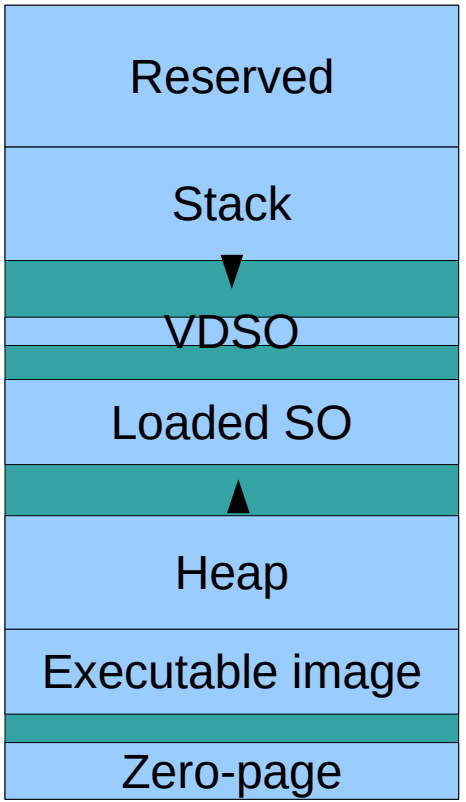
Ассемблер x86/x64
Лекция 16

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)

Адресное пространство процесса



- Нулевая страница — защита от обращений по указателю NULL
- Стек расширяется вниз автоматически
- Куча растет вверх по запросу
- Текущее состояние карты памяти: /proc/${PID}/maps

Стек как структура данных

- “Обычный” стек (растет вверх):
  int s[SIZE]; int sp = 0; // первый свободный элемент
  push: s[sp++] = value;
  pop : value = s[--sp];
- Процессорный стек (растет вниз)
  int s[SIZE]; int sp = SIZE; // первый занятый элемент
  push: s[--sp] = value;
  pop: value = s[sp++];

Использование стека

- Область стека размещается “в верху” 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)

Выравнивание стека

- Linux x86 не требует но рекомендует, а MacOS требует **выравнивания стека** по 16 байтам
- При вызове подпрограммы первый аргумент должен находиться по адресу, кратному 16
  XXXXXXX0 – (последняя 16-ричная цифра 0)
- Адрес возврата: XXXXXXXC
- Сохраненный EBP: XXXXXXX8
- Если выравнивание стека неизвестно:
  and $-16, %esp
  смещает ESP вниз на правильную границу

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

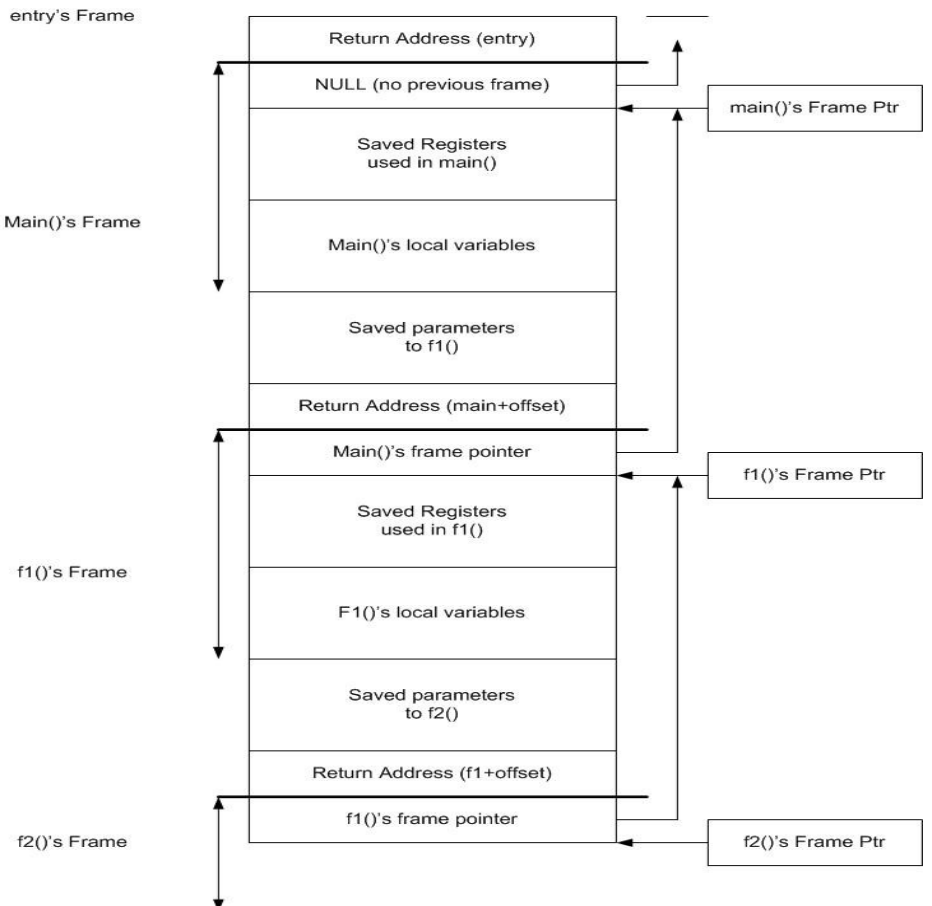
Стековый кадр (stack frame)

- 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
  ```
- Стандартый 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

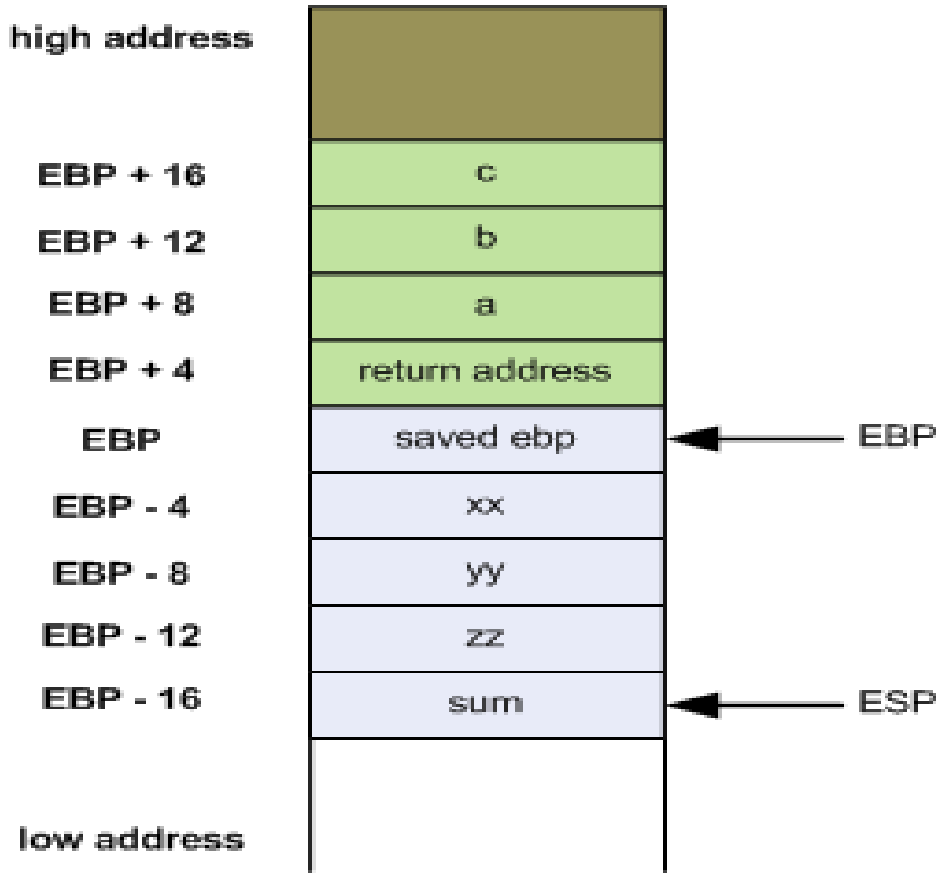
- Для доступа к параметрам подпрограммы используются положительные смещения относительно %ebp:
  movl 8(%ebp), %eax // доступ к 1-му парам.
- Ниже %ebp хранятся сохраненные регистры и область под локальные переменные

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

Unix x64 calling convention

- Передача параметров: сначала rdi, rsi, rdx, rcx, r8, r9 для целых/указателей и xmm0-xmm7 для floating-point, затем стек
- Возврат значения: rax, rax:rdx, xmm0 для floating point
- Callee-saved registers: rbx, rbp, rsp, r12-r15
- Scratch registers: rax, rcx, rdx, rsi, rdi, r8 – r11
- Stack alignment: 16
- Специальная “красная зона” - 128 байт ниже (rsp)
- В функциях с переменным числом арг. Rax содержит число floating-point аргументов

Работа с 64-битными целыми

- 64-битные целые требуют по несколько инструкций для обработки
  - 64-битное значение в паре регистров (напр. %eax и %edx)
  - Логические операции – отдельно для младшей и старшей половины
  - Сложение: ADD для младшей половины, ADC для старшей
  - Вычитание: SUB для младшей половины, SBB для старшей
  - Умножение, деление: вспомогательные функции (находятся в libgcc или аналогичной библиотеке)

Сдвиги 64-битных чисел

- Можно сдвигать по одному биту и использовать CF:
  shl 1, low
  rol 1, high
- Специальные инструкции shld/shrd для старшей части:
  shld   low, count, high-dest
- Пусть число в %eax:%edx, тогда сдвиг на 7:
  shld %eax, 7, %edx
  shl   7, %eax