Архитектура Компьютера и Операционные Системы

и как можно с этим бороться

Лекция 1

Ключевые компоненты операционной системы.

Ядро (Kernel) — программа, которая при составе ОС запускается самой первой, и это единственная программа, которая имеет привилегированный режим выполнения на процессоре и может взаимодействовать с оборудованием напрямую.

А как можно использовать функции ядра? Можно напрямую использовать системные вызовы, можно использовать какие-то высокоуровневые библиотеки (например, стандартная библиотека Си). (Core libraries)

Служебные сервисы (supplementary services) ведут логи, осуществляют обработку сетевых соединений.

Есть также некоторый минимум, чтобы пользователь мог взаимодействовать — user interaction environment.

Процессы.

Процесс — это некоторый экземпляр одной программы. Процессы могут быть программами, которые запускает пользователь, фоновыми сервисами, которые запускаются одновременно с загрузкой системы, либо обычные команды, выполняемые в терминале.

Основные команды:

- `ps` – отображает список процессов в терминале (только для текущего терминала),
- `ps -A` – то же самое, только для всех связанных терминалов,
- `pstree` – отображает иерархию процессов.

У каждого процесса есть порядковый номер.

UNIX программы.

UNIX программы — это некоторые бинарные исполняемые файлы, которые можно запустить в качестве определенного процесса. Программой может быть любой бинарный файл, который имеет атрибут выполняемый, либо любой текстовый файл, который имеет в качестве признака символы #! (эта последовательность символов, если находится в первой строке, указывает путь до интерпретатора, которому должна быть скормлена данная программа в случае запуска).

UNIX пользователи.

Пользователь root, имеющий UID=0, может делать все. Остальные пользователи не могут делать ничего, если им явно не разрешено.

В целях безопасности многие фоновые службы запускаются из-под отдельных фейковых пользователей и из-за этого в системе есть огромное количество пользователей.

Непривилегированные пользователи могут временно иметь повышенные привилегии командами

- `su` – запускает терминал из-под пользователя root,
- `sudo` – выполняет любую команду из-под root,
- run executable that has SUID attribute and root owner,
- run executable that has flexible “capabilities” flags.

Межпроцессное взаимодействие в UNIX.

Способы:

- сигналы (для коротких сообщений),
- каналы и сокеты (для последовательных данных),
- процессы могут объединяться большими кусками данных через разделяемую память.

The boot process.

Во время запуска компьютера процессор находится в привилегированном режиме, дальше загрузчик (или UEFI?) выполняется в привилегированном режиме, дальше он запускает в привилегированном режиме ядро, после этого ядро запускает обычные процессы (процесс — это то, что с точки зрения центрального процессора работает в обычном непривилегированном режиме и кроме как общаться с ядром больше ничего делать не может). Самый первый процесс называется либо init, либо systemd, в зависимости от ОС. Процесс init запускает shell скрипт, который запускает все остальное. В случае с systemd он читает конфиг, выстраивает дерево зависимостей (что нужно запустить), и запускает все дочерние процессы.

Services (Daemons).

Процессы, запущенные init-ом в самом начале, называются демонами в UNIX системах и службами в Windows. Какими бывают демоны?

- `dhclient` – если компьютер подключен к сети, то этот демон периодически держит активным полученный ip адрес,
- `getty` – графический терминал?
- `sshd` – демон, который отвечает за подключение по протоколу ssh,
- `crond`,
- `ntpd` or `chronyd` – служба для синхронизации времени,
- `syslogd` – демон, который ведет системные логи

`#!/bin/sh`

Когда входим в систему первое, что запускается, это интерпретатор shell (это то, что позволяет выполнять последовательности команд). В разных системах реализован по-разному:

- FreeBSD: distinct shell program,

- Mainstream Linux'es: symlink to **bash**,
- Debian: symlink to **bash**,
- Alpine: symlink to **busybox**,
- MacOS X: symlink to **zsh**.

System Daemon (systemd)

Это некоторый процесс, который управляет всеми остальными процессами, при этом сам является обычной программой и обычным процессом.

Objectives:

- eliminate shell script interpretation,
- start services in concurrent way.

init из-за запуска интерпретатора работает дольше, чем systemd, который выстраивает лес (дерев) в каком порядке скрипты нужно запускать и может это делать параллельно.

systemd Concepts.

В UNIX системах есть команда init, которая позволяет переключить в систему какой-то из режимов выполнения.

User Sessions: Classic Unix Way.

Что происходит во время логина? После логина в систему запускается программа под названием login, которая проверяет, что пользователь корректно вел логин и пароль либо авторизовался другим доступным образом. Дальше запускается командный интерпретатор для данного пользователя и в нем можно запускать команды. В случае systemd запускаются также дополнительные службы в непривилегированном режиме.

Background Tasks.

- No associated terminal
  - signal SUGHUP
  - command **nohup** prevents SIGHUP delivery
- Output sent by mail
  - redirect output to file
- Каждая фоновая задача должна быть запущена в единственном экземпляре. Делается это с помощью lock файлов.

Лекция 2

User Session

- Each user has its own UID
- Users might be members of several groups
- User authentication:
  - login process (TTY console)
  - graphical login (sddm, kdm, gdm)
  - remote login (legacy, telnet, ssh)

- Increase rights by typing password
  - su and sudo
  - graphical tools to enter password

Пользователь — это некоторое целое число (необязательно наличие имени).

Пользователи могут объединяться в группы, причем один пользователь может быть в нескольких группах.

Авторизация пользователя происходит либо через консоль (текстовую?). Когда система только что загрузилась может быть возможность графического входа в систему, которая реализуется через дисплейный менеджер. Кроме того, пользователи могут подключатся удаленно, либо по протоколу ssh на всех современных системах, еще был старый протокол без шифрования plain текстом telnet. Сейчас telnet используется только в некоторых роутерах.

Ну и иногда пользователям необходимо повышать свои привилегии до прав суперпользователя (либо какого-то другого пользователя). Это может производится либо командами su/sudo, либо какими-то графическими tool-ами.

Pluggable Authentication Modules (PAM)

- Set of libraries to provide common API to check user login capability
- Several authentication ways:
  - /etc/passwd & etc/shadow
  - LDAP service
  - Biometry

Подключаемые модули идентификации — это некоторый набор библиотек, которые предоставляют API для авторизации пользователя, для проверки определенных прав.

А каким образом эти библиотеки позволяют выполнять авторизацию? Классический UNIX-овый способ — использовать текстовый файл etc/passwd, который содержит список всех пользователей и их параметров. Можно также использовать опечатку пальцев или инфракрасную камеру.

Creating Users

- Tools to create users:
  - useradd – standard Linux tool
  - adduser – interactive command-line tool
- User creation properties:
  - Name, groups, password, shell
  - Initial home directory contents
    (usually to be copied from /etc/skel)

adduser на самом деле скрипт, который вызывает useradd и создает нового пользователя (используется, чтобы не писать кучу аргументов).

После того, как пользователь создан, создается каталог пользователя, в который копируется содержимое домашнего каталога по умолчанию (во многих Linux дистрибутивах это содержимое располагается в файле etc/skel).

(Обычно в файлах, которые находятся в домашнем каталоге и начинаются с точки, хранятся какие-то настройки.)

Login successful! What's next?

- Each user has its own shell (see /etc/passwd)
- The shell program initializes environment variables from ~/.profile and ~/.bashrc
- Common environment variables for all Users might be specified at /etc/profile and /etc/bashrc

После логина запускается некоторый шелл, который берется с /etc/passwd. Кроме того, перед запуском шелла выполняется настройка переменных окружения (переменные окружения, это какие-то значения переменных, которые хранятся в том числе и в дочерних процессах). Также, когда запускается командный интерпретатор bash, выполняется дополнительная настройка, где тоже можно прописывать указания к переменным окружения.

Принципиальное отличие файлов ~/.profile и ~/.bashrc то, что переменные окружения, которые прописываются в профиле и то, что выполняется при инициализации профиля, это не зависит от того, какой именно шелл есть у пользователя, а в bashrc используется только bash.

#!/bin/sh

- FreeBSD: distinct shell program
- Mainstream Linux'es: symlink to bash
- Debian: symlink to dash
- Alpine: symlink to busybox
- MacOS X: symlink to zsh

All of them are POSIX-compliant.

Кроме bash-а бывают и другие оболочки с разными недостатками и достоинствами.

Environment variables

- profile – common for all shells
- bashrc (any other RC) – for selected shell

Aliases (shell-specific syntax)

Есть переменная окружения PATH, которой прописано где нужно искать файлы, которые мы запускаем без указания полного пути.

Нужно сделать export для установки переменных окружения. А что означает конструкция "export название". Допустим мы объявили какую-то переменную, и она существует в текущем экземпляре того шелла, который интерпретирует этот текст. Как только шелл завершит свою работу, либо шелл запустит еще какие-то дочерние процессы, никому эти переменные станут неизвестны. export означает, что данные переменные должны быть доступны всем дочерним процессам, а не только в текущем сеансе.

Есть еще переменная окружения LANG используется программами для того, чтобы понять на каком языке реализовать интерфейс пользователя, и переменная LC_ALL,

которая описывает какую локаль использовать. А в чем принципиальное отличие между LANG и LC_ALL? LANG обычно используется самими прикладными приложениями, либо какими-то высокоуровневыми фреймворками для того, чтобы понять на каком языке по умолчанию отображать интерфейс, а переменные LC_ALL обычно используются стандартной C библиотекой, в частности в функциях ввода-вывода, в форматировании дат и т.д., т.е. это более низкоуровневая вещь. Причем значения LANG и LC_ALL обычно подразумевают не только страну, язык, но и используемую кодировку.

Alias — это некоторый текст, который превращается в какую-то команду.

Почему команда cd не может быть в принципе реализована в виде отдельной программы? По той причине, что она меняет текущее состояние процесса.

Text Encodings

- Wide symbol (wchar_t, std::wstring):
  - typedef std::basic_string<wchar_r> std::wstring;
  - 16 bits for MSVC
  - 32 bits for glibc
- Sequence of bytes (char, std::string):
  - typedef std::basic_string<char> std::string;
  - each element is one byte

Есть два вида кодировок (способы кодирования символов). Один способ — использовать несколько байт на один символ. Либо можно хранить текст как последовательность отдельных байт.

Example: Russian Language

- ANSI (7 bits): just US/UK English language support
- KOI8-R and KOI8-U: one bit to switch language
- OEM Encoding (IBM866): hardware support
- ANSI Encoding (CP1251): all cyrillic symbols from Microsoft

man 7 utf8

```
0x00000000 - 0x0000007F:
0xxxxxxx

0x00000080 - 0x000007FF:
110xxxxx 10xxxxxx

0x00000800 - 0x0000FFFF:
1110xxxx 10xxxxxx 10xxxxxx

0x00010000 - 0x001FFFFF:
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

0x00200000 - 0x03FFFFFF:
111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

0x04000000 - 0x7FFFFFFF:
1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
```

Универсальным способом, который покрывает абсолютно все языки и при этом является обратно совместимым с кодировкой ASCII — кодировка utf8.

wide character — это 2 или 4 байта на символ, но это не кодировка utf16, а что-то, что численно совпадает с кодировкой utf16 (тут тоже может быть переменное количество байт).

User Login

- login + shell
- startx + xinitrc
- Graphical session components:
  - X11 server (Wayland in future)
  - Display manager
  - Window manager
  - Desktop application

Есть разные переменные окружения, разные алиасы, которые являются атрибутом процесса и затем наследуются всеми дочерними процессами.

А что происходит после того, как пользователь сделал логин и получил возможность запускать новый процесс? После процесса login запускается shell. Затем можно набирав команду startx запустить графический интерфейс из-под своего имени, которая выполняет содержимое файла xinitrc.

В современных системах (если это не серверная конфигурация) запускается сразу сеанс с графическим рабочим столом, который в свою очередь состоит из нескольких составных частей. Ключевой момент — это некоторый X server, который отвечает за взаимодействие с видеокартой, клавиатурой и всем остальным. Дальше, одна из ключевых компонент — это display manager, тот самый, который отвечает за авторизацию пользователя, ввод пароля и запуск нужного сеанса. Дальше есть менеджер окон и еще может быть (опционально, необязательно) какая-то программа, которой прописана свойство X сервера, что она всегда находится фоном.

X server

- Server is a process responsible to accept connections from client
- Any GUI application requires connection to DISPLAY via socket
- Connection might be local or network

```
ssh -X user@hostname
```

Сервер — это какой-то процесс, который принимает входящие соединения от одного или нескольких клиентов и как-то их обрабатывает. Сервер — это некоторая программа, которая имеет доступ к видеокарте, клавиатуре, мышке.

Когда запускаем некоторое графическое приложение, то оно устанавливает соединение через локальный сокет с сервером. Переменное окружения DISPLAY содержит IP адрес и номер дисплея, которому нужно подключиться (env | grep DISPLAY в терминале).

Applications

- Software packages:

    - plain archives (FreeBSD)
    - RPM (rpm to install, rpmbuild to build)
    - DEB (dpkg to install, debuild to build)
- Package:
    - set of files
    - install/uninstall scripts
    - metainformation: name, version, description, provides/requires

Приложение — это некоторые пакеты, которые могут быть как обычными архивами, так и архивами нестандартного формата в некоторых UNIX дистрибутивах.

Пакет — это какой-то набор файлов, которые относятся к определенному приложению, к определенной команде и т.д. Плюс к этому некоторые дополнительный набор информации — метаинформация о том, что это за пакет, версия, описание. Ну и самое главное в пакетах кроме самих файлов содержится еще дополнительная информация о зависимостях пакетов между собой.

Applications

- Packages dependencies on each other
    - libraries of exact versions
    - software might be split into several packages: noarch and arch-specific parts
- Package managers:
    - solve dependencies
    - download from remote repositories
- Examples:
    - apt-get (apt) from Debian/Ubuntu (dpkg)
    - yum (dnf) for Fedora (rpm)
    - apt-get for AltLinux (rpm)

Есть зависимости от разных библиотек, в том числе бывают требования к минимальной версии или даже конкретной версии.

Есть высокоуровневые инструменты apt-get, yum и т.д., задача которых при попытке установить какой-то софт найти метаинформацию из локальных репозиториев, либо из удаленных сетевых, выстроить дерево зависимостей, все выкачать и установить в нужном порядке. Вот этим и занимаются пакетные менеджеры, которые работают с репозиториями. При этом это надстройки, которые вызывают соответствующие утилиты.

Source-Based Installation

- Most programs for Linux are Open Source
- Archive of source files
- Requires development packages for dependent libraries
- Usually but not always uses common installation method:

```
./configure

make

make install # as root
```

Есть более жесткий способ установки — это ставить что-то из исходных текстов.

В каждом пакете есть скрипт под названием configure. Это обычный баш скрипт, который генерирует файл под названием makefile.

Source-Based Installation

- The root directory / has bin, lib etc.
  - emergency set of files for some distros
- The /usr directory has similar layout
  - software managed by packaged system
- The /usr/local directory
  - software NOT managed by package system, default prefix for ./configure scripts (might be overridden by –prefix=… option)

Search path: PATH=/bin:/usr/bin:/usr/local/bin

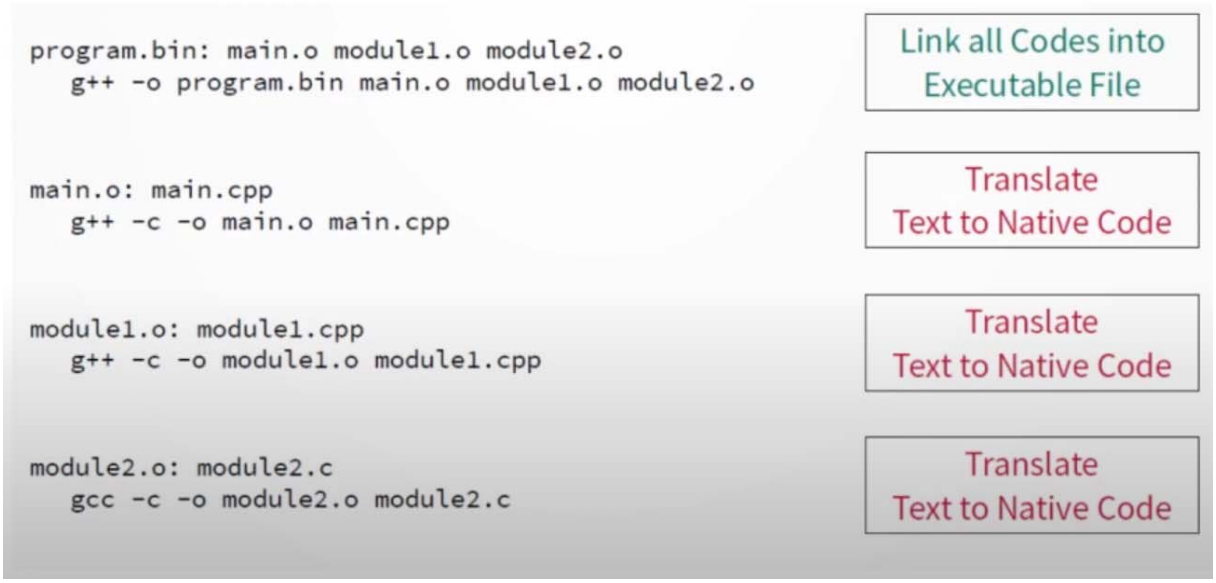
Как устроены иерархии каталогов? Есть корневой каталог, который содержит bin, lib, какие-то утилиты, есть еще какой-то странный каталог usr, который содержит те же самые подкаталоги с тем же расположением. В корневой каталог помещается все, что востребовано в качестве минимальной системы, т.е. минимальный набор файлов, который необходим, чтобы можно было посмотреть состояние системы, что-то подмонтировать и т.д.

Makefile

- A list of targets and rules to build targets
- Might have various syntax (GNU make, BSD make, Microsoft nmake

Ну и что такое makefile, который был создан скриптом configure? Это некоторый текстовый файл, который содержит список целей и некоторые последовательности команд для достижения этих целей.

Compile stages



Executables and Libraries

- Static Library
  - just an indexed archive of object files

    - rarely in use today
- Dynamic Library
    - like a program executable file
    - has no entry point
    - contains a symbol table

    (objdump -t)

- gcc/clang options:
    - -L – add path to search libraries
    - -l – provide canonical library name to link

Что можно собирать с помощью makefile-а? Какие бывают виды целей? Это могут быть как отдельные программы, так и отдельные библиотеки.

Библиотеки тоже бывают разные. Бывают статические библиотеки, т.е. вот мы получили некоторый набор объектных файлов, дальше их запихиваем с помощью команды ??? в индексированный архив и по необходимости при линковке оттуда файлы извлекаются и сделается линковка с нужными программами.

Динамические библиотеки представляют собой те же самые elf файлы, что и обычные программы, только у них дополнительно есть таблица символов и нет точки входа.

Makefile Generation

- ./configure script
    - pros: requires just a shell and basic tools
    - cons: hard to maintain. Solution: autottools
- qmake/jam/scons/custom scripts
- cmake

Файл configure генерируется с помощью пакета autotools.

CMake

- Universal C and C++ tool to generate:
    - Makefiles (GNU and Microsoft)
    - Projects for Visual Studio, XCode and CodeBlocks
- A lot of helper modules for Open Source packages

Software Distribution

- Provide project file to build: ./configure, CMakeLists.txt etc.
- Provide ./debian/ files (debuild) for Ubuntu/Debian
- Provide package.spec (rpmbuld) for various distributions
- Ubuntu Launchpad https://launchpad.net
- openSUSE Build Service https://buld.opensuse.org

Лекция 3 - Integer Arithmetic

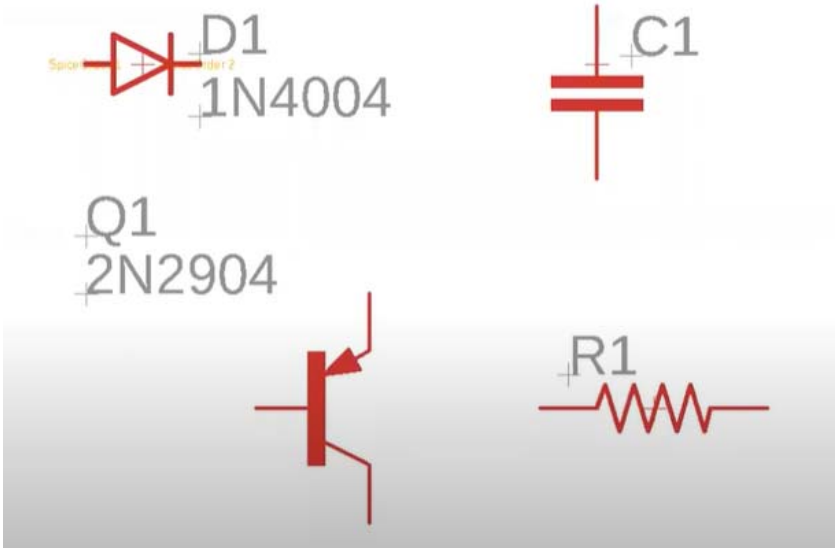
Hardware Abstraction Levels

1. “p” and “n” semiconductors, conductors, capacitors, resistors
2. Transistors, diodes

3. Logic gates
4. Arithmetic – Logical Unit
5. CPU itself
6. Complete computer system

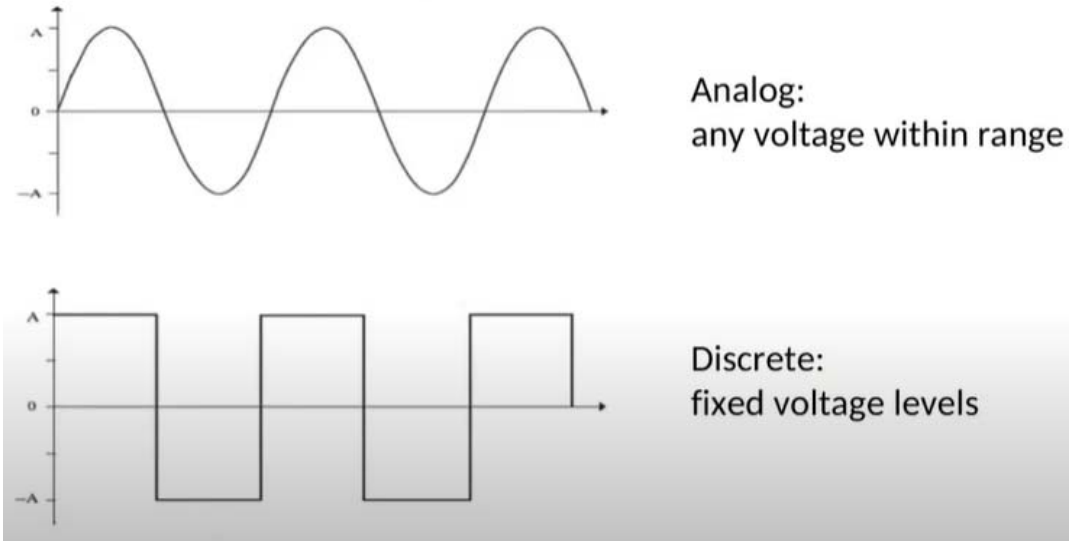
Есть какие-то элементарные блоки, из которых строятся дальше какие-то электрические элементы, они в свою очередь образуют какие-то логические элементы, и главная штука в любом компьютере — это центральный процессор. В свою очередь в центральном процессоре ключевой блок — это арифметический-логический блок, который выполняет все вычисления.

Atomics



Будем считать атомарными неделимыми объектами некоторые базовые блоки с области радиоэлектроники — диоды, транзисторы, емкости (конденсаторы) и сопротивления (резисторы).

Signal Types



У нас есть электрическая цепь, по которой ходят какие-то сигналы. Сигнал — это изменение напряжения, которое можно регистрировать и дальше с ним что-то делать.

Сигналы бывают двух видов — либо аналоговые (микрофон, наушники), либо цифровые, которые более надежные. Принципиальное отличие между этими двумя видами сигналов то, что в цифровых мы фиксируем только два состояния — высокие или низкие.

Arithmetic-Logical Unit

- Core CPU unit to implement integer operations:
  - bitwise logic
  - bitwise shifts and rotations
  - sum/subtract
  - multipy/divide
- Program control commands are arithmetic too

Из предыдущих маленьких блоков составляется арифметически-логическое устройство, которое должно выполнять операции над значениями маленьких областей памяти, которые называют регистрами.

Integer Overflow (add or subtract operation)

- Unsigned values:
  - modulo value
  - might be checked by criteria X + Y = Z Z >= X and Z >= Y
- Signed values:
  - like unsigned by hardware design
  - Undefined Behavior by C or C++ standard

Overflow Harm

- Inaccurate results
- Infinite loops
- Depends on input data

Implicit Checking (gcc extention)

- Non-Standard Functions
  -std=gnu11 vs -std=c11
- Available in recent versions of gcc/clang

Sanitizers

- Additional code generated by compiler
- Some options:
  - Undefined Behavior: -fsanitize=undefined
  - Address: -fsanitize=address
- Requires runtime support
- Much faster than valgrind

На случаи runtime-а есть механизм, когда можно контролировать определенные ситуации, который называется санитайзером (в языках C и C++).

Чем отличается программа с санитайзером от программы без санитайзера? Понятно, что она будет работать медленнее.

valgrind позволяет контролировать проблемы, связанные с памятью.

Multiple Bytes Value

- Minimum data size is 8 bits
- Each processor can read/write machine word at one clock
- Bit order not specified
- Byte order implementation defined

Разрядность процессора — число бит, которые процессор может прочитать за один такт.

Big-Endian vs Little-Endian

- Big-Endian:
  looks like human natural representation — high bytes first, low bytes last
  Example: most RISC processors by default
- Little-Endian:
  low bytes first, high bytes last
  Example: Intel family processors, ARM
- Most RISC processors might switch between Big/Little endian

Есть два основных класса архитектур big endian и little endian.

Structs Packing

- Machine word size is 32 bits (x86)
- Compiler optimizes layout for load/store speed but not memory conservation

Размер структуры должен быть кратен размеру машинного слова.

typedef struct SomeStruct {} __atribute__((packed)); // GCC and CLang specific extension

Последнее позволяет занимать в точности столько памяти, сколько суммарно составляют поля структуры.

Where to Use Packed Structs

- Binary data files
- Network packets
- Device drivers

Лекция 4 – Команды процессора

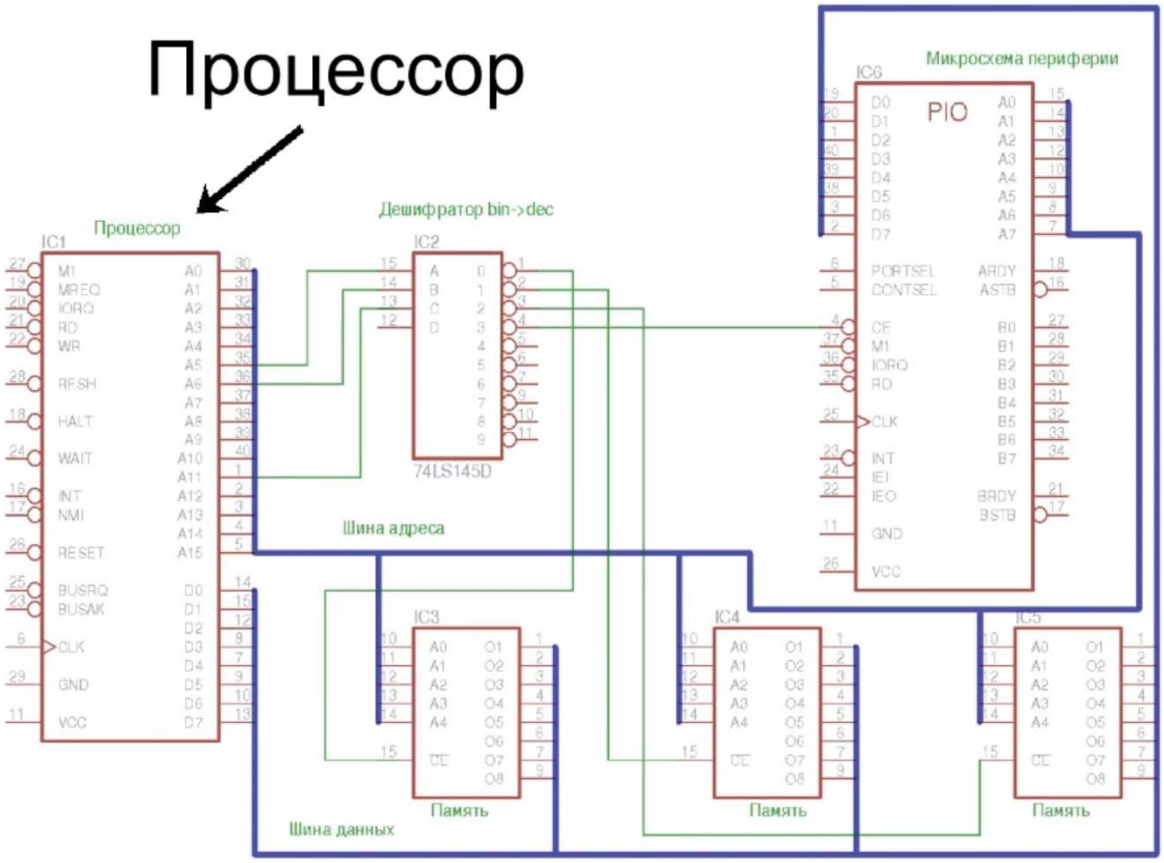
Процессор

У процессора есть шина адресов и шина данных. Что делает процессор? Он только умеет манипулировать целыми числами, и все! При этом значение адреса тоже целое число.

Как происходит общение процессора с внешним миром, например с памятью или периферийными устройствами ввода-вывода? Восставляется определенный адрес на шине адреса, который дальше дополнительно может быть декодирован и

определено устройство, с которым процессор обращается, и есть шина данных с размером машинное слово — это через которую что-то передается.

Шина — последовательность контактов. Бывают шины параллельные, бывают и последовательные. Параллельные шины обычно имеют большое количество контактов для того, чтобы за один такт передавать большое количество данных.



С точки зрения пользователя (на примере i386)

Регистры	
%eax	%esi
%ebx	%edi
%ecx	%ebp
%edx	%esp

Флаги		
ZF	CF	SF

Указатель на текущую команду
PC (32 бит)

Что еще есть у процессора кроме арифметико-логического устройства? У процессора есть несколько регистров. Регистр — некоторая элементарная ячейка памяти внутри самого процессора, с которой можно выполнять любые действия. Вне регистров, с произвольными данными в памяти процессор ничего делать не умеет. Т.е. предварительно требуется загрузить что-то из памяти либо из какого-то устройства, выполнить операцию и после этого сохранить все обратно.

Флаги — это некоторые бинарные значения, хранятся в некотором регистре.

Команды x86

Байты	0	1	2	3	4	5
nop	0x00					
halt	0x10					
rrmovl *rA*, *rB*	0x20	*rA rB*				
irmovl *V*, *r*B	0x30	0x8 *rB*	*V*			
call *Dest*	0x80	*Dest*				
pushl *rA*	0xA0	rA 0x8				
popl *rA*	0xB0	rA 0x8				

Команды кодируются переменным количеством байт

Какие команды выполняет процессор

- Арифметические
- Управляющие

Представление структур программы в виде простых инструкций

```
while (true)
{
    <statement 1>
    . . . .
    <statement N>
}
```

```
Loop_Start:
    <statement 1>
    . . . .
    <statement N>
    br %Loop_Start
```

```
while (<cond>)
{
    <statement 1>
    . . . .
    <statement N>
}
```

```
Loop_Start:
    %cond = . . .
    br i1 %cond,
        label %Loop_Body,
        label %Loop_End
Loop_Body:
    <statement 1>
    . . . .
    <statement N>
    br %Loop_Start
Loop_End:
    . . . .
```

```
for (<init>;<cond>;<incr>)
{
    <statement 1>
    . . . .
    <statement N>
}
```

```
    <init statements>
Loop_Start:
    %cond = . . .
    br i1 %cond,
        label %Loop_Body,
        label %Loop_End
Loop_Body:
    <statement 1>
    . . . .
    <statement N>
    <increment statement>
    br %Loop_Start
Loop_End:
    <cleanup statements>
    . . . .
```

Наборы команд Z80, x86 и PDP-11/VAX

Complex Instruction Set Computing (CISC)

- Их много — на все случаи жизни
- Кодируются переменным числом байт
- Разные режимы адресации
- Упрощают жизнь программисту на ассемблере/машинных кодах

Наборы команд делятся на 2 больших класса. Один из них (наиболее старых) — набор составных сложных инструкций.

Примеры команд Intel x86 со сложной логикой

loop Address

1. Уменьшает значение регистра %ecx на 1
2. Если значение %ecx==0,
   то %eip+=sizeof(loop),
   иначе %eip=Address

movl *src*, Offset(*base*, *index*, Size)

1. Считывает значение из регистра *index*
2. Умножает его на Size
3. Прибавляет к нему значение из регистра *base*
4. Прибавляет к нему значение Offset
5. На шине адреса выставляет полученное значение
6. Записывает значение из регистра *src* в память

Программирование CPU

Совсем на низком уровне:

- Машинные коды
- Ассемблер
- Макро-ассемблер

Высокоуровневые языки:

- 1966: BCPL (Basic Combined Programming Language)
- 1969: язык Би (B)
- 1972: язык Си (C)

Конвейер

- Instruction Fetch
- Instruction Decode
- Execute
- Memory Access
- Register Write Back

Каждая команда проходит несколько стадий. Сначала ее нужно загрузить из памяти в некоторый локальный кэш (fetch). Дальше команда должна быть декодирована (decode). Что значит, что нужно выделить по определенной битовой маске, во-первых, что это за команда, какие у нее операнды, определить какому блоку процессора она относится, и вот только после этого происходит ее фактическое выполнение. Дальше, если у команды есть результат (а у большинства команд он есть), его нужно записать обратно. Для этого сначала нужно выставить некоторое число на шине адреса и после этого возможно записать значение какого-то определенного регистра в буфер, связанный с шиной данных.

Проблемы конвейеризации

- Инструкции имеют разную длину в байтах
- Разные инструкции выполняются за различное число тактов
- Работа как с памятью, так и с регистрами — задействуются разные блоки процессора

Процессоры AVR, ARM, MIPS, PowerPC, …

Reduced Instruction Set Computing (RISC)

- Только простейшие инструкции фиксированной длины и (для большинства) с одинаковым временем выполнения
- Адресация только R-R

Обращение к памяти RISC vs CISC

x86

```
01 05 64 94 04 08     addl 0x8049464, %eax
```

6 байт в одной составной команде

AVR

```
c4 e3               ldi 29, Low(0x1234)
d2 e1               ldi 28, High(0x1234)
18 80               ld  r1, Y
01 0c               add r0, r1
```

8 байт в четырех простых командах

RISC для x86 ISA

- i486 – появление конвейера только для подмножества простых команд
- i586 (Первый Пень) – два конвейера: для простых команд и для не очень простых команд
- i686 (современная архитектура IA-32) – микроядро RISC; команды CISC предварительно транслируется во внутреннее представление самим процессором (микрокод)

Современные RISC-процессоры

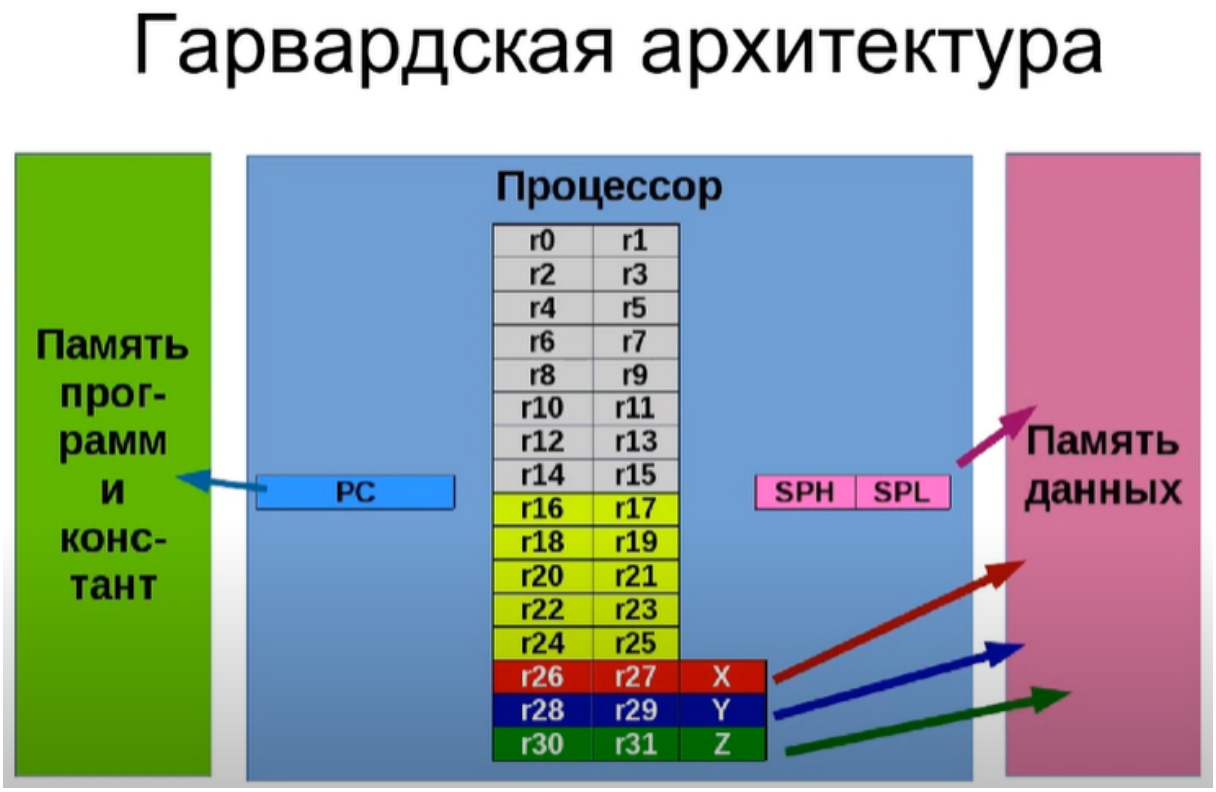
- PowerPC: Playstation 3, XBox 360, суперкомпы IBM
- MIPS: WiFi/Bluetooth-адаптеры, DSP в телевизорах
  Процессоры 1890ВМ9Я (2 ядра, 1ГГц) и 1907ВМ028
  (всего 1 ядро и смешные 150МГц, зато в космос летает)
- ARM: весь китайский ширпотреб (iPhone etc.)
- AVR: Arduino и его клоны + еще много где встречается

Микроконтроллеры AVR

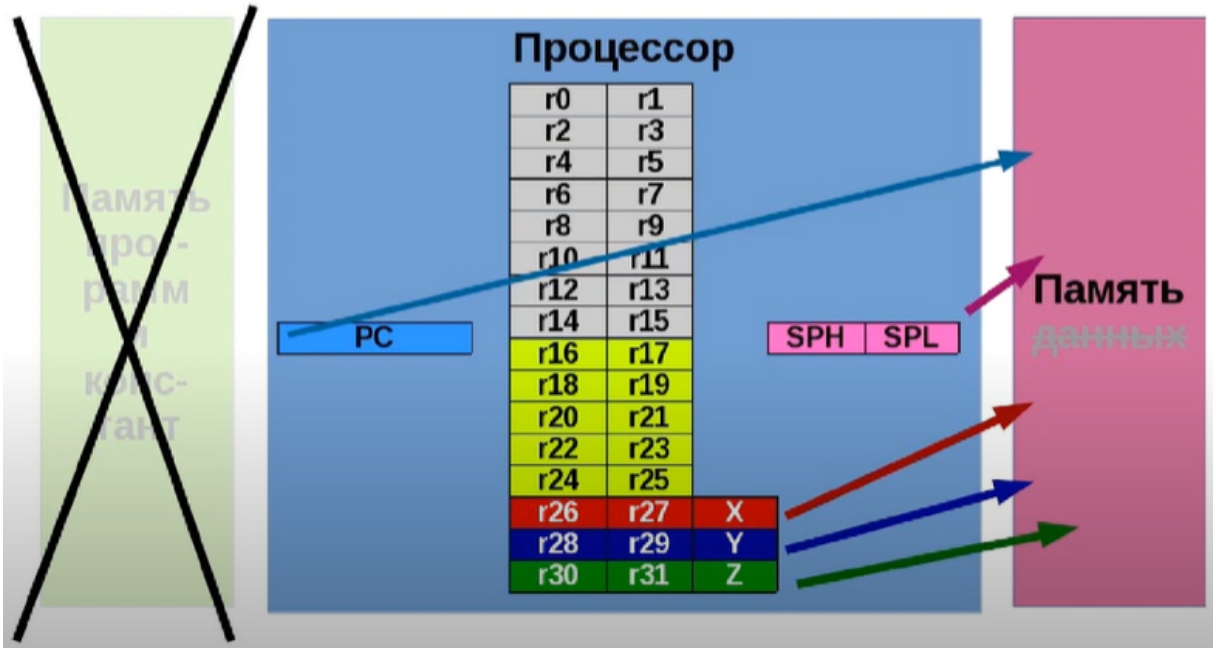
- Гарвардская архитектура, а не Фон-Неймана
- 8 бит
- Встроенная память ОЗУ и ПЗУ

По сути — “система на кристалле”

Микроконтроллер — это система на кристалле, очень маленькая, слабенькая, возможности которой на все хватают. Это полноценный компьютер, поэтому обычно у них нет никаких внешних шин адреса и данных, есть просто простые шины USB, SBI, именно для подключения периферийных устройств.



Архитектура Фон-Неймана



Для массовых компьютеров распространена архитектура Фон-Неймана, а гарвардская архитектура используется там, где устройство делается с ограниченной функциональностью, которое должно просто надежно работать.

Микроконтроллеры AVR

Atmel ATtiny13A

Процессор 8-бит

Встроенная SRAM (64 байт)

Встроенная Flash-память (1К)

Встроенная EEPROM (64 байт)

Таймер

АЦП (10 бит)

- Всего 8 выводов у микросхемы

Архитектура ARM (32 бит)

- Единый набор команд для разных типов устройств:
  - Cortex-M – микроконтроллеры
  - Cortex-A – процессоры для ПК/смартов/таблеток
  - Cortex-R – промышленные процессоры
- FPU является опциональным, есть не на всех ядрах
- На некоторых ядрах есть SIMD-инструкции
- 13 32-битных регистров общего назначения

- Инструкции условного выполнения (убрали из архитектуры AArch64)

Вещественная арифметика – IEEE 754

Представление IEEE754

Single-Precision (32 бит); B = 127		
S	E (8 бит)	M (23 бит)

Double-Precision (64 бит); B = 1023		
S	E (11 бит)	M (52 бит)

$$Value = (-1)^{S} \cdot 2^{E-B} \cdot (1 + M / (2^{23 \text{или} 52} - 1))$$

Специальные значения

S	E	M	Значение
0	0	0	+0
1	0	0	-0
0	11...11	0	+ ∞
1	11...11	0	- ∞
0	11...11	≠ 0	Signaling NaN
1	11...11	≠ 0	Quiet NaN
0	0	≠ 0	Денормализованные значения
1	0	≠ 0	

Денормализованные числа

Single-Precision (32 бит)		
S	0000 0000	M (23 бит)

Double-Precision (64 бит)		
S	000 0000 0000	M (52 бит)

$$Value = (-1)^{S} \cdot M / (2^{23 \text{или} 52} - 1)$$

Денормализованные значения — расширение стандарта, и не гарантируется, что в каждой архитектуре будет их поддержка.

Операции: умножение

$<S,E,M> = <S_1,E_1,M_1> \cdot <S_1,E_1,M_2>$

1. Вычислить
$S = S_1 \wedge S_2$
$E = E_1+E_2$
$M = 1.M_1 \cdot 1.M_2$

2. { M>>=1 ; E++ }пока переполнение M

Операции: сложение

$<S,E,M> = <S_1,E_1,M_1> \cdot <S_1,E_1,M_2>$

1. Вычислить $E_{diff}=E_1-E_2$
2. Нормализовать M_2 на E_{diff} бит
3. Значения:
$E = E1$
$M = M1 +/- M2$
$S = sign(-1^{S1}M_1 + -1^{S2}M_2)$

Реализации FPU

- Расширенный набор команд (ARM VFP):
  - Дополнительные команды
  - Дополнительные 32 регистра
- Сопроцессор x86:
  (gcc -mfpmath=387)
  - Команды, которые CPU делегирует FPU
  - Взаимодействие через стек
- Команды SSE (Pentium-III+, x86-64):
  (gcc -msse -mfpmath=sse)
  - Используются регистры xmm
  - Используются скалярные команды SSE

Точность

Floating Point

- Single Precision – 32 bit ≈ 10^{37}
- Double Precision – 64 bit ≈ 10^{307}

Лекция 5 – Языки ассемблера и двоичный код

Стадии трансляции

- Препроцессинг текста; на выходе — текст
- Из исходного текста — Abstract Syntax Tree и таблицы символов; формат — внутренний для реализации компиляторов
- Из AST получаем:
  - либо код на языке ассемблера (текст)
  - либо двоичный код

Если говорим про компилируемый язык, то сначала выполняется препроцессинг — обрабатываются директивы. Дальше проводим синтаксический разбор, строится абстрактное дерево разбора. Дальше выполняется либо статический анализ кода, либо генерируется какой-то побочный продукт.

Языки ассемблера

- Последовательность команд — линейная
- Нет вложенных конструкций
- Не бывает вычислимых выражений (точнее, это syntax sugar)
- Текст распознается регулярным выражением, а не контекстно-свободной грамматикой

Принципиальное отличие ассемблера от других языков то, что в нем нет структур, значит и вложенностей тоже, и текст можно распознавать регулярным языком.

- Из AST получаем:
  - либо код на языке ассемблера (текст)
  - либо двоичный код
- Из текста программы можно выделить отдельные фрагменты, которые кодируют бинарные инструкции
- Последовательность бинарных инструкций позволяет восстановить текст на ASM

Кодирование команд

- RISC: одна команда — это машинное слово
  - зная начало кода, можно извлечь любую инструкцию
- CISC: одна команда — это произвольное количество байт
  - нужно прочитать и декодировать несколько байт, чтобы найти следующую команду

Команды и регистры

- Процессор, в общем случае, может выполнять некоторые действия только над значениями регистров
- Регистр — не просто быстрая ячейка памяти, а ячейка, к которой можно обратиться напрямую
- Доступ памяти — достаточно сложная операция
- Но: некоторые процессоры (x86) имеют разные способы адресации, позволяющие адресовать память

Команда — это некоторые действия над регистрами.

Кодирование команд (AVR)

ADD – Add without Carry

Description

Adds two registers without the C Flag and places the result in the destination register Rd.

Operation:

(i) (i) Rd ← Rd + Rr

	Syntax:	Operands:	Program Counter:
(i)	ADD Rd,Rr	0 ≤ d ≤ 31, 0 ≤ r ≤ 31	PC ← PC + 1

16-bit Opcode:

0000	11rd	dddd	rrrr

LDI – Load Immediate

Description

Loads an 8-bit constant directly to register 16 to 31.

Operation:

(i) Rd ← K

	Syntax:	Operands:	Program Counter:
(i)	LDI Rd,K	16 ≤ d ≤ 31, 0 ≤ K ≤ 255	PC ← PC + 1

16-bit Opcode:

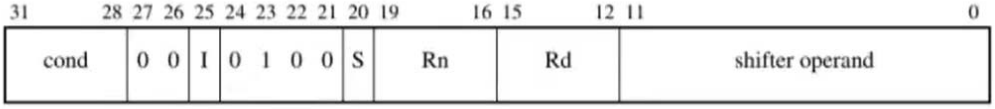
1110	KKKK	dddd	KKKK

Что нужно закодировать (или что делает ассемблер)

- Какая именно команда
- С какими регистрами она работает
- Константы:
  - могут встречаться в численных операциях
  - адреса (метки) в инструкциях перехода

Кодирование ARM-32

ADD



- **cond** - условие выполнения команды
- **I** - флаг, определяющий, что закодировано в **shifter_operand**
- **S** - флаг, определяющий, нужно ли обновлять регистр статуса
- **Rn**, **Rd** - первый аргумент операции и куда записать результат

Примеры:

```
ADD r0, r1, r2  // r0 ← r1 + r2
ADDS  r0, r1, r2  // r0 ← r1 + r2, обновить флаги Z,V,C,N
ADDEQ  r0, r1, r2  // Если Z, то r0 ← r1 + r2
ADD r0, r1, r2, lsl #3  // r0 ← r1 + (r2 << 3)   [5bit shift; 2bit type; 0; 4bit reg]
ADD r0, r1, #0xFF  // r0 ← r1 + 0b0000'1111'1111
ADD r0, r1, #0x200 // r0 ← r1 + (0b0000'0010 ROR 0b1100)
```

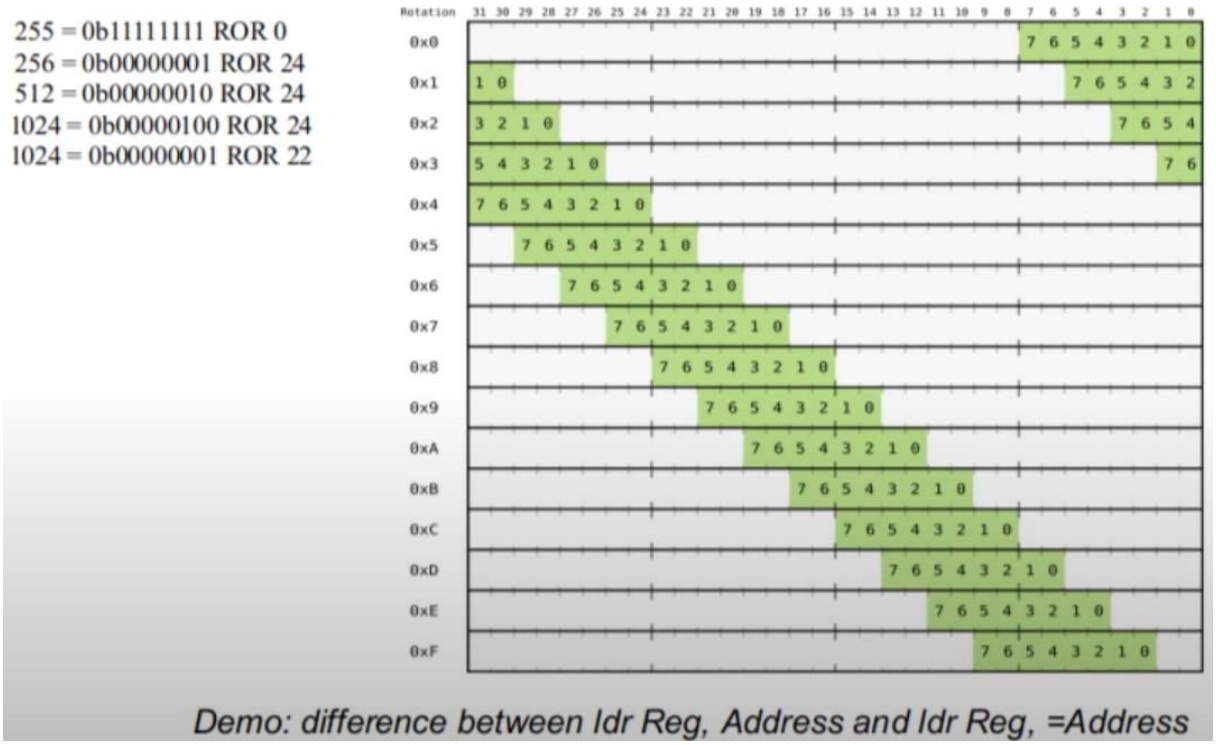
Ключевая проблема

Нельзя впихать невпихуемое

- Длина команды — 32 бит
- Часть из них заняты OPcode и номерами регистров

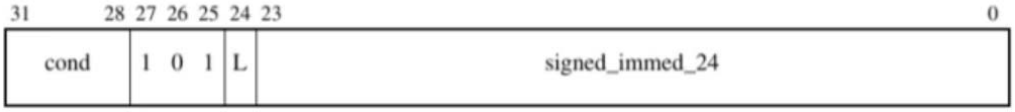
Циклический сдвиг констант (ARM)

- Под константы зарезервировано 12 бит
- 8 бит — значение
- 4 бит — сдвиг со вращением



Кодирование ARM-32

B, BL



$$+/- 2^{23} \quad \text{- это } +/- 8\text{Мб}$$

Переходы на метки

- В пределах +/- 8Мб – нет проблем
- Прыжок на далекую функцию — через “трамплин”:
    - одна из причин использования PLT

Procedure Linkage Table

```
function@plt:
   add  ip, pc, #0
   add  ip, ip, #OFFSET_TO_TABLE_BEGIN
   ldr  pc, [ip, #OFFSET_TO_FUNCTION_INDEX]
```

Мы перескакиваем на некоторую инструкцию, которую мы знаем относительно нашей текущей позиции, где хранится этот код (и он хранится не очень далеко от нашей основной программы), и содержит, по сути, некоторую таблицу, в которой записаны адреса (причем реальные 32-битные адреса) отдельных функций из каких-то сторонних библиотек (которые находятся очень далеко от нас). Таблица находится в некотором фиксированном месте и ее адрес определяется на этапе компиляции (точнее на этапе компоновки). Дальше происходит загрузка какой-то произвольной программы, и одна из стадий (до того, как выполняется функция нижнее подчеркивание main) загрузчик обязан найти все зависимые библиотеки и после этого загрузить библиотеки в память. И следующий этап — это пройтись по заголовкам ELF файлов и по им таблицам символов, и прописать нашей программе таблицу под названием Procedure Linkage Table. Собственно, из этой таблицы и берутся все используемые функции.

Что означает загрузить в регистр program counter какое-то значение по определенному фиксированному адресу из таблицы? Это означает переключить выполнение на определенную функцию.

Procedure Linkage Table

- Решает проблему “длинных” прыжков
- Позволяет размещать часть кода в заранее неизвестном месте адресного пространства

Про кодирование команд

- Расширения ARM:
    - Thumb — 16 битные инструкции
    - Jazelle — декодирование байткода Java

Про кодирование команд

- Регистровые машины выполнения
- Стековые машины выполнения

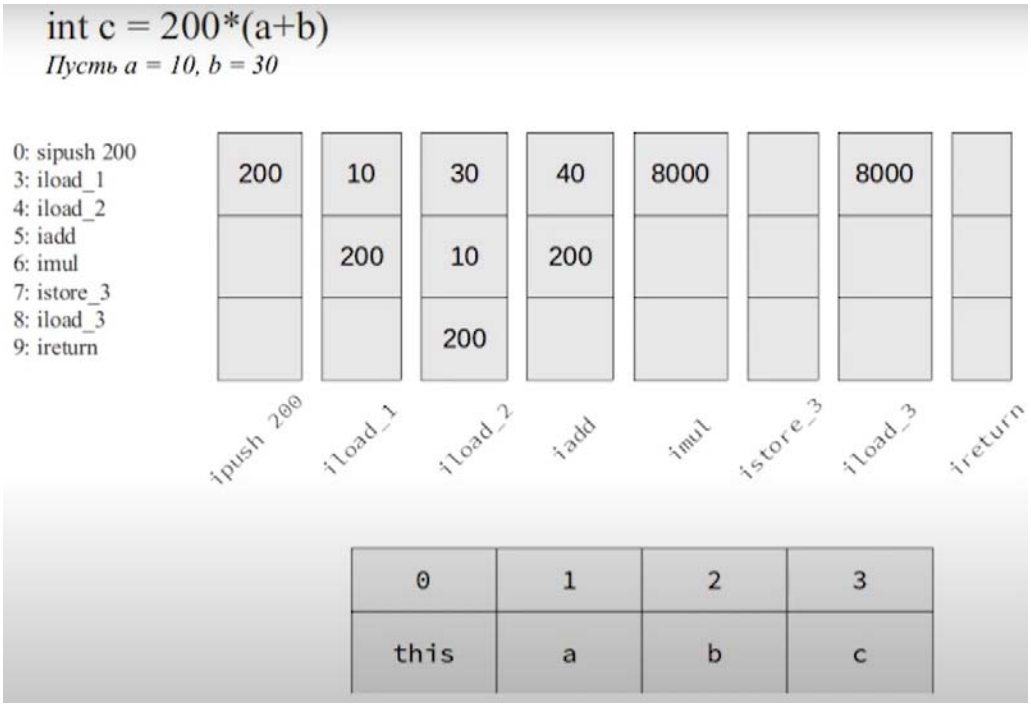
Какие вообще бывают разные способы выполнения команд. Есть два вида. Все процессоры так называемые регистровые машины выполнения. Т.е. если пишем эмулятор, то нужно эмулировать ограниченное количество регистров и писать код таким образом, чтобы вместится в это ограниченное количество регистров, ну а для всего остального периодически использовать подгрузку из памяти.

Другой класс машин. В случае с Java есть два вида байткода. Один из них это классический ??? ??? байткод, который обычной средой Java десктопно генерируется это как раз стек, принимающий для стековой машины выполнения

Байткод Java (вывод javap)

```
....
2: int someFunc(int a, int b){
3:     int c = 200*(a+b)
4:     return c;
5: }
....
```

```
int someFunc(int, int);
  descriptor: (II)I
  flags:
  Code:
    stack=3, locals=4,
args_size=3
     0: sipush 200
     3: iload_1
     4: iload_2
     5: iadd
     6: imul
     7: istore_3
     8: iload_3
     9: ireturn
LineNumberTable:
     line 3: 0
     line 4: 8
```



Уровни абстракции

- Высокоуровневый язык
- Intermediate Language:
  - bytecode (Java, CLI или PyPy)
  - LLVM — абстракция от ассемблера разных архитектур
- [Язык ассемблера]
- Двоичный код

Зачем понимать кодирование команд

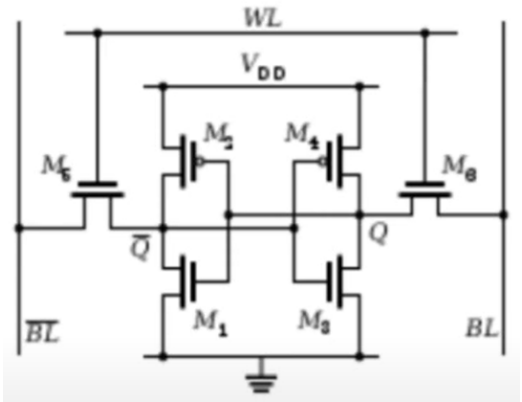
- Just-In-Time компиляция
- Эмулятор компьютерной системы
- Трансляция команд
  - Apple Rosetta
  - qemu user mode

Лекция 6 — Способы ускорения выполнения кода

SRAM

- Время чтения — 1 такт
- Время записи — 2 такта
- Тактовая частота — в зависимости от размеров транзистора

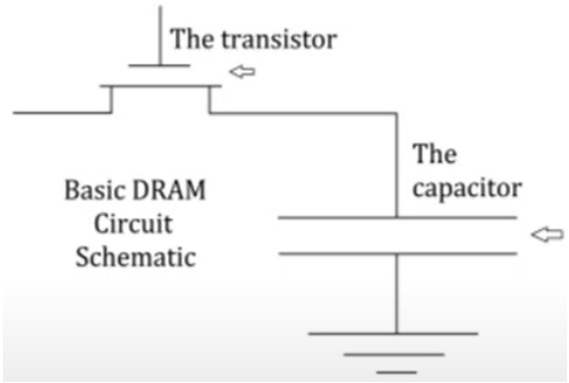
В процессоре есть регистры. Вообще говоря, классический процессор, если не рассматриваем микроинструкции, которые умеют адресовать напрямую память, то все операции выполняются над регистрами и кроме как с регистрами процессор не умеет ничего делать. Регистры работают очень быстро, они обычно построены на ячейках статической памяти, к которым можно одномоментно обращаться. Регистров мало, поэтому приходится задействовать оперативную память и для реализации оперативной памяти есть две основных технологии. Ну, во-первых, память можно построить на транзисторах, каждая ячейка памяти — от 4 до 6 транзисторов. Это называется статическая память, где время чтения — это ровно 1 такт процессора, и вот время записи в разных реализациях бывает либо 1 такт, либо 2, но тоже очень быстро.



DRAM

- 1 транзистор + 1 конденсатор
- Конденсатор требует перезарядки после каждого чтения
- Чтение/запись затратны по времени
- Периодическая регенерация (каждые 64 нс) из-за утечек

Еще бывает другой тип памяти — это транзистор, который забирает ячейку + конденсатор. На микросхеме конденсатор — это две параллельно идущие дорожки на керамическом подложке и заряд конденсатора определяется 1 либо 0.



В чем разница между транзисторной схемой статической ячейки памяти и схемой с конденсатором и запирающим транзистором? То, что конденсаторы + запирающие транзисторы занимают намного меньше места и поэтому объемы памяти, которые можно сделать на схеме с динамической памятью, заведомо сильно превышают те объемы памяти, которые можно реализовать на микросхеме той же площади используя статическую память. Более того сами конденсаторы на самом деле даже не нужно отдельно ставить, поскольку на микросхеме при разводке схемы, если использовать правильную топологию размещения, то в качестве конденсаторов можно сами проводники использовать.

DRAM vs SRAM

DRAM	SRAM
• 2 элемента на ячейку • Медленный доступ • Требуется схема для перезарядки конденсаторов	• 6 элементов на ячейку • Время доступа ограничивается тактовой частотой • Простая шина

Локальность доступа

- Локальность по времени
  - фрейм текущей функции
  - статические и thread-local переменные, используемые в настоящий момент
- Локальность по расположению данных
  - код программы или библиотеки
  - структуры и массивы (в т.ч. — многомерные)

Возникает иногда проблема, связанная с локальностью доступа. Т.е. есть данные, которые нам нужны именно сейчас, и мы примерно знаем, что нам не нужна вся память, нам нужен только определенный кусок и почему бы не запихать его в какую-то более быструю память. Такая память называется кэш.

Информация в Linux

- sys/devices/system/cpu/cpu0/cache
  - index0
  - index1
  - index2
  - index3
- cpuX — отдельное ядро (они одинаковые, так что все равно)
- indexY — отдельные кэши кэша

Т.е. кэш — некоторый небольшой кусок памяти, который имеет размер заведомо меньше, чем размер всей памяти, которую можем адресовать, но работает намного быстрее и при этом хранит нужный нам кусок.

Уровни кэша (в x86)

- L1 (index0 + index1 для Intel) — самый близкий кэш микроинструкций и текущих данных, с которыми оперируют микроинструкции

- L2 — общий кэш, связанный с ограниченным количеством ядер (как правило — одним, но не всегда)
- L3 — общий кэш, связанный со всеми ядрами

Характеристики кэшей

- Уровень в иерархии кэшей [level]: число от 0 до 2
- Размер кэша [size]: от 32Кбайт (L1) до 35Мб (L3 в топовых Intel Xeon, их тех, что сейчас можно купить в Москве, для Core-серии характерный размер 6…8Мб)
- Тип [type]: Data, Instruction или Unified
- Какими ядрами используется [shared_cpu_list]: номера ядер

Ключевая проблема

- Запихать невпихуемое

(вся память в несколько Gb не может быть ужата до нескольких Mb)

Причины кэш-промахов

- Первое обращение к определенной области памяти
- Данные были выгружены из-за ограниченного размера кэша
- Данные были выгружены из-за ограниченной ассоциативности

Проблема, когда пытаемся обратиться какому-то участке памяти и этот участок отсутствует в кэше, называется кэш-промах.

Характеристики кэшей

- Размер данных в кэш-линии [coherency_line_size]: 64 для Intel
- Количество ассоциативных каналов в кэше [ways_of_associativity]: 8 для уровней L1-L2, 12-16 для уровней L3
- Количество наборов в кэше [number_of_sets]: 512 для Core i3, 8192 для Xeon E3-1230

Как устроен кэш

- **Блок** — минимально адресуемый объем данных в кэше. 64 байта для Intel
- **Кэш-линия** — блок + метаданные, определяющие адрес в памяти
- **Набор** — связан с некоторым адресом в оперативной памяти. Для L3 — 12 линий по 64 байт = 768 байт
- Весь кэш состоит из независимых наборов. Для Core i5/Gen10 размер кэша L3 6Мб = 64 байта в блоке * 12 линий в наборе * 8192 наборов

Что значит Cache-Friendly

- По возможности использовать непрерывные блоки данных

- Выравнивать данные по границе кэш-линии: **_Alignas** (64) в Си 2011

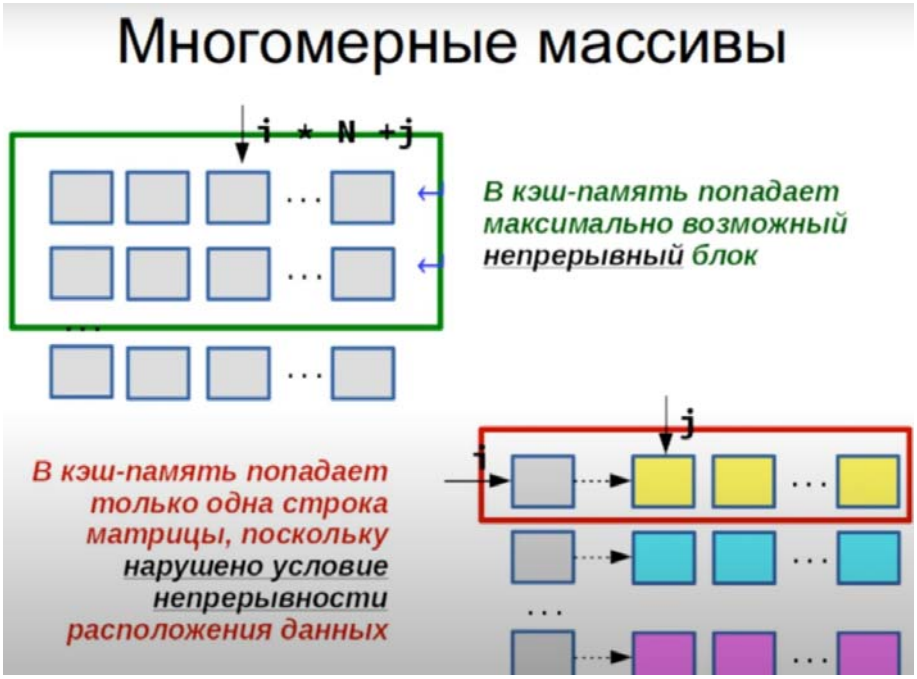
Hint: 64 байта = 512 бит = 4 регистра SSE = 2 регистра AVX = 1 регистр AVX-512

Что с большой вероятностью автоматически попадает в кэш? Например локальные переменные в стеке.

Если мы хотим максимально использовать кэш, чтобы какие-то части целиком попадали в одну кэш линию, то нужно, чтобы все данные начинались с адреса, который делится нацело на размер блока. Это можно сделать выравниванием данных.

Как использовать кэш?

- В наборах инструкций нет команд управления кэшем
- Компиляторы Си/C++ и пр. могут генерировать код для предзагрузки данных
- Увеличить вероятность попадания в кэш можно размещая данные последовательно



Дополнительные ограничения

```
void some_func(const restrict * data) /* C99 */
{
   // компилятор имеет право сгенерирвать код для загрузки
   // содержимого по указателю 'data' в кэш
   . . .
}

/* C++, CLang/GCC */
#define restrict __restrict__

/* C++, MSVC */
#define restrict __restrict
```

Ключевое слово restrict некоторое дополнительное ограничение на входные данные (по аналогии с const), что мы не имеем право это модифицировать. restrict означает, что мы обещаем компилятору, что мы никак не будем (даже косвенным образом) модифицировать данные, и компилятор имеет право сгенерировать код, чтобы предзагрузить данные в кэш.

Наглядный пример: matrix dot

```
typedef std::vector<int> row_t;
typedef std::vector<row_t> matrix_t;

matrix_t dot(const matrix_t &A, const matrix_t &B) {
    const size_t rows = A.size();
    matrix_t R(rows, row_t(rows, 0));
    for (size_t i=0; i<rows; ++i) {
        for (size_t j=0; j<rows; ++j) {
            int sum = 0;
            for (size_t k=0; k<rows; ++k) {
                sum += A[i][k] * B[j][i];
            }
            R[i][j] = sum;
        }
    }
    return R;
}
```

```
typedef std::vector<int> matrix_t;

matrix_t
dot(const size_t N, const matrix_t &A, const matrix_t &BT)
{
    matrix_t R(N*N, 0);
    for (size_t i=0; i<N; ++i) {
        for (size_t j=0; j<N; ++j) {
            int sum = 0;
            for (size_t k=0; k<N; ++k) {
                sum += A[i*N+k] * BT[i*N+k];
            }
            R[i*N+j] = sum;
        }
    }
    return R;
}
```

Есть такой процесс, как исследование на производительность. Это называется профилировка.

Профилирование

- Требуется отладочная информация (-g)
- Терпение… Valgrind работает медленно
- Чем дольше ждем — тем точнее результат
- На выходе получаем XML-файл cachegring.out.PID
- Смотрим файлик в KCacheGrind/QCacheGrind

```
valgrind --tool=cachegrind \
    ПРОГ [АРГ0][ ... АРГn]
```

Инструменты профилирования позволяют исследовать любой процесс для того, чтобы понять какие места у него самые слабые.

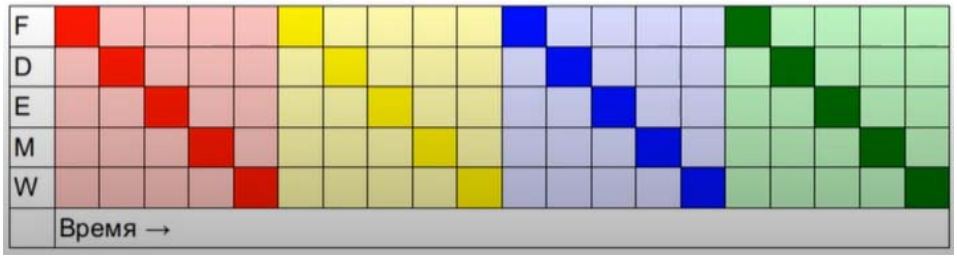
Pipelining & Out-of-order execution

Еще про оптимизации

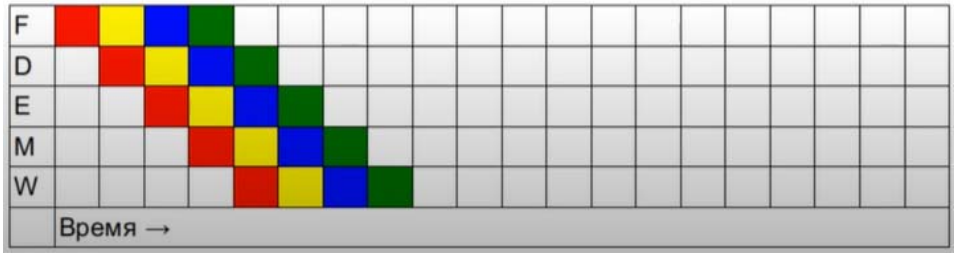
Процессоры не должны прочитать всю команду, ее обработать, затем приступать к выполнению следующей.

Стадии выполнения команд

- Instruction Fetch
- Instruction Decode
- Execute
- Memory Access
- Register Write Back



Каждая стадия задействует только отдельные блоки процессора, которые независимы друг от друга. Последовательно выполнять все инструкции — долго.



Сверхдлинные конвейеры

- Общая идея: упрощаем инструкции, выполняемые на каждом шаге
- Но набор инструкций зафиксирован ISA
    - Для CISC-архитектуры используется микрокод
    - Для RISC это тоже возможно
- Длины конвейера:
    - Современные Intel/AMD/ARM: 8…15
      только в самых современных процессорах Intel топовая частота 4ГГц, наиболее распространены от 1 до 2,5ГГц
    - PowerPC: 20
      Порог 4ГГц (POWER6) пройден в 2010 году, на 22nm — 5ГГц (POWER7)

- Pentium 4: 31
  10 лет назад характерная частота ~3ГГц

Чем выше тактовая частота, тем длиннее конвейер.

Идея спекулятивного выполнения

- Распараллеливаем поток инструкций на несколько функциональных устройств
- В случае условного выполнения — все равно выполняем; если не угадали ветку — результат отбрасываем
- Это эффективно для микроинструкций
- Компилятор может переставлять процессорные инструкции местами

Meltdown

В переводе с английского: взрыв ядерного реактора, фиаско, облом

Это проблема, которой подвержены интеловские процессоры.

- Критическая уязвимость, опубликованная 26 января 2018 г.
- Windows, Linux, Mac, — не имеет значения; проблема в железе, а не софте
- Скомпрометированы все процессоры Intel, старшие процессоры PowerPC, и последние ARM-Cortex

Уязвимость использует два фактора:

1. Спекулятивное выполнение инструкций
2. Нахождение в кэше данных, которые могут быть “чужими”

Спекулятивное выполнение

```
Псевдо-ассемблер x86:

1: mov     rax ← address
2: div     rbx / 0
3: mov     rbx ← [rax]
```

- Инструкция div выполняется долго и завершается ошибкой
- Несмотря на это спекулятивно выполняются все последующие инструкции
- Это приводит к загрузке в кэш данных без проверки доступа
- Результат div отбрасывается, но данные остаются в кэше

Побочный канал связи

- Побочный канал связи — способ косвенно выяснить недоступные данные
- Для проверки нахождения данных в кэше — можно замерить время доступа

Использование Timing-attack

- Работает медленно, но верно

Псевдо-ассемблер x86:

```
1: mov     rax ← address
2: div     rbx / 0
3: mov     rbx ← [rax]
4: rbx  =  (rbx & 0xF) << 6
5: mov     rbx ← [arr+rbx]

...
N-1: rdtsc
N  : mov   rbx ← [arr+rbx]
N+1: rdtsc
...
```

Как бороться?

- Минимизация доступа к памяти ядра или гипервизора
- Рандомизация размещения данных ядра в памяти
- Снижение точности perf-таймеров
- Запрет на кэширование при спекулятивном выполнении

Защита от Meltdown — снижение производительности до 30%

Векторные инструкции

- 128-битные регистры XMM0…XMM15
- Вертикальные инструкции:
  - одновременные действия над несколькими частями (арифметические операции)
- Горизонтальные инструкции:
  - по частям вектора получить скаляр (найти мин/макс, скалярное произведение)

Векторные инструкции

- MMX — регистры XMM, целочисленная арифметика
- SSE — регистры XMM, вещественная арифметика
- AVX — регистры YMM (256 бит)
- AVX-512 — регистры ZMM (512 бит)

BLAS (Basic Linear Algebra Subprograms)

- Программный интерфейс для типовых операций из области линейной алгебры
- Существуют различные реализации:

    - Netlib BLAS (эталонная реализация)
    - Intel Math Kernel Library (x86-64)
    - OpenBLAS (для x86, x86-64 и ARM)
    - ATLAS (x86, определение набора в runtime)
    - NVBLAS (для nVidia GPU)
    - clBLAS (для OpenCL, поддерживается AMD)

Про выравнивание

- Задача: минимизировать количество тактов на чтение вектора
- Решение: размещать вектор в памяти таким образом, чтобы адрес его начала был кратен размеру вектора
- Cons: это не всегда возможно
- Используются отдельные инструкции для выровненных и не выровненных данных
- Пример: movaps vs movups
- Выравнивание можно гарантировать:
    - Явным размещением по определенному адресу в куче
    - **_Alignas(Кол-во-байт)** в остальных случаях (стандарт C11)
- Соглашения о вызовах функций (x86):
    - стандартное cdecl — аргументы на стеке
    - WinAPI stdcall — аргументы на стеке, но очищает вызывающая функция
    - для системных вызовов fastcall — аргументы через регистры
- Соглашения о вызовах функций (x86-64):
    - аргументы передаются через регистры
    - вещественные значения через — xmm
    - требуется выравнивание стека по границе 16 байт (128 бит)

Лекция 7 — Interrupts. The Kernel. System Calls

AVR Architecture Overview

- 8 bit (Data), 16-bit (Instructions)
- 2.7 - 5.5V power
- Integrated ports
- Built-in SRAM + EEPROM
- Variants:
    - ATMega — full profile
    - ATTiny — small size
- Programming using:
    - Arduino IDE (requires firmware)
    - gcc-avr (C, C++)
    - avra (assembly language)

Hardware Events

1. Button press/release
2. Incoming data frame received
3. Internal or external timer signal
4. Invaders arrived

Interrupts handling

- Fixed-address interrupt vector
- Each event has mapping to specific code location
- JUMP to implementation or do nothing

Atmel ATTiny13 program layout

	function main code	
	function hanlder code	
0x0009	ADC	RET
0x0008	WDT	RET
0x0007	TIM0_COMPB	RET
0x0006	TIM0_COMPA	RET
0x0005	ANA_COMP	RET
0x0004	EE_RDY	RET
0x0003	TIM0_OVF	RET
0x0002	PCINT0	RET
0x0001	INT0	JUMP handler
0x0000	RESET	JUMP main

Когда включаем устройство, то начинается выполнение с нулевого адреса. Что такое нулевой адрес? Т.е. процессор начинает перебирать все инструкции с нуля, и мы можем записать самую первую инструкцию — это прыжок на какую-то нашу главную функцию — условный main, которая содержит основную программу, либо просто всю основную программу разместить с нулевого адреса.

Если у нас есть какие-то периферийные устройства, то для них существует фиксированный адрес, называемый вектор прерываний, т.е. в памяти для инструкций должны быть указаны явные инструкции, которые вызываются при наступлении определенного события.

Что произойдет при нажатии кнопки, которая подключена к вводу ??? ? Процессор приостановит выполнение программы, которую он до этого выполнял и в program counter будет записано значение соответствующее указанному событию.

Hardware Interrupt Handler

- Function with no arguments and return value that might be called at **random time**
- Handler is responsible to save and restore current program state (a set of callee-saved registers)

Modern Interrupts Subsystem

- I/O APIC (Advanced Programmable Interrupt Controller)

- Use within PCI/PCIExpress bus
- Controller responsible to catch interrupts and store them in queue
- Priority has no manner

Classic x86 Interrupt Handling

- Each device has its own IRQ number
- IRQ numbers ordered by priority
- CPU receives `INT` signal and stops execution of current context
- Each interrupt has its own address at interrupt vector
- Interrupt vector might have more values than hardware implementation

`int` number instruction

Что должен сделать процессор, когда получает прерывание? Он должен приостановить выполнение текущей программы, дальше опросить ??? прерывания, получить некоторое число и выбрать с этим числом какой-то код из таблицы прерываний.

Interrupt Mask

- Interrupts might be prevented globally
- Use-cases:
  - prevent interrupt handling while another handling in process
  - critical real-time code execution
- Some architectures support NMI (Non Maskable Interrupts): Zilog Z80

Classic x86 Interrupt Handling

- 0 — Initial state (Reset)
- 1…15 — Hardware interrupts
- 16+ — Software interrupts initiated by ‘int’

Initial Interrupt Vector

- BIOS subroutines
- Handles hardware events
- Useful I/O subroutines

У нас есть большой вектор прерываний, где есть некоторые отсылки на инструкции, которые мы выполняем.

Кто прописывает нам некоторые функции, которые обрабатывают прерывание? Есть такая микросхема под названием BIOS (basic input output system), которая содержит весь минимальный необходимый для работоспособности компьютера после включения набор функций.

AH — command, AL — Argument

BIOS

- `INT 0x10` — screen text subsystem
- `INT 0x13` — disk subsystem
- `INT 0x15` — UART subsystem

- `INT 0x16`, `INT 0x17` — keyboard subsystem

DOS (Disk Operating System from Microsoft)

- `INT 0x21` — DOS API functions

Interrupt Processing (x86)

- Save EIP (RIP) into the stack
- Set flag IF into 1 value
- Jump to instruction at address IDTR + offset
- IDTR register:
    - low 16 bits — table size
    - high bits — table physical address

Before Handler Call:

- Reset Out-of-order execution
- Switch current virtual address table
- Replace stack pointer value

Processed by Interrupt Handler:

- Save current state: register values + flags
- … // do something
- Restore state

Что должен сделать процессор, когда происходит прерывание. Во-первых, он должен сбросить весь свой конвейер (инструкции выполняются стадиями). Сбрасывается все подряд, переключается таблица виртуального адресного пространства, заменяется стек, после чего происходит выполнение той функции, которая выполняет обработку прерывания. В свою очередь сама функция ответственна за то, чтобы после своего выполнения оставить минимальное количество следов, что ее не просили, т.е. сделать сохранение на стеке значения регистров, флагов, ну и перед выходом все вернуть обратно.

The Kernel

- The first program to be launched from OS
- The only program that has full control to hardware

Единственная программа, которая как-то может регистрировать прерывания — это ядро.

x86 Protected Mode

Regular Mode

- Each process might access only its own memory
- Each process Virtual Address Space starts from 0 address
- Processes can’t interact hardware

Privileged Mode

- Full access to physical (non-virtual) memory

- Access to I/O ports
- Access to some commands not available from regular mode

Механизм для взаимодействия обычных процессов с ядром называется системный вызов. Т.е. системный вызов — это некоторая штуковина, которая позволяет нам обратиться к функциональности ядра для того, чтобы выполнить те операции, которые в обычном режиме процессору сделать не???.

The Kernel

- Just a regular ELF (*nix) or MS PE program file
- Starts before CPU leveraged its privileges
- Responsible to interact all hardware

Что такое ядро? Ядро — это обычная программа, в случае с UNIX системами — это самый обычный ELF файл, и программа запускается самой первой, после чего ядро обязана понизить приоритеты режим работы процессора до не приоритета. Т.е. что происходит, когда запускаем какое-то ядро? Ядро запускается, инициализирует вектор прерываний, инициализирует оборудование и как только ядро готова к работе, оно понижает привилегии процессора и само ничего больше не может делать.

Kernel Startup

- Re-initialize devices (PCI-USB)
  BIOS subroutines became not available after switching from real to protected mode
- Initialize new interrupt vector
- Find, load and start all drivers matched by PCI ID or USB ID
- Leverage processor privilege level
- Launch **The First User Process**: init, system, launchd etc.

Kernel Interaction (x86)

- All processes are unprivileged
- The only way to access hardware is to switch to Kernel Mode
- All interrupts switch CPU into privileged mode
- Use **`int`** command to access Kernel subroutine (System Call)

Для того, чтобы перейти в выполнение программы, которая выполняет привилегированный код, необходимо прерывание. Как только возникает прерывание процессор переключается в привилегированный режим работы и выполняет какой-то код, соответствующее прерывание которого мы получили (который мы храним в таблице прерываний).

Kernel Types

- Monolithic Kernels
  One big program to be executed at privileged level
  Examples: all old OSes, BSD and Linux
- Microkernels
  Only small event router to be executed at privileged level, most subsystems work like regular user processes
  Examples: MINIX3, QNX, Mach
- Hybrid

Modular but not one big program to be executed at privileged level
Examples: Windows, Mac, BSD and Linux

Kernel Modules

- Kernel is a program to be executed for specific hardware
- Kernel implement routines not related to hardware
- Might load ***device drivers*** or ***kernel modules***

Что такое модуль ядра? Если говорить терминами Windows, у нас есть какие-то драйверы устройств, но это собственно основное значение для чего нам нужны какие-то модули, хотя если вы точно знаете какую-то конфигурацию оборудования ровно под которой должно работать ваше ядро, то никто не мешает вам запустить конфигураторы ядра, собрать его под конкретную железяку, лишнее выкинуть, сделать один монолит и запустить его в качестве единственного ядра и все.

(Драйвер — это некоторая часть ядра, оформленная в виде отдельной библиотеки, которая знает, как взаимодействовать с каким-то конкретным оборудованием. Выполняется также в пространстве ядра, т.е. имеет очень высокие привилегии и прямой доступ к оборудованию.)

- lsmod — show loaded modules
- modprobe — check and try to load module
- insmod — force module loading
- rmmod — unload module

/etc/modprobe.d — modules configuration

How To Load Modules While Boot

- Each module is an ELF file
- Files to be stored in File system

Key problems:

- Disk system must be available to access FS
- File System must be available to load modules

How To Load Modules While Boot

- Compile all required component into the kernel
- Provide minimal filesystem image to be loaded by bootloader (initrd)

System Calls

- `int 0x80` (Linux, BSD, but not MacOS)
- `sysenter/sysleave` instruction (Intel x86, nut not AMD processors)
- `syscall/sysret` instructions (AMD-64/x86_64)

У нас есть ядро, в ядре есть какая-то функциональность, эта функциональность расширяется с помощью модулей ядра, и для того, чтобы выполнить какую-то функцию с ядра есть два варианта: можно выполнить прерывание, вызывается `int` 0x80, выполняется код из таблицы прерываний, который дальше обрабатывает те

аргументы, которые мы передали через регистры, и ядро понимает, что от него требуется вызвать некоторую функцию, эта функция выполняется.

В какой-то момент в интеловских процессорах появились отдельные инструкции под названием sysenter для того, чтобы не использовать достаточно тяжелый механизм работы с прерываниями. Компания AMD разработал инструкцию syscall, которая делает то же самое, только в 64-битной архитектуре.

`INT 0x80`

- Unified interrupt to access all OS features in Linux/BSD
- EAX stores system call number `/usr/include/asm/unistd_32.h`
- Parameters stored at EBX, ECX, EDX, ESI, EDI, EBP
- Return value at EAX
- Calling conventions differ from C language!

Что происходит при выполнении прерывания инструкции `int 0x80`. В регистре EAX перед вызовом прерывания должен быть записан номер этого прерывания. У системного вызова могут быть параметры; они записываются в оставшихся не более чем 6 регистрах. Вызываем `int 0x80` и тем самым переключаемся на выполнение ядра. Как только ядро закончил выполнять системный вызов, то возвращаемый результат хранится в EAX.

System Calls (Linux/BSD)

- Section 2 for manual pages
- syscall function to call arbitrary system call

syscall занимается тем, что перекладывает аргументы из стека в требуемые регистры, после чего в регистре EAX записывает требуемое нам число — номер системного вызова, и выполняет инструкцию `int 0x80`.

System Calls (MacOS)

- Highly customized XNU (Darwin project) Kernel:
  - Mach microkernel
  - Some system calls from FreeBSD implementation compiled into Kernel
  - Some system calls from FreeBSD implementation compiled into userland-processes
- 32-bit systems: uses 0x80, 0x81, 0x82 interrupts for various APIs
- 64-bit systems: uses additional (number << 24) in syscall number to choose API
- syscall function is deprecated for recent MacOS

Example: Hello World (32-bit)

```
static const char S[] = "Hello";
write(1, S, sizeof(S));
/* man 2 write
    #include <unistd.h>
    ssize_t write(int fd, const void *buf, size_t count);
*/
```

```
#include <asm/unistd_32.h>
    .intel_syntax noprefix
    mov eax, __NR_write  // system call number
    mov ebx, 1           // first argument
    mov ecx, S_ptr       // second argument
    mov edx, 6           // third argument
    int 0x80             // just do it!

S:     .string "Hello"
S_ptr: .long S
```

Функция _start — функция, которая содержится в стандартной Си библиотеке и в свою очередь делает инициализацию Си библиотеки, после чего вызывает пользовательскую функцию main.

64-bit syscall instruction

- Works up to 50% faster than interrupt
- RAX — syscall number
- Arguments passed via RDI, RSI, RDX, R10, R8, R9
- Spoils contents of RCX and R11
- All arguments are 64-bit but not 32-bit values (in comparison to `INT` instruction)

В чем отличие от инструкции INT, которая тем не менее все равно работает на 64-битной архитектуре. На 64-битной архитектуре Intel можно выполнять программы, которые скомпилированы для 32-битной и именно для поддержки совместимости с 32-битным кодом предназначены прерывания.

linux-vdso.so (linux-gate.so)

- Virtual “library” to be mapped into process virtual address space
- Implements some Kernel functions: `__vdso_clock_gettime`, `__vdso_get_cpu`, `__vdso_gettimeofday`, `__vdso_time`

man 7 vdso

Еще один способ вызова отдельных функций ОС — это использование виртуального пространства ядра. У каждой программы есть зависимость от динамического загрузчика ELF файлов, который должен найти все библиотеки от которых зависит, загрузить в память, после этого передать управление функцию _start. Есть у многих программ зависимость от стандартной Си библиотеки — тоже какой-то файл. И есть еще одна зависимость, которая не связана вообще ни с каким файлом — linux-vdso.so. Это виртуальная библиотека, которая реально не существует — это просто часть адресного пространства процесса, на которой отображаются реализации нескольких функций с ядра.

Лекция 8 — Exec Files. Boot Loaders

Compile stages

1. Source to binary translation
2. Link into **executable file**

Из чего состоит трансляция исходных текстов в исполняемый файл? Стадии компиляции заключаются, во-первых, в том, что нужно распарсить исходный текст программы, разобраться в ее структуре и после построения дерева разбора пройтись по этому дереву, сгенерировать некоторый бинарный код, которому однозначно соответствует некоторый ассемблерный код. Дальше, чтобы запустить этот код нужно его где-то разместить.

x86 Executable File Formats

- Flat-form binary
  - no headers, just code from zero offset
  - examples: .com and .sys in DOS/CP-M
  - flat loaders
- Executable and Linkable Format (ELF)
  - standard for the most UNIXes
- Mark Zbykowsky Portable Executable (MZ PE)
  - standard for Windows
  - UEFI boot loader
- March-O
  - multiple architectures support
  - Darwin and Apple OSes

ELF

- Magic-bytes: {0x7F, ‘E’, ‘L’, ‘F’}
- Binary header before actual contents:
  - processor architecture
  - processor and file itself bits capacity
  - entry point
  - segments positions

Если вы каким-нибудь текстовым редактором откроете исполняемый файл типа ELF, то в самом начале вы увидите какую-то кракозябру с кодом 7f в шестнадцатеричной записи данных, дальше буковки ELF. Это так называемые magic bytes — некоторая последовательность байт фиксированного размера, которые присутствуют в любом бинарном файле, которые используются для того, чтобы определить тип данного файла. Дальше следует некоторый бинарный заголовок, в котором определяется для какой процессорной архитектуры предназначается данный файл. Кроме того, определяется разрядность процессора, ну и разрядность определяет размер машинного слова, которая в дальнейшем будет в том числе использована для парсинга самого ELF файла. Есть разные сегменты — data, text и т.д., у которых есть определенное положение внутри самого файла ELF. Ну и самое главное, которое должно быть в ELF файле — это некоторая точка входа, откуда нужно выполнять программу.

ELF Interpreter

- Special program (/lib[64]/ld-linux.so) to load a program into the memory
  - File all required libraries
  - Load found libraries and file itself into memory
  - Allocate memory into stack

- Jump to Entry Point

Интерпретатор ELF — программа, которая должна быть вызвана для того, чтобы выполнить какой-то произвольный ELF файл.

How to Find a Library?

- Check config file /etc/ld.so.conf
- Check for LD_LIBRARY_PATH environment variable
- Check for DT_RPATH and DT_RUNPATH sections from ELF file
- gcc -Wl, -rpath, DIRECTORY
  Substitutions available:
  - $ORIGIN — ELF file directory
  - $PWD — current directory
  - $LIB — string ‘lib’ or ‘lib64’
  - $PLATFORM — platform name, like ‘arch’ output

Dynamic Library vs Executable

- The same ELF format
- Executable must have an entry point
- Library must have a symbol table

ELF файл является библиотекой, если он экспортирует таблицу символов.

На что влияет позиционная независимость с точки зрения исполняемого файла? Если мы код компилируем как позиционно независимый, то он может быть загружен в произвольный область памяти.

DOS and Windows Executables

- Mark Zbykowsky Portable Executable (MZ PE)
- “Portable” means several headers for DOS, Win32 and .NET
- Does not use position-independent code:
  - each library has its own address to load
  - might cause to dependency hell
  - might not work in case of limited virtual address space

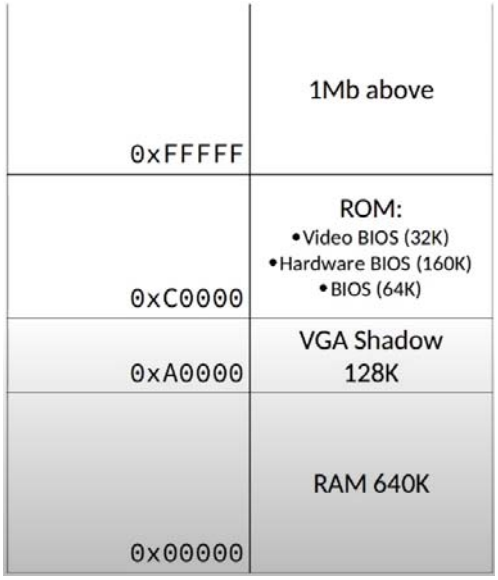
Flat-Form Binary

- No headers
- Just ‘call’ file begin address after load
- Last instruction must be ‘ret’
- On x86 usually used only in 16-bit real mode

x86 Real Mode [historical reference]

- Initial Intel x86 processor state
- Address space is limited to 20-bits pointer size (1MiB), RAM size is 640KiB
- All memory is split into 16-bits capacity accessible **segments** (64 KiB)
- Segment start positions
  - CS — code segment
  - SS — stack segment
  - DS — current data segment

1MiB of address space [historical reference]



- Complete address space at Intel x86 Real Mode
- “640KiB is enough to all!” (c) Bill Gates
- VGA Shadow — text only access to video card
- ROM — firmware
- i80286 processor has an extension to 24-bits capacity in Real Mode

Software Interrupts

- **`int NN`** command
- Like hardware interrupt from the handler’s point of view
- Before OS: use BIOS routines
- OS can (but not responsible to) change interrupts vector

THE KERNEL IT’S LOADING

x86 Operating System Loading Stages

The Kernel

- The first program to be started from OS
- Have full access to anything

Сначала ядро нужно загрузить в память и передать ему управление (до загрузки ядра мы ничего поделать не можем). Этим занимается программа, который называется загрузчик.

Boot Loader

- Find a program into the disk
- Load it and place into the memory
- Launch it!

Что делает загрузчик? Ядро — некоторый файл, который нужно сначала найти где-то, загрузить его в память на нужное место и запустить.

Classic PC-style Boot Loader

- Disk boot order specified by BIOS
- First 512 bytes contains Master Boot Record (MBR):
    - disk is bootable in case if last MBR bytes values are {0x55, 0xAA}
    - 64 bytes before bootable magic bytes are Primary Disk Partition Table
    - first 446 (512-2-64) bytes might store executable code for Boot Loader

Boot Loader Capabilities

- Runs at Real Mode
- Might access Video Shadow Memory
- Can use BIOS routines to access:
    - disks
    - keyboard
    - COM-ports

Boot Loaders Overview

- Simple: ntldr (Windows), loadlin (Linux)
- Universal: GRUB (GRand Universal Bootloader)
    - able to load about any Operating System
    - able to choose OS and ask for boot options
    - graphical background design
    - **446 bytes are enough?**

GRUB staging

- Stage 1: MBR code finds and loads core.img
- Stage 2: core.img and its plugins is responsible to load The Kernel

ELF Image Loading (x86)

- Switch to 24-bit address space mode (Gate A20 feature of i80286)
- Load Kernel file from disk and place image **strictly after** 1MiB of address space
- Find GRUB-specific Magic Bytes in The Kernel image: 0x1BADB002
- GRUB is not able to parse arbitrary ELF file, but entry point is located near Magic Bytes
- Switch off hardware interrupts
- Switch into The Protected Mode
- Launch!

GUID Partition Table

- GUID — Global Unique IDentifier
  128-bits unique key for disk partition
- Has no matter SATA Port Number the disk attached to

GUID разбиение — это когда каждому диску присваивается какой-то уникальный 128-битный ключ. Для чего это делается? Для того, чтобы независимо от того как вы разместили ваши разделы на диске, независимо от того как вы подключаете этот диск вы могли подмонтировать произвольный раздел либо снова загрузиться независимо от того в какую SATA port ???.

GUID Partition Table

- MBR — maximum partition size is ~2Tb
- GPT — has no limit in practice (~100TB)

- MBR — 4 primary partitions, might have Extended partition type to create secondary-level partitions
- GPT — 264 primary partitions
- Has special partition of type EfiBoot (FAT32 by implementation) that stores UEFI boot data

Unified Extensible Firmware Interface

- API to create small programs that stored into Firmware memory:
  - hardware test tools
  - system configuration GUI
  - bool loader
- UEFI programs might be stored both in Firmware and EfiBoot partions by specified URI:
  EFI\BOOT\BOOTX64.efi
- EfiBoot in Linux usually mounts to /boot/efi

- Kernel executable format (sic!) is MZ PE
- Programs do not use interrupts, but can access Firmware UEFI API
- Runs at Protected but not Real Mode

Лекция 9 — Operating System Components. File Systems

Operating Systems Purpose

- Computational Systems might have various architectures
- Processor Architectures might have various memory management models
- Interactions using I/O ports, DMA and interrupts handling

Operating System is a set of components to implement abstraction levels from hardware to high-level APIs.

Core Components

- The Kernel
- Minimalistic set of Standard Libraries
- Minimalistic set of Standard Tools

- High-Level Libraries
- High-Level Framework Components

- Desktop Environment
- Server Environment

API Level Abstractions

Standard Libraries

- C Language: ISO/IEC 9899:2018 (C17)
- C++ Language: ISO/IEC 14882:2017 (C++17)

Operating System Interaction

- Portable Operating System Interface based on UNIX (POSIX): IEEE 1003.1-2017

Что покрывается стандартом POSIX? Системные вызовы + какие-то функции, которые необязательно являются системными вызовами (в некоторых системах обычные функции, в других — системные вызовы).

- Applications
- High-Level Libraries and Frameworks
- libstdc++ (Linux, *BSD), msvcp.dll (Window) or libc++ (MacOS)

Implements POSIX standard API:

- glibc, bionic, uclib, BSD libc, libSystem.dylib
- Kernel system calls

Как реализуется функциональность стандарта POSIX? В первую очередь это какие-то функции языка Си, которые являются оболочками поверх системных вызовов.

Libraries Interaction

- Libraries to be loaded within a program by ld.so
- Libraries contents live within the same address space
- Functions usually must be called indirectly via Procedure Linkage Table
- Function arguments might be any values within process address space

Как библиотеки используются? Они загружаются одновременно с самой программой, размещаются в памяти и живут целиком в адресном пространстве процесса.

Принципиальное отличие между использованием функций библиотек и каких-то локальных функций заключается в том, что функции библиотек лежат далеко от функций программы и нужно сделать длинные прыжки, перезаписать program counter.

Linux Kernel Interaction

- `INT 0x80` instruction (x86-32)
  - EAX stores syscall number
  - EBX, ECX, EDX, ESI, EDI, EBP — arguments, EAX — retval
- SYSCALL instruction (x86-64)
  - RAX stores syscall number
  - RDI, RSI, RDX, R10, R8, R9 — arguments, RAX — retval
- SWI 0x00 instruction (ARM/EABI)
  - R7 stores syscall number
  - R0, R1, R2, R3, R4, R5, R6 — arguments, R0 — retval
- Arguments might be just integers:
  - integer values
  - pointers

Linux Kernel Interaction

- Virtual Dynamic Shared Object — pseudo-library with no real library file
- Kernel Space to User Space memory mapping
- x86_64 functions:
    - __vdso_clock_gettime
    - __vdso_getcpu
    - __vdso_gettimeofday
    - __vdso_time

Не все системные вызовы реализуются через переключение контекста в адресное пространство ядра. Есть некоторые системные вызовы, которые реализуются как обычные функции, которые на самом деле находятся в адресном пространстве ядра, но временно можно переключиться на это адресное пространство. Они отображаются как некоторая виртуальная библиотека, с которой не связан никакой библиотечный файл. Это делается для того, чтобы очень быстро вызывать некоторые функции, которые не требуют каких-то специальных высоких привилегий и могут быть выполнены в адресном пространстве пользователя.

Kernel Subsystems

- Device Drivers
- Memory Management Services
- Process and thread scheduler
- Inter-Process communications
- **Virtual File System**

UNIX Virtual File System

- Common file hierarchy for all mounted devices and network locations
- Common API to access files
- Tree-based file lookup by naming conventions

UNIX File Types

- Regular File
- Directory
- Devices:
    - Block Devices
    - Character Devices
- Symbolic (but not hard!!!) Links
- FIFO (named pipes)
- Sockets

Regular Files

- Just stores data
- Random-access to data:
    - allows lseek
    - allows fflush(stdin)
- Data structure for contents has no meaning for The Kernel
- Hi-Level APIs might be sensitive for file format. Example: fopen ‘b’ flag for binary data

Обычный файл — это просто имя, связанное с определенным контентом. Принципиальная особенность обычного файла — можно произвольным образом обращаться к любой его части.

Directories

- Directory — just a file of specified format to store directory entries
- File contents are struct dirent entries
- struct dirent is implementation defined
- POSIX defines at least struct dirent contents:
    - inode
    - record size (by file name length)
    - file name up to NAME_MAX (256) bytes

Links

- Symbolic Links — special files to store target file names
- High-Level APIs and most system calls threat links link targets
  Exception: lstat system call
- Hard Links — just additional names for files

Символическая ссылка — это файл (еще одного типа), который содержит некоторый текст, указывающий на имя файла, на который эта ссылка ссылается. Жесткая ссылка — это просто еще одно имя файла.

Devices

- File-like API to access hardware
- Character devices allow send/receive data
- Block devices allow random access like regular files

Устройства — это, по сути, то, что валяется в /dev. Устройства бывают двух типов. Блочное устройство — это некоторая абстракция, которая позволяет обращаться с некоторым устройством используя обычные Си-шные и плюсовые функции для работы с файлами (регулярными). Другой тип файлов — это символьные устройства.

В чем принципиальное отличие терминала от диска? Терминал в отличии от диска является символьным последовательным специальным файлом, а не блочным, и от блочных они отличаются тем, что в случае с символьными устройствами вы можете только читать и писать, но при этом вы не можете произвольным образом обратиться к произвольному участку данного файла.

FIFO and sockets

- Inter-Process Communication
- Just file names to access but no real storage
- FIFO: First-in-First-out pipes
- Sockets: more complicated to allow many-to-one process communications

X сервер — программа, которая имеет прямой доступ к оборудованию — к видеокарте, клавиатуре, мышке и к ней подключаются отдельные клиенты — это вот графические приложения — терминал, рабочий стол, шелл.

File Systems

- Logical: VFS root/
- Physical: each filesystem mounts to specific subtree

- Physical file system structure is self-sufficient
- It is not possible to create hard links to another mount point

Physical File System Types

- Disk-oriented: FAT, NTFS, EXT2/3/4, XFS
- With no structure: SWAP
- Network: SMBFS (Windows), NFS (UNIX), SSHFS (Universal)
- Virtual: TMPFS, OVERLAYFS

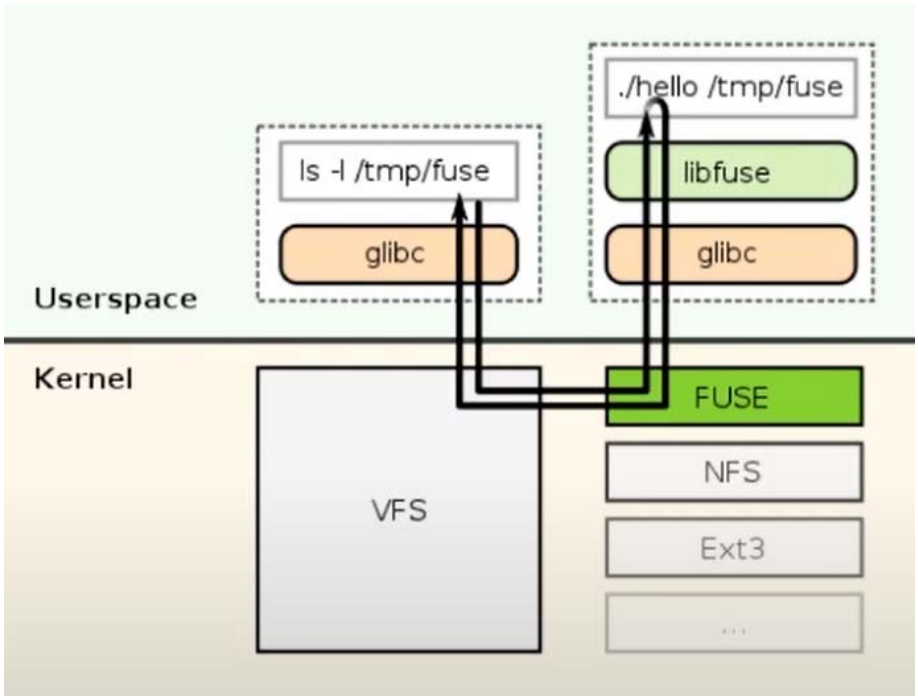
VFS Files Mapping

- Session number st_dev for each physical FS:
- major (24 bit) — device type
- minur (8 bits) — instance number
- /dev/loop might map any file to FS “device”
- inode value is a key to ident a file within physical file system
- Foreign FS’s inodes emulated by FS driver

VFS and File Descriptors

- (st_dev, st_ino) references distinct files
- Process can’t access file until it opens
- File Descriptor — integer number within distinct process
- Each process has its own file descriptors table
- Each process has maximum file descriptor value (usually 1024)

Filesystem in Userspace (FUSE)



- Kernel-space connector to VFS
- User-space implementation for arbitrary file system
- Might be implemented using various programming languages

Есть подсистема — называется FUSE — который реализует следующим образом: предоставляется некоторый интерфейс, доступный для взаимодействия с пространством ядра (в случае с Linux это специальное псевдоустройство defuse) и реализуется некоторый пользовательский процесс, который запускается, открывает устройство defuse и из него входящие события как-то обрабатывает.

Data Integrity

- The Linux Kernel uses as most memory available to cache I/O
- Actual data flushes on umount or sync command
- There might be additional device buffers
- File might be forced to flush for data integrity:
  `int fsync(int fd)`

Physical Storages

- HDD — magnetic high density disk(s)
- Specifications:
  - connection interface
  - capacity (1K = 1000 bytes, but not 1024)
  - access time
  - rotation speed (5400, 7200, 10K, 15K RPM)
  - data transfer rate (from 70 to 200Mb/sec)
  - buffer size (from 8 to 128Mb)

Physical Storages

- SSD — flash memory
- SLD — 1 bit/cell
  MLC — 2 bit/cell
  TLC — 3 bit/cell
- Specifications:
  - connection interface (SATA or M2)
  - capacity
  - access time / data transfer rate
  - **read/write cycles lifetime**

SSD Lifetime

- SLC — 100 000 write cycles
- MLC — from 3 000 to 40 000 write cycles
- TLC — less than 2 000 write cycles

- Cells usage balanced by controller
- Cells to be dead in some future mark as “used up”
- TRIM operation decreases SSD capacity

Connection Interfaces (historical reference)

- Parallel-ATA (also known as IDE):
  - 40 pins/line, Master and Slave
  - typical motherboard had 2 IDE ports
- SCSI:
  - 40, 68 or 80 pins/line
  - devices connected in sequential order, bus ends with “terminator” device

ATA Interface (Advanced Technology Attachment)

- Logic level for IDE (PATA) and serial (SATA)
- ATA PIO Mode: CPU reads and writes data via I/O ports
- UDMA Mode: controller has access to memory
- Data sent as commands
- ATA Packed Interface (ATAPI) — some commands for SCSI emulation

SCSI Interface

- Universal interface to attach not only disks
- Data sent as commands
- Modern implementation:
  Serial Attached SCSI (SAS interface)

Advanced Host Controller Interface (SATA AHCI)

- Native logical interface for SATA to drop IDE/ATA Lecacy compatibility
- Allows Power Management and Hot-Swap
- BIOS-based devices allow to switch SATA interface mode

- Be careful: some systems (like Windows) might operate just the same mode as installed

RAID (Redundant Array of Inexpensive Disks)

- Data replication for increased reliability
- Data interleaving for increased speed rate

- Requires several disks
- Might be implemented by hardware or software support

- RAID-0
  - at least 2 disk devices
  - data interleaving mode
  - read/write data bandwidth increased linearly by disk count
  - !!! One disk death causes data loss !!!
- RAID-1
  - at least 2 disk devices
  - data replication mode
  - read data bandwidth increased linearly by disk count
  - write data bandwidth is the worst case disk bandwidth
- Historical RAID Levels
  - RAID-2: interleaving + Hamming code

    - RAID-3: byte interleaving + parity disk
    - RAID-4: block interleaving + parity disk
- RAID-5 and RAID-6: block interleaving + parity blocks at each drive

RAID Implementation

- Hardware
    - Distinct hardware controller
    - Typical for server-oriented motherboards
    - Requires OS device driver
    - Be careful! Controller death -> data might be lost

    - Controls by dmraid in Linux
    - /dev/dmX for Linux
- Software
    - Implemented at Kernel Level
    - Additional CPU overhead

    - Controls by mdadm in Linux
    - /dev/mdX for Linux

RAID-Like Implementation

- LVM (Logical Volume Manager)
    - Linux-specific
    - Might use various FS types
    - Looks like Partitions
- File System Level
    - ZFS (Solaris, FreeBSD, MacOS)
    - BtrFS (Linux)

File System Concepts

- User-level:
  file access by its name (string)
- Process-level:
  opened file descriptor (int)
- Kernel-level:
  VFS entry (tuple of st_dev and inode)
- FS-level:
  blocks of data associated to inode

File System Concepts (implemented at ext2)

- Blocks of fixed size
- Block is a minimal accessible FS entry
- Group of blocks — continuous set of blocks
- Blocks might have special purpose

Blocks Fragmentation

- Files might change their size

- Files can’t be placed continuously

- Might increase access time for mechanical HDD
- Not actual problem for multi-task OSes
- Old Windows systems had “defrag.exe” tool

File System Concepts

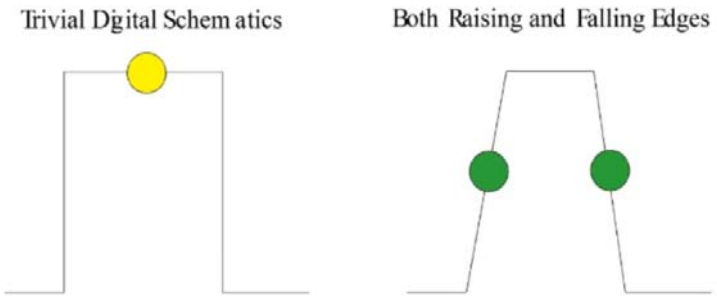
- Special blocks
  - bit mask of used/free blocks
  - inode map tables
- SuperBlock
  - special purpose block at fixed size position
  - stores FS metainformation

Linux File Systems for Disks

- ext2/3/4 — the most robust
- XFS (from SGI) — suitable for storing large files
- BtrFS — mostly experimental, but not so reliable
- ReiserFS — once upon a time was the most effective for large amount of small files

Лекция 10 — Memory Allocation

Physical Memory



- Logical: Address Bus + Data Bus
- Modern implementations: bus to send commands using Double Data Rate

Physical Memory

- Specifications:
  - individual chip clock
  - bus clock
  - MT/s (Mega Transfers Per Second) — twice as clock
  - bandwidth: several bits at a time
- Example: Kingston HyperX PC4-24000
  - bus clock: 1500MHz
  - 3000 MT/s
  - 24000 Mb/s

Physical Memory

- Command-oriented data transfer via bus

- Latency — time between command itself and stream of data
- Cache on CPU helps make fast random-access

Physical Address Space

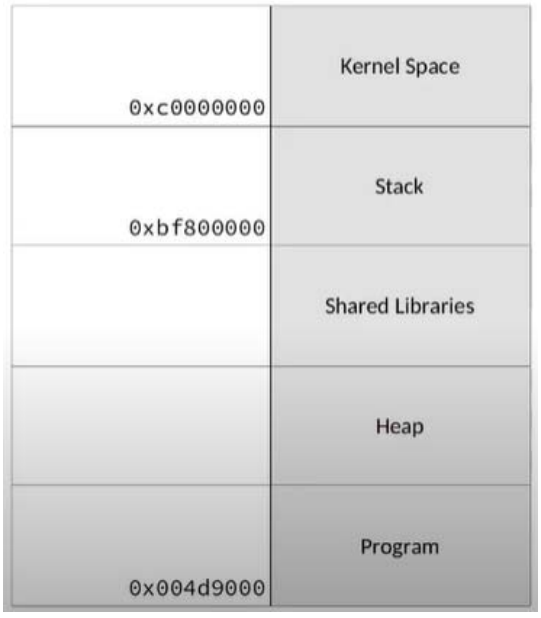
- Address range of the size 2^(Bits):
- Where bits are:
    - 32-bits for ARM, legacy x86
    - 36 bits for x86-32 (Pentium+) — PAE Mode
    - 64 bits for x86-64 and all 64-bit platforms

Process Address Space

- Any value that matches pointer size
- Each process has its own memory
- Each process pointer addresses starts from 0
- Limitations:
    - 32-bit systems: 4Gb
    - 64-bit x86-64: 256Tb (only 48 bits available)

Process Address Space Example (x86-32)

- Full Address Space
  0x00000000
  0xffffffff
- User-Accessible is 3Gb (Linux) or 2Gb (Windows)
- Upper addresses accessible only by The Kernel



/proc/process_number/maps

- Text file at virtual File System **procfs**
- Subdirectory **self** is a link to process itself

cat /proc/self/maps

```
004d9000-004e1000 r-xp 00000000 08:02 655669   /usr/bin/cat
004e1000-004e2000 r--p 00007000 08:02 655669   /usr/bin/cat
004e2000-004e3000 rw-p 00008000 08:02 655669   /usr/bin/cat
|- range addresses |   |        |- st_dev |    |- mapped file name
                   |   |                  |
    access flags - |   |           inode -|
                   |
          file offset -|
```

Memory Management Schemes

- x86 Real Mode Segmentation
- 2-Level Memory Paging (x86)
- 3-Level Memory Paging (x86+PAE)
- 4-Level Memory Paging (x86_64)

Real Mode Segmentation

- 16-bits Registers Value (up to 64K)
- Address space is much greater (1Mb for traditional IBM PC)
- Segment types:
    - code segment (CS)
    - stack segment (SS)
    - data segment (DS)

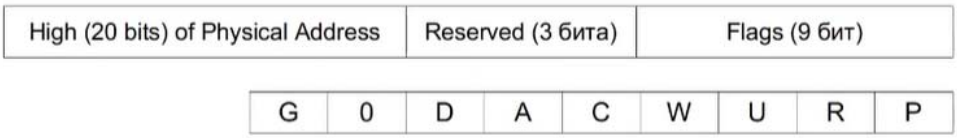
Physical_Addr = (Segment << 4) + Offset

Memory Paging

- Memory split into **pages** of equal size
- Each page has its own attributes

x86-32 Memory Paging

- CR2 register value points to Level-1 Table for current process
- Each Level-1 Entry points to Level-2 Table
- Level-2 Table contains entries on individual pages
- Common page specifications for x86-32:
    - page size 4K
    - each table has up to 1024 entries



- G (global) — ignored in modern implementations
- S (size) = 0 – page size (0 for 4K)
- D (dirty) — has modifications
- A (accessed) — was accessed
- C (caching) — disable cache for page

- W (write-through) — write-through instead of write-back caching
- U (userspace) — page accessible by regular user
- R (read+write) — write access
- P (present) — page exists at physical memory

Page Fault

- P==0
- Processor throws an exception
- The Kernel handles Page Fault exception:
    - page might be at swap area but not in memory
    - page mapping to file has not read yet
    - write to page copy (Copy-on-Write)

Бывают ситуации, когда страница в памяти отсутствует. При попытке обращения к такой странице возникает ошибка под названием Page Fault. При такой ошибке возникает сигнал, который обрабатывается ядром.

В каких ситуациях может быть Page Fault? Существует такое понятие как раздел подкачки либо файл подкачки, когда размер виртуального адресного пространства заведомо больше, чем объем доступной памяти. У вас есть несколько запущенных процессов, и они выполняются не совсем параллельно, последовательно и хотят много памяти. Но поскольку физической памяти если у вас меньше, то часть данных, которые очень редко используются либо использовались очень давно могут периодически уходить на жесткий диск в swap раздел либо swap файл. В этом случае они реально выгружаются из памяти и помечаются в таблице страниц как отсутствующий (в флаг P проставляется 0), когда процессор пытается обратиться к этой странице возникает ошибка, соответственно ядро подгружает эту страницу из swap раздела или swap файла и выполнение продолжается.

Еще одна возможная ситуация — это вы пытаетесь записать в некоторую страницу, которая пока еще не создана. Обычно происходит при реализации концепции Copy-on-Write, т.е. все страницы у нас наследуются дочерними процессами, когда вы создаете новый процесс он наследует все ваши страницы памяти и реально когда вы запускаете дочерний процесс, у вас содержимое не копируется, у вас копируются только таблицы страниц, ну а затем любой дочерний процесс который использует ту же самую память если он ее использует только для чтения, то это будет отсылка к одной и той же физической памяти. Когда это бывает осмыслено? Ну, например вы запускаете два экземпляра одной и той же программы. Она уже один раз была загружена в файл, зачем в физической памяти хранить второй экземпляр? Поэтому те части, которые доступны только для чтения и хранятся ровно в одном экземпляре и в таблицах памяти хранятся указатели, позволяющие добраться до одной и той же области физической памяти если память у вас доступна на чтение + запись, то опять же они хранятся в одном экземпляре, но до первой попытки что-то записать. При первой попытке записать также получаете Page Fault и ядро в этот момент создает копию отдельной страницы, которая связана уже с другим процессом, т.е. модифицирует таблицу страниц для текущего процесса.

Hardware Support

- MMU (Memory Management Unit)

calculate actual pointer transparently to program
- TLB (Translation Lookaside Buffer)
  keeps memory mapping tables for current process
- Page size is CPU but not OS-specific implementation
- x86(64):
  - 4Kb (typical usage)
  - 2Mb or 2Gb (Huge Pages if Kernel support enabled)

Kernel Space

- U==0
- Not accessible from user mode
- Accessible from kernel mode
- System call using sysenter/syscall instructions has no effect on TLB, but just switches CPU mode

В чем принципиальное отличие syscall/sysenter от программных прерываний? Программное прерывание переключает содержимое кэша TLB, т.е. сбрасывает все эти страницы, погружает страницы, которые соответствуют ядру, в случае если выполняете системный вызов с помощью команды syscall, то содержимое TLB точно такое же, только для доступа к памяти в ядре используется верхняя часть виртуального адресного пространства.

Allocators

- C++: default new and new[] — uses malloc/calloc, might be customized
- C: **functions** (not system calls!) malloc/calloc allocates memory from The Heap
- Allocation implementations might not be compatible with each other

В чем принципиальное отличие функции malloc от calloc? Malloc выделяет память в куче и там может быть какой-то мусор. calloc гарантирует, что у вас там не будет мусора, а все будет заполнено нулями.

Heap Allocations

- **brk** system call: legacy, but simple.
  Default glibc malloc uses for heap up to 128K (configurable via mallopt(3))
- **mmap** system call: flexible pages allocation

`mmap`

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

- `addr` — recommended or required (with `MAP_FIXED` flag) address where to place region
- `length` — region size
- `prot` — protection flags (`EXEC/READ/WRITE/NONE`)
- `flags` — allocation flags (`SHARED` — shared access for processes of `PRIVATE` — use Copy-on-Write)
- `fd`, `off_t` — file to map and its offset or -1 and 0 values. Not portable! See notes for each OS

Modern malloc Implementation

- jemalloc (FreeBSD, 2006)
- Modern BSDs and Linux have similar implementations
- Several **arenas** (twice by processors count by default)
- Different threads associated to different areas to leverage lock bottlenecks

Memory Limitations

```
[/.../victor]$ ulimit -a
core file size          (blocks, -c) unlimited
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 45543
max locked memory       (kbytes, -l) 64         # non-swappable memory
max memory size         (kbytes, -m) unlimited  # maximum memory size
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192       # stack size
cpu time               (seconds, -t) unlimited
max user processes              (-u) 4096
virtual memory          (kbytes, -v) unlimited  # address space size
file locks                      (-x) unlimited
```

```
int getrlimit(int resource, struct rlimit* rlim);

int setrlimit(int resource, const struct rlimit* rlim);
```

Out-of-Memory Killer

OOM Killer:

- pick a victim to kill
- free memory

`/proc/…/oom_score`

  - unsigned value
  - 0 by default
  - higher value — higher probability to be killed

Out-of-Memory Killer

- `proc/[pid]/oom_score` — current killness rating
- To be calculated by The Kernel
- More memory consumption — higher score
- Privileged processes — less score

OOM Adjustment

- `proc/[pid]/oom_adj`
  - [legacy] values from -17 to +17

- `proc/[pid]/oom_score_adj`
    - [legacy] values from -1000 to +1000

Overcommit

- Delayed memory allocation
- Can't check for memory available

```
int * ptr = malloc(TooMuchSize);

getchar();

ptr[0] = 123;
```

System Parameters Tuning

- `sysctl` command
- `/etc/sysctl.d/*.conf` files on boot
- Tunable parameters:
    - vm.overcommit_memory (`/proc/sys/vm/overcommit_memory`) Overcommit strategy
    - vm.overcommit_ratio (`/proc/sys/vm/overcommit_ratio`) Overcommit rate (in percents of memory)

System Parameters Tuning

- OVERCOMMIT_ALWAYS (1)
- OVERCOMMIT_GUESS (0)
- OVERCOMMIT_NEVER (2)

- Total memory size:
  swap + mem_size * (overcommit_ratio/100)

Лекция 11 — Процессы

Что такое процесс

- Процесс — это экземпляр программы в одном из состояний выполнения
- Процесс — это изолированное виртуальное адресное пространство в UNIX-системах

Атрибуты процесса

Память:

- Значения регистров процессора
- Таблицы и каталоги страниц виртуального адресного пространства
- Private и Shared страницы памяти
- Отображение файлов в память
- Отдельный стек в ядре для обработки системных вызовов

Файловая система:

- Таблица файловых дескрипторов
- Текущий каталог
  почему нет программы cd?
- Корневой каталог
  root его может менять
- Маска атрибутов создания нового файла umask

Другие атрибуты:

- Переменные окружения
- Лимиты
- Счетчики ресурсов
- Идентификаторы пользователя и группы

Информация о процессах

- Команда ps — список процессов
- Программа top — потребление ресурсов
- Файловая система /proc

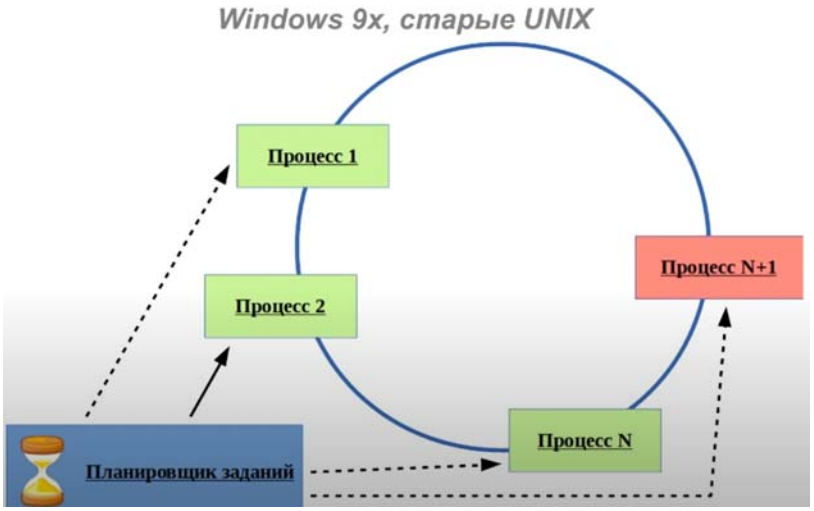
Жизненный цикл процесса

- Выполняется — ***R***unning
- Остановлен — s***T***opped
- Временно приостановлен
  - ***S***uspended — может быть завершен
  - ***D***isk suspended — не может быть завершен
- Исследуется — ***t***racing
- Зомби — ***Z***ombie

Примеры переходов

```
sleep(10);
// переход из R в S

read(0, buffer, sizeof(buffer));
// возможен переход из R в S

read(fd, buffer, sizeof(buffer)); // где fd - файл
// возможен переход из R в D

_exit(5);
// переход из R в Z

raise(SIGSTOP);
// переход из R в T
```

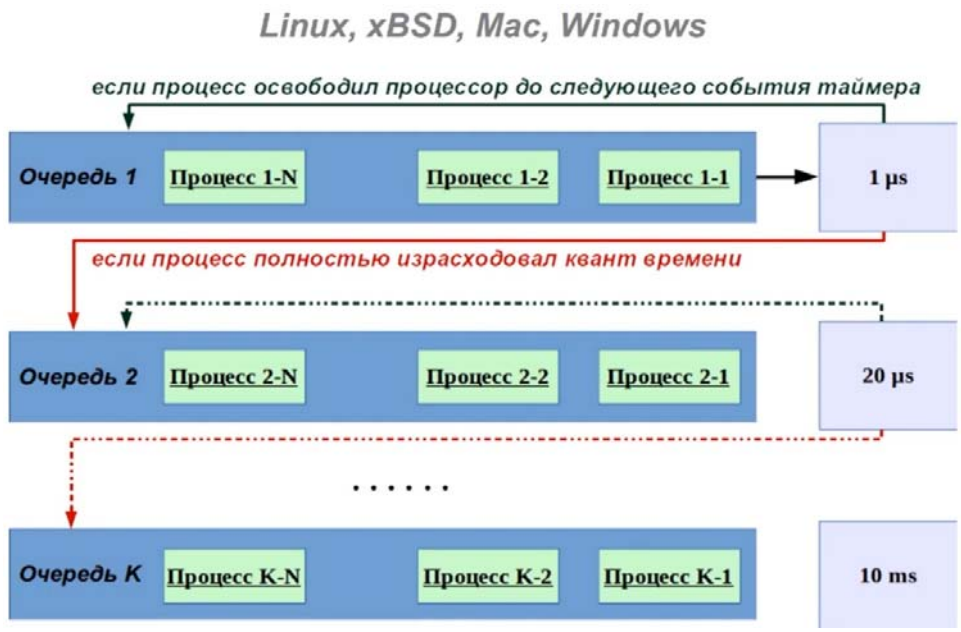
Round Robin



Приоритет

- Значение от -20 (самый высокий) до +19 (самый низкий)
- Численное значение — сколько раз пропустить планировщиком заданий
- Команды nice и renice
- Системный вызов `nice(int inc)`
- Только root может повышать приоритет

Multilevel Queue



Ничегонеделание

```
while (1) {
   // do nothing - just waste CPU
}
```

```
while (1) {
   sched_yield(); // OK
}
```

Создание процесса

```
pid_t resut = fork()
```

Создает **<u>копию</u>** текущего процесса

- -1 == result — ошибка
- 0 == result — для дочернего процесса
- 0 < result — для родительского процесса, тогда result — это Process ID

Копия процесса

- Память, регистры etc. — точная копия (кроме регистра %eax/%rax)
- НЕ копируются:
  - Process ID [getpid()], Parent ID [getppid()]
  - Сигналы, ожидающие доставки
  - Таймеры
  - Блокировки файлов

Ограничения

- /proc/sys/kernel/pid_max [32768]
  максимальное число одновременно запущенных процессов
- /proc/sys/kernel/threads-max [91087]
  максимальное число одновременно выполняющихся потоков (каждый процесс — уже один поток)

Дерево процессов

- процесс с номером 1 — ~~**init**~~ **systemd**
- У каждого процесса, кроме ~~**init**~~ **systemd** есть свой родитель
- Если родитель умирает, то его родителем становиться процесс с номером 1
- Если ребенок умирает, про это узнает его родитель

Завершение работы процесса

- Системный вызов _exit(int)
- Функция exit(int)
- Оператор `return INT` в `main`

Завершение работы

- Функция exit:
  - вызывает обработчики завершения, зарегистрированные функцией atexit
  - сбрасывает потоки stdio
  - удаляются файлы, созданные tmpfile
  - вызывается системный вызов _exit

Ожидание завершения процесса

- `pid_t waitpid(pid_t pid, int *status, int flags)`

- pid — ID процесса, или -1 для произвольного дочернего, или <1 для группы процессов

- status — куда записать результат работы
- flags — флаги ожидания:
  - 0 — по умолчанию
  - WNOHANG — не ждать, а только проверить
  - WUNTRACED — считать событие sTopped

Код возврата

- Процесс может завершиться добровольно, используя _exit(0<=code<=255)
- Процесс может быть принудительно завершен сигналом

```
int status;
waitpid(child, &status, 0);
if (WIFEXITED(status)) {
   printf("Exit code: %d", WEXITSTATUS(status));
}
else if (WIFSIGNALED(status)) {
   printf("Terminated by %d signal", WTERMSIG(status));
}
```

Zombie (<defunc>) — процессы

- После своего завершения процесс еще не похоронен — его статус zombie
- Удалением зомби из таблицы процессов занимается родитель — вызовом wait или waitpid
- Если зомби не удалять — получается эффект fork-бомбы

exec — замещение тела процесса другой программой

man 3 exec

```
int execl(const char *path, const char *arg, ..., /* 0 */)
int execlp(const char *path, const char *arg, ..., /* 0 */)
int execle(const char *path, const char *arg, ..., /* 0 */, char * envp[])

int execv(const char *path, char * const argv[])
int execvp(const char *path, char * const argv[])

#ifdef _GNU_SOURCE
int execvpe(const char *path, char * const argv[], char * const envp[])
#endif
```

- Передача параметров
  - ‘l’ — переменное число аргументов
  - ‘v’ — массив параметров
- Передача переменных окружения
  - ‘e’ — дополнительно задается envp
- Поиск программы в PATH
  - имя программы может быть коротким

Признак конца строки — ‘0’

Признак конца массива — NULL

```
int execvpe(const char *path, char * const argv[], char * const envp[])

int main(int argc, char *argv[])
int main(int argc, char *argv[], char *envp[]) // not portable!

char *getenv(const char *name); // POSIX
```

Команда env — отображение переменных окружения. Примеры:

- PATH — где искать программы
- LD_LIBRARY_PATH — где ld должна искать библиотеки
- HOME — домашний каталог

Атрибуты процесса, сохраняемые exec

- Открытые файловые дескрипторы
- Текущий каталог
- Лимиты процесса
- UID, GID
- Корневой каталог — только root

SUID-флаг

- Дополнительный атрибут выполняемого файла
- Означает, что файл запускается от имени того пользователя, который является владельцем файла

setuid / getuid vs geteuid

- `setuid(uid_t)` — установить Effective UID
- `getuid()` — получить реальный UID
- `geteuid()` — получить effective UID

Лимиты

- Объемы памяти:
  - адресное пространство
  - стек
  - RSS
- Открытые файлы
- Количество процессов
- Время (процессорное!) работы процесса

Лекция 12 — Взаимодействие процессов: сигналы

Завершение работы процесса

- Добровольное:
  - выход из функции main
  - вызов функции `exit`
  - системный вызов `_exit`
- Подходит для детерминированного выполнения: у программы есть начало и конец

Сигнал — это что-то, что посылается процессу для его завершения.

Завершение работы процесса

- Принудительное — отправкой сигнала:
  - команда **`kill`**
  - команда **`killall`**
  - запуск через **`timeout`**
  - кнопочки Ctrl+C
  - закрытие вкладки терминала
  - выключение или перезагрузка
- Это все — штатные ситуации, которые нужно учитывать
- Событие может произойти **асинхронно**

Завершение работы процесса

- Ошибка, за которую можно поплатиться:
  - нарушение сегментации
  - запись в закрытый канал или сокет
  - деление на ноль
  - недопустимая инструкция
  - нарушение assertion
- Действие по умолчанию — кто-то прибивает процесс отправкой **сигнала**

Сигнал

- Асинхронное событие, которое может отправить процессу:
  - ядро
  - другой процесс, если у него есть право на это
- Что может сделать процесс:
  - игнорировать
  - самовыпилиться (иногда с Core Dump для gdb)
  - изменить состояние: sTopped | Running
  - сделать что-то еще

Сигналы

man 7 signal

(серые сигналы отправляются ядром, остальные обычно пользователем)

(красные сигналы нельзя переопределить)

Номер	Имя	По умолчанию	Описание
1	SIGHUP	Term	обрыв соединения
2	SIGINT	Term	Ctrl+C
3	SIGQUIT	Core	Ctrl+\
4	SIGILL	Core	плохая инструкция
6	SIGABRT	Core	abort()

Номер	Имя	По умолчанию	Описание
9	SIGKILL	Term	убийство
11	SIGSEGV	Core	что-то плохо с памятью
13	SIGPIPE	Term	Broken pipe
15	SIGTERM	Term	завершение работы
17	SIGCHILD	Ign	завершился дочерний проц.
18	SIGCONT	Cont	Команда fg
19	SIGSTOP	Stop	Ctrl+Z
23	SIGURG	Ign	Socket urgent data

Core Dump

- Снимок памяти процесса
- Полезен для отладки
- В современных системах управляется systemd

```
/usr/sbin/sysctl kernel.core_pattern
            ulimit -c
```

Чем отличается сочетание клавиш Ctrl+\ от Ctrl+C? Отличается тем, что помимо завершения работы здесь генерируется dump памяти (dump памяти обычно генерируется в случае возникновения каких-то ошибок).

Бывают ситуации, когда ядро посылает, например, если вы пытаетесь выполнить какую-то инструкцию, которая отсутствует, то процессор генерирует прерывание, прерывание обрабатывается ядром и ядро отправляет процессу SIGILL. Не путаете с SIGKILL! Здесь точно так же генерируется Core Dump и процесс завершает свою работу.

Что такое дамп памяти? Это целиком снимок памяти для всего процесса, т.е. фактически это дамп всего адресного пространства, который может быть полезен в том случае, если у вас есть отладочная информация, процесс завершился не очень корректно ну а дальше с помощью отладчика вы можете загрузить этот дамп в том числе на другой машине, из-под которой вы выполняете разработку, как-то его исследовать.

Обработка сигналов [deprecated]

```
#include <signal.h>

// Тип sighandler_t - только в Linux!
typedef void (*sighandler_t)(int);


sighandler_t
signal(int signum, sighandler_t handler);
```

Обработка сигналов [deprecated]

```
#include <signal.h>

// Тип sighandler_t - только в Linux
typedef void (*sighandler_t)(int);

// Тип sig_t - только в *BSD
typedef void (*sig_t)(int);


sighandler_t
signal(int signum, sighandler_t handler);
```

Немного о стандартах

- Сигналы System-V (Solaris)
- Сигналы BSD (в том числе Linux)

gcc -std=c99 *v.s.* gcc -std=gnu99

~~#define _BSD_SOURCE~~

#define _DEFAULT_SOURCE

#define _GNU_SOURCE

Сигналы BSD vs System-V

- В System-V обрабатываются один раз, в BSD — до отмены обработчика
- В System-V во время обработки сигнала может быть вызван обработчик другого сигнала, в BSD — блокируется до завершения обработки
- В System-V блокирующие системные вызовы завершают свою работу (EINTR в errno), в BSD — автоматически перезапускаются (кроме sleep, pause, select)

Обработка сигналов [modern way]

```c
#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);
};

int
sigaction(int signum,
          const struct sigaction *act,
          struct sigaction *oldact /* might be NULL */);
```

Системный вызов signal

- Можно использовать только с обработчиками SIG_IGN (=1) и SIG_DFL (=0)
- В Linux системный вызов ведет себя в стиле System-V
- В стандартной библиотеке Си, если объявлен макрос _BSD_SOURCE или _DEFAULT_SOURCE, то signal — это функция-оболочка поверх sigaction, а не signal

Обработчик сигнала

- Может быть вызван в **произвольный момент времени**
  - использует текущий стек (хотя это можно настроить)
  - для x86_64 гарантируется RedZone размером 128 байт
  - может использовать ограниченный набор функций: только асинхронно-безопасные (AS-Safe)

Async Signal Safety

- Классы функций:
  - Небезопасные (Unsafe)
  - Потоко-безопасные (MT-Safe)
  - Асинхронно-безопасные (AS-Safe)
- Множества функций AS-Safe и MT-Safe не совпадают! Пример: `fwrite`
- Полный список функций: `man 7 signal-safety`

Обработчик сигнала

- Тип данных **`sig_atomic_t`**
  - целочисленный
  - **`int`** для большинства платформ
  - гарантируется «атомарность» чтения/записи, но только на уровне обработки сигнала
- Ключевое слово **`volatile`**
  - указывает компилятору, что ни в коем случае нельзя оптимизировать использование переменной

Обработчик сигнала

- Может использовать только асинхронно-безопасные функции
- Должен выполняться как можно быстрее
- Правильная стратегия: проставить флаг о том, что поступил сигнал, а затем его проверить, не нарушая логики работы программы

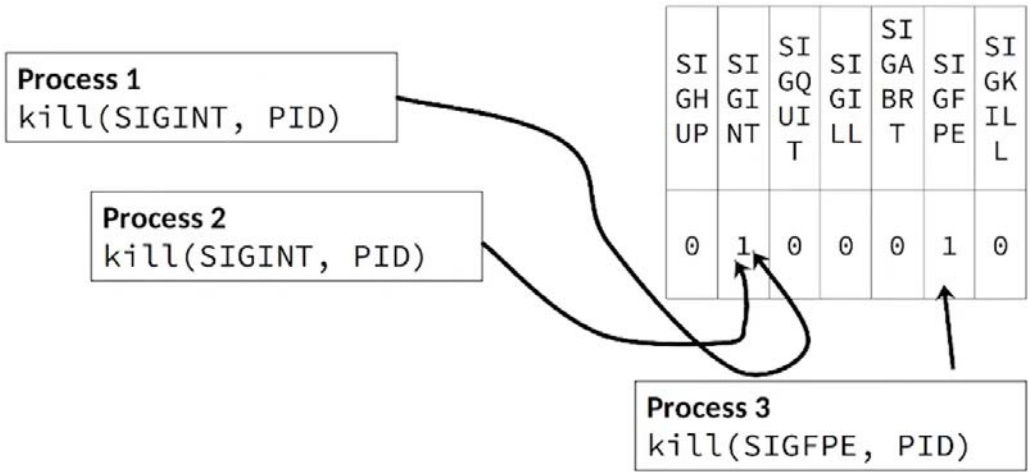
Примеры обработчиков сигналов

- Для SIGTERM и SIGINT — корректно завершить свою работу
- Для SIGINT возможно запросить «Вы уверены?»
- Для SIGHUP — перечитать файл с настройками
- Для SIGCHILD — прочитать код возврата дочернего процесса
- Для SIGSEGV —попытаться (без каких-либо гарантий) сохранить важные данные

Маска сигналов, ожидающих доставки

- Один из атрибутов процесса
- Не наследуется при клонировании системным вызовом fork

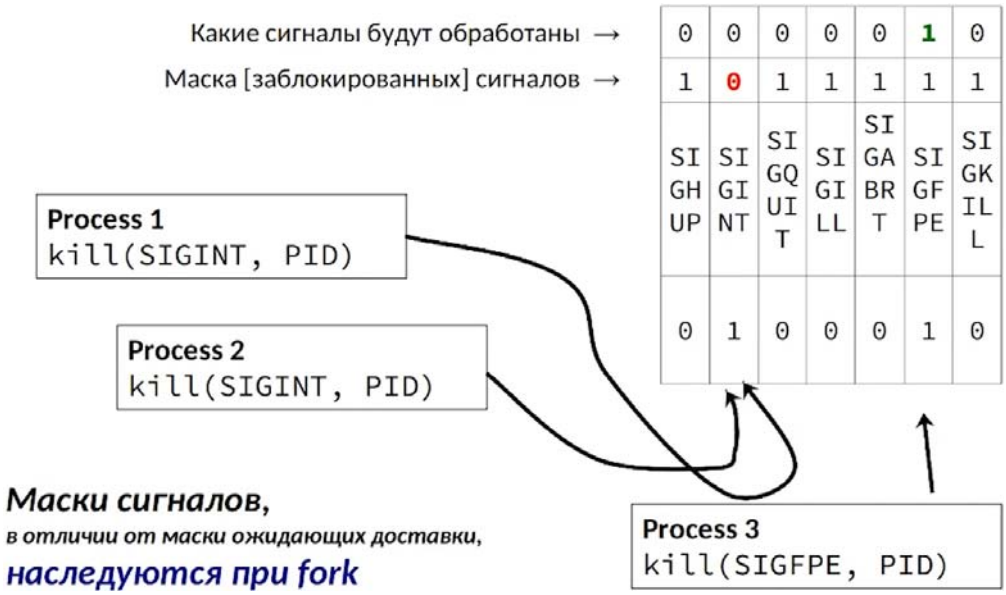
Учитывается только факт наличия сигнала, но не их количество!



Доставка сигнала

- Произвольный процесс или ядро устанавливают нужный флаг в маске другого процесса
- Выполнение продолжается до тех пор, пока не планировщик не выберет другой процесс
- Когда планировщик заданий добирается до того процесса, который получил сигнал, то первым делом проверяется маска сигналов, ожидающих доставки

Маски сигналов (у каждого thread-а своя)



Изменение маски сигналов

- Установка маски для процесса

```
sigprocmask(int how,
            const sigset_t *mask,
            sigset_t *old_mask)
```

- Установки маски для отдельной нити (если используется многопоточность)

```
pthread_sigmask(int how,
                const sigset_t *mask,
                sigset_t *old_mask)
```

Множества сигналов

- Стандартом не регламентируется содержимое sigset_t
- Для инициализации и модификации используются функции (man sigsetops)
  - sigemptyset
  - sigfillset
  - sigaddset
  - sigdelset
  - sigismember

Заблокированные сигналы не исчезают, а откладываются до тех пор, пока не будут разблокированы.

Ожидание поступления сигнала

- **pause**() — приостановить выполнение до прихода и обработки **любого незаблокированного** сигнала
- **sigsuspend**(**sigset_t** *set) — приостановить выполнение до прихода **инвертированного множества** сигналов, **игнорируя все остальные**

Файловый дескриптор signalfd (*только Linux*)

```
int signalfd(int old_signal_fd_or_minus_1,
             const sigset_t *mask,
```

```
                int flags)
```

- Создает файловый дескриптор для “чтения” событий о поступающих сигналах из множества mask
- Можно читать объекты типа **`struct signalfd_siginfo`**

Обычные сигналы

- Имеют стандартное назначение для большинства UNIX-подобных систем
- Процесс может проверить факт того, что во время его паузы ему были доставлены сигналы, но не их количество
- Обрабатывать поступившие сигналы процесс имеет право в произвольном порядке

Сигналы реального времени

POSIX.1b real-time extensions; реализованы в Linux

- Имеют номера от SIGRTMIN до SIGRTMAX
- Могут быть использованы как обычные сигналы (доставка через `kill`)
- Могут быть доставлены через очередь доставки (в этом случае гарантируется порядок доставки и количество)
- Могут нести дополнительную информацию — целое число в поле `si_value` структуры `siginfo_t`

Отправка сигналов реального времени

```
POSIX:
sigqueue(int pid,
         int signum,
         const union sigval value)
union sigval {
  int   sival_int;
  void* sival_ptr;
};

LINUX:
sys_rt_sigqueueinfo(
   int pid,
   int signum,
   const siginfo_t
         *uinfo
)
```

Что такое сигнал реального времени? Это сигнал — порядок доставки которого всегда гарантирован и отличается от обычных сигналов тем, что не проставляет флаг в маске сигналов ожидающих доставки, а складывается в очередь сигналов и помимо информации о том что это за сигнал и от какого PID он отправлен можно передавать произвольное целочисленное значение размером не более чем машинное слово.

Лекция 13 — Межпроцессное взаимодействие: каналы

Процесс в UNIX (и не только)

- Изолированное адресное пространство
- Виртуальность адресного пространства гарантируется процессором
- Побочный эффект: требуются дополнительные действия для взаимодействия разных процессов

Способы взаимодействия

- Сигналы (про них уже рассказывалось)
  - Обычные сигналы UNIX
  - POSIX Real-Time Extensions

  Для обычных сигналов не гарантируется доставка
  RT-сигналы могут передавать только одно машинное слово
- Разделяемые страницы памяти:
  mmap(…, MAP_SHARED, …)
- Обычные файлы
  Проблема синхронизации: данные могут быть записаны в отображаемые файл в произвольный момент времени

BSD flock и POSIX file lock

- Системный вызов flock:
  - один производитель и много потребителей
  - простая семантика:
    read-only, write-access, unlock
  - блокируется файл целиком
- POSIX file lock:
  - fcntl(int fd, F_SETLK, struct flock*)
  - в структуре struct flock можно указать отдельные диапазоны

Команда flock

- Сценарий использования: возможность запуска произвольной команды ровно в одном экземпляре
  flock **-w 1 ИМЯ_LOCK_ФАЙЛА КОМАНДА**

Способы межпроцессного взаимодействия

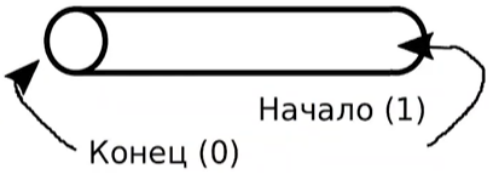
- Каналы
  - именованные: файл типа fifo
  - не именованные: pipe для родственных процессов
- Разделяемая память
  - mmap для родственных процессов
  - mmap на временный файл
- Сокеты
  - локальные (UNIX) сокеты: специальный тип файла
  - сетевые сокеты

Каналы в UNIX

```
      ls        -l        |      wc
#  команда   аргумент   палочка  команда
```

- Запускается две команды
- Стандартный поток вывода первой команды перенаправляется на стандартный поток ввода второй команды
- Используемый при этом механизм — канал (pipe)

Канал в UNIX

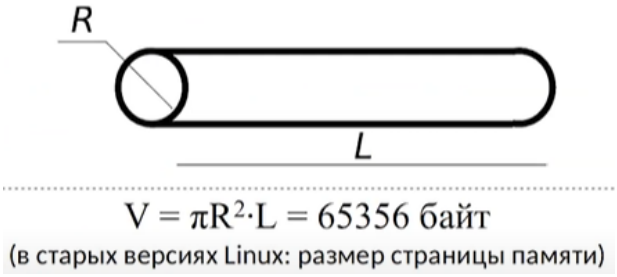


- Системный вызов pipe(int fds[2])
  - параметр — куда записать пару дескрипторов
  - индекс 0 — для чтения, индекс 1 — для записи

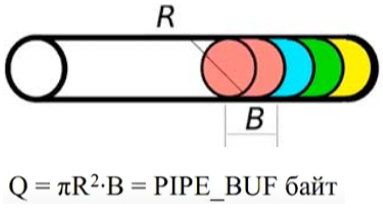
Использование каналов

- Межпроцессное взаимодействие
  - процессы наследуют открытые файловые дескрипторы
  - после создания канала можно делать fork+exec
- В пределах одного процесса:
  - однопоточность — пока есть место
  - использовать разными потоками выполнения

Канал в UNIX



Канал в UNIX



- Гарантируется атомарность чтения/записи блоков размером PIPE_BUF
- Для Linux (x86) размер PIPE_BUF = 4096
- По стандарту POSIX.1-2001 размер PIPE_BUF >= 512

Когда возникает проблема атомарности

- Несколько потоков в пределах одного процесса разделяют один канал
- Несколько процессов имеют файловые дескрипторы, связанные с одним каналом

Запись в канал

- Если в канале есть место и его противоположная часть открыта **хотя бы одним** процессом — происходит запись
- Если нет места:
  - запись блокируется до тех пор, пока кто-то не прочитает; либо
  - если установлен флаг O_NONBLOCK, то write завершается с ошибкой EAGAIN
- Если канал никем не открыт на чтение, то получаем сигнал SIGPIPE и write завершается с ошибкой EPIPE

Неблокирующий ввод-вывод

- При открытии файлов можно указать флаг O_NONBLOCK
- Чтение и запись из/в неблокирующие дескрипторы не приводят к остановке, а возвращают ошибку EAGAIN
- Для уже созданных файловых дескрипторов можно использовать fcntl

Чтение из канала

- Если есть данные — читается все, что есть
- Если пустой буфер и противоположная сторона открыта хотя бы в одном процессе — ожидание данных (кроме O_NONBLOCK)
- Если закрыты все файловые дескрипторы для записи — признак конца файла

Dead Lock

- Ситуация, когда пытаемся читать что-то из канала, в который никто не собирается писать
- При запуске процессов обычно проявляется из-за забытого close у родителя

Каналы в UNIX

```
      ls        -l        |       wc
#  команда   аргумент  палочка  команда
```

- 0 и 1 — стандартные потоки ввода и вывода
- Что нам создаст pipe — иногда предсказуемо, но в общем случае — random

dup

- dup — создание копии ФД:
  - int dup(int oldfd);
  - какое-то совсем legacy
- dup2 — создание копии ФД с номером
- dup3 — создание копии ФД с номером и флагами; Linux only

Копия ФД — очень низкоуровневый аналог Shared Pointer в C++

dup vs dup2

```
int dup2(int oldfd, int newfd);
```

1. Закрыть newfd, если он открыт
2. newfd = dup(oldfd)

Две операции происходят атомарно!

Именованные каналы

- Обычные каналы могут связывать только **родственные** процессы
- Пример:
  `ls -l | wc`
  Процесс bash создает канал, затем запускает два дочерних, которые наследуют эти файловые дескрипторы
- Можно договориться о том, как найти канал в неродственных процессах: по имени

Именованные каналы

- FIFO — специальный тип файла
- Создается функцией mkfifo (оболочка поверх mknod) или командой mkfifo
- Для превращения в файловый дескриптор — обычная операция открытия файла
- Ведет себя как неименованный канал

Каналы: в реальной жизни

- Файловые дескрипторы 0, 1 и 2
- Использование сторонних программ как компонент
- Типовой пример: отладчик gdb
  … а еще gcc
- Обычно используются неименованные каналы (pipe), а не FIFO

Ограничения каналов для IPC

- Однанаправленность
- Проблемы с атомарностью, если несколько читателей/писателей

Примитив: socket pair

- Только FreeBSD и Linux
- Двунаправленный канал связи

```
socketpair(AF_LOCAL,
           SOCK_STREAM,
           0,
           out: int fds[2]
           ) → err
```

Краткое введение в сокеты

- Краткое, потому что этому посвящена следующая лекция
- Сокеты бывают разных видов, но используются только два из них:
  - локальные (AF_LOCAL)
  - сетевые (AF_INET или AF_INET6)

В чем принципиальное отличие сокетов от каналов? На каждое подключение дочернего процесса создается отдельный файловый дескриптор и это позволяет разделять отдельных клиентов, которые к вам могут подключаться.

Лекция 14 — Сокеты и сети

Способы взаимодействия

- Через файловый ввод-вывод
- Разделяемые страницы памяти (mmap)
- Каналы (pipes)
- Именованные каналы (FIFO)
- Сокеты — позволяют одному процессу общаться сразу со многими процессами

Сокет

- Файловый дескриптор для ввода/вывода
- Одно имя позволяет создавать подключения нескольких процессов (клиентов) к одному процессу (серверу)
- Пространство имен двух видов:
  - UNIX-сокет: файл в локальной файловой системе
  - TCP/IP-сокет: адрес хоста + номер порта

Создание сокета

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

Создает новый сокет и возвращает его файловый дескриптор

- domain — тип семейства адресов (**AD_UNIX**, **AD_INET**, AF_IPX, AF_APPLETALK, AF_BLUETOOTH и др.)
- type — **SOCK_STREAM** или **SOCK_DGRAM**
- protocol — **0** — выбирается автоматически; **IPPROTO_TCP** и др.

- Сокет создан, но еще не готов к использованию
- Файловый дескриптор может быть наследован (fork) или клонирован (dup2)
- Дескриптор должен быть закрыт после использования (close)

Подготовка сокета к работе

- Для клиента — нужно к кому-нибудь подключиться
  - *системный вызов* **connect**
- Для сервера — нужно анонсировать имя и ожидать подключений:
  - объявить имя с помощью **bind**
  - создать очередь входящих подключений и переключить в режим прослушивания **listen**
  - принять подключение **accept**

Объявление имени

```
int bind(int socket,
         const struct sockaddr *addr,
```

```
        socklen_t address_len)
```

Связывает сокет с некоторым адресом

- `struct sockaddr_in` — адрес IPv4
- `struct sockaddr_in6` — адрес IPv6
- `struct sockaddr_un` — адрес локального UNIX-сокета

Объявление имени

- Обязательно для приема входящих соединений
- Может потребоваться (но не обязательно) для исходящих соединений

Переключение сокета в режим прослушивания

```
listen(int sockfd, int backlog)
```

- backlog — размер очереди сообщений
- если подключается много (>backlog) процессов, то у них возникает ошибка подключения
- константа `SOMAXCONN` (128 в Linux) определяет максимальный размер очереди

Создание соединения

- `connect` — подключить сокет к другому процессу (в том числе на другом компьютере)
- `accept` — дождаться подключения, возвращает файловый дескриптор подключенного сокета для ввода-вывода

Ввод-вывод через сокеты

```
ssize_t recv(int socket, void* buffer, size_t buf_size, int flags)
```

Читает данные из сокета в указанный буфер. Флаги:

- `MSG_PEEK` — отметить данные как непрочитанные
- `MSG_OOB` — получить данные повышенного приоритета (Out-of-Band; в случае TCP — Urgent Data)
- `MSG_WAITALL` — дождаться полного получения данных и выдать их за один вызов

```
ssize_t send(int socket, const void *buffer, size_t size, int flags)
```

Записывает данные из буфера в сокет. Флаги:

- `MSG_OOB` — передать данные повышенного приоритета (Out-of-Band; в случае TCP — Urgent Data)
- `MSG_NOSIGNAL` — не отправлять процессу сигнал SIGPIPE, если сокет закрыт

```
read(Socket, Buffer, Size) → recv(Socket, Buffer, Size, 0)
```

```
write(Socket, Buffer, Size) → send(Socket, Buffer, Size, 0)
```

Обработка входящих подключений

1. Создать сокет [socket]

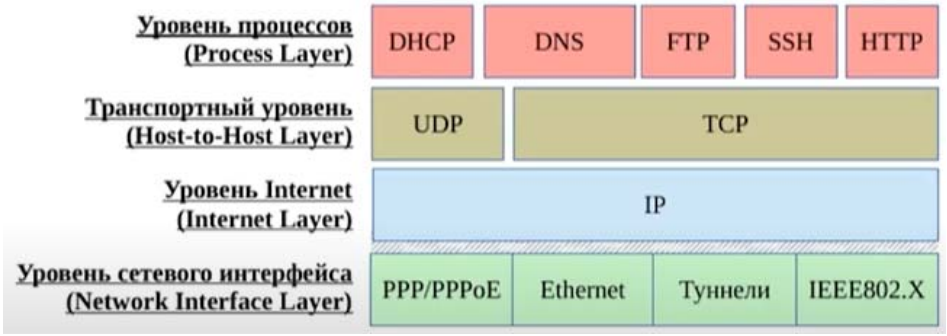
2. Связать с именем [bind]
3. Переключить в режим прослушивания и создать очередь подключений [listen]
4. Принимать очередное подключение [accept] в виде сокета
5. Ввод-вывод [read/write] или [recv/send]
6. Закрыть сокет, принятый через [accept]
7. goto шаг 4

Межпроцессное взаимодействие (UNIX-сокеты)

- Для домохозяек
  - X-сервер
  - DBUS-сервер
  - PulseAudio-сервер
- Для серьезных применений
  - отдельные веб-приложения
  - SQL-сервер
- Стандартное размещение сокетов:
  - `/var/run`
  - `/tmp/*`

Сетевое взаимодействие

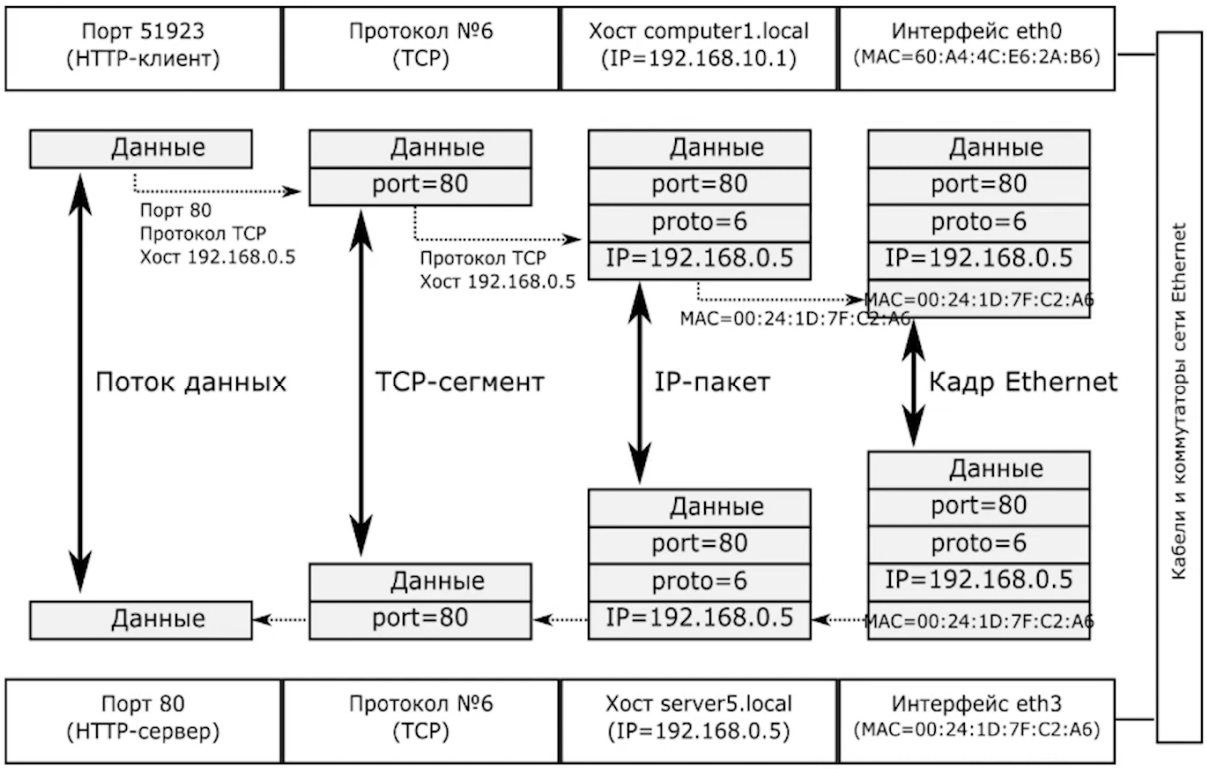
Стек TCP/IP



Модель OSI

Стек TCP/IP		Модель OSI (Open Systems Interconnections)	Примеры
Уровень процессов		Уровень приложений (Application)	HTTP, FTP, SSH, Telnet
		Уровень представления (Presentation)	ASCII, GZIP, binary
		Уровень сеанса (Session)	NetBIOS, SSL
Транспортный уровень		Уровень транспорта (Transport)	TCP, UDP
Уровень Internet		Уровень сети (Network)	IPv4, IPv6, IPX, AppleTalk
Уровень сетевого интерфейса		Уровень канала (Data Link)	PPP, IEEE 802.2 (Ethernet)
		Физический уровень (Physical)	USB, IEEE 802.11 IEEE 802.3 (Ethernet)

Передача данных в TCP/IP



Ethernet

DST MAC	SRC MAC	Length	Payload	CRC-32
6 байт	6 байт	2 байта	от 46 до 1500 байт (параметр MTU)	4 байта

Длина:

- 46…1500 — размер данных в фрейме
- 1536+ — тип данных в пакете

IPv4

Байты	0	1	2	3
0...3	Версия и размер заголовка	Тип службы	Размер IP-пакета	
4...7	ID группы пакетов		Флаги и смещение	
8...11	TTL	Номер протокола	Контрольная сумма заголовка	
12...15	Адрес отправителя			
16...19	Адрес получателя			
20...24	Дополнительные опции			

(как с этим можно бороться — 59:35)

Имя хоста

- IP-адрес — это 32-битное (IPv4) или 128-битное (IPv6) число
- Символическая нотация — определяется через службу DNS
- Зарезервированы адрес `127.0.0.1` и имя `localhost` — текущий компьютер
- Каждому DNS-имени может соответствовать несколько IP-адресов
- С каждым IP-адресом может быть связано несколько DNS-имен

Служба Domain Name System как раз ставит соответствие между определенными именами (человеческими) IP-адреса.

UDP

Байты	0	1	2	3
0..4	Порт отправителя		Порт назначения	
5..8	Длина пакета		Контрольная сумма	

TCP

Байты	0	1	2	3
0...3	Порт отправителя		Порт получателя	
4...7	Порядковый номер пакета			
8...11	Порядковый номер подтверждаемого пакета (с флагом ACK)			
12...15	Размер заголовка в 32-битных словах | 000 | NS | CWR | ECE | URG | ACK | PSH | RST | SYN | FIN		Размер окна (буфера для приема данных, ожидаемых при ответе)	
16...19	Контрольная сумма заголовка и данных		Указатель на порядковый номер пакета, в котором заканчивается приоритетных блок данных	
20.....	Дополнительные опции			

Номера портов

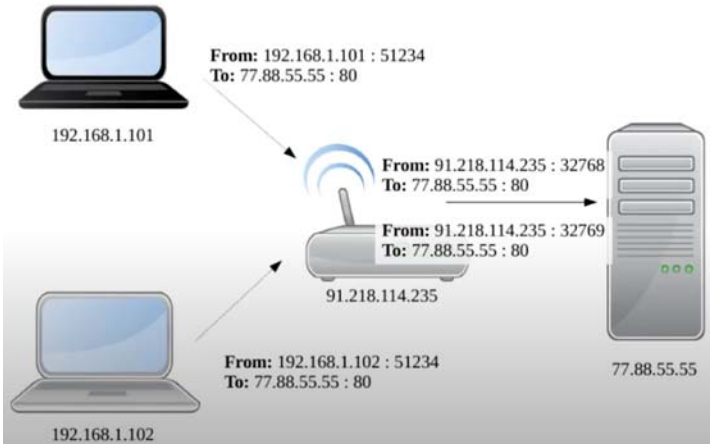
```
0             - не используется
20, 21        - FTP
22            - SSH
25            - SMTP
80            - HTTP
137,138,138   - NetBIOS
143           - IMAP
443           - HTTPS
465           - SMTPS
631           - CUPS
993           - IMAPS
```

1024…65535 — порты для клиентских соединений (старые UNIX-системы и Windows XP)

32768…65535 — порты для клиентских соединений (Linux)

49152…65535 — порты для клиентских соединений (BSD, Windows Vista+)

Переименование



Порты с номерами <1024

- Зарезервированы под стандартные службы
- Открыть может только root
- Запускать сервера под правами root — опасно!
  Решение проблемы:
- Создать сокет
- Создать дочерний процесс, который наследует fd сокета для выполнения настройки
- Дочерний процесс повышает свои привилегии до root и вызывает bind

Пакет authbind в Debian/Ubuntu

Лекция 15 — Многопоточность

Потоки

- Thread (aka поток, нить, легковесный процесс) — единица планирования времени процессора
- У каждого потока свой стек и состояние процессора
- В одном процессе может быть много потоков, — все они разделяют общее адресное пространство

Главный поток

- При запуске процесса работает один поток, выполняющий функцию `_start`
- Главный поток может запускать другие потоки
- Произвольные потоки могут запускать потоки
- Все потоки равнозначны; в отличии от процессов нет иерархии “родитель-ребенок”

exit

- Системный вызов `_exit` завершает работу процесса
- Системный вызов `pthread_exit` завершает работу текущего потока
- Можно завершить главный поток, но оставить работать остальные

Стек

Области стека могут чередоваться с динамически загружаемыми библиотеками.

Стек

- По умолчанию размер стека нового потока совпадает с размером стека главного потока
- В POSIX можно явно указывать размер стека и его размещение
- Ниже стека находится Guard Page
- **Никакой изоляции: любой поток может обращаться к памяти другого потока**

Процесс vs поток

- С точки зрения планировщика заданий, процесс и поток — это одно и то же
- Системный вызов `sched_yield` работает с текущим потоком; в случае однопоточной программы — с главным потоком
- Ограничение на “количество процессов” в системе Linux —это ограничение на суммарное количество потоков

Процесс vs поток

Процессы	Потоки
• Полная изоляция друг от друга • Падение одного процесса не влияет на работоспособность остальных • Процессы могут иметь разные привилегии • Взаимодействие - только через IPC	• Все находится в одном адресном пространстве • Сигнал (в том числе вследствии дефектов ПО) убивает все потоки, а не только проблемный • Все потоки работают под одним пользователем • Простое взаимодействие

Атрибуты различных потоков

Общие	Различающиеся
• Process ID • Parent process ID • Group ID/Session ID • Терминал • UID/GID • Открытые дескрипторы • Блокировки файлов • Обработчики сигналов • umask, cwd, root dir • Лимиты	• Thread ID • Стек потока • Стек обработчика сигналов • Приоритет выполнения • Маска сигналов pthread_sigmask • Значение errno

<errno.h>

- С точки зрения программиста — “глобальная переменная”
- Реализация — через макрос

```
/* The error code set by various library
functions.  */
extern int
  *__errno_location (void)
  __THROW __attribute_const__;
# define errno (*__errno_location ())
```

Реализации многопоточности

- POSIX (Linux, *BSD, MAC)
  `<pthread.h>`
  `Pthread_create(…)`

- WinAPI (Windows)
  `<Windows.h>`
  `CreateThread(…)`

WinAPI Threads

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES   lpThreadAttributes,
  SIZE_T                  dwStackSize,
  LPTHREAD_START_ROUTINE  lpStartAddress,
  LPVOID                   lpParameter,
  DWORD                   dwCreationFlags,
  LPDWORD                 lpThreadId
);
```

Минимальный размер стека — 64Кб

Стандарт C++11

- Универсальная реализация для всех платформ
  `<thread>`
  `std::thread(…)`

- Не поддерживает тонкую настройку параметров потока, например размер стека

Стандарт C11

- Стандарт определяет интерфейс

```
<threads.h>
void call_once(
        once_flag *flag,
        void (*func)(void)
);
```

- **Поддержка потоков C11 не реализована ни в одном компиляторе**

Запуск процессов

- Все потоки равноценны → можно вызвать `fork()` из любого потока
- При создании процесса создается копия всей памяти, включая стеки всех потоков
- В новом процессе будет существовать **только один поток**, который продолжает выполнение потока, вызвавшего `fork()`

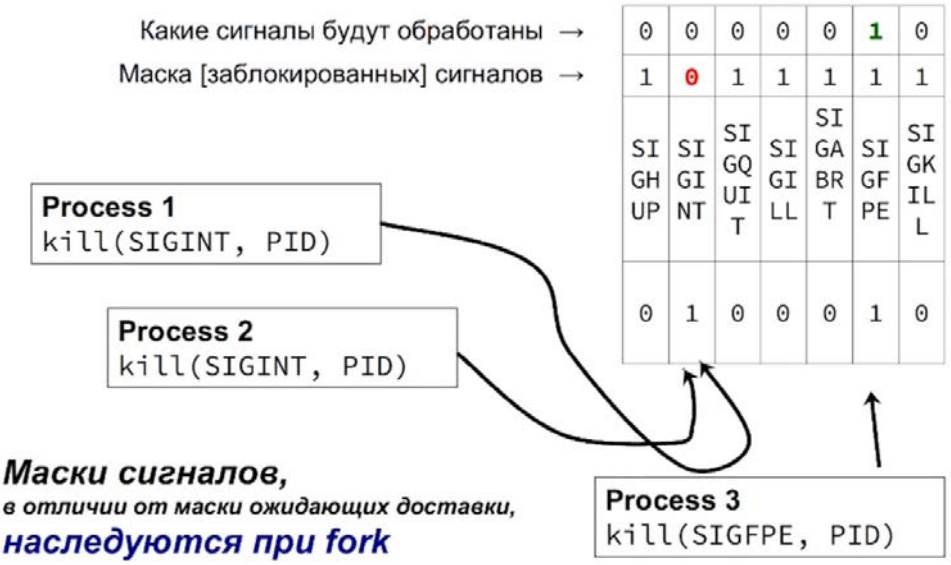
Запуск процессов

- Могут остаться заблокированные мьютексы/семафоры → deadlock
- Если предполагается использование `fork`, то необходимо зарегистрировать с помощью `pthread_atfork` функции, которые принудительно освободят блокировки

Выполнение exec

- Вызов `exec` полностью заменяет адресное пространство процесса
- После вызова exec остается только один поток
- Комбинация `fork+exec` относительно безопасна

Маски сигналов (у каждого thread-а своя)



Когда какой-нибудь другой процесс либо ядро отправляет сигнал вашему процессу, то в маске сигналов, ожидающих доставки будет проставлен соответствующий флаг, дальше этот флаг будет обработан каким-то из потоков.

Обработка сигналов

- Маска сигналов, ожидающих доставки — это свойство **процесса**, а не отдельного потока
- Обработчик сигнала находится в общем адресном пространстве для всех потоков
- Выполнение обработчика сигнала — в **произвольном** потоке
- Для отдельных потоков можно блокировать доставку сигналов (`pthread_sigmask`)

Принудительное завершение

- Антипаттерн проектирования. Следует избегать его использования
- Два способа принудительного завершения потоков:
  - умеренной жестокости: поток может быть остановлен в cancelation point — большинство системных вызовов
  - садомазохистский: поток прибивается принудительно, даже если там `while (1) {}`. Но это происходит не мгновенно, а когда планировщик заданий выберет отмененный поток

Что такое cancelation points? Это некоторые функции (системные вызовы), которые в процессе своей работы проверяют нужно ли выполнять текущий поток, либо ненужно, потому что он помечен как поток, который нужно завершить. В том случае, когда поток должен быть завершен, то его выполнение приостанавливается и поток завершает свою работу. Если такого флага нет, то системный вызов выполняет свою работу так, как должен выполнять и ничего особенного не происходит.

Принудительное завершение

- Функция `pthread_cancel` “завершает” работу указанного потока: отправляет запрос на остановку
- Поток может запретить обработку таких запросов: `pthread_setcancelstate(PTHREAD_CANCEL_DISABLE)`

Принудительное завершение

- При завершении могут остаться:
  - заблокированные семафоры и мьютексы
  - незакрытые файловые дескрипторы
  - потерянные указатели на память в куче
  - просто недоделанная работа
- Либо запрещаем остановку глобально, либо регистрируем “аварийную” функцию очистки для каждого ресурса: `pthread_cleanup_push`

Thread-Local Storage

- Глобальные переменные доступны всем потокам
- Глобальные переменные — зло, но иногда бывают необходимы

- Ключевые слова `_Thread_local` (C11) и `thread_local` (C++) объявляют переменные уникальными для каждого потока
- Значения TLS-переменных не изолированы от других потоков

Глобальные переменные

- Доступ к глобальным переменным из разных потоков приводит к data race
- Значения глобальных переменных могут меняться косвенно — через вызов функций стандартной библиотеки
- Многие функции помечены как Thread-Safe: они реализуют блокировки (ценой потери производительности)
- Пример: `getc` vs `getc_unlocked`

Как бороться с Data Race

- Атомарные переменные — выровнены по границе машинного слова <stdatomic.h> (C11)

```
typedef _Atomic _Bool atomic_bool;
typedef _Atomic char atomic_char;
typedef _Atomic signed char atomic_schar;
typedef _Atomic unsigned char atomic_uchar;
typedef _Atomic short atomic_short;
typedef _Atomic unsigned short atomic_ushort;
typedef _Atomic int atomic_int;
```

Необходимо использовать специальные атомарные операции

- Принудительные барьеры: мьютексы и семафоры

Проблемы проектирования

- Накоплена большая кодовая база для однопоточного выполнения
- Самый простой способ использовать такой код — ставить блокировки везде
- Этот способ — самый неэффективный

Пример: Python

- Запуск потока — это `pthread_create` + инициализация структуры `PyThreadState`
- Все честно: каждый поток имеет свой стек и выполняется как отдельная задача

но

- Используется разделяемый экземпляр `PyInterpreterState`

Global Interpreter Lock (GIL)

(как с ним бороться — 1:04:38)

Пример: Python. Как бороться

- Не использовать Python
- Для распараллеливания использовать модуль multiprocessing вместо multithreading
- Параллельную часть вынести в Си-код, который не трогает `PyInterpreterState`

```
Py_BEGIN_ALLOW_THREADS  // освобождение GIL
...... // какой-то Си-код, не использующий Py API
Py_END_ALLOW_THREADS    // захват GIL
```

Потоки внутри ядра

- Цель — распараллелить задачи внутри ядра между разными ядрами процессора
- Выполняются в адресном пространстве ядра
- В Linux всем потокам внутри ядра назначаются фиктивные ProcessID, которые процессами не являются
- Примеры:
  - kswapd — управляет разделом подкачки
  - kworker — [как правило] операции ввода-вывода

Giant Kernel Lock

- Глобальная блокировка, которая приводит к тому, что выполняется только один поток
- Процессы могут блокировать работу других процессов даже в многоядерных системах, вызывая системные вызовы
- Проблема — в Legacy-коде отдельных драйверов или подсистем
- В BSD проблема была окончательно устранена в 2009 году (FreeBSD 8.0)
- В Linux проблема была окончательно устранена в 2009 году (2.6.39)

BeOS / HaikuOS

https://www.haiku-os.org/

- Система изначально спроектированная для использования на SMP
- Каждая подсистема ядра работает в отдельном потоке

Лекция 16 — Неблокирующие сокеты. Событийно - ориентированное программирование

Взаимодействие с некоторыми клиентами

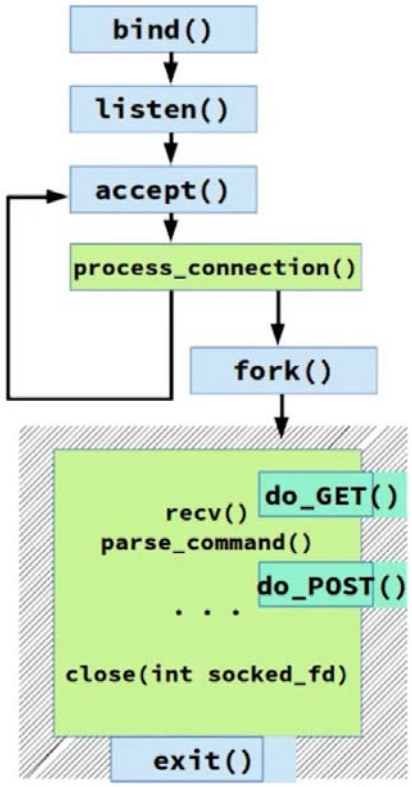
- socket
- bind
- listen
- **accept — прием соединения от клиента**
  - **read/write — блокирующий ввод-вывод**
  - close — закрыть сокет связанный

Решения проблемы многозадачности

1. На каждого клиента — отдельный процесс
   (+) дополнительная изоляция клиентов друг от друга
   (-) усложнение доступа к общим данным
- **accept — прием соединения от клиента**
  - **read/write — блокирующий ввод-вывод**
  - close — закрыть сокет связанный

2. На каждого клиента — отдельный легковесный процесс (thread)

- **accept — прием соединения от клиента**
  - **read/write — блокирующий ввод-вывод**
  - close — закрыть сокет связанный



(+) Проблема очередности обработки перекладывается на планировщик заданий

(−) Большой overhead, если клиентов ~1000+

3. Обработка в одном процессе, но операции ввода-вывода нужно сделать неблокирующими

- **accept — прием соединения от клиента**
  - **read/write — НЕблокирующий ввод-вывод**
  - close — закрыть сокет связанный

```
#include <fcntl.h>

int fcntl(int fields, int cmd, …);
```

- Управление файловыми дескрипторами
- Команда cmd — что сделать с файловым дескриптором
- Команда cmd может иметь аргументы
- `dup, flock` — могут быть реализованы через `fcntl` в Linux и BSD

Пример fcntl

- `dup(fd) → fcntl(fd, F_DUPFD, 0)`
- `dup2(fd, val) → fcntl(fd, F_DUPFD, val)`
- `flock(fd, op) →`

```
fcntl(fd, F_SETLK, &{
        .l_type = op,
        .l_whence = SEEK_SET,
        .l_start = 0,
```

```
            .l_len = file_size,
            .l_pid = getpid()
    })
```

Флаги

- fcntl(fd, F_GETFL) → int
- fcntl(fd, F_SETFL, int flags)

Примеры флагов:

- O_APPEND
- O_CREAT
- O_RDONLY
- **O_NONBLOCK**

Установка флагов:

- При создании (open)
- После создания (accept)

Блокировка вывода

- Вывод в локальный файл или устройство — буферизация, пока позволяет оперативная память
- Вывод в TCP-сокет — запись в буфер, размер которого определен в `/proc/sys/net/ipv4/tcp_wmem`:
  ```
  4096    16384    4194304
  min      def       max
  ```
  Настройка:
  `/proc/sys/net/core/wmem_default`
- При попытке вывести больше, чем есть место в буфере — вызов write *блокируется*.

Неблокирующий ввод-вывод

- Флаг **`O_NONBLOCK`**
- Для каналов используется системный вызов `pipe2(int pipefd[2], int flags)` *Linux-only*
- Для сокетов или уже открытых файловых дескрипторов — используется fcntl:
  ```
  int flags = fcntl(fd, F_GETFL);
  fcntl(fd, F_SETFL, flags | O_NONBLOCK);
  ```

Что если данных нет (или буфер заполнен для write)

- Для неблокирующего файлового дескриптора — ошибка чтения или записи `EAGAIN` (в переменной `errno`)
- В случае этой ошибки можно заняться более важными делами, и попробовать осуществить ввод-вывод позже

Неблокирующий accept

- В случае `accept` блокирующей операцией является ожидание нового подключения

- Для неблокирующего сокета, если нет подключений, — ошибка `EAGAIN`

Сервер без блокировок I/O

- 1 файловый дескриптор для приема входящих соединений
- + по одному дескриптору на каждого клиента
- Проверяются поочередно, пока не будет готовность данных (отсутствие ошибки `EAGAIN`)
- **Загрузка процессора 100%, даже при простое**

(Как с этим можно бороться — 27:50)

Поддержка событий со стороны ядра

- Регистрируется множество событий, на которые нужно реагировать
- Процесс переходит в состояние sleep
- Ядро обязано разбудить процесс при возникновении события

События, которые можно отслеживать

- Готовность чтения из дескриптора:
  - поступили данные в канал/сокет
  - запрос на подключение
  - признак конца файла
- Готовность записи в дескриптор
  - появилось место в буфере вывода
- Поступил сигнал
- Истекло время ожидания

`select` (BSD) и `poll` (SysV)

```
int select(int nfds, fd_set *readfds,
        fd_set *writefds, fd_set *exceptfds,
        struct timeval *timeout);
```

- Ожидание готовности до тех пор, пока не возникает событие на одном из ФД
- Возвращает количество готовых ФД и модифицирует значения указанных множеств
- Принадлежность ФД к множеству определяется функцией `FD_ISSET(int fd, fd_set &fds)`

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

- Работает аналогично `select`, но имеет немного отличающийся API

Окошки

- BSD-style select (<Winsock2.h>)
- SysV-style WSAPoll (<Winsock2.h>)

Проблема 100К входящих подключений

- Сложность select/poll: O(N), где N — количество отслеживаемых, а не готовых файловых дескрипторов

- Для медленного ввода-вывода и большого количества подключений — неоправданный overhead

FreeBSD Kernel Queue

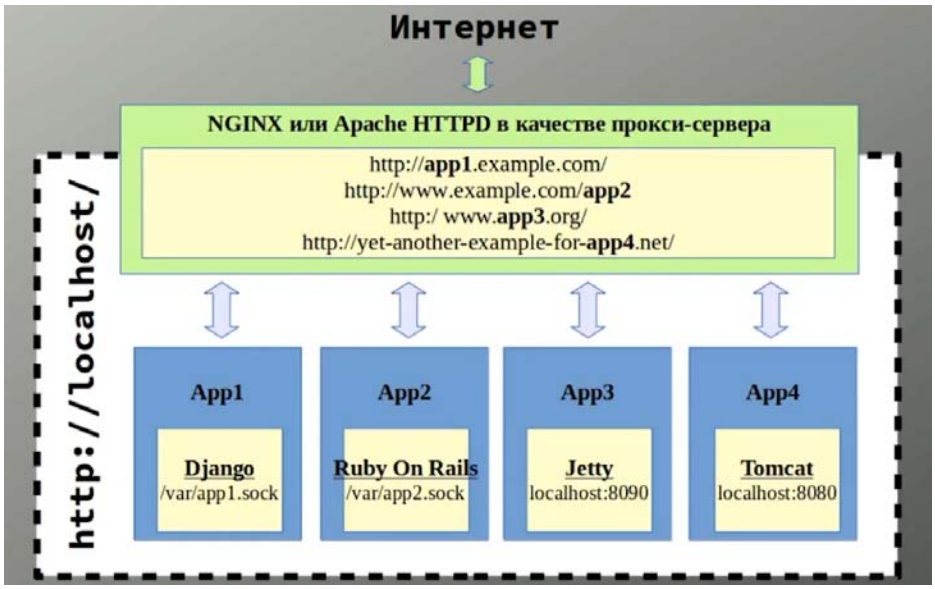
- В ядре создается очередь, куда складываются определенные события
- Пользовательский процесс ожидает от ядра, когда появится хотя бы одно событие; при его возникновении заполняется буфер из структур событий в пространстве процесса
- Количество элементов в заполненном буфере — те, для которых возникло событие

```
int kq_fd = kqueue(); // создание очереди kqueue
int fd = ...          // какой-то файловый дескриптор

struct kevent event;
event.data.fd = fd;              // чтобы знать, кто именно готов
event.filter = EVFILT_READ;      // интересует Read-Ready
event.flags = EV_ADD|EV_ENABLE; // действия при
                                 // возникновении события
// Регистрация ФД
kevent(kq_fd,           // куда добавить
       &event, 1,       // массив event и его размер
       NULL, 0, NULL); // здесь не нужно

// Ожидание события
kevent(kq_fd,           // очередь
       NULL, 0,         // здесь не нужно
       struct kevent *, int buf_size // массив, куда положить
       const struct timespec *to); // таймаут ожидания или NULL
```

NGINX

- Веб сервер, ориентированный на десятки тысяч одновременных подключений
- Использует фиксированное количество потоков
- Реализован через очередь ядра
- Нет изоляции на уровне процессов → только статика и прокси

HTTP-прокси



(как с этим можно бороться — 39:34)

Пингвины vs черти

```
int kq_fd = kqueue() epoll_create1(0); // создание очереди
int fd = ...           // какой-то файловый дескриптор

struct kevent epoll_event event;
event.data.fd = fd;
event.filterevents = EVFILT_READ EPOLLIN;
event.flags = EV_ADD|EV_ENABLE;

// Регистрация ФД
kevent epoll_ctl(kq_fd,  // куда добавить
          EPOLL_CTL_ADD, // добавить дескриптор в наблюдение
          fd, &event);   // ФД и событие

// Ожидание события
kevent epoll_wait(kq_fd, // очередь
       struct epoll_event *, int buf_size // массив, куда положить
       int timeout); // таймаут ожидания в ms или -1
```

Кросс-платформенность

- Библиотека libev [http://software.schmorp.de/pkg/libev.html]
- Boost.Asio [https://www.boost.org/doc//libs/1_60_0/doc/html/boost_asio.html]

Окошки Windows Vista +

- I/O Completion Ports [https://docs.microsoft.com/en-us/windows/desktop/fileio/i-o-completion-ports]
- Работает с очередью ядра, в отличии от select/WSAPoll
- Сообщает о том, что выполнено некоторое действие, а не готовность к выполнению, как kqueue/epoll

Событийно-ориентированное программирование

- **epoll_wait** блокирует текущий поток, пока не возникает некоторое *событие*
- работает с *файловыми дескрипторами*

Источники событий [только Linux]

- **signalfd** — поступление сигналов
- **timerfd_create** — периодические нотификации
- **eventfd** — произвольные события внутри программы

Лекция 17 —

Разделяемая память. Примитивы синхронизации

Взаимодействие процессов

- Через файловый ввод-вывод

- Каналы (pipes)
- Именованные каналы (FIFO)
- Сигналы реального времени
- **Разделяемые страницы памяти (mmap)**
- Сокеты — позволяют одному процессу общаться сразу со многими процессами

Каналы (pipe и FIFO)

- Пара файловых дескрипторов
- Гарантируется последовательность доставки данных

Чтение

```
char buf[BUF_SIZE];
ssize_t cnt;
while (
(cnt=read(fd,buf,sizeof(buf)) > 0)
{
   ....
}
```

Запись

```
char *data = ...
size_t size = ...

write(fd, data, size);
```

Обмен данными активно задействует ядро

```
ssize_t sendfile(int out_fd, int in_fd,
                 off_t *offset, size_t count)
```

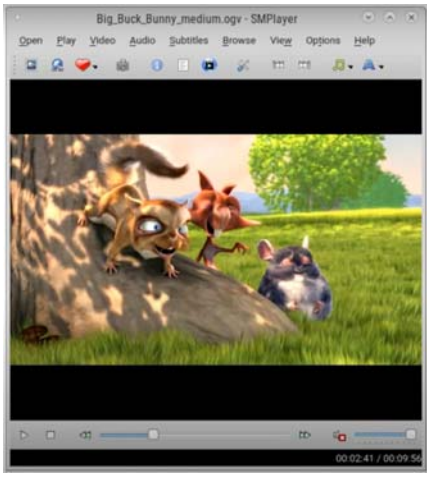
- Нестандартный системный вызов Linux и некоторых BSD
- Исключает многократное копирование данных между пространством пользователя и пространством ядра
- Не очень удобен, если данные генерируются on-the-fly

Задействованные процессы:

1. X-сервер
2. консольный плейер mplayer
3. GUI для mplayer

Взаимодействие:

1. Между mplayer и GUI — через каналы
2. Между mplayer и X — *разделяемая память*



Разделение памяти

- G (global) — страница глобальная
- D (dirty) — страница была модифицирована
- A (accessed) — был доступ к странице
- C (caching) — запрещено кэширование
- W (write-through) — разрешена сквозная запись
- U (userspace) — есть доступ обычному процессу
- R (read+write) — есть права на запись
- P (present) — страница находится в физической памяти

Разделение памяти

- Модуль MMU (Memory Management Unit) выполняет вычисления адресов по таблицам
- Кэш TLB (Translation Lookaside Buffer) хранит таблицы памяти для текущего процесса

Поддержка на уровне ОС

Unix System-V	BSD UNIX
SVR4 (1988) - Inter Process Communications: • Shared Mem • Semaphores • Message Queues	4.3BSD (1986) - реализация mmap POSIX Real-Time Extensions (1993-1995)

Разделяемая память (System-V style)

- `ftok` — связать “имя файла” с некоторым ключом key_t
- `int shmget(key_t, sizet_t, int flags)`
  создать или открыть сегмент разделяемой памяти
- `void* shmat(int id, void *addr, int flags)`
  присоединить сегмент в адресное пространство текущего процесса

Разделяемая память (System-V style)

- Используется отдельное пространство имен-ключей
- Ядро хранит отдельную таблицу сегментов
- Память доступна всем процессам по имени
- **Последний процесс должен удалить разделяемый сегмент**

Разделяемая память (BSD style)

```
void *mmap(void *addr, size_t length,
           int prot, int flags,
           int fd, off_t offset);
```

- `addr` — рекомендуемый адрес размещения
- `length` — размер отображаемой области
- `prot` — флаги защиты памяти (EXEC/READ/WRITE/NONE)
- `flags` — флаги доступа
  - **MAP_SHARED — совместный доступ разным процессам**,
  - **MAP_PRIVATE — использование Copy-On-Write**
- `fd`, `off_t` — файл для отображения (если отображение на файл)

Разделяемая память (BSD style)

- Параметр MAP_ANONYMUS
- Позволяет указать -1 в качестве файлового дескриптора отображаемого файла → данные не будут сохраняться в файл

Разделяемая память (BSD style, именование файлов)

- Создаем файл, доступный нескольким процессам
- Открываем его через mmap
- Не злоупотребляем msync
- PROFIT!
- кто-то должен потом файлик удалить
- нужно место на диске

Удаление файла

- Файл — это пара <dev_id, inode>
- У каждого файла есть атрибут nlink
- Операция “удаления” — это nlink—
- Файл становится недоступен при:
  - nlink==0
    &&
  - закрыты все файловые дескрипторы во всех процессах

Файловая система tmpfs

- Временное хранилище, все файлы будут удалены при выключении
- Имеет фиксированный размер, который можно легко изменить без потери данных
- Занимает в памяти столько, сколько данных записано
- Может задействовать swap, если память закончилась

```
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
```

POSIX shm

- `int shm_open(name, flags, mode)`
  аналогично open внутри `/dev/shm`
- Ограничения на имя:
  - начинается с “/” (иначе не будет работать в FreeBSD, но будет работать в MacOS X и Linux)
  - не может внутри содержать символ “/” (не будет работать в Linux, но будет работать в BSD-системах)

    - максимальная длина — `NAME_MAX` (ограничение Linux, но не *BSD)
- Режимы открытия:
    - `O_RDONLY`
    - `O_RDWR`
- Особенности реализации:
    - Linux: Функция glibc, а не системный вызов
    - FreeBSD и MacOS X: функция до FBSD<=7.0, потом syscall

Проблема гонки данных

1 процессор

- Переключение планировщиком задач активного потока выполнения

N процессоров

- Одновременное выполнение нескольких потоков выполнения

Гонка данных и shared mem

- Несколько процессов имеют одну область памяти в своем адресном пространстве
- Никто не знает, что делают другие
- Требуется побочный канал связи для синхронизации
- **Сигнал? Канал?**

Семафор

- Целочисленный счетчик (беззнаковый)
- Операции:
    - увеличить значение (освободить)
    - попытаться уменьшить значение (захватить); если значение счетчика уже равно 0, то ждать, пока кто-то другой его не освободит
- Операции инкремента/декремента — **атомарные**

Атомарность объекта

Необходимые условия:

1. Умещается в размер машинного слова
2. Выровнен по границе машинного слова

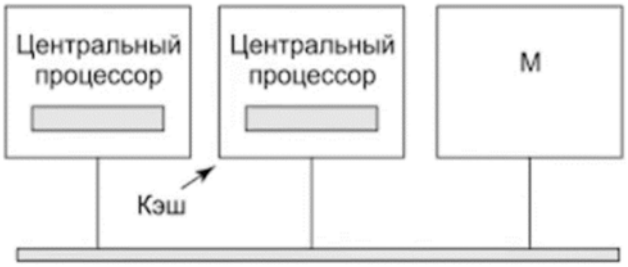
Пример: `sig_atomic_t`

3. **Должен быть одинаковым для всех ядер в любой момент времени**

Реализуется специальными атомарными операциями, поддерживаемыми как x86, так и ARM

Разделяемая память в UMA-системах [Uniform Memory Access]

- Все процессоры ядра используют общую шину памяти
- При одновременном доступе шина памяти блокируется → простой всех ядер
- Проблема решается наличием отдельного кэша для каждого ядра



Compare-And-Swap [cmpxchg для x86]

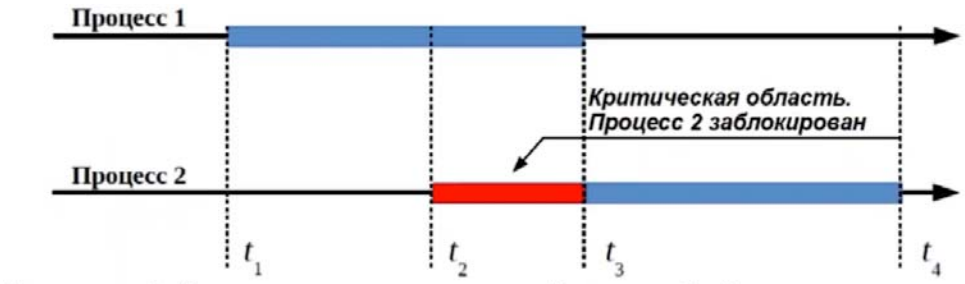
```
bool compare_and_swap(int* addr, int* old_val, int new_val)
        bool result = false;
        if (*addr == *old_val) {
                *addr = new_val;
                result = true;
        }
        else {
                *old_val = *addr;
        }
        return result;
}
```

Семафоры POSIX

```
#include <semaphore.h>
```

- Атомарные счетчики sem_t*, которые хранятся в специальных страницах памяти
- Реализованы операции:
  - `sem_post` — освободить семафор (+1)
  - `sem_wait` — захватить семафор (-1)
  - `sem_getvalue` — прочитать значение, не блокировать

Семафоры для синхронизации




```
Процесс 1 {
    . . . .
    . . . .
    P(S);      // t1
    . . . .
    V(S);      // t2
    . . . .
}
```

```
Процесс 1 {
    . . . .
    . . . .
    P(S);      // t1
    . . . .    // t3
    V(S);      // t4
    . . . .
}
```

- Proberen (дат.) - пробовать; пытаться; испытывать; подвергать испытанию; делать опыты; отведывать
- Verhogen (дат.) - возвышать; повышать; поднимать; увеличивать; усиливать; переводить в высший класс; увеличить

Семафоры POSIX

- Безымянные (для использования несколькими потоками внутри процесса или родственными процессами)

```
sem_init(sem_t *sem,
         /* 0 — в MAP_PRIVATE
            1 — в MAP_SHARED*/
         int pshared,
         uint32_t initial)
```

- Именованные (межпроцессное взаимодействие)

```
sem_open(const char *name,
         int oflag /* like open */,
         mode_t mode /* like open */,
         uint32_t initial);
```

Мьютексы

- Бинарный семафор с initial = 1
- Две операции: заблокировать и разблокировать
- **Ограничение 1**: только внутри одного адресного пространства
- **Ограничение 2**: кто захватил, тот и должен освободить

Mutex – Futex (Fast Userspace muTEX)

- Две операции: wait и wake
- Выполняются в пространстве ядра
- Захват не заблокированного мьютекса происходит без системного вызова

Условные переменные

- Специальный тип данных с двумя операциями: ждать событие и уведомить
- Работают в пределах одного адресного пространства

Lock-Free

- Специализированные структуры данных, не использующие блокировки
- Проще всего реализовать стек и очередь
- Базируются на операции Compare-And-Swap

Лекция 18 —

1. Динамическая загрузка библиотек
2. Компьютерные сети на низком уровне

Программа на низком уровне

foo.c

```
int var = 10;
void foo1 () {
}
void foo2 () {
  foo1();
}
```

bar.c

```
extern int var;
extern void foo2();
extern void bar () {
  int loc = var;
  foo2();
}
```

foo.o

```
0x0010: int var = 10
0x0014: void foo1 ()
. . . .
0x00A0: void foo2 () {
          call foo1 # 0x0014
0x00A2: ret }
```

bar.o

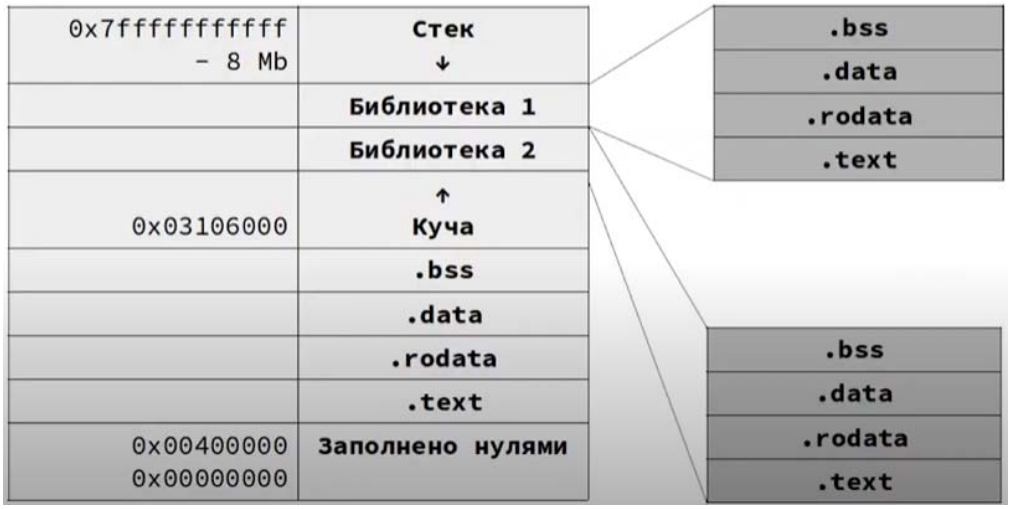
```
0x0000: void bar () {
          alloca loc
0x0002: loc = foo.var
0x0004: call foo.foo2
0x0006: ret }
```

foo_bar.elf

```
0x0010: int var = 10
0x0014: void foo1 ()
. . . .
0x00A0: void foo2 () {
          call foo1 # 0x0014
0x00A2: ret }
0x00A4: void bar () {
          alloca loc
0x00A6: loc = var # 0x0010
0x00A8: call foo2 # 0x00A0
0x00AA: ret }
```

- **foo_bar.elf** — это программа, если существует точка входа по определенному адресу
  Для Linux x86:
  - по умолчанию 0x804800
  - настраивается `-Wl, -Ttext-segment=ADDR`
- **foo_bar.elf** — это динамически загружаемая библиотека, если существует таблица символов
  `-Wl, --export-dynamic` или `-shared`

Размещение библиотек в памяти



`/lib[64]/ld-linux.so`

1. Разместить файл в памяти
2. Загрузить библиотеку из переменной окружения LD_PRELOAD
3. Найти все необходимые библиотеки, построить граф зависимостей
4. Загрузить все библиотеки в память
5. Создать таблицы GOT и PLT
6. Передать управление программе на точку входа (функция _start)

Опции gcc/clang

- **-fPIC** — скомпилировать исходный текст в позиционно-независимый код, который может быть использован в библиотеках

- **-fPIE** — аналогично **-fPIC**, но код также может быть использован для исполняемых файлов
- **-pie** — скомпоновать позиционно-независимый выполняемый файл

Position-Independent Executable

- Файл одновременно может быть использован как программа, так и библиотека
- Ядро ОС может размещать файл случайным образом (ASLR)

Address Space Layout Randomization

- OpenBSD — используется по умолчанию
- Linux поддерживается начиная с версии 2.6.12 — используется, если программа PIE
- В Ubuntu пакеты собираются с PIE по умолчанию

PIC/PIE в мирных целях

- Разделяемые библиотеки — больше возможностей по размещению в адресном пространстве
- Возможность подгружать библиотеки и выгружать их во время выполнения программы — **plugin**-ы

Что такое plugin

- Самодостаточный набор функций и классов
- Может быть отдельно протестирован
- Может быть реализован другой командой разработчиков или на другом языке программирования, либо вообще не иметь исходных текстов

Механизм dlopen/dlsym

```
void *dlopen(const char *lib_name, int flags)
HMODULE LoadLibraryA(LPCSTR lib_name, DWORD flags)
--- загрузить библиотеку

void *dlsym(void *lib, cons char *func_name)
FARPROC GetProcAddress(HMODULE lib, LPCSTR func_name)
--- найти функцию в библиотеке
```

```
#include <dlfcn.h>
void some_func() {
    void* lib =
       dlopen(
          "libSDL.so",
          RTLD_LAZY
       ); // Check for NULL!!!

    void* func_ptr =
       dlsym(lib, "SDL_Init");
          // Check for NULL!!!

    (*func_ptr)(); // Call
}
```

```
#include <Windows.h>
void some_func() {
    HMODULE lib =
       LoadLibraryA(
          "winhttp.dll"
       ); // Check for NULL!!!

    FARPROC func_ptr =
       GetProcAddress(
         lib, "WinHttpConnect"
       ); // Check for NULL!!!

    (*func_ptr)(....); // Call
}
```

В соглашении Linux резервируется так называемая нулевая страница памяти при запуске процессов, заполняется нулями и предназначена для того, чтобы не получить какой-то совершенно непредсказуемый мусор в том случае, если вы наступите на нулевой указатель, либо на что-то очень близкое к нулевому значению адреса.

Вызов функции по имени

```
def func():
    print("Hello, World!");

func_name = input("Please enter func name: ")

func_ref = globals()[func_name]

func_ref()  # Hello, World!
```

Вызов функции по имени

```
#include <dlfcn.h>

void callable() {}

void some_func() {
    void* lib =
       dlopen(NULL, 0);

    void* func_ptr =
       dlsym(lib, "callable");
          // Check for NULL!!!

    (*func_ptr)(); // Call
}
```

```
#include <Windows.h>

__declspec(dllexport)
void callable() {}

void some_func() {
    HMODULE lib =
       LoadLibrary(NULL);

    FARPROC func_ptr =
       GetProcAddress(
         lib, "callable"
       ); // Check for NULL!!!

    (*func_ptr)(); // Call
}
```

https://docs.python.org/3/library/ctypes.html

- общая идея — использовать dlopen/LoadLibrary для загрузки произвольных библиотек во время выполнения
- доступ к функциям по Си-имени
- сигнатуры НЕ контролируются

C++

- Пространства имен
- Классы
- Перезагрузка функций

```
namespace some {
  namespace package {
    class ClassInPackage {
        ClassInPackage() {
          // constructor
        }
    };
  }
}
```

```
> objdump -t module.o
...
_ZN4some7package14ClassInPackageC1Ev
...
```

Microsoft Visual C++

- Совместим с WinGW только для Си-имен
- Для классов дополнительно требуется _declspec(dllexport)

```
namespace some {
  namespace package {
    class ClassInPackage {
        ClassInPackage() {
          // constructor
        }
    };
  }
}
```

```
??0ClassInPackageInterface@@QEAA@XZ
```

C++ в динамически подгружаемых библиотеках

- Для использования экземпляров класса линковщику требуется все реализованные методы класса
- Проблема может быть решена использованием *чистых виртуальных интерфейсов*

```
class InterfaceOne {
  // no fields!
  // no regular methods!
public:
  virtual void a() = 0;
  virtual void b() = 0;
};

class InterfaceTwo {
public:
  virtual void b() = 0;
  virtual void c() = 0;
};
```

```
class SomePlugin
  : public InterfaceOne
  , public InterfaceTwo
{
public:
  SomePlugin() {}
  void a() {}
  void b() {}
  void c() {}
private:
  // might be fields here
};
```

ClassLoader: C++ vs Java

- Сигнатуры методов

- Информация о наследовании
- Решается сторонними инструментами — QPluginLoader из QtCore [www.qt.io]

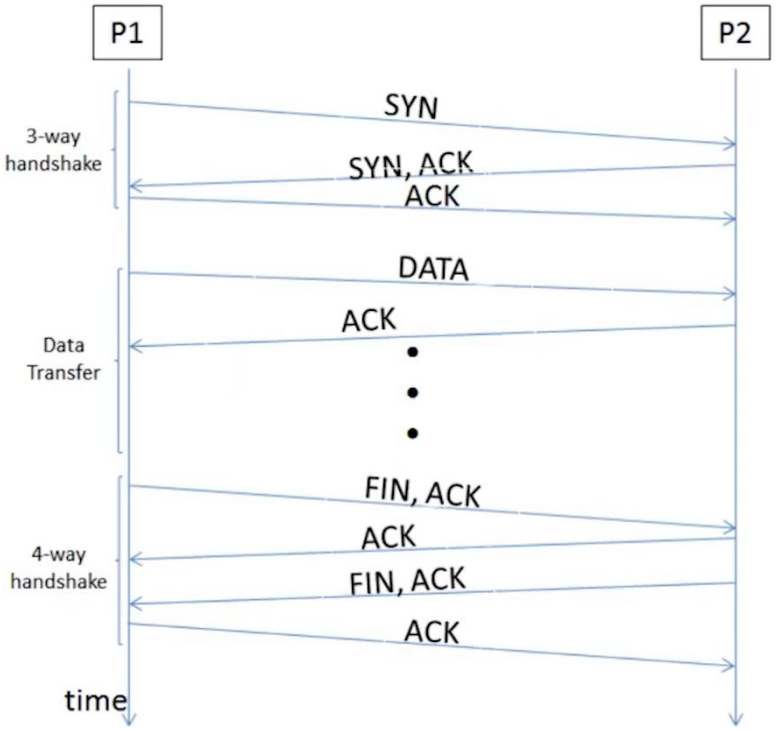
Еще раз про сети: немного на более низком уровне

Сетевое взаимодействие

Стек TCP/IP		Модель OSI (Open Systems Interconnections)	Примеры
Уровень процессов		Уровень приложений (Application)	HTTP, FTP, SSH, Telnet
		Уровень представления (Presentation)	ASCII, GZIP, binary
		Уровень сеанса (Session)	NetBIOS, SSL
Транспортный уровень		Уровень транспорта (Transport)	TCP, UDP
Уровень Internet		Уровень сети (Network)	IPv4, IPv6, IPX, AppleTalk
Уровень сетевого интерфейса		Уровень канала (Data Link)	PPP, IEEE 802.2 (Ethernet)
		Физический уровень (Physical)	USB, IEEE 802.11 IEEE 802.3 (Ethernet)

Взаимодействие по TCP

1. Установка соединения (флаг SYN)
2. Подтверждение установки соединения (SYN+ACK)
3. Двусторонний обмен данными — с флагом ACK кроме полезной нагрузки указывается номер подтверждаемого пакета
4. Закрытие соединения — пакет с флагом FIN



Взаимодействие по TCP

- Все пакеты имеют порядковый номер, который присваивает ядро ОС
- Задача ядра — сделать видимость непрерывного потока данных в обе стороны
- Ввод-вывод — как с обычными каналами read/write
- `socket( AF_..., SOCK_STREAM, 0 )`

SOCK_STREAM: кроме TCP

- Протокол Novell SPX (historic)
- Локальное взаимодействие AF_UNIX

```
socket( AF_... , SOCK_STREAM, 0 )

последний параметр - номер протокола

0 - автоматический выбор для пары AF_...,
                SOCK_...
```

Поточное взаимодействие: HTTP over TCP (порт 80)

Запрос

```
GET /index.html HTTP/1.1
Host: www.example.com
Connection: keep-alive
DNT: 1
User-Agent: Mozilla/5.0 . . .
Accept-Encoding: gzip, deflate
Accept-Language: ru, en
```

Ответ

```
HTTP/1.1 200 OK
Server: Apache
Content-Type: text/html
Transfer-Encoding: chunked
Date: Mon, 27 Apr 2015 13:40:00

<html>
  <head></head>
  <body><p>It works!</p></body>
</html>
```

Инструменты для взаимодействия

- telnet — терминал для текстового ввода-вывода через сокеты
- netcat или nc — аналог cat, но работает с сетевыми соединениями

Шифрование

- Уровень сеанса в модели OSI
- В модели TCP — это уровень процессов
- Поток данных передается по TCP, его интерпретация — задача процесса в пространстве пользователя

```
> openssl s_client -connect www.yandex.ru:443
```

Типы сокетов

- SOCK_STREAM — двунаправленный поток данных
- **SOCK_DGRAM** — односторонние сообщения (UDP)
- SOCK_RAW — пакеты IP (уровень сети)
- SOCK_PACKET (Linux) — фреймы Ethernet (уровень канала)

Пакеты UDP

- Односторонние сообщения

- Максимальная длина:
  MTU – sizeof(ip_header) – sizeof(udp_header)
- Процесс-получатель сам обязан формировать ответ

Байты	0	1	2	3
0..4	Порт отправителя		Порт назначения	
5..8	Длина пакета		Контрольная сумма	

UDP vs TCP

- Не требуется установление соединения (SYN/SYN+ACK) и его завершение (FIN/FIN+ACK)
- Не контролируется порядок пакетов
- Не требуется отправка пакетов в подтверждение получения

Сценарии использования:

1. Торренты
2. Эмуляция уровня канала (VPN)

Протоколы HTTP

- HTTP/0.9 и HTTP/1.0 (1991..1996)
  - исторические реализации, сейчас почти никем не поддерживаются
- HTTP/1.1 (1999)
  - Текстовый формат взаимодействия
  - Обязателен заголовок Host
  - Можно не закрывать соединение после выполнения запроса (параметр Connection:close vs Connection:keep-alive)
- HTTP/2 (2015)
  - Бинарное, а не текстовое взаимодействие
  - Передача нескольких файлов одновременно по одному соединению
- HTTP/3 (Draft, поддерживается dev-версиями браузеров)
  - **Использование протокола UDP (прослойка Google QUIC) вместо TCP**

Канальный уровень сети

- Адресация — по MAC-адресам
- На физическом уровне — пассивные (хабы) или активные (свитчи) коммутаторы
- MAC-адрес связан с конкретным устройством и не меняется (в теории, но не на практике) при конфигурации

Канальный уровень сети

- Привязка к MAC-адресу устройства
- Достаточно знать только своих соседей
- Выделяется отдельное устройство — маршрутизатор для выхода во внешнюю сеть

- Соответствие между IP-адресами и MAC-адресами через протокол ARP

Как узнать MAC-адрес

- Формируется Ethernet-кадр, содержимое которого не IP-пакет, а ARP-запрос с требуемым адресом
- MAC-отправителя — того, кто отправляет
- MAC-получателя — широковещательный FF:FF:FF:FF:FF:FF
- Хост с подходящим IP отправляет ARP-ответ
- Инициирующая сторона кэширует ответ (время жизни — 30 секунд для Linux)

```
/usr/sbin/arp -a # отобразить таблицу arp
```

MAC-адрес получателя	MAC-адрес отправителя	Дополнительные опции	Длина	Данные	Контрольная сумма
6 байт	6 байт	4 байта	2 байта	от 46 до 1500 байт (параметр MTU)	4 байта

Сетевой уровень IPv4

- У каждого хоста есть IP-адрес (для IPv4 не гарантируется уникальность)
- “Серые” адреса не доступны из сети Интернет (например 192.168.x.y)
- Для доставки IP-пакета может потребоваться цепочка маршрутизаторов

```
   /bin/route # (в POSIX-системах)
     /usr/sbin/ip r # (в Linux)
/usr/sbin/netstat -r # (FreeBSD/MacOS)
```

TCP-пакеты могут приходить в разном порядке, поскольку какие-то пакеты определенные участки сети могут проходить по разным маршрутам для балансировки нагрузки.

Назначение IP-адресов

- Статическое — явное указание параметров сетевого интерфейса
- Динамическое (протокол DHCP)

- **DHCPDISCOVER** от 0.0.0.0:68 к 255.255.255.255:67
  - запросить предложения от DHCP-серверов в локальной сети
- **DHCPOFFER** от одного из серверов конкретному MAC
  - предложение свободного адреса
- **DHCPREQUEST** к конкретному серверу
  - запрос получения ранее предложенного адреса
- **DHCPACK** от сервера
  - подтверждение выдачи адреса

.....

- **DCHCPRELEASE** серверу
  - освобождение (перед выключением)

Реализация DHCP

- UDP пакеты поверх IP
- Сервер — на порту 67
- В сообщении OFFER кроме IP-адреса содержится дополнительная информация:

- адрес маршрутизатора
- адреса DNS-серверов

```
/sbin/dhclient
```

DNS

- IP адрес определяет маршрут до хоста
- Доменное имя — должно быть запоминающимся, например: `скатать-акос-без-регистрации-и-смс.рф`
- Служба DNS — иерархическая система каталогов соответствия имен и IP-адресов
- Сервер DNS — работает поверх UDP и TCP на 53 порту; на маршрутизаторах — кэширование
- Можно отправлять запросы любым серверам (Google: 8.8.8.8)

```
/usr/bin/nslookup # (простая команда)
  /usr/bin/dig # (более продвинутая)
```

Как работает DNS? DNS формирует некоторый специальный запрос, дальше отправляется обычный UDP пакет с запросом на определенный сервер (например гугловский, либо сервер вашего провайдера) на порт 53, дальше получаете ответ, проверяете сопоставление того, что ID запроса с ответом совпадают, ну и извлекаете из него ответ и дальше как-то его используете.

Доменные имена

- Полная форма заканчивается символом точки (но для упрощения это часто не требуется от пользователя)
- Несколько типов:
  - **A** — IPv4 адрес хоста для соединения
  - **AAAA** — IPv6 адрес хоста для соединения
  - **MX** — имя хоста для электронной почты
  - **CNAME** — имя хоста для соединения

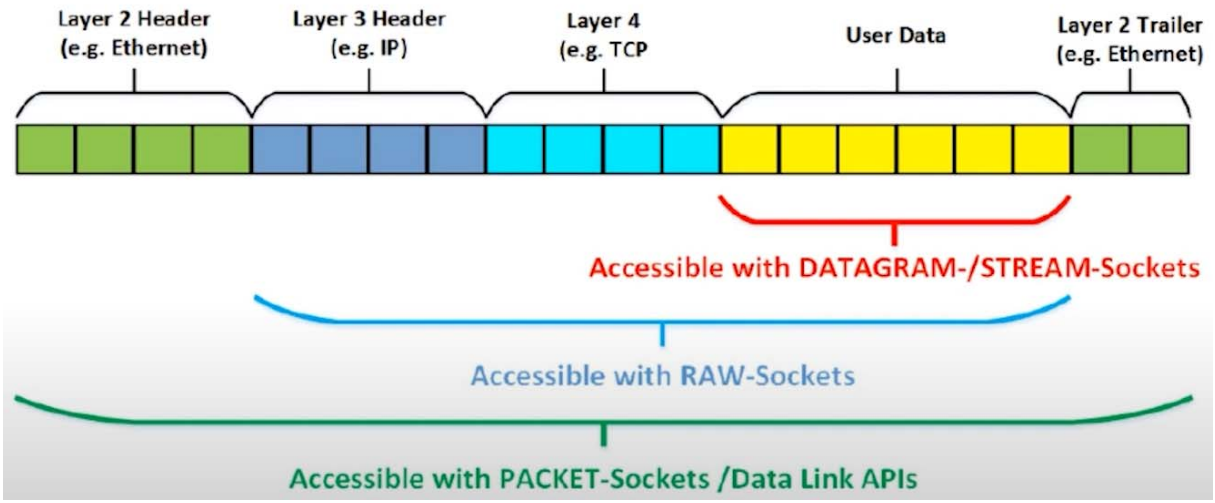
Типы сокетов

- SOCK_STREAM — двунаправленный поток данных
- SOCK_DGRAM — односторонние сообщения (UDP)
- **SOCK_RAW** — пакеты IP (уровень сети)
- **SOCK_PACKET** (Linux) — фреймы Ethernet (уровень канала)

RAW-сокеты позволяют передавать неструктурированные данные.

*Картинка из статьи **Introduction to RAW-sockets***

Jens Heuschke, Tobias Hoffmar et al. Technische Universitat Darmstadt. 2017l



Работа с сокетами

- `int fd = socket(…)` — создать сокет
- `connect(fd, …)` —подключиться
- `send/recv` — обмен данными

Стадия **connect** настраивает заголовки по умолчанию, связанные с *конкретным типом* сокета.

RAW-сокеты

```
sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)
```

- Работают только из-под рута или настроенным CAP_NET_RAW
- Можно настроить, чтобы добавлялись IP-заголовки (setsockopt IP_HDRINCL)

Сокеты канального уровня

```
sockfd = socket(AF_PACKET, SOCK_RAW,
                htons(ETH_P_ALL) // фильтр
                )
```

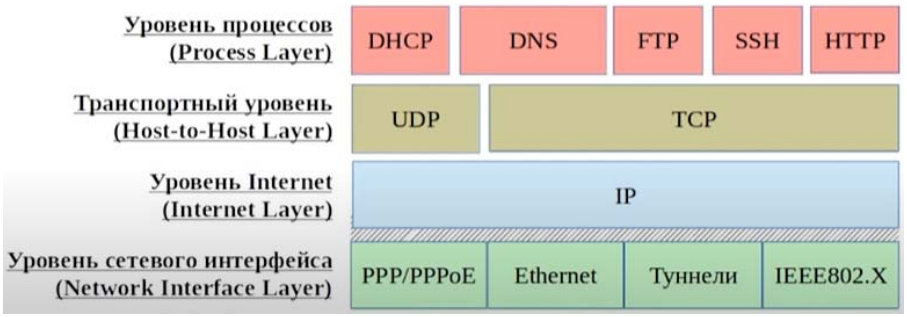
Низкоуровневые операции

- Протоколирование действий
- Фильтрация траффика
- Реализация нестандартного протокола передачи данных

Как вообще реализованы в современных UNIX-системах отдельные протоколы? Это обычные модули ядра, которые отвечают за определенный сетевой уровень и выполняют кодирование данных.

Лекция 19

Иерархия TCP/IP



Модель OSI

Стек TCP/IP		Модель OSI (Open Systems Interconnections)	Примеры
Уровень процессов		Уровень приложений (Application)	HTTP, FTP, SSH, Telnet
		Уровень представления (Presentation)	ASCII, GZIP, binary
		Уровень сеанса (Session)	NetBIOS, SSL

HTTP: уровень приложений

[выделяется подуровень представлений]

- Request: Accept-Encoding: gzip, deflate
- Response: Content-Encoding: gzip

Уровень сеанса

- До начала взаимодействия выполняются предварительные действия по согласованию сеанса
- После согласования сеанса, передаваемые данные могут отличаться от того, что видно на уровне приложений
- Пример: HTTPS соединение
  `> openssl s_client -connect www.yandex.ru:443`

Transport Layer Security

- До 1999 года стандартом *де факто* был Secure Socket Layer (**SSL**) — Netscape
- TLS — стандартизирован Internet Engineering Task Force (**IETF**) как Request for Comments (**RFC**):
  - TLS 1.0 — 1999 год
  - TLS 1.1 — 2006 год
  - TLS 1.2 — 2008 год
  - TLS 1.3 — 2018 год, поддерживается OpenSSL начиная с 1.1.1

Transport Layer Security

- Договоренность об отдельном номере порта для TLS-сессий, например 443 вместо 80

или

- Переключение в режим TLS после установки соединения, например команда STARTTLS в почтовых серверах

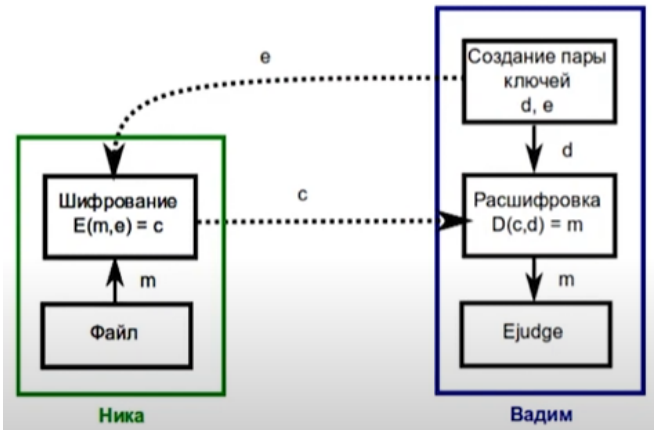
Зачем

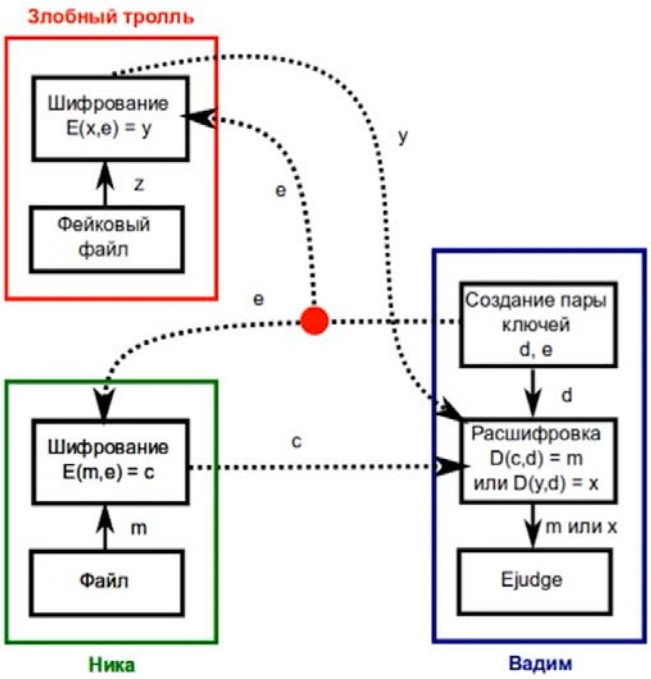
- Проверка подлинности веб-ресурса для исключения атаки MITM
- Шифрование данных
- Кому-то: для зарабатывания денег из воздуха*

Криптографические алгоритмы

- Хеш-функции: CRC32, MD5, SHA
  - по блоку исходных данных строится некоторое скалярное значение фиксированного размера
  - операция хеширования не обратима
  - пример использования: хранение паролей
- С симметричным ключом:
  DES (1977), AES (1998), ГОСТ 28147-89, Кузнечик (ГОСТ 34.12-2015)
  - исходные данные разбиваются на блоки фиксированного размера
  - к каждому блоку применяется алгоритм с использованием ключа того же, либо кратному ему размера
  - операция обратима при использовании того же самого ключа
  - пример использования: шифрование данных на диске
- С асимметричной парой ключей:
  RSA (1977), ГОСТ Р 34.10-2018 (1994)
  - генерируется пара ключей: открытый для шифрования и закрытый для дешифрования
  - зная открытый ключ нельзя восстановить закрытый
  - примеры использования: TLS и SSH; цифровая подпись с использованием третей стороны (сертификата)

Взаимодействие





Сертификат открытого ключа

- Содержит
  - открытый ключ
  - срок действия
  - метаданные
  - цифровую подпись сертификата:
    хэш сертификата, зашифрованный закрытым ключом
- Генерируется третей стороной — удостоверяющим центром

Сертификат открытого ключа

- Удостоверяющий центр — одна из организаций: Symantec, Thawte, StartSSL, Let's Encrypt генерирует пару ключей <**a**, **b**>
- Ключ **a** доступен всем: Нике и Вадиму
- Ника отправляет свой ключ **e** в удостоверяющий центр, его хэш шифруется ключом **b**, получаем сертификат **s**
- На стадии согласования ключей Ника отправляет Вадиму сертификат **s**, он расшифровывается ключом **a**, получается хэш, который сравнивается с хэшем ключа **e**
- Если хэши совпадают — OK; если нет, то есть риск спалиться

Взаимодействие по TLS

- Сервер первым отправляет свой сертификат, содержащий открытый ключ
- У клиента есть база открытых ключей CA, одним из которых подписан сертификат сервера; выполняется проверка подлинности
- Клиент генерирует пару ключей и отправляет свой открытый ключ, предварительно зашифровав его открытым ключом сервера

Корневые сертификаты

- Покупаются у сторонней организации
- Иногда бывают утечки, которые сопровождаются отзывами сертификатов
- Могут быть корпоративными:
    - устанавливаются дополнительно в браузеры или системы клиентов
    - предназначены для подписи сертификатов отдельных серверов инфраструктуры

Реализация шифрования

- Стандарт де-факто: библиотека libcrypto из фреймворка OpenSSL
- В 2014 году была обнаружена уязвимость Heartbleed
- После обнаружения — форк LibreSSL [https://www.libressl.org] от проекта OpenBSD, совместимый по API

LibreSSL

- Библиотека libcrypto
  реализация алгоритмов шифрования
- Библиотека libssl
  реализация SSL/TLS, совместимая с OpenSSL по API
- Библиотека libtls
  реализация упрощенного API для работы с TLS

OpenSSL/LibreSSL

Команда openssl — куча режимов работы

- openssl enc
  (де)шифрование симметричным ключом
- openssl md5/sha256/sha512/…
  вычисление хэш значения
- openssl genrsa
  создание пары асимметричных ключей
- openssl s_client
  сеанс TLS с удаленным сервером

Как сделать SSL на сервере

Классика:

- Сгенерировать пару ключей по RSA
  openssl genrsa
- Сформировать запрос на подпись сертификата
  openssl req
- Заплатить денег порядка 22Круб/год
- Получить подписанный файл сертификата, и положить на сервере
- Положить на сервере закрытый ключ в **безопасное** место

Упрощенный подход: https://letsencrypt.org/getting-started/

Лекция 20 — Kernel Modules. Filesystem in USEr Space

The Kernel

- The first program to be launched after the bootloader
- Runs at privileged processor level
- Has access to everything

The Kernel

- Monolithic — placed at some continuous physical address and implements all the functionality (old UNIXs)
- Microkernel — small program to coordinate user-space services (MINIX3, QNX)
- Hybrid — split into kernel program and loadable modules (most modern OSes)

Kernel Modules

- EIF-files located at /lib/modules/*/*.ko
- Might be loaded by root using insmod

- Modules might have dependencies
- modprobe tool loads all dependent modules

Kernel Modules

- Minimum API requirements:
    - int init_module(); //returns 0 on success
    - void clean_module();

Kernel Taint

- Loading of non GPL-compatible modules means the Kernel is unsecure
- The <tainted> state remains until reboot
  `/proc/sys/kernel/tainted`
- MODULE_LICENSE(…) macro prevents this

Kernel Modules Parameters

- Macro module_param(VAR, TYPE, ACCESS)
- Accessible via /sys/modules/*/parameters
- /etc/modprobe.d/*.conf allows to pass parameters at boot stage

Kernel Symbol Table

- No standard C library
- All functions are implemented by kernel itself or by its modules
- Global Symbols visibility /proc/kallsyms
- Might be hidden by <static> modifier

Kernel Routines

- Memory Management
    - routines like kmalloc/kfree
    - no <heap> for kernel!
    - memory might be allocated either for Kernel space or User space
- Basic string operations
- String conversions like strtol and sprint

https://www.kernel.org/doc/htmldocs/kernel-api/index.html

System Calls

- Not required to use any special instructions within Kernel space
- Until Linux 2.6:
  - extern void *sys_call_table[]
  - Might be replaced by ANY kernel module!
- Since Linux 2.6:
  - wrapper functions sys_SYSCALL
  - *sys_call_table might be accessible by kallsyms_lookup_name (GPL modules only, might be disabled by kernel configuration)

/boot/System.map

- Logging purposes: syslogd/klogd
- Might be used by some 3rd-party-kernel modules for build

- Other files at /boot:
  - vmlinux.gz — compressed Kernel ELF image
  - vmlinuz — compressed Kernel EFI MZ-image
  - config — kernel build configuration
  - initrd — initial minimalistic root FS image

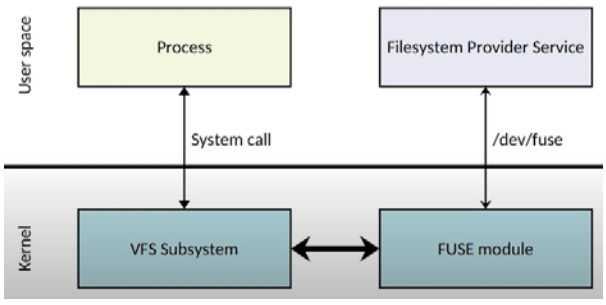
Kernel Modules Purposes

- Implement device access modules
  - access via PCI interface
  - access via USB interface
- Implement file systems
  - ext[2,3,4] is not a module, but compiled into image
  - Linux has support for many additional and <foreign> file systems

File System Implementations

- In-Kernel
  - pros: is fast
  - cons: hard to develop, licensing problems (ZFS)
- In user space
  - much slower
  - might use 3rd party libraries
  - no licensing limitations

FUSE



Что такое псевдоустройство /dev/fuse? Это специальный символьный файл, т.е. здесь ввод-вывод осуществляется последовательно. При монтировании файловой системы происходит обращение к /dev/fuse, где мы указываем свой процесс ID и сообщаем о том, что мы готовы слушать такой там точка монтирования и обрабатывать с нее запросы. Дальше, когда происходит обращение к нашей файловой системе, то происходит обращение через файловый дескриптор, но уже не через /dev/fuse, а через файловый дескриптор, который вы передали при монтировании файловой системы, ядро вам записывает какие-то события и ваш пользовательский процесс ssh демон либо sshfs демон либо еще какой-то пользовательский процесс, который обрабатывает виртуальную файловую систему просто получает из этого файлового дескриптора набор команд и дальше его обрабатывает.

/dev/fuse

- Character device for Linux implementation
- Userland process reads requests and sends responses
- Not a socket but several processes might use in a way of sockets I/O:
    - open(«/dev/fuse») → FD
    - mount(…, pass FD as option)
- Non-root processes use helper fusermount:
    - has capability to mount
    - interacts daemon via socketpair

High-Level API

- libfuse (prior to 3.0)
- libfuse3 (since 3.0)
  https://github.com/libfuse/libfuse

Custom FS architecture:

- parse command line options and initialize
- register callback functions
- launch fuse main loop

File Systems Implemented with FUSE

- sshfs — file transfer via protocol SSH
- unionfs — make union of several directories
- fusesmb — access Windows shares
- encfs — encrypt data
- ntfs-3g — access Windows NTFS filesystem

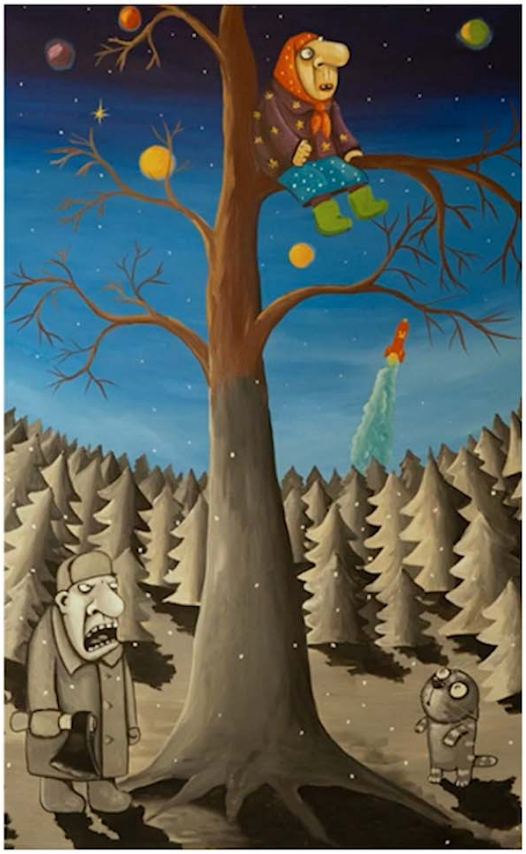
Not only C/C++

- fusepy (python 3.x)
- python-fuse (python 2.x)

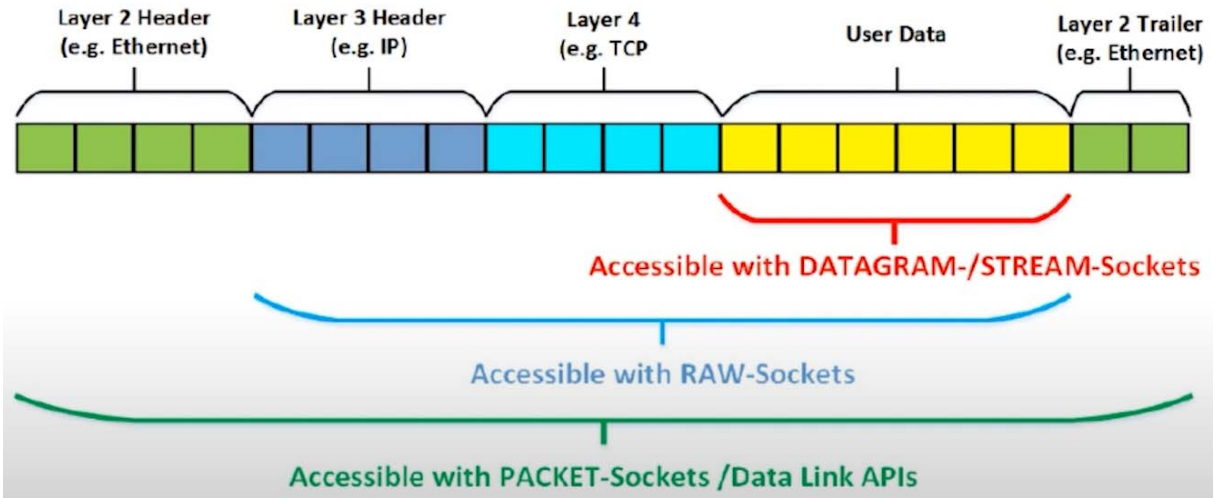
Usages:

- GMailFS [outdated] — access GMail
- YouTubeFS [outdated] — access YouTube

====== Summary Matched: fuse =======
simple-mtpfs.x86_64 : Fuse-based MTP driver
lxcfs.x86_64 : FUSE based filesystem for LXC
bindfs.x86_64 : Fuse filesystem to mirror a directory
enblend-doc.x86_64 : Usage Documentation for enblend and enfuse
mp3fs.x86_64 : FUSE filesystem to transcode FLAC to MP3 on the fly
archivemount.x86_64 : FUSE based filesystem for mounting compressed archives
jmtpfs.x86_64 : FUSE and libmtp based filesystem for accessing MTP devices
gphotofs.x86_64 : A FUSE filesystem module to mount your camera as a filesystem
golang-github-google-slothfs.x86_64 : FUSE filesystem for light-weight, lazily-loaded, read-only Git
curlftpfs.x86_64 : CurlFtpFS is a filesystem for accessing FTP hosts based on FUSE and libcurl



Лекция 21 — Berkley Packet Filter



RAW Sockets

IPv4 but not TCP or UDP:

```
sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)
```

- Might be created by root or using `CAP_NET_RAW` capability
- IP-level header might be filled out by Kernel (setsockopt `IP_HDRINCL`)

Everything over the wires or air:

```
sockfd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL))
```

- Linux-only (AF_PACKET address family)
- Protocol type is a ‘filter’ to catch various Ethernet frame types

Packet-Level Socket Blinding

Regular TCP/UDP+IPv4 address:

```
struct sockaddr_in {
  // AF_INET
  sa_family_t    sin_family;

  // 16 bit port
  in_port_t      sin_port;

  // 32 bit IP
  struct in_addr sin_addr;

};
```

Packet-Level hardware address:

```
struct sockaddr_ll {
  // AF_PACKET
  uint16_t sll_family;

  // Ethernet proto
  uint16_t sll_protocol;

  // If index
  int      sll_ifindex;

  /*
  some extra
  unimportant fields
  */
};
```

ifconfig, ip a l, man 7 netdevice

Packet Handling

- tcpdump utility
- dumpcap (Wireshark)

TCP Dump

```
tcpdump -I INTERFACE “FILTERS”
```

[Classic] Berkeley Packet Filter

```
# tcpdump -d "udp and ip"
(000) ldh      [12]                    ; 12-th byte
(001) jeq      #0x86dd  jt 6  jf 2 ; 0x86dd = IPv6
(002) jeq      #0x800   jt 3  jf 6 ; 0x0800 = IPv4
(003) ldb      [23]                    ; 23-th byte
(004) jeq      #0x11    jt 5  jf 6 ; 0x11 = 17 = UDP
(005) ret      #262144                 ; Very big value
(006) ret      #0                      ; Empty packet
```

man 8 bpf

linux-sources/Documentation/networking/filter.txt

[Classic] Berkeley Packet Filter

- 64-bit RISC like instructions

```
struct sock_filter {
```

```
    __u16 code; // op code
    __u8  jt;   // true-condition offset
    __u8  jf;   // false-condition offset
    __u32 k;    // constant values
};
```

- Inspired by Motorola 6502 ISA
- Interpreted by Virtual Machine

[Classic] Berkeley Packet Filter

Example: filter DNS to Google

```
filter_google_dns:
    ldh     [12]            ; 16 bit Eth proto value after MACs
    jne     #0x0800, fail   ; 0x0800 = IPv4
    ldb     [23]            ; IP header one byte proto number
    jne     #17, fail       ; 17 = UDP, 6 = TCP
    ld      [30]            ; 32 bit IPv4 address value
    jne     #0x08080808, fail   ; 8.8.8.8
success:
    ret     #-1             ; -1 == 0xFFFFFFFF (maximum size)
fail:
    ret     #0              ; 0 is empty
```

[Classic] BPF Code

- Runs in Kernel mode
- Limitations:
    - not more than 4096 instructions
    - prohibited backward jumps (not loops)
- Code loading – verification – attach
- Error ‘invalid argument’ on ‘setsockopt’

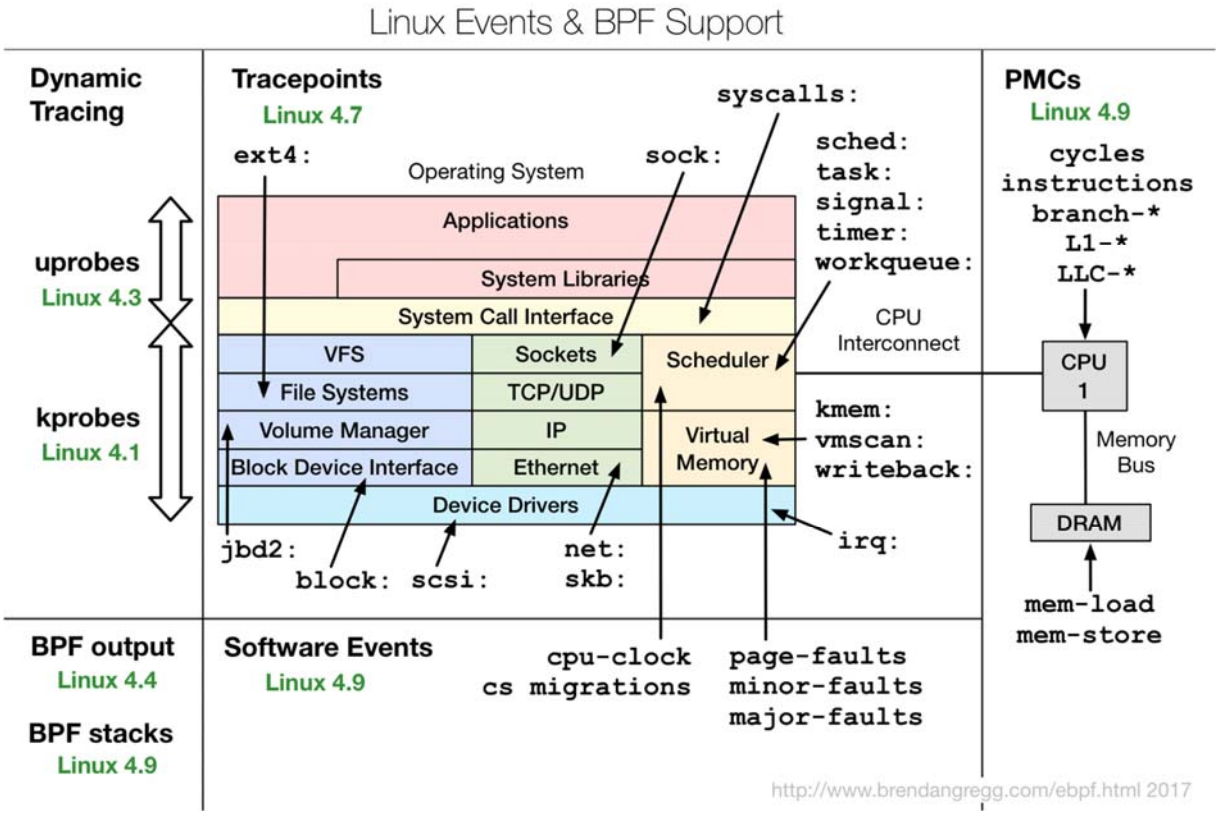
Extended BPF

Linux Kernels from 2016

- JIT compilation for x86_64
- More complicated API
- High-level language support (based on LLVM)

eBPF Features

- In-Kernel code
- Interation to/from userland via **maps**
- Features
    - Packet filtering
    - Packet preprocessing
    - Kernel-level tracing
    - Kernel-level monitoring



Resources:

- eBPF Introduction, Tutorials & Community Resources
https://ebpf.io/

- BPF Compiler Collection
https://github.com/iovisor/bcc

- Linux eBPF Tracing Tools
http://www.brendangregg.com/ebpf.html