

# АКОС

Осень 2021 – Весна 2022

# jmp

- [Общая информация](#)

Первый модуль

- [Введение в Linux и базовые инструменты разработки](#)
- [Регулярные выражения](#)
- [Командный интерпретатор bash и утилита sed](#)
- [Целочисленная и вещественная арифметика](#)
- [Архитектура AArch64](#)
- [Архитектура x86-64](#)
- [Векторные вычисления и набор команд AVX](#)

# open

Второй модуль

- Системные вызовы
- Низкоуровневый файловый ввод и вывод
- Атрибуты файлов и файловых дескрипторов
- Отображение файлов на память
- Запуск и завершение работы процессов
- Запуск программ через fork-exes

# goto

## Третий модуль

- Копии файловых дескрипторов и неименованные каналы
- Сигналы
- Сигналы реального времени
- Сокеты TCP/IP
- Мультиплексирование ввода-вывода
- Многопоточность
- Синхронизация потоков
- Низкоуровневое сетевое взаимодействие

# goto

## Четвертый модуль

- [Библиотеки функций и их загрузка](#)
- [CMake](#)
- [HTTP, cURL](#)
- [Шифрование](#)
- [FUSE](#)
- [Python Extending & Embedding](#)

# Общая информация

# Семинарист

- Сергей
- Ведёт семинары, смотрит код, принимает задачи, помогает разобраться
- NLP/ML @ ABBYY
- Беспокоить в Telegram: @eaglemango

# Ассистент

- Лиза
- Смотрит код, принимает задачи, помогает разобраться
- SWE Intern @ Intel
- Беспокоить в Telegram: @de1iza



# Полезные ссылки

- Программа курса: <https://vk.cc/c5xzyo>
- Запись на сдачи: <https://vk.cc/c5xzCW>
- Контесты: <https://ejudge.atp-fivt.org>
- Эта презентация: <https://vk.cc/c5xG02>
- Ссылка на Slack: <https://vk.cc/c5BvK9>

# Введение в Linux и базовые инструменты разработки

# \$ *command*

- *man* - мануалы
  - *man man* - мануалы по мануалам (да, серьёзно)
- *touch* - создание файлов
- *mkdir* - создание каталогов
- *echo* - вывод строки
- *pwd* - вывод текущего каталога
- *whoami* - вспомнить самое важное
- *alias* - создание ярлыка для другой команды
- *cat* - вывод содержимого файла

# \$ *command*

- *cd* - навигация по каталогам
  - *.* - текущий каталог
  - *..* - каталог уровнем выше
  - *~* - домашняя директория текущего пользователя
- *cp* - копирование файлов
- *mv* - перемещение файлов (можно использовать для переименования)
- *rm* - удаление файлов (можно использовать для удаления системы)
- *ls* - вывод содержимого текущего каталога
  - *ls -l* - вместе с дополнительной информации о файлах
  - *ls -a* - вместе со скрытыми файлами

\$ *command*

Если было упущено что-то важное, то оно обязательно появится здесь

# Исполняемые файлы

- Бинарные файлы, начинающиеся с Magic Bytes `7f 45 4c 46` (ELF, Executable and Linkable Format)
  - Можно посмотреть с помощью `xxd`
- Текстовые файлы, начинающиеся с `#!` (шебанг) и пути к интерпретатору
- У нас должны быть права на исполнение
  - Мы можем их установить: `chmod +x`

# Компиляторы

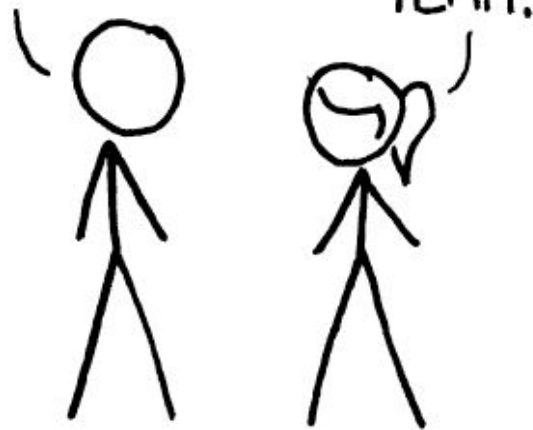
- GCC
- Clang
- Microsoft Visual C++
- Можно написать свой (не рекомендуется)

# HOW STANDARDS PROLIFERATE:

(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)

SITUATION:  
THERE ARE  
14 COMPETING  
STANDARDS.

14?! RIDICULOUS!  
WE NEED TO DEVELOP  
ONE UNIVERSAL STANDARD  
THAT COVERS EVERYONE'S  
USE CASES.



SOON:

SITUATION:  
THERE ARE  
15 COMPETING  
STANDARDS.



# Этапы компиляции

- Выполнение директив препроцессора (*gcc -E*)
- Трансляция
  - *gcc -c* - в машинный код
  - *gcc -S* - в код на языке ассемблера
- Линковка (*ld*)

# Директивы препроцессора

- *#include*
- *#define* (константы, макросы)
- *#ifdef, #ifndef*
- Предопределённые константы (например, `__LINE__` или `__FILE__`)

# Автоматизация сборки

- Производится *make*, следуя содержимому Makefile
- Для генерации Makefile в больших проектах существует *stake*

# Code Style

- Много стандартов с одной целью - привести код к единому читаемому виду
- Удобно использовать *clang-format* для автоматического форматирования
- Конфиг нашего курса: <https://vk.cc/c5xFth>

# Code Style

- Названия переменных и функций
- Инициализация переменных
- Освобождение выделенной памяти
- Структурирование кода
- Комментарии

# Дебаг

- *printf("pochemu ne rebotayet")* каждую строку может не помочь
- Для поиска ошибок удобно использовать специально обученный дебаггер (например, *gdb*)
  - Предварительно необходимо скомпилировать программу с флагом *-g*
  - Для удалённой отладки можно использовать *gdbserver*

# Регулярные выражения



**Steeve**

@ifosteve



The plural of regex is regrets

10:21 PM · Nov 1, 2019 · Twitter Web App

---

**3,610** Retweets   **180** Quote Tweets   **13.2K** Likes



# Классы символов

- `.` - любой символ, кроме переноса строки
- `[ABC]` - один из перечисленных символов
- `[^ABC]` - любой символ, кроме перечисленных
- `[A-Z]` - любая из латинских букв в верхнем регистре
- `[a-z]` - любая из латинских букв в нижнем регистре
- `[0-9]` - любая цифра

# Классы символов

- `\w` - `[a-zA-Z0-9_]`
- `\W` - `[^a-zA-Z0-9_]`
- `\d` - `[0-9]`
- `\D` - `[^0-9]`
- `\s` - `[\f\n\r\t\v]`
- `\S` - `[^\f\n\r\t\v]`

# Квантификаторы

- $n^*$  - 0 или более символов  $n$
- $n^+$  - 1 или более символов  $n$
- $n?$  - 0 или 1 символ  $n$
- $n\{2\}$  - ровно два символа  $n$
- $n\{2,\}$  - два или более символов  $n$
- $n\{2,4\}$  - от двух до четырёх символов  $n$

# Группы

- $(...)$  - группа символов
- $(A|B)$  - A или B
- $\setminus 1, ..., \setminus 9$  - выбор ранее встреченной группы

# Позиция внутри строки

- $\wedge$  - начало
- $\$$  - конец
- $\backslash b$  - граница слова
- $\backslash B$  - не граница слова

# Стандарты

- POSIX BRE (basic regular expressions)
- POSIX ERE (extended regular expressions)
- PCRE (Perl-compatible regular expressions)

# Полезные ссылки

- Удобный инструмент для тестирования регулярок: <https://regex101.com/>
- Головоломки, связанные с регулярками: <https://regexcrossword.com/>
- Регулярка для проверки корректности почти всех почтовых адресов: <https://emailregex.com/> (кажется, сайт упал, ну и хорошо)

# Командный интерпретатор `bash` и утилита `sed`



# Команды интерпретатора

- Обычно являются внешними программами
  - Лежат в одном из каталогов в `PATH` (*echo \$PATH*, затем изучаем содержимое с *ls*)
- Но не всегда
  - Программы, изменяющие текущее окружение (например, *cd*, *export*, *ulimit*, *exit*)
- Вообще не всегда
  - Некоторые команды могут быть реализованы как встроенные (*man builtins*)

# Кавычки бывают разные

- ‘Одинарные’ - текст внутри остаётся неизменным
- `Обратные` - выполняется команда внутри и возвращается её результат
  - Того же результата можно добиться с помощью конструкции `$(command)`
- “Двойные” - допускает экранирование, подстановку переменных и вложенные обратные кавычки
  - “`$hello`” и “`\$hello`” - разные вещи

# \$

- \$? - код возврата последней команды
- \$0, ..., \$9 - аргументы, с которыми была вызвана команда
  - Как и у программ на С, нулевой аргумент - это название
- \$# - количество аргументов
- \$@ - список аргументов, начиная с первого
- \$\* - строка, содержащая список аргументов, начиная с первого

# Функции

- Могут быть объявлены одним из трёх способов:
  - *f()* (только такой вариант совместим с sh)
  - *function f()*
  - *function f*
- Переменные, объявленные внутри, становятся глобальными
  - Этого можно избежать с помощью ключевого слова *local*
- Доступ к аргументам производится так же, как и к аргументам команды
- Можно импортировать функции из соседнего скрипта
  - *. file\_name*

# Перенаправление вывода

- Перенаправить результат выполнения одной команды другой можно с помощью вертикальной черты | (получается что-то вроде композиции)
- Также можно записать в файл результат выполнения команды
  - `cmd > out.txt` - перезаписав содержимое файла
  - `cmd >> out.txt` - дописав в конец файла
- Более того, содержимое файла можно передать на поток ввода программе
  - `program < in.txt`

# Условное выполнение последовательности команд

- *cmd1 && cmd2* - вторая команда выполнится только в случае успешного выполнения первой; код возврата - это код возврата второй команды
- *cmd1 || cmd2* - вторая команда выполнится только в случае неуспешного выполнения первой; код возврата - это 0 или код возврата второй команды
- Существуют особые команды, дополняющие эти конструкции
  - *true* - код возврата всегда 0
  - *false* - код возврата всегда 1

# Условная конструкция if - then - fi

- *if* также является командой и принимает любую другую команду в качестве аргумента, за истину принимается нулевой код возврата
- Для выполнения логических операций существует конструкция *[ ... ]*
  - Поведение то же, что и для команды *test*

# Что мы можем проверить

- `$x -eq $y` - равенство
- `$x -gt (-lt, -ge, -le) $y` - различные неравенства
- `-n (-z) $s` - строка не пустая (пустая)
- `$s1 = $s2` - равенство строк
- `-e $path` - существование пути
- `-f $file` - существование файла
- `-d $dir` - существование директории
- `-x $file` - существование файла с правами на выполнение



# Цикл while - do - done

- Выглядит так же, как и конструкция if - then - fi

# Цикл for - do - done

- Используется для итерации по элементам списка
  - *for item in \$@* - простой список
  - *for filename in \*.txt* - список, генерируемый по маске файлов
- В bash и zsh существует ещё одна конструкция
  - *for (( i=0; i<10; i++ ))*

# Арифметика

- `sh` умеет вычислять только целочисленные выражения
  - Закljučаем выражение `$(( ... ))` и радуемся
- Для работы с вещественнозначными выражениями существует `bc`
  - Поддерживает тригонометрические функции, натуральный логарифм и экспоненту
  - По умолчанию арифметика целочисленная, запускать нужно с флагом: `bc -l`

# Массивы

- Инициализация: `array=(2 3 5 7 11)`
- Обращение по индексу: `${array[$i]}`
  - В bash индексация с нуля, а в zsh с единицы
- Размер массива: `${#array[@]}`

# \$ grep

- Фильтрует строки входных данных и оставляет только соответствующие шаблону
  - Для использования ERE необходимо использовать соответствующий флаг: *grep -E*

# \$ cut

- Разбивает строки по разделителю и оставляет только части, соответствующие указанным индексам
  - Разделитель указывается после флага *-d*
  - Индексы указываются после флага *-f* (индексация начинается с единицы)

# \$ sed

- Поточковый редактор, работает с последовательностью команд, разделённых символом ;
- Краткий список команд:
  - *d* - удаление
  - *a* - добавление текста после текущей позиции
  - *i* - добавление текста в текущую позицию
  - *s* - замена по шаблону
- Перед командой можно указать одну из позиций:
  - *Число* - номер строки (индексация начинается с единицы)
  - *Два~числа* - номер строки и шаг (индексация начинается с единицы)
  - *Символ \$* - последняя строка
  - */РЕГУЛЯРОЧКА/* - строки, содержание указанный шаблон
    - Как и в случае с *grep*, при использовании ERE нужен флаг *-E*

# Целочисленная и вещественная арифметика



# Целочисленные типы

- *char* - содержит в себе 1 байт (или же *CHAR\_BIT* байт)
  - Определение *CHAR\_BIT* можно взять из *<limits.h>*
  - Стандартом не определено наличие знака (можно задавать флагами компилятора)
- *short* - содержит не менее 16 бит
- *int* - содержит не менее 16 бит
- *long* - содержит не менее 32 бит
- *long long* - содержит не менее 64 бит
- *long long long* - существует только как пасхалка в *gcc*
- *int8\_t*, *int16\_t*, *int32\_t*, *int64\_t* - содержат фиксированное число бит
  - Находятся в *<stdint.h>*
  - Имеются и беззнаковые версии

# Знаковые и беззнаковые типы

- *unsigned* - беззнаковый тип
- *signed* - знаковый тип
  - Старший бит отвечает за знак
    - Если 0, то далее идёт число в прямом коде
    - Если 1, то далее идёт число в дополнительном коде:  $\sim(-number) + 1$

# Переполнение

- Ситуация, когда размера типа данных не хватает для представления числа
- В случае беззнаковых чисел эквивалентно взятию по модулю
- В случае знаковых чисел Undefined Behaviour
- В *gcc* можно использовать нестандартные функции, которые делают проверку на переполнение
  - Подробнее тут: <https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html>

$$0.1 + 0.2 = ?$$

$$0.1 + 0.2 = 0.30000000000000004$$

# Вещественнозначные типы

- *float* - 4 байта (1 бит на знак, 8 на экспоненту, 23 на мантиссу)
- *double* - 8 байт (1 бит на знак, 11 на экспоненту, 52 на мантиссу)

## Вычисление значения для float

$$\text{value} = (-1)^{\text{sign}} \times 2^{(E-127)} \times \left( 1 + \sum_{i=1}^{23} b_{23-i} 2^{-i} \right)$$

# Чтение бит вещественного числа

- С помощью указателей
  - Получаем адрес вещественного числа в памяти
  - Приводим указатель к типу *void*
  - Приводим указатель к целочисленному типу, вмещающему соответствующее число бит
- С помощью *union*
  - Создаём *union* с вещественнозначным и соответствующим по размеру целочисленным типом
- Получаем биты, с которыми мы уже умеем работать



# Специальные значения

- *PlusInf/MinusInf* - все биты в мантиссе нулевые, в экспоненте - единицы
- *MinusZero* - знаковый бит единица, остальные - нули
- *NaN* - ненулевая мантисса, все биты в экспоненте - единицы
  - *SignalingNaN* - вызывается прерывание процессора (например, деление на 0)
  - *QuietNaN* - прерывание процессора не вызывается (например, сложение бесконечностей)
  - Маски можно найти здесь: <https://vk.cc/c623XH>
- Денормализованные числа - нулевая экспонента
  - Используются для представления чисел, близких к нулю
  - Вычисляются по упомянутой формуле, но без экспоненты и неявной единицы

# Архитектура AArch64

# Cross Compilation

- Сборка программ для другой платформы (отличная архитектура или операционная система)
- Для сборки нам нужен специально обученный компилятор
  - В нашем случае готовые бинарники можно взять из проекта Linaro
- Для запуска нужен эмулятор
  - Например, *qemu*

# AArch64

- 64-битное расширение архитектуры ARM
- ARM часто встречается в телефонах и прочих небольших устройствах, в которых важно энергопотребление
  - Есть даже суперкомпьютер: <https://vk.cc/c6dhaY>
- Есть полная документация: <https://vk.cc/c6dhas>
  - Плюсы: полезно и интересно
  - Минусы: 8696 страниц и всего 1 семинар

# Целочисленные регистры

- $x0, \dots, x7$  - передача аргументов и возвращение значения из функции
- $x8, \dots, x18$  - нет гарантии, что не изменятся после вызова функции
- $x19, \dots, x28$  - гарантируется, что останутся неизменными
- $x29$  - указатель на границу фрейма функции
- $x30$  - link register ( $lr$ ), хранит адрес возврата
- $x31$  - stack pointer ( $sp$ ), указатель на вершину стека
  - Использовать можно только как  $sp$ , потому что в некоторых контекстах  $x31$  может вести себя иначе
- Регистры  $w0, \dots, w31$  - те же самые, но используют только 32 младших бита
- $xzr$  - всегда хранит ноль
- $pc$  - program counter, указатель на следующую инструкцию

# Арифметические операции

- *add dest, a, b* - сложение
- *sub dest, a, b* - вычитание
- *mul dest, a, b* - умножение
- *madd dest, a, b, c* - multiply add ( $a * b + c$ )
- *udiv dest, a, b* - unsigned div, беззнаковое деление
- *sdiv dest, a, b* - signed div, знаковое деление

# Побитовые операции

- *and dest, a, b* - побитовое И
- *eor dest, a, b* - exclusive or, побитовое исключающее ИЛИ
- *or dest, a, b* - побитовое ИЛИ
- *asr dest, a, b* - arithmetic shift right, арифметический сдвиг вправо
  - Сдвигает все биты направо, младший пропадает, старший принимает значение предыдущего старшего
- *lsl dest, a, b* - logical shift right, логический сдвиг вправо
  - Сдвигает все биты направо, младший пропадает, старший становится нулём
- *lsl dest, a, b* - logical shift left, логический сдвиг влево
  - Сдвигает все биты влево, старший пропадает, младший становится нулём

# Копирование и приведения типов

- *mov dest, a* - копирует значение регистра *a* в регистр *dest*
- *uxtb dest, a* - unsigned extend byte, берёт *uint8\_t* из *a* и добивает нулями до размера регистра *dest*
- *uxth dest, a* - unsigned extend halfword, то же, но для *uint16\_t*
- *uxtw dest, a* - unsigned extend word, то же, но для *uint32\_t*
- *sxtb, sxth, sxtw* - signed версии товарищей выше



# Флаги

- *C* - Carry, беззнаковое переполнение
- *V* - oVerflow, знаковое переполнение
- *N* - Negative, отрицательный результат
- *Z* - Zero, обнуление результата
- Для того, чтобы произошли изменения в флагах, нужно добавить к инструкциям суффикс *s*

# Метки

- Именованные относительные адреса в программе
- Переходы к ним могут быть безусловными
  - *b label* - branch, переход к метке *label*
  - *bl label* - branch and also link, переход к метке *label* с сохранением адреса возврата в *lr*
    - Чтобы вернуться, нужно использовать команду *ret*
  - *br* и *brl* для перехода к меткам, которые располагаются дальше 128Mb
- Также переходы бывают условными, на основе флагов
  - Флаги могут меняться арифметическими операциями с суффиксом *s*
  - А также командой сравнения регистров *cmp a, b*

# Условия перехода

- Добавляются к инструкции *b* в качестве суффикса
- *eq, ne* - equal / not equal
- *ge, le* - greater / less or equal
- *gt, lt* - greater / less
- *mi, pl* - minus / plus

# Взаимодействие с памятью

- *ldr dest, [a]* - прочитать содержимое по адресу *a* и сохранить результат в *dest*
- *str a, [dest]* - прочитать содержимое регистра *a* и записать по адресу *dest*
- Суффиксы *b, sb, h, sh, w, sw* можно использовать для чтения или записи операндов меньшего размера
- В AArch64 есть сахар в виде указания смещения
  - *ldr a, [b, offset]*

# Архитектура AArch64 (снова)

# Выравнивание полей структур в С

- Порядок полей соответствует порядку определения
- Размер структуры кратен размеру машинного слова (4 байта)
- Данные внутри машинных слов “прижимаются” к границам
  - В gcc можно добиться создания “упакованных” структур с помощью `__attribute__((packed))`

# Вызов функций

- В случае функций, расположенных в скомпонованной программе, всё тривиально
- В случае функций из библиотек всё немного сложнее
  - Адреса при компоновке неизвестны
  - Адреса реальных функций становятся известны только на этапе загрузки программы
  - Для того, чтобы до них добраться, используются PLT (Procedure Linkage Table) и GOT (Global Offset Table)
    - Подробнее тут: <https://habr.com/ru/post/106107/>

# Виртуальные адреса

- *ASLR* (Address Space Layout Randomization)
  - Важные структуры случайным образом меняют своё расположение в адресном пространстве процесса (сам исполняемый файл, подгружаемые библиотеки, куча, стек)
  - Библиотеки, полученные с флагом *-fPIC* являются позиционно-независимыми



## Ещё инструкции

- *adr a, label* - загружает в *a* адрес метки *label*
- Пока хватит

# Директивы препроцессора

- *.global symbol* - делает символ *symbol* видимым для линковщика
- *.space size, fill* - выделяет *size* байт, заполняя каждый из них значением *fill* (по умолчанию 0)
- *.data* - начало секции с данными
- *.section .rodata* - начало секции с read only данными
- *.text* - начало секции с кодом
- *.string str* - объявление строки
- *.byte* - объявление однобайтовой переменной
- *.hword, .word, .quad* - 2 байта, 4 байта и 8 байт соответственно
- *.macro name args ... .endm* - объявление и определение макроса
- *.equ symbol, expression* - определение символической константы

# Архитектура x86-64

# Синтаксис

- *AT&T*

- Наиболее часто встречается в Unix-системах (AT&T Bell Laboratories)
- В инструкциях сначала src, а затем dest: *movl \$5, %eax*
- Суффиксы для обозначения размера операндов (*q, l, w, b*): *addl \$4, %esp*
- Константы начинаются со знака \$, а регистры - с %
- Адресация происходит в следующем виде: *OFFSET(BASE, INDEX, SCALE)*

- *Intel*

- Наиболее часто встречается в DOS и Windows
- В инструкциях сначала dest, а затем src: *mov eax, 5*
- Размеры операндов определяются размером регистра
- Автоматически определяются константы и регистры
- Адресация происходит в следующем виде: *[BASE + INDEX \* SCALE + OFFSET]*
- Включается закливанием *.intel\_syntax noprefix*

# Целочисленные регистры (16 бит)

- General Purpose Registers
  - *AX* - accumulator
  - *BX* - base index (для работы с массивами)
  - *CX* - counter
  - *DX* - accumulator extension
  - *AH* - старший (high) байт, *AL* - младший (low) байт
- *SP* - stack pointer, указывает на вершину стека
- *BP* - base pointer, указывает на основание стека (получаем стекфрейм)
- Address Registers
  - *SI* - source index (для операций со строками)
  - *DI* - destination index (для операций со строками)
- *FLAGS* - регистр, хранящий флаги
- *IP* - instruction pointer, указывает на следующую инструкцию

# Целочисленные регистры (32 и 64 бит)

- Регистры расширены до 32 бит и начинаются с *E* (*EAX*, *EBX*, ...)
  - Обращение к младшим 16 битам выглядит так: *AX*, *BX*, ...
- Регистры расширены до 64 бит и начинаются с *R* (*RAX*, *RBX*, ...)
  - Обращение к младшим 32 битам выглядит так: *EAX*, *EBX*, ...
- *R8*, ..., *R15* - новые регистры общего назначения, не имеют какой-то определённой цели

# Инструкции

- *add DST, SRC* - сложение (*sub, and, or, xor, mov* аналогично)
- *inc (dec) DST* - инкремент (декремент)
- *neg DST* - унарный минус
- *not DST* - инверсия
- *imul (mul) SRC* - знаковое (беззнаковое) умножение *SRC* на *rax* (результат в *eax, edx*)
- *cmp DST, SRC* - сравнение (меняются флаги)
- *test DST, SRC* - побитовое сравнение через AND (меняются флаги)
- *push (pop) R* - положить на стек (взять со стека)
- *movsxd DST, SRC* - move with signed extension to dword

# Флаги

- *ZF* - zero flag, получили ноль
- *SF* - sign flag, получили отрицательное число
- *CF* - carry flag, получили перенос из старшего бита результата
- *OF* - overflow flag, получили переполнение знакового результата



# Управление ходом программы

- *jmp label* - безусловный переход к метке *label*
- *j\*\* label* - условный переход, вместо \*\* подставляется нужный суффикс:
  - *z, nz* - флаг ZF есть / нет
  - *c, nc* - флаг CF есть / нет
  - *o, no* - флаг OF есть / нет
  - *g, ge* - greater, greater or equal (для знаковых)
  - *l, le* - less, less or equal (для знаковых)
  - *a, ae* - above, above or equal (как greater, но для беззнаковых)
  - *b, be* - below, below or equal (как less, но для беззнаковых)
- *call label* - сохранение адреса возврата на стеке и переход к *label*
  - *ret* - взять адрес возврата со стека и перейти по нему
- *loop label* - уменьшает счётчик *ECX* и переходит к *label*, если он не ноль

# Соглашения о вызовах (System V AMD64 ABI)

- Используется в Linux, FreeBSD, macOS и по факту является стандартом среди Unix-подобных систем
- Первые шесть целочисленных аргументов передаются через регистры *RDI, RSI, RDX, RCX, R8, R9*
- Первые восемь вещественных аргументов передаются через регистры *XMM0, ..., XMM7*
- Всё, что не влезло в регистры, передаётся через стек справа налево
- Возвращаемое целочисленное значение передается через *RAX* (дополнительно *RDX*, если не результат не помещается в 64 бит), а вещественнозначное - через *XMM0*
- Вызываемая функция должна сохранять регистры *RBX, RSP, RBP, R12, ..., R15*
- Выравнивание стека равно 16 байтам

# Векторные вычисления и набор команд AVX

# Сопроцессор x87

- Операции над вещественными числами раньше выполняли отдельные математические сопроцессоры
- Современные процессоры в них не нуждаются, потому что в них поддержка вещественных операций и так встроена
- Взаимодействие с x87 организовано как взаимодействие со стеком

# SIMD наборы инструкций

- *SIMD* - Single Instruction Multiple Data
- *SSE* - Streaming SIMD Extension
  - Регистры *XMM0*, ..., *XMM15*, по 128 бит каждый
  - Работа с ними может происходить как в скалярном, так и в упакованном режимах
- *AVX* - Advanced Vector Extensions
  - Регистры *YMM0*, ..., *YMM15*, по 256 бит каждый
  - Существующие SSE инструкции меняют только младшие части новых регистров
  - Трёхоперандный синтаксис
  - Для большинства инструкций отсутствует требование по выравниваю в памяти
- *AVX-512*
  - Регистры *ZMM0*, ..., *ZMM15*, по 512 бит каждый

# Скалярные инструкции

- AVX инструкции имеют префикс *v*
- Суффикс определяет тип операции и точность представления
  - *ss* - Scalar Single
  - *sd* - Scalar Double
- Основные инструкции выглядят точно так же (*vaddsd*, *vsubsd*, *vmulsd*, ...)
- Существуют дополнительные инструкции для удобства
  - *vminsd*, *vmaxsd* - минимум и максимум соответственно
  - *vsqrtsd* - вычисление квадратного корня
- Можно делать и преобразования типов
  - *vcvtsd2si* - Convert Scalar Double to Scalar Integer
  - *vcvtsi2sd* - наоборот
- Сравнения
  - *vcomisd* - Compare Scalar Double and Set EFLAGS

# Векторные инструкции

- Позволяют выполнять операции сразу над несколькими числами
- Суффикс определяет тип операции и точность представления
  - *ps* - Packed Single
  - *pd* - Packed Double
- Перед описанными выше суффиксами операция *mov* может также указывать, выровнена ли память (*u* - Unaligned, *a* - Aligned)
  - Например, *vmovapd*

# Системные вызовы



# Зачем нужны системные вызовы?

- Операционная система делает для нас следующее:
  - Управляет ресурсами устройства
  - Строит и предоставляет нам абстракции для каких-то осмысленных действий
    - Работа с файлами или процессами, ввод и вывод данных
- Системный вызов - это способ попросить ядро выполнить для нас определённое действие
  - Набор системных вызовов - это интерфейс, который нам предоставляет операционная система
  - Чем богаче этот интерфейс, тем больше у операционной системы возможностей

# Примеры системных вызовов

- Управление процессом
  - *fork, waitpid, exec, exit*
- Управление файлами
  - *open, close, read, write, lseek, stat*
- Управление каталогами и файловой системой
  - *mkdir, rmdir, link, unlink, mount, unmount*
- Прочие
  - *chdir, chmod, kill, time, brk*

# Способы вызова

- x86-64 ASM
  - *rax* - номер системного вызова
  - *rdi, rsi, rdx, r10, r8, r9* - аргументы
  - Инструкция *syscall* - совершение вызова
- C
  - Для большинства системных вызовов есть C-обёртки (*man 2*)
  - Есть функция *syscall* для вызова произвольного системного вызова
- Дефайны для номеров системных вызовов лежат в `<sys/syscall.h>`

# Линковка без стандартной библиотеки

- Чтобы обеспечить себе прекрасную жизнь без стандартной библиотеки, нужно всего лишь провести линковку с флагом *-nostdlib*
- Функции *main* у нас больше нет, вход в программу - метка *\_start*

Низкоуровневый файловый ввод и вывод

# Файловый дескриптор

- Целое число, которое соответствует некоторому открытому файлу в рамках процесса
  - 0, 1 и 2 уже заняты под *stdin*, *stdout* и *stderr*
- Число файловых дескрипторов ограничено (*ulimit -a*), поэтому их нужно вовремя освобождать

# Системный вызов *open*

- Открывает существующие или создаёт новые файлы, для доступа возвращает файловый дескриптор
- Поддерживает следующие параметры открытия файла:
  - *O\_RDONLY* - только чтение
  - *O\_WRONLY* - только запись
  - *O\_RDWR* - чтение или запись
  - *O\_APPEND* - запись в конец файла
  - *O\_TRUNC* - обнуление файла
  - *O\_CREAT* - создание файла
  - *O\_EXCL* - ошибка при создании файла, если он существует

# Обработка ошибок

- Если мы знаем, что работа функции завершилась ошибкой, а сама функция использует *errno*, то вывести эту ошибку можно с помощью функции *perror*
- Не все функции используют такой подход, поэтому использование *perror* может оказаться лишним



# Атрибуты доступа к файлу

- *r (read), w (write), x (execute)*
- Доступ к файлу описывается тремя такими восьмеричными цифрами:
  - Первая - для владельца
  - Вторая - для группы владельца
  - Третья - для остальных
- Например, если владелец может всё, группа не может писать, а остальные могут только читать, то права будут закодированы так:
  - Владелец -  $rwx = 7$
  - Группа -  $r-x = 5$
  - Остальные -  $r-- = 4$
  - Итого 0 -  $0754$

# Чтение и запись в файл

- Происходит с помощью системных вызовов *read* и *write* соответственно
  - Файловый дескриптор можно получить с помощью *open*

# Системный вызов *lseek*

- Позволяет перемещать курсор по файлу
- Работает в нескольких режимах:
  - *SEEK\_SET* - явное указание позиции
  - *SEEK\_CUR* - смещение относительно текущей позиции
  - *SEEK\_END* - смещение относительно конца

# Пишем под винду 🧐

- Для компиляции Windows- программ под Linux нам понадобится кросс-компилятор, а для запуска - *wine* (это, кстати, не просто эмулятор)
  - Подробнее в ридинге:  
[https://github.com/victor-yacovlev/mipt-diht-caos/tree/master/practice/file\\_io#компиляция-и-запуск-windows-программ-из-linux](https://github.com/victor-yacovlev/mipt-diht-caos/tree/master/practice/file_io#компиляция-и-запуск-windows-программ-из-linux)
- Для типов данных существует куча тайпдефов
  - Полный список:  
<https://docs.microsoft.com/en-us/windows/win32/winprog/windows-data-types>

# Всё ещё пишем под винду 🕶️

- Аналог *open* с флагом *O\_CREAT*
  - *CreateFile*: <https://docs.microsoft.com/ru-ru/windows/win32/api/fileapi/nf-fileapi-createfilea>
- Аналоги *read* и *write*
  - *ReadFile*: <https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-readfile>
  - *WriteFile*: <https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-writefile>
- Аналог *lseek*:
  - *SetFilePointer*:  
<https://docs.microsoft.com/ru-ru/windows/win32/api/fileapi/nf-fileapi-setfilepointerex>

# Атрибуты файлов и файловых дескрипторов

# Получение метайнформации о файле

- Получить метайнформацию можно разными способами, используя следующие системные вызовы:
  - *stat* - по имени
  - *fstat* - по файловому дескриптору
  - *lstat* - по имени, без перехода по символическим ссылкам
- Эти вызовы принимают указатель на структуру *stat*, которую затем заполняют; структура имеет следующие полезные поля:
  - *st\_ino* - номер inode
  - *st\_mode* - тип файла и режим доступа
  - *st\_nlink* - число hard-ссылок
  - *st\_size* - размер в байтах
  - *man 2 stat* - остальные поля (сюда не влезли)

# Типы файлов в POSIX

- Основные
  - Регулярные файлы
  - Каталоги
  - Символические ссылки
  - Блочные устройства (например, жёсткие диски)
  - Символьные устройства (например, принтеры, терминалы)
  - Именованные каналы
  - Сокеты
- Жесткие ссылки отдельным типом не являются :(
- Тип закодирован вместе с режимом доступа в *stat.st\_mode*
  - Чтобы проверить тип, можно воспользоваться одним из соответствующих макросов:  
*S\_ISREG, S\_ISDIR, S\_ISLNK, S\_ISBLK, S\_ISCHR, S\_ISFIFO, S\_ISSOCK*



# Проверка доступа к файлу

- Для того, чтобы достать из *stat.st\_mode* права доступа, можно воспользоваться макросами вида *S\_I(R|W|X)(USR|GRP|OTH)*
  - Например, *S\_IWGRP* - получить права группы на запись
  - Также есть макросы вида *I\_RWX(U|G|O)* для проверки на всё сразу
- Идентификаторы пользователя и группы можно получить с помощью *getuid* и *getgid* соответственно
  - Эти значения как раз и сравниваются с *stat.st\_uid* и *stat.st\_gid* для проверки прав
- Удобнее всего использовать системный вызов *access*
  - Он принимает путь к файлу, а также комбинацию флагов *R\_OK*, *W\_OK*, *X\_OK* и *F\_OK* (чтение, запись, исполнение и существование соответственно)

# Удаление файла

- Удалить файл (по факту уменьшить счётчик ссылок *stat.st\_nlink*) можно с помощью системного вызова *unlink*
  - Увеличить счётчик ссылок можно с помощью системного вызова *link*
- Если счётчик уменьшается до нуля, файл можно считать удалённым
  - Однако пока у какого-нибудь процесса есть файловый дескриптор, связанный с этим файлом, процесс всё ещё может получить к нему доступ

# Символические ссылки

- Создать символическую ссылку можно с помощью системного вызова *symlink*
- Прочитать содержимое символической ссылки можно с помощью системного вызова *readlink*
- Чтобы получить метаданные самой символической ссылки, а не файла, на который она указывает, нужно использовать *lstat*

# Атрибуты файловых дескрипторов

- Системный вызов *fcntl* позволяет управлять открытыми файловыми дескрипторами
  - Команда *F\_DUPFD* - дублирование файлового дескриптора (пригодится нам позже)
  - Команда *F\_GETFL* - получение атрибутов открытия
  - Команда *F\_SETFL* - изменение атрибутов открытия
    - Менять можно далеко не всё: например, мы можем установить *O\_APPEND*, *O\_NOATIME*, *O\_NONBLOCK* (пригодится нам позже)

Отображение файлов на память

# Виртуальная память

- Проблема 1: для одновременной работы нескольких процессов каждому из них нужно своё адресное пространство, причём процессы не должны как-либо друг другу мешать
- Проблема 2: запускаемая программа может и не поместиться полностью в память
- Для решения этих проблем была изобретена виртуальная память
  - Каждая программа имеет собственное адресное пространство, разбиваемое на страницы (страница - непрерывный диапазон адресов)
  - Страницы отображаются на физическую память
  - Для запуска программы одновременное присутствие всех страниц не нужно (недостающие страницы подгружаются автоматически)

# Страничная организация памяти

- Отображением виртуальных адресов на физические занимается диспетчер памяти (Memory Control Unit)
- Виртуальное адресное пространство разбивается на страницы (обычно размером 4Кб), им соответствуют страничные блоки в физической памяти
- К сожалению, виртуальных страниц слишком много, поэтому если мы обращаемся к какой-то отсутствующей странице (*page fault*), операционной системе нужно каким-то образом освободить страничный блок, относящийся к другой виртуальной странице

# Таблицы страниц

- Для построения соответствия между виртуальными и физическими адресами используются таблицы страниц
- Записи в ней состоят примерно из следующего:
  - Номер страничного блока
  - Смещение внутри этого блока
  - Бит присутствия-отсутствия
  - Бит защиты (какой тип доступа есть к странице)
  - Бит модификации (для проверки необходимости сброса данных на диск)
  - Бит ссылки (для проверки наличия недавних обращений к странице)
  - Бит блокирования кэша (когда кэширование недопустимо)



# Проблемы

- Обращение по адресу должно быть быстрым, но мы тратим время на обработку виртуального адреса
  - Решение: буфер быстрого преобразования адреса (*Translation Lookaside Buffer, TLB*) хранит в себе часто используемые адреса
- Таблицы страниц для больших объёмов памяти оказываются огромного размера и их не хочется хранить целиком
  - Решение 1: многоуровневые таблицы страниц
  - Решение 2: инвертированные таблицы страниц
- При отсутствии у страницы страничного блока, нужно освободить какой-то другой и отдать его запрашиваемой странице
  - Решение: один из эффективных алгоритмов замещения

# Алгоритмы замещения

- Когда происходит *page fault*, операционной системе нужно выбрать, какую страницу освободить; однако в этом вопросе много тонкостей (например, не всегда нужно освобождать давно не использованную страницу), поэтому существует множество алгоритмов:
  - *Not Recently Used, NRU*: делим страницы на 4 класса по важности
  - *FIFO*: заводим список страниц и освобождаем пришедшие первыми
  - *Часы*: создаем циклический буфер и ищем первую страницу с  $R = 0$
  - *Least Recently Used, LRU*: стараемся сохранить недавно использованные
  - *Not Frequently Used, NFU*: более простая в реализации, чем LRU
  - *Алгоритм рабочего набора*
  - *WSClock*: часы + алгоритм рабочего набора

# Системный вызов *mmap*

- Выделяет в адресном пространстве процесса область по требуемому адресу
  - Может быть связана с открытым файлом
  - Может быть связана с областью оперативной памяти (флаг *MAP\_ANONYMOUS*)
- Выделение происходит постранично
  - Размер страницы можно узнать с помощью вызова *sysconf(\_SC\_PAGE\_SIZE)*
- Атрибуты доступа
  - *PROT\_READ*, *PROT\_WRITE*, *PROT\_EXEC*, *PROT\_NONE*
- Другие флаги
  - *MAP\_FIXED* - требование выделить память по указанному адресу, иначе по ближайшему
  - *MAP\_SHARED* - страницы разделяются с другими процессами, изменения в файле синхронизируются
  - *MAP\_PRIVATE* - страницы с другими процессами не разделяются, файлы менять нельзя

# Системный вызов *munmap*

- Освобождает страницы, выделенные системным вызовом *mmap*

Запуск и завершение работы процессов

# Процессы

- Процесс - это абстракция над железом, которая делает возможным параллельное выполнение программ
  - Даже если у вас всего один физический процессор
- У каждого процесса есть свои:
  - Идентификатор процесса (Process ID, PID)
    - Их число ограничено, за счёт этого и работает форк-бомба
  - Виртуальное адресное пространство
  - Файловые дескрипторы
- Очень важно: потоки и процессы - это разные понятия

# Иерархия процессов

- Между процессами может существовать связь родитель-ребёнок
- Корнем этого дерева можно считать процесс с *PID=1*
  - В случае, если дети какого-то процесса “осиротеют”, они могут стать детьми этого процесса
- Процессы можно объединять в группы процессов (например, все процессы, запущенные из одного окна терминала)
- Группы процессов можно объединять в сеансы (например, вход пользователя в систему)

# Реализация процессов

- Операционная система поддерживает записи о процессах в *таблице процессов*
- Примеры полей в этой таблице
  - Управление процессом: регистры, pid, родитель, группа, используемые ресурсы
  - Управление памятью: указатели на сегменты
  - Управление файлами: рабочий каталог, файловые дескрипторы, uid, gid
- Переключением процессов занимается *планировщик*



# Системный вызов *fork*

- Системный вызов создаёт новый процесс, при этом дочерний процесс является точной копией родительского
  - Отличить их можно по возвращаемому значению *fork* (а он возвращает *pid*)
    - *pid* == 0 - дочерний процесс
    - *pid* > 0 - родительский процесс
    - Иначе - произошла ошибка (можем вывести через *perror*)
- При завершении процессы должны вызвать системный вызов *exit* с кодом возврата (число от 0 до 255)
  - 0 - обычно означает успешное завершение
  - Ненулевой код возврата по сути хранит в себе какую-то информацию (например, ошибку)

# Системный вызов *wait*

- Родительский процесс обязан получит информацию о завершении работы дочерних процессов
  - Если этого не делать, дочерние процессы перейдут в состояние *Zombie* и будут просто занимать место в таблице процессов
- Сделать это можно с помощью одного из системных вызовов:
  - *wait* - дождаться завершения произвольного дочернего процесса
  - *waitpid* - дождаться завершения дочернего процесса с определённым *pid*
  - *wait3*, *wait4* - аналогично первым двум, но можно также получить информацию об использовании процессом ресурсов
- Если дочерних процессов несколько, то лучше пользоваться *waitpid* или *wait4*, поскольку порядок завершения работы заранее не определён

# Получение информации от дочернего процесса

- Только что упомянутые системные вызовы позволяют также получать информацию от дочернего процесса, которую можно декодировать с помощью макросов:
  - *WIFEXITED(wstatus)* - ненулевое значение, если процесс был завершён системным вызовом *exit*
  - *WIFSIGNALED(wstatus)* - ненулевое значение, если процесс был завершён с помощью сигнала
  - *WEXITSTATUS(wstatus)* - код возврата дочернего процесса
  - *WTERMSIG(wstatus)* - номер сигнала, с помощью которого был завершён процесс

Запуск программ через fork-exec

# Системный вызов `exec`

- Позволяет подменить образ текущего процесса на другую программу, при этом сохранив атрибуты процесса
  - Обычно сначала делают `fork`, а затем в дочернем процессе вызывают `exec`
- Есть несколько различных вариантов этого вызова, различить их можно с помощью суффиксов:
  - `l` - передаётся переменное число аргументов (как, например, в `printf`), в конце `NULL`
  - `v` - передаётся массив аргументов, в конце `NULL`
  - `e` - дополнительно передаются переменные окружения
  - `p` - найти программу в переменной `PATH`
- При успешном выполнении `exec` мы уже имеем дело с другой программой
  - Поэтому маркером наличия ошибки может служить выполнение следующей за `exec` строки

Копии файловых дескрипторов и  
неименованные каналы

# Системный вызов *fcntl*

- Производит манипуляции над файловыми дескрипторами
- Мы уже говорили про него [ранее](#)

```
#include <fcntl.h>
```

```
#include <unistd.h>
```

```
int fcntl(int fd, int cmd);
```

```
int fcntl(int fd, int cmd, long arg);
```

# Дублирование файловых дескрипторов

- Ищем свободный  $\geq arg$  дескриптор и делаем его копией *fd*
- Копия разделяет с оригиналом
  - исходный файл
  - смещение внутри него
  - права доступа к нему
  - флаги
- У копии всегда опущен флаг *CLOEXEC*

```
int fd = open("cats.txt", O_RDONLY);  
int new_fd = fcntl(fd, F_DUPFD, 42);
```



# Дублирование файловых дескрипторов (чутка иначе)

- В POSIX есть более удобные способы дублирования файловых дескрипторов:
  - *dup* - ищет для копии наименьший свободный дескриптор
  - *dup2* - даёт копии указанный дескриптор (даже если его перед этим надо закрыть)
- Свойства у копии такие же, как и в случае использования *F\_DUPFD*

```
#include <unistd.h>
```

```
int dup(int fd);
```

```
int dup2(int fd, int new_fd);
```

# Способы взаимодействия между процессами

- Обычные файлы (люди делали так до создания АКОСа)
- Сигналы (пройдём позже)
- Сигналы реального времени (пройдём позже)
- Неименованные каналы
- Именованные каналы
- Сокеты (пройдём позже)
- Разделяемая память (пройдём позже)

# Системный вызов *pipe*

- Создаёт пару связанных файловых дескрипторов, называемых *неименованным каналом* или *pipe*
- Первый дескриптор предназначен только для чтения, а второй — только для записи
- Работать с ними можно с помощью знакомых нам уже системных вызовов *read* и *write*, но нужно помнить некоторые нюансы
- Канал не резиновый, у него ограниченный размер

```
#include <unistd.h>

int pipe(int pipefd[2]);

// Пример создания
int pipefd[2];
pipe(pipefd);
```

# Ситуации при записи

- В буфере есть место под новые данные
  - *write* пишет данные и завершает работу
- В буфере нет места
  - *write* блокируется до тех пор, пока в буфере не появится достаточно места
- Нет ни одного читателя
  - *write* завершает работу с ошибкой *Broken Pipe*

# Ситуации при чтении

- В буфере есть данные
  - *read* читает данные и завершает работу
- Буфер пустой, есть хотя бы один писатель (!!!)
  - *read* блокируется до тех пор, пока в буфер что-то не положат
- Буфер пустой, нет писателей
  - *read* завершает свою работу и возвращает 0

# Deadlock при чтении

- Давайте вспомним, что происходит с файловыми дескрипторами при *fork*
  - Правильно, они копируются
- Если мы не закроем копию дескриптора на запись, то дескриптор на чтение будет ждать записи, которой никогда не случится



```
const char msg[] = "026 is the best";

int pipefd[2];
pipe(pipefd);

if (fork() != 0) {
    write(pipefd[1], msg, sizeof(msg));
    close(pipefd[1]);

    char buffer[4096];
    while (read(pipefd[0], buffer, sizeof(msg)) != 0);
} else {
    // Нереальный флекс
    while(1) sched_yield();
}
```

# Характеристики каналов

- У каналов есть как минимум две числовые характеристики:
  - Размер буфера
  - Гарантированный блок для атомарной записи
- POSIX гарантирует, что они будут составлять не менее 512 байт
- Обычно буфер имеет размер 64Кбайт и записать в него можно атомарно 4Кбайт
  - Размер буфера можно менять

```
// Размер атомарной записи
```

```
#include <limits.h> // PIPE_BUF
```

```
// Получение и изменение размера буфера
```

```
int pipefd[2];
```

```
pipe(pipefd);
```

```
int pipe_size = fcntl(pipefd[1], F_GETPIPE_SZ);
```

```
fcntl(pipefd[1], F_SETPIPE_SZ, 1337);
```

# Неблокирующий ввод-вывод

- Если мы не хотим блокироваться при записи или чтении, то мы можем использовать неблокирующий ввод-вывод
- В таком случае вместо блокировки соответствующие системные вызовы будут возвращать -1 и писать *EAGAIN* в *errno*

```
// Можем указать при открытии
```

```
int fd = open("nonblock.me", O_RDONLY | O_NONBLOCK);
```

```
// Можем указать и потом
```

```
int fd = open("nonblock.me", O_RDONLY);
```

```
int old_flags = fcntl(fd, F_GETFL);
```

```
fcntl(fd, F_SETFL, old_flags | O_NONBLOCK);
```



# Системный вызов *mkfifo*

- Создаёт *именованный канал*
- Отличается от неименованного канала только созданием и открытием, поведение при взаимодействии то же самое
- Решает следующую проблему: неименованные каналы можно использовать либо внутри одного процесса, либо между процессом и его наследниками
- FIFO является отдельным типом файла

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
int mkfifo(const char* path, mode_t mode);
```

Сигналы

# Сигналы

- Ещё один способ взаимодействия между процессами
- Представляет из себя асинхронную отправку коротких сообщений
  - Очень коротких: процесс получает только номер сигнала
- Текущий процесс может получить сигнал от:
  - Ядра
  - Другого процесса того же пользователя
  - Другого процесса пользователя root
  - Самого себя

# Поведение при получении сигнала

- В зависимости от полученного сигнала процесс может по-разному на него реагировать:
  - *Term* - завершиться
  - *Ign* - проигнорировать получение сигнала
  - *Core* - завершиться и сгенерировать core dump
  - *Stop* - приостановить выполнение
  - *Cont* - возобновить выполнение, если оно было приостановлено

# Системный вызов *alarm*

- Заводит таймер по истечении которого процесс получает сигнал *SIGALRM*
- Мы можем отменить текущий таймер и получить количество секунд, оставшихся до доставки *SIGALRM*
- По умолчанию процесс завершает работу после получения сигнала (*Term*)

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);

// Ждём SIGALRM через 10 секунд
alarm(10);

// Можем отменить предыдущий таймер
uint32_t time = alarm(0);
```

# Функция *abort*

- Провоцирует получение сигнала *SIGABRT*
- Невозможно заблокировать (об этом позже)
- По умолчанию процесс завершает работу и генерирует core dump после получения сигнала (*Core*)

```
#include <stdlib.h>
```

```
void abort(void);
```

```
// Прерываем исполнение  
abort();
```

```
// Используется в ассертах  
#include <assert.h>
```

```
assert(2 + 2 == 5);
```

# Сигнал *SIGPIPE*

- Процесс получает этот сигнал, когда пытается что-то написать в канал, на другом конце которого нет читателей
- По умолчанию процесс завершает работу после получения сигнала (*Term*)

```
// Создаем канал
int pipefd[2];
pipe(pipefd);

// Закрываем единственного читателя
close(pipefd[0]);

// Ша как напишем...
const char msg[] = "Hello my only friend!";
write(pipefd[1], msg, sizeof(msg));

// Broken pipe...
```

## Другие примеры сигналов

- *SIGKILL (Term)* – пора завершать работу (нельзя проигнорировать)
- *SIGSEGV (Core)* – ошибка, связанная с памятью
- *SIGTERM (Term)* – пора завершать работу
- *SIGCHLD (Ign)* – завершился дочерний процесс
- *SIGSTOP (Stop)* – Ctrl+Z (нельзя проигнорировать)
- *SIGCONT (Cont)* – возобновление работы
- *SIGURG (Ign)* – в сокете появились важные для чтения данные
- *SIGINT (Term)* – Ctrl+C
- *SIGQUIT (Core)* – Ctrl+\



# Завершение дочернего процесса

- Считается, что завершённый сигналом процесс не имеет кода возврата
- Родительский процесс может обработать любую из этих двух ситуаций с помощью макросов *WIFEXITED* и *WIFSIGNALED*

```
pid_t pid = fork();

if (pid > 0) {
    int status = 0;
    waitpid(pid, &status, 0);

    if (WIFEXITED(status)) {
        // Дочерний процесс был завершён через exit
        int return_code = WEXITSTATUS(status);
    }

    if (WIFSIGNALED(status)) {
        // Дочерний процесс был завершён сигналом
        int signum = WTERMSIG(status);
    }
}
```

# Системный вызов *kill*

- Отправляет сигнал определённому процессу или группе процессов
- Несмотря на название, отправляет не только *SIGKILL*
- В качестве номеров системных вызовов стоит использовать предопределённые константы

```
#include <signal.h>
#include <sys/types.h>

int kill(pid_t pid, int sig);

// Определённый процесс
kill(1337, SIGKILL);

// Процессы текущей группы
kill(0, SIGKILL);

// Процессы текущего пользователя
kill(-1, SIGKILL);

// Процессы группы pid
int pid = ...;
kill(-pid, SIGKILL);
```

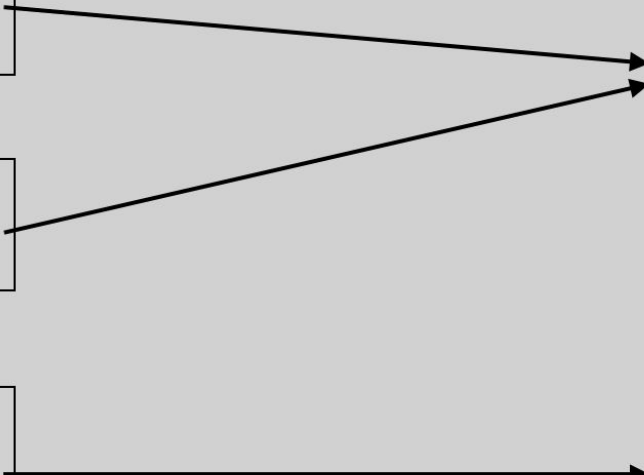
# Маска доставки сигналов

- Перед тем, как быть обработанным, сигнал фиксируется в маске доставки сигналов
- Даже если он туда попал, не факт, что он обработается ровно в этот момент: сигнал может быть заблокированным (об этом позже)
- Маска способна хранить только факт наличия сигнала, но не их количество
  - Если процессу прилетела парочка *SIGINT*, то сохранится информация только о первом
- Если процесс получил несколько сигналов, то порядок их обработки не определён
- Не наследуется при *fork*

**Process 1**  
`kill(SIGINT, PID)`

**Process 2**  
`kill(SIGINT, PID)`

**Process 3**  
`kill(SIGTERM, PID)`



0	SIGHUP
1	SIGINT
0	SIGQUIT
0	SIGILL
0	SIGABRT
0	SIGFPE
1	SIGTERM

# Множества сигналов

- Набор сигналов можно закодировать в специально обученной структуре

```
#include <signal.h>

// Пустое множество
sigemptyset(sigset_t* set);

// Полное множество
sigfillset(sigset_t* set);

// Добавить сигнал
sigaddset(sigset_t* set, int signum);

// Удалить сигнал
sigdelset(sigset_t* set, int signum);

// Проверить наличие сигнала
sigismember(sigset_t* set, int signum);
```

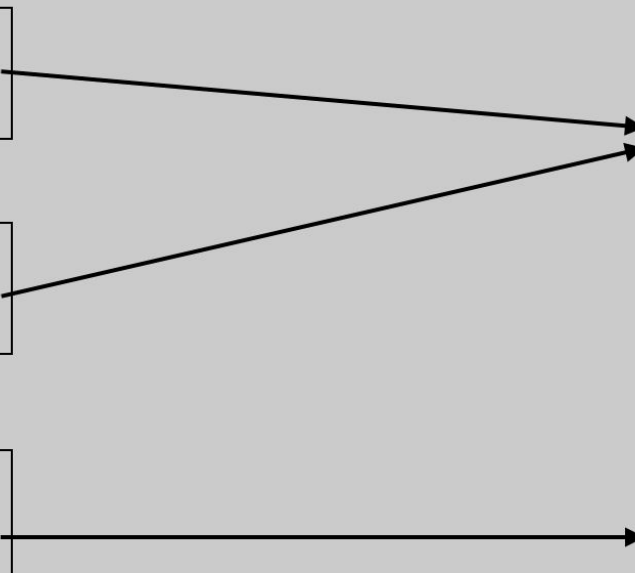
# Маска блокировки сигналов

- Поскольку обработка сигналов прерывает исполнение, иногда хочется обезопасить участок кода и временно не обращать внимание на определённые сигналы; реализуется это с помощью маски блокировки
- При этом просто игнорировать сигналы не очень хорошо, поэтому после разблокировки заблокированные сигналы будут доставлены
- *SIGSTOP* и *SIGKILL* заблокировать нельзя
- Маска блокировки наследуется при *fork*

**Process 1**  
`kill(SIGINT, PID)`

**Process 2**  
`kill(SIGINT, PID)`

**Process 3**  
`kill(SIGTERM, PID)`



1	0	SIGHUP
0	0	SIGINT
1	0	SIGQUIT
1	0	SIGILL
1	0	SIGABRT
1	0	SIGFPE
1	1	SIGTERM

# Системный вызов *sigprocmask*

- Позволяет взаимодействовать с маской блокировки сигнала
- Есть три режима работы *how*:
  - *SIG\_SETMASK*
    - Установить маску целиком
  - *SIG\_BLOCK*
    - Заблокировать из маски
  - *SIG\_UNBLOCK*
    - Разблокировать из маски
- *old\_set* можно занулить, если эта информация нам не нужна

```
#include <signal.h>

int sigprocmask(int how, sigset_t* set,
sigset_t* old_set);

// Инициализируем маску
sigset_t blocked;
sigemptyset(&blocked);

// Добавляем в неё сигнал
sigaddset(&blocked, SIGINT);

// Устанавливаем маску
sigprocmask(SIG_SETMASK, &blocked, 0);
```



# Обработка сигналов

- Для всех сигналов, кроме *SIGSTOP* и *SIGKILL* можно зарегистрировать программные обработчики
- Сделать это можно с помощью системного вызова *signal*, однако он не является кроссплатформенным, поэтому мы его опустим (почитать можно в ридинге)
- В связи с фактом выше, пользоваться мы будем *sigaction*

# Структура *sigaction*

- Является одним из аргументов одноимённого системного вызова
- Нам будет интересно поле *sa\_handler* – указатель на функцию обработчик
  - В качестве аргумента принимает номер сигнала
- Примеры флагов:
  - *SA\_RESTART*
  - *SA\_SIGINFO*
  - *SA\_RESETHAND*

```
#include <signal.h>

struct sigaction {
    void      (*sa_handler) (int);
    void      (*sa_sigaction) (int,
siginfo_t*, void*);
    sigset_t   sa_mask;
    int        sa_flags;
    void      (*sa_restorer) (void);
};
```

# Системный вызов *sigaction*

- Позволяет устанавливать действие при обработке сигнала:
  - Указатель на обработчик
  - *SIG\_DFL* – по умолчанию
  - *SIG\_IGN* – игнорировать
- *oldact* можно занулить, если эта информация не нужна

```
#include <signal.h>

int sigaction(int signum, const struct
sigaction* act, struct sigaction* oldact);

// Простейший обработчик
void foo(int signum) { }

// Инициализируем sigaction
struct sigaction act;
act.sa_handler = foo;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;

// Устанавливаем обработчик на SIGINT
sigaction(SIGINT, &act, NULL);
```

# Атомарность

- Существует целочисленный тип, который гарантирует атомарность чтения и записи при переключении исполнения программы и обработчика сигнала

```
#include <signal.h>
```

```
sig_atomic_t counter = 0;
```

# Signal Safety

- Обработчики сигналов должны быть реализованы максимально просто
- Далеко не все функции можно безопасно в них использовать
  - Как правило, системные вызовы входят в список допустимых
  - Такие функции как *printf* запрещены из-за наличия внутреннего буфера
  - За подробностями и списку можно обратиться к *man 7 signal-safety*

Сигналы реального времени

# Real-Time Signals

- Стандартные сигналы имеют три больших недостатка:
  - Отсутствие порядка доставки
  - Отсутствие поддержки числа сигналов
  - Между процессами можно передать лишь номер сигнала
- На помощь приходят сигналы реального времени
  - Гарантируют порядок доставки
  - Соответственно поддерживают несколько сигналов одного типа
  - Позволяют передать дополнительную информацию
- Для любых наших нужд нам доступны сигналы с *SIGRTMIN* до *SIGRTMAX*
  - По умолчанию их поведение – *Term*

# Функция *sigqueue*

- Добавляет сигнал в очередь
- Через структуру *sigval* можно передать дополнительную информацию
  - Немного, но лучше, чем было
- Получить эту информацию можно с помощью обработчика с тремя аргументами

```
#include <signal.h>
```

```
union sigval {  
    int    sival_int;  
    void*  sival_ptr;  
};
```

```
int sigqueue(pid_t pid, int sig, const  
union sigval value);
```



# Отправка и получение RTS со значением

// Отправка

// Инициализируем значение для отправки

```
sigval_t send_value;
```

```
send_value.sival_int = 2326;
```

// Отправляем сигнал процессу pid

```
sigqueue(pid, SIGRTMIN+1, send_value);
```

// Получение

```
void handler(int signum, siginfo_t* info,  
void* ucontext) {
```

```
    sigval_t value = info->si_value;
```

```
    // Используем как хотим
```

```
    value.sival_int;
```

```
}
```

// Инициализируем sigaction

```
struct sigaction act;
```

```
act.sa_sigaction = handler;
```

```
sigemptyset(&act.sa_mask);
```

```
act.sa_flags = SA_SIGINFO;
```

// Регистрируем обработчик

```
sigaction(SIGRTMIN+1, &act, NULL);
```

# Сокеты TCP/IP

# Модель TCP/IP

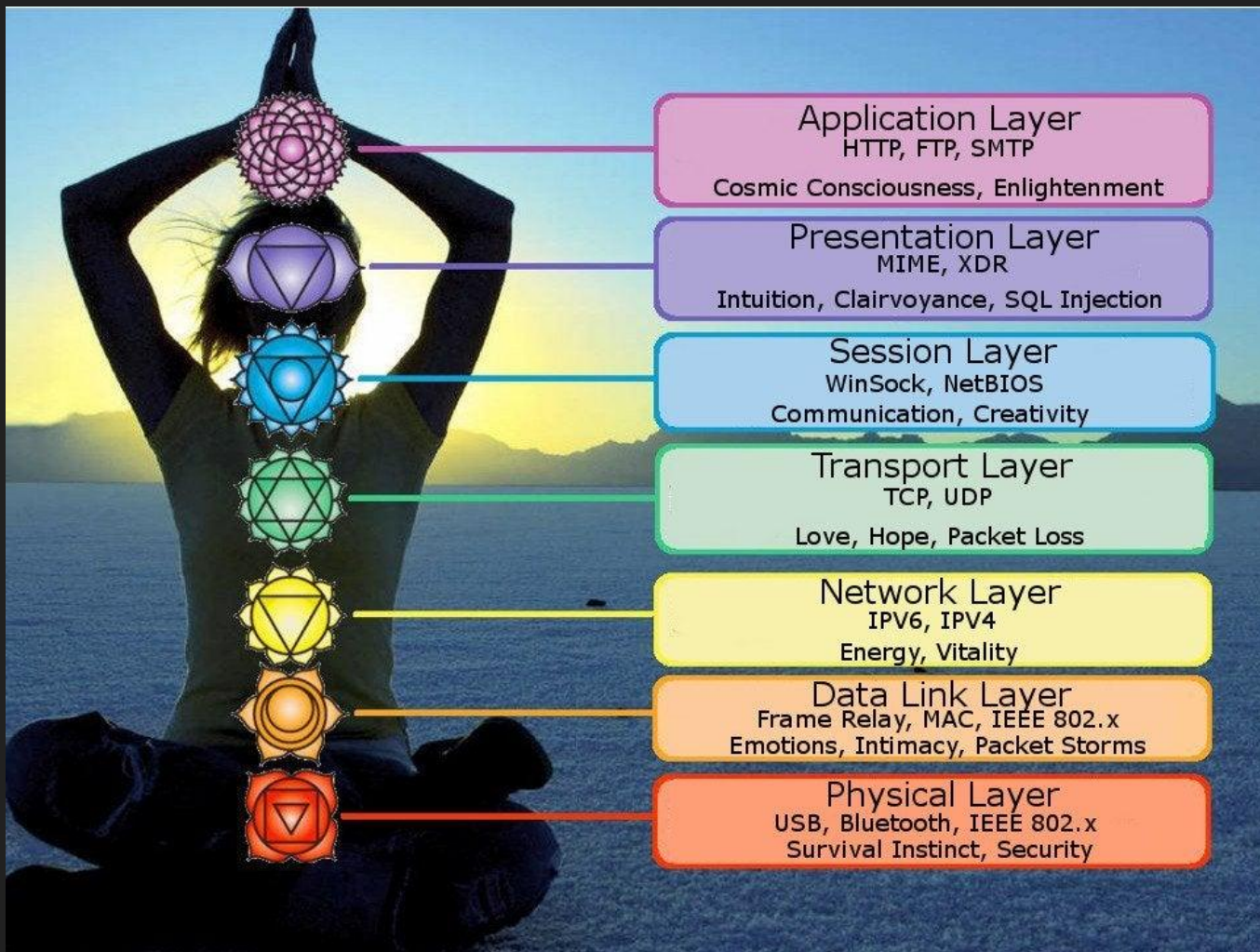
- Описывает передачу данных от источника информации к получателю
- Содержит в себе 4 уровня: прикладной, транспортный, межсетевой и канальный
- Можно считать, что по этой модели работает практически весь Интернет

<b>Прикладной</b> (Application layer)	напр., HTTP, RTSP, FTP, DNS
<b>Транспортный</b> (Transport Layer)	напр., TCP, UDP, SCTP, DCCP <i>(RIP, протоколы маршрутизации, подобные OSPF, что работают поверх IP, являются частью сетевого уровня)</i>
<b>Сетевой (Межсетевой)</b> (Network Layer)	Для TCP/IP это IP <i>(вспомогательные протоколы, вроде ICMP и IGMP, работают поверх IP, но тоже относятся к сетевому уровню; протокол ARP является самостоятельным вспомогательным протоколом, работающим поверх канального уровня)</i>
<b>Уровень сетевого доступа</b> <b>(Канальный)</b> (Link Layer)	Ethernet, IEEE 802.11 WLAN, SLIP, Token Ring, ATM и MPLS, физическая среда и принципы кодирования информации, T1, E1

# Модель OSI

- *Open Systems Interconnection model*
- Описывает различные уровни взаимодействия устройств по сети
- В такой модели каждый уровень имеет свою функцию и выполняет только её
- Самый верхний уровень самый абстрактный, а самый нижний, наоборот, описывает физические свойства взаимодействия
- Если планируете работать с сетями, то это классика, это знать надо

Модель					
Уровень (layer)		Тип данных (PDU <sup>[15]</sup> )	Функции	Примеры	Оборудование
Host layers	7. Прикладной (application)	Данные	Доступ к сетевым службам	HTTP, FTP, POP3, WebSocket	Хосты (клиенты сети), Межсетевой экран
	6. Представления (presentation)		Представление и шифрование данных	ASCII, EBCDIC, JPEG, MIDI	
	5. Сеансовый (session)		Управление сеансом связи	RPC, PAP, L2TP, gRPC	
	4. Транспортный (transport)	Сегменты (segment) / Датаграммы (datagram)	Прямая связь между конечными пунктами и надёжность	TCP, UDP, SCTP, Порты	
Media <sup>[16]</sup> layers	3. Сетевой (network)	Пакеты (packet)	Определение маршрута и логическая адресация	IPv4, IPv6, IPsec, AppleTalk, ICMP	Маршрутизатор, Сетевой шлюз, Межсетевой экран
	2. Канальный (data link)	Биты (bit)/ Кадры (frame)	Физическая адресация	PPP, IEEE 802.22, Ethernet, DSL, ARP, сетевая карта.	Сетевой мост, Коммутатор, точка доступа
	1. Физический (physical)	Биты (bit)	Работа со средой передачи, сигналами и двоичными данными	USB, RJ («витая пара»), коаксиальный, оптоволоконный), радиоканал	Концентратор, Повторитель (сетевое оборудование)



# Протокол HTTP

- *HyperText Transfer Protocol*
- Протокол прикладного уровня, предназначенный для передачи данных
- Общение происходит в режиме клиент-сервер
- Примеры использования:
  - Веб-сайты
  - HTTP-ручки у различных API
- *HTTPS (HTTP Secure)* – версия протокола с поддержкой шифрования
  - Как правило, защита реализуется по протоколу *TLS (Transport Layer Security)*



# HTTP запрос

Метод URI HTTP/Версия

(empty line)

GET /cat.jpg HTTP/1.1

(empty line)

# HTTP запрос с заголовками

Метод URI HTTP/Версия

Заголовок: Значение заголовка

(empty line)

GET /cat.jpg HTTP/1.1

Host: example.com

DNT: 1

(empty line)

# HTTP запрос с заголовками и телом

```
GET /cat.jpg HTTP/1.1
```

```
Host: example.com
```

```
DNT: 1
```

```
Content-Length: 13
```

```
(empty line)
```

```
HELLO, WORLD!
```

# HTTP ответ

HTTP/Версия	Код	ответа	Пояснение
-------------	-----	--------	-----------

(empty line)

HTTP/1.1	200	OK	
----------	-----	----	--

(empty line)

HTTP/1.1	404	Not Found	
----------	-----	-----------	--

(empty line)

# HTTP ответ с заголовками и телом

HTTP/1.1 200 OK

Server: nginx/1.2.3

Content-Length: 13

(empty line)

HELLO, WORLD!



200  
OK

[http.cat](http://cat)

# Протокол UDP

- *User Datagram Protocol*
- Протокол транспортного уровня, предназначенный для доставки данных
- Для отправки датаграммы даже не нужно устанавливать соединение
- Датаграмма может не прийти, прийти дважды, прийти в неправильном порядке, но если она придёт, то гарантируется, что придёт она целая
- Используется в системах, чувствительных ко времени:
  - Серверы, отвечающие на кучу небольших запросов
  - Стриминг
  - Онлайн-игры
- *Я бы рассказал вам шутку про UDP, но боюсь, что она до вас не дойдёт*

# Протокол TCP

- *Transmission Control Protocol*
- Протокол транспортного уровня, предназначенный для доставки данных
- Требует предварительной установки соединения
- Гарантирует надежную передачу сегментов без потерь в правильном порядке
- Приличные пользователи протокола должны закрывать соединение
- Используется когда критична сохранность данных



# Протокол IP

- *Internet Protocol*
- Протокол сетевого уровня, предназначенный для маршрутизации
- IPv4
  - Каждому узлу сети ставится в соответствие IP-адрес длиной 4 байт
    - Пример: 74.125.205.113
    - localhost: 127.0.0.1
- IPv6
  - В связи с проблемой исчерпания адресов IPv4, необходим переход на IPv6
  - Каждому узлу сети ставится в соответствие IP-адрес длиной 16 байт
    - Пример, fe80:0:0:0:200:f8ff:fe21:67cf

# Протокол ARP

- *Address Resolution Protocol*
- Протокол канального уровня, предназначенный для определения MAC-адреса компьютера по его IP-адресу
  - Пример: F0:98:9D:1C:93:F6
- Пока просто как пример, веселье будет в конце модуля

# Сокеты

- Файловые дескрипторы, предназначенные как для чтения, так и для записи
- Могут использоваться для взаимодействия:
  - Разных процессов на одном хосте
  - Разных процессов на разных хостах
- *man 7 socket*

# Системный вызов *socket*

- Возвращает файловый дескриптор сокета
- *domain* – семейство протоколов для взаимодействия
  - *AF\_UNIX* – локальное
  - *AF\_INET* – IPv4
  - *AF\_INET6* – IPv6
  - *AF\_PACKET* – OSI 2
- *type* – семантика взаимодействия
  - *SOCK\_STREAM* – TCP
  - *SOCK\_DGRAM* – UDP
  - *SOCK\_RAW*

```
#include <sys/socket.h>
```

```
#include <sys/types.h>
```

```
int socket(int domain,  
           int type,  
           int protocol);
```

```
// Создаём сокет
```

```
int socket_fd = socket(AF_INET,  
                      SOCK_STREAM,  
                      0);
```

# Системный вызов *socketpair*

- Создаёт пару сокетов, которыми можно пользоваться как каналами, но дескрипторы предназначены и для записи, и для чтения
- Сокеты поддерживают обработку события закрытия соединения (увидим позже)

```
#include <sys/socket.h>
#include <sys/types.h>

int socketpair(int domain, int type,
               int protocol, int sv[2]);

// Создание пары сокетов
int sv[2];
socketpair(AF_UNIX, SOCK_STREAM, 0, sv);

// Или так
int sv[2];
socketpair(AF_UNIX, SOCK_DGRAM, 0, sv);
```

# Роль TCP-клиента

- В наших задачах TCP-клиент будет иметь следующий увлекательный жизненный цикл:
  - Создание сокета с помощью *socket* (сокет пока не готов к взаимодействию)
  - Подключение к серверу с помощью *connect*
  - Общение с сервером через сокет с помощью *read* и *write*
  - Закрытие соединения
    - Сегмент с флагом *FIN* (с точки зрения протокола)
    - Мы можем использовать системный вызов *shutdown*
    - Соединение мог закрыть сервер, тогда *read* нам вернёт 0 или -1
  - Закрытие сокета

# Структура `sockaddr_in`

- Задаёт адрес в сети IPv4
- Значение полей:
  - *sin\_family* – семейство
  - *sin\_port* – номер порта в сетевом порядке байт
  - *sin\_addr* – адрес в сетевом порядке байт
- Перевести номер порта в сетевой порядок байт можно с помощью функции *htons*
- Перевести адрес из текстового вида в сетевой порядок байт можно с помощью функции *inet\_addr*

```
#include <netinet/in.h>
#include <sys/socket.h>

struct sockaddr_in {
    sa_family_t    sin_family;
    in_port_t      sin_port;
    struct in_addr sin_addr;
};

// Инициализируем структуру для
// подключения к серверу
struct sockaddr_in ipv4_addr = {
    .sin_family = AF_INET,
    .sin_port = port,
    .sin_addr = ip};
```

# Системный вызов *connect*

- Связывает сокет *sockfd* с адресом *addr*
- Есть несколько видов *addr*:
  - *sockaddr\_un* (*AF\_UNIX*)
  - *sockaddr\_in* (*AF\_INET*)
  - *sockaddr\_in6* (*AF\_INET6*)

```
#include <sys/socket.h>
#include <sys/types.h>

int connect(int sockfd,
            const struct sockaddr* addr,
            socklen_t addrlen);

// Подключаемся к серверу
connect(socket_fd,
        (struct sockaddr*)&ipv4_addr,
        sizeof(ipv4_addr));
```



# Системный вызов *shutdown*

- Оповещает противоположную сторону о закрытии соединения
- Завершать TCP соединение нужно именно таким образом
- Параметр *how* может принимать следующие значения:
  - *SHUT\_RD*
  - *SHUT\_WR*
  - *SHUT\_RDWR*

```
#include <sys/socket.h>

int shutdown(int sockfd, int how);

// Закрываем соединение и сокет
shutdown(socket_fd, SHUT_RDWR);
close(socket_fd);
```

# Роль TCP-сервера

- В наших задачах TCP-сервер совершает следующие действия:
  - Создание сокета с помощью *socket* (сокеты пока не готовы к взаимодействию)
  - Ассоциация сокета с некоторым адресом с помощью *bind*
  - Создание и поддержка очереди входящих подключений с помощью *listen*
  - Соединение с клиентом с помощью *accept* и дальнейшее общение
  - Завершение общения с клиентом с помощью *shutdown*
  - Завершение работы сервера и закрытие сокета сервера

# СИСТЕМНЫЙ ВЫЗОВ *bind*

- Связывает сокет с адресом
- Если указать 0.0.0.0, то будем слушать все адреса (в случае, если на устройстве больше одного IP)

```
#include <sys/socket.h>
#include <sys/types.h>

int bind(int sockfd,
         const struct sockaddr* addr,
         socklen_t addrlen);

// Связываем сокет с адресом
bind(socket_fd,
      (struct sockaddr*)&ipv4_addr,
      sizeof(ipv4_addr));
```

# Системный вызов *listen*

- Позволяет использовать сокет для принятия входящих соединений
- Размер очереди *backlog* ограничен, максимальное значение записано в константе *SOMAXCONN*

```
#include <sys/socket.h>
#include <sys/types.h>

int listen(int sockfd, int backlog);

// Создаём очередь на входящие
// подключения
listen(socket_fd, SOMAXCONN);
```

# Системный вызов *accept*

- Принимает одно входящее соединение
- Вторым и третьим аргументы содержат информацию о клиенте, если нам всё равно, пишем *NULL*
- В случае успешного завершения возвращает файловый дескриптор, через который будет происходить общение с клиентом
- Если входящие подключения отсутствуют, то вызов блокируется

```
#include <sys/socket.h>
```

```
#include <sys/types.h>
```

```
int accept(int sockfd,  
           struct sockaddr* addr,  
           socklen_t* addrlen);
```

```
// Получаем сокет для работы с клиентом  
int client_fd = accept(socket_fd, NULL,  
NULL);
```

# Мультиплексирование ввода-вывода

# Неблокирующий ввод-вывод

- Как его включить обсуждали [ранее](#)
- Системные вызовы *read*, *write*, *assert* не блокируются, а возвращают *-1*, при этом записывая значение *EAGAIN* в *errno*

```
while ((read_result =  
        read(fd, buffer, 4096)) > 0) {  
    // Успешное чтение, можно ещё раз  
}  
  
if (read_result == -1 && errno == EAGAIN) {  
    // Пока что новых данных нет  
  
} else if (read_result == 0) {  
    // EOF  
  
} else {  
    perror("Unhandled situation");  
}
```

# C10k Problem

- В какой-то момент компьютеры достигли достаточной мощности для поддержания 10\_000 соединений в секунду, однако bottleneck'ом выступала алгоритмическая часть
- Для того, чтобы получать информацию об обновлениях дескрипторов, их нужно было полностью обойти (например, механизм *poll*), то есть  $O(N)$  времени
- Этот подход улучшили и получили решение, которое за  $O(1)$  позволяет получить ready list, – *epoll* (в FreeBSD есть аналог *kqueue*)
- Компьютеры снова стали лучше, теперь актуальна проблема C10m, которая решается эффективным масштабированием и управлением памятью



# Механизм *epoll*

- Предназначен для наблюдения за файловыми дескрипторами и получения информации о связанных с ними событиях
- Предполагаемый сценарий использования:
  - Регистрируем наш *epoll* с помощью *epoll\_create*
  - Добавляем в *epoll* интересующие нас дескрипторы с помощью *epoll\_ctl*
    - Они составляют так называемый interest list
  - С помощью *epoll\_wait* ожидаем, пока произойдут события
  - Получаем список готовых дескрипторов (ready list) и обрабатываем их
  - Снова *epoll\_wait...*
  - По окончании удаляем все дескрипторы с помощью *epoll\_ctl*, закрываем дескрипторы

# Edge-Triggered vs Level-Triggered

- Есть два способа оповещения о событиях: Level-Triggered (по умолчанию) и Edge-Triggered (флаг EPOLLET)
- Рассмотрим следующий сценарий:
  - Создадим пайп, добавим в *epoll* дескриптор на чтение
  - Вызовем *epoll\_wait* и заблокируемся
  - Пусть кто-то запишет нам 2Кб данных
  - Разблокируемся и прочитаем 1Кб
  - Снова *epoll\_wait*
  - Теперь два случая:
    - Level-Triggered: мы разблокируемся, потому что не дочитали все данные
    - Edge-Triggered: мы не разблокируемся, пока не произойдёт новое событие

# Edge-Triggered vs Level-Triggered

- То есть в случае Edge-Triggered мы узнаём о событии единожды и должны будем сами проследить за тем, что прочитаем все данные, а в случае Level-Triggered мы будем просыпаться до тех пор, пока не устраним изначальную причину оповещения
- Первый подход позволяет избежать нам пробуждения на событии, которое уже обрабатывается
- Интересные сценарии рассмотрим позже, после потоков

# Высоконагруженные серверы

- Примером популярного веб-сервера может служить *Apache HTTP Server*
  - Проблему быстрой обработки нескольких клиентов он решает с помощью создания выделенного потока или процесса на каждого клиента, что, очевидно, вызывает большие затраты по ресурсам
  - Умеет генерировать динамический контент
- Не менее важные примеры — это *nginx* и *lighttpd*
  - Под капотом они используют механизмы, работающие по принципу *epoll*
  - Благодаря этому и получается мультиплексирование: один поток или процесс может обслуживать сразу несколько клиентов
  - Не умеют генерировать динамический контент, только статика

# Reverse Proxy

- Как правило, за любым сайтом стоит больше одной машины
- Reverse Proxy – это специально обученный сервер, который распределяет внешние запросы по внутренней сети, а затем отправляет ответы обратно
  - Клиент этого даже не узнает
- Для такой задачи можно использовать *nginx*

# Многопоточность



# 0

Столько слайдов по этой теме пока что здесь есть :(

# Синхронизация потоков



# Mutex

- Прimitives синхронизации, гарантирующий *mutual exclusion* (взаимное исключение)

```
#include <pthread.h>

pthread_mutex_init(pthread_mutex_t* mutex,
const pthread_mutexattr_t* attr);
pthread_mutex_destroy(pthread_mutex_t* mutex);

// Инициализируем так
pthread_mutex_t lock =
PTHREAD_MUTEX_INITIALIZER;

// Или так
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);

// Не забываем уничтожить в конце
pthread_mutex_destroy(&lock);
```

# Mutex

- Захватить мьютекс может только один поток
- Если мьютекс уже захвачен, то
  - *lock* – ждём
  - *trylock* – не ждём
- Отпустить мьютекс может только тот поток, который его захватил

```
#include <pthread.h>

pthread_mutex_lock(pthread_mutex_t* mutex);
pthread_mutex_trylock(pthread_mutex_t*
mutex);
pthread_mutex_unlock(pthread_mutex_t* mutex);

// Захватываем мьютекс
pthread_mutex_lock(&lock);

// Критическая секция

// Отпускаем мьютекс
pthread_mutex_unlock(&lock);
```

# Задача о философaх

- За круглым столом обедают философы. Между соседями положили по вилке. Для того, чтобы начать приём пищи, философ должен взять по вилке в каждую руку. Нужно разработать такую модель поведения, при которой не окажется голодающих философов.

Philosophers then:



I will discover the secrets  
of the Universe

Philosophers in 1965:



this dood next to me  
took mi fork, can't eat :(

# Condvar

- Примитив синхронизации, позволяющий реализовать ожидание выполнения какого-то условия

```
#include <pthread.h>

pthread_cond_init(pthread_cond_t* c, const
pthread_condattr_t* attr);
pthread_cond_destroy(pthread_cond_t* c);

// Инициализируем так
pthread_cond_t cv =
PTHREAD_COND_INITIALIZER;

// Или так
pthread_cond_t cv;
pthread_cond_init(&cv, NULL);

// Не забываем уничтожить
pthread_cond_destroy(&cv);
```

# Condvar

- При ожидании с каждым кондваром ассоциирован ещё и мьютекс
- Поток должен захватить этот мьютекс и только потом вызвать *wait*, после чего мьютекс отпустится сам
- Далее поток будет ожидать оповещения, которое может прийти как только ему (*signal*), так и всем ожидающим (*broadcast*)

```
#include <pthread.h>

pthread_cond_wait(pthread_cond_t* c,
pthread_mutex_t* m);
pthread_cond_timedwait(pthread_cond_t*
c, pthread_mutex_t* m, const struct
timespec *timeout);
pthread_cond_signal(pthread_cond_t* c);
pthread_cond_broadcast(pthread_cond_t*
c);
```

# Condvar

- Обратите внимание на то, что ожидание происходит в цикле
- У нас нет гарантии того, что потоки проснутся только после оповещения, это называется *spurious wakeup*, поэтому перед выходом из ожидания нужно проверить условие ещё раз

```
// Встаём на ожидание
pthread_mutex_lock(&lock);
while (!condition) {
    pthread_cond_wait(&cv, &lock);
}

// Критическая секция

// Отпускаем мьютекс
pthread_mutex_unlock(&lock);
```

# Атомарные переменные

- Для переменных, вмещающихся в машинное слово можно гарантировать удобные атомарные операции
- В Си (начиная с C11) такую переменную можно пометить атрибутом *\_Atomic*
- Общий интерфейс у атомарных переменных следующий:
  - *void Store(object, new\_value)*
  - *T Load(object)*
  - *T Exchange(object, new\_value)*
  - *bool CAS\_weak(object, expected, new\_value)*
  - *bool CAS\_strong(object, expected, new\_value)*
  - *T FetchMod(object, operand)*



# Lock-Free структуры

- Использование примитивов синхронизации не только имеет накладные расходы, но и заставляет другие потоки ждать
- Иногда можно реализовать структуру так, что она будет работать без блокировок
- Например, можно реализовать lock-free односвязный список (получается, и стек)

# Проблема АВА

- Рассмотрим следующий сценарий:
  - Поток  $T1$  читает из ячейки значение  $A$
  - Выполнение переходит потоку  $T2$
  - Поток  $T2$  записывает в ячейку значение  $B$ , а затем снова значение  $A$
  - Выполнение переходит потоку  $T1$
  - Потoku  $T1$  кажется, что с ячейкой ничего не произошло
- Если подобные сценарии не влияют на работу алгоритма, то всё хорошо

Низкоуровневое сетевое взаимодействие

# Протокол UDP

- *User Datagram Protocol*
- Уже рассматривали [здесь](#)

```
#include <sys/socket.h>
```

```
#include <sys/types.h>
```

```
// Создание UDP-сокета
```

```
int socket_fd = socket(AF_INET, SOCK_DGRAM, 0);
```

# Общение без установки соединения

- Для отправки сообщения используется *sendto*
  - В него мы обязаны передать адрес получателя, потому что предварительно не подключаемся к нему
- Для получения сообщения используется *recvfrom*
  - Нужно не забыть поставить адрес на прослушивание с помощью *bind*
  - Если нам не нужен адрес отправителя, можем оставить *NULL*

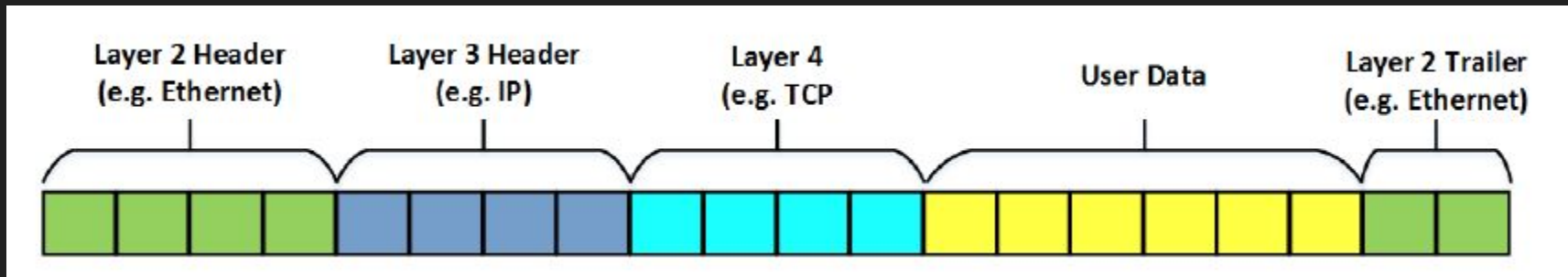
```
// Пример отправки сообщения
const char MSG[] = "I love UDP";
sendto(socket_fd, MSG, sizeof(MSG), 0,
        &ipv4_addr, sizeof(ipv4_addr));

// Пример получения сообщения
bind(socket_fd, &ipv4_addr, sizeof(ipv4_addr));

char buffer[4096];
recvfrom(socket_fd, buffer, 4096, 0,
        NULL, NULL);
```

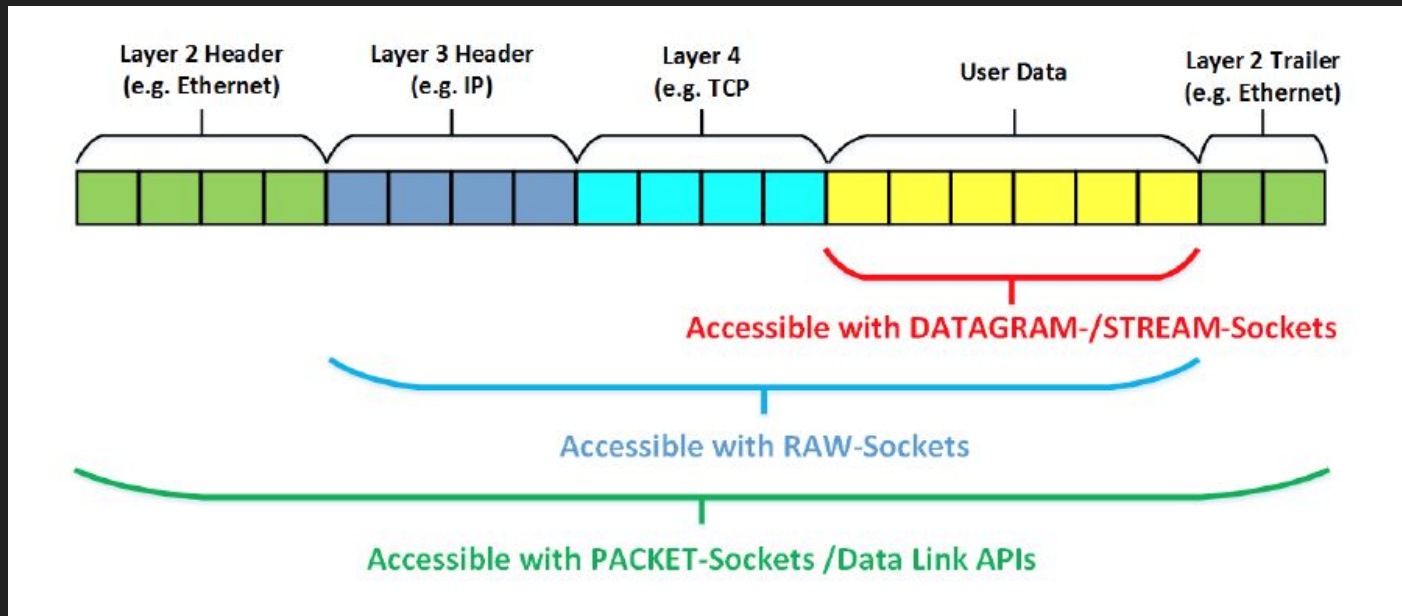
# Низкоуровневое сетевое взаимодействие

- Мы привыкли передавать по сети только высокоуровневые данные (числа, HTTP-запросы)
- На самом деле всё, что передаётся по интернету выглядит как набор вложенных друг в друга пакетов, относящихся к разным протоколам
  - Для каждого протокола нужен заголовок, содержащий необходимую информацию



# Низкоуровневое сетевое взаимодействие

- Для получения доступа к заголовкам используются сырые сокеты



# Низкоуровневое сетевое взаимодействие

- Тип используемого сокета зависит от поставленных нами целей

```
// ICMP-пакет в нашем распоряжении
int icmp_socket_fd = socket(AF_INET, SOCK_RAW,
IPPROTO_ICMP);
```

```
// IPv4-пакет в нашем распоряжении
int ip_socket_fd = socket(AF_INET, SOCK_RAW,
IPPROTO_RAW);
```

```
// ARP-пакет в нашем распоряжении
int arp_socket_fd = socket(AF_PACKET,
SOCK_DGRAM, htons(ETH_P_ARP));
```

```
// Собираем любой пакет, начиная с L2
int any_socket_fd = socket(AF_PACKET, SOCK_RAW,
htons(ETH_P_ALL));
```



# ICMP

- *Internet Control Message Protocol*
- Протокол сетевого уровня, который используется для передачи информации об ошибках; у сообщений есть тип, который определяет их содержание
- Используется в утилите *ping*
  - Проверяет наличие маршрута (тип 8 – echo-запрос, тип 0 – echo-ответ)
- Используется в утилите *tracert*
  - Не только проверяет наличие, но и строит сам маршрут
    - Идея следующая: утилита отправляет пакеты, постепенно увеличивая время их жизни, на каждом шаге запоминая крайний маршрутизатор

# DNS

- *Domain Name System*
- Протокол, используемый для получения данных о доменах (как правило, для получения IP-адреса сервера)
- Данные лежат в *записях*:
  - *A* – IPv4 адрес
  - *AAAA* – IPv6 адрес
  - *MX* – почтовые серверы данного домена
  - *NS* – DNS-серверы данного домена
  - *TXT* – любая дополнительная информация

# Библиотеки функций и их загрузка

# Библиотеки

- Мы можем создавать исполняемый код, который будут использовать другие программы
- Для этого нам нужно создать библиотеку
  - Статическую
    - `$ gcc -c -fPIC -o libcaos.a libcaos.c`
  - Динамическую
    - `$ gcc -shared -fPIC -o libcaos.so libcaos.c`
- Не забудьте про позиционнезависимость (флаг `-fPIC`)

# Динамические библиотеки

- Как правило, есть библиотеки, которые очень часто используются в программах
- Очевидно, проще положить один экземпляр в файловой системе и разрешить использовать его другим программам при необходимости
- Такой подход заметно сокращает размеры исполняемых файлов
- Динамические зависимости можно посмотреть с помощью утилиты *ldd*

# Использование динамических библиотек

- Для того, чтобы передать линковщику информацию об используемых библиотеках и их местонахождении, можно использовать флаги `-l` и `-L`
- Пример:
  - `$ gcc -lcaos -L. main.c`
    - Воспользуется библиотекой `libcaos.so`, находящейся в текущей папке

# Использование динамических библиотек

- При запуске программы будет произведён поиск требуемых библиотек в стандартных каталогах (*/lib, /usr/lib, /usr/local/lib*)
  - Дополнительные каталоги определяются в *LD\_LIBRARY\_PATH*
- Если библиотека имеет иное заранее известное расположение, его можно указать при линковке
  - *\$ gcc main.c -lcaos -L. -Wl,-rpath -Wl,'\$ORIGIN'*
    - С помощью флага *-Wl* мы передали линковщику информацию о том, что используемые динамические библиотеки нужно искать в каталоге, в котором находится исполняемый файл
    - Если смотреть на бинарник, то хранится эта информация в атрибуте *DT\_RUNPATH*

# Загрузка библиотек во время выполнения

- Динамические библиотеки можно не привязывать к программе, а загружать прямо во время рантайма и брать оттуда нужные символы
- *dlopen* возвращает нам хендл библиотеки, который в дальнейшем пригодится для поиска символов
- *dlclose* закрывает этот хендл

```
#include <dlfcn.h>
```

```
void* dlopen(const char* filename, int  
flags) ;
```

```
int dlclosе(void* handle) ;
```



# Обработка ошибок

- В случае ошибки *dlopen* возвращает *NULL*, а *dlclose* – ненулевое значение
- Для того, чтобы получить информацию об ошибке, можно воспользоваться функцией *dLError*

```
#include <dlfcn.h>
```

```
char* dLError(void);
```

# Чтение символов

- Когда мы получили хендлер, связанный с библиотекой, мы можем получить адрес, связанный с интересующим нас символом
- Например, мы можем достать указатель на функцию

```
#include <dlfcn.h>
```

```
void* dlsym(void* handle, const char*  
symbol) ;
```

# CMake

# CMake

- В первом модуле мы говорили о *Makefile*
  - Если коротко, то это способ автоматизации сборки
    - Мы единожды описываем в *Makefile* необходимые для сборки действия, потом нам достаточно запустить утилиту *make*
- К сожалению, полученный *Makefile* не является кроссплатформенным
- Эта проблема может быть решена с помощью *CMake*
  - Теперь мы описываем этапы сборки с помощью абстрактных сущностей и получаем *CMakeLists.txt*
  - Утилита *cmake* на основе этого файла получает платформозависимый *Makefile*, на который уже можно натравить утилиту *make*
- Где почитать
  - [Ридинг](#)
  - [Доки](#)
  - Любой крупный Open Source проект (например, [Telegram](#))

HTTP

# Путешествие в прошлое

- Со времён [недавнего семинара](#) суть HTTP не изменилась
- Основные методы:
  - *GET* – предназначен для получения данных пользователем
  - *POST* – наоборот, отправка пользовательских данных
  - Поддержка методов и поведение при их использовании зависит от сервера и клиента
    - Так, можно сочинить свои методы и использовать их
- Взаимодействовать по HTTP с сервером можно с помощью браузера или утилиты *telnet*
  - *telnet ya.ru 80*
- Также взаимодействовать можно и по HTTPS
  - *openssl s\_client -connect yandex.ru:443*

# Получение IP адреса

- До этого в задачах нам сразу давали IP адрес сервера, но сейчас всё резко изменилось
- К счастью, уже есть специально обученные функции, которые позволяют нам это сделать

```
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
```

```
int getaddrinfo(const char* restrict node,
                const char* restrict service,
                const struct addrinfo* restrict hints,
                struct addrinfo** restrict res);
```

```
void freeaddrinfo(struct addrinfo* res);
```

```
const char* gai_strerror(int errcode);
```

# Получение IP адреса

- По факту нам нужно только указать доменное имя, порт и критерии, по которым нужно выбрать сокет
- Не забываем освобождать место от созданных структур
- Библиотека не использует *errno*, поэтому существует специальная функция для обработки ошибок

```
// Критерии для сокета
struct addrinfo hints;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;

struct addrinfo* result = NULL;

int getaddrinfo_result =
getaddrinfo(server_hostname, server_port, &hints,
&result);

if (getaddrinfo_result != 0)
    fprintf(stderr, "getaddrinfo: %s\n",
gai_strerror(getaddrinfo_result));
```



cURL

# \$ curl

- Базовой, но достаточно мощной, утилитой для сетевого взаимодействия является *curl*
  - В основном используется для HTTP и HTTPS, но также поддерживает и другие протоколы прикладного уровня
- Основные флаги:
  - *-X METHOD* – запрос с указанным методом (по умолчанию *GET*)
  - *-H "Header: value"* – дополнительно передать заголовок
  - *-L* – следовать по redirect'ам
  - *-o filename* – сохранить ответ в файл
  - *-d "field1=val1&field2=val2"* – отправка данных
  - *-data-binary "data"* – отправка данных
  - *-data-binary @filename* – отправка содержимого файла
- Наше любимое: *\$ curl parrot.live*

# libcurl

- Кроме утилиты у нас в распоряжении есть API, которым мы можем пользоваться прямо из кода
- Использовать мы будем упрощённый *easy* интерфейс
- У библиотеки достаточно подробная документация и куча [примеров](#)

HTTP/2

# HTTP/2

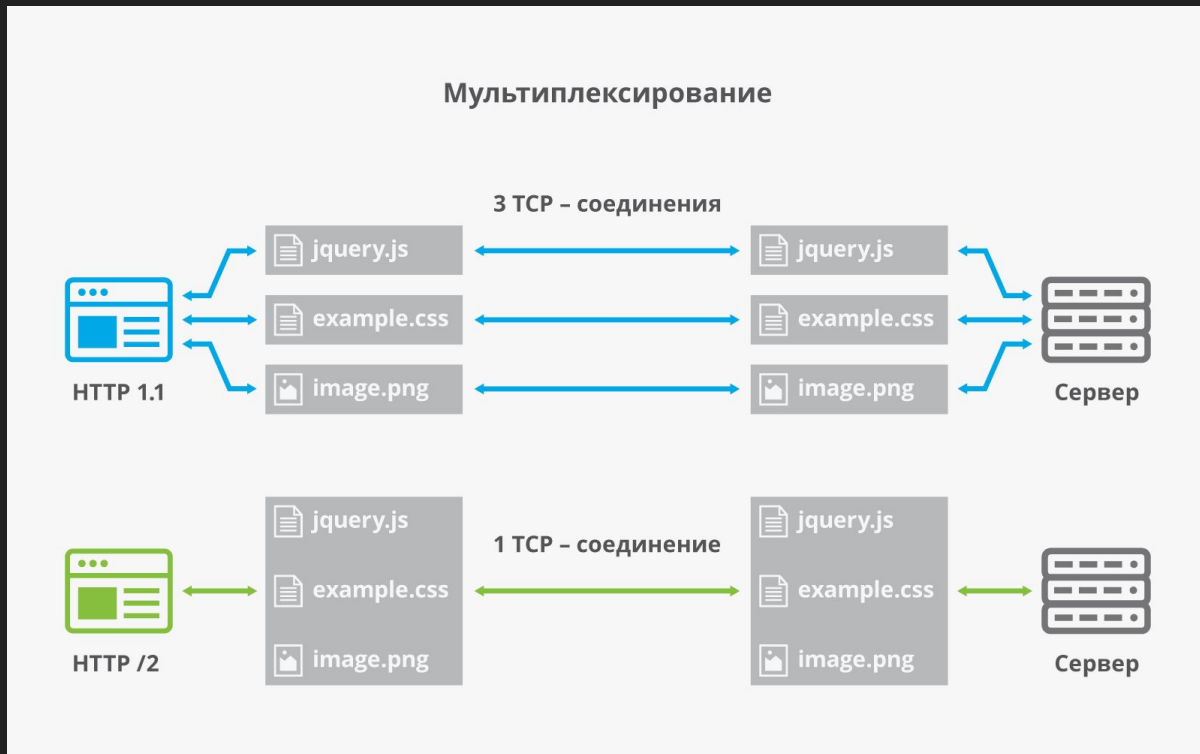
- Новая версия протокола в основном нацелена на увеличение скорости работы
- У HTTP/1.1 был ряд недостатков, которые с каждым годом его использования всё более ярко выражались
- Первыми попытались исправить эти недостатки в Google, описав протокол [SPDY](#)
- Позже при появлении HTTP/2 Google остановили поддержку SPDY, чтобы дать возможность развиваться новому протоколу

# Мультиплексирование

Наиболее важным нововведением является мультиплексирование

Число TCP-соединений в браузере ограничено, поэтому раньше возникала проблема Head-Of-Line Blocking

Теперь запросы выполняются внутри одного соединения конкурентно



# Остальные изменения

- *Протокол стал бинарным*

- Текстовым он был для удобства пользователя, но в свою очередь это добавляло некоторый оверхед, да и из-за этого было [несколько вариантов](#) парсинга сообщения
- В целом ничего страшного, программы всё ещё могут приводить пакеты в человекочитаемый вид, когда они приходят

- *Появилось сжатие заголовков*

- Современные веб-страницы (ужасны) загружают десятки файлов ресурсов, соответственно, в каждом таком запросе присутствуют заголовки
- Как оказалось, если их сжимать, то можно получается хорошая экономия места
- В SPDY использовался GZIP, однако против него существует атака [CRIME](#)
- Авторы протокола HTTP/2 предложили свой безопасный алгоритм сжатия HPACK

# Остальные изменения

- *Server Push*

- Перед тем, как сделать запросы файлов ресурсов, браузеру необходимо загрузить и распарсить основной HTML файл
- Сервер знает, какие файлы запросит клиент, поэтому может сразу быть на шаг впереди и отправить их клиенту при первом же запросе

- *Приоритизация*

- Пользователю важно как можно быстрее увидеть результат своих действий, в то же время этот результат состоит из кучи частей: HTML, CSS, JS, изображения, шрифты
- Разумно в первую очередь загружать ресурсы, которые в данный момент добавят пользователю больше всего информации
- Например, какой толк грузить шрифты или CSS, если не загрузился текст?



# HTTP/2 и безопасность

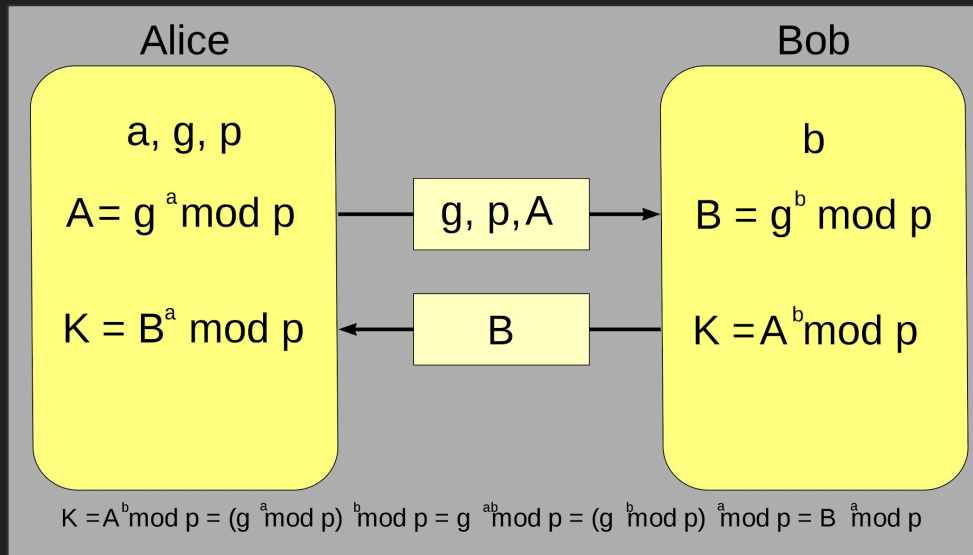
- По умолчанию стандарт **не требует** использование защищенного соединения
- Однако разработчики всех браузеров решили дружно не поддерживать HTTP/2 в пользу HTTPS/2
- Получается у протокола есть бонусная фишка: если используется вторая версия протокола, то это почти всегда HTTPS/2
  - Исключение: gRPC в локальной сети

# HTTPS

- Мы много говорили о нём, давайте теперь копнём немного глубже
- HTTPS – это HTTP over TLS (ранее использовался SSL), то есть кроме передачи данных нас теперь волнует и защищённость соединения
- Если по шагам, то теперь подключение выглядит так:
  - Клиент подключается к серверу и запрашивает защищённое соединение
  - Клиент сообщает, какие он поддерживает методы шифрования
  - Сервер выбирает наиболее надёжный из предложенных алгоритмов и сообщает клиенту
  - Сервер отправляет клиенту свой сертификат (вместе с публичным ключом)
  - Клиент проверяет, что сертификат выдан надёжным источником
  - Если клиента и сервер всё устраивает, они генерируют сеансовый ключ
  - Все дальнейшие запросы защищены сеансовым ключом

# Протокол Диффи-Хеллмана

- На прошлом слайде мы опустили очень важную часть – генерацию сеансового ключа. А как это сделать?
- Ранее для этого использовался RSA, но сейчас принято использовать протокол [DH](#) (Diffie-Hellman)

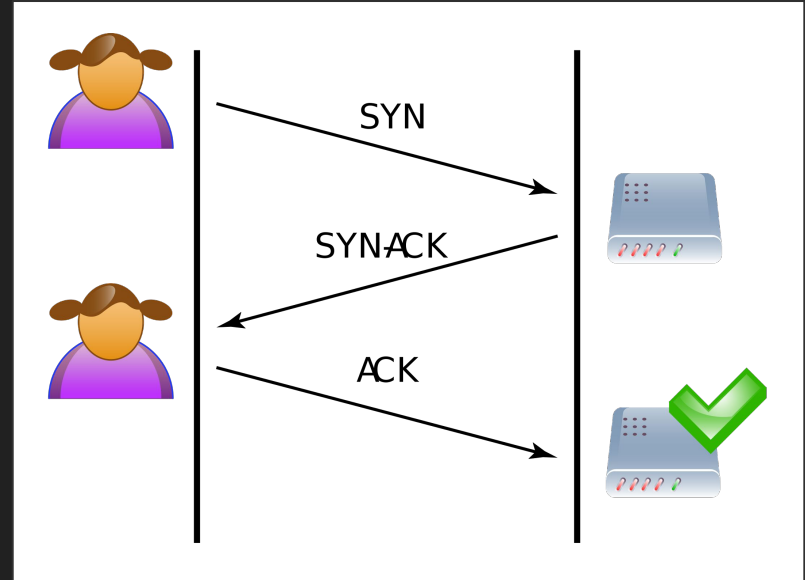


# Атаки на HTTP

- Как и любой протокол, HTTP имеет свои уязвимости
  - В основном они нацелены на *DoS (Denial-of-Service)*, потому что определить, является ли запрос настоящим, – это нетривиальная задача
- *Man-in-the-Middle*
  - Если не проверять сертификат у authority, то может возникнуть иллюзия ложной безопасности
  - Между клиентом и сервером может вклиниться злоумышленник и настроить безопасное соединение и с клиентом, и с сервером, при этом он может читать plaintext данные
- *Понижение HTTPS до HTTP*
  - Если на сайте не настроен автоматический переход на безопасный протокол, то можно вынудить пользователя зайти по небезопасному соединению и, например, прочитать трафик

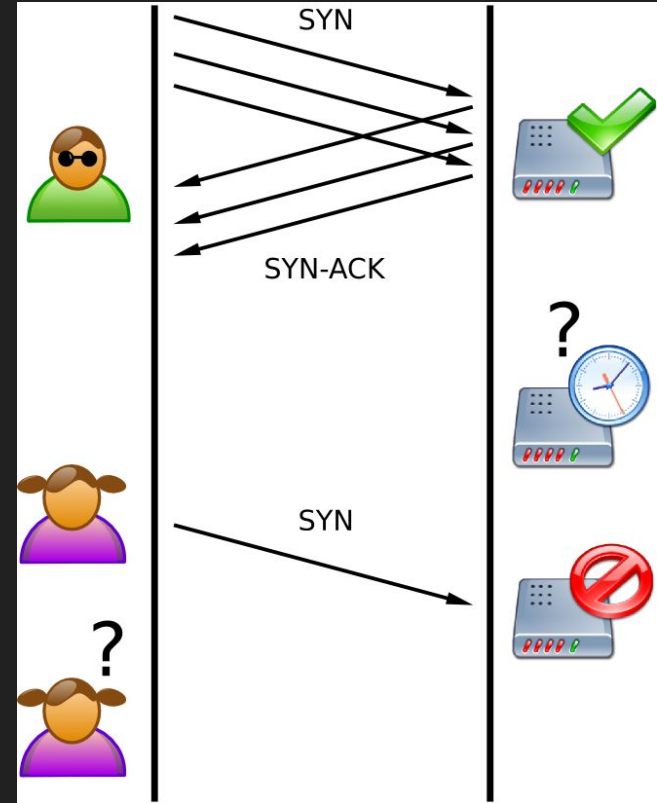
# SYN Flood

- Как известно, HTTP работает поверх TCP
- Для установки TCP-соединения используются специальные SYN-пакеты
- За три шага клиент и сервер договариваются о том, что сейчас будут общаться (*TCP Three-Way Handshake*)



# SYN Flood

- А давайте мы отправим кучу SYN-пакетов серверу, но даже не будем ждать ответ
- В то же время приличный сервер должен подождать ответ от клиента, а то вдруг у него медленный интернет
- Таким образом можно забить TCP канал и другие пользователи не смогут связаться с сервером



# Атаки на HTTP

- *GET Flood / POST Flood*

- Давайте отправим очень много валидных запросов, чтобы сервер не мог обслужить других клиентов
- Проблема злоумышленника 1: чтобы отправить HTTP запрос, нужно установить соединение, то есть раскрыть IP-адрес. Решается эта проблема ботнетами
- Проблема злоумышленника 2: если сервер заметит подозрительно много запросов с одних и тех же адресов, он может предложить ввести капчу. Здесь уже сложнее

- *Reverse Bandwidth Flood*

- Давайте замучаем сервер не входящими, а исходящими запросами, то есть заставим его отправлять очень много данных, чтобы забить канал

# Атаки на HTTP

- *Low and Slow*

- Давайте будем эмулировать медленного клиента, от которого HTTP запрос приходит посимвольно
- Отличить злоумышленника от медленного юзера действительно сложно, при этом сервер будет тратить ресурсы на обработку обоих

- *Cache Bypassing*

- В основном серверы работают через *CDN (Content-Delivery Network)*
  - Например, [Cloudflare](#) (который в том числе защищает от DDoS-атак)
- Для ускорения часть контента кэшируется и хранится на распределенных серверах
- Давайте будем просить у сервера контент, который не может лежать в кэше
- Если попросим очень много, то даже несмотря на CDN сервер может перестать обрабатывать запросы



# Непрямые атаки

- Чтобы положить сайт, необязательно атаковать сервер напрямую, всегда можно найти какой-нибудь обходной путь (выключить сервер из сети, ы)
- Отличным примером является [DDoS Attack on Dyn](#)
  - Как вы помните, DNS серверы нужны для того, чтобы по удобночитаемому символьному имени получать IP-адрес
  - А давайте мы возьмём и уроним крупного DNS-провайдера с помощью огромного ботнета из принтеров, камер и прочих IoT устройств
  - Победа! Пользователи не могут получить доступ к крупнейшим сайтам с, казалось бы, отличной защитой от атак

# Дополнительные материалы

- Говорить про всякие защищенные штуки и их взлом довольно интересно, но интереснее трогать это всё руками:
  - [Задачи на протокол DH](#)
  - [Куча задач на категорию Web](#)
- Если непонятно с чего начать, то можно посмотреть:
  - [Курсы на cryptohack](#)
  - [Канал SPbCTF](#)

# Шифрование

# А зачем оно мне нужно?

- Прежде всего шифрование нужно для скрытия информации от третьих лиц
- Согласно Википедии, шифрование обеспечивает три состояния безопасности информации:
  - *Конфиденциальность* – третьи лица не могут напрямую увидеть передаваемую информацию (вы же её зашифровали как бы)
    - Не путать с *анонимностью*
  - *Целостность* – информация не будет изменена в процессе передачи (для изменения нужно сначала её расшифровать)
  - *Идентифицируемость* – можно определить источник информации

# Криптография

- Наука, которая занимается созданием и анализом различных шифров
- В свою очередь шифры можно разделить на группы:
  - *Симметричные* – для шифрования и дешифрования используется один и тот же ключ
  - *Асимметричные* – ключи для шифрования и дешифрования различны
  - *Блочные* – разбивают данные на блоки и работают с ними
  - *Поточные* – работают с потоком данных
- Почему криптографические алгоритмы должны быть Open Source?

# Криптографическая стойкость

- По сути нам хочется оценить, насколько тяжело шифр поддается криптоанализу (читайте “взлому”)
- Любой шифр можно взломать перебором, если он не является *абсолютно криптостойким*
  - Как правило, это перебор ключа, поэтому если возможных ключей очень много, то и перебор займёт кучу времени
- Абсолютная стойкость означает то, что злоумышленник не сможет получить никакой полезной информации из текста. Клод Шеннон выдвинул следующие требования к такой функции:
  - Каждый ключ используется ровно один раз
  - Ключ статистически надёжен (все символы равновероятны, независимы и случайны)
  - Длина ключа равна или больше длины сообщения

# Как защитить свой шифр?

- Построить абсолютно стойкую систему сложно
- Для использования подойдут и *достаточно стойкие* системы, для которых оцениваются следующие критерии:
  - Вычислительная сложность полного перебора
  - Наличие уязвимостей
- Подходы к построению системы с высокой стойкостью:
  - Учёт существующих методов взлома и защита системы от них
  - Составить шифр так, чтобы его сложность была эквивалентна сложной вычислительной задаче (например, факторизация очень большого числа)

# Симметричное шифрование

- Предполагает использование одного и того же ключа как для шифрования, так и для дешифрования
- Характеризовать задачу симметричных шифров можно так: используя короткий ключ безопасно передать длинное сообщение
- Известные представители: *DES, AES*
  - Второй из них настолько распространен, что для него есть [специальные ассемблерные инструкции](#)
- Как мы помним, симметричные шифры делятся на
  - Поточные – одновременно обрабатывается только один байт последовательности
  - Блочные – последовательность делится на блоки, которые обрабатываются независимо
    - В конце в зависимости от *Mode of Operation* блоки собираются в одно сообщение



# AES (2001)

- Advanced Encryption Standard
- Характеристики:
  - Работает с блоками размера 128 бит
  - Поддерживает ключи размера 128, 192 и 256 бит
  - В зависимости от ключа происходит 10, 12 или 14 *раундов*
- Под *раундом* подразумевается применение последовательных обратимых операций:
  - *SubBytes* – замена байт на другие с помощью специальной таблицы
  - *ShiftRows* – сдвиг байт внутри строки (содержимое блока делится на 4 строки)
  - *MixColumns* – обратимое смешивание байт внутри столбцов
  - *AddRoundKey* – смешиваем раундовый ключ с текущим состоянием и генерируем ключ на следующий раунд

# Криптостойкость AES

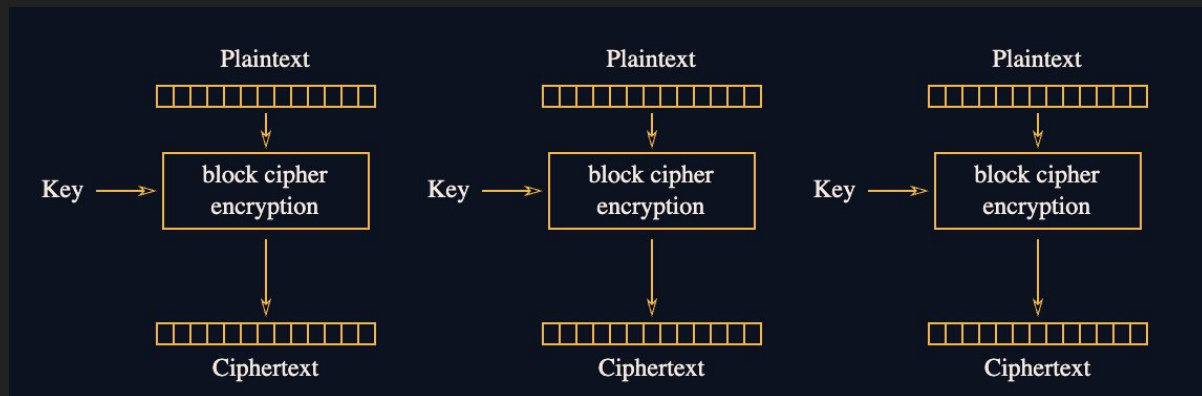
- Шифр был признан настолько надежным, что использовался в АНБ для защиты информации
- Однако криптоаналитиков смущала такая простота шифра
  - Несмотря на это найти уязвимость в математическом обосновании алгоритма пока не удалось
- Можно ли атаковать не сам алгоритм, а его реализацию?
  - Да, например, оценивать время работы операций шифрования
    - Однако для этого нужно иметь доступ к компьютеру жертвы

# ECB (Electronic Codebook)

Применяем шифрование к отдельным блокам, а потом их конкатенируем

Выглядит ненадежным, да?

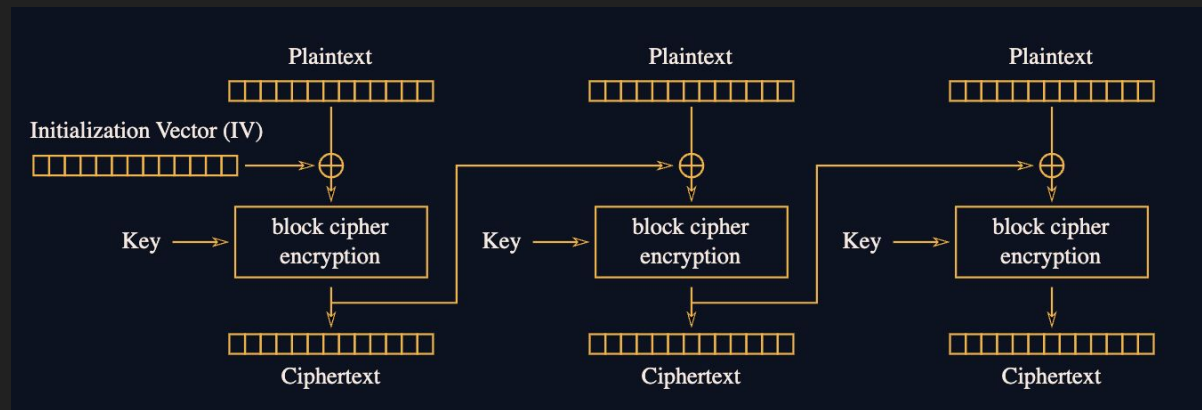
С другой стороны, оно параллелится, но безопасностью жертвовать нельзя



# CBC (Cipher Block Chaining)

Будем использовать  
шифротекст с предыдущего  
шага

На первом шаге шифротекста  
нет, поэтому воспользуемся  
случайно сгенерированным  $IV$



# GCM (Galois/Counter Mode)

- Если посмотреть на алгоритмы, которые используются в протоколе [TLS](#), то можно увидеть, что сейчас в качестве симметричного метода шифрования принято использовать [AES GCM](#)

# Асимметричное шифрование

- Теперь у нас в распоряжении есть пара из:
  - *Public Key* – открытый ключ, можно рассылать кому угодно (друзьям, родителям, хакерам)
  - *Private Key* – закрытый ключ, нужно держать в строгом секрете
- Открытым ключом можно зашифровать сообщение так, что его можно будет прочитать только с помощью закрытого ключа
- Также асимметричное шифрование можно применять для электронной подписи
- Основывается на вычислительно сложных задачах, для которых не существует известного полиномиального решения

# RSA (1977)

- [Rivest, Shamir, Adleman](#)
- Алгоритм асимметричного шифрования, основывающийся на сложности факторизации больших целых чисел
- Размер ключа от 2048 до 4096 бит

# Алгоритм создания ключей RSA

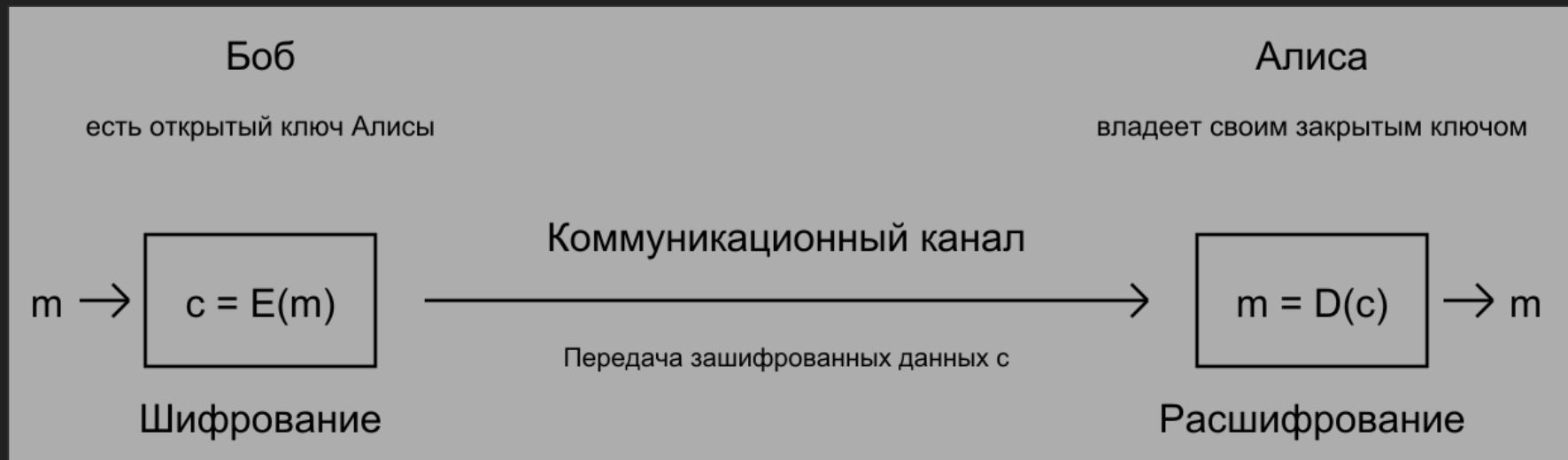
- Генерируем два случайных больших простых числа  $p$  и  $q$
- Вычисляем модуль  $n = pq$
- Вычисляем функцию Эйлера для модуля  $\phi(n) = (p - 1)(q - 1)$
- Выбираем открытую экспоненту  $e$ , взаимно простую с  $\phi(n)$ 
  - Как правило, используют 17, 257 и 65537
  - Значение 3 считается небезопасным
- Находим число  $d$  такое, что  $de = 1 \pmod{\phi(n)}$
- Получаем две пары:  $(e, n)$  и  $(d, n)$  – открытый и закрытый ключи соответственно



# Обмен сообщениями в RSA

- Алиса предоставляет свой открытый ключ  $(e, n)$
- Боб шифрует сообщение  $m$  следующим образом:
  - $c = m^e \pmod n$
- Боб передаёт сообщение Алисе
- Алиса дешифрует шифротекст  $c$  следующим образом:
  - $m = c^d \pmod n$
- А как возвести строку в степень???

# Обмен сообщениями в RSA



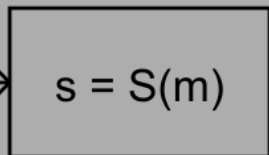
# Цифровая подпись

- Для верификации противоположной стороны общения используется цифровая подпись
- Алгоритм следующий (пусть Алиса хочет себя подтвердить):
  - Алиса шифрует текст закрытым ключом:  $s = m^d \pmod n$
  - Алиса передаёт пару  $(m, s)$  Бобу
  - Боб с помощью открытого ключа получает  $m' = s^e \pmod n$
  - Если  $m$  и  $m'$  равны, то сообщение было отправлено Алисой
    - Ева не могла добиться такого же эффекта без закрытого ключа Алисы

# Цифровая подпись

Алиса

владеет своим закрытым ключом



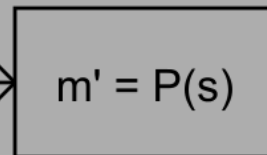
Генерация подписи

Коммуникационный канал

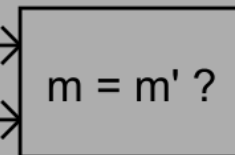
Передача сообщения и подписи

Боб

есть открытый ключ Алисы



Вычисление прообраза



Сравнение сообщения  
и прообраза

$m$

$m$

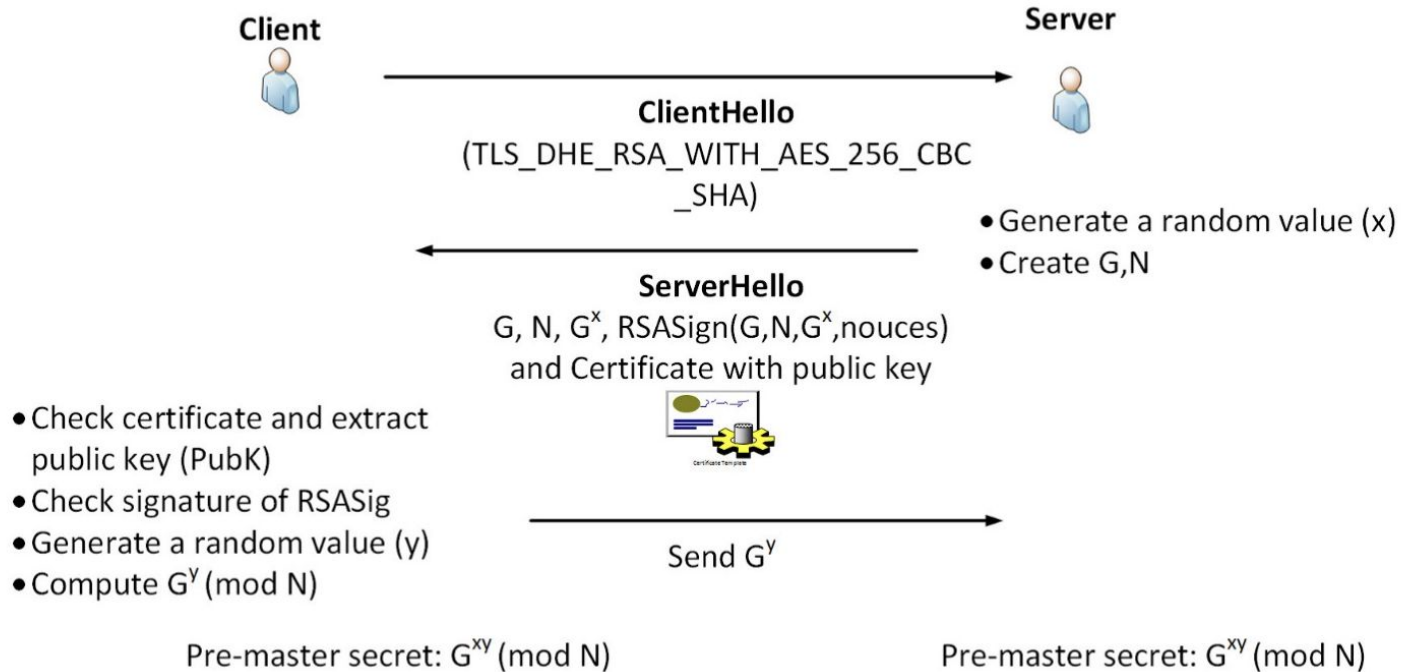
# DHE-RSA

- Вспомним протокол Диффи-Хеллмана для генерации общего сеансового ключа
- Между Алисой и Бобом может встать Ева (злоумышленник) и устроить атаку Man-in-the-Middle
  - Ева создаёт сеансовые ключи с Алисой и Бобом, при этом у неё на руках всегда будет plaintext
  - Для Алисы и Боба всё будет выглядеть так, будто между ними никого нет

# DHE-RSA

- Давайте не оставим Еве ни единого шанса
- Будем генерировать новый ключ на каждый сеанс (Е - Ephemeral) с помощью протокола Диффи-Хеллмана (DH)
- Для того, чтобы предотвратить М-i-t-M, Алиса подпишет передаваемые числа, чтобы Боб мог быть уверен в их подлинности

# DHE-RSA



# Хэширование

- Рядом с шифрами всегда тусуются хэш-функции
- Если в случае шифров нам нужно было получить обратимое преобразование, то здесь такой цели не стоит
- Требования к хэш-функциям следующие:
  - Результат фиксированного размера
  - Сильное изменение результата при небольшом изменении входа
  - Отсутствие коллизий
- Варианты использования:
  - С алгоритмической частью вы знакомы
  - Проверка чек-сумм
  - Хранение паролей
  - Обезличивание данных



# MD5 (1991)

- *Message Digest 5*
- Устаревшая, но когда-то очень широко используемая хэш-функция
- Размер хэша – 128 бит

# SHA-2 (2002)

- *Secure Hash Algorithm Version 2*
- Семейство хэш-функций, в которые входят такие известные представители как *SHA-256* и *SHA-512*

# OpenSSL и LibreSSL

- Открытые библиотеки, представляющие криптографию в Linux
- У библиотек очень похожее API, но у второй более богатая [документация](#)
- Различные примеры использования можно найти в [ридинге](#)

# Всё ли так просто с хэшированием паролей?

- Конечно, хранить хэши паролей не в plaintext уже хорошо, но в некоторых случаях это может быть бесполезно
- Существуют специально обученные таблицы, в которых по хэшу можно найти исходный текст
- Хорошей идеей будет *посолимь* (*salt*) текст перед хэшированием

# libcrypto

- Как и в случае cURL, нам хочется иметь API для использования в нашем любимом языке программирования C
- Работа хэш-функций *HASH* в этой библиотеке состоит из трёх этапов:
  - *HASH\_Init* – инициализация
  - *HASH\_Update* – добавление порции данных
  - *HASH\_Final* – результат работы хэш-функции
  - [Доки для семейства SHA](#)

# Дополнительные материалы

- Инструменты
  - [CyberChef](#) – по сути швейцарский нож
  - [CrackStation](#) – взлом хэшей
- Задачи
  - [AES](#)
  - [RSA](#)
  - [Хэш-функции](#)

FUSE

# Работа с директориями

- Вспомним, что директории также являются файлами
- Для поддержки иерархии ФС они хранят в себе соответствие имени файла и его *inode*
- При открытии директории с помощью системного вызова *open* можно указать флаг *O\_DIRECTORY*, чтобы проверить, что мы точно открыли директорию (иначе словим *-1*)

```
int dir_fd = open("test_dir", O_RDONLY |  
O_DIRECTORY);
```

```
printf("%d\n", dir_fd); // OK
```

```
int file_fd = open("test_file", O_RDONLY |  
O_DIRECTORY);
```

```
printf("%d\n", file_fd); // -1
```



# Работа с директориями

- Директории – особые файлы, поэтому для работы с ними существуют специальные функции
- Нужно учитывать, что в каждой директории находится как минимум две записи
  - . – текущая директория
  - .. – родительская директория

```
// Открытие
DIR* opendir(const char* name);
DIR* fdopendir(int fd);

// Закрытие (нужно освободить память)
int closedir(DIR* dirp);

// Чтение содержимого директории
struct dirent* readdir(DIR* dirp);

// Перемещение указателя dirp
void seekdir(DIR* dirp, long loc);
long telldir(DIR* dirp);
```

# Работа с директориями

- Дополнительные функции, которые могут пригодиться

```
#include <unistd.h>

// Имя текущей директории
char* getcwd(char* buf, size_t size);
char* getwd(char* buf);
char* get_current_dir_name(void);

// Сменить текущую директорию
int chdir(const char* path);
int fchdir(int fd);
```

# Виртуальная файловая система (VFS)

- К одному компьютеру может быть подключено сразу несколько устройств, каждое из которых имеет свою файловую систему
- Как добиться того, чтобы это всё могло сосуществовать?
- Концепция виртуальной файловой системы заключается в следующем:
  - Необходимо вынести общую часть всех файловых систем на отдельный уровень, с которого будут вызываться расположенные ниже конкретные файловые системы (очень похоже на виртуальные методы в C++)
    - Мы уже знаем этот интерфейс – это интерфейс POSIX для работы с файлами
  - Пока ФС предоставляет требуемые VFS функции, VFS не знает и не заботится о том, где данные хранятся и что из себя представляет эта ФС

# FUSE

- *Filesystem in Userspace*
- Механизм, позволяющий реализовывать пользовательские файловые системы
- Сам *fuse* является модулем ядра, для доступа к нему используется API *libfuse*
  - *High-Level API* – работа с названиями файлов
  - *Low-Level API* – работа с *inode* напрямую
- Результат – обычная программа, которая при запуске монтирует пользовательскую ФС
- Документация и хороший ридинг

# Python Extending & Embedding

# Запуск интерпретатора из C

- Запускать код на Python можно прямо из программы на C!
- Для этого нужно инициализировать интерпретатор, а затем просто вызвать необходимую функцию
  - Запуск кода из строки
  - Запуск кода из файла
- В конце нужно не забыть провести деинициализацию

```
#include <Python.h>

// ...

Py_Initialize();

// Произвольный код
PyRun_SimpleString("print('023 026')");

// Произвольный файл
FILE* script = fopen("sample_script.py", "r");
PyRun_SimpleFile(script, "sample_script.py");

Py_Finalize();
```

# Обращение к Python-объектам

- В силу отсутствия ООП в С, все объекты – это *PyObject\**
  - Следить за типами нужно самим (*Py\*\_Check*)
- Основные типы: *PyLong*, *PyFloat*, *PyTuple*, *PyList*, *PyDict*, *PyUnicode*
  - Соответствующие им методы начинаются с *Py\*\_*
- Каждый объект хранит счётчик ссылок
  - *Py\_INCREF(ptr)*
  - *Py\_DECREF(ptr)*

```
PyObject* result_matrix =  
PyList_New(matrix_size);
```

```
PyObject* result_row =  
PyList_GetItem(result_matrix, i);
```

# Написание собственных модулей

- Каждый модуль должен иметь функцию инициализации *PyInit\_\**
- В ней мы должны указать название модуля и указатель на список методов

```
PyMODINIT_FUNC PyInit_matrix() {  
    static PyModuleDef module_def = {  
        .m_base = PyModuleDef_HEAD_INIT,  
        .m_name = "matrix",  
        .m_size = -1,  
        .m_methods = matrix_methods,  
    };  
  
    return PyModule_Create(&module_def);  
}
```



# Список методов

- Запись в списке методов состоит из названия, указателя на функцию, вида аргументов и строки документации
- Виды аргументов:
  - *METH\_NOARGS*
  - *METH\_VARARGS*
    - Позиционные
  - *METH\_KEYWORDS*
    - Ключевые
  - *METH\_VARARGS|METH\_KEYWORDS*
    - И те, и те

```
static PyMethodDef matrix_methods[] = {  
    {  
        .ml_name = "dot",  
        .ml_meth = dot,  
        .ml_flags = METH_VARARGS,  
        .ml_doc = "Args: (size, matrix,  
matrix)"  
    },  
  
    // End of list  
    { NULL, NULL, 0, NULL }  
};
```

# Получение аргументов

Переменное число аргументов  
получается за счёт того, что  
они передаются через кортеж

Для распаковки существуют  
специальные [функции](#)

```
PyObject* dot(PyObject* self, PyObject* args_tuple) {  
    int matrix_size = 0;  
    PyObject* matrix_A = NULL;  
    PyObject* matrix_B = NULL;  
  
    if (!PyArg_ParseTuple(args_tuple, "iOO", &matrix_size,  
                                                                    &matrix_A,  
                                                                    &matrix_B)) {  
        return NULL;  
    }  
  
    // ...  
}
```

Q&A



**ВСЁ!**