Лекция 13

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

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

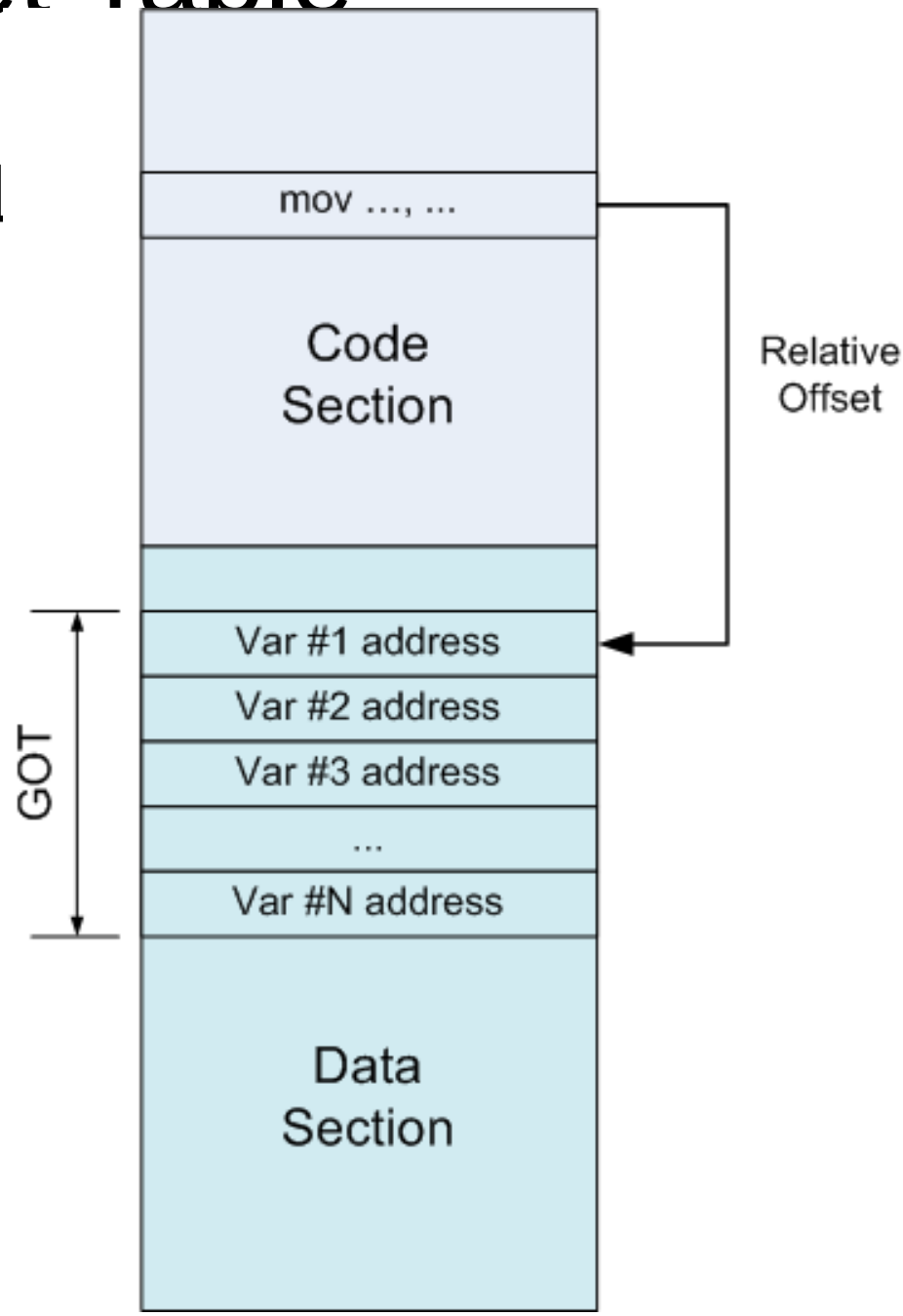
- Если программа неработоспособна при загрузке по адресу, отличному от прописанного в исполняемом файле – программа **позиционно зависима**
- **Позиционно-независимый код** (PIC – position independent code) – сохраняет работоспособность при загрузке с любого адреса в памяти
- Полезен:
  - В динамических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

Global Offset Table (GOT)

- Секция .got в исполняемом файле содержит абсолютные адреса переменных
- При загрузке программы на выполнение загрузчик корректирует адреса в таблице GOT, чтобы они соответствовали актуальному размещению программы в памяти

Global Offset Table

- _GLOBAL_OFFSET_TABL
- 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 (stub) 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

Ссылки

- http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries

Изоляция процессов

- Операционная система изолирует процессы друг от друга и от аппаратуры компьютера
- Виртуальная память – адресное пространство каждого процесса изолировано
- Процесс работает в **пользовательском режиме** (user mode) и не может выполнять “чувствительные” инструкции (настройка MMU, работа с внешними устройствами...)

Ядро ОС

- Ключевая компонента ОС
- Работает все время работы компьютера от загрузки до выключения
- Работает в **привилегированном режиме** (privileged mode или kernel mode)
- Управляет внешними устройствами, распределяет ресурсы между процессами

Системный вызов

- Процесс в пользовательском режиме не может выполнить ввода-вывода (нет прав)
- Для выполнения ввода-вывода процесс вызывает ядро ОС
- Ядро ОС от имени и с проверкой прав процесса выполняет запрошенную операцию
- Управление возвращается в процесс, он продолжает работу в режиме пользователя
- Вызов ядра – **системный вызов**

Системный вызов

- Все системные вызовы ядра занумерованы
- На Linux:
  - #include <asm/unistd_32.h> - для x86
  - #include <asm/unistd_64.h> - для x64
- На Linux/x86
  - Номер системного вызова передается в %eax
  - Параметры вызова в %ebx, %ecx, %edx, %esi, %edi, %ebp
  - Системный вызов: int $0x80
  - Результат возвращается в %eax

Системные вызовы

- Системные вызовы задокументированы в терминах языка C:
- Документация: man 2 SYSCALL, например
  man 2 write
- Исключение: системный вызов exit задокументирован как _exit
- Стандартная библиотека libc содержит вспомогательные функции (“мосты”)
  - Удовлетворяют стандартным соглашениям о вызовах
  - Обеспечивают подготовку параметров и выполнение системного вызова
- Такие функции стандартной библиотеки для удобства тоже называются системными вызовами

Некоторые системные вызовы

- Чтение
  ssize_t read(int fd, void *buf, size_t count);
  пока для наших целей fd == 0 – стандартный поток ввода
- Запись
  ssize_t write(int fd, const void *buf, size_t cnt);
  fd == 1 – запись на стандартный поток вывода
- Завершение работы:
  void _exit(int status);

Системные вызовы

- Системные вызовы предоставляют минимально необходимый интерфейс, те операции, которые невозможно или неэффективно выполнять без вызова ядра
- Предоставление удобного интерфейса – задача библиотек, работающих в пользовательском режиме
- Библиотечные функции для выполнения ввода-вывода используют системные вызовы
- Например: write и printf или std::cout

Область видимости vs время жизни

- Область видимости переменной 'x' – точки в исходном тексте программы, в которых идентификатор 'x' обозначает эту переменную
  - Вложенная область видимости может перекрыть переменную из объемлющей области видимости
- Время жизни переменной – промежутки времени во время выполнения программы, в которые под эту переменную выделена память

Классы памяти

- Глобальные переменные
  - Существуют от момента запуска программы на выполнение до завершения работы
  - Память под глобальные переменные резервируется в исполняемом файле
- Локальные переменные
  - Существуют от момента входа в блок до момента выхода из блока
  - Память резервируется на стеке или в регистрах: стековый фрейм создается при входе в функцию и уничтожается при выходе
  - Обращения к локальным переменным транслируются в обращения к текущему стековому фрейму относительно регистра фрейма

Управление динамической памятью (кучей)

- Обеспечить работу функций malloc, calloc, free, realloc, new, new[], delete, delete[]
- Память может запрашиваться фрагментами произвольного размера
- Память может освобождаться в произвольный момент времени
- Стандартные стратегии обслуживания (стек, очередь) неприменимы

Управление кучей

- Память может запрашиваться и освобождаться в нескольких нитях одновременно
- К структурам данных предъявляются разные требования по выравниванию, поэтому данные выравниваются по максимально жесткому требованию (4 байта – x86, 16 байт - x64)
-
- **Выделение памяти в куче намного медленнее, чем в стеке!**

Проблемы динамической памяти

- Необходимость блокировки в многонитевых программах
- Постепенное дробление больших непрерывных фрагментов памяти на маленькие
- Постепенная фрагментация динамической памяти
- Постепенный рост размера виртуального адресного пространства и невозможность возврата памяти ядру ОС

Фрагментация

p1 = malloc(4); p2 = malloc(4);

free(p1); p3 = malloc(2); p4 = malloc(1);

free(p2); p5 = malloc(2); p6 = malloc(1);

free(p3); free(p5);

Управление адресным пространством процесса

- Системный вызов sbrk() - изменить адрес конца сегмента данных

```
void  *sbrk(intptr_t increment);
```

- Сразу после загрузки исполняемого образа break addess — это конец сегмента данных
- sbrk возвращает предыдущее значение

Запрос памяти у ядра

- Когда при очередном вызове функции выделения памяти запрос не может быть удовлетворен, с помощью sbrk запрашивается порция памяти у ядра
- Как правило, память не возвращается ядру, даже если это возможно

Стратегии распределения

- Битовый массив блоков
- Списки свободных/занятых блоков

Битовый массив блоков

- Память разбивается на блоки выделения фиксированного размера (например, 16 байт)
- Каждому блоку выделения ставится в соответствие 1 бит в битовом массиве: 0 — блок свободен, 1 - занят

1	1	1	1	1	0	0	1	1	1	1	0	0	0

80 байт занято	32 св.	64 байта	48 байт

Списки блоков

- Вариант с одиним списком:
  - В памяти поддерживается двусвязный список выделенных/свободных блоков
  - Структура дескриптора блока:

```
struct memdesc {
    struct memdesc *prev;
    size_t size;
    unsigned char data[0];
};
```

  - Флаг свободный/занятый - младший бит size

Выделение памяти malloc

- Размер выделения округляется вверх до размера выравнивания (8 байт), 0 байт → 8
- В памяти ищется подходящий свободный блок
- Если свободного блока нет, запрашивается блок памяти с помощью sbrk() и помечается как свободный
- Найденный блок при необходимости дробится, формируется дескриптор блока памяти и возвращается указатель на поле data

Освобождение памяти free

- Из переданного указателя ptr вычитается 8, таким образом получаем указатель на дескриптор блока памяти
- Блок памяти помечается как свободный, при необходимости сливается с непосредственно предшествующим и/или следующим свободным блоком

Алгоритмы выделения блоков

- Первый подходящий
- Самый подходящий
- Быстрый подходящий: поддерживаются список свободных блоков наиболее часто запрашиваемых размеров
-
- Существует много различных алгоритмов управления динамической памятью для разных ситуаций, нет однозначно наилучшего

Ошибки работы с динамической памятью

- Типичные ошибки:
  - Memory overrun (выход за положительную границу)
  - Memory underun (выход за 0)
  - Use after free
  - Double free
  - Free of non-allocated pointer
- Приводят к порче списков свободных блоков и падению программы в какой-то момент позже
- Программа valgrind — отладчик работы с динамической памятью