

Лекция 11

Принцип адресуемой памяти

- Принцип фон-Неймана: все ячейки памяти пронумерованы от 0 до N, число-номер ячейки - “адрес ячейки”.
- Множество (теоретически возможных) адресов – адресное пространство
- Принцип фон-Неймана – одно адресное пространство для программы и ее данных
- Гарвардская архитектура – несколько адресных пространств:
 - Пространство адресов кода программы
 - Пространство адресов данных программы

Изоляция процессов

- Процесс – программа в состоянии исполнения
- Как правило, операционные системы изолируют процессы друг от друга
- Каждый процесс работает в своем - **виртуальном** адресном пространстве
- Операционная система (ядро) распределяет физическую память (**физическое** адресное пространство) между процессами
- Преобразование виртуальных адресов в физические выполняется в MMU (memory management unit) процессора

Виртуальное адресное пространство

- В дальнейшем предполагаем, что программа работает в виртуальном адресном пространстве
- Для x86 виртуальное адресное пространство – 32-битное ($2^{32} = 4\text{GiB}$ адресов)
- Для x64 виртуальное адресное пространство – 48-битное
- Каждый байт в памяти адресуется индивидуально
- Порядок байт в слове для x86/x64 – Little Endian

Метод адресации памяти

- Методы адресации – способы получения адреса операндов инструкции в памяти
- Общий вид обращения к памяти:
OFFSET(BREG, IREG, SCALE)
адрес вычисляется по формуле:
$$\text{BREG} + \text{OFFSET} + \text{IREG} * \text{SCALE}$$
- BREG – базовый регистр (общего назн.)
- IREG – индексный регистр (общего назн.)
- SCALE – {1, 2, 4, 8}, по умолчанию 1
- OFFSET – базовый адрес в памяти или смещение

Обращения к памяти

- Примеры:

`(%eax)` // адрес находится в `%eax`

`16(%esi)` // адрес равен `%esi + 16`

`array(,%eax)` // адрес равен `array + %eax`

`array(,%eax,4)` // адрес равен `array + %eax*4`

`(%ebx,%eax,2)` // адрес: `%ebx + %eax * 2`

`-4(%ebx,%eax,8)` // адрес: `%ebx-4+%eax*8`

Примеры использования

- Разыменование указателя

```
char *p; int c = *p;
```

(если p загружен в %eax)

```
movsbl (%eax), %eax // в %eax будет c
```

- Доступ к глобальному массиву

```
unsigned short array[N]; int x = array[i];
```

(если i загружено в %esi)

```
movzwl array(,%esi,2), %eax // результат в %eax
```

Примеры использования

- Доступ к массиву

```
int *p; int i;
```

```
x = p[i];
```

// пусть p находится в %ebx, i в %esi

```
movl    (%ebx, %esi, 4), %eax // x – в %eax
```

- Если размер элемента массива не 1, 2, 4, 8, потребуется операция умножения или несколько сложений и сдвигов

Примеры использования

- Доступ к полю структуры

```
struct Str { int f1; int f2; };
```

```
struct Str *p;
```

```
int x = p->f2;
```

// пусть p находится в %ebx

```
movl    4(%ebx), %eax // x в %eax
```

- Любой доступ к памяти может быть представлен как комбинация разыменовывания, доступа к элементу массива, доступа к полю структуры

Инструкция lea

- Вместо обращения к памяти и сохранения в регистре значения из памяти в регистре сохраняется адрес

```
leal (%eax, %eax, 8), %eax  
// %eax = %eax * 9
```

Типизация в ассемблере

- В ассемблере целое число (32 бит) может быть:
 - Знаковым целым числом
 - Беззнаковым целым числом
 - Указателем любого типа
- Тип никак не привязан к ячейке/регистру, в котором хранится число
- Интерпретация числа зависит от выполняющейся инструкции

Структура адресного пространства

- Код программы и данные, загружаемые из исполняемого файла (образ программы)
 - Содержит разные секции исполняемого кода, в том числе `.text`, `.data`, `.bss`
- Основной стек процесса
- Область динамической памяти (куча)

Использование стека

- Область стека размещается “вверху” доступного адресного пространства
- Стек растет “вниз”
- На x86 в стек можно сохранять только 32-битные значения
- `%esp` – указывает на первый занятый элемент
- `push %eax` – можно условно расписать как
`subl $4, %esp`
`movl %eax, (%esp)`
- `pop %eax` – можно условно расписать как
`movl (%esp), %eax`
`addl $4, %esp`

Вызов подпрограмм

- `call sub` // вызов подпрограммы:
 `push %eip`
 `mov $sub, %eip`
 // это условный код, точнее будет
 `add $(sub-%eip), %eip`
- `ret` // возврат из подпрограммы
 `pop %eip`
- В стеке накапливаются адреса возврата для вызываемых подпрограмм
- Вложенные и рекурсивные вызовы, пока хватает стека

Сохранение регистров

- Регистров немного, они очень нужны в программах
- По типу использования регистры делятся на:
 - 1)Для передачи параметров
 - 2)Для возврата значения
 - 3)“Рабочие” (scratch) регистры
 - 4)Сохраняемые (callee-saved) регистры

Вызывающий код

call subroutine

- После возврата из подпрограммы 'subroutine' регистры
 - 2) содержат возвращенное значение
 - Значение регистров 1) не определено
 - Значение регистров 3) не определено
 - Значение регистров 4) сохраняется
- Если подпрограмма использует регистры 4), они должны быть сохранены в начале и восстановлены перед возвратом из нее

Регистры на x86

- Возвращаемое значение хранится в %eax или в %eax:%edx (если 64 бита, %eax – младшая половина, %edx – старшая)
- (если не используются для возврата значения) %eax, %ecx, %edx – рабочие (scratch) регистры
- %ebx, %esi, %edi, %ebp – сохраняемые регистры

Пример:

- Сохранение регистров:
push %ebp
push %ebx
push %esi
push %edi
- Восстановление регистров:
pop %edi
pop %esi
pop %ebx
pop %ebp
- Не обязательно сохранять/восстанавливать все регистры, достаточно только те, которые будут использоваться

Передача параметров

- Один из возможных вариантов: через стек
- Если размер меньше 32 бита, преобразовывается к int
pushl \$'\n'
call putchar
- 64-битные значения передаются так, чтобы в памяти хранились как LE-значения
pushl \$0
pushl \$1
// сохранили в стек число 1LL

Очистка стека после возврата

- Тот код, который вызвал подпрограмму, должен очистить стек после возврата из этой подпрограммы

```
pushl    '$a'  
call     putchar  
add      $4, %esp
```

- Если забыть почистить стек, целостность стека будет нарушена – программа скорее всего упадет

Передача нескольких параметров

- Параметры заносятся в стек в обратном порядке, то есть в стеке они размещаются в прямом порядке

```
push    $10  
push    $str  
call    printf  
add     $8, %esp
```

...

```
str: .asciz "%d\n"
```

Соглашение о вызовах

- Соглашение о вызовах (calling convention) – правила взаимодействия подпрограмм по вызовам
 - Правила использования регистров процессора
 - Регистры, используемые для возврата значения
 - Регистры, используемые для передачи параметров
 - Рабочие регистры
 - Сохраняемые регистры
 - Правила использования стека процессора
 - Порядок занесения аргументов в стек
 - Порядок очистки стека
 - Требования на выравнивание регистра указателя стека
 - Как передаются и возвращаются структуры

Calling convention на x86

- Для x86 (по историческим причинам) существует более 10 разных CC
- Стандартное соглашение на Linux (cdecl):
 - %eax или %eax:%edx для возврата значения
 - %eax, %ecx, %edx – scratch
 - %ebx, %esi, %edi, %ebp – callee-saved
 - Параметры передаются через стек
 - Параметры заносятся в обратном порядке
 - Стек очищается тем, кто вызвал (caller-cleaned)

Необходимость Calling Convention

- Ключевой элемент ABI для обеспечения совместимости бинарных компонент системы
- Следование Calling Conventions необходимо для вызова подпрограмм стандартных библиотек и для того, чтобы подпрограммы могли быть вызваны из стандартных библиотек

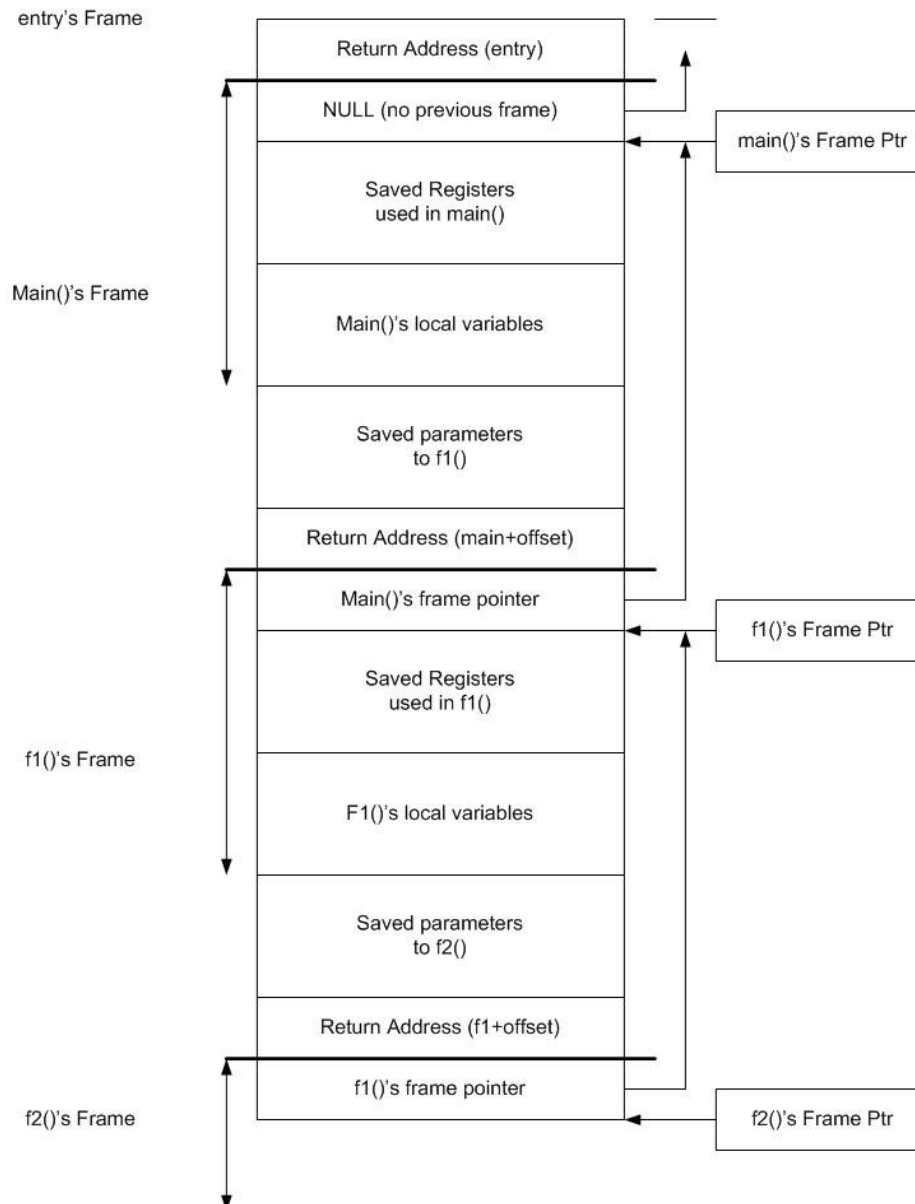
Стековый кадр (stack frame)

- Организуют блоки данных каждой из подпрограмм на стеке в виде списка
- Позволяют получить всю цепочку вызовов от текущей точки и до подпрограммы самого внешнего уровня
- Необходимы:
 - Когда нужно проходить по цепочке вызовов (C++ exception handling)
 - Для отладки
 - При выделении памяти заранее неизвестного размера на стеке

Организация стекового кадра

- Регистр `%ebp` хранит адрес стекового кадра текущей подпрограммы
- Стандартный пролог (prologue)
`pushl %ebp`
`movl %esp, %ebp`
- Стандартный эпилог
`popl %ebp`
`ret`
- Так: `(%ebp)` – это адрес стекового кадра предыдущей подпрограммы, `((%ebp))` – пред-предыдущей...
- Самый внешний стековый кадр хранит 0

Цепочки ВЫЗОВОВ



Использование стекового кадра

- Для доступа к параметрам подпрограммы используются положительные смещения относительно `%ebp`:
`movl 8(%ebp), %eax` // доступ к 1-му параметру.
- Ниже `%ebp` хранятся сохраненные регистры и область под локальные переменные

Локальные переменные

- Выделение памяти:

```
pushl    %ebp
```

```
movl     %esp, %ebp
```

```
subl     $16, %esp
```

// выделено 16 байт под лок. Переменные

- Освобождение:

```
movl     %ebp, %esp
```

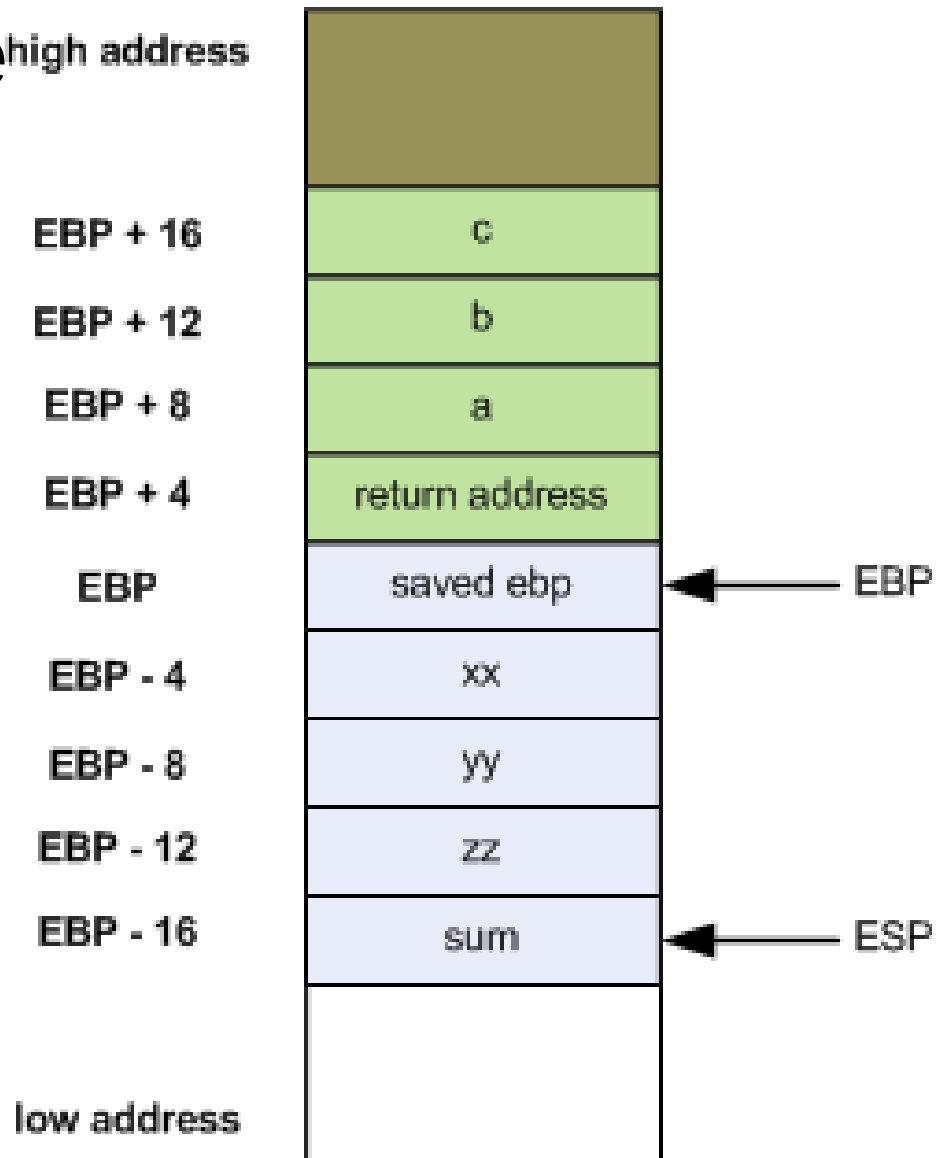
```
popl     %ebp
```

```
ret
```

Локальные переменные

- Локальные переменные размещаются «ниже» адреса возврата

-
- $-4(\%ebp)$ // xx
- $-16(\%ebp)$ // sum
- $\%ebp - \$16$ // &sum



Выделение памяти на стеке

- В Си есть функция `void *alloca(size_t sz);`
- Массивы переменного размера
- Достоинства:
 - Очень быстрое выделение (1 инструкция)
 - Автоматическое освобождение
- Недостатки:
 - Нельзя контролировать время жизни (освобождается автоматически при выходе)
 - Сложно контролировать нехватку памяти
 - Стек, как правило, имеет ограниченный размер