Информатика

Архитектура x86. Ассемблер NASM

Гирик Алексей Валерьевич

Университет ИТМО
2022

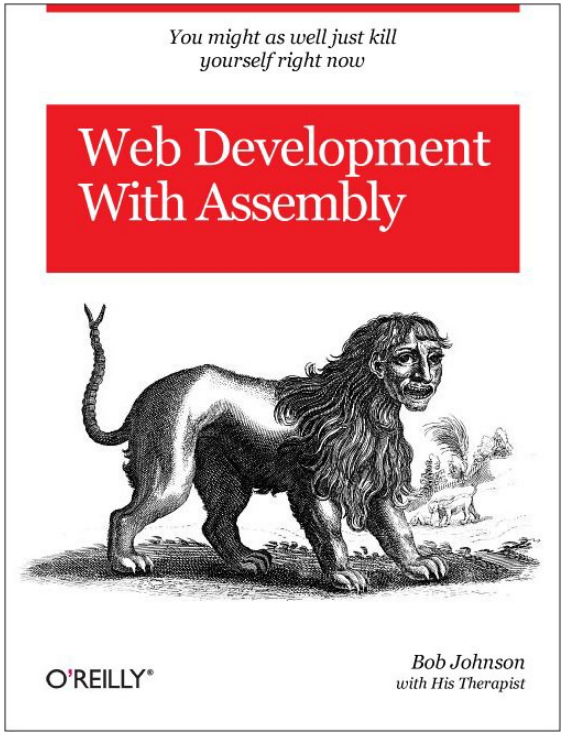
Материалы курса

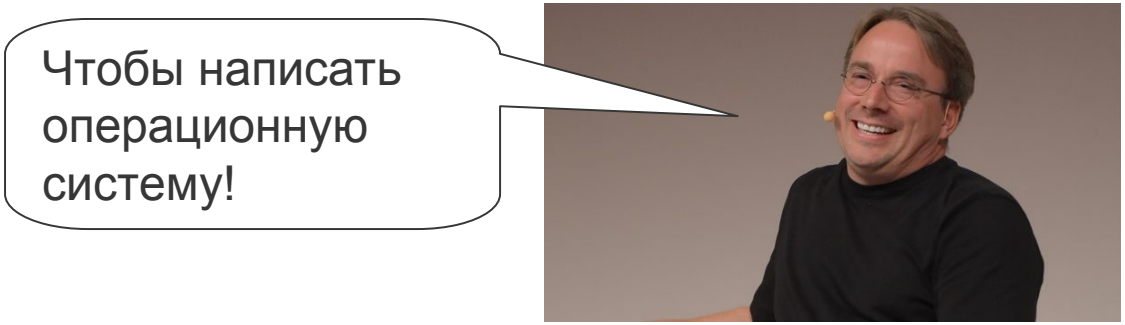
- Презентации, материалы к лекциям, литература, задания на лабораторные работы
  - shorturl.at/jqRZ6

Архитектура x86

Зачем изучать ассемблер?

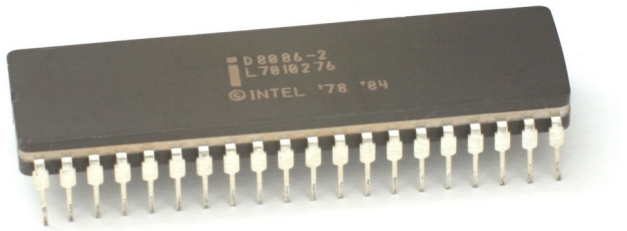
- для лучшего понимания принципов работы программ
- для развития навыка чтения дизассемблированного кода
- для решения задач оптимизации кода и поиска проблем с производительностью





История архитектуры x86

- 1978, Intel 8086
- 1982, Intel 80286
  - protected mode
- 1985, Intel 80386
  - 32-bit
- 2003, AMD Athlon 64
  - 64-bit, x86-64



Intel 8086, 1978

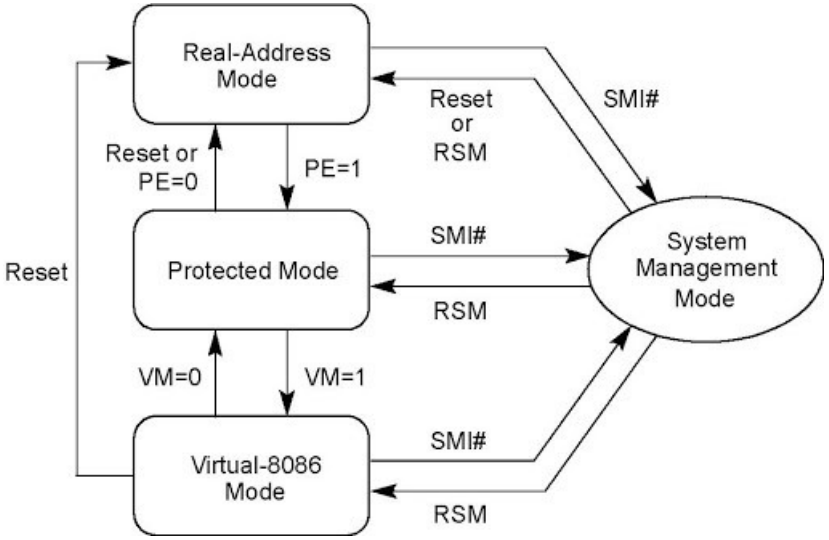
Подробнее:

https://en.wikipedia.org/wiki/X86

https://www.youtube.com/watch?v=PJmPBWQE8Uk

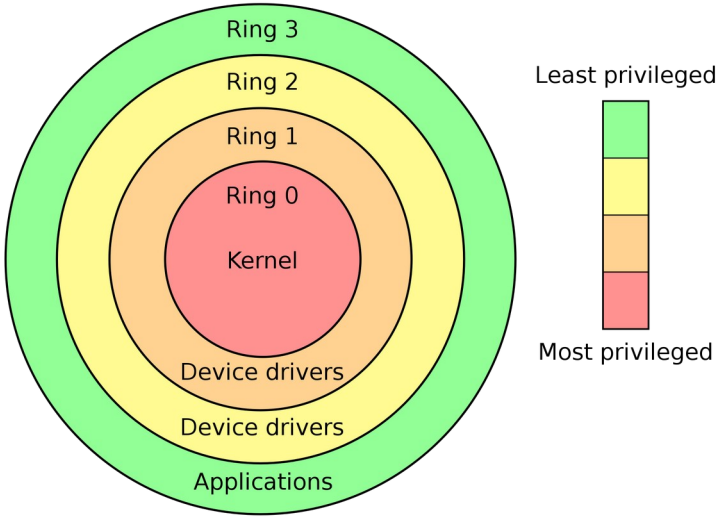
Режимы работы процессоров x86

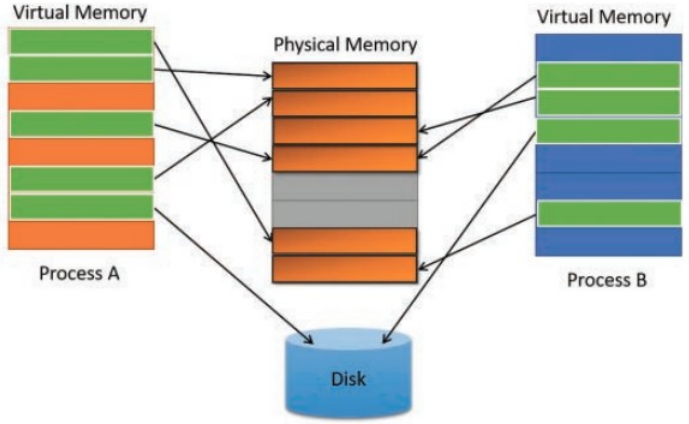
- **реальный**
- **защищенный режим виртуальной адресации**
- режим виртуального 8086
- прочие
  - SMM
  - unreal
  - ...



Защищенный режим виртуальной адресации

- ядро ОС выполняется с максимальным уровнем привилегий
- обычные программы выполняются с минимальным уровнем привилегий, они лишены возможности напрямую взаимодействовать с оборудованием
- у каждого процесса свое адресное пространство, которое постранично отображается на физическую память
- процессы ничего не знают друг о друге и "думают", что им принадлежит вся доступная память и все процессорное время

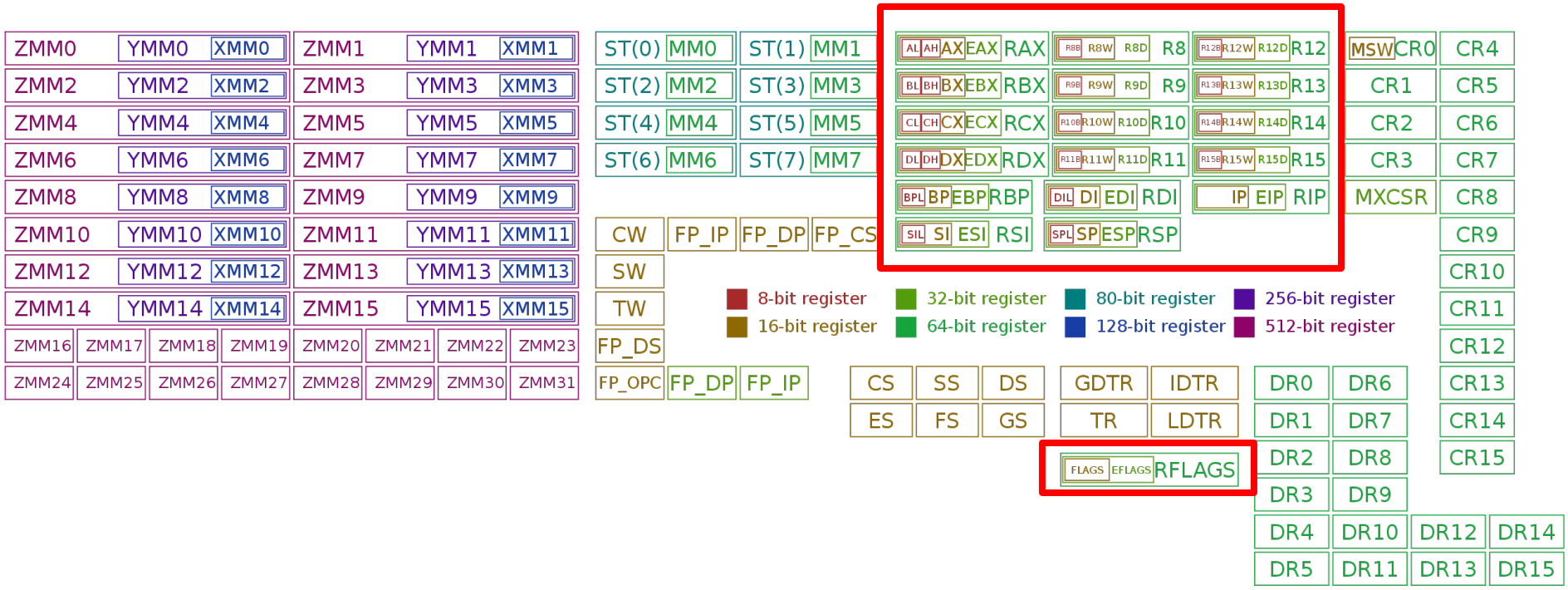




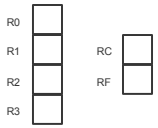
В ОС Linux посмотреть список запущенных процессов можно с помощью

- ps -ef
- top или htop

Архитектура x86-64

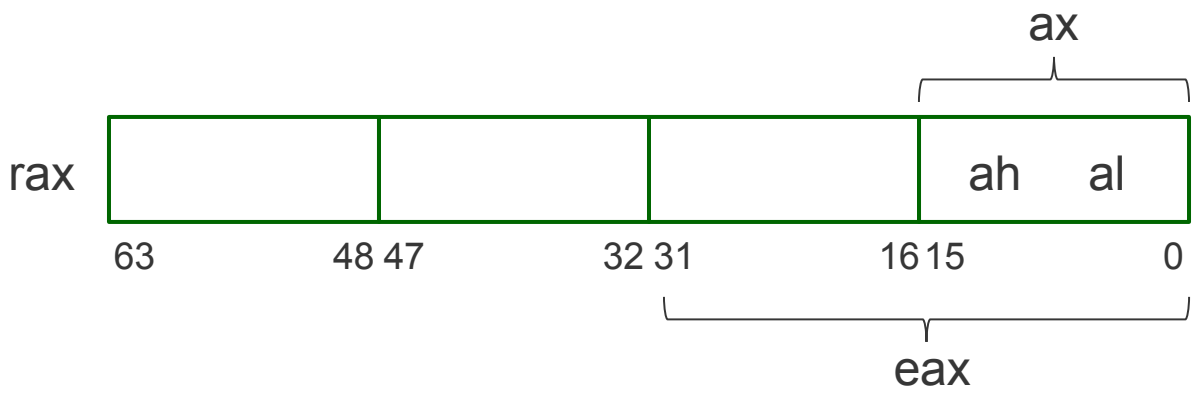


https://en.wikipedia.org/wiki/X86



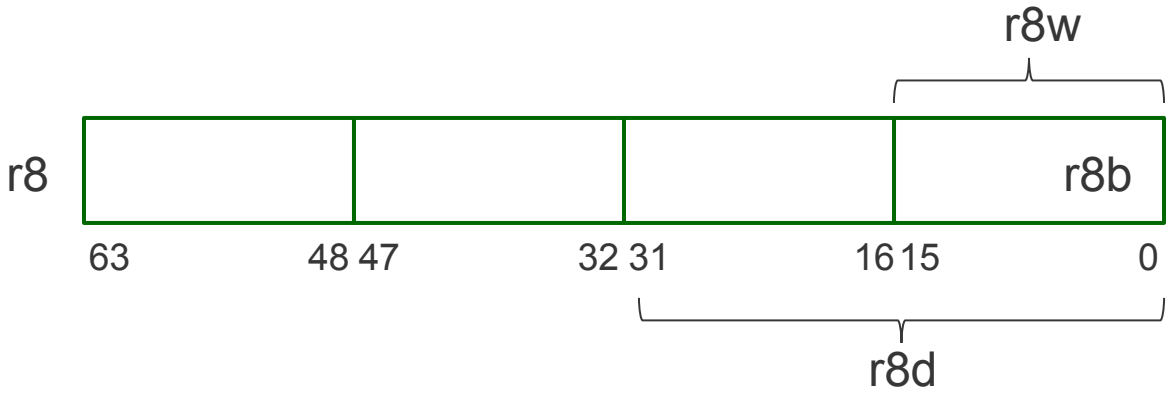
Регистры x86-64

- 8 "старых" регистров как бы общего назначения
  - rax
  - rbx
  - rcx
  - rdx
  - rbp
  - rsp
  - rsi
  - rdi



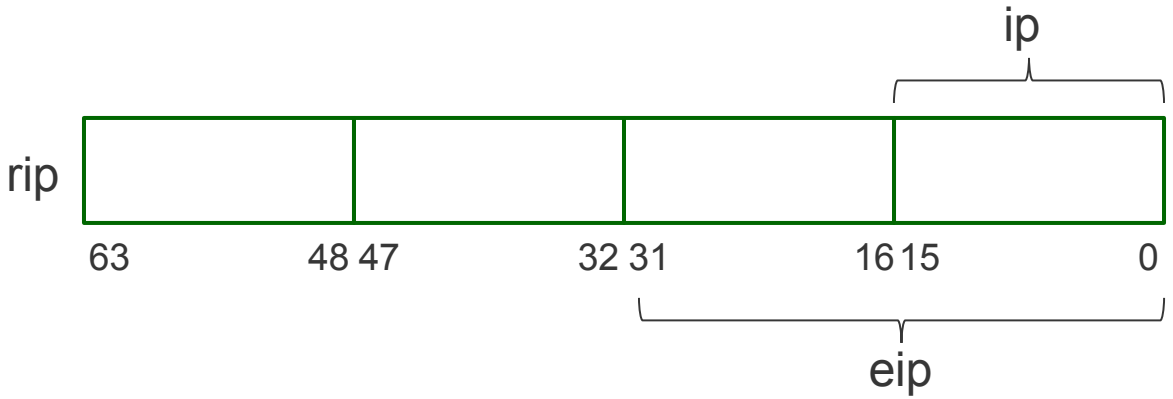
Регистры x86-64

- 8 "новых" регистров общего назначения
  - r8
  - r9
  - r10
  - r11
  - r12
  - r13
  - r14
  - r15



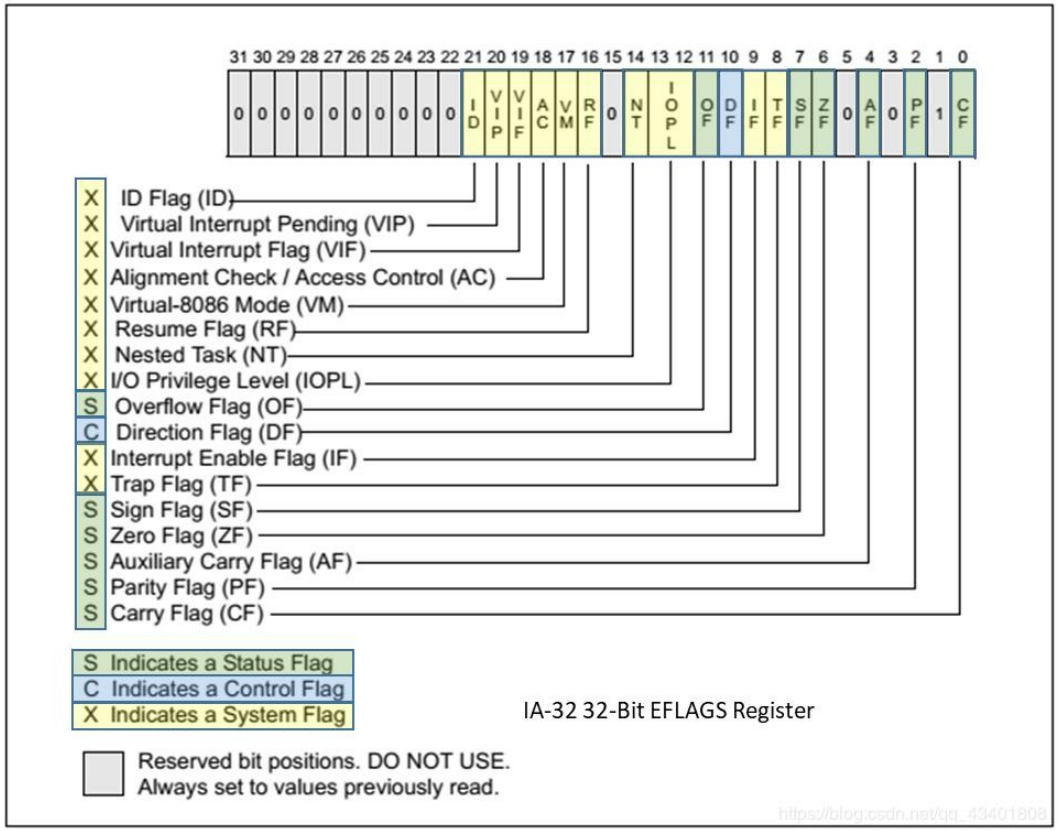
Регистры x86-64

- регистр-указатель команд
  - rip

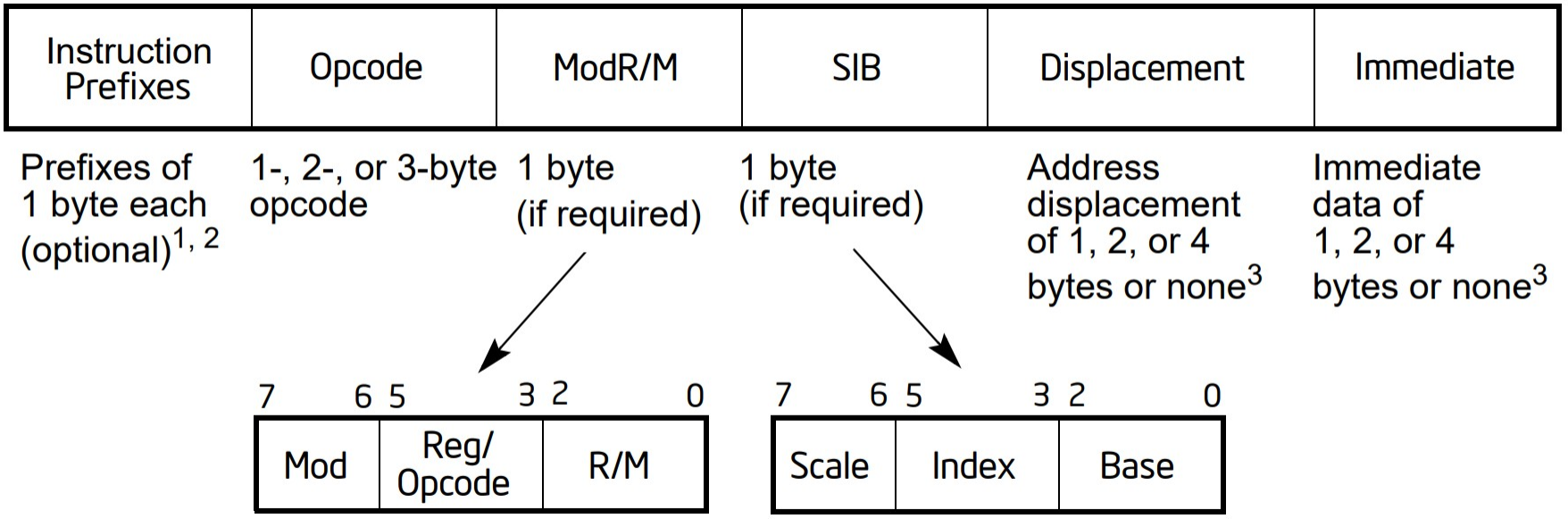


Регистры x86-64

- регистр флагов
  - rflags
    - CF
    - PF
    - AF
    - ZF
    - SF
    - OF



Система команд x86



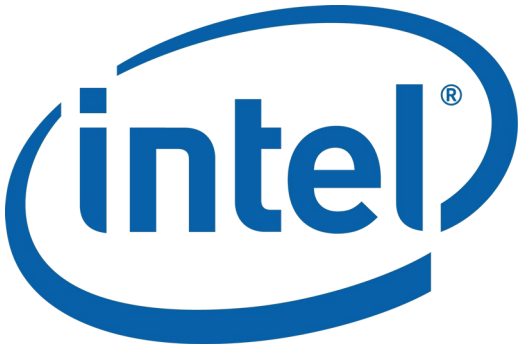
- нерегулярная сртруктура
- переменная длина (от 1 до 15 байтов)

Источник знаний о x86 и x86-64

Intel

Intel® 64 and IA-32 architectures

software developer's manual



https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html

Как написать 'Hello, world' на ассемблере?

```
#include <stdio.h>

int main() {
   printf("Hello, world!\n");
   return 0;
}

$ gcc -o hello hello.c

# AT&T syntax
$ objdump -d hello

#Intel syntax
$ objdump -d --disassembler-options=intel hello
$ objdump -d -Mintel hello
```

Получить ассемблерный код для программы на C

```
# AT&T syntax again
$ gcc -S hello.c

# Intel syntax
$ gcc -S -masm=intel hello.c

# Уберем все директивы
$ grep -v '^\s*\.' hello.s
main:
   push            rbp
   mov             rbp, rsp
   lea             rdi, .LC0[rip]
   call            puts@PLT
   mov             eax, 0
   pop             rbp
   ret
```

Собрать ассемблерный код с помощью gcc

```
# Собрать программу на C
$ gcc -o hello1 hello.c

# Собрать из ранее полученного hello.s
$ gcc -o hello2 hello.s

# Получаем идентичные программы
$ ls -la hello*

# ... ну, почти идентичные
$ vimdiff <(xxd hello1) <(xxd hello2)
```

Возвращаясь к вопросу

Как написать хеллоуворлд на ассемблере для x86-64 в Linux?

Очевидно, сначала требуется ответить на другой вопрос:
как вывести строку на экран в программе на ассемблере?

А также следует задать еще один вопрос:
какой ассемблер использовать?

Введение в язык ассемблера для x86-64

Ассемблеры для x86

- **NASM**
  - https://www.nasm.us/

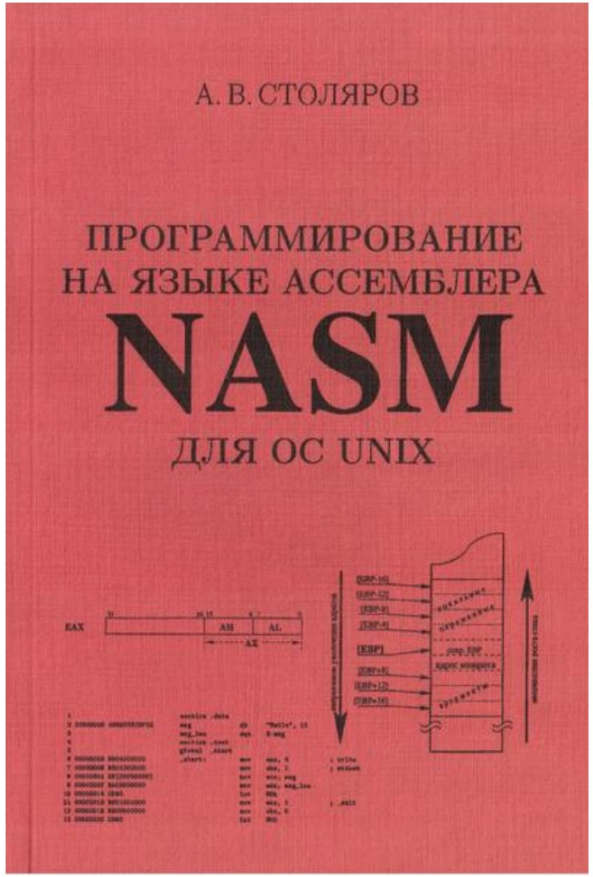
- GAS
  - https://www.gnu.org/software/binutils/

- FASM
  - http://flatassembler.net/
- MASM
  - https://docs.microsoft.com/en-us/cpp/assembler/masm
- WASM
- TASM
- ...

Учебник по ассемблеру

А.В. Столяров

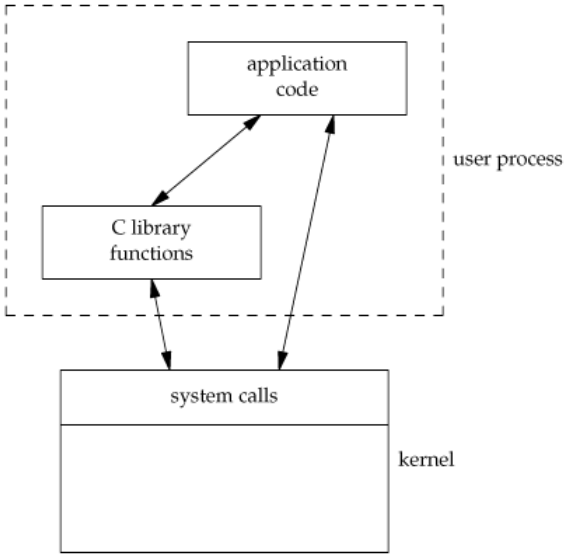
Программирование на языке ассемблера NASM для ОС UNIX



Так все-таки, как вывести строку?

Есть две возможности:

- вызывать функции из библиотеки C
  - например, puts(), printf() и другие
- обращаться к операционной системе напрямую, т.е. делать *системные вызовы*



Системные вызовы

```
# Список системных вызовов
$ man syscalls

# Нас интересует вызов write()
$ man 2 write
```

Как узнать номер системного вызова?

Можно посмотреть таблицу системных вызовов в исходном коде ядра:

```
<kernel>/arch/x86/entry/syscalls/syscall_64.tbl
```

Можно использовать магию:

```
# Получить номер системного вызова для SYS_xxx
$ printf SYS_xxx | gcc -include sys/syscall.h -E - |
  grep -v '#' | grep .

# Нас интересует вызов write()
$ printf SYS_write | gcc -include sys/syscall.h -E - |
  grep -v '#' | grep .
1
```

write()

```
#include <unistd.h>

ssize_t write(
                int fd,                 // первый аргумент
                const void *buf,        // второй
                size_t count);          // третий
```

fd – дескриптор файла:
  0 – стандартный поток ввода
  1 – стандартный поток **вывода**
buf – указатель на буфер с данными:
  в нашем случае – на **строку**
count – количество байтов в буфере
  в нашем случае – **длина** строки

write()

```
#include <unistd.h>

int main() {
   char *str_ptr = "Hello, world!\n";
   write(1, str_ptr, 14);

   return 0;
}

$ gcc -o hello_sys hello_sys.c
```

Заготовка программы на ассемблере

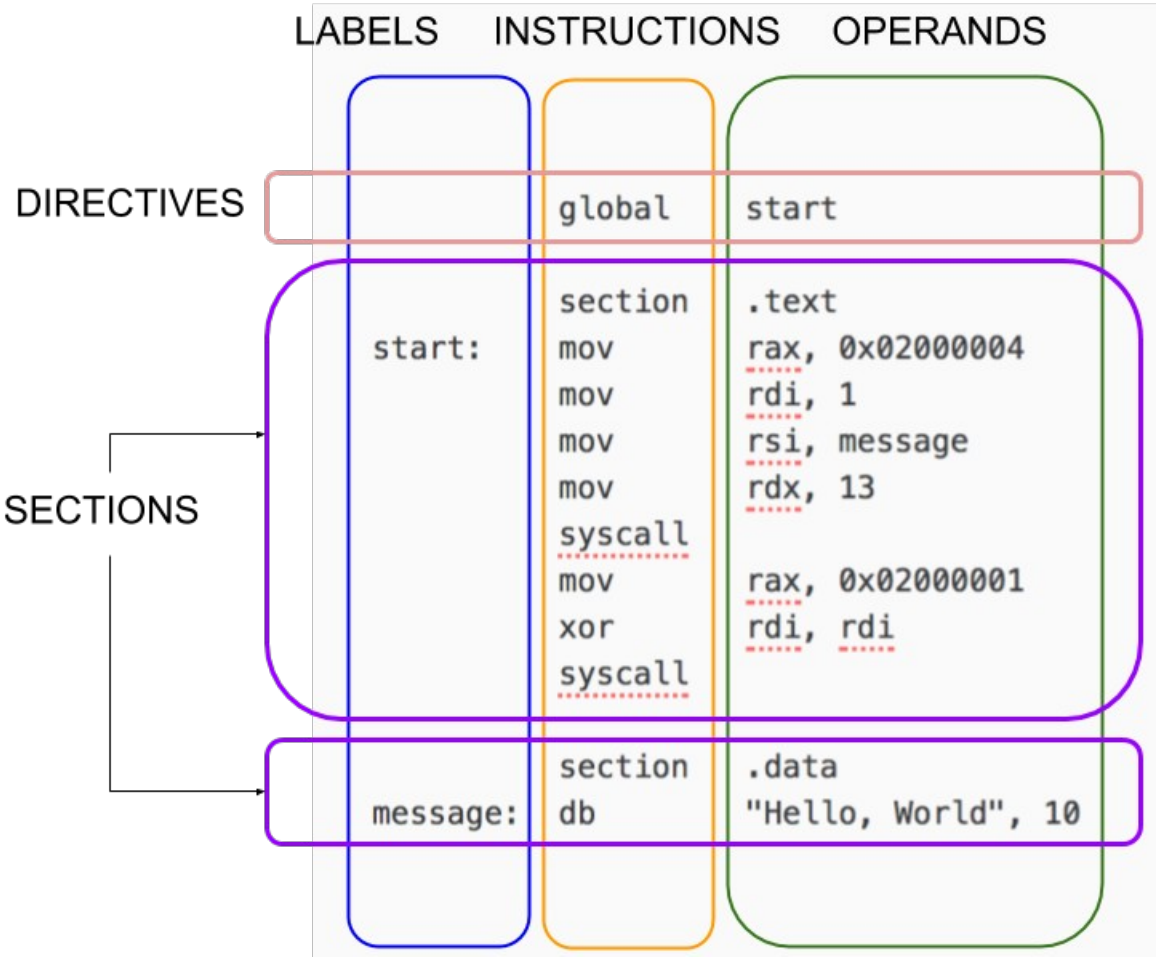
```
                global _start                ; точка входа
                                             ; в программу


                section .text
_start:         ; тут будет наш код


                section .data
                ; тут будут данные и
                ; определения констант

                section .bss
```

Структура ассемблерной программы



Зародыш хеллоуворлда

```
                global _start


                section .text
_start:         ; но как передать аргументы??
                syscall

                section .data
msg             db      'Hello, world!', 10 ; `\n`
msg_len         equ     $ - msg
sys_write       equ     1
```

Соглашения о вызовах aka calling conventions

Соглашения о вызовах определяют порядок передачи аргументов в функции. Для системных вызовов x86-64 в ОС Linux используется следующее соглашение о вызовах:

Вход:

rax	номер системного вызова
rdi	первый аргумент
rsi	второй аргумент
rdx	третий аргумент
r10	четвертый аргумент
r8	пятый аргумент
r9	шестой аргумент

Выход:

rax	результат вызова (отрицательное значение – ошибка)

Подробнее про системные вызовы:
https://blog.packagecloud.io/eng/2016/04/05/the-definitive-guide-to-linux-system-calls

Почти работающий хеллоуворлд

```
                global _start

                section .text
_start:         mov rax, sys_write
                mov rdi, 1                  ; 1 = stdout
                mov rsi, msg
                mov rdx, msg_len
                syscall

                section .data
msg             db      'Hello, world!', 10
msg_len         equ     $ - msg
sys_write       equ     1
```

Как получить исполняемый файл?

- Сначала нужно выполнить ассемблирование файла с исходным текстом с помощью ассемблера NASM и получить в результате объектный файл:

```
$ nasm -felf64 hello.S
```

- После этого с помощью системного редактора связей превратить объектный файл в исполняемый файл:

```
$ ld -o hello hello.o
```

- Можно сделать все одной командой:

```
$ nasm -felf64 hello.S && ld -o hello hello.o
```

Программу требуется корректно завершить

```
; ...
                mov rax, sys_exit
                xor rdi, rdi                ; exit code 0
                syscall

                section .data
; ...
sys_exit        equ 60
```

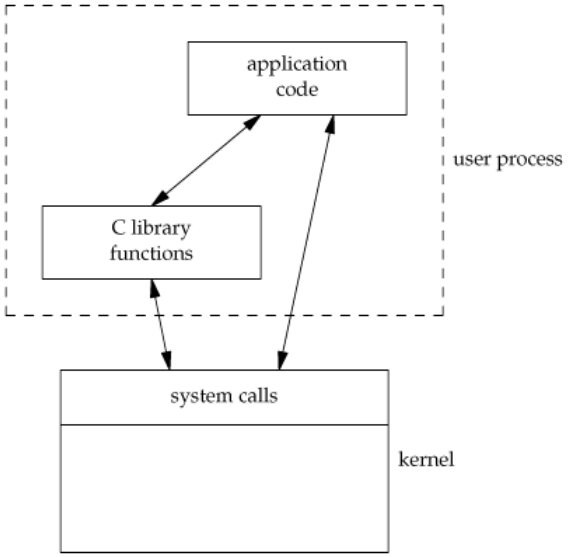
Завершение программы и возврат кода ошибки

По принятому в POSIX-системах соглашению нулевой код завершения программы означает, что она выполнилась успешно. Если код завершения отличен от нуля – это код ошибки.

```
$ echo ./hello
$ echo $?
0
```

Хеллоуворлд с вызовом функций из стандартной библиотеки C

- Обычные функции (в том числе из стандартной библиотеки C) вызываются немного иначе, чем системные вызовы, т.е. имеют немного другое соглашение о вызовах
- Необходимым становится *связывание* со стандартной библиотекой C
  - это удобнее делать с помощью gcc



Пара слов о стандартной библиотеке C

```
#include <stdio.h>

int main() {
   printf("Hello, world!\n");
   return 0;
}
```

это **НЕ** стандартная библиотека,
а всего лишь подключение
одного из заголовочных файлов
стандартной библиотеки

- Обычные Заголовочные файлы в C – это не библиотеки, а просто определения типов и прототипы фукнций, которые необходимы для компиляции
- Сама библиотека подключается на этапе редактирования связей, что возможно в двух вариантах
  - статически
  - динамически

Соглашение о вызовах для обычных функций

В 64-разрядной Linux используется соглашение System V AMD64 ABI:

Вход:

целые числа или указатели (слева направо):

rdi, rsi, rdx, rcx, r8, r9, далее – помещаются в стек (справа налево)

вещественные числа (слева направо):

xmm0, ... , xmm7, далее – помещаются в стек (справа налево)

Выход (возвращаемое значение функции):

целые числа или указатели:

rax	если помещается в 8 байтов
rax, rdx	если помещается в 16 байтов

вещественные числа:

xmm0	если помещается в 8 байтов
xmm0, xmm1	если помещается в 16 байтов

Подробнее про System V AMD64 ABI:
https://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI

Соглашение о вызовах для обычных функций

Пример:как вызвать функцию с прототипом

```
int foo(int x, long y, double z, char *s);
```

в 64-разрядной Linux?

```
mov       rdi, dword [адрес_x]
mov       rsi, [адрес_y]
movsd     xmm0, qword [адрес_z]
mov       rdx, адрес_s
call      foo
```

Обратите внимание, что ассемблер NASM использует директиву `dword`, а не `dword ptr` для уточнения размера операнда!

Хеллоуворлд с библиотечными вызовами

```
            global main
            extern puts

            section .text
main:
            mov rdi, msg
            call puts               ; вызов функции puts

            ret                     ; возврат из функции

            section .data
msg         db      'Hello, world!', 0
```

Как получить исполняемый файл?

По очереди:

```
$ nasm -felf64 hello_libc.S
$ gcc -no-pie -o hello_libc hello_libc.o
```

Или одной командой:

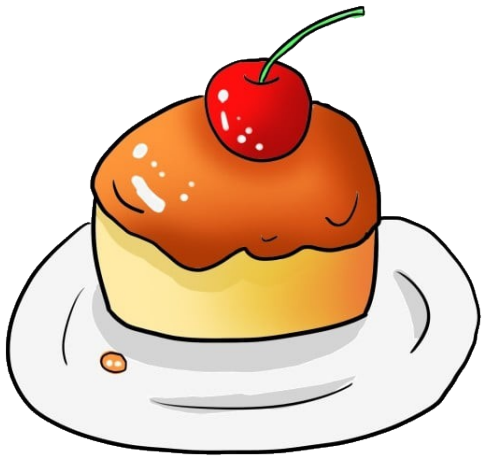
```
$ nasm -felf64 hello_libc.S && gcc -no-pie -o hello_libc
  hello_libc.o
```

Вызов функции, написанной на ассемблере, из основной программы на C

```
// файл hello_main.c

void hello_func();

int main() {
   // Здесь только вызов, сама функция
   // определена в файле hello_func.S
   hello_func();
   return 0;
}
```



Вызов функции, написанной на ассемблере, из основной программы на C

```
; файл hello_func.S

            global hello_func
            extern puts

            section .text
hello_func:
            mov rdi, msg
            call puts
            ret

            section .data
msg         db      'Hello, world!', 0
```

Как получить исполняемый файл из нескольких объектных?

Сначала скомпилируем основной модуль:

```
$ gcc -c hello_main.c
```

Затем модуль с функцией hello_func:

```
$ nasm -felf64 hello_func.S
```

Теперь соберем их вместе:

```
$ gcc -no-pie -o hello_func hello_main.o hello_func.o
```

Резюме

- В настоящее время изучать ассемблер необходимо в основном для понимания принципов работы процессоров и применения этих знаний в задачах обратной разработки, оптимизации кода и разработки под микроконтроллеры без поддержки C
- При написании программ на ассемблере для ОС Linux есть возможность использовать как системные вызовы напрямую, так и функции из библиотек, например, из стандартной библиотеки C

Задание к следующей лекции

- Столяров. Программирование на языке ассемблера NASM для ОС UNIX
  - гл. 1
  - гл. 2 до §2.6