Операционные системы

Содержание

Существовали 4 принципа архитектуры фон Неймана — «люди жили в раю». А потом случилось грехопадение: эти принципы начали нарушать. Ну и что, что фон Нейман сказал, что так надо — давайте не будем так делать. В итоге: **«голову вытянет — хвост увязнет»**.

Из-за того что память **неоднородная**, мы получаем одни проблемы. Если доступ

к памяти возможен не только через процессор — получаем другие проблемы. Плюсы у этих решений есть, но костыли тоже нужны.

Асимптотики алгоритмов — это идеальность. В реальных боевых условиях процессор может дать процессорное время, а может и не дать. Никто не выделит алгоритму целое ядро. То же самое касается доступа к диску, памяти, сети.

На этом курсе мы будем убирать мифы из головы.

Перетаскивание папки с SSD на флешку с помощью мышки не ощущается чудом. Хотя под капотом — разные файловые системы (в одном случае двухмерные координаты, в другом — трёхмерные).

Закон сохранения энергии работает везде. Если у вас был опыт спортивного программирования (Codeforces и прочее), вы знаете: можно написать программу, идеальную по памяти, или идеальную по времени. Но одновременно идеальную и по тому, и по другому — нельзя. Чем-то всегда приходится жертвовать.

ИИ не заберёт рабочие места. Он сместит точку принятия решения.

О защите дипломов. Студенту задают вопрос: «А что, если упадёт это?» Студент радостно отвечает: «А я это предусмотрел, у меня есть вот это». Ему встречный вопрос: «А что, если это упадёт?» И студент понимает, что так можно задавать вопросы бесконечно. Это философский вопрос, который ведёт в никуда.

Противоречивость требований

Со школы и младших курсов есть мнение, что главное — пройти автотесты, тесты на функциональность. Но есть и **нефункциональные требования**:

- Если я резервирую — мне нужно синхронизировать.
- Если синхронизирую — теряю производительность.
- И так далее.

Проклятие инженера: если я что-то делаю лучше, что-то другое получается хуже.

Главное противоречие — справедливость и эффективность.

Пример с очередью на кассе: стоят люди с разным количеством товаров. Подходит бедный студент с одной бутылкой воды. - Для **кассира** — всё равно, кого обслуживать первым: общее время обслуживания одно и то же. - Для **магазина** эффективнее пропустить вперёд того, у кого мало товаров: очередь короче, новые

покупатели не уйдут. Можно даже поставить охранника-менеджера, который будет так делать. - Но что скажут люди в очереди? — «Это несправедливо!»

> Bash появился в 1978 году. Было масса попыток его заменить: PHP, Perl, Python. Но bash остался — и, кажется, никуда не денется.

Организация курса

- Лабораторные — 50 баллов
- 6 контрольных — 30 баллов (вопросы в открытом виде)
- Экзамен — 20 баллов

Эволюция Операционных Систем

> **Операционная система** — это *базовая* (без неё нельзя обойтись) *системная* (не прикладное ПО — само по себе ценности не имеет, но без неё ничего сделать нельзя) программа, управляющая работой вычислительного узла (без ОС нет доступа к железу) и реализующая универсальный интерфейс между аппаратным обеспечением, программным обеспечением и пользователем.

Архитектура фон Неймана

Если зайти в диспетчер задач, видно сотни процессов. И все эти процессы хотят пользоваться железом…

4 принципа архитектуры фон Неймана: 1. **Однородность памяти** — данные и код хранятся в одной памяти. 2. **Интерпретация** — содержимое ячейки трактуется в зависимости от контекста. 3. **Адресность** — каждая ячейка памяти имеет уникальный адрес. 4. **Программное управление** — программа определяет последовательность выполнения.

Этап 1. Подпрограммы, диспетчер, прерывания

На этом этапе появилось **переиспользование кода** (конец 40-х, начало 50-х; ещё нет ни C++, ни даже ассемблера — только инструкции, написанные двоичными битами).

Идея: возьмём часть памяти и сразу заложим туда реализации типовых функций (поиск, сортировка и т. п.). Фактически создаём библиотеку подпрограмм.

Программа идёт сверху вниз по инкременту. Но нужно как-то перепрыгивать на ячейку начала подпрограммы. Нужно ещё: - передать аргумент, - сохранить место, откуда прыгнули (адрес возврата).

Тогда в память подпрограммы кладётся адрес возврата. Но эти адреса перезаписывать нельзя.

Возникает проблема: если я понял, что реализацию синуса можно эффективнее переписать, мне придётся **перемещать и переписывать весь код**, потому что байт в байт у меня не выйдет.

Решение — динамическая таблица (диспетчер). У диспетчера есть таблица: левая часть всегда неизменная (ключ — название подпрограммы), правая — адрес начала. Фактически это просто индексация массива.

Кроме того, диспетчеру можно передать стек. Так диспетчер становится универсальным управленцем.

> Диспетчер решал проблему **линковки**, **автоматизации загрузки** и **повторного использования кода**.

Контроллеры и прерывания

Появилась идея предзагружать данные в ОЗУ. Но ЦПУ может либо гнать данные, либо их обрабатывать — по архитектуре фон Неймана одновременно нельзя. Решение: **создадим процессор поменьше и попроще, который умеет только гонять данные** — это контроллер устройства (Storage controller).

Процессор может только давать контроллеру инструкции перегнать данные. Но мы не знаем, закончил ли контроллер перегонку — это ведь физическая штука (магнитная лента, перфокарта…). Нужен механизм оповещения — **прерывание**. Контроллер сообщает процессору, что закончил.

> **Прерывание** — обращение (сигнал), поступающее от внешнего устройства в центральный процессор, сообщающее о наступлении некоторого события, приостанавливающее исполнение текущего потока команд и передающее управление обработчику этого прерывания.

Структурные программы и пакеты

Раньше считалось, что программы — это монолиты. Затем начали появляться **структурные программы**. Теперь можно было дробить работу между разными программистами: кто-то пишет один метод, кто-то — другой. Появились **модули** — это не просто методы, но и связанные с ними данные (например, предрассчитанные таблицы). Из модулей появились **пакеты**.

Оверлейное программирование

В условиях ограниченной памяти нужно было исполнять код, который заведомо весит больше, чем доступно памяти. Решение — **оверлеи**: мы выгружаем ненужный кусок кода и подгружаем нужный.

Теперь диспетчер получает третью табличку — с номерами модулей. Она нужна для подгрузки нужных кусков кода.

Хочется минимизировать время простоя — появляется **планировщик**.

Получилась программа-диспетчер, надстройка над фон-неймановской архитектурой, позволяющая эффективнее пользоваться процессором.

Но мы оптимизировали ситуацию только для одной программы. А хочется, чтобы пока одна программа долго что-то вычисляет (не кушая память), параллельно могла работать другая. Идея: разместить в памяти несколько программ и сделать оптимизацию многомерной — на множестве программ и устройств. Так появляется второй этап.

Этап 2. Мультипрограммные операционные системы

Разделили память на несколько программ. Процессор будет выполнять команду то одной программы, то другой. Звучало прекрасно, но реализовать это было очень трудно. Нужно разделить процессорное время — **Processor Sharing**.

Это **псевдопараллельность**: на самом деле в каждый момент времени работает одна программа, просто очень быстро переключаемся.

> **Марковский процесс** — каждый шаг зависит только от предыдущего состояния.

Кооперативная многозадачность Идея: «Я в коде поставлю точки, и диспетчер будет решать — то ли мне обратно вернуть управление, то ли переключить-

ся на другой процесс». Но программисты не хотели вставлять эти кусочки кода. Появилась идея делать ветвление так, чтобы эти точки **обходить**. В итоге была только вероятность, что процессорное время будет делиться равномерно.

В 60-е от кооперативной многозадачности отказались. Но потом, в 2020-е, к ней вернулись: горутины, корутины Kotlin, fiber'ы. Появились state-машины.

> До середины 90-х не верили в программистов и в код — верили только в железо. ПО воспринималось как что-то вспомогательное.

Вытесняющая многозадачность Хорошо, кооперативная многозадачность не помогла. Давайте сделаем это «железно». Добавили **кварцевые часы**, которые регулярно вызывают прерывания. Диспетчер задач через равные промежутки времени проверяет планировщик и отдаёт управление программам.

> RISC-архитектура сменяется CISC-архитектурой. Появляется та архитектура процессора, которой мы пользуемся сейчас.

Регистровый контекст Нельзя просто так остановиться в любой точке — в регистрах могут быть нужные данные. Появляется понятие **регистрового контекста**. Обработчик прерывания таймера становится не таким простым: нужно не просто перейти к обработке другой программы, но и **выгрузить один контекст и подгрузить другой**.

Итог: > **Processor Sharing = Таймер прерываний + Регистровый контекст**

Виртуальная память

Программа компилируется — компилятор указывает адрес в памяти для переменной. Но адрес дан относительно нуля. Мы ведь не знаем, куда именно загрузится программа. Решение — **виртуальная память**; эту задачу повесили на диспетчер.

> **Виртуальная память** — абстракция, позволяющая при разработке или компиляции программного кода использовать адресацию от нуля, а в момент загрузки или исполнения заменять абстрактные (виртуальные) адреса на физические.

С одной стороны, мультипрограммность делается ради эффективности. С другой — мы тратим ресурсы на переключение между программами. **Парадокс**: мы тратим ресурсы, чтобы их экономить.

Защита памяти и привилегированный режим

С мультипрограммностью появилась возможность из одной программы случайно (или специально) полезть в память другой.

> Реальный случай: сервер в Беркли, рядом запущена программа студента и расчёт траектории баллистической ракеты. Программа студента случайно записала что-то в память программы ракеты. Ракету запустили — она сбилась с курса. Осталось приносить дипломатические извинения.

Появляется задача **защиты памяти** — программа должна работать только со своей памятью. Вводится понятие **привилегированного режима**: код в этом режиме может использовать любой кусок памяти.

> «Мы раздали людям в форме автоматы». Это было единственным решением — не давать привилегированного режима нельзя, иначе не сможем защищаться. Но с другой стороны, у кого-то может «полететь кукуха».

Так мы получили **Program Isolation**.

Файлы

Изолировали ОЗУ — но есть ещё и ПЗУ. Зачем в памяти лежит функция, которой я неделю не пользуюсь? Пусть лежит на ПЗУ, а оверлеем подтянем, когда понадобится.

Появляется концепция **файла**: некоторой области данных даём имя, нужное только для программиста.

Межпроцессное взаимодействие и каналы

Появляется необходимость коммуницировать между программами. Программисты захотели объединять несколько программ в пакеты. Чтобы они могли взаимодействовать, нужно было просить привилегированный процесс дать разрешение.

> В 1-й лабе есть `Pipe`: `stdout` левой команды закрывается на `stdin` правой команды.

Планировщик теперь должен учитывать связи между процессами. Например, в «Ленте»: очередь огромная за салатами (память), но кассы (процессор) пустые. Это уже не марковские процессы.

Виртуальная машина (ранний смысл термина)

Появилась концепция **Virtual Machine** — это ещё далеко до современного понятия виртуализации.

> **Лирическое отступление — миф о вечном возвращении.** Жизнь человека — последовательность процессов. Если взять бесконечную ось времени, то на ней эти последовательные процессы будут бесконечно повторяться — и это снова будем мы. Бесконечное количество одинаковых наших жизней.

Каждая программа живёт внутри виртуальной памяти, а операционная система распределяет ресурсы между разными «виртуальными процессами». Именно в этот момент появился термин **операционная система**.

Компания **Burroughs** создала компьютер B5000, диспетчер которого назывался **MCP (Main Control Program)**.

Этап 3. Сетевые операционные системы

Узкое место — ввод/вывод. У суперкомпьютеров был один общий ввод и вывод. Был **Teletype**: рулон бумаги, замыкающиеся магнитики. При вводе — ты их замыкаешь; при выводе программа замыкает магнитики и бьёт молоточками по бумаге. В то время это было прорывом — предмет обихода печатает, возникало ощущение «искусственного интеллекта».

Оператор вводил программы, запускал, записывал вывод и отдавал заказчику.

Бизнес-модель машинного времени

> Хотите рассчитать маршрут для логистической компании. Нанимаете программиста, он пишет код. Но запустить негде. Отправляете программиста из Сан-Франциско в Бостон в командировку. Машинист вводит код, запускает, ловит деление на ноль, выдаёт отчёт. Программист едет обратно: «всё плохо». Дорого, неэффективно.

Появление сетей Компания **AT&T** взяла кредиты и протянула медные провода по всей США — это была телефонная связь.

Появился **модем**.

Но теперь нет умного оператора, который проверял правильность кода и выставлял счёт (bill). Нужно это чем-то заменить. Появляются **аккаунты** и **биллинг**. Как понять, кто именно подключается? Появляется л**огический вход (logical login)** и 3 этапа аутентификации:

1. **Идентификация** — присваиваем человеку логическое имя (пин-код, биометрию, хэш и т. д., которые хранятся в системе).
2. **Аутентификация** — процесс проверки, сопоставление хэшей.
3. **Авторизация** — предоставление ресурсов после успешной аутентификации.

Позже появится **многофакторная аутентификация**.

Суперкомпьютинг становится бизнесом. США — большая страна: на восточном побережье ночь, на западном день. Компьютеры на одном побережье погибают от нагрузки, на другом простаивают. Возникает идея соединить компьютеры модемной связью и **раскидывать job’ы** между ними.

Постепенное нарастание функционала ОС

Операционные системы становились всё сложнее, обрастали новым функционалом и требовали больше ресурсов. Разработчики упёрлись в стену: писать под каждую железяку с нуля становилось невозможно дорого.

Так подошли к идее создания **универсальных операционных систем** (раньше говорили «мобильные», но сейчас слово вызывает путаницу — лучше **переносимые**: возможность перемещать ОС с одного железа на другое).

- **Linux** можно поставить на любую железяку (сложно представить устройство, куда нельзя было бы поставить Linux).
- **macOS** на любую железяку уже не поставить (хотя пул поддерживаемых устройств всё ещё внушительный).

Раньше ОС писались под конкретный десяток моделей, не более. Начался процесс **унификации аппаратных платформ**.

- Если проект уникальный — нужны уникальные специалисты. Это дорого, такого специалиста дорого заменить.
- Обычного джависта при этом можно заменять как шестерёнку в большой системе: один не подошёл — взял другого. (Меркантильненько, но такова жизнь.)

Так пришли к идее аппаратных платформ: «давайте все процессоры будут иметь одну систему команд». Появилась платформенная разработка, выросли большие гиганты, которые потом станут IBM. Начались работы в Xerox PARC (раньше они были гигантами; мышка, принтер — всё родилось в Xerox PARC, но неудачный менеджмент погубил их величие).

Раньше, по сути, кто делал железо — тот и писал ОС. После появления аппаратных платформ ОС стали унифицироваться под эти платформы.

Языки высокого уровня

Появляется идея языков высокого уровня — гораздо приятнее писать не команду `MUL` с какими-то адресами, а просто ставить звёздочку, а машина пусть сама команды подставляет.

Так возникло разделение на **компилируемые** и **интерпретируемые** языки.

Интерпретируемые языки были медленными и сложными в отладке (нет всего контекста, чтение только построчное), но давали возможность писать под разные

системы — код работал везде, где есть интерпретатор. Но интерпретатор тоже надо было написать.

Создать железо, которое поддерживало бы все языки программирования — невероятно дорого.

Начали унифицировать и ОС: написать её на языке высокого уровня. Но как? Высокоуровневый язык должен компилироваться, а компилятор — часть языка ОС… Классическая проблема яйца и курицы.

Эту задачу решили в Bell Labs.

Феномен Bell Labs

Обычно частные компании не могут заниматься фундаментальными исследованиями — нужно обосновать инвесторам трату миллиардов на то, что может не дать результата. («Дайте нам десятки миллиардов на коллайдер, мы поищем бозон Хиггса. Но не факт, что найдём».)

Bell Labs смогла удивить всех. В её стенах трудились 8 нобелевских лауреатов. Доходы были сверхвысокими — из-за них даже пересматривали налогообложение компаний.

Что сделали в Bell Labs: - Изобрели **полевой транзистор**, создав базу для всей современной электроники. - Придумали **ПЗС-матрицу**, на которой работает любая цифровая камера. - Разработали процессы эпитаксиального выращивания кристаллов (привет TSMC). - Заложили основы **теории связи** — формула Шеннона. - Открыли **реликтовое излучение**. - И многое другое.

Bell Labs совместно с MIT работали над проектом **MULTICS** (Multiplexed Information and Computing Service). Затем три товарища — **Кен Томпсон, Брайан Керниган и Деннис Ричи** — решили развивать свою версию, форк MULTICS. Их система не обещала быть крутой и эффективной, но метила в **универсальность**.

Так на свет появилась UNICS

Эти же люди позже создадут язык **C**.

Для начала они написали систему с нуля на ассемблере. Три человека справились за полгода.

> **1 января 1970 года была запущена UNICS 1.**

Она пока ничем принципиально не отличалась от других ОС — просто была написана своими руками на ассемблере. С этой даты, кстати, ведётся отсчёт **UNIX-времени (Unix timestamp)**.

Не было компилятора языка высокого уровня — разрабатывают **язык B** (буква A была занята ассемблером). B создавался как промежуточный шаг.

> Кстати, `awk` — последняя буква K это **K**ернниган.

Получив интерпретатор B, который работал на UNICS, они переписали саму UNICS на B. Появилась **UNICS 2**. Цепочка замкнулась, но было медленно — B интерпретируемый.

Тогда приступили к созданию языка **C**. Компилятор для C написали на B.

- **1973 год — UNICS 3:** есть компилятор C, написанный на B.
- **Конец 1973 — UNICS 4:** ядро переписано на C.
- **1975 — UNICS 5:** полностью переписана на C.
- **1978 — UNICS 7:** дорабатывались разные вещи, появился `sh` (shell, предок bash).

AT&T и антимонопольная служба

На AT&T (которой принадлежит Bell Labs) наезжает антимонопольная служба. Формально телекоммуникационная компания не может самостоятельно заниматься продажей операционных систем.

Был придуман смарт-мув: в США если бизнес связан с университетом, можно обойти много законов. Код UNICS отдали **университету Беркли**. Они создают коммерческое предприятие — **Berkeley Software Distribution**.

Так появилась **BSD**, от которой позже родятся FreeBSD, OpenBSD, NetBSD.

> «Закон большой тройки»: в любой высокотехнологической отрасли есть 3 главных игрока, держащих 80–90% рынка.

3 университета вовлечены в развитие индустрии: **MIT, Berkeley, Stanford**.

Стэнфорд явно проигрывал гонку. Они взяли ОС от Беркли, но ушли больше в научную сторону. У них с Беркли были разногласия в сетевых технологиях: Беркли развивали **NCP**, Стэнфорд — **TCP/IP**. Стэнфорд решает сделать форк — появляется **SUN** (Stanford University Network). На базе этого форка Стэнфорд создаёт коммерческую ОС **Solaris**.

UNIX: новое имя старой системы

AT&T продолжает попытки обойти антимонопольное законодательство. Годы Paper Work дают результат: им позволяют развивать свою ОС с MIT, но они не могут использовать бренд UNICS — он отдан Беркли. Они делают изящный ход: появляется **UNIX**. Фонетически читается одинаково, но юридически — другое название.

От UNIX появляются коммерческие ветки: - **HP-UX** (Hewlett-Packard) - **AIX** (IBM) - **IRIX** (Silicon Graphics)

1983 год — SystemV и Ричард Столлман

Появляется важный форк — **SystemV (SysV)**, берущий начало от BSD.

В то время царил **патентный беспредел**: можно было взять чужую разработку, добавить копеечную деталь, запатентовать — и подать в суд на оригинального автора.

В этой атмосфере **Ричард Столлман** начинает свой крестовый поход против «акул бизнеса».

Он создаёт **манифест свободного ПО** с 4 принципами (и как истинный С-программист начинает нумерацию с нуля): 0. Свобода **запускать** программу (классически — копировать). 1. Свобода **распространять** копии. 2. Свобода **изучать**, как программа устроена (открытый код). 3. Свобода **улучшать** программу и создавать на её основе новые произведения.

Столлман придумывает концепцию **свободной лицензии**.

Знак **Copyright** обозначает, что копия легальна и разрешена автором именно как **экземпляр**, но дальше копировать нельзя. Купил книгу Гарри Поттера, купил принтер — но если будешь печатать копии, автору не заплатишь.

Столлман придумывает **Copyleft** — перевёрнутый копирайт, который работает с точностью до наоборот.

Но пока это юридические приколы. Нужна ОС, написанная под лицензией Copyleft. Для этого нужно писать всё с нуля. Столлман запускает проект **GNU** — рекурсивный акроним «GNU is Not Unix».

В то время уже все писали на С. Команда Столлмана переписывает свободный компилятор С — получается **gcc** (GNU C Compiler) под лицензией GPL. Но у Столл-

мана нет экспертизы написать своё ядро, и людей ему не дают.

Студент, который бросил вызов

Хельсинкский университет (тот самый, который Пушкин назвал «приютом убогого чухонца»). Один студент изучает **Minix** — учебную ОС, написанную профессором Танненбаумом, — но она ему не нравится. Он решает написать свою ОС, во многом отрицая принципы Танненбаума.

В одиночку написать универсальное решение нереально — он делает ставку на конкретную архитектуру, **Intel 8086** (та самая, которая станет x86).

В это время уже появляются первые сети: FidoNet, Gopher (суслик, роющий норки). Люди подключаются к этому студенту, помогая с кодом. Слухи доходят до MIT и до самого Танненбаума.

Появляется провокационная тема — **«Linux устарел»**. Студентика как раз звали **Линус Торвальдс**. Линус пишет в ответ жёсткое письмо, отстаивая свои принципы. Завязывается переписка в духе «Да кто ты такой?».

Столлман приходит к Линусу: «давай возьмём твой код, твоё монолитное ядро. Но проект будет называться GNU». Линус соглашается, но с условием: чтобы в названии было его имя. Так появляется **GNU/Linux**.

Компания **RedHat** начала распространять свою версию Linux по подписке. Позже на основе исходного кода Red Hat появляется **CentOS** — оттуда убрали всё проприетарное, оставив полностью свободную систему. Сообщество CentOS позже погибло, но есть продолжатели дела.

1989 год — NeXT, Apple, Microsoft

К концу 80-х появляется много ОС с исходным кодом. Параллельно появляются частные бизнесы, собирающие свои конфигурации и дописывающие ПО под конкретные ниши.

Появляется компания **NeXT** (XT — отсылка к известным в те времена XT-платформам). Они создают ОС **NeXTSTEP**. 8 лет компания худо-бедно существовала. В **1997 году** их выкупил гигант…

Персональные компьютеры были очень дорогими. Возникает нужда в ПК для домохозяйки или дизайнера — не для программиста и математика. Появляются

мощные программы для дизайнеров, но дизайнер — антиматематик: либо ты технарь, либо творческий человек.

Дизайнерам Linux не подходил — командная строка, белые буковки на чёрном фоне, не для них. Нужна ОС, работающая «из коробки». Писать её с нуля — нет. Бизнес-идея: купить готовое, ребрендировать, повесить знаменитое обкусанное яблоко. Этим гигантом и был **Apple**. Из NeXTSTEP появилась свободная часть Apple — **Darwin** (юридический шаг, чтобы обходить лицензии). На основе всего этого появилась **macOS**.

The Windows

История Windows начинается иначе. Она создавалась с нуля концептуально. **MS-DOS** изначально рассчитывалась под домашнее использование: пусть менее функциональна, но удобна для обычных людей. Нересурсоёмкие задачи (текст, сёрфинг), удобный интерфейс. Это произвело фурор. Microsoft начала вкладываться в Entertainment.

Сначала Windows был надстройкой над MS-DOS. В Windows 2.0 не было пересечения окон. В Windows 3.0 — пересекающиеся окна. Позже — поддержка сети.

Дальше Microsoft принимает решение сделать графический интерфейс **основным** и интегрировать ядро с GUI (Linux на это не пошёл). Появляется **Windows 95** — но это была «подделка»: интеграция была формально, реально все вызовы шли в ядро MS-DOS. Это было нужно, чтобы обкатать идею на фокус-группе. Появилась версия Windows XP и далее…

> *Добро пожаловать в современность* □♂ □ □ □

Нормальная компания живёт от кодовой базы. Твой накопленный код — это твоё богатство. Время уникальных решений прошло: уже не засесть в гараже и не написать что-то уникальное — всё когда-то уже было написано.

> **Архитектор Гауди в Барселоне** строил дома без чётких границ, все грани были скруглёнными. Собор Саграда Фамилия он не успел достроить — и никто не может: невозможно повторить его подход к работе.

То же и в IT: если не использовать архитектурные паттерны, решение может быть крутым и ценным, но не будет способно развиваться.

> **Карл Росси** — Аничков дворец устойчив к износу, окна не запотевают, летом прохладно, зимой тепло. Реставрация почти не нужна. Гениальный человек, удерживавший все аспекты в голове. Но таких людей — единицы.

В большом здании нужно провести вентиляцию, водопровод, электрику. Стык труб у главного электрощитка — не лучшая идея. Один специалист не может быть экспертом по всем дисциплинам одновременно.

Бурдж Халифа. Как с высоты 540 м смыть канализацию? Как обеспечить воду в кране? Обычное решение — труба огромного диаметра под огромным давлением. Там много необычных уникальных решений.

То же возникает и в программировании.

Виды архитектуры

- **Функциональная** — с позиции пользователя. Что именно пользователь хочет видеть, где, как и почему. Ему безразличны паттерны и сложности разработки.
- **Информационная** — информационные потоки, информационные объекты, процессы. Какие объекты выделяем для функциональности; как удаляем, добавляем, соединяем.
- **Системная** — с позиции инженерии. Middleware-слои: паравиртуализация, контейнеризация. Разделяем функционал с точки зрения безопасности, стабильности, надёжности. Программные, пользовательские, аппаратные интерфейсы.
- **Программная** — как организован код. Паттерны, фреймворки. Как реализован код, чтобы его можно было масштабировать. *(Разработчики ОС не любят ООП.)*

- **Архитектура данных** — информационные объекты в виде структур данных, связи между ними.

Цель существования ОС

(Тут плохой перевод: *goal* — цель как конкретика, *aim* — цель глобально. Мы говорим про **aim**.)

- **Производительность**
- **Надёжность**
- **Безопасность**

Всё это нужно обеспечить для четырёх объектов: **Software, Hardware, Data, Interfaces**.

Аналогично БД: добавляем ACID — падает производительность. Добавляем безопасность — падает надёжность и производительность. А здесь объектов 4, и все 3 свойства для каждого должны быть обеспечены.

Потеря сгоревшей железки — ерунда, просто затрата. Еврейская мудрость, которой поделился преподаватель: **«Если проблему можно решить деньгами — это не проблема, а затрата».**

Но потеря данных может быть фатальной.

> **Пример из аэропорта Хитроу (Лондон).** Накатили обновление ПО — там была ошибка в непокрытом тестами фрагменте. У админа была некорректно настроена система бэкапов — они затирали друг друга. Решили быстро откатиться — бэкап оказался поломан. Код откатили, но данные не восстановить. А данные были — настройки линий багажа, регистрации. Никто не работал десятки лет без этой системы. Аэропорт почти на 3 суток выбыл из нормальной работы. Потери — 6-значные суммы фунтов стерлингов.

Функциональная архитектура

Выделим 4 метафункции и в каждой — более мелкие функции.

1. Управление исполнением и разработкой пользовательского ПО

- API

- Управление исполнением
- Обнаружение и исправление ошибок
- Организация ввода/вывода (I/O)
- Хранилище данных
- Мониторинг ресурсов

API. Не зря Керниган и Ричи 6 лет страдали и создавали основы UNIX — интеграцию компилятора и ядра. Раньше API не было. Вы пишете `fopen()` — и открывается дескриптор. Вы не задумываетесь, почему файл открылся, какая файловая система, как проверялись права доступа. Вы абстрагированы. `fopen()` дёрнула библиотеку, библиотека — API ядра, ядро дальше делает своё.

Управление исполнением. Нажал на ярлычок — волшебным образом запустился браузер. Код попал в память, отрисовалось окошко, процесс получил приоритеты, встал в очереди. Очереди оптимизированы под нового участника. И мы за этим не наблюдаем — как ребёнок не задумывается, как еда попадает в холодильник.

Обнаружение и исправление ошибок. Раньше при делении на 0 — чёрный экран, нет сигналов с клавиатуры; непонятно, ошибка это или вечный цикл. Чтобы закрыть — выдернуть из розетки. В 2000 строк кода для поиска ошибки нужно было написать сотни строк отладочного вывода. Сейчас IDE укажет на конкретную строчку. О том, как это реализовано, поговорим позже — но сложная логика: чтобы вывести информацию о падении на делении на 0, нужно сформировать её в видеопамяти и вывести. Как это сделать, если процессор «стоит»? Существует исключительное состояние процессора.

I/O. Разная скорость шин, разные протоколы. Воткнули принтер — он начал печатать. Под капотом — стек драйверов. Ввод/вывод должен быть **буферизуем**, потому что он асинхронен и нагрузка неравномерна. *(Фильтр Калмана — штука, сглаживающая неравномерности процесса во времени, узнаем о нём на матане/статистике.)*

У нас все железяки физические — на них действуют законы физики. Проезжающий трамвай по заледеневшим проводам может вызвать искру, электромагнитный импульс — и битик при передаче испортится. Это нужно проверять.

Хранилище данных. Эффективно записывать и доставать данные. С учётом контроля доступа и требований безопасности — файловые системы и т. п.

Мониторинг ресурсов. Запустил приложение — медленно работает. *Compilation*

Error легко пофиксить, *Runtime Error* — тоже. Но что если программа работает без ошибок, но всё лежит? Как `my.itmo`: процессор загружен на 100%, памяти много, сетевых запросов толком нет. Активность есть, результатов нет — например, **spin lock**, опрос одной и той же переменной. Инструменты: `top`, `htop`, сетевые мониторы. С помощью Grafana сопоставляем графики во времени и находим, что проблема не там, где казалось.

2. Оптимизация использования ресурсов

- Решение многокритериальных задач

С самого начала ОС должна оптимально использовать аппаратные ресурсы. Цена железки уже не главное — 2-5 тысяч за blade-сервер не дорого. Дороже **электричество**: вы привели 2 киловатта и должны их ещё **отвести** (КПД компьютера ничтожно — 99.5% энергии уходит в тепло). Сейчас лучшие умы стараются хоть немного поднять КПД.

ЦОД в Гренландии — всё хорошо. А в Дубае? Нормальная температура внутри ЦОД ~18 °C. Нагреть легко, охладить тяжело — и охлаждение может стоить дороже самого ЦОДа.

Человек, заплативший за blade-сервер, хочет, чтобы он работал на 100%.

Решение многокритериальных задач. Критерии противоречат друг другу. Как это решить?

1-й подход — суперкритерий. Вводим весовые коэффициенты:

$$\hat{k} = \alpha k_1 + \beta k_2 + \gamma k_3,\ \alpha + \beta + \gamma = 1$$

Есть понятие **TCO (Total Cost of Ownership)** — стоимость владения, можем выражать $\hat{k}$ в деньгах. Этим занимается эконометрика. Если бы функция была гладкой и всюду дифференцируемой — это было бы красивой математической задачей. Но в реальности приходится решать численными методами.

> **Boeing летает под Windows LC.** Каждый самолёт индивидуален. Лётчик-испытатель составляет огромную инструкцию для конкретного борта. Угол открытия закрылков зависит от десятков параметров и должен высчитываться за секунду. Это делает Windows. А что если она в этот момент решит обновиться или дефрагментировать память? — Никакого расчёта за секунду не выйдет.

2-й подход — цикл Деминга:

Коэффициенты высчитываются на конкретное время. - **P (Plan)** — спланировали на ближайшее время. - **D (Do)** — выполнение плана. - **C (Check)** — план и факт сравниваются. Если что-то изменилось — обнаруживается расхождение. - **A (Act)** — устранение расхождения (например, поднять приоритет процессу, недополучившему ресурсов).

3. Поддержка эксплуатации

- Диагностика
- Восстановление

ОС очень сложна и, как любая система, работающая в открытом мире, будет ломаться. Совершенное ПО невероятно дорого. Если ошибка происходит с вероятностью 0.01%, экономически нецелесообразно с ней бороться.

4. Поддержка развития ОС

- Обновление
- Изменение конфигурации

Цикл создания ОС от каркаса до релиза — около 4 лет. Огромная кодовая база. Время эксплуатации — по 20 лет. До сих пор есть машины на Windows XP, FreeBSD 5–6. Будут находиться ошибки и уязвимости, появляться новые требования. На живой работающей машине поменять что-то и сохранить целостность — очень тяжело.

Ставим сервер под веб — со временем понимаем, что не тянет, и он становится доменным сервером. Профиль нагрузки меняется — нужно уметь отключать модули и включать другие в реальном времени.

На прошлом занятии разобрали функциональную архитектуру.

Информационная архитектура

1. **Управление процессами**
   - Дескриптор процесса PCB (Process Control Block)
     - Идентификация
     - Ресурсы
     - История
   - Очереди планирования
2. **Управление памятью**
   - Виртуальная память
   - Защита памяти
3. **Управление файлами**
   - Карта размещения блоков (в памяти)
   - Каталог имён
4. **Управление внешними устройствами**
   - Драйверы
   - Plug & Play
5. **Защита данных и администрирование**
   - БД для идентификации, аутентификации, авторизации
   - Аудит (журнал)
6. **Пользовательский интерфейс**
   - CLI
   - GUI

Компьютер кроме как с информацией ни с чем работать не может. Любой физический объект должен быть представлен как **информационный объект**.

Подсистема управления процессами

Нас интересует не сам код приложения — код статичен. А вот **запущенная программа** — живая: запрашивает память, открывает сокеты, общается с другими процессами. У неё всё время меняется нагрузка. Она может уйти в сон или активизироваться по приходу данных.

Поэтому появилось понятие **процесса**.

> **Процесс** — это программа в момент выполнения, от момента запуска до завершения. Как спектакль: пьеса (код) одна, а каждый спектакль

(процесс) — разный.

PCB (Process Control Block) Процесс существует в системе, пока существует его дескриптор. Удалили PCB — процесс умер. В Linux максимум **65536 процессов** (2^{16}) — это и техническое ограничение, и безопасность: чтобы один пользователь не мог создать миллион процессов и положить систему.

Что внутри PCB? 1. Идентификаторы - В любой ОС можно получить PID — Process ID. - В Linux есть Parent PID — древовидная структура процессов. - В Windows тоже есть Parent PID, но он может не использоваться. - Идентификатором может быть UID пользователя. - В Linux есть ещё **эффективный UID** (Effective UID).

2. Ресурсы Подкаталог `FD` (File Descriptors) — ссылки на объекты типа файлов. В нём всегда есть 3 стандартных: - `0` — stdin - `1` — stdout - `2` — stderr

В этом и заключается прозрачность Linux. Когда пишем `cat /proc/.../status`, самого файла `status` не существует — `cat` делает запрос к ядру, ядро достаёт свою структуру, преобразует в текст и возвращает текст.

3. История Используется для статистических алгоритмов. Если процессор работал так последние 10 секунд — предположим, и дальше будет работать так же. Происходит **предсказание** поведения процесса.

Очереди планирования Если на ресурс претендуют несколько процессов: - Можем реализовать систему супермаркета — кто первый, тот и получает. - Можем делать осмысленно — пропускать вперёд тех, у кого мало «покупок».

- **Планировщик Linux** — красно-чёрное дерево.
- **Планировщик Windows** — двумерный массив.

Это не значит, что один плохо, а другой хорошо. У них разные алгоритмы. Для ввода/вывода — тоже свои очереди.

Управление памятью

Виртуальная память. В момент компиляции вы не знаете реальных свободных физических адресов. Нужно подменять адреса. Хранить маппинг виртуальных и физических адресов для каждого процесса нужно — но это кушает память, которую мы и хотим сэкономить. **TLB-кэш**: можно сделать эффективно по памяти, но

вычисление адресов будет медленным. Либо наоборот.

Защита памяти. Нужно где-то хранить сведения «свой/чужой». Нужно «окрасить» все объекты памяти цветами по количеству процессов и проверять цвет при доступе. Легко красить большими кусками памяти, но тогда — проблема дефрагментации. Как хранить? Линейный массив? Об этом — позже.

Управление файлами

На хранилище раньше были человекочитаемые данные, на каком-то этапе перешли к машиночитаемым.

Фон Нейман сказал, что ОЗУ линейно адресуемо. Но хранилище — как правило, многомерное. У HDD — **3-мерная цилиндрическая система координат** (номер пластины, номер дорожки, номер сектора). Это нужно отображать в линейную систему. А на SSD адресация уже двумерная — и мы можем перетянуть файл с 3D-HDD на 2D-SSD просто мышкой.

Карта размещения блоков и каталог имён 1 файл — 1 каталог (родитель) в традиционных ФС.

А в Linux можно создать файл, принадлежащий нескольким каталогам одновременно — это пощупаем в 3-й лабе. Получается не дерево, а **направленный граф**.

В Linux любой каталог — это файл. Поэтому в Linux нельзя создать файл с тем же именем, что и каталог рядом.

> В Linux защиты от дурака нет: «если ты root — ты крут».

Команда `ln` создаёт жёсткую ссылку. На каталог жёсткую ссылку создать нельзя — даже под root'ом будет `access denied`. Только со специальным флагом и под root'ом можно, но в `man` к этому флагу — огромный дисклеймер: «все ошибки человечества после этого будут на твоей совести». Это сделано, чтобы избежать **бесконечного цикла из ссылок**.

Управление внешними устройствами

Драйвер (дословно — «водитель, управляющий»). Работа с железом — привилегия ядра.

Как ядро узнает, что вы подключили веб-камеру? Производитель должен написать код под конкретную ОС, который будет работать с камерой. Нужны: - **про-**

токол взаимодействия, - **система команд**.

Программисты драйверов должны знать спецификацию ядра.

Семейства драйверов — плохо: в них страдает параметризация. Скачиваешь общий драйвер для линейки принтеров Brothers — он берёт минимальные общие параметры (минимальная скорость, без переворачивания листов), даже если у вас всё это есть.

Раньше (80-е, конец 90-х) драйверы вручную линковались внутри кода ядра, после чего нужно было пересобирать ядро. Пару раз упадёшь — и установка одного драйвера растягивалась на несколько суток. Любой сисадмин обязан был знать C.

Дальше Microsoft (которому надо было продавать ОС домохозяйкам, не знающим C) развивает игровую индустрию, делает упор на железо, и появляется **Plug & Play**. Консорциум производителей договорился: при подключении устройство сообщает свой UID; по UID находим драйвер в базе/облаке/на диске, и автоматический механизм интегрирует драйвер с ядром. Драйверы умели вешаться на универсальные интерфейсы. Но всё равно для защищённой области памяти нужна была перезагрузка.

> Работало это всё криво — администраторы называли систему **Plug & Pray**.

Защита данных и администрирование

БД для аутентификации. - Идентификация — сохранили хэш. - Аутентификация — проверили хэш. - Авторизация — сопоставить логического пользователя (UID) и ресурс (файл, процесс).

Ресурсов много, пользователей много. Матрица доступа — огромная и **разреженная**. Алгоритмы на разреженных матрицах обычно неэффективны — $O(n^2)$.

- У **Linux** — **мандатно-дискреционный доступ**. При первом обращении он долго «думает», дальше проверяется наличие мандата.
- У **Microsoft** — **Active Directory**.

Аудит (журнал). ОС живёт в открытом мире, среди пользователей с **естественным интеллектом**. Они всегда победят искусственный — у них есть фантазия. ИИ может обобщать чужие фантазии, но придумать действительно новую атаку — нет. «Генерал всегда готовится к прошедшей войне».

Аудит позволяет посмотреть, как это происходило, кто что сделал. Можно ловить аномалии — например, пользователь поменял пароль 42 раза за 20 минут. Возможно, паранойя; а возможно — перебирает и сохраняет результаты хэш-функции.

Пользовательский интерфейс

CLI — Command Line Interface.

GUI. Нельзя запускать с root'а (точнее, можно, но с дисклеймером «не валяй дурака»). Нужно запускать от обычного пользователя. Но GUI должен взаимодействовать с ядром — иначе через SSH нельзя было бы перезагрузить сервер. Используется механизм с битом **SUID**.

> На сервера никогда не ставят GUI.

Windows и современная macOS имеют сильно интегрированный GUI.

Системная архитектура

Код в **режиме ядра резидентен** — находится в оперативной памяти по фиксированным адресам. Виртуализация памяти ему не нужна, код ядра минует маппинг виртуальных адресов — это сильно быстрее.

Если весь функционал ОС запихать в ядро — всё будет очень быстро, но память сожрётся. И не факт, что весь код нужен в моменте. Если в ядре оставить только часто вызываемое — производительность и безопасность плачут.

Что такое надёжность? Во время тестов — просто всё проверить. Обратная сторона монолита — **микроядро**. Протестировать 10 тысяч строк микроядра легко.

5 принципов разработки ОС

1. **Модульная организация.** Даже если это монолит — это не помойка кода. Любой аллокатор памяти, планировщик — это модуль. Помогает независимо разрабатывать и планировать высокоуровневую архитектуру.
2. **Функциональная избыточность.** Проектируем то, чем будут пользоваться не только сейчас, но и потом. > Когда проектировали файловую систему **ext2**, заложили возможность адресовать до 2 ТБ. В начале 90-х это казалось избыточным — диски были по 10 МБ. Но к концу 2000-х 2 ТБ стало нормой.

Беда была в другом — нельзя было создать большой файл, потому что поле для размера было ограничено.

3. **Функциональная избирательность.** Должен быть механизм отключения излишней функциональности. > Windows XP рассчитывалась на 256 МБ ОЗУ, но медиана была 128 МБ. ОС кушала много памяти. В сети быстро появились «колотушки» — твики regedit, отключавшие ненужное.
4. **Параметрическая универсальность.** Не хардкодить константы (no magic numbers).
5. **Концепция многоуровневой вычислительной системы.** Даже минимальный монолит имеет 3 слоя. Нельзя в один компонент заложить и аппаратно-зависимое, и программное — иначе любое изменение потребует переделывать всё. Нужно абстрагировать аппаратную часть от программной.

Ядро всегда работает в привилегированном режиме. Оно **резидентно** — находится в физической памяти, минуя виртуальную.

Но если всё загнать в ядро — оно будет весить очень много. Нужно искать компромисс.

1. Монолитная архитектура

Любая процедура в любой момент может вызывать другую. Удобно: любой компонент может уйти в любой другой. Тем не менее, должна быть иерархия — иначе доработка системы будет сложной.

Выделили 3 слоя: 1. **Main program** — интерфейс к пользовательскому ПО (Software). 2. **Services** 3. **Utilities**

> **Системный вызов** — обращение пользовательского приложения к ядру операционной системы с целью попросить/предоставить ресурс или выполнить привилегированную операцию.

Как это работает: ядро найдёт PID процесса, достанет SP (stack pointer), возьмёт там вектор параметров, перенесёт это в ядро и выполнит системный вызов. Результат (параметры) упаковываются и возвращаются в SP. Выполняется команда выхода из привилегированного режима, включается защита памяти.

Логика одна и та же независимо от архитектуры ядра — возможна разница только в работе со стеком. Когда системных вызовов много, `main` становится перегруженным. И хочется, чтобы системные вызовы были обёрнуты удобной библиотекой.

Этот слой из процессов вырос в **API** — множество интерфейсов.

2. Многослойная монолитная архитектура

Она остаётся монолитной (всё в ядре), но слоёв больше:

1. **Hardware**
2. **Слой аппаратной поддержки ядра**
3. **Машинно-зависимые модули (HAL — Hardware Abstraction Layer)**

> **Чипсет** — то, что обеспечивает вашу совместимость или несовместимость с ОС.

Вызов запроса кванта непрерывной памяти и подобные операции идут в слой «Базовые механизмы ядра». **Базовые механизмы ядра** супероптимизированы,

написаны на ассемблере. **Менеджеры ресурсов** максимально оптимизированы алгоритмически (с использованием стохастики).

3. Микроядерная архитектура

Одна из концепций, которую продвигал **Танненбаум**.

`APP1` сделает системный вызов в ядро. Системный вызов вызовет `API2`, тот вернёт результат, ядро вернёт результат `APP1`. `APP1` обработает результат, запросит ещё что-то — и снова системный вызов…

Теряем скорость работы ядра, но **освобождаем память**, в которой может лежать пользовательская программа.

Windows — это среднее между двумя мирами (гибрид).

4. Наноядерная архитектура

Работают специфические ОС, которые называются **гипервизорами**. Например, при покупке виртуального сервера в AWS или Yandex Cloud.

Гипервизор имеет очень мало драйверов — только основные (для памяти, для дисков). Всё остальное вынесено в ОС, работающие на уровне usermode. Нет сложных алгоритмов выделения памяти.

5. Экоядерная архитектура

Та архитектура, где Hardware может всё время меняться. В ядро закладываем менеджеров и защищаем их — не даём ошибкам железа повлиять на ядро.

Модульное ядро. Гибридные ядра

Гибридные ядра нужны в основном для научных задач — тестирование и отладка планировщиков и подобное.

Любой системный вызов выполняется в рамках процесса.

> **Процесс** — набор исполняемых команд, ассоциированных с ним ресурсов и контекста исполнения, находящийся под управлением операционной системы.

Если в цикле запускаете `grep` — запускается один и тот же код, но у каждого процесса свои дескрипторы ввода/вывода.

Сигналы

В рамках концепции привилегированного доступа выдали возможность подавать процессам **сигналы**. Когда подаёте `kill`, команда не доставляет сигнал процессу напрямую — она отправляется ядру. Сигналы `SIGKILL (9)` и `SIGSTOP (19)` **нельзя перехватить** — это обеспечивается ядром. Иначе можно было бы написать вредоносный код, который нельзя завершить.

С помощью `renice` можно поменять приоритет процесса. Но в современных планировщиках это работает иначе, чем в старых.

Многопоточность

Появилась многопоточность: - **Физическая многоядерность** (cores) - **Виртуальные потоки** (threads, hyper-threading)

Есть процесс, выполняющийся на конкретном ядре. Например, осветляет картинку: к каждому пикселю прибавляет константу. Остальные ядра простаивают. С точки зрения алгоритма можно сделать многопоточное осветление, но нужно: - (а) расшарить память и обеспечить целостность данных (что сложно из-за разной скорости процессов); - (б) разрешить нескольким потокам работать с одной памятью, ослабив защиту памяти.

> **Поток (thread)** — набор исполняемых команд и контекста исполнения, разделяющий все или часть ресурсов с другими потоками данного процесса и находящийся под управлением операционной системы.

Синхронизация потоков нужна, потому что приходится вручную контролировать целостность данных.

Пример: вычислить дробь — числитель и знаменатель в разных потоках, третий поток делит. В идеальном мире знаменатель должен быть готов раньше деления.

Но может оказаться, что числитель вычислен, а знаменатель ещё равен 0 — ловим деление на ноль.

Возвращение к кооперативной многозадачности — Fiber'ы

Маятник возвращается. Вернулись к тому, от чего уходили — к кооперативной многозадачности. Появились **горутины**, **fiber'ы** (волокно из нити Thread). Их называют ещё **Lightweight Thread**.

Идея: один настоящий поток может разложиться на несколько таких «легковесов». Thread может состоять из нескольких Fiber'ов (а может и не состоять, если технология не используется).

> **Fiber** — набор исполняемых команд в контексте конкретного потока, находящийся под управлением пользовательского приложения.

В Kotlin это реализовано с помощью конечных автоматов.

Распределение ресурсов между процессами

Пример: запущены IDEA и Chrome. Кажется, должно достаться поровну, 50/50. Но у Chrome 99 процессов, а у IDEA — 1. Ресурсы делятся как 99:1. Упс. С позиции ОС нет разницы, что они порождены одним процессом, — все процессы плюс-минус равны.

Основные функции подсистемы управления процессами

- Создание
- Обеспечение ресурсами
- Изоляция
- Диспетчеризация
- Планирование
- Синхронизация
- Межпроцессное взаимодействие
- Завершение

Сегодня — про **создание** и **завершение**. - *Обеспечение ресурсами* размазано: ресурсы разные, и для них разные планировщики. - *Изоляция* — это изоляция ОЗУ, ПЗУ, сети. - *Диспетчеризация* — процессы меняют состояния (один выполняется, потом спит; другой просыпается). Граф состояний приведёт к задаче планирования. - *Синхронизация* — процессы влияют друг на друга, в race condition

возникают критические ситуации. - *Межпроцессное взаимодействие* — это 3-я лаба.

Создание процесса

Любой процесс создаёт другой процесс. А как создаётся первый? В каждой ОС есть **костыль** для рождения первого процесса. В Unix процессы образуют строгое классическое **иерархическое дерево** — у каждого процесса ровно один родитель.

`d` в `systemd` — это **daemon** (system daemon).

Системные вызовы для создания процессов

- **`fork()`** — дословно «ответвиться». Создаёт копию текущего процесса.
- **`exec*()`** — заменяет текущий процесс новой программой.
- **`clone()`** — более тонкий контроль (используется для потоков в Linux).

Пример:

```
#!/bin/bash
cat file.txt
```

Bash породил потомка с именем `cat`.

> **Если зомби-апокалипсис привёл к тому, что закончились PID'ы — сервер зависнет настолько, что даже войти нельзя.** Открыть терминал — породить процесс. Даже если зашли, посмотреть процессы через `ps` — тоже процесс…

Диспетчер процессов родит процесс и вернёт параметры — одним из 18 параметров будет PID.

Диспетчеризация и клонирование процессов

Время дискретно — происходят события, которые меняют одно дискретное состояние на другое.

Развитие событий во времени описывается **автоматами**: - **Вершины** — возможные состояния процесса. Они не привязаны к конкретному времени; в каждый момент процесс находится в какой-то вершине. - **Рёбра** — возможные переходы между состояниями.

Двухсостоянийная модель

Исторически было 2 состояния: - **Running** (исполняется) — используется процессор. - **Waiting** (ожидает) — чего-то ждёт.

Ребро между ожиданием и исполнением становится **очередью**.

Из Running у процесса 2 выхода: завершиться или уйти в Waiting.

Пока стоял в очереди, операция I/O завершилась — он перейдёт в Running, когда подойдёт очередь. Но может быть и так, что очередь подошла, а I/O ещё не завершено — процесс снова уйдёт в Waiting. Это **непродуктивно**: дёргать процесс, который не может выполняться.

Трёхсостоянийная модель

Появилось состояние **Runnable** (готов).

Любой процесс попадает в **Runnable** (готовность). Мы всегда начинаем с вычислений (нужно запросить дескриптор и т. п.). Сразу обеспечить Running мы не можем.

Если процесс синхронно своему коду уходит в ожидание — он идёт в **Waiting**. В Waiting появляются свои очереди — как правило, к устройствам ввода/вывода. Это состояние называется **Direct Waiting** — процесс ожидает конкретное событие (семафор, I/O).

В какое состояние переходить из Waiting? Сразу в Running нельзя — будет конкуренция двух очередей. Значит, в **Runnable**.

Состояния процессов в Linux

`R` (Running) — процесс либо прямо сейчас исполняется, либо будет исполняться в ближайшее время.

`D` (Direct Waiting / Uninterruptible Sleep) — ожидает завершения конкретного события, которое выведет его в Runnable. Процесс сначала разгребёт накопившийся перечень событий — обработает сигналы, вызовет системные вызовы.

`S` (Sleeping) — процесс спит. Это, как правило, **демоны**: те же веб-серверы, если запросов нет, большую часть времени спят.

Чем S отличается от D? В обоих процесс ждёт. Но: - В **D** он ждёт конкретный сигнал. - В **S** он забуферизует любой сигнал и сразу переведёт его в Runnable.

Пример работы веб-сервера в S: 1. Пакет приходит, ядро парсит TCP-заголовок. 2. По дескрипторам портов определяется, кто слушает порт. 3. Слушающему процессу посылается WakeUp. 4. Процесс уходит в Runnable. 5. Обработчик читает с вершины стека и обрабатывает запрос. 6. После обработки — снова `Sleeping`.

`T` (Stopped/Terminated) — приостановлен. На жаргоне — «состояние комы»: процесс ставят на паузу.

Почему у «рождения» нет состояния?

Чтобы выдать состояние, нужно выдать PID. Пока PID нет — процесса нет в `ps aux`. Соответственно, состояние «рождение» — это и есть процесс создания.

Состояние «карантин» (Exception)

Если процесс поймал Exception, ядру придёт прерывание от чипсета. Это переполнение памяти, выполнение недоступной команды процессора и т. п.

Состояние «карантина» — что-то вроде второго шанса. Если процесс многопоточный, Linux даст ему ещё попытку. Может, это рассинхронизация потоков? Может, через 10 мс деление на ноль исчезнет — знаменатель досчитается? Может, забыли мьютекс? Может, `malloc` не успел вызваться? Стоит попробовать через 10–20 мс.

> **ОС очень гуманная.** Она никогда не убивает процесс — она **предлагает ему самоубиться**. Почему? Если ОС вдруг убьёт какую-нибудь транзакцию, какой-то дяденька потеряет шестизначную сумму долла-

ров. Такую систему никто не купит. **Дефолтные обработчики сигналов — это самоубийства.**

Метрики

- **Время ожидания** (непродуктивное время).
- **Время отклика** (для интерактивных процессов важно — смотрим что-то в реальном времени).

Что считать «хорошо»? Нужны граничные свойства алгоритма.

Свойства алгоритмов планирования

1. **Предсказуемость.** На похожих данных алгоритм должен давать похожие результаты. Например, время работы алгоритма. *Настоящие специалисты не используют O-нотацию — в реальности оценки превращаются в тыкву.* Пример: генетические задачи — мутации, время их схождения — случайная величина с большой дисперсией.
2. **Масштабируемость.** В современном мире для планировщиков $O(n)$ недопустимо, надо хотя бы $O(\log n)$. Был алгоритм планировщика с провокационным названием **O(1)** (где константа была 104). Его заменили на $O(\log n)$. Асимптотически $O(1)$ эффективнее, но в реальности $O(\log n)$ оказался лучше.
3. **Реализуемость.** Есть слепые тесты — алгоритм заменяется рандомайзером. Если рандомайзер лучше планировщика — зачем такой алгоритм? **Критерий истины — практика.** Накладные расходы нужно минимизировать.

Параметры планирования

Хотим выстроить план — для этого надо что-то знать о процессах. Есть n-мерное пространство — n-мерный куб, каждая сторона которого — параметр. Сводим задачу к поиску экстремума функции. Если бы функция была всюду дифференцируемой и непрерывной — задача была бы простой. Но это только в математике.

Параметры можно делить на: - **статические** vs **динамические**, - **системы** vs **процессы**.

- **Статические параметры системы**: характеристики железа.
- **Динамические параметры системы**: процент утилизации процессора, утилизации канала связи и т. д.
- **Статические параметры процесса**: права доступа (если процесс вовсе не может открывать сокет — не учитываем его в сетевом планировщике), и т. п.

- **Динамические параметры процесса**:
  - **CPU-Burst** — сколько процесс будет непрерывно работать на процессоре, пока не уйдёт в I/O (в тактах).
  - **I/O-Burst** — аналогично с I/O: через сколько вернётся.

2 класса алгоритмов

- **Вытесняющее планирование** — есть способ прервать процесс и передать ресурс следующему.
- **Невытесняющее планирование** — нет.

Алгоритмы планирования

1. FCFS (First-Come-First-Served) А что если перевернуть очередь?

Полное время сократилось в 2 раза, время ожидания — в 5 раз.

Справедливость говорит: FCFS. **Эффективность говорит**: пропустить вперёд «студента с водой». Но охранник не знает CPU-Burst процессов.

2. RR (Round-Robin) С теми же процессами:

Поставим квант $K = 1$:

Но накладные расходы будут колоссальными.

3. SJF (Shortest Job First)

4. Гарантированное планирование Пусть есть N процессов. Каждый i-й процесс характеризуется двумя величинами: - T_i — текущее полное время (время в системе); - τ_i — текущее время исполнения.

В идеале $\tau_i \approx \frac{T_i}{N}$. Вводим **коэффициент справедливости**:

$$R_i = \frac{\tau_i \cdot N}{T_i}$$

Тот, у кого R_i минимален, — самый обделённый. Вводим квант: пока процесс выполняется, числитель растёт только у него, знаменатель — у других. Через какое-то время самым обделённым станет уже кто-то другой. Так аккуратно и справедливо раздаётся процессорное время.

Сломалось две вещи: 1. R_i — вещественное число. Сортировка по вещественным числам сложна — сравнение двух вещественных чисел дорого, тем дороже, чем выше точность. 2. **Подверженность хакингу.** Студенты за либеральные ценности — только пока их не собираются отчислить. Что делали студенты в общаге? Заходил в 7 утра, запускал процесс-заглушку. До 12 утра дописывал реальный код. К моменту, когда код появлялся, τ_i маленькое, а знаменатель уже накопился — процесс получал максимальные ресурсы. Началась гонка таких программ.

> Через 40 лет к этой идее вернутся и докажут, что так делать всё-таки можно.

Многоуровневая очередь

Аналогия с управлением программой: - 4-курсник, пишущий диплом и уже стажирующийся. - 1-курсник, который ещё никак себя не показал. - Студент с 3 академами.

Как расставить рейтинг? Появился новый студент, надо вставить между 378-м и 379-м местом. Вещественные числа? Сдвигать всех начиная с 378? Возникает идея — **многоуровневая очередь**.

- Внутри очереди — Round-Robin.
- Процесс может исполняться только если нет процессов в более приоритетной очереди.

Со старых времён был компьютер с аптаймом 6 лет. Взяли дамп памяти — для изучения. В дампе планировщика был процесс, который так и не выполнился за много лет. Он был где-то на дне.

Появился алгоритм типа **«лифт»** — и **многоуровневая очередь с обратной связью**.

4 требования к планировщику (исходно противоречивые)

1. **Поддержка внешнего управления приоритетами.** Владелец сервера должен иметь возможность кому-то дать приоритет. Так или иначе придётся поддерживать многоуровневую очередь.
2. **Эффективность использования ресурсов.** Минимальное время простоя процессора (статус IDLE). Минимальное время «деструктивной работы». Процессы должны максимально быстро покидать память. Минимум переключений в режим ядра (смена контекста дорогая). Эффективно использовать кэш процессора (меньше тратить на перерасчёт маппинга виртуальной памяти).
3. **Минимизировать накладные расходы алгоритма.** Везде использовать целочисленную (а лучше битовую) арифметику. Хорошая асимптотика — $O(n)$ считается плохой, выше не рассматривается.
4. **Минимизировать риски тупиков. > Deadlock**: несколько процессов блокируют ресурсы — процесс А заблокировал ресурс R_1 и ждёт R_2, процесс В заблокировал R_2 и ждёт R_1. > > Другая ситуация: процесс стоит в очереди I и ждёт ресурс А. Ресурс А занят другим процессом, стоящим в очереди II и ждущим исполнения. С точки зрения компьютера всё хорошо — но они в бесконечном ожидании.

   Оказывается, есть и экономическая выгода от справедливости.

Планировщик Windows

Windows использует **32-уровневую очередь**.

- **Realtime-уровень** — системные процессы, которые не меняют очередь и никогда её не покидают.
- С 16 по 31 — системные процессы.
- С 1 по 15 — пользовательские.
- На одном уровне между процессами — Round-Robin.
- На **0-м уровне** — процесс, обнуляющий память.

Классы приоритетов процессов

Класс	Приоритет
Realtime	24
High	13
Above Normal	10
Normal	8
Below Normal	6
Idle	4

Зачем 32 очереди, если классов всего 6? Из-за **уровней насыщения потоков** (их 7):

Уровень насыщения	Дельта
TimeCritical	+15
Highest	+2
Above Normal	+1
Normal	±0
Below Normal	-1
Lowest	-2
Idle	-15

TimeCritical и **Idle** — краевые, экстренные. Выйти за уровни 1–15 нельзя.

Критерий эффективности Windows XP / 95 / 98 прогнозировал CPU Burst. Потом поняли, что современные процессы становятся всё более интерактивными — это уже не марковские цепи.

Стали **поощрять** «хорошие» процессы (наказывать не надо) — те, что быстрее уходят в I/O. Когда процесс возвращается из I/O, ему **поднимают очередь**. Величина надбавки — эмпирические коэффициенты, которые Microsoft не разглашает. Например: - дисковые операции — самая низкая надбавка (+1) — скорее всего, ты прочитал и пойдёшь читать ещё; - чтение из USB-порта — +2; - интерактивные режимы (звук, клавиатура) — нужно, чтобы речь была плавной, текст печатался плавно.

Квант: в Windows 7 — 12 тиков таймера.

Планировщик Linux

O(1) 140 многоуровневых очередей.

В Linux между процессами на одном уровне — **FIFO**.

Появляется вторая очередь — **Non-Active**, первая называется **Active**. Рано или поздно даже из самой приоритетной очереди все процессы окажутся в Non-Active. В этой очереди нет голодания. Когда очередь Active закончится — меняем указатели местами: Non-Active становится Active.

Но эту очередь можно обойти. Создаём интерактивный процесс — он поднимается на максимальный приоритет. Затем меняет поведение и в конце своего времени **создаёт новый процесс и умирает**. Новый процесс на том же уровне — и снова к концу времени создаёт нового и умирает. Так процесс может выполняться бесконечно.

Костыль: при создании процесса процессорное время делится **пополам** между родителем и потомком.

CFS (Completely Fair Scheduler)

> Одну из лучших реализаций сделал **Эндрю Малдар** (Ingo Molnár), закончил Будапештский университет ещё во времена Варшавского договора, когда университет был наполовину МГУ. Когда рухнул железный занавес, эмигрировал в США, попал в IBM/Red Hat. Его заметил Линус Торвальдс и доверил оптимизировать планировщик. Со времён университета он помнил идею Танненбаума и её критику.

Есть две характеристики: - **Execution time** — целочисленное время выполнения. - **Max execution time** — максимальное.

Но вставка в очередь долгая. Малдар ходил на алгосы — выбрал сбалансированное дерево, **Red-Black Tree**.

Всё хорошо, но где `nice`? 3 из 4 требований решены, но **первое — нет**. Решение: **`nice` влияет на течение времени.** При `nice = 0` 1 мс равно 1 мс. Чем приоритетнее процесс — тем быстрее течёт его время ожидания и медленнее — время выполнения.

Проблема возникает с **синхронизацией**.

Время дискретно — события могут «накладываться» по времени. Особенно при работе с **неразделяемыми ресурсами**.

> Печатаем на принтере, передаются страницы. Половина страницы передана — таймер щёлкнул, управление передалось другому процессу, который тоже хочет принтер, — и начал передавать другую страницу. Получили мешанину.

Аналогично — сетевой доступ: половина пакета от одного процесса, половина от другого. Нужна возможность **занимать ресурс одним процессом**.

4 условия корректной синхронизации

1. **Взаимоисключение (Mutual Exclusion).** Одновременно в критической секции относительно ресурса может находиться не более одного процесса.
2. **Прогресс.** Не должно быть ситуации: ресурс свободен, есть процесс, готовый его использовать, но реализация алгоритма этого не позволяет («светофорчик»).
3. **Отсутствие голодания.** Не должно быть ситуации, когда процесс неограниченно долго ожидает ресурс, передаваемый другим процессам.
4. **Отсутствие тупиков** (определение дадим дальше) — кольцевое ожидание ресурсов.

Попытки реализовать взаимоисключение

Переход в однопрограммный режим У процесса есть механизм, запрещающий прерывания, пока он не выполнит парную операцию. Используется внутри ядра, но в userspace — плохая идея.

Попытка 1: Замок (Lock) Введём `shared`-переменную:

```
shared int lock = 0;

Pi() {
    while (lock);     // ждём
    lock = 1;
    {critical_section};
```

```
    lock = 0;
}
```

Проблема: в момент между `while(lock)` и `lock = 1` может произойти прерывание, и два процесса окажутся в одной критической секции. Если процессы асинхронны — всё хорошо; если в **race condition** — это неизбежно.

```
shared int turn = 0;

Pi() {
    while (turn != i);
    {critical_section};
    turn = 1 - i;
}
```

Попытка 2: Строгое чередование

```
shared int ready[2] = {0, 0};

Pi() {
    ready[i] = 1;
    while (ready[1-i] == 1);
    {critical_section};
    ready[i] = 0;
}
```

Попытка 3: Флаги готовности Один процесс поднял свой флаг, не успев проверить чужой. Другой — то же самое. **Оба в тупике** — нарушено условие прогресса.

```
shared int ready[2] = {0, 0};
shared int turn;

Pi() {
    ready[i] = 1;
```

```
    turn = 1 - i;
    while (ready[1-i] == 1 && turn != i);
    {critical_section};
    ready[i] = 0;
}
```

Попытка 4: Алгоритм Петерсона

```
shared int lock = 0;

Pi() {
    while (test_and_set(&lock));   // атомарно
    {critical_section};
    lock = 0;
}
```

Попытка 5: Аппаратная поддержка — Spinlock

Семафоры

```
Semaphore S;

P(s):  while (s == 0)
           block(process);
       unblock(process);
       s = s - 1;

V(s):  s = s + 1;
```

Классическая задача: Producer-Consumer

Два процесса: **producer** (производитель) и **consumer** (потребитель).

```
Semaphore mutex = 1;
Semaphore empty = N;
Semaphore full  = 0;
```

```
Producer() {
    while (true) {
        produce_data;
        P(empty);
        P(mutex);
        put_data;
        V(mutex);
        V(full);
    }
}

Consumer() {
    while (true) {
        P(full);
        P(mutex);
        get_data;
        V(mutex);
        V(empty);
        consume_data;
    }
}
```

Отсутствие прогресса — ресурс доступен, но никто им не пользуется.

Но может быть и другая ситуация — процессов много, ресурсов мало. Возникает **голодание**.

Проблема читателей/писателей

Характерна для очередей.

- Если пришла операция чтения — пусть читает параллельно.
- Если операция записи — монополизирует устройство.

Где ломается логика? - p_0 начал чтение. - p_2 начал ждать конца чтения, чтобы записать. - p_1 начал чтение, пока p_0 ещё читал.

Решение: разрешаем параллельное чтение только когда в очереди **нет ожидающих записи**.

SLA (Service Level Agreement)

Оцениваем время ожидания r.

> У проблем синхронизации **нет эффективных решений с гарантиями** — есть только статистические алгоритмы.

Тупики (Deadlock)

> Множество процессов находится в состоянии тупика, если каждый из этих процессов ожидает событие, которое может быть вызвано только другим процессом из этого множества.

Как бороться?

Дейкстра и тут решил — точнее, попытался.

Гегель вывел познание как 3 этапа: 1. Тезис 2. Антитезис 3. Синтез

Но для этого нужен оппонент. Дейкстра придумал модель, понятную на уровне здравого смысла, которая описывала бы проблему тупика и помогала бы её решить.

Проблема обедающих философов

- За круглым столом сидят 5 философов. Между каждой парой соседей — 1 вилка.
- Философы либо размышляют, либо едят.
- В тарелках — нескончаемые спагетти.
- Чтобы есть, нужно **обе вилки** (спагетти скользкие).

Три состояния: - **Размышление** (продуктивное, тратит жизненную энергию). - **Голод** (сил размышлять нет). - **Насыщение** → возврат к размышлению.

Философы не хотят разговаривать друг с другом. Нужны правила.

Стратегия 1: Левая → правая → ешь → положи

1. Проголодался — возьми левую вилку.
    - Не удалось — жди.
    - Удалось — возьми правую.
        - Не удалось — жди.
        - Удалось — ешь, потом положи обе.

Проблема: если все одновременно проголодались, все взяли левую, никто не может взять правую — **deadlock**.

Стратегия 2: Если правую не взять — положи левую и попробуй снова Все одновременно взяли левую, никто не взял правую, все положили левую — и снова взяли левую…

Появляется livelock — система подаёт признаки жизни, но не прогрессирует. По мониторингам не виден.

Стратегия 3: Положить левую и подождать случайное время Берём левую, пробуем правую — не получилось, кладём левую и ждём случайное время. Кому-то да повезёт.

Но всё равно есть комбинация времени, при которой получается livelock.

> Случайная последовательность генерируется алгоритмом — а **детерминированный алгоритм не может выдать настоящее случайное число**.

Возвращаемся к проблеме тупиков. Нужно либо разрешить философам общаться, либо ввести **официанта**, выдающего вилки. Но официант — узкое место, ещё один планировщик. А что если ресурсов 10 тысяч? 10 тысяч планировщиков? Кажется, будет проблема с реализацией.

Дейкстра признал, что задача пока не имеет универсального решения.

Условия Коффмана

Спустя 5 лет вышла статья **Коффмана**, в которой была сформулирована теорема.

> **Тупик возникнет тогда и только тогда, когда одновременно выполнены 4 условия:**
>
> 1. **Mutual Exclusion** — любой ресурс, участвующий в тупике, должен быть неразделяемым.
> 2. **Hold and Wait** — процесс имеет право, удерживая ресурс, ожидать другой.
> 3. **No Preemption** — только сам процесс может отдать ресурс; забрать у него нельзя.
> 4. **Circular Wait** — каждый процесс ожидает событие, зависящее от другого процесса (кольцо).

Отсюда появились методы решения проблемы (и поиска livelock'ов). У Танненбаума описаны алгоритмы поиска тупиков, но они **только теоретические** — на практике не работают.

Нужно построить ОС, в которой гарантированно не будет выполнено хотя бы одно из условий.

Как ломать условия Коффмана

1. Mutual Exclusion (сделать ресурс разделяемым). Принтер — у него есть pooling, очередь печати. В офисе на 150 сотрудников: не отправлять сразу на печать, а **буферизовать** в очередь. Приложению сказать «всё, ты напечатал», а очередь сама разберётся. Условия: - неинтерактивность (нет обратной связи); - уверенность, что очередь не переполнится.

2. Hold and Wait. Двухфазная транзакция: сначала получаем **все** нужные ресурсы, потом работаем.

3. No Preemption. Возможность отнимать ресурсы.

4. Circular Wait. Пронумеровать ресурсы, требовать брать их в порядке возрастания номеров. Теоретически работает, но на практике упирается в сложность.

Снизить вероятность тупика можно, реализовав комбинацию этих подходов.

Три подхода к работе с тупиками

1. **Обнаруживать.**
2. **Предотвращать.**
3. **Игнорировать.** Снижаем вероятность и не гарантируем отсутствие тупиков.

   Весь инфобез держится на балансе защиты и вероятности атаки: с одной стороны — стоимость взлома и потенциальная прибыль атакующего; с другой — стоимость защиты.

То же и с тупиками в крупных онлайн-магазинах: у 1 клиента из 100 возникнет тупик — извинимся и дадим бонус на 300 рублей.

> **Случай Маятина**: за городом заказал доставку «Перекрёстка» на сумму свыше 10 тысяч (положена бесплатная доставка). Связь была плохой — добавил товар и сразу нажал «оформить». Полетели две транзакции: сначала пришло оформление, потом добавился товар. Сумма не пересчиталась. Написал в поддержку: «Всё конечно хорошо, но изоляцию транзакций в СУБД сделайте».
>
> **Случай с Google** — объявили конкурс на задачи для квантовых компьютеров: компьютер есть, а что на нём считать — непонятно.
>
> **Случай с Яндекс.Облаком** — начиная с 20 петабайт упираешься в алгоритмы: нельзя просто так взять и достать данные из ячейки — перебирать ячейки будет невероятно долго.