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

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

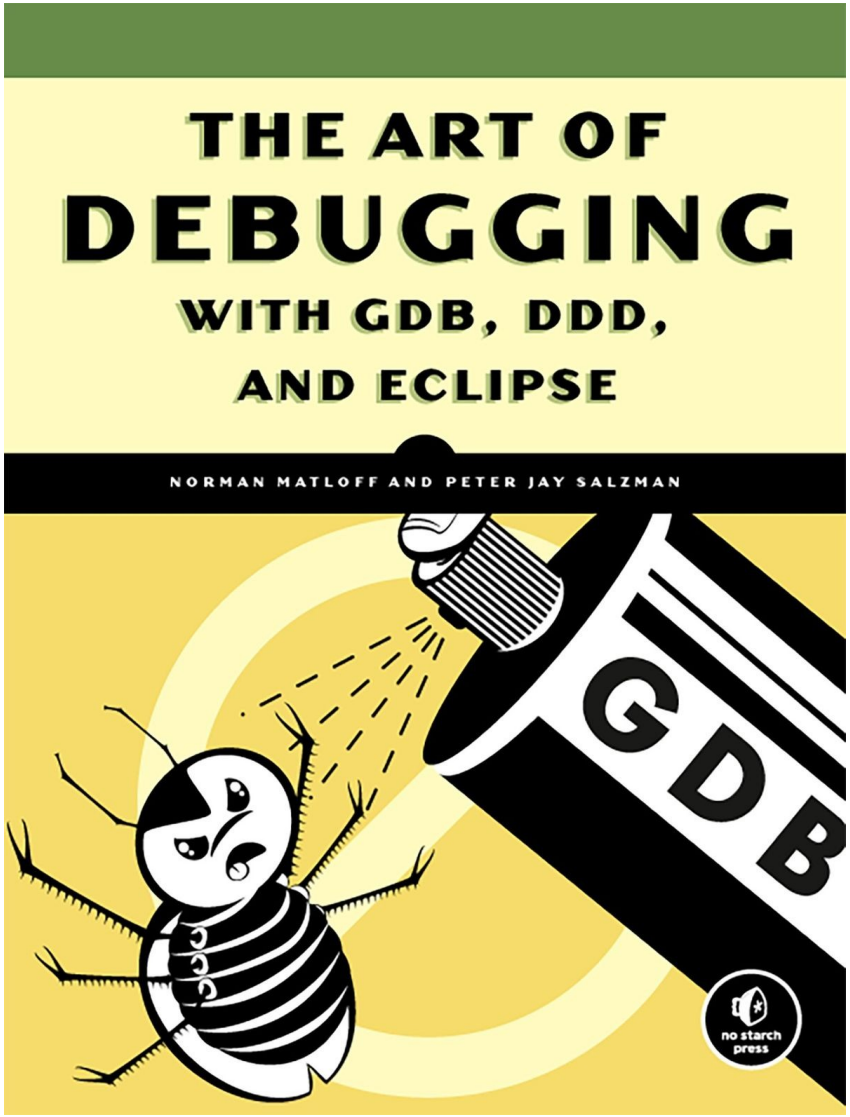
- Основные команды:
  - help <cmd> – показать справку по cmd
  - break <func> – точка останова в func
  - run – запустить программу
  - next, step – выполнить следующую строку исходного кода
  - ni, si – выполнить следующую инструкцию
  - list – показать исходный код
  - disas /s – показать код+инструкции
  - 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

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 какоое-то значение предполагалось возвращать)

Как вызвать подпрограмму?

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

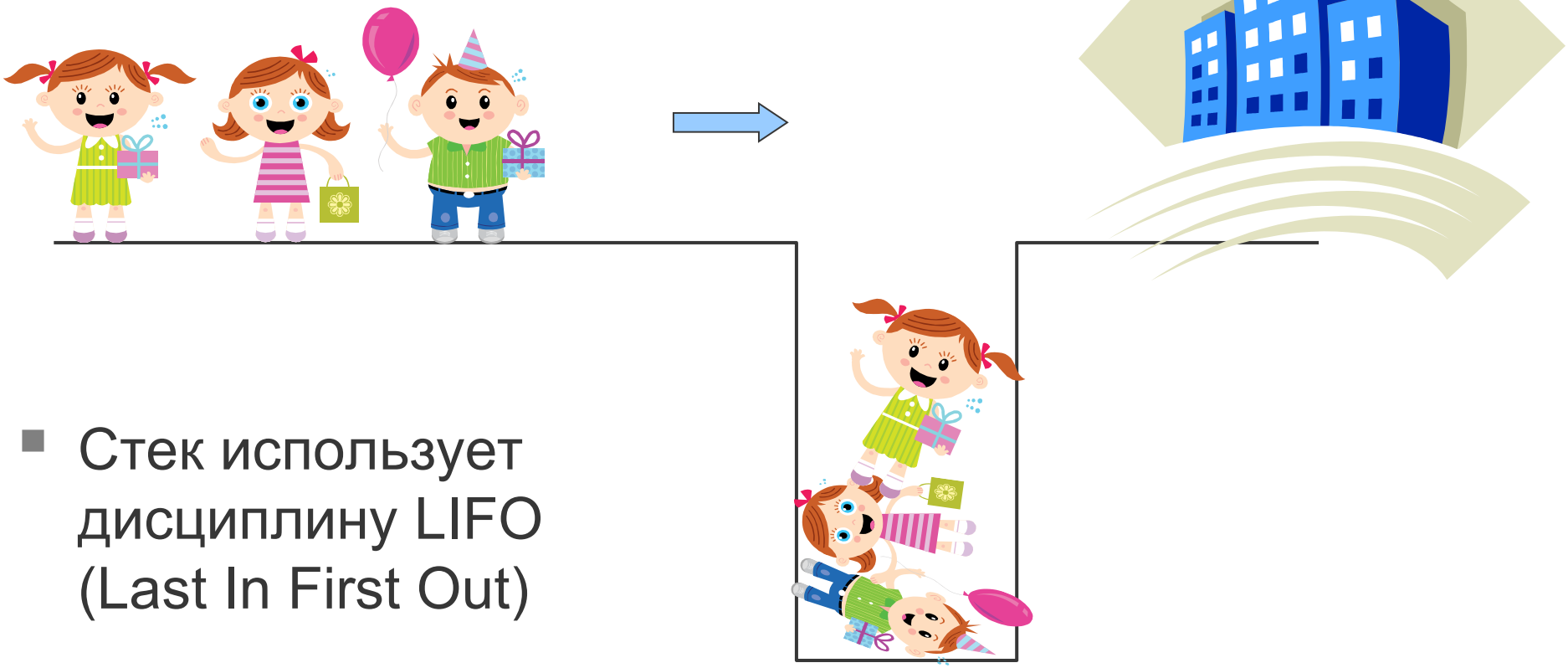
```
        r0 <- адрес_возврата
        rc <- адрес_подпрограммы_foo            ; вызов
        ...

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

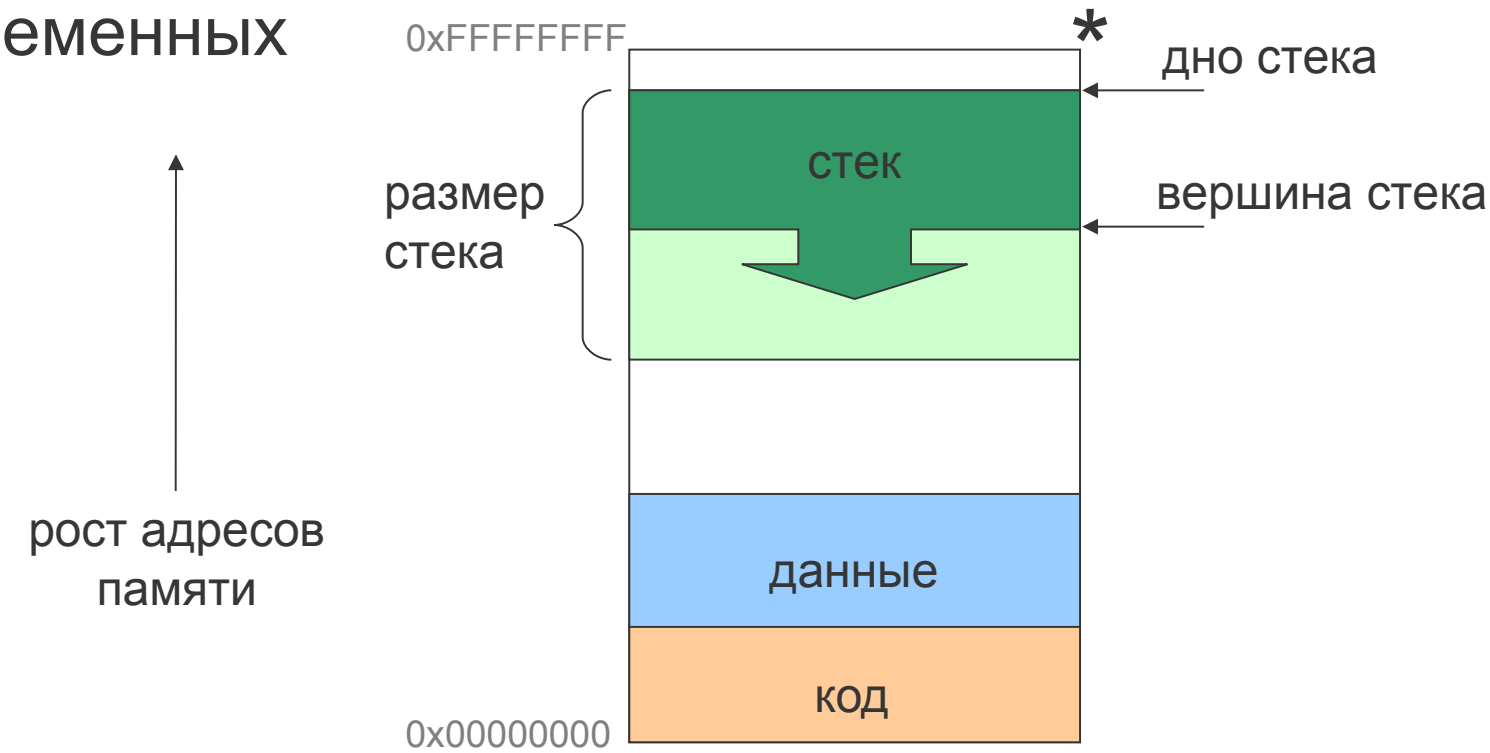
- Физический смысл стека – яма



- 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



* – данная иллюстрация не отражает истинного вида адресного пространства процесса в ОС Linux

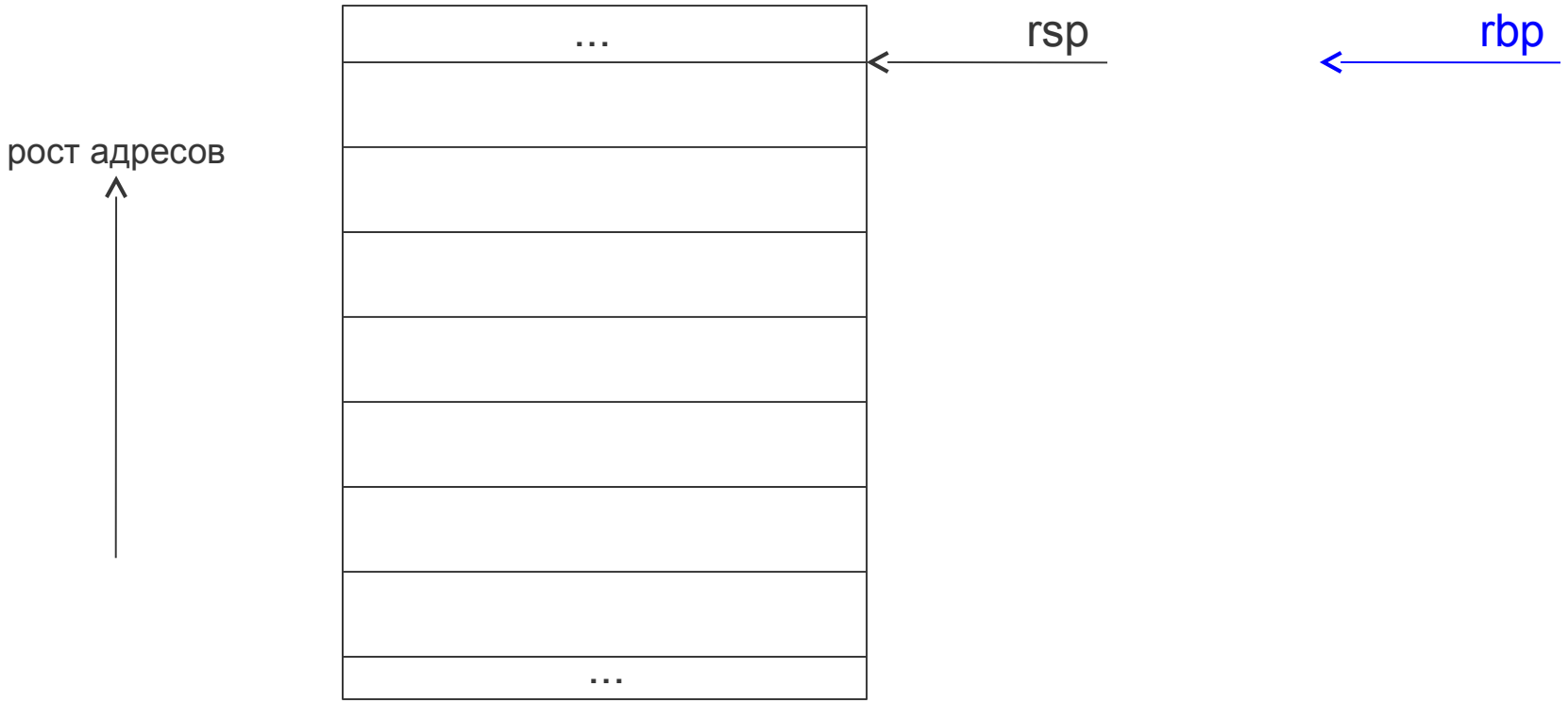
Вызов и возврат из подпрограмм

- 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);
=> 0x00005555555469a <+18>:     mov    edi,eax
   0x00005555555469c <+20>:     call   0x55555555464a <fib>
   0x0005555555546a1 <+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

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-ный член ряда

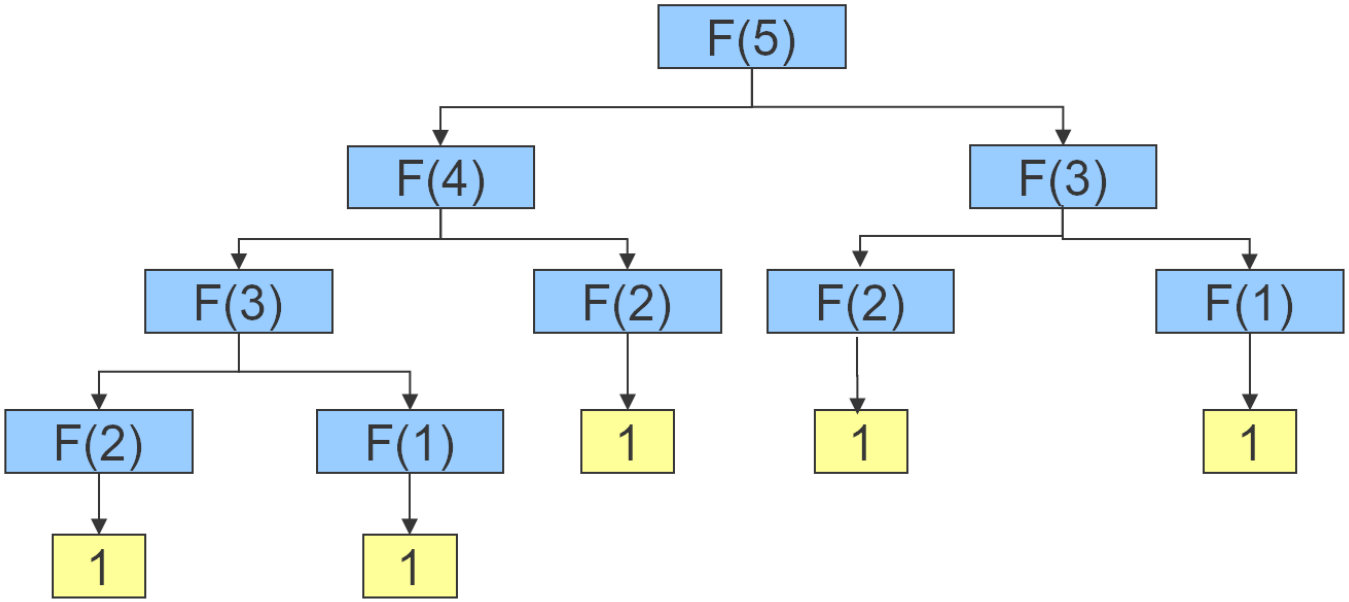
```
int fib(int n) {
  ...
}

...

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

Рекурсивные функции

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

Рекурсивные функции

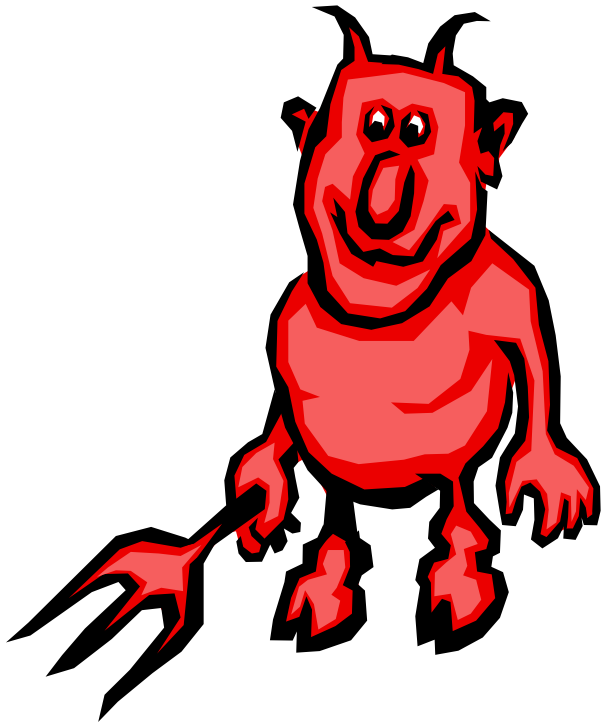
```
int fib(int n) {
  if (n < 3) return 1;
  return fib(n-1) + fib(n-2);
}
```



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