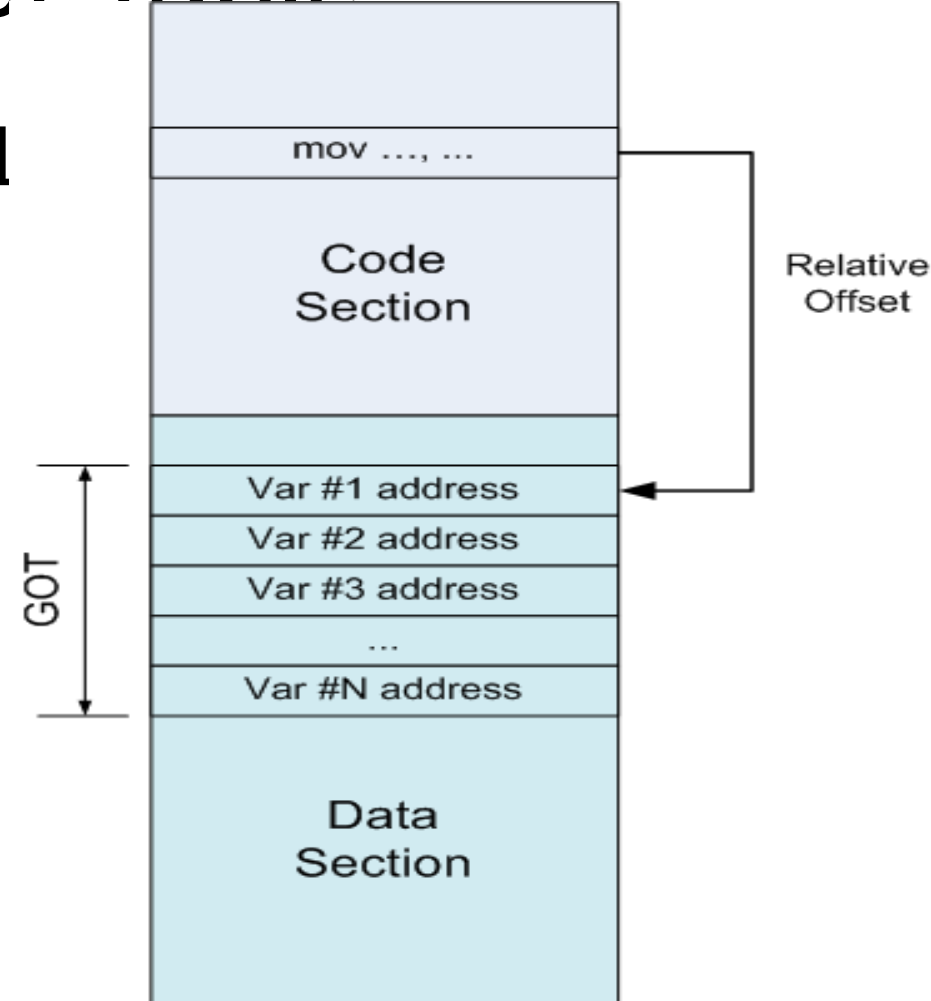
- Секция `.dynsym` содержит таблицу динамических символов, то есть имен, требуемых из данного файла или предоставляемых данным файлом
- Секция `.rel.dyn` содержит таблицу перемещений для переменных, т. е. адреса ячеек памяти, которые должны быть изменены
- Секция `.rel.plt` содержит таблицу перемещений для внешних функций

# Global Offset Table

- Чтобы не модифицировать код программы, все перемещения находятся в специальных секциях
  - .got – для переменных
  - .got.plt – для функций
- После работы загрузчика таблица .got содержит абсолютные адреса переменных

# Global Offset Table

- `_GLOBAL_OFFSET_TABLE`
- `L@GOT`



# Использование GOT

- `_GLOBAL_OFFSET_TABLE_` - это смещение относительно **адреса текущей инструкции** до адреса GOT
- `LABEL@GOT` - это смещение относительно GOT до ячейки памяти, в которой хранится правильный адрес, по которому размещается LABEL



# Procedure Linkage Table (PLT)

- В динамически скомпонованных программах часть GOT отводится под хранение адресов функций из динамических библиотек
- Если `printf` – функция из динамической библиотеки, то `call printf` заменяется на `call printf@plt`
- `printf@plt` - специальная функция (трамплин) для передачи управления в библиотеку

# Procedure linkage table (PLT)

080483f0 <dynlink>:

80483f0:	ff 35 04 a0 04 08	pushl	0x804a004
80483f6:	ff 25 08 a0 04 08	jmp	*0x804a008
80483fc:	00 00	add	%al, (%eax)

08048410 <printf@plt>:

8048410:	ff 25 10 a0 04 08	jmp	*0x804a010
8048416:	68 08 00 00 00	push	\$0x8
804841b:	e9 d0 ff ff ff	jmp	80483f0 <dynlink>

0804a000 <\_GLOBAL\_OFFSET\_TABLE\_>:

804a000:	14 9f 04 08	.int	0x8049f14 <_DYNAMIC>
804a004:	00 00 00 00	.int	0
804a008:	00 00 00 00	.int	0
804a00c:	06 84 04 08	.int	0x8048406
804a010:	16 84 04 08	.int	0x8048416

# Lazy binding

- При первом вызове `<printf@plt>` управление попадет в динамический загрузчик. В стеке будет передано смещение на дескриптор загружаемой функции
- Динамический загрузчик запишет в GOT адрес функции `printf` в загруженной динамической библиотеке
- Все последующие вызовы будут передавать управление сразу на `printf` в динамической библиотеке

# Зачем нужны трамплины

- Компилятор и ассемблер не имеют информацию, в каком режиме будет компоноваться программа
  - они всегда генерируют инструкцию CALL FUNC для вызова функции FUNC
  - Компоновщик (linker) при необходимости заменит CALL FUNC на CALL FUNC@PLT и сгенерирует трамплин FUNC@PLT
- Если в Си-коде берется адрес функции, например, &printf, то компоновщик заменяет на &printf@plt без модификации машинного кода
- Позволяют реализовывать ленивое связывание (lazy binding)

# ССЫЛКИ

- <http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries>
- <https://www.cs.stevens.edu/~jschauma/810/elf.html>
- <https://ejudge.ru/study/3sem/elf.html>
-