????????? ????????? ? ??????

Интересный учебник

Еще интересный ресурс

Типы памяти

Регистровая память Самый быстрый способ хранения данных. Процессоры имеют набор регистров, которые могут использоваться для хранения данных.

Оперативная память (RAM) Это основное место хранения данных программы. В отличие от регистров, доступ к памяти менее быстрый, но объём значительно больше. Оперативная память делится на несколько сегментов:

Данные (Data)
Стек (Stack)
Код (Code)
В Assembler можно использовать сегментирование для организации памяти.

Стек
Стек — структура данных, в которой операции записи и чтения выполняются по принципу “последним пришёл — первым ушёл” (LIFO). Стек используется для хранения локальных переменных, адресов возврата при вызове функций, а также для управления процессом вызова процедур.

Куча
Куча (heap) — это область памяти, в которой динамически выделяются блоки памяти во время выполнения программы. Работа с кучей требует явного управления памятью (например, с помощью системных вызовов операционной системы).

Структура программы

```
section .data
msg db "hello, world", 0
section .bss
section .text
global main
main:
```

```
mov rax, 1
mov rdi, 1
mov rsi, msg
mov rdx, 12
syscall
mov rax, 60
mov rdi, 0
syscall
```

Страницы:

- .data
- .bss
- .txt

.data

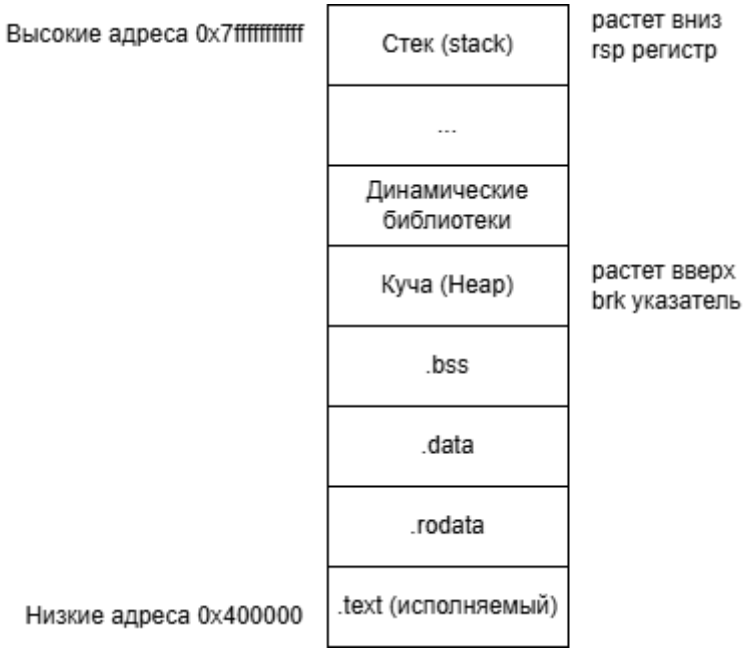
Переменные: <varname> <type> <value>
Константы: <constant_name> equ <value>

.bss

Неинициализированные данные. resb - байт, resw - слово, resd - двойное слово, resq - двойное длинное слово

.txt Программа.

Карта памяти:



Адреса в памяти:

```
section .data
    code_msg    db "Code (.text):   0x%lx", 10, 0
    data_msg    db "Data (.data):   0x%lx", 10, 0
    heap_msg    db "Heap (break):   0x%lx", 10, 0
    stack_msg   db "Stack (rsp):    0x%lx", 10, 0
    diff_msg    db "Stack - Heap:   %ld bytes", 10, 0

section .text
    global main
    extern printf

main:
    push rbp
    mov rbp, rsp

    ; Адрес кода (самой функции main)
    lea rax, [rel main]
    mov rdi, code_msg
    mov rsi, rax
    call printf

    ; Адрес данных
    mov rdi, data_msg
    mov rsi, code_msg        ; любая метка из .data
    call printf

    ; Адрес кучи (break)
    mov rax, 12              ; sys_brk
    mov rdi, 0
    syscall
    mov rdi, heap_msg
    mov rsi, rax
    call printf

    ; Адрес стека (rsp)
    mov rdi, stack_msg
    mov rsi, rsp
    call printf

    ; Разница между стеком и кучей
```

```
    mov rax, 12             ; снова получаем break
    mov rdi, 0
    syscall
    mov rbx, rax            ; heap в rbx
    mov rax, rsp            ; stack в rax
    sub rax, rbx            ; stack - heap

    mov rdi, diff_msg
    mov rsi, rax
    call printf

    pop rbp
    mov rax, 0
    ret
```

Примерный результат выполнения:

Code (.text): 0x400500
Data (.data): 0x600800
Heap (break): 0x1ae20000
Stack (rsp): 0x7fffffffe010
Stack - Heap: 2147358720 bytes (примерно 2GB)

Выделение и освобождение стека:

```
section .text
    global main

main:
    push rbp
    mov rbp, rsp

    ; Текущий rsp
    mov r12, rsp

    ; Выделяем 1KB в стеке
    sub rsp, 1024
    ; rsp УМЕНЬШИЛСЯ!
    ; r12 (старый rsp) > rsp (новый rsp)
    ; Восстанавливаем
    mov rsp, rbp
```

```
    pop rbp
    ret
```

Выделение и освобождение кучи:

```
section .text
    global main

main:
    ; Текущий break
    mov rax, 12
    mov rdi, 0
    syscall
    mov r12, rax             ; сохраняем старый break

    ; Увеличиваем кучу на 1KB
    mov rax, 12
    mov rdi, r12
    add rdi, 1024
    syscall
    mov r13, rax             ; новый break

    ; r13 > r12 - куча ВЫРОСЛА!

    ret
```

Если куча и стек встречаются, то SEGFAULT или ENOMEM при попытке выделить память.
Предотвращение столкновений:

```
# Посмотреть текущие лимиты
ulimit -a

# Ограничить стек для теста
ulimit -s 1024  # 1MB стек
./program

# Ограничить кучу
ulimit -d 65536  # 64MB куча

# Просмотр карты памяти для процесса
```

```
cat /proc/.../maps
```

Проверка в коде:

```
safe_stack_allocation:
    mov rax, rsp
    sub rax, desired_size
    cmp rax, [heap_upper_bound]
    jl .stack_heap_collision
    ; Иначе безопасно
    sub rsp, desired_size
    ret

.stack_heap_collision:
    ; Обработка нехватки памяти
    mov rax, -1
    ret
```

Динамическое выделение памяти

Выделяется через системные вызовы или стандартные библиотеки (например, malloc из libc). При старте программы выделяется куча только под текущие потребности приложения. Затем при наличии динамических элементов размер увеличивается.

Выделить дополнительную память можно через увеличение кучи и через выделение сегмента.

Работа с кучей.

Получение текущего адреса кучи (heap):

```
Для x32:
mov eax, 45
mov ebx, 0
int 0x80        ; В eax вернётся текущий адрес конца кучи

Для x64:
mov rax, 12
mov rdi, 0
syscall         ; В rax вернётся текущий адрес конца кучи
```

Кучу можно использовать через sys_brk Базовое использование sys_brk:

```
section .data
    success_msg db "Heap: allocated %d bytes at address 0x%lx", 10, 0
    error_msg db "Heap: allocation failed!", 10, 0

section .bss
    initial_break resq 1
    current_break resq 1

section .text
    global main
    extern printf

main:
    push rbp
    mov rbp, rsp

    ; 1. Получаем текущую границу кучи
    mov rax, 12                 ; sys_brk
    mov rdi, 0
    syscall
    mov [initial_break], rax
    mov [current_break], rax

    ; 2. Выделяем 16KB памяти
    mov rdi, rax
    add rdi, 16384              ; 16 * 1024 = 16384 байт
    mov rax, 12                 ; sys_brk
    syscall

    ; 3. Проверяем успешность
    cmp rax, [current_break]
    jle .error

    mov [current_break], rax    ; сохраняем новую границу

    ; 4. Используем выделенную память
    mov rdi, [initial_break]    ; начало выделенной области

    ; Записываем некоторые данные
    mov byte [rdi], 'H'
```

```
    mov byte [rdi + 1], 'e'
    mov byte [rdi + 2], 'l'
    mov byte [rdi + 3], 'l'
    mov byte [rdi + 4], 'o'
    mov byte [rdi + 5], 0

    ; Выводим информацию
    mov rdi, success_msg
    mov rsi, 16384              ; размер
    mov rdx, [initial_break]    ; адрес
    mov rax, 0
    call printf

    jmp .exit

.error:
    mov rdi, error_msg
    mov rax, 0
    call printf

.exit:
    pop rbp
    mov rax, 0
    ret
```

Ключевые принципы

- Всегда проверяйте успешность выделения памяти
- Выравнивайте запросы по границам страниц (4KB)
- Отслеживайте использование чтобы вовремя увеличивать кучу
- Используйте умные стратегии выделения (пулы, slab-аллокаторы)

Работа с сегментами памяти.

```
section .data
    msg db "Hello from allocated memory in x64!", 10
    msg_len equ $ - msg

section .bss
    allocated_ptr resq 1  ; 64-битный указатель
```

```
section .text
global _start

_start:
    ; Выделяем 2 страницы памяти с помощью mmap
    mov rax, 9          ; sys_mmap
    xor rdi, rdi        ; адрес = NULL (ядро выбирает)
    mov rsi, 8192       ; размер: 2 страницы (8192 байта)
    mov rdx, 0x7        ; PROT_READ | PROT_WRITE | PROT_EXEC
    mov r10, 0x22       ; MAP_PRIVATE | MAP_ANONYMOUS
    mov r8, -1          ; без файла
    xor r9, r9          ; смещение = 0
    syscall

    test rax, rax
    js error            ; если ошибка (отрицательное значение)

    mov [allocated_ptr], rax

    ; Копируем строку в выделенную память
    mov rdi, rax        ; назначение
    mov rsi, msg        ; источник
    mov rcx, msg_len    ; длина
    rep movsb           ; копируем байты

    ; Выводим строку из выделенной памяти
    mov rax, 1          ; sys_write
    mov rdi, 1          ; stdout
    mov rsi, [allocated_ptr]
    mov rdx, msg_len
    syscall

    ; Освобождаем память
    mov rax, 11         ; sys_munmap
    mov rdi, [allocated_ptr]
    mov rsi, 8192
    syscall

    ; Выход
    mov rax, 60         ; sys_exit
```

```
    xor rdi, rdi        ; код 0
    syscall

error:
    ; Обработка ошибки
    mov rax, 60         ; sys_exit
    mov rdi, 1          ; код ошибки 1
    syscall
```

Работа с сегментами считается сложнее, но эффективнее в контексте выделения и освобождения.

Выполнение кода из кучи. Это называется JIT-компиляция (Just-In-Time) или динамическое генерирование кода.

```
section .data
    success_msg db "Executing code from heap!", 10, 0
    after_exec_msg db "Back from JIT code! Return value: %d", 10, 0

section .bss
    code_buffer resb 4096       ; буфер для кода

section .text
    global main
    extern printf, mprotect

main:
    push rbp
    mov rbp, rsp

    ; 1. Выделяем память под код (уже есть code_buffer)

    ; 2. Записываем машинный код в буфер
    mov rdi, code_buffer

    ; Генерируем простую функцию: mov rax, 42; ret
    mov byte [rdi], 0x48        ; mov rax, 42
    mov byte [rdi + 1], 0xC7
    mov byte [rdi + 2], 0xC0
    mov byte [rdi + 3], 0x2A
    mov byte [rdi + 4], 0x00
```

```
    mov byte [rdi + 5], 0x00
    mov byte [rdi + 6], 0x00
    mov byte [rdi + 7], 0xC3    ; ret

    ; 3. Делаем память исполняемой
    mov rax, 10                 ; sys_mprotect
    mov rdi, code_buffer        ; адрес
    and rdi, ~0xFFF             ; выравниваем до границы страницы
    mov rsi, 4096               ; размер
    mov rdx, 7                  ; PROT_READ|PROT_WRITE|PROT_EXEC
    syscall
    test rax, rax
    jnz .error

    ; 4. Вызываем код из кучи!
    mov rdi, success_msg
    call printf

    mov rax, code_buffer        ; получаем указатель на функцию
    call rax                    ; ВЫЗЫВАЕМ КОД ИЗ КУЧИ!

    ; 5. rax содержит возвращаемое значение (42)
    mov rsi, rax
    mov rdi, after_exec_msg
    call printf

    jmp .exit

.error:
    ; Обработка ошибки
.exit:
    pop rbp
    mov rax, 0
    ret
```

Важные моменты безопасности

1. mprotect обязателен, без PROT_EXEC флага будет SEGFAULT:

```
; Без этого - SEGFAULT!
mov rax, 10                ; sys_mprotect
mov rdi, code_addr
mov rsi, size
mov rdx, 7                 ; PROT_READ|PROT_WRITE|PROT_EXEC
syscall
```

2. W^X политика

В современных системах часто включена политика W^X (Write XOR Execute):

- Память не может быть одновременно записываемой и исполняемой
- Нужно сначала записать код, потом сделать исполняемым

```
section .data
    code_bytes db 0xB8, 0x01, 0x00, 0x00, 0x00, 0xC3  ; mov eax,1; ret

section .text
global _start

_start:
    ; Выделяем память с правами READ|WRITE
    mov rax, 9          ; mmap
    xor rdi, rdi        ; адрес
    mov rsi, 4096       ; размер
    mov rdx, 0x3        ; PROT_READ | PROT_WRITE (но НЕ PROT_EXEC!)
    mov r10, 0x22       ; MAP_PRIVATE | MAP_ANONYMOUS
    mov r8, -1
    xor r9, r9
    syscall
    mov rbx, rax        ; сохраняем адрес

    ; Копируем код в выделенную память
    mov rdi, rax
    mov rsi, code_bytes
    mov rcx, 6
    rep movsb

    ; Пытаемся выполнить код - ЭТО ВЫЗОВЕТ SEGFAULT!
```

```
; call rbx  ; ← segmentation fault!

; Сначала меняем права на READ|EXECUTE
mov rax, 10         ; mprotect
mov rdi, rbx        ; адрес
mov rsi, 4096       ; размер
mov rdx, 0x5        ; PROT_READ | PROT_EXECUTE
syscall

; Теперь можно выполнять
call rbx            ; работает!
```

Revision #21
Created 5 November 2025 19:00:55 by Admin
Updated 12 November 2025 03:16:34 by Admin