

# Лекции по Операционным системам

Сверстал: Кузякин Никита Александрович

По лекциям ИТМО

Плейлист с лекциями — [тут](#)

## СОДЕРЖАНИЕ

|  |          |
|--|----------|
| <b>I   Основы архитектуры ПК и операционных систем</b>     | <b>3</b> |
| 1   Архитектура компьютерных систем .....                  | 4        |
| 2   Обзор элементов компьютерных систем .....              | 7        |
| 2.1   Процессор .....                                      | 7        |
| 3   Общие сведения об операционных системах .....          | 9        |
| 3.1   Функции OS.....                                      | 9        |
| 3.2   Оператор ЭВМ .....                                   | 9        |
| 3.3   Пакетная обработка .....                             | 9        |
| 3.4   Многозадачность .....                                | 9        |
| 3.5   Разделение времени .....                             | 10       |
| 4   Основные задачи OS .....                               | 11       |
| 4.1   Управление процессами.....                           | 11       |
| 4.2   Виртуальная память .....                             | 12       |
| 4.3   Безопасность .....                                   | 12       |
| 4.4   Диспетчеризация и планирование ресурсов .....        | 13       |
| 5   Современные архитектурные концепции OS .....           | 14       |
| 5.1   Архитектура ядер .....                               | 14       |
| 5.2   Многопоточность .....                                | 15       |
| 5.3   SMP и ASMP .....                                     | 15       |
| 5.4   Виртуализация .....                                  | 16       |
| 6   Основные понятия надежности операционной системы ..... | 17       |
| 6.1   Надежность и отказоустойчивость .....                | 17       |
| 6.2   Сбои .....   | 17       |
| 7   Общая архитектура UNIX / Linux.....                    | 19       |
| 8   Общая архитектура Windows .....                        | 20       |
| 9   Средства для отладки Linux .....                       | 22       |
| 9.1   Стандартные средства.....                            | 22       |
| 9.2   /proc .....  | 23       |
| 9.3   Трассировщики .....                                  | 23       |
| 9.4   perf .....   | 23       |
| 9.5   System tap .....                                     | 25       |

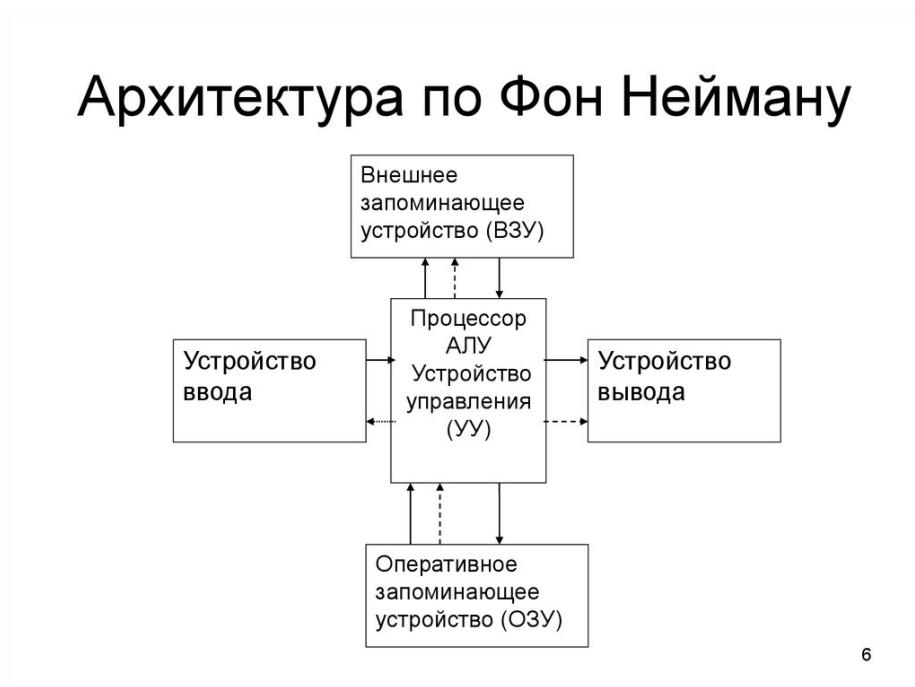
|  |           |
|--|-----------|
| 9.6 Kernel debugger .....                            | 25        |
| 10 Средства для отладки Windows .....                | 27        |
| 10.1 Встроенные средства .....                       | 27        |
| 10.2 SysInternals.....                               | 27        |
| 10.3 Отладчик ядра WinDbg и KD .....                 | 28        |
| <b>II Процессы</b>                                   | <b>28</b> |
| 11 Основы процессов .....                            | 28        |
| 11.1 Вычисления .....                                | 28        |
| 11.2 Процесс. Характеристики процесса .....          | 28        |
| 11.3 Состояние процессов и разделение ресурсов ..... | 29        |
| 12 Пейджинг и свопинг .....                          | 33        |
| 12.1 Paging Swapping.....                            | 33        |
| 12.2 Дополнительные состояния процессора .....       | 34        |
| 13 Управление процессами .....                       | 35        |
| 13.1 Управляющие таблицы .....                       | 35        |
| 13.2 Образ процесса .....                            | 35        |
| 13.3 Функции OS.....                                 | 37        |
| 13.4 Процессы SVR4 .....                             | 38        |
| 14 Потоки .....                                      | 39        |
| 14.1 Понятие потока .....                            | 39        |
| 14.2 Связь потоков и процессов .....                 | 39        |
| 14.3 Состояния потока .....                          | 40        |
| 14.4 Варианты реализации .....                       | 40        |
| 14.5 Закон Амдала .....                              | 41        |
| 15 Полезные утилиты .....                            | 42        |

## Часть I

# Основы архитектуры ПК и операционных систем

## 1 Архитектура компьютерных систем

Первоначальными двумя архитектурами компьютерных систем являются Гарвардская и Неймановская архитектуры.



6

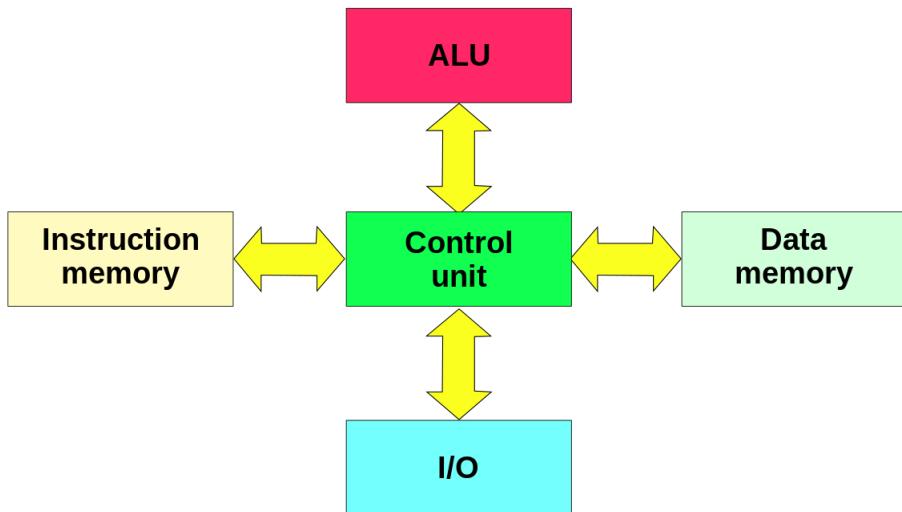


Рисунок 1 – Гарвардская архитектура ЭВМ

Любая вычислительная машины состоит из управляющего устройства (организует вычисления) и арифметико - логического устройства (производит вычисление арифметических операций), а также различных видов памяти.

В архитектуре фон Неймана предполагается, что есть единое управляющее устройство, память при этом общая (и данная, и программа в одно блоке).

#### **Принципы архитектуры фон Неймана:**

- Принцип однородности памяти — команды и данные хранятся в одной и той же памяти (внешне неразличимы).
- Принцип адресности — память состоит из пронумерованных ячеек, процессору доступна любая ячейка.
- Принцип программного управления — вычисления представлены в виде программы, состоящей из последовательности команд.
- Принцип двоичного кодирования — вся информация, как данные, так и команды, кодируются двоичными цифрами 0 и 1.

#### **UMA / NUMA**

В архитектуре UUMA подразумевается, что все устройства являются одноранговыми. Т.е у любого устройства в системе равные права на доступ к памяти и системные характеристики обращения к ней.

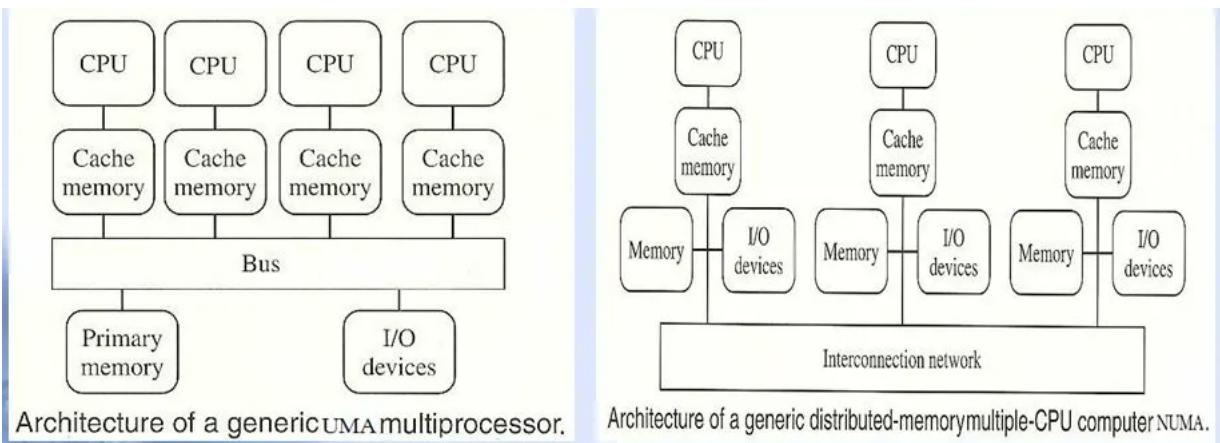


Рисунок 2 – Гарвардская архитектура ЭВМ

Минусом данной архитектуры является, то что тяжело организовать доступ к памяти для большого числа процессоров.

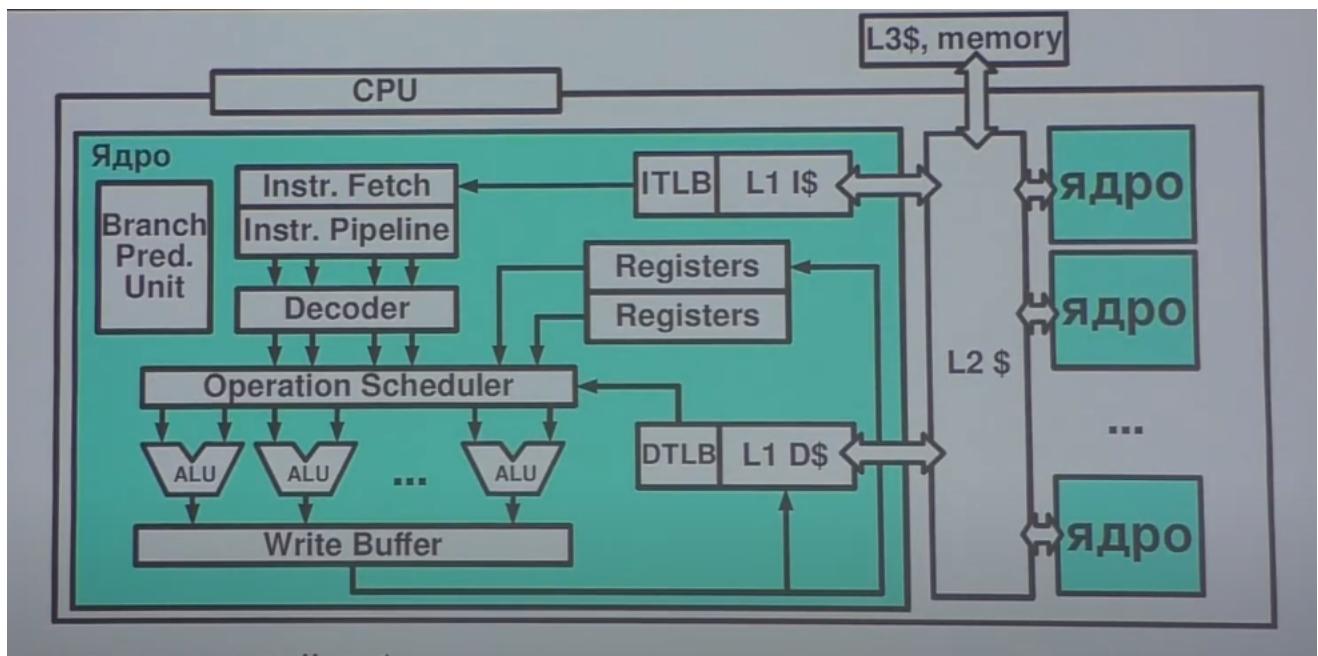
В архитектуре **NUMA** у нас есть память, которая находится ближе к какому-то процессору и память, которая доступна через коммутатор (передает данные через порты).

Адресное пространство для данной архитектуры является общим.

Огромным плюсом является, что можно заменять ее части прямо во время работы, что сильно повышает надежность системы.

## 2 Обзор элементов компьютерных систем

### 2.1 Процессор



Составляющие:

1. Арифметико-логическое устройство (АЛУ), выполняющее действия над операндами.
2. Буфер ассоциативной трансляции (TLB) — хранит информацию, есть ли такие-то данные в данном кэше.
3. Кэш процессора, используемый микропроцессором компьютера для уменьшения среднего времени доступа к компьютерной памяти. Делится на L1 и L1 d. Один из них хранит набор инструкций для работы с кэшем, другой данные.
4. Регистры для хранения данных, адресов и служебной информации.
5. Декодер команд.
6. Буфер для записи — хранит данные, пока буфер не освободится для записи.
7. Branch Pred. Unit — предполагает куда будут записаны данные, по какому адресу (последовательно или с каким-то отступом).
8. Instr. Pipeline — это метод реализации параллелизма на уровне команд в пределах одного процессора.

Важно помнить, что процессор выполняет команды последовательно. Пока один компонент выполняет одно действие, другой выполняет другое (они не останавливаются пока одни данные пройдут от начала до конца).

*Определение 1. Виртуальная память* — это подход к управлению памятью компьютером, который скрывает физическую память (в различных формах, таких как: оперативная память, ПЗУ или жесткие диски) за единым интерфейсом, позволяя создавать программы, которые работают с ними как с единым непрерывным массивом памяти с произвольным доступом.

|                 | Объем           | Тд        | *       | Тип       | Управл.    |
|-----------------|-----------------|-----------|---------|-----------|------------|
| CPU             | 100-1000 б.     | <1нс      | 1с      | Регистр   | компилятор |
| L1 Cache        | 32-128Кб        | 1-4нс     | 2с      | Ассоц.    | аппаратура |
| L2-L3 Cache     | 0.5-32Мб        | 8-20нс    | 19с     | Ассоц.    | аппаратура |
| Основная память | 0.5Гб-4Тб       | 60-200нс  | 50-300с | Адресная  | программно |
| SSD             | 128Гб-1Тб/drive | 25-250мкс | 5д      | Блочн.    | программно |
| Жесткие диски   | 0.5Тб-4Тб/drive | 5-20мс    | 4м      | Блочн.    | программно |
| Магнитные ленты | 1-6Тб/к         | 1-240с    | 200л    | Последов. | программно |

Управляется компилятором — означает, что именно компилятор определяет, как именно ваша программа будет взаимодействовать с данным блоком памяти, те что в какие регистры запишется и тд.

### **3 Общие сведения об операционных системах**

#### **3.1 Функции OS**

- Разработка программ.
- Выполнение программ.
- Доступ к устройствам ввода / вывода.
- Контролируемый доступ к файлам.
- Доступ к системе и системным ресурсам.
- Обнаружение и обработка ошибок.
- Учет пользования и диспетчеризация ресурсов.
- Предоставление ключевых интерфейсов (ISA — набор команд, ABI — бинарный интерфейс приложения, API — интерфейс прикладных программ).

#### **3.2 Оператор ЭВМ**

Что должен делать оператор?

1. получить программу с данными от программиста.
2. подготовить программу к загрузке.
3. загрузить программу и компилятор.
4. запустить программу на вычисление.
5. распечатку с результатом передать программисту.

Минусами оператора ЭВМ является: наличие расписания машинного времени и долгое время подготовки к работе.

#### **3.3 Пакетная обработка**

В следствие минусов оператора ЭВМ появилась пакетная обработка.

Появился первый Системный монитор, который включал в себя: обработчик прерываний, драйверы устройств, планировщик заданий, интерпретатор командного языка и было отведено пространство под пользовательские программы и данные.

#### **3.4 Многозадачность**

Одним из главных минусов первых ЭВМ было то, что вовремя вывода, ввода или работы других устройств процессор простоявал. В следствие этого появилась концепция многозадачности.

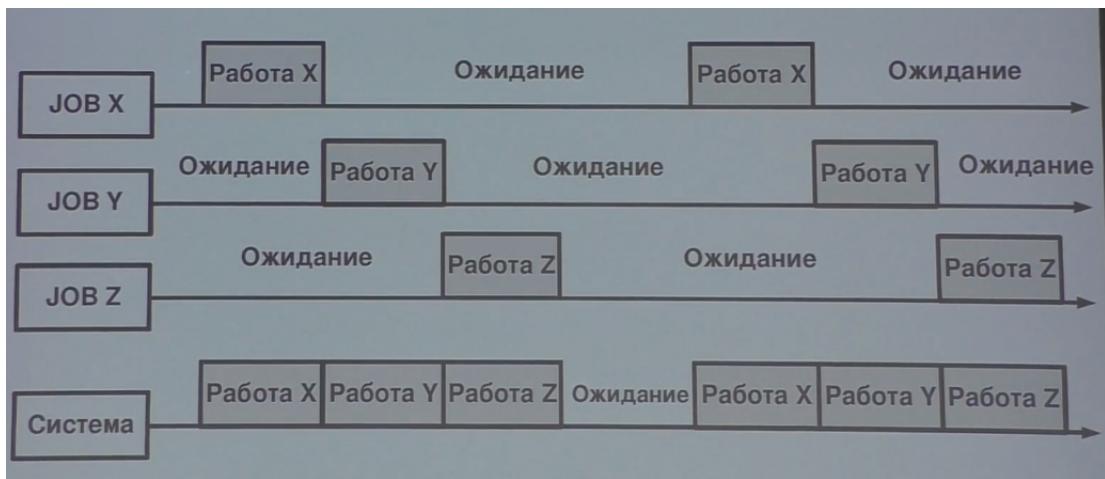


Рисунок 3 – Схема многозадачности первых ЭВМ

### 3.5 Разделение времени

Следующим нововведением в ЭВМ стало исключение оператора и добавление пользователей. Каждому пользователю выдавалось часть времени процессора с использованием квантового времени. В следствие этого появились проблемы разделения ресурсов и защита одних программ от других.

## 4 Основные задачи OS

### 4.1 Управление процессами

*Определение 2. Процесс* (с точки зрения обывателя) — экземпляр программы во время ее исполнения.

*Определение 3. Процесс* (с точки зрения OS) — единица потребления ресурсов OS, в которой существует последовательность действий, текущее состояние и набор связанных ресурсов.

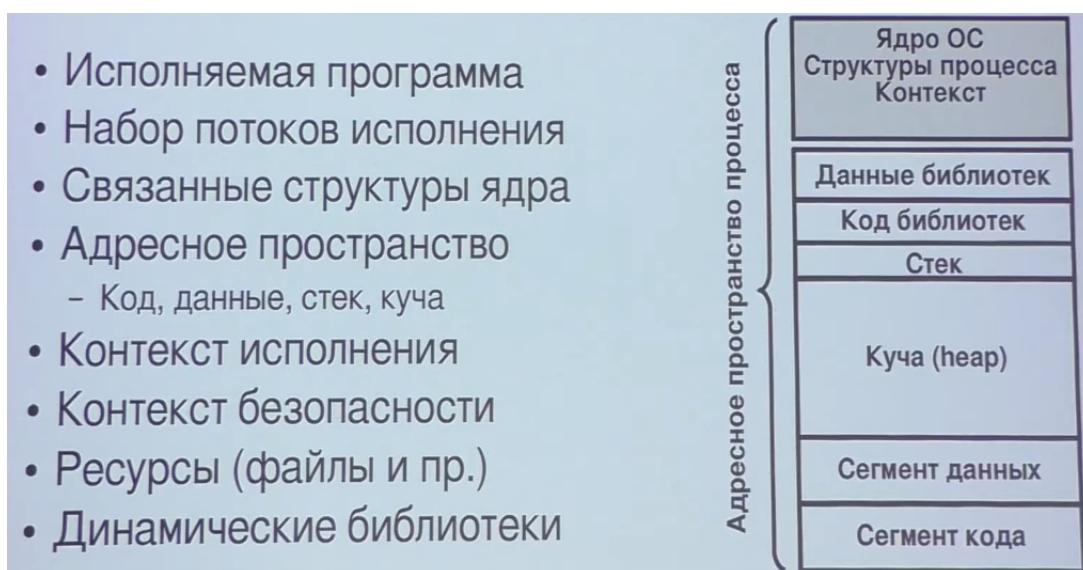


Рисунок 4 – Структура процесса

Для того, чтобы создать процесс, необходимо создать все части адресного пространства представленного на рисунке 4.

Процесс создается не так быстро, поэтому для вычислений на процессоре можно просто создать поток (по сути он будет представлять набор регистров) и с помощью него провести вычисления. Это все и является контекстом.

Когда создается процесс, ядро OS должно построить для ресурсов, которое он будет потреблять систему (описание ресурсов) (в линуксе task structure).

Проблемы современных процессов:

- Защита памяти процессов — недетерминированное поведение процесса, к примеру обращение не к своей памяти, может нарушить другие процессы.
- Взаимные блокировки — есть два процесса, один из них захватил один ресурс, другой другой, и они пытаются также добавить к себе захваченный другим процессом ресурс. (deadlock, livelock, starvation)

- Проблема синхронизации — тк у нас может быть несколько процессов, а адресное пространство для них одно.
- Взаимное исключение доступа ресурсов.

## **4.2 Виртуальная память**

Для решения проблемы с единым адресным пространством была придумана виртуальная память.

Управление памятью:

- Изоляция процессов.
- Управление выделением и освобождением памяти (аллокаторы и менинг памяти).
- Поддержка модулей (модульности) — динамическая загрузка и выгрузка модулей.
- Защита и контроль доступа — права на сегменты памяти.
- Долговременное хранение — запись информации на диск.
- Страницочный обмен.

*Определение 4. Виртуальная память* — отдельное виртуальное адресное пространство для каждого процесса и ядра.

Также виртуальная память подразумевает, что некоторые страницы нельзя выгружать из памяти, к примеру если они используются в большом количестве процессов.

## **4.3 Безопасность**

Также важный аспект OS это то, на сколько она безопасна, на сколько она обеспечивает безопасность данных.

Самым важным аспектом безопасности является протокол работы с информацией.

Что должна обеспечивать OS:

1. Безопасность доступа к системе — защита от несанкционированного доступа.
2. Конфиденциальность — невозможность неавторизованного доступа к данным.
3. Целостность данных — защита данных от неавторизованного и нецелостного изменения.
4. Аутентификация и авторизация.

#### **4.4 Диспетчеризация и планирование ресурсов**

Что важно учесть при планирование ресурсов (с точки зрения OS):

- Равноправие — пользователи, процессы и тд должны получать ресурсы равноправно. (Интересный факт: в UNIX приоритет процесса развернутого окна на 15 пунктов выше других).
- Дифференциация отклика — в некоторых задачах нужно понизить время отклика, к примеру в задачах выполняющихся в реальном времени.
- Общесистемная эффективность.
- Планировщик процессов, дисков и тд.

## 5 Современные архитектурные концепции OS

### 5.1 Архитектура ядер

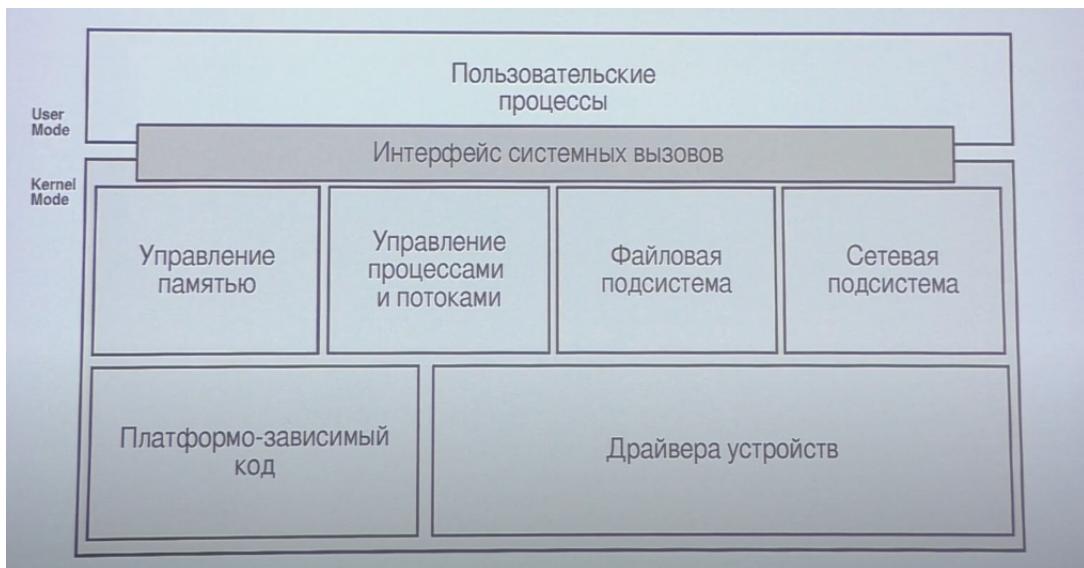


Рисунок 5 – Схема ядра ОС

Управление памятью, процессами и потоками, файловая подсистема и сетевая подсистема работают на основе драйверов и платформо - зависимого кода.

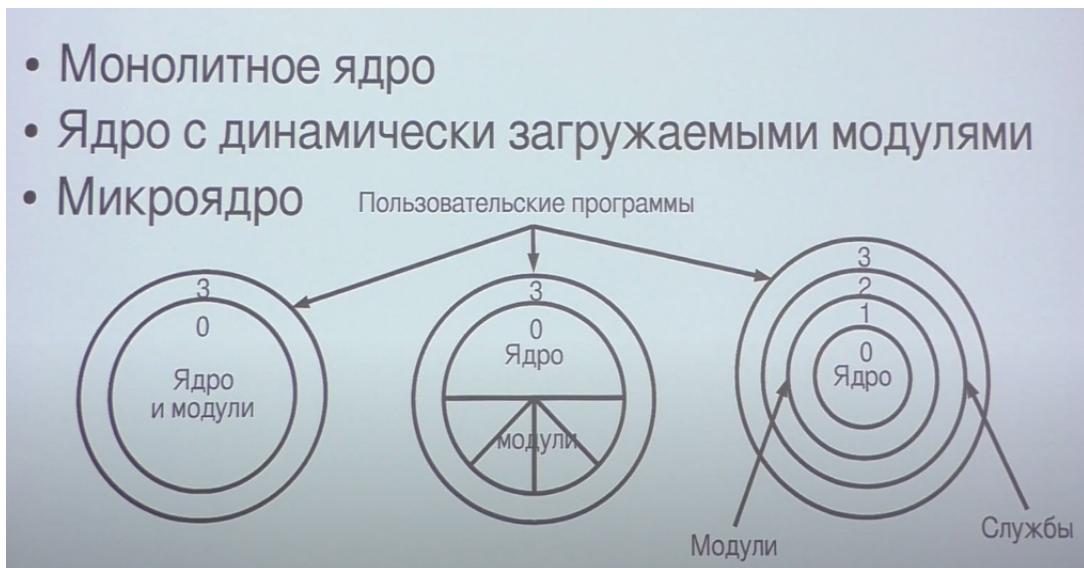


Рисунок 6 – Виды архитектур ядер ОС

Монолитное ядро — подразумевает, что для изменения чего-то в ядре придется перекомпилировать OS. Подходит для систем где набор устройств

определен и не будет изменяться. 0 уровень — ядро и встроенные в него модули, 3 уровень — пользовательские программы. 1 и 2 не используются.

Ядро с динамически загружаемыми модулями имеет возможность загрузить модули во время выполнения операционной системы.

Микроядро — концепция, в которой само ядро занимается базовыми задачами: диспетчеризация процессов и выделение памяти. 1 и 2 уровни занимают остальные задачи, реализованные в виде сервисов. Пользовательские приложения работают на 3 уровне. Из-за частого переключения контекстов это работает очень медленно.

## 5.2 Многопоточность

Из-за сложности создания процесса была придумана концепция реализации внутри процесса потоков. Thread — нить / поток.

Библиотека порождающая потоки на UNIX системах Posix Threads.

Существует множество концепций реализации потоков, они будут рассмотрены в следующих параграфах.

## 5.3 SMP и ASMP

**Symmetric multiprocessing** — процессы равны, процесс выполняется на нескольких процессорах одновременно. Это дает следующие плюсы: простота разработки и производительность, более высокая надежность (при отказе выполнить процесс, его могут выполнить другие), масштабируемость приложений, динамическое добавление ресурсов процессора.

**Asymmetric multiprocessing** — в системе с асимметричной многопроцессорностью не все процессоры играют одинаковую роль. Например, система может использовать (либо на аппаратном, либо на уровне операционной системы) только один процессор для выполнения кода операционной системы, или поручать только одному процессору выполнение операций ввода-вывода. В других AMP-системах все процессоры могут выполнять код операционной системы и операции ввода-вывода, так что с этой стороны они ведут себя как симметричная многопроцессорная система, но определенная периферийная аппаратура может быть подсоединенна только к одному процессору, так что со стороны работы с этой аппаратурой система предстаёт асимметричной. Более дешевая альтернатива в системах, которые поддерживали SMP.

**Многопоточность ! = Многопроцессорность**

## 5.4 Виртуализация

**Виртуальные машины (интерпретаторы)** — по сути программы, которые работают под выполнением другой программы. Как примеры: JS в браузере, python, JAVA VM. Это позволяет поднять уровень абстракции.

*Определение 5. Интерпретация* — построчный анализ, обработка и выполнение исходного кода программы или запроса, в отличие от компиляции, где весь текст программы, перед запуском анализируется и транслируется в машинный или байт-код без её выполнения.

**Контейнеры приложений** — позволяет писать приложения один раз и запускать их где угодно. Разработчики могут создавать и развертывать приложения быстрее и безопаснее, чем при традиционном подходе к написанию кода — когда он разрабатывается в определенной вычислительной среде, а его перенос в новое место, например из тестовой среды в продуктивную, часто приводит к ошибкам выполнения кода.

*Определение 6. Контейнер приложения* — экземпляр исполняемого программного обеспечения (ПО), который объединяет двоичный код приложения вместе со всеми связанными файлами конфигурации, библиотеками, зависимостями и средой выполнения.

Смысл и главное преимущество технологии в том, что контейнер абстрагирует приложение от операционной системы хоста, то есть остается автономным, благодаря чему становится легко переносимым — способным работать на любой платформе.

Примеры: Docker, Solaris containers, Linux containers.

**Аппаратурная виртуализация** — виртуализация с поддержкой специальной процессорной архитектуры. В отличие от программной виртуализации с помощью данной техники возможно использование изолированных "гостевых" операционных систем.

Примеры: Virtual BOX, KVM.

**Облачные технологии** — по сути облачная виртуализация, главным плюсом является, что в случае сбоя одной физической системы, данные иммигрируют на другую систему и продолжат выполняться. Данные технологии построены на базе аппаратурной виртуализации.

## **6 Основные понятия надежности операционной системы**

### **6.1 Надежность и отказоустойчивость**

**Отказоустойчивость** — способность системы продолжать работу при аппаратных или программных ошибках.

Для обеспечения отказоустойчивости нужно:

- Избыточность аппаратуры(двойное, тройное резервирование).
- Аппаратная "горячая"замена компонентов.
- Программная поддержка OS выведения компонентов из системы и их подключения.
- Организация уровней хранения RAID в дисковой подсистеме.

**Надежность** — вероятность бесперебойной работы системы до времени  $t$ , при условии ее корректной работы в  $t = 0$ .

Среднее время наработка на отказ MTTF =  $\int_0^x R(t) dt$ , включает в себя время на перезагрузку, ремонта или замены неисправного компонента, установки (переустановки) OS или ПО.

Коэффициент доступности — процент времени, когда система или служба доступна для запросов пользователей.

Простой (downtime) — время, в течение которого система недоступна

Безотказная работа — когда система находится в продуктивной работе.

### **6.2 Сбои**

Какие бывают отказы:

- Ошибочное состояние аппаратуры или ПО в результате сбоя компонентов.
- Ошибки оператора.
- Физические помехи окружающей среды.
- Ошибки проектирования, программирования, структуры данных и тд.
- Могут быть: постоянные, временные (однократные или периодические).

Методы резервирования:

- Физическая избыточность (компонентов, серверов).
- Временная избыточность (повтор вычислений).
- Информационная избыточность (ECC, RAID).

Методы повышения отказоустойчивости:

- Изоляция процессов
- Разрешение блокировок при параллелизме
- Виртуализация
- Точки восстановления и откаты

## **7 Общая архитектура UNIX / Linux**

В UNIX появилась ключевая концепция: все есть файл или процесс. Также появился принцип: одна программа — одна функция. Также использовалась концепция минимизации ядра, реализация на С и унификация файлов.

**SINGL UNIX Specification** — общее название для семейства стандартов, которым должна удовлетворять операционная система, чтобы называться "UNIX".

**UNIX системы** — AIX, MAC OS X, Solaris.

**UNIX like системы** — Linux, Free BSD, Open Solaris.

**Подробно архитектуру ядра Linux можно посмотреть по ссылке:**

<https://makelinux.github.io/kernel/map/>

Основные подсистемы Linux:

- Процессы и планировщик задач — создает, управляет и планирует процессы.
- Виртуальная память — выделяет виртуальную память для процессов и управляет ею.
- Физическая память — управляет пулом кадров страниц и выделяет страницы для виртуальной памяти.
- Файловая система — представляет глобальное иерархическое пространство имен для файлов и функции для работы с файлами.
- Драйверы символьных устройств — управление устройствами, которые требуют от ядра отправки и получения данных по одному байту.
- Драйверы блочных устройств — управление устройствами, которые читают и записывают данные блоками.
- Сетевые протоколы (TCP/IP) — поддержка пользовательского интерфейса сокетов для набора протоколов.
- Драйверы сетевых устройств.
- Ловушки и отказы — обработка генерируемых прерываний.
- Прерывания — обработка прерываний от периферийных устройств.
- Сигналы и IPC — управление межпроцессорным взаимодействием.

## 8 Общая архитектура Windows

В первых версиях Windows была надстройкой над операционной системой DOS. В данный момент у Windows есть несколько линеек: для мобильных устройств, для ПК и серверная.

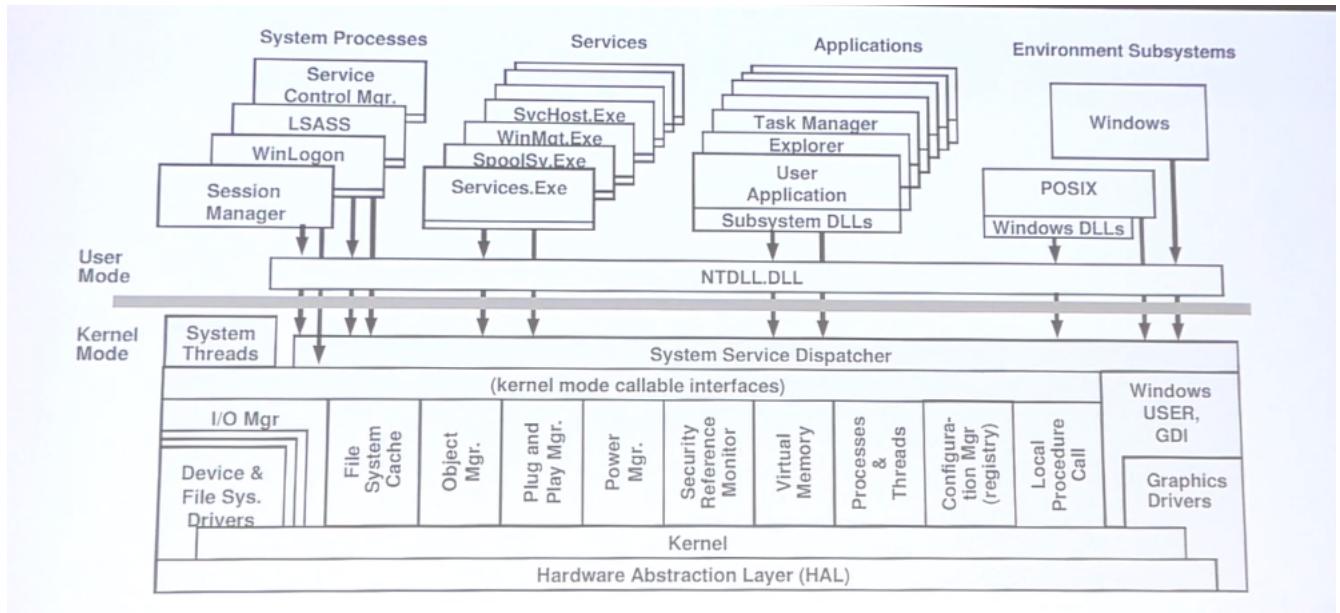


Рисунок 7 – Архитектура Windows

Как не странно архитектура Windows очень похожа на архитектуру UNIX.

Интерфейс системных вызовов — NTDLL.dll

Также подсистемы Windows похожи на подсистемы в Linux.

Plag and play Mgr. — система позволяющая легко ставить драйверы, без указания портов и тд.

Одним из отличий является графический интерфейс интегрированный в ядро.

**WinAPI** — это библиотеки динамической компоновки (DLL), которые являются частью Windows операционной системы. Они используются для выполнения задач, когда сложно написать эквивалентные процедуры. Например, Windows предоставляет функцию с именем FlashWindowEx, которая позволяет сделать заголовок строки приложения чередующимся между светлыми и темными оттенками. Позволяет легко написать приложение, которое будет совместимо с любой версией Windows.

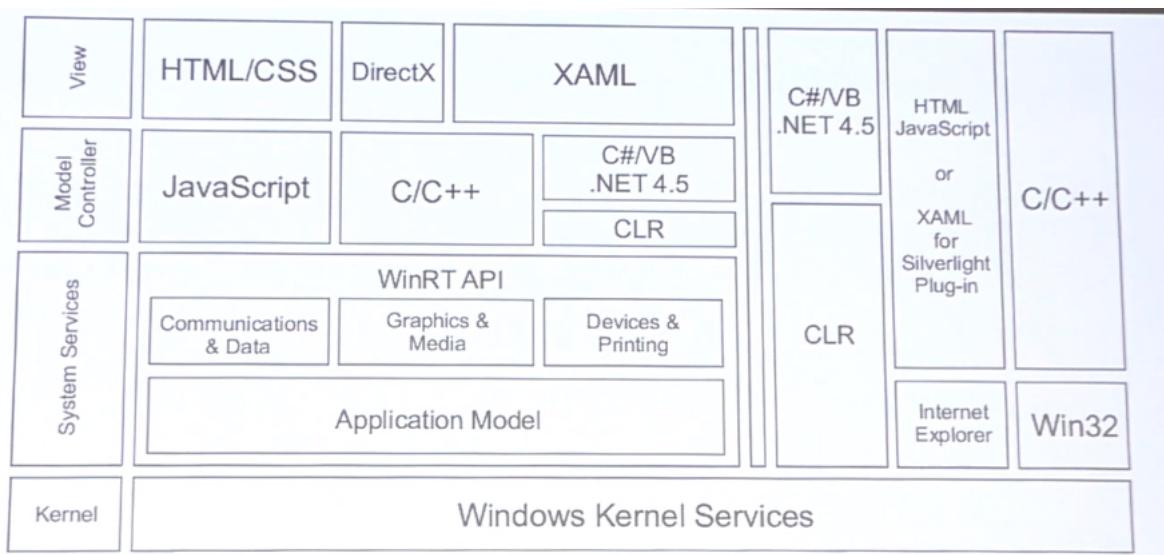


Рисунок 8 – Схема WinAPI

Другие важные компоненты Windows:

- Гипервизор Hyper-V — запуск гостевых операционных систем.
- Firmware — содержимое энергонезависимой памяти любого цифрового вычислительного устройства — микрокалькулятора, сотового телефона, GPS-навигатора и т. д., в которой содержится его программа.
- Terminal Servers.
- Объекты — все вещи в системе сделаны в виде объектов.
- Реестр.
- Оснастки — специальная вспомогательная программа для администрирования выделенного пула задач.

## 9 Средства для отладки Linux

### 9.1 Стандартные средства

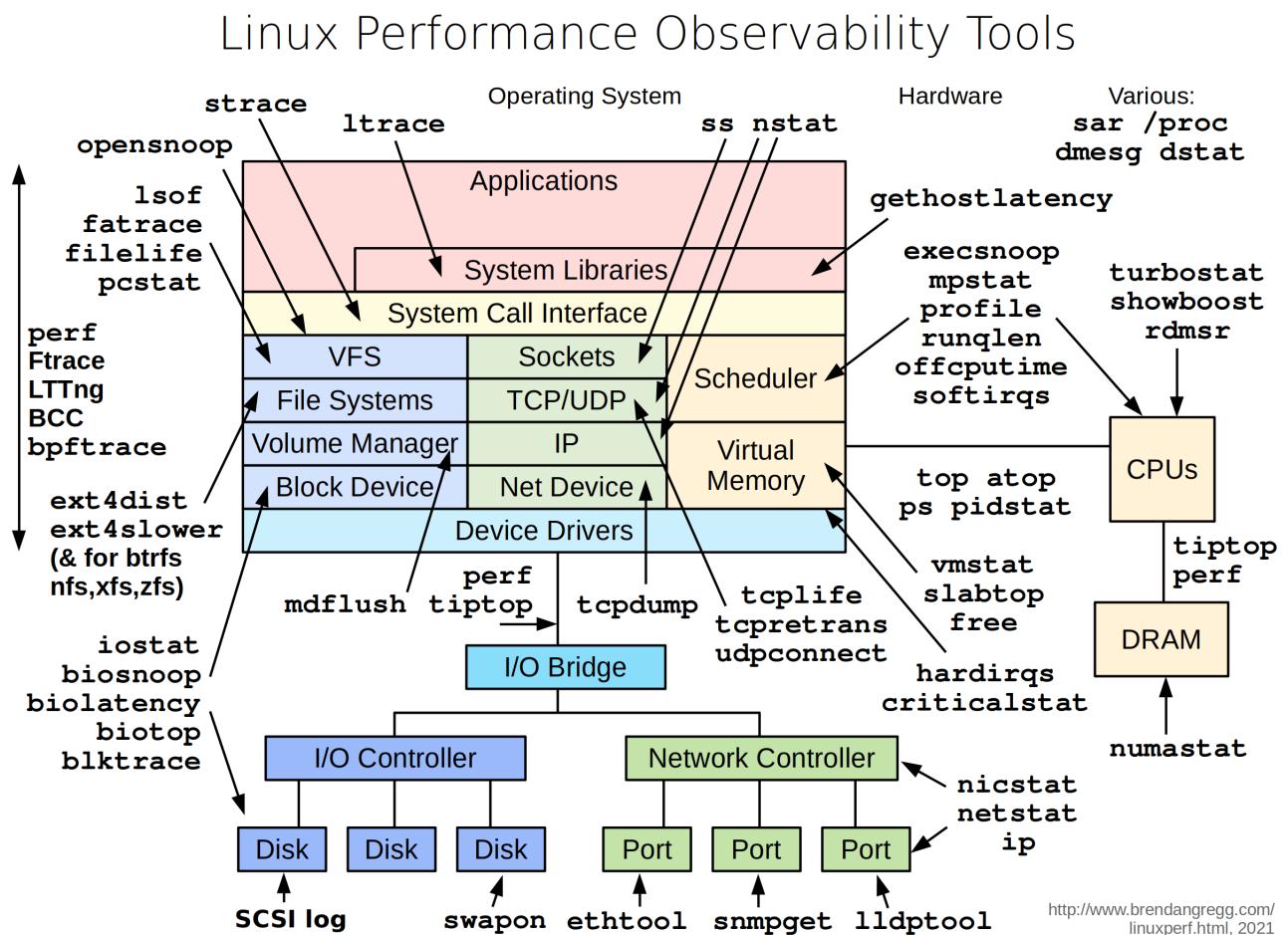


Рисунок 9 – Утилиты для отладки линукс

Ядро операционной системы накапливает большое количество различных счетчиков. И все представленные на рисунке 9 утилиты, по сути просто дают доступ к этим счетчикам, те никаких доп вычислений не производится.

### Стандартные средства наблюдения за счетчиками

**sar** — утилита, которая позволяет посмотреть информацию о счетчиках любой подсистемы Linux.

Подробно ознакомиться можно здесь: <https://greendail.ru/node/monitoring-proizvoditelnosti-linux-na-primere-sar>

- Процессор: ps, top, tiptop, turbostat, rdmsr, numastat, uptime
- Виртуальная память: vmstat, slabtop, pidstat, free
- Дисковая подсистема: iostat, iotop, blktrace
- Сеть: netstat, tcpdump, iptraf, ethtool, nicstat, ip
- Интерактивные (типа top) или с указанием количества запуска и интервала (типа sar)
- Некоторые работают только с правами root!

Рисунок 10 – Другие встроенные утилиты

Утилиты обычно двух типов: интерактивные (можно изменять параметры системы) и статичные (просто предоставляют информацию).

## 9.2 /proc

**/proc** — виртуальная файловая система, которая содержащая файлы статистики и управляющая модулями ядра. По сути вся информация представлена в виде файловой системы. К примеру, информация о процессоре будет лежать в каталоге `/proc/cpuinfo`.

## 9.3 Трассировщики

- Трассировка системных вызовов: strace
- Трассировка вызовов библиотек: ltrace
- Трассировка lock -ф: bpftrace

Также одно из средств отладки, с помощью которого легко увидеть логи системных вызовов.

## 9.4 perf

Профилировщики — собирает системную информацию, которую вы указали.

Основное предназначение профилировщиков — это взять ваше готовое приложение и посмотреть, что находится в ядре во время его запуска.

Суть в том, что perf может собрать весь стэк трейс запущенной программы.

Естественно, запущенный perf будет вносить задержку в работу всей системы.

Но у нас есть флаг -F #, где # — частота сэмплирования, измеряемая в Гц.

К примеру perf record df -h запишет данные любой команды Perf, которую вы хотите сохранить для использования в будущем.

```
• usage: perf [-version] [-help] [OPTIONS] COMMAND [ARGS]
• The most commonly used perf commands are:
  • bench      General framework for benchmark suites
  • c2c        Shared Data C2C/HITM Analyzer.
  • config     Get and set variables in a configuration file.
  • data       Data file related processing
  • diff       Read perf.data files and display the differential profile
  • evlist    List the event names in a perf.data file
  • ftrace     simple wrapper for kernel's ftrace functionality
  • kallsyms  Searches running kernel for symbols
  • kmem      Tool to trace/measure kernel memory properties
  • list      List all symbolic event types
  • lock      Analyze lock events
  • mem       Profile memory accesses
  • record    Run a command and record its profile into perf.data
  • report    Read perf.data and display the profile
  • sched     Tool to trace/measure scheduler properties (latencies)
  • script    Read perf.data and display trace output
  • stat      Gather performance counter statistics on command
  • timechart Tool to visualize total system behavior
  • top       System profiling tool.
  • probe     Define new dynamic tracepoints
  • trace     strace inspired tool
```

Рисунок 11 – Базовые команды в perf

Одно из удобных визуальных представлений, что сохраняет профилировщик — FlameGraph.

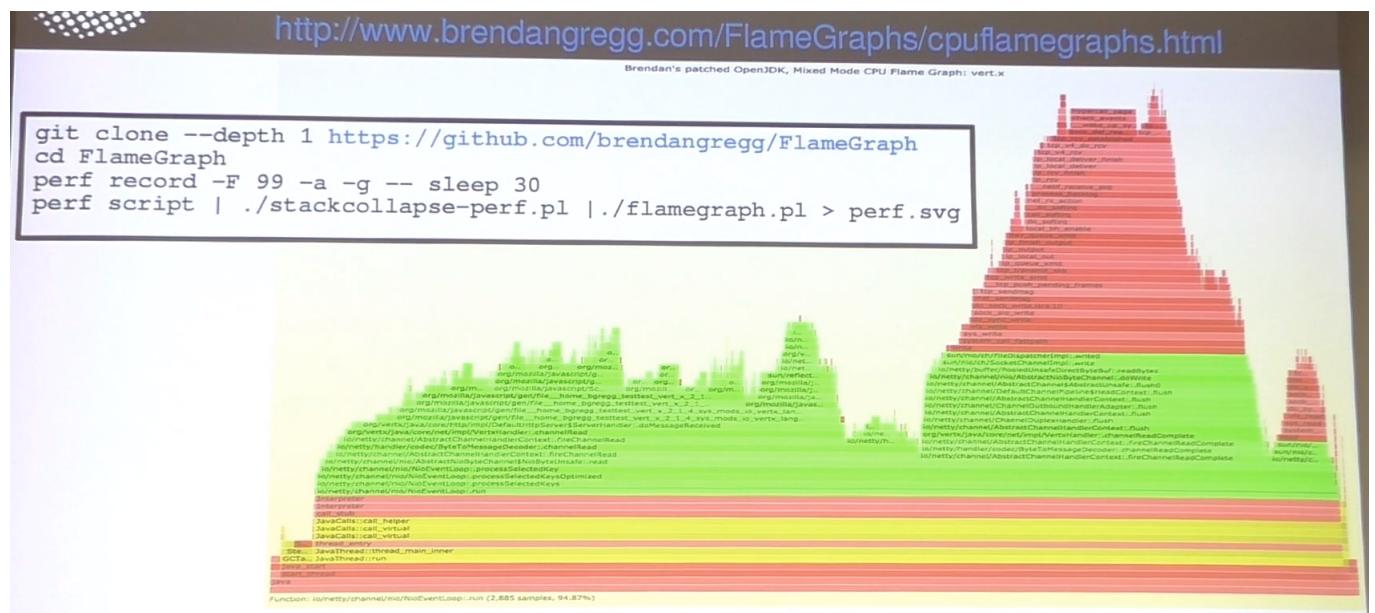


Рисунок 12 – FlameGraph

## 9.5 System tap

Еще одно средство сбора информации о подсистеме ядра или пользователя, при этом имеет минимальное воздействие на систему. SystemTap по сути имеет скриптовый синтаксис.

Основная идея SystemTap состоит в том, чтобы обозначить события и назначить для них обработчики.

Во время выполнения скрипта, SystemTap занимается мониторингом событий и, как только произойдет событие, ядро системы выполнит обработчик. Событиями могут быть начало или конец сессии SystemTap, срабатывание таймера и другие.

Обработчиком является последовательность скриптовых операторов, которые будут выполнены после срабатывания события. Обычно обработчики извлекают информацию из контекста события или выводят информацию на экран.

Сессия SystemTap начинается тогда, когда мы выполняем скрипт. В это время происходит следующая последовательность действий:

1. Сначала SystemTap проверяет библиотеку «тапсетов» на наличие использованных в скрипте;
2. Потом SystemTap транслирует скрипт в Си (язык программирования) и запускает системный компилятор, чтобы создать модуль ядра из скрипта;
3. SystemTap загружает модуль и активирует все события в скрипте;
4. Как только происходит событие выполняется обработчик данного события;
5. Когда все события выполнены, модуль выгружается и сессия завершается;

## 9.6 Kernel debugger

Существует два режима у отладчика ядра: локальный отладчик (предустановлен в системе) и удаленный (предоставляет информацию об системе, находящейся на другом компьютере).

Чтобы включить компиляцию kdb, вы должны сначала включить kgdb. Параметры компиляции тестов kgdb описаны в главе kgdb test suite

<https://docs.kernel.org/dev-tools/kgdb.html>.

Kdb - это упрощенный интерфейс в стиле оболочки, который можно использовать на системной консоли с клавиатурой или последовательной консолью. Вы можете использовать его для проверки памяти, регистров, списков

процессов, dmesg и даже установки точек останова для остановки в определенном месте. Kdb не является отладчиком исходного кода, хотя вы можете устанавливать точки останова и выполнять некоторые базовые элементы управления запуском ядра. Kdb в основном предназначена для проведения некоторого анализа, чтобы помочь в разработке или диагностике проблем ядра.

## **10 Средства для отладки Windows**

### **10.1 Встроенные средства**

**Windows SDK** — включает в себя: отладчик, множество утилит, поддерживающих сборку приложений.

**DTrace on Windows** — по сути System tap был скопирован с Dtrace для Windows, тк что по сути у них похожий функционал.

**Администрирование** — Disk Cleanup, Performance Monitor (очень удобное средство, в котором удобно можно задать параметры), Resource Monitor, Registry Editor Services, System Configuration и тд. (чтобы попасть туда нужно перейти по такому пути: Control Panel -> System and Security -> Administrative Tools).

**Task manager** — ну, тут не нужно лишних слов.

Также ссылка на сторонние программы: <https://habr.com/ru/company/ua-hosting/blog/280578/>

### **10.2 SysInternals**

Множество скриптов и программ для управления, диагностики, устранения неполадок и мониторинга всей среды Microsoft Windows. Автор Марк Руссинович, в настоящее время сотрудник Microsoft (Соавтор книги Windows Internals).

Top SysInternals utils:

- PsList and PsKill – просмотр и остановка процессов (в том числе и удаленно)
- Process Explorer - просмотр ресурсов процесса, замена Task Manager
- Process Monitor – просмотр связанных с процессом ресурсов реестра
- Autoruns - поиск автозапускаемых программ
- Contig - дефрагментирует конкретный файл
- PSFile – позволяет показать открытые файлы, в том числе и удаленно
- MoveFile - перемещает заблокированные файлы во время перезагрузки. • Sync - синхронизация файловой системы
- TCPview - информация о открытых сетевых соединениях
- SDelete - удалить файлы и папки без возможности восстановления

Тут очень много средств для отлавливания вирусов.

### **10.3 Отладчик ядра WinDbg и KD**

По умолчанию Windows не загружается в режиме отладчика, для этого есть специальные команды.

livekd (SysInternals) — позволяет перейти в режим отладки без перезагрузки системы.

Для использования WinDbg необходимо получить символьную информацию (тк в ядре информация хранится в виде 1 и 0, то нужно создать папку для хранения символов переменных, **ДЛЯ КАЖДОЙ СБОРКИ ВИНДЫ НУЖНА СВОЯ СИМВОЛИЧЕСКАЯ ИНФОРМАЦИЯ**).

Пример вызова отладчика:

```
set _NT_SYMBOL_PATH=srv*c:\symbols*http://msdl.microsoft.com/download/symbols
C:\Program Files (x86)\SysinternalsSuite>livekd64.exe -w -k "C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\windbg.exe"
```

Рисунок 13 – FlameGraph

## **Часть II**

# **Процессы**

### **11 Основы процессов**

#### **11.1 Вычисления**

По сути наша система компьютерная состоит из ресурсов: процессор, память устройства ввода-вывода. Приложения, которые мы пишем, решают практическую задачу: Входные данные → Обработка → Выходные данные. OS по сути находится между оборудованием и приложением. По сути OS представляет абстракции ресурсов и предоставляет их пользователям.

#### **11.2 Процесс. Характеристики процесса**

Каждая из подсистем рассматривает процесс в своем ключе.

И так, повторим, процесс это:

- Выполняемая программа;

- Экземпляр программы, выполняющийся на компьютере;
- Сущность, которая может быть назначена процессору и выполнена на нем;
- Единица активности, характеризуемая выполнением последовательных команд, текущим состоянием и связанным с ней множеством системных ресурсов;

Характеристики процесса в момент времени:

- Уникальный идентификатор;
- Состояние (выполнение, очередь, ожидание);
- Приоритет по отношению к другим процессам;
- Счетчик команд;
- Указатели на область памяти процесса;
- Контекст процесса (регистры, user / kernel);
- Статус ввода-вывода;
- Счетчики системных ресурсов;
- Права доступа процесса;
- и тд;

### 11.3 Состояние процессов и разделение ресурсов

Рассмотрим ситуацию, где есть два процесса с одинаковым приоритетом и числом доступных ресурсов. Вспомним, что в таком случае у нас для этих процессов будет выделено одинаковое количество процессорного времени.

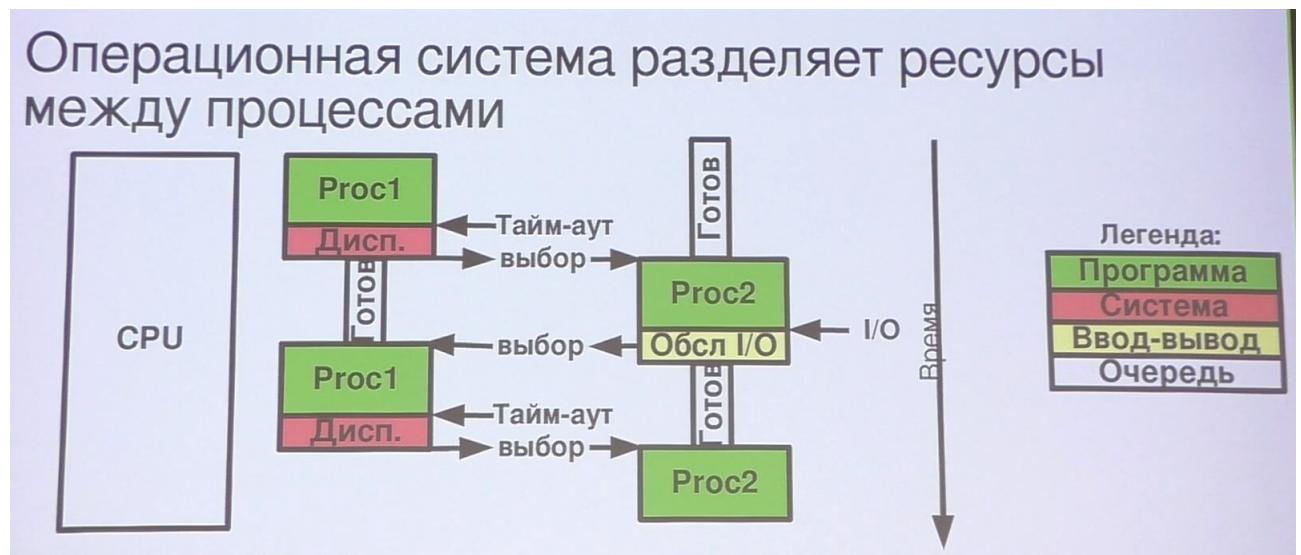


Рисунок 14

Как происходит распределение ресурсов процессора между процессами?

*Определение 7. Квант* — единица времени, периодичность, с которой система проверяет, не выполняется ли процесс слишком долго.

Раз в квант система проверяет, закончилось ли процессорное время у процесса, если да, то диспетчер берет другой, а этот снова кладет в очередь.

Это все делается для того, чтобы примерно одинаковые процессы выполнялись одинаковое количество времени.

### Рассмотрим состояния процесса

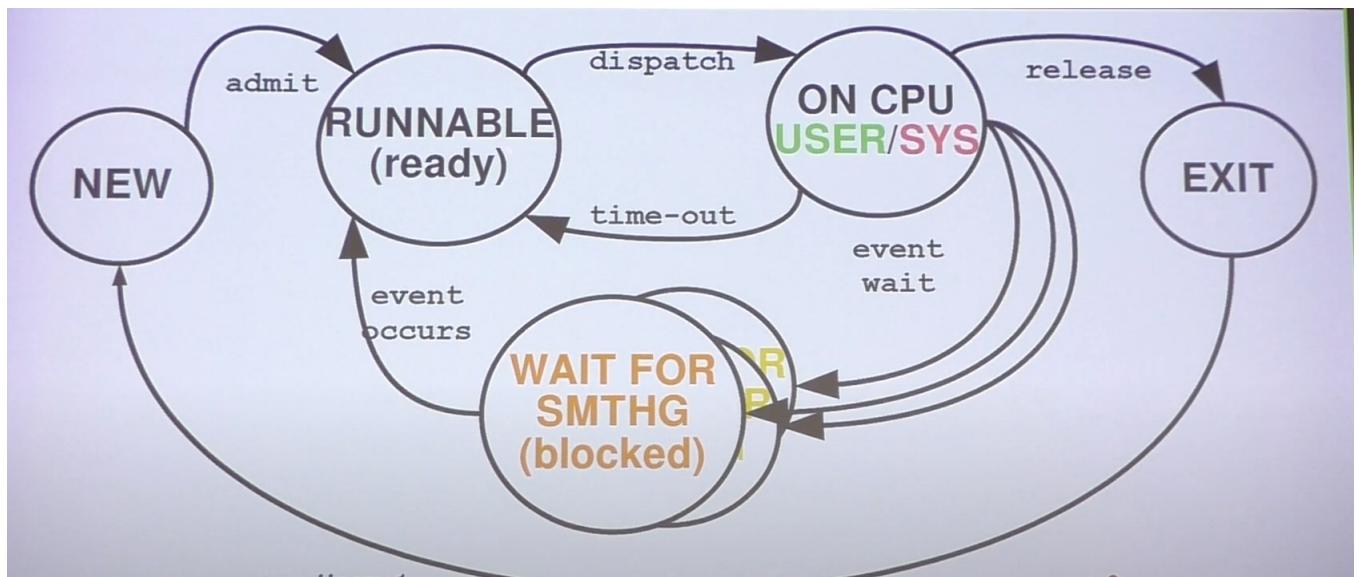


Рисунок 15 – Цикл работы процесса

- Ready — готов к выполнению, это значит процессор может взять этот процессор на исполнение. Есть все необходимые ресурсы, которые позволяют ему выполнится. К примеру, страницы памяти, сигнал завершения ввода вывода;
- On CPU — если процессор свободен, то мы можем назначить на него процесс. Процесс может работать в двух режимах user и sys. Во время выполнения процесса эти режимы могут переключаться;
- Выход из состояния On CPU в двух случаях — процесс выполнен или у него закончилось системное время;

**New state** — процесс создан, но еще не размещен в очереди процессов, готов к исполнению.

**Exit state** — процесс не может продолжить выполнение. (Структуры процесса еще существуют).

Причины попадания в состояние:

- Нормальное завершение (вызов exit);
- превышение лимитов на время выполнения;
- Недостаток памяти;
- Ошибки границ и защиты памяти;
- Арифметическая ошибка;
- Ошибка ввода-вывода;
- Неправильная или привилегированная инструкция;
- Команда оператора или OS;
- Завершение или запрос родительского процесса;

**Runnable state** — процесс обладает всеми ресурсами для выполнения, но нет возможности исполниться.

Причины попадания в состояние:

- Низкий приоритет по сравнению с другими процессами;
- Ожидания освобождения CPU;
- Закончился квант времени;

**Runnable state** — процесс выполняется на процессоре.

Процесс остается в этом состояния если:

- не истек квант времени;
- Ожидание на спин-блокировке;
- В Runnable состоянии нет процессов с более высоким приоритетом;
- Обслуживание высокоприоритетных прерываний;
- Нет блокирующих вызовов (ввод вывод, ожидание блокировки);

**Wait (blocked) state** — ожидания событий OS, освобождения блокировки. Процесс не расходует ресурсов CPU, процесс может находиться в ожидании неопределенно долго. Бывает процесс не убивается, пока он не будет разблокирован (самое простое, перезагрузить OS).

Состояние продолжается пока:

- освободится блокировка;
- Придет сообщение OS о наступление ожидаемого события.

## 12 Пейджинг и свопинг

### 12.1 Paging Swapping

Как правило, основной памяти всегда мало, если появляется свободная память программисты сразу пытаются ее заполнить. А также в системе обычно большое количество запущенных процессов.

Давайте поместим блокируемый процесс на диск и освободим основную память для других процессов.

*Определение 8. Paging* — выгрузка или загрузка неиспользуемых страниц процесса на диск.

*Определение 9. Swapping* — выгрузка всего процесса, кроме критически важных для ядра структур.

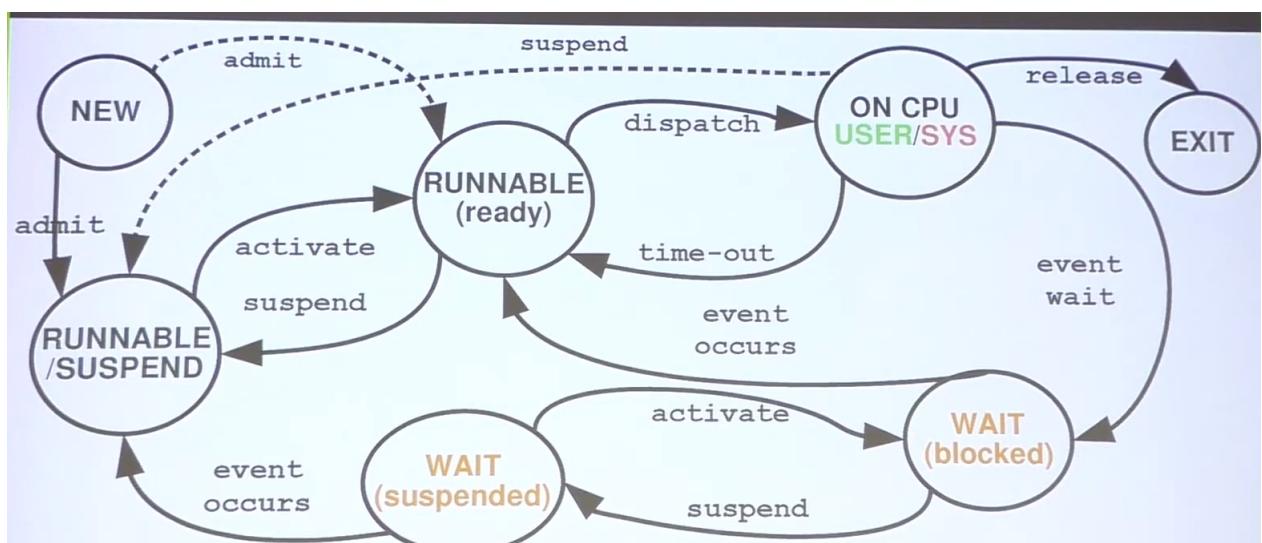


Рисунок 16 – Модель цикла процесса с 7 состояниями

На картинке можно наблюдать цикл состояния процесса с Paging / Swapping.

Изменения в том, что мы разделили состояние Wait и Runnable на две части.

Почему стрелочка из New идет в Runnable Suspend? Тк процесс создается долго, в современных системах, создается минимальный костяк процесса, без мепинга, и отправляется в приостановленное состояние, чтобы можно было быстро отчитаться о создание процесса.

Также интересное замечание, что может создаться такая ситуация, что при большом количестве процессов, может выстроиться очередь на блокировку процессов, тогда вся система начнет глобально тормозить.

## 12.2 Дополнительные состояния процессора

**Wain / Suspend state** — процесс приостановлен и выгружен в область подкачки. По событию Suspend процесс выгружается на диск. По событию Active загружается в основную память. (Данные действия повышают нагрузку на дисковую подсистему).

Причины попадания в состояние:

- Длительное ожидание события операционной системы;
- Недостаток памяти (зачем держать в памяти процесс, который не имеет возможности исполниться).

**Runnable / Suspend state** — процесс готов к выполнению, но выгружен из памяти.

Причины:

- Был неготов, выгружен, но произошло событие, которое позволяет выполниться;
- Desperate memory conditions;
- Команда пользователя;
- Создание процесса в минимальном варианте, без, к примеру создания сегментов памяти.

## 13 Управление процессами

### 13.1 Управляющие таблицы

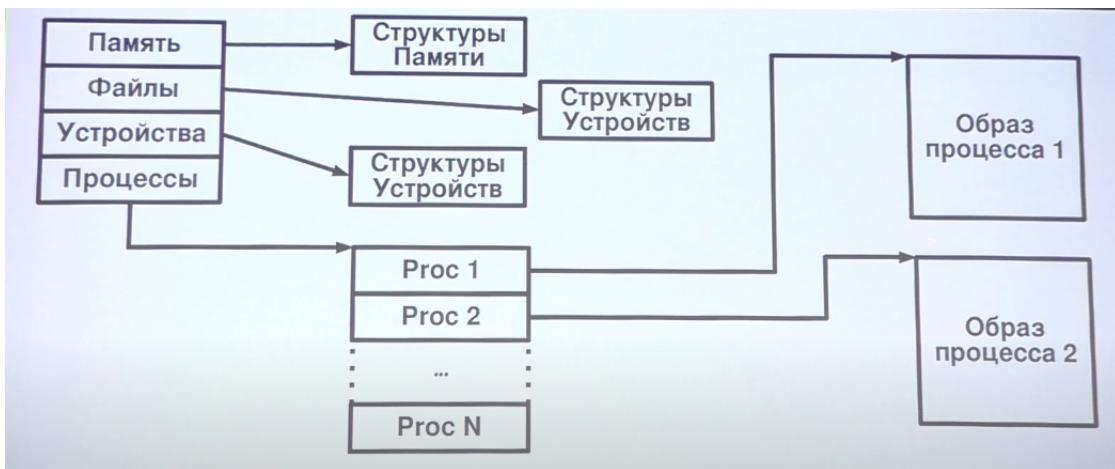


Рисунок 17 – Общая структура управляющей таблицы

### 13.2 Образ процесса

Допустим у нас есть код написанный на языке С.

```
#include <stdio.h>

int var_int_data[1024]; static char var_char_data[4096];

void foo(int var_inc) {
    int var_a = 10;
    static int var_sa = 10;

    var_a += var_inc;
    var_sa += var_inc; printf("a = %d, sa = %d\n", var_a, var_sa);
}

int main(int argc, char**argv) {
    int var_i;

    for (var_i = 0; var_i < 10; ++var_i)
        foo(var_i);
}
```

Давайте попробуем понять, где что находится?

```

serge@ra:/tmp$ gcc -o a_prog -g a.c
serge@ra:/tmp$ gdb a_prog
GNU gdb (Ubuntu 9.1-0ubuntu1) 9.1
[...]
(gdb) break foo
Breakpoint 1 at 0x1149: file a.c, line 7
(gdb) run
Starting program: /tmp/a_prog
Breakpoint 1, foo (var_inc=32767) at a.c:7
7 {
(gdb) step 2
10     var_a += var_inc;
(gdb) print &var_char_data
[...] &var_int_data, &var_a, &var_sa, &var_inc
$4 = (char (*)[4096]) 0x555555558040 <var_char_data>
$3 = (int (*)[1024]) 0x555555559040 <var_int_data>
$5 = (int *) 0x7fffffffde5c #var_a
$6 = (int *) 0x555555558010 <var_sa>
$7 = (int *) 0x7fffffffde4c #var_inc

```

| Address            | Kbytes | RSS | Dirty | Mode  | Mapping      |
|--------------------|--------|-----|-------|-------|--------------|
| ...                |        |     |       |       |              |
| 0000555555555000   | 4      | 4   | 4     | r-x-- | a_prog       |
| ...                |        |     |       |       |              |
| 0000555555559000   | 8      | 0   | 0     | rw--- | [anon]       |
| 00007ffff7dc0000   | 148    | 148 | 0     | r---- | libc-2.31.so |
| 00007ffff7de5000   | 1504   | 484 | 8     | r-x-- | libc-2.31.so |
| ...                |        |     |       |       |              |
| 00007ffff7fab000   | 12     | 12  | 12    | rw--- | libc-2.31.so |
| 00007ffff7fae000   | 24     | 20  | 20    | rw--- | [anon]       |
| 00007ffff7fc0000   | 12     | 0   | 0     | r---- | [anon]       |
| 00007ffff7fce000   | 4      | 4   | 4     | r-x-- | [anon]       |
| 00007ffff7fcf000   | 4      | 4   | 0     | r---- | ld-2.31.so   |
| 00007ffff7fd0000   | 140    | 140 | 24    | r-x-- | ld-2.31.so   |
| ...                |        |     |       |       |              |
| 00007ffff7ffd000   | 4      | 4   | 4     | rw--- | ld-2.31.so   |
| 00007ffff7ffe000   | 4      | 4   | 4     | rw--- | [anon]       |
| 00007ffff7fffe000  | 132    | 12  | 12    | rw--- | [stack]      |
| ffffffffffff600000 | 4      | 0   | 0     | --x-- | [anon]       |
| total kB           | 2368   | 960 | 120   |       |              |

Рисунок 18

Запомним: rw — сегмент данных или куча (anon), а также у нас снизу есть еще и стек. Строки где одни ——, значит, что к этим страницам обратиться нельзя.

Пройдемся дебагером, и увидим что массив чаров у нас ушел в сегмент данных, по адресу ....8040. А массив интов, ушел в кучу по адресу ....9040. Важно помнить, что разные компиляторы ведут себя по разному. Также мы видим, статические переменные попали в стек.



Рисунок 19 – Управляющий блок процесса

В управляемом блоке процесса содержится информация о: инициализации процесса, какие регистры ему принадлежат, а также многое другое).

Если смотреть на наш процесс внутри операционной системы, то есть несколько очередей, в которых содержатся ссылки на управляющие блоки процессов. Все эти структуры тесно связаны между собой, иногда переходят из одной очереди в другую.

### 13.3 Функции OS

Функции OS связанные с процессами:

#### Управление процессами

- Создание и завершение процессов;
- Планирование и диспетчеризация процессов;
- Переключение процессов;
- Синхронизация и поддержка обмена информацией между процессами;
- Организация управляющих блоков процессов

#### Управление вводом-выводом

- Управление буферами;
- Выделение процессам каналов и устройств ввода - вывода

#### Управление памятью

- Выделение адресного пространства процессам;
- Управление страницами и сегментами;
- Пейджинг и Свопинг;

#### Функции поддержки

- Обработка прерываний;
- Учет пользования ресурсами;
- Текущий контроль системы;

**Вспомним, что** создание процесса включает в себя:

- Присвоение уникального идентификатора;
- Выделение памяти;
- Инициализация PCB;
- Постановка процесса в очередь ядра;
- Создание потоков ввода-вывода;

- Создание других управляющих структур данных.

### 13.4 Процессы SVR4

На картинке представлен цикл создания процесса в UNIX.

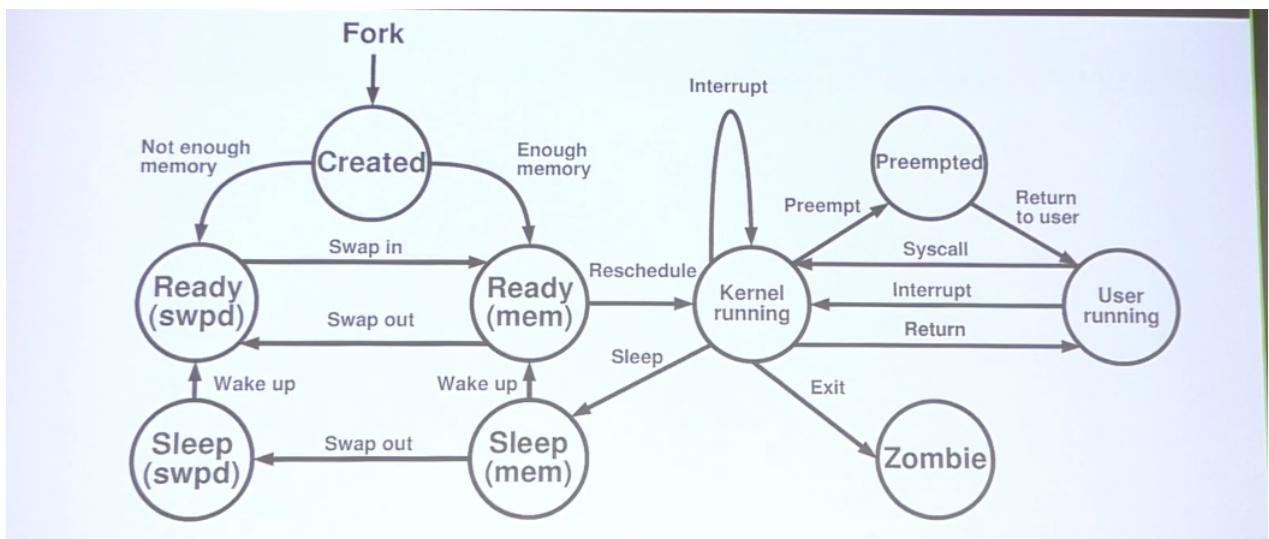


Рисунок 20 – Создание процессов в UNIX

Можем наблюдать явно использование процесса в user и kernel режимах.

## 14 Потоки

### 14.1 Понятие потока

Абстрактная модель процесса подразумевает владение ресурсами, а также планирование и диспетчеризацию.

По сути, процесс является единицей группировки ресурсов, а Thread — единица выполнения программного кода.

Thread содержит: Состояние выполнения, Сохранение контекста потока, Стек, Локальные переменные, Доступ к памяти и ресурсам процесса владельца.

### 14.2 Связь потоков и процессов

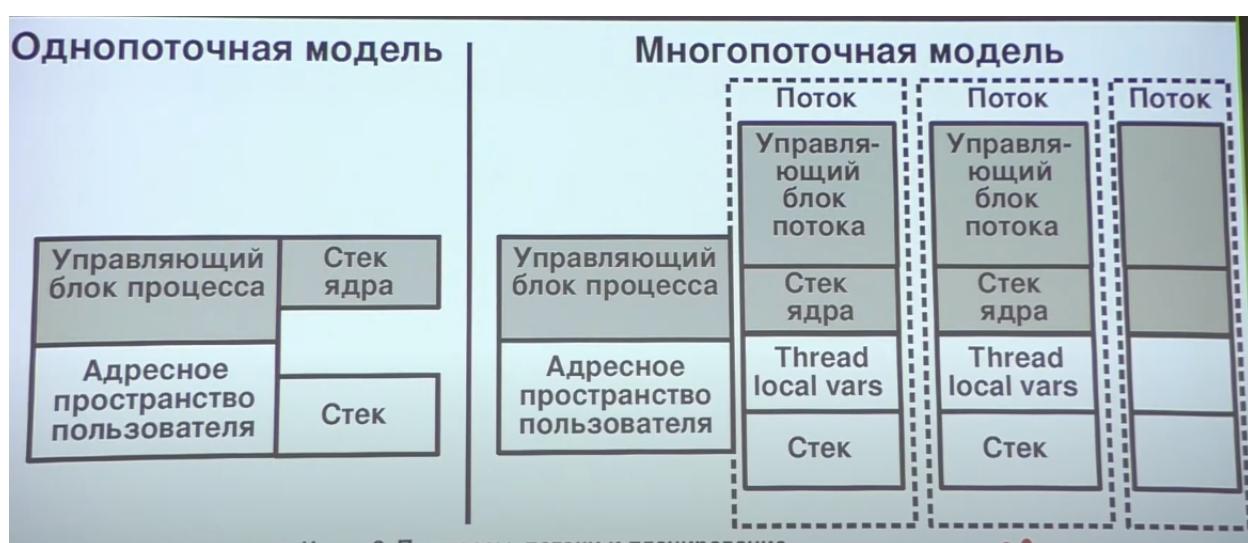


Рисунок 21 – Связь структур ядра и потока

Как мы видим у нас также есть управляемый блок процесса. Важно заметить, что структуры отвечающие за управления у потоков разные, а ресурсы одни.

Главное преимущество потоков в быстродействие, все действия для них выполняются быстрее (Переключение между потоками быстрее, тк переключается только контекст, убиваются они тоже быстрее, но лучше лишний раз не убивать, тк это все равно энергозатратно). Даже в однопоточных моделях есть преимущества (работа в приоритетном и фоновом режиме, асинхронная работа частей программы, модульная структура программы).

### 14.3 Состояния потока

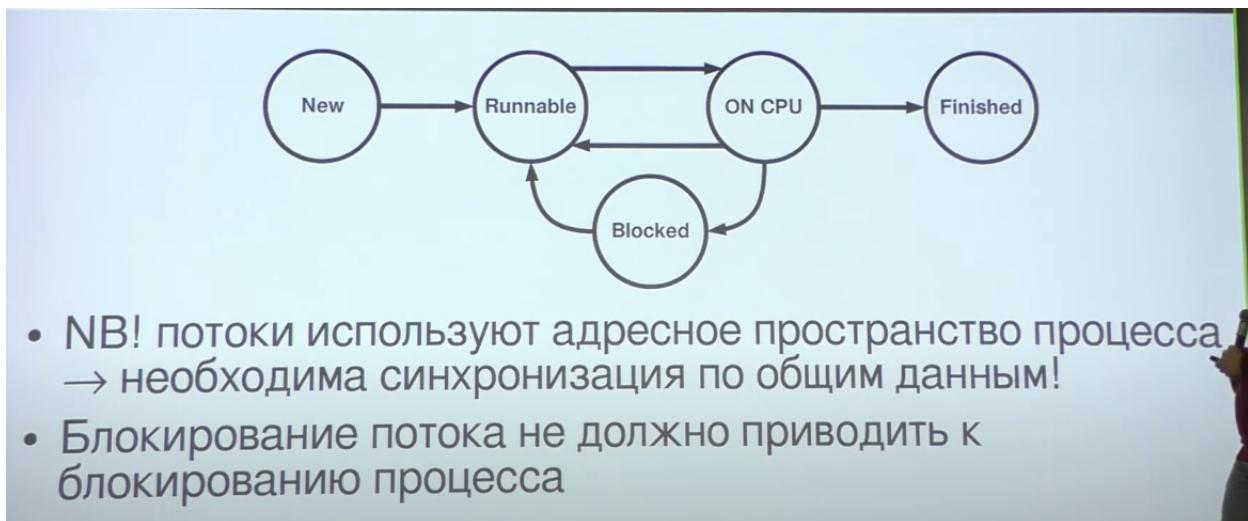


Рисунок 22 – Состояние потоков

Как мы видим, состояния потоков не сильно отличаются от состояний процессов, за исключением того, что нет структур отвечающих за диспетчериизацию.

### 14.4 Варианты реализации

User level Thread — реализуется библиотеками или приложениями на стороне пользователя.

Kernel level Thread — реализуется ядром.

Основная разница заключается в том, что для User lvl для переключения между потоками не нужно уходить в ядро.

В данный момент чаще всего используется модель KLT, сколько потоков мы породили на уровне пользователя, столько получим и на уровне ядра.

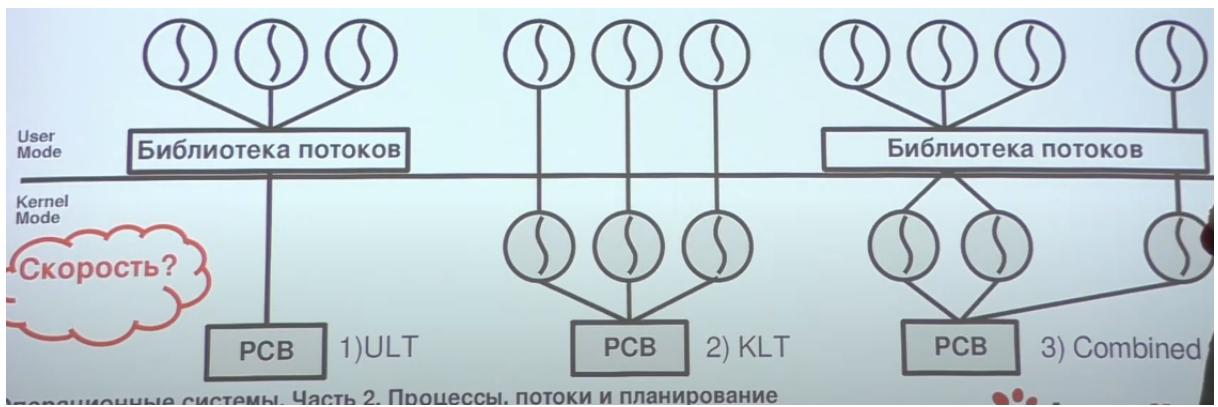


Рисунок 23 – Модели потоков

## 14.5 Закон Амдала

На сколько можно ускорить программу на N процессах с использованием потоков?

**Теорема 1. Закон Амдала** — ускорение = время работы на одном процессоре / время работы на N процессах.

$$\text{ускорение} = \frac{T*(1-f)+T*f}{T*(1-f)+\frac{T*f}{N}} = \frac{1}{(1-f)+\frac{f}{N}}$$

Где T — время работы, f — доля распараллеливания [0,1). Когда f мало, использование параллельного выполнения неэффективно. Когда N стремится к бесконечности, то ускорение ограничено  $\frac{1}{1-f}$ .

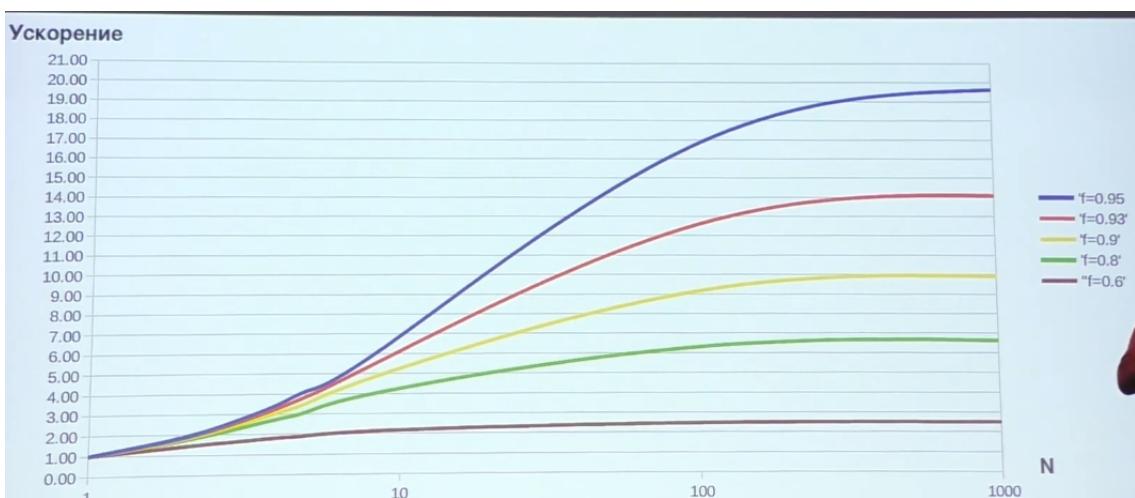


Рисунок 24 – Визуализация закона Амдала

## **15 Полезные утилиты**

Linux kernel map — <https://makelinux.github.io/kernel/map/>

Сайт с рекомендациями по отладке Linux —

<https://brendangregg.com/linuxperf.html>