

# Информатика

## Архитектура x86. Ассемблер NASM. Основные команды. Отладка

Гирик Алексей Валерьевич

Университет ИТМО  
2022

# Материалы курса

- Презентации, материалы к лекциям, литература, задания на лабораторные работы
  - [shorturl.at/jqRZ6](https://shorturl.at/jqRZ6)



# Terms & Conditions

	2	3	4	5
Л/р № 1	?	+	+	+
Л/р № 2	?	+	+	+
Правильных ответов в тестах	¬_(ツ)_/	>= 8	>= 19	>= 31

# Отладка программ

# Что такое отладчик

- Отладчик (debugger) – программа, которая позволяет выполнять другую программу в пошаговом режиме и наблюдать за состоянием памяти отлаживаемой программы
- Отладчик может быть интегрирован в среду разработки (Visual Studio, XCode) или поставляться отдельно (gdb, lldb)
- Отладку можно выполнять на уровне исходных кодов или машинных инструкций

# Что можно узнать с помощью отладчика

- Значения переменных во время выполнения
- Стек вызовов функций
- Значения аргументов функций
- Состояние памяти
- Состояние регистров процессора
- Если происходит исключение, то
  - тип исключения
  - место возникновения

# Отладчик gdb

- Основные команды:
  - ❑ `help <cmd>` – показать справку по `cmd`
  - ❑ `break <func>` – точка останова в `func`
  - ❑ `run` – запустить программу
  - ❑ `next`, `step` – выполнить следующую строку исходного кода
  - ❑ `ni`, `si` – выполнить следующую инструкцию
  - ❑ `list` – показать исходный код
  - ❑ `disas /s` – показать код+инструкции
  - ❑ `frame` – показать текущее положение
  - ❑ `kill` – остановить выполнение программы
  - ❑ `quit` – выход

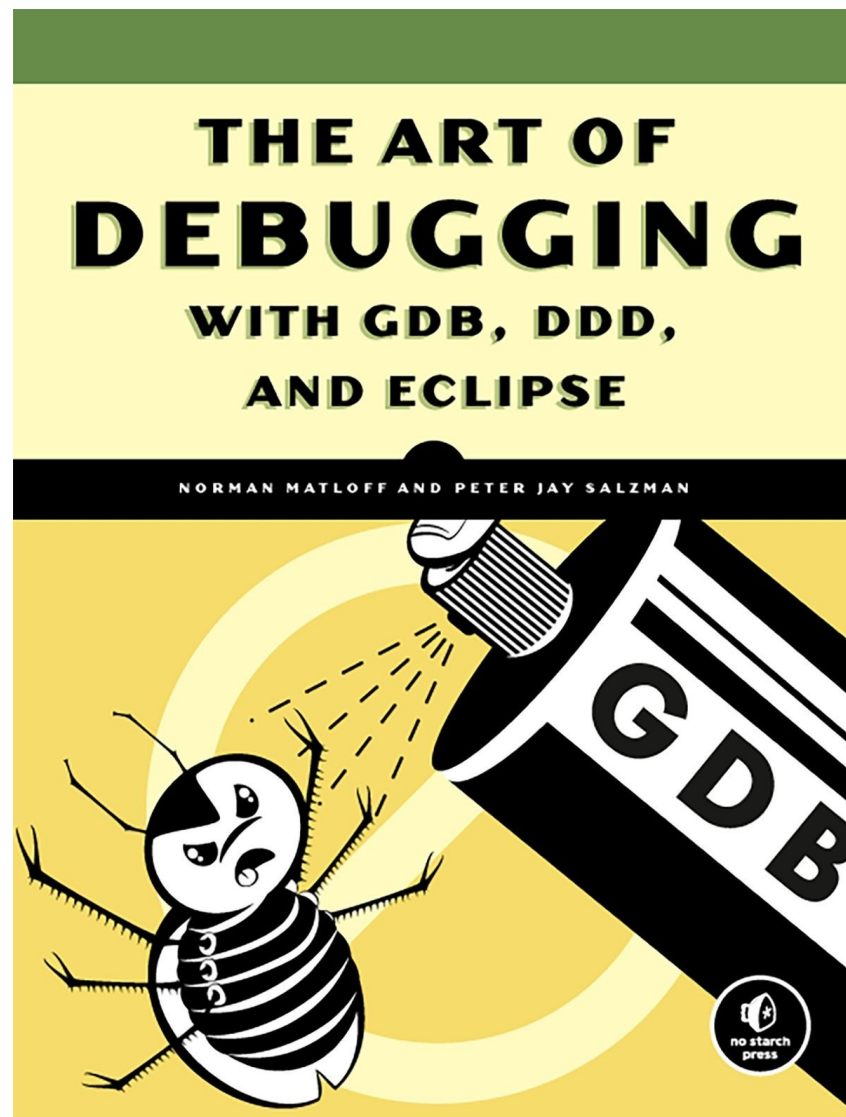
# Отладчик gdb

- Информация о текущем состоянии:
  - ❑ `info reg` – состояние регистров
  - ❑ `info local` – локальные переменные
  - ❑ `info arg` – значения аргументов функции
  - ❑ `p <переменная>` – вывести значение переменной
  - ❑ `x/<формат> <объект>` – вывести значение объекта (переменной, адреса,...) в заданном формате
  - ❑ `bt` – показать состояние стека вызовов
  - ❑ `display <объект>` – выводить значение объекта после каждой команды



# Книжка по gdb

- *N. Matloff, P. Salzman*  
The Art of Debugging with  
gdb, ddd and Eclipse



# gdb dashboard

<https://github.com/cyrus-and/gdb-dashboard>

```
GDB dashboard

Output/messages
17 for (i = 0; i < text_length; i++) {

Assembly
0x0000555555551ec 48 8b 45 f8 encrypt+103 mov rax,QWORD PTR [rbp-0x8]
0x0000555555551f0 48 01 d0 encrypt+107 add rax,rdx
0x0000555555551f3 31 ce encrypt+110 xor esi,ecx
0x0000555555551f5 89 f2 encrypt+112 mov edx,esi
0x0000555555551f7 88 10 encrypt+114 mov BYTE PTR [rax],dl
0x0000555555551f9 48 83 45 f8 01 encrypt+116 add QWORD PTR [rbp-0x8],0x1
0x0000555555551fe 48 8b 45 f8 encrypt+121 mov rax,QWORD PTR [rbp-0x8]
0x000055555555202 48 3b 45 e8 encrypt+125 cmp rax,QWORD PTR [rbp-0x18]
0x000055555555206 72 bb encrypt+129 jb 0x555555551c3 <encrypt+62>
0x000055555555208 90 encrypt+131 nop

Breakpoints
[1] break at 0x0000555555552d9 in xor.c:56 for xor.c:56 hit 1 time
[2] break at 0x000055555555199 in xor.c:13 for encrypt hit 1 time
[3] break at 0x000055555555521b in xor.c:27 for dump if i = 5
[4] write watch for output[10] hit 1 time

Expressions
password[1 % password_length] = 101 'e'
text[i] = 32 ' '
output[i] = 69 'E'

History
$$1 = 0x55555559260 "\f\032\v\006\022\004\032\001\037E": 12 '\f'
$$0 = 0x7fffffffef2c "hunter2": 104 'h'

Memory
password
0x00007fffffffef2c 68 75 6e 74 65 72 32 00 64 6f 65 73 6e 74 20 6c hunter2-doesnt-1
text
0x00007fffffffef34 64 6f 65 73 6e 74 20 6c 6f 6f 6b 20 6c 69 6b 65 doesn't look like
0x00007fffffffef44 20 73 74 61 72 73 20 74 6f 20 6d 65 00 48 4f 53 -stars-to-me-HOS

output
0x0000555555559260 0c 1a 0b 07 0b 06 12 04 1a 01 1f 45 00 00 00 .....E....
0x0000555555559270 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

Registers
rax 0x000055555555926b rbx 0x0000000000000000 rcx 0x0000000000000065
rdx 0x0000000000000045 rsi 0x0000000000000045 rdi 0x00007fffffffef40
rbp 0x00007fffffffef40 rsp 0x00007fffffffef00 r8 0x0000000000000003
r9 0x000000000000a330 r10 0x0000555555559010 r11 0x0000000000000030
r12 0x00005555555550a0 r13 0x00007fffffffef60 r14 0x0000000000000000
r15 0x0000000000000000 rip 0x0000555555551f9 eflags [ IF ]
cs 0x0000000033 ss 0x00000002b ds 0x00000000
es 0x00000000 fs 0x00000000 gs 0x00000000

Source
12 /* obtain the lengths */
13 password_length = strlen(password);
14 text_length = strlen(text);
15
16 /* perform the encryption */
17 for (i = 0; i < text_length; i++) {
18     output[i] = text[i] ^ password[i % password_length];
19 }
20 }
21

Stack
[0] from 0x0000555555551f9 in encrypt+116 at xor.c:17
[1] from 0x0000555555552f0 in main+139 at xor.c:56

Threads
[1] id 8 name xor from 0x0000555555551f9 in encrypt+116 at xor.c:17

Variables
arg password = 0x7fffffffef2c "hunter2": 104 'h'
arg text = 0x7fffffffef34 "doesn't look like stars to me": 100 'd'
arg output = 0x55555559260 "\f\032\v\006\022\004\032\001\037E": 12 '\f'
loc password_length = 7
loc text_length = 28
loc i = 11

>>> |
```

# Вызов подпрограмм

# Подпрограммы

- Функции, процедуры, подпрограммы
  - имена имеют значение для программиста, для исполнения программы нужны только адреса
- С точки зрения ассемблера подпрограмма – просто метка, отмечающая начало некоторого кода

```
_proc:    push rbp  
          ...  
          call _proc
```

# Что значит – вызвать подпрограмму?

- Нужно передать подпрограмме фактические аргументы
- Затем необходимо запомнить адрес, куда требуется вернуться
- Перед возвратом нужно сохранить возвращаемое значение
- После возврата нужно получить возвращаемое значение (если какое-то значение предполагалось возвращать)

# Как вызвать подпрограмму?

- если в процессоре нет поддержки вызова подпрограмм, то есть для переходов доступна только инструкция `jmp`

```
r0 <- адрес_возврата  
rc <- адрес_подпрограммы_foo          ; вызов  
...
```

```
foo:                                     ; r0 занят  
...  
rc <- r0                                ; возврат
```

# Как вызвать подпрограмму?

- возникает логичный вопрос – что делать, если из подпрограммы foo нужно вызвать подпрограмму bar?
- нужно каким-то образом сохранить содержимое регистра r0 перед вызовом bar
- возможное решение – сохранить содержимое регистра в памяти

# Стек

- Физический смысл стека – яма



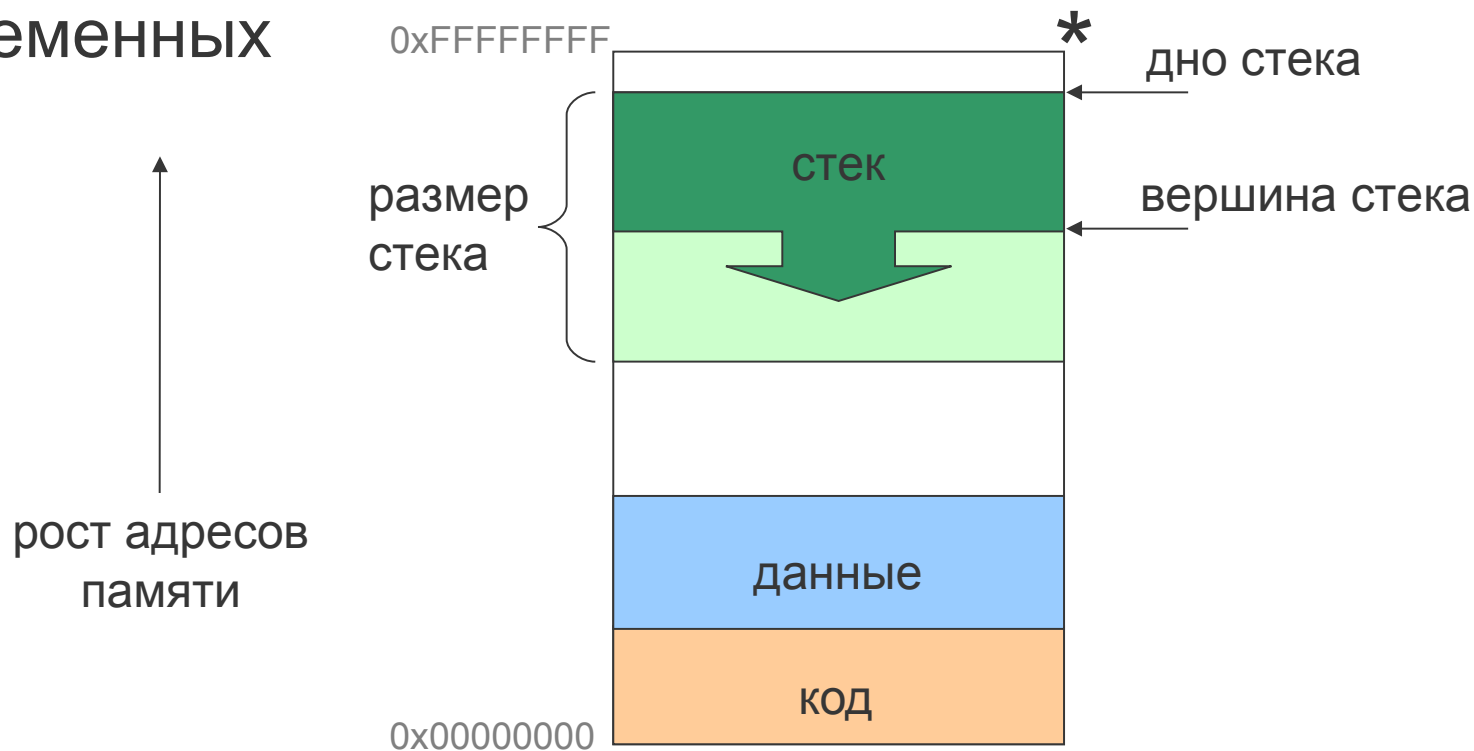
- Стек использует дисциплину LIFO (Last In First Out)





# Операции со стеком

- Стек – это область памяти в адресном пространстве процесса, которая используется для хранения активационных записей функций и локальных переменных



# Вызов и возврат из подпрограмм

- в настоящее время практически все процессоры поддерживают вызов подпрограмм на аппаратном уровне
- **call <OP>**
  - `call 0x55004046` ; вызов по адресу
  - `call fib` ; вызов по метке (адресу)
  - `call [rax]` ; вызов по адресу в регистре
- **ret**
  - `ret` ; извлечь из стека адрес возврата  
; и выполнить переход по нему

# Вызов функции

```
11      int res = fib(n);
```

```
=> 0x000055555555469a <+18>:      mov     edi, eax
    0x000055555555469c <+20>:      call    0x55555555464a <fib>
    0x00005555555546a1 <+25>:      mov     DWORD PTR [rbp-0x4], eax
```



# 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](https://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI)

# Рекурсия и рекурсивные вызовы

# Рекурсия

- Числа Фибоначчи

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

$$F_1 = 1$$

$$F_2 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

- **Рекурсия** — способ общего определения объекта или действия через себя, с использованием ранее заданных частных определений

# Рекурсивные функции

- Напишем функцию, вычисляющую n-ый член ряда

```
int fib(int n) {
```

```
    ...
```

```
}
```

```
...
```

```
printf("%d", fib(5));
```

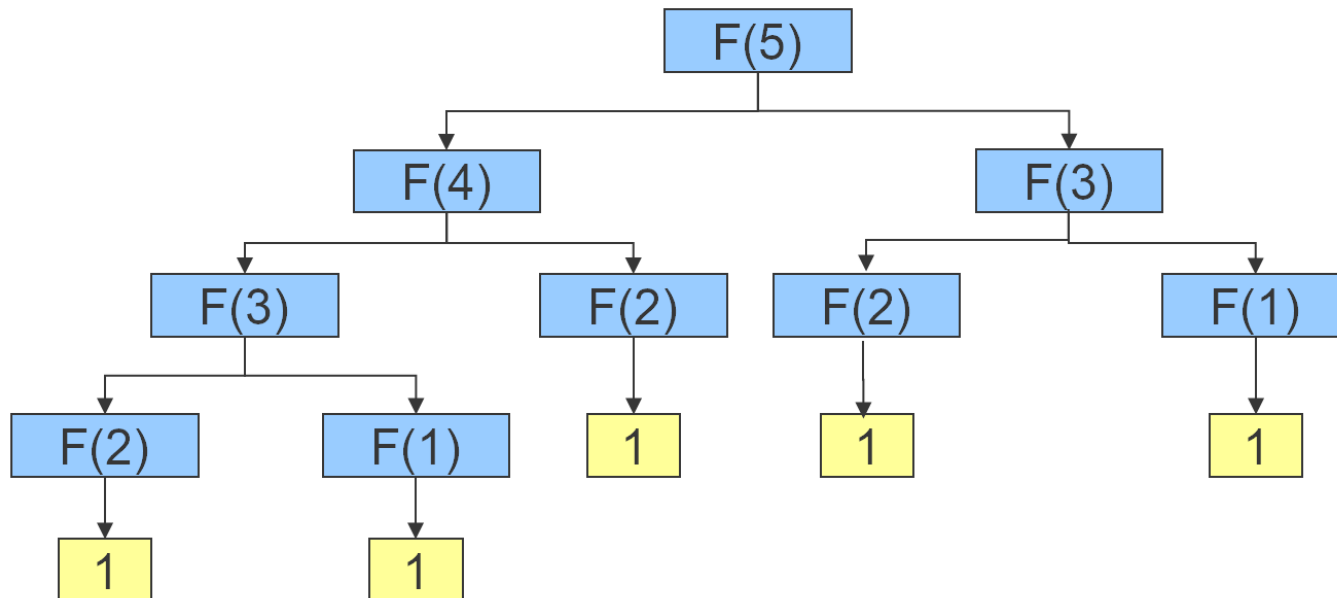
# Рекурсивные функции

```
int fib(int n) {  
    if (n < 3) return 1;  
    return fib(n-1) + fib(n-2);  
}
```



# Рекурсивные функции

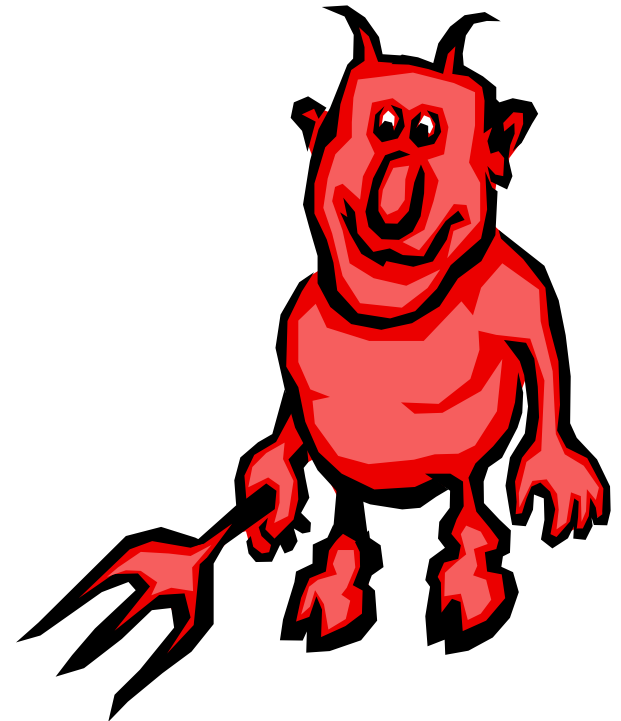
```
int fib(int n) {  
    if (n < 3) return 1;  
    return fib(n-1) + fib(n-2);  
}
```



# Конечная и бесконечная рекурсии

```
void bad_func() {  
    ...  
    bad_func();  
    ...  
}
```

Бесконечная рекурсия – зло!



# Когда должна прекратиться рекурсия?

- любое рекурсивное вычисление должно когда-либо завершиться
- любой рекурсивный алгоритм должен содержать условия остановки рекурсии

```
int fib(int n) {  
    if (n < 2) return 1;  
    return fib(n-1) + fib(n-2);  
}
```

# Задание к следующей лекции

- Столяров. Программирование на языке ассемблера NASM для ОС UNIX
  - гл. 2 §2.6 –§2.9
  - гл. 3

