# Оси

Коченюк Анатолий

25 декабря 2021 г.

## 0.1 Введение

В современном мире не представить вычислительный узел без операционной системы.

**Пример.** Супермаркет с одной кассой и тремя покупателями:

- с водой, человек очень хочет пить
- корзинка на ужин
- тележка на неделю

Если пропустим вперёд парнишку, его вклад будет крайне маленький. Но он выйдет из магазина и проходящий может зайти увидев очередь из двух человек.

Если же выстроить наоборот, то новые покупатели не заходят заходить в ваш супермаркет.

Хочется иметь монополию на власть в смысле порядка очереди

Многие процессы связаны с социальными процессами.

**Определение 1.** Операционная система – базовое системное программное обеспечение, управляющее работой вычислительного узла и реализующее универсальный интерфейс между аппаратным обеспечением, программным обеспечение и пользователем.

базовое – первое, что появляется и последнее, что умирает

системное – само пользу конечному пользователю не приносит, но без него не работает прикладное

управляет – достигает некоторых целевых показателей

# Глава 1

# Этапы эволюции ПО

## 1.1 Программы-диспетчеры

Конец сороковых, компьютеры работают строго по архитектуре Фон-Неймана. Есть устройство ввода и вывода.

4 принципа:

- однородность памяти – код и данные в единой памяти
- адресность – оперативная память это линейно-адресуемое пространство и мы можем обратиться в любой момент к любой её ячейке.
- программное управление – программа представляет собой набор инструкций в память. Процессор по ходу своей работы поочерёдно берёт и на такте выполняет следующую инструкцию
- кодирования – всё, и данные, и инструкции, кодируются с помощью двоичного кода.

**Задача 1** (Повторное использование кода. Автоматизация загрузки и линковки)**.** В те времена комп часто использовался для физических вычислений. Операторы заметили, что заново вводят одни и те же инструкции по многу раз. А почему бы не попробовать вынести куда-то.

Идея: вынести “подпрограмму” в конкретный адрес. МЫ переходим в него из своего кода.. но как вернутся, а ещё передать что-то хочется.

Появляются диспетчеры, которые управляют таким выводом

**Задача 2** (Оптимизация взаимодействия с устройствами ввода-вывода)**.** В классической архитектуре все устройства работают с памятью через процессор. Задача подкачки довольно простая, для неё не нужно всей мощности процессора.

Задача: осветлить картинку. Задача, которую можно делать незавсимо для всех пикселей

Идея: контроллер – связан с памятью, RAM и частично с процессором.

Предсказать время подкачки нельзя, потому что совершенно разные носители с разными гарантиями. Код не может знать подкачались ли уже данные или нет. А хочется уметь на это реагировать, т.е. ждать пока завершится процесс подкачки.

Обработчик прерываний – меняет значение флага, опрашиваемый в бесконечном цикле, пока контроллер не даст флаг, что всё сделано. Функционал диспетчера разрастется, теперь умеет обрабатывать прерывания.

SPOOL – simultaneous operation online

**Определение 2.** Перывание – сигнал, поступающий от внешнего устройства к центральному процессору, приостанавливающей исполнение текущего потока команд и передающий управление обработчике прерываний.

**Задача 3** (Однопрограммная пакетная <..>)**.** Одно приложение – много программ. Кроме того может ещё набор констант.

Появляется термин пакет – как совокупность программ.

Пока исполняем одну, можем загружать другую или даже другие и тут возникает вопрос: вот мы завершли один процесс, а какой исполнять следующим. Возвращаемся к задаче про супермаркет. Если взять идею пропускать маленького и реализовать её вот так втупую, то мы рискуем попасть в программное голодание, когда постоянно подгружается что-то маленькое и проходит вперёд.

## 1.2 Мультипрограммные операционные системы

Что привело к их появлению: программы становятся более сложными и разнообразными. Неэффективно используем ресурсы, потому что исполняем программы от начала до конца.

Идея: несколько программ можно исполнять параллельно. Простая идея привела к огромным сложностям.

Программы обычно исполняются “псевдо-параллельно”

**Задача 4** (Обеспечение разделения времени процессора)**.** Сделал аппаратное решение, будет генерироваться прерывание каждый тайминг, которое будет запускать планирование того, что выполнять следующим.

Когда мы останавливаем процессор, в регистрах что-то есть и это что-то нужно. Идея: откатываться назад, но непонятно насколько, может я давно что-то туда положил. Забыть тоже нельзя, код дальше расчитывает на эти регистры. Регистровый контекст приходится где-то сохранять (чтобы его потом найти), подгружать такой же контекст от другого процесса и запускать его.

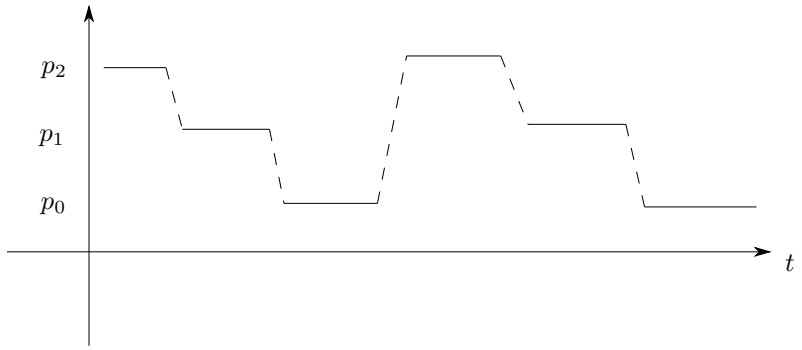


Рис. 1.1: pseudo-parallel

**Задача 5** (Обеспечение разделения памяти)**.** Когда мы пишем код, переменные заменяются на адреса. Программный ноль совпадал с реальным и всё было хорошо. Но у нас много программ и память может быть где угодно. Идея: виртуальная память – при сиполнении подменять виртуальные адреса, считающие, что они в реальном нуле, на физические.

**Задача 6** (Обеспечение защиты данных программы от деятельности других программ)**.** Люди пишут код с ошибками – закон вселенной. Если программы многих людей, а мы своей ошибкой зашли в код чужой программы, то будет нехорошо.

Идея: защита памяти. аппаратное решение – при обращении к памяти понимать свой-чужой.

Но теперь не может работать наш диспетчер, он же должен быть изолированным. А ему нужно что-то взять, куда-то записать..

Идея: кому-то нужно разрешить. Привилегированный режим. Грубо говоря отключаем защиту памяти. Этот привилегированный режим приводит к пониманию современной системы:

System call – обращение пользовательской программы к ядру ОС с требованием предоставить ресурс или выполнить привилегированную операцию.

Теперь OS – универсальный интерфейс. У неё монопольная власть, на уровне неё мы пытаемся эффективно использовать ресурсы

**Задача 7** (Планирование выполнения программ и использования русрсов)**.** У каждого контроллера своя очередь, у каждого жёсткого диска своя очередь, у сетевого узла, у процессора, ... А они ещё и связаны друг с другом

Хочется максимально эффективно всё заменеджить, что суммарно всё максимально быстро исполнилось.

Очередь и ре<..>

**Пример.** Хотим напечатать что-то на принтере. Передаём данные, прервались, перключились на другую программу, а она тоже хочет печатать.. Неразделяемый ресурс.

Идея: ставим блок, чтобы только первый мог давать данные.

Но может быть дедлок.. 1 захватит ресурс 1, 2 захватит ресурс 2, в какой-то момент они хотят получить другой ресурс, не отжав первый – тупик.. Дейкстра занимался этим лет 15.

**Задача 8** (Универсальный доступ к информации на внешних устройствах)**.** Линейная адресация $\to$ файл, каталог

**Задача 9** (Обеспечение коммуникации между программами)**.** Комфортная работа множества программ, а что если они будут передавать друг-другу данные.

ctrl+C ctrl+V – использование буффера, требует ручного управления.

Сигналы, передача из stdout одного в stdin в другого, ..

Появляется понятие виртуальной машины – приложение живёт отдельно и не знает, что есть другие приложения. С этим понятием появляется и термин операционной системы. Теперь делегирование всех операций лежит именно на ОС.

С первой ОС сложно, не понятно кого считать уже ОС.

1963 – компьютер B5000 с ОС MCP – Master Control Programm

## 1.3 Сетевые операционные системы

Компьютеры тогда – только большие и очень дорогие компьютеры.

Затраты на доставку программного кода до машины начинают превалировать

AM – амплитудная модуляция, FM – частотная модуляция. Способы обозначть 0 или 1 в синусоиде.

Модем – модулятор-демодулятор. Теперь проблемы с безопасностью. Раньше был один конкретный подконтрольный оператор, с которого можно было в случае чего спросить. А теперь надо защищаться от людей. Появляются понятии учётной записи, аутентификации.

Появляются компании, специализирующиеся на предоставлении компьютерного времени. У такой компании могло быть уже несколько компьютеров.

Могла быть большая нагрузка в Чикаго и простаивать компьютер в Бостоне. Тогда строили линию ATM между Чикаго и Бостоном, чтобы перенаправлять звонки.

## 1.4 Универсальные операционные системы

Мотивация: в 60-х все ОС были платформозависимые, невозможность повторного использования кода.

Идея: создать универсальную ОС на любую платформу

Парадокс: чтобы разрабатывать под такую ОС, на ней должен быть компилятор языка высокого уровня

Платформа переносимая, она сама написана на языке высокого уровня

В решении участвовалось подразделение компании AT&T – Bell Labs

Открыли реликтовое излучение, изобрели транзистор, открыли фотоэффект, матрица, “Математическая теория связи”

MULTICS – привязана жёстко к набору платформ. multiplexed

UNICS – uniplexed. Пишется полностью на ассемблере. Первая редакция запускается 01.01.1970

В 70 продолжаются разработки и разрабатывается язык B – интерпретируемый язык. UNICS переписывается на B

Керниган и Ричи разрабатывают C, встраивают его компилятор в UNICS (который ещё на B)

конец 75 – ed.4 ядро на C

Первая универсальная система

ed.7 1978 – последняя редакция UNICS, в ней появляется bash

В дело вступает антимонопольная служба США. AT&T весь код передала университет (первым – Бэркли). UNICS → UNIX.

Бэркли создаёт дочернее предприятие BSD (- software distribution)

Free BSD, Open BSD, ...

MIT, Barkley, Stanford – три университета

SUN (stanford university network), SUN OS .. Solaris

Проблема: коммерческие юниксы начинают патентовать решения. BSD лицензия защищает только имя автора. Многие пользовались этим, чтобы закрывать для других целые направления развития ОС

Манифест Столлмана – 4 свободы (0,1,2,3) программного изучения: использовать, изучать&адаптировать, распространять копии, публиковать

©→ ↄ

Gnu is Not Unix

gcc – gnu c compiler

Студент Хельнского университета начинает интересоваться MINICS и преобразовывать. Появляется новая неожиданно-популярная система. Таненбаум делает пост: Линукс устарел

Студент – Линус Торвальдс.

ему остаётся только доказать, что его система качественная. Столлман предлагает ему подключится к GNU. Линус соглашается, но с условием, что GNU переименуется в GNU/Linux – 1983-4

1989 – NeXT создаёт ОС NeXTSTEP

1997 – Darwin → MacOS. Apple хочет выйти на рынок компьютеров и покапает NeXTSTEP вместе со всем, что у неё есть. Что-то добавляет из FreeBSD, что-то сами дописывают.

Уровни:

- Функциональные – с позиции пользователя
- Информационная – потоки данных, структурированные информационные объекты
- Системная – интерфейсы: аппаратные, пользовательские, ..
- Программная – ООП, функциональная, ...
- Данных

**Определение 3.** Цель ОС – обеспечить производительность надёжность и безопасность исполнения пользовательского ПО, эксплуатации железа, хранения и передачи данных и диалога с пользователем.

Функции ОС:

- Управление разработкой и исполнением ПО
  - API
  - Управление исполнением
  - Обработка и обнаружение ошибок
  - Доступ к устройствам I/O
  - Доступ к хранилищу
  - Мониторинг ресурсов
- Оптимизация использования ресурсов. Хотим много всего, что противоречит друг с другом. Критериальные задачи $\hat{K} = \alpha K_1 + \beta K_2 + \gamma K_3$

  Real-time ОС - гарантируется время отлклика. Представим самолёт. При посадке ему нужно открыть закрылки. Если слишком сильно – мы перелетим, слишком слабо – упадём. Бортовому компьютеру нужно быстро посчитать этот угол.

  Условный критерий $\hat{K} = \alpha K_1 + \beta K_2|_{K_3 > z}$
- Поддержка эксплуатации. ОС должна иметь средства диагностики и восстановления на случай, когда железо ломается (inevitably происходит). Любая ОС умеет откатываться к последней удачной конфигурации.

- Поддержка развития самой ОС. ОС живёт дольше программного и аппаратного обеспечения. Может содержать ошибки и уязвимости..

Подсистемы:

- Управления процессами
  - Дескрипторы (PCB)
  - Планировщики. Для разных типов ресурсов выстраивают оптимальную очередь
- Управления памятью
  - Виртуальная память
  - Защита памяти
- Усправления файлами
  - Символьные имена → физические адреса
  - Управление каталогами
- Упрваление внешними устройставмаи
  - Драйвера. Устройств сотни тысяч, для каждого должна быть подпрограмма, которая знает как с ним работать.
  - Plug&Play (Plug&Pray)
- Защита данных
  - Аутентификация и авторизация
  - аудит
- API
  - Разработка ПО
  - Исполнение ПО
- Пользовательский интерефейс
  - CLI
  - GUI

Вопрос: Что будет в ядре стал главным. Почему? Привелигированный режим, резидентность ядра (не меняет адреса после загрузки)

Принципы ОС:

- Модульная организация (в любом сложном ПО такое есть)
- Функциональная избыточность – функционал ОС существенно больше реального сценария её использования. Удобно для разработки, не удобно для эксплуатации

- Функциональная избирательность – всегда есть способ сохранить из всего многообразия функций только те, которые нужны для конкретного сценария.
- Параметрическая универсальность – при разработке нужно максимально не загонять себя в константные рамки

## 1.5 Поддержки концепций многоуровневой иерархической системы

Модули:

- Ядра
- Работающих в пользовательском режиме

### 1.5.1 Монолитная Архитектура

Все видят всё. Преимущества производительности.

Три слоя:

- main program (software)
- Services
- Utilities (hardware)

Хорошая модель пока немного не очень сложного кода. Потом стало понятно, что три слоя это мало.

### 1.5.2 Многослойная архитектура

Не отдельная архитектура, а скорее концепция

### 1.5.3 Микроядерная архитектура

Часть слоёв вынесется в пользовательский режим.

Удобно: абстрагирование через системные вызовы, часть отдаём в подкачку и экономим память

Неудобно: небезопасно, в подкачке что-нибудь можно заменить и при подкачке оно исполниться как часть ОС; могут возникать проблемы с дедлоками и прочими тупиками.

Линукс – монолитное, но модульное ядро.


Рис. 1.2: many

## 1.6 Процессы

**Определение 4.** совокупность набора исполняющихся команд, ассоциированных с ним ресурсов и контекста исполнения, находящиеся под управлением операционной системы.

запускаем программу – последовательно исполняем набор команд. Если запустить вим в двух разных терминалах, это будут те же наборы команд, но разные процессов.

Поэтому ещё добавляет ассоциированные ресурсы. Но и здесь не ставят точку. Код ядра будет выполнятся в контексте вашего процесса. Есть стек пользователя и стек ядра. Ещё есть регистровый контекст, который сохраняется при переключении процессов. команды + ресурсы + контекст

И наконец важно, что процессом управляет операционная система

PCB – дескриптор процесса

- – PID – process identificator
  - PPID – parent PID
  - UID – User ID
  - ...
- ресурсы
- история использования ресурсов

**Определение 5.** Процесс – множество потоков

Поток – совокупность набора исполняющихся команд и контекста исполнения, находящиеся под управление операционной системы и разделяющие ресурсы некоторого процесса.

Поток – тоже находится под управление ОС. Планированием ОС занимается на уровне потоков.

Раз ОС занимается переключеним потоком, значит она умеет сохранять контекст конкретного потока.

Если потоков много, то возникает ограничение на их количество. Очень сложно прогнозировать правильное количество потоков. Двух уровней мало, не можем управлять распределением времени на потоки

**Определение 6.** fiber (light-weight thread) (волокно в нити – thread'e) – набор команд, разделяющий ресурсы и контекст исполнения одного потока и находящийся под управлением пользовательского приложения.

Куча вопросов, а чего мы хотим? Не будет занимать в пространстве ядра никаких структур.

Кооперативная многозадачность облегчённых потоков. Внутри волокон есть безусловные переходы, вызывающие планировщик, чтобы он решил выполнятся дальше или нет.

Поток в Erlang – fiber.

Корутина.

Но этих трёх уровнях всё ещё мало

**Замечание.** Хром создаёт процессы отдельно для вкладок. Но даже если их там, 200, другие приложения не помирают, получая $\frac{1}{200}$ ресурсов

**Определение 7.** job / C(ontrol)Group – настраивание квот. На количество подпроцессов, на интернет-трафик, ...

Процесс:

- Создание процесса
- Обеспечение ресурсами
- Изоляция
- Планирование

- Диспетчеризация
- Межпроцессное взаимодействие
- Синхронизация
- Завершение

### 1.6.1 Создание

Создать процесс – создать стркутуры данных – ProcessControlBlock. В любой операционной системе процесс порождается другим процессом. Часто создание процесса называют рождением.

В Линуксе процессы образуют дерево и порождаются клонированием. Начальный процесс, PID = 1, init/systemd. Как создаётся первый процесс.. сложно, оно часто выглядит как набор костылей. у начального процесса PPID = 0. Есть два процесса, у которых PPID = 0: PID = 1, PID = 2, два дерева: пользовательские и ядерные процессы.

Чтобы завершить процессы, должны завершиться все его потомки.

**Замечание.** Зашли в баш по ssh, запустили веб-сервер. Хотим закрыть баш, но оставить сервер, ради которого мы собственно и зашли. Для такого есть механизм процессов-демонов. При завершении работып процесса-родителя, процесс переподвесился выше.

Если процесс завершился аварийно, и его дети не успели получить код завершения, они подвешиваются выше, чтобы код завершения уже их кто-нибудь прочитал.

ctrl+C – послать сигнал завершения

ctrl+Z – послать сигнал SYGSTOP

Если у приостановленного процесса завершился подпроцесс, от него осталась строка в таблице с его PID'ом и кодом завершением. А PID'ы штуки конечные. А если они закончатся, то даже команду kill нельзя будет послать, потому что это отдельные процесс. Такой подпроцесс называется зомби, а описанная ситуация – зомби-апокалипсис.

Порождение клонированием: fork (полная копия адресного процесса, со всеми ресурсами (указателями на их дексрипторы), подменяется значение PID), подменяется код на код подпроцесса.

Такая система не позволяется подпроцессу иметь большие права, чем сам процесс, потому что он их наследует, копирует

В Windows ситуация другая почти с начала: Есть Диспетчер/Менеджер процессов. Каждый процесс создаётся им. Может быть больше прав у дочернего процесса

Самая простая модель: два состояния: ожидание и исполнение. При рождении он попадает в ожидание.

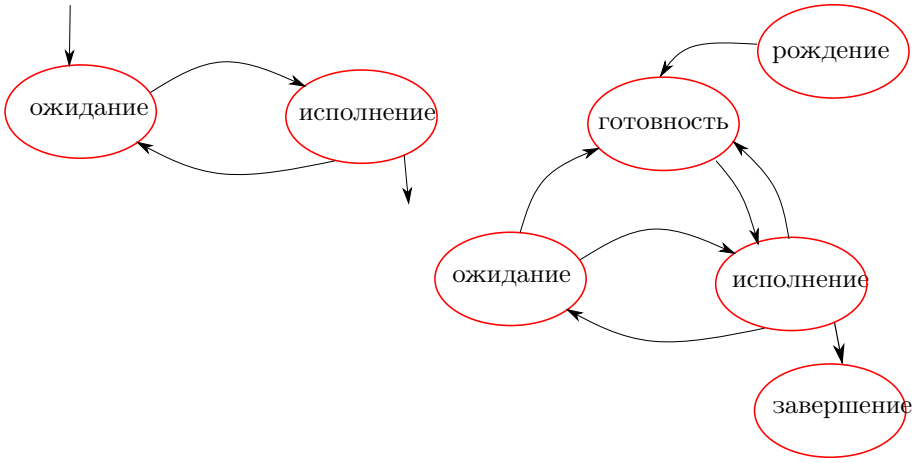


Рис. 1.3: 2states

Проблемы: процессы с исключениями не умирают сразу (для обработки ошибок) и иногда пытаются сделать то же самое (деление на ноль например) много раз, чем съедают ресурсы. Новое состояние – готовность

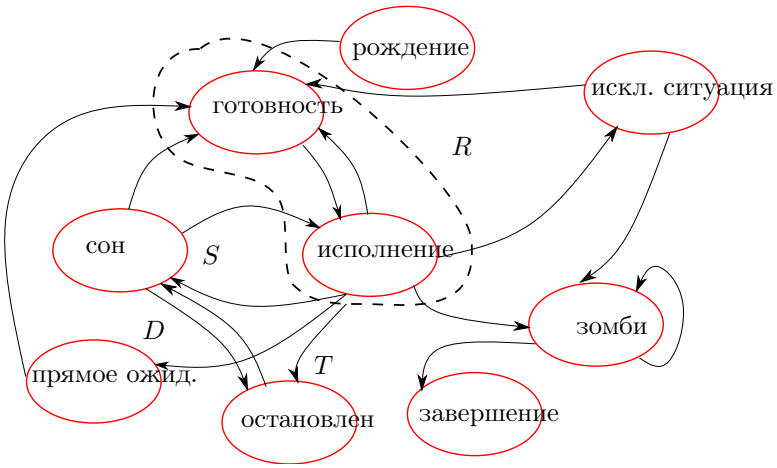


Рис. 1.4: manystates

## 1.7 Планирование

Планировать один шаг плохо, планировать надолго тоже много

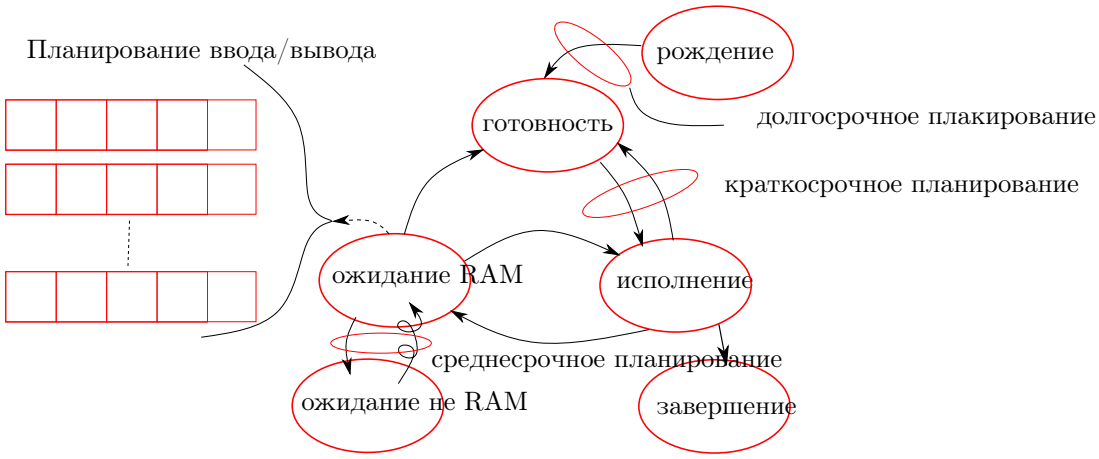


Рис. 1.5: plans

Критерии алгоритмов: справедливости, эффективность. Решения оптимальные по обоим критериям совпадают только если все процессы равны.

Новые критерии: полное время исполнения. Например при компиляции чего-нибудь большого хочется побыстрее получить результат и не важно что там по справедливости и ресурсам с другими процессами

Ещё один: для пользователя важно, чтобы не было ожидания

Последний: для пользователя важно время отклика. Например когда набираешь в vim'е хочется видеть каждую буковку по мере того как она печатается, хотя с точки зрения ресурсов хорошо буфферизировать строчку и один раз её выводить.

Свойства:

1. Предсказуемость – на одинаковых параметрах одинаковый результат.
2. Минимальные накладные ресурсы
3. Масштабируемость. .

Параметры планирования:

- Статические параметры системы – не поменяются в этом экземпляре ОС
- Динамические параметры системы – свободные

- Статические параметры процесса – не поменяются во времени
- Динамические процессы:
  - CPU-burst
  - I/O-burst

Алгоритмы планирования:

- Вытесняющие – по сигналу можем отобрать процесс исполнения
- Невытесняющие – штука начала исполняться, завершила, дала ход следующим

Первый алгоритм: First Came – First Served (FCFS). На деле очередь (FIFO)

CPU-bursts: $\begin{cases} p_0 & 13 \\ p_1 & 4 \\ p_2 & 1 \end{cases}$

Если очередь $p_0p_1p_2$, то $T = 18 \quad \widetilde{\tau}_{\text{испл}} = \frac{13+17+18}{3} = 16 \quad \widetilde{\tau}_{\text{ожид}} = \frac{0+13+17}{3}$

Но если $p_2p_1p_0 \quad T = 18 \quad \widetilde{\tau}_{\text{испл}} = 8 \quad \widetilde{\tau}_{\text{ожид}} = 2$

RoundRobin (RR).

$k = 4$ – квант, по прошествии которого очередь движется.

Пусть очередь всё та же $p_0p_1p_2$

$T = 18 \quad \widetilde{\tau}_{exec} = 11.(6) \quad \widetilde{\tau}_{wait} = 5.(6)$

$k = 1 \quad \widetilde{\tau}_{exec} = 10 \quad \widetilde{\tau}_{wait} = 4$

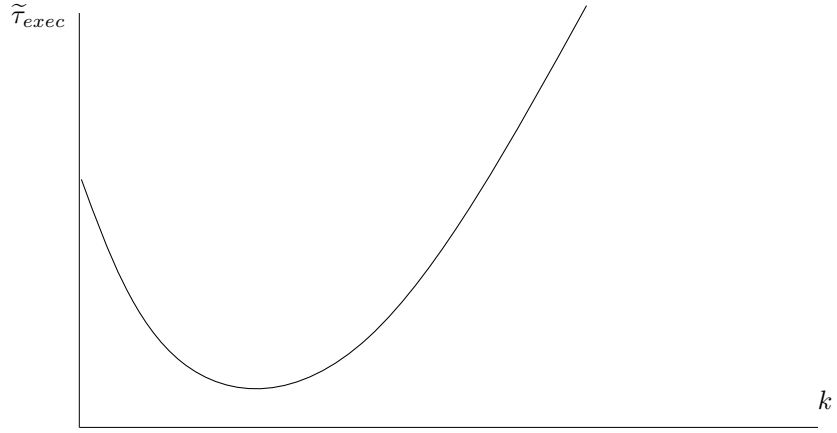


Рис. 1.6: rr

Shortest Job First (SJF) – обычно вытесняющее по кванту. Понятно что делает

Гарантированное планирование:

- $N$ – количество процессов
- $T_i$ – время процесса в системе
- $\tau_i$ – время исполнения процесса

$\tau_i \sim \frac{T_i}{N}$

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

Два минуса:

- Не очень устойчив к "взлому". Можно сделать так, чтобы твой процесс пропускали вперёд
- коэффициент вещественный. Постоянно упорядочивать вещественные числа.. ну такое

На идею посмотрели и отложили лет на 20.

**Пример.** Есть вебсервер и СУБД. СУБД не распараллелена и становиться усзким местом. Хочется, чтобы ОС поняла, что СУБД очень нужна.

Многоуровневая очередь.

Давайте зафиксируем возможное количество приоритетов. Давайте раздавать процессам приоритеты. Очевидно они могут совпасть. Внутри одного приоритета сделаем Round Robin. А всего так: пока есть кто-то с меньшим приоритетом, процессы ждут.

**Пример.** в MIT в 1967 остановили компьютер с аптаймом 7 лет и обнаружили там процесс, который 7 лет назад был поставлен и так и не исполнился.

Многоуровневая очередь с обратной связью – свзять с тем сколько процессреально тратит. Есть константное количество приоритетов и кванты в них увеличивающиеся с умееьшением приоритета.

Но теперь нет внешнего управления приоритетом

Хочется:

1. Внешнее управление приоритетом.
2. Эффективное использование ресурсов:
   (a) процесс меньше простаивает
   (b) как можно быстрее покидал оперативную память

(c) Уменьшить переключения в режим ядра и переключения процессов.

3. Минимизировать накладные расходы:
   (a) Процессорное время на исполнение самого планировщика (сами процессы хочется исполнять)
   (b) Память для планировщика
4. Минимизировать риски возникновения блокировок

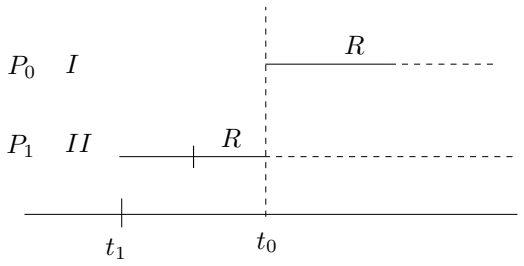


Рис. 1.7: lock

1.
2. SJF + изменяемые кванты переключения + раздельные очереди и алгоритм балансировки между ними
3. целочисленная / битовая арифметика, алгоритмы за n или лучше, в идеале за константу
4. гарантированное планирование

## 1.8 Конкретные планировщики

### 1.8.1 Планировщики Windows

32 очереди. Чем больше номер, тем больше приоритет. Нулевая выделена. 1-15 – dynamics, 16-31 – real time

Процесс в real-time очереди остаётся там и не меняет свою чередь. dynamic – ОС будет своими алгоритмами менять положение.

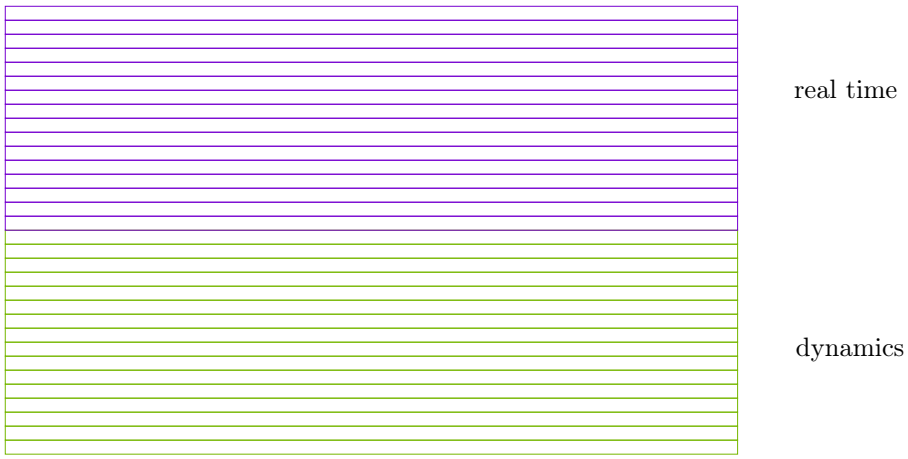


Рис. 1.8: win1

Классы приоритетов процессов:

- 24 – Realtime
- 13 – High
- 10 – Above Normal
- 8 – Normal
- 6 – Below Normal
- 4 – Idle

По умолчанию процесс попадает в очередь 8. В какой-то момент там может быть достаточно тесно. Внутри Round Robin с квантом в 12 тиков таймера (одинаковый для всех очередей)

Уровни насыщения потоков:

- +15 time critical
- +2 highest
- +1 above normal
- 0 normal
- -1 below normal
- -2 lowest
- -15 idle

Это дельты работают с ограничением по группе, т.е. 8‘ + ‘15 = 15, выше он не поднимется.

Хороший процесс – интерактивный процесс. Пользовательский ввод, ожидание на семафоре. Хочется такие поощрять. За выход из ожидания, он получает подъём в зависимости от того откуда он вышел. После этого, если он становится менее интерактивным, долго хочет что-то делать, то у него постепенно уменьшается приоритет

### 1.8.2 Планировщики Линукса

$O(1)$ 140 очередей от -20 до 100. 100 – real time, 40 dynamics

Внутри натуральные очереди, FIFO. Если процесс завершился, не израсходовав свой квант, он ставиться в конец очереди с этим остатком. По прошествии кванта, процесс перемещается в Non Active очередь в копии всей этой структуры. В какой-то момент кванты заканчиваются у всех процессов, даже с самым минимальным приоритетом, до них доходит. И тут мы меняем местами active и non-active структуры из 140 очередей.

CFS – completely fair schedule

1. вещественные вычисления нафиг
2. по памяти хочется как-то пооптимизированнее.

Решили вернуться к гарантированному планированию. Отказались от многоуровневых очередей! Теперь одна очередь

У каждого процесса две величины:

- execution time – время исполнения, которое уже
- max execution time – сколько мы должны, т.е. сколько он стоит в очереди.

Храним очередь отсортированной по execution time. Max execution time растёт с разной скоросттью в зависимости от его хорошести (параметра NICE)

Выбирается процесс с наименьшим execution time’ом и ему даётся квант равны его max execution time. По прошествии его, у него накапливается execution time и в следующий раз он попадёт в исполнение не скоро.

## 1.9 Синхронизация процессов

### 1.9.1 Взимоисключния

Пролог – критическая секция – эпилог

Критическая секция

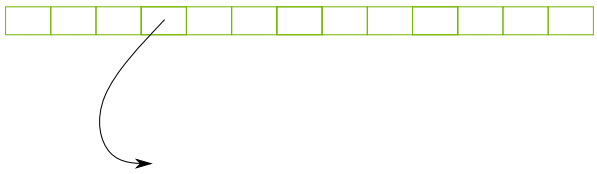

Рис. 1.9: cfs

### 1.9.2 Прогресс

2 процесса находятся в race condition, когда они одновременно подошли к захвату блокировки

### 1.9.3 Отсутствие голодания

Может быть такая ситуация, когда два процесса по очереди забирают себе ресурс и какой-то третий процесс всегда ждёт – голодает – такого не хочется

### 1.9.4 Отсутствие тупиков

Два процесса имеют два ресурса и оба хотят получить второй, не отпуская первый.

Всё это вместе – идеальный мир.

Пусть $P_0$ и $P_1$ находятся в race condition

1. Алгоритм Замка

```
1              shared int lock = 0;
2
3              Pi{ ...
4                  while(lock);
5                  lock = 1;
6                  critical section
7                  lock = 0;
8                  ...}
9
```

Между while и lock = 1 может произойти прерывание и мы попалём в критическую секцию с двух процессов

2. Строгое чередование
3. Флаги готовности

### 1.9.5 Алгоритм Петерсона (алгоритм взаимной вежливости)

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

**Определение 8.** Множество процессов находится в тупиковой ситуации, если каждый процесс из множества ожидает событие, которое может вызвать только другой процесс из этого же множества.

**Теорема 1** (Кофман)**.** Тупиковая ситуация возникнет при одновременных 4 условиях:

1. Mutual Exception – невозможность двум процессам держать один и тот же ресурс
2. Hold & Wait – процесс может удерживая ресурсы запрашивать другие ресурсы (по другому: пересекаются критические области)
3. No Preemption (условие неперераспределяемости) – ресурс, который выделен процессу может освободить только сам процесс
4. Circular Wait – существует кольцевая цепь процесс, в которой каждый процесс ждёт доступа к ресурсы, удерживаемому другим процессом

А что делать при нарушении одного из этих условий Кофман не сказал..

Пути:

- игнорировать
- предотвращать
- ищем и исправляем

Большинство ОС: предотвращают по максимуму, но все не могут. Каждая старается уменьшить вероятность возникновения каждого из условий.

Попытки решения:

1. процесс может кидать задачу в очередь, не ожидая отклика от, допустим, принтера.

2. Берём сразу все ресурсы, которые нам нужны. Если какой-то не дали, нужно всё отдать. Это не всегда можно.

3. Можем у спящего процесса отобрать ресурс, который он держит, передать другому и потом вернуть как будто и не отбрали. Это тоже ограниченно можно делать, например если блокировка только на чтение

4. Нумеруем все ресурсы натуральными числами. Разрешаем любому ресурсу брать следующий ресурс только если у него номер больше, чем у всех ресурсов, которые он уже взял. Периодически мы перенумеровываем ресурсы и это обеспечивает выполнение кого-нибудь. Звучит как огромные расходы, но зато есть гарантия.

**Задача 10** (Проблема читателей и писателей)**.** Два процесса попеременно читают так, что всегда хотя бы один читает. Один процесс хочет записать, но ему никогда не дают

Заводим очередь. Периодически даём шанс записям и блокируем новые чтения.

## 1.10 Память

Фон-Нейман – одна большая линейно-адресуемая память для кода и данных.

Отказываясь от этого мы теряем (об этом далее), но придерживание этому приводит в не очень эффективным решениям. Есть области памяти, куски кода или данных, которые часто нужны, а есть о которых никто может уже и не знать.

- Регистры СРИ – байты и 0.1ns
- L1 Cache – КБайты 0.5ns
- L2 Cashe – МБайты 5ns
- RAM – ГБайты 50ms
- ....
- HDD – ТБайты 5ms

Swapping, подкачка/откачка данных между RAM и HDD.

- Перемещающий загрузчки – один раза всё пересчитываем и дальше работам по физическим адресам. Но сам пересчёт долгий и не рациональный. Пример: открываем браузер, он переадресовывает всё, а ты используешь 5% его функционала.

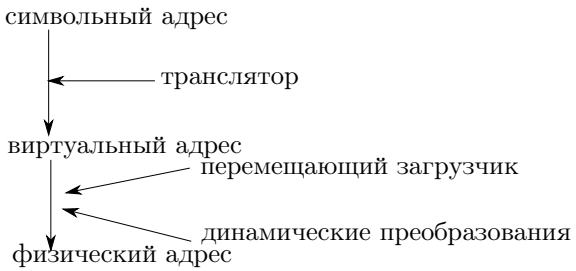


Рис. 1.10: address

- Динамические преобразования – пересчитываем адрес при необходимости. Минус: если что-то нужно несколько раз – столько раз нужно пересчитать.

Подкачивать:

- Целиком адресное пространство – целостность данных
- Фрагментами – сложно обеспечить единую защиту
- Если взяли большой блок, его может не быть куда обратно вернуть в памяти.

Подход:

- Линукс – раздел подкачки. В файловой системе много всякого функционала, проверки доступа, журналируемость, .. Для подкачки решили сделать отдельный маленький драйвер. Высокая защита, скорость. Но если налажал с размером раздела, то смерть.
- Винда – файл подкачки. Ограничен только размером =всего= диска.

Стратегия распределения памяти:

1. без подкачки
2. с подкачкой

### 1.10.1 без подкачки

Нет проблемы разнородных адресных пространств.

- Фиксированные разделы:
  - равные по размеру.

    Давайте для каждого байта хранить занят он или нет.. в памяти?.. ой, девятая часть памяти ушла..

    Давайте разобьём адресное пространство на разделы равного размера, пронумеруем. И тогда например pid процесса = номеру раздела. Очень просто пересчитывать адрес. Элементарно с защитой. Но тт плюсы заканчиваются

    Вообще это как-то нерационально. Если разделы сделать большие, то количество их будет малым, а если маленький, то сложно писать код (нужно самому писать алгоритм подкачки для своей программы).
  - разные по размеру

    А что если сделать и маленькие и больший и будем стремится к тому, чтобы процесс занял минимальное возможное для него адресное пространство. Точно предсказать сколько каких процессов никак. Есть вероятности максимум.

    Кидаем процесс в минимальный достаточный по размеру. Это лучше, чем равные, ведь он попадает в оптимальный раздел. Но проблема: может прийти много мелньких процессов, занять всё и не оставить памяти для больших процессов.

    Вспомним про очереди. Если нет подходящего раздела, занимаем место в очеди и ждём. Очереди $\rightarrow$ голодание, теперь нам могут не давать родиться. И ещё проблемка.. очереди тоже надо хранить.. в памяти..
- Динамические разделы

  Будем давать памяти сколько просят. М, теперь пересчитывать виртуальный адрес это боль, потому что куча блоков с разными смещениями.

  А что произойдёт когда мы доходим до конца. А где-то в рандомных местах освобождают память процессы. И ой, у нас появился штрих-код фрагментации, а не память, и поместить большой процесс становится около-нереально.

  Если долго есть проблема, то мы останавливаем работу вообще всего и начинаем всё дефрагментировать.

  В фоновом режиме двигать по одному? А кого.. и в каком порядке.. Можно дергать тех, кто уже стоит в очереди и ждёт устройства ввода-вывода например. Но если мы посередине перемещения получим этот доступ, то очередь будет простаивать, что плохо.

### 1.10.2 С подкачкой

- Страничная организация
- Сегментно-страничная фрагментация

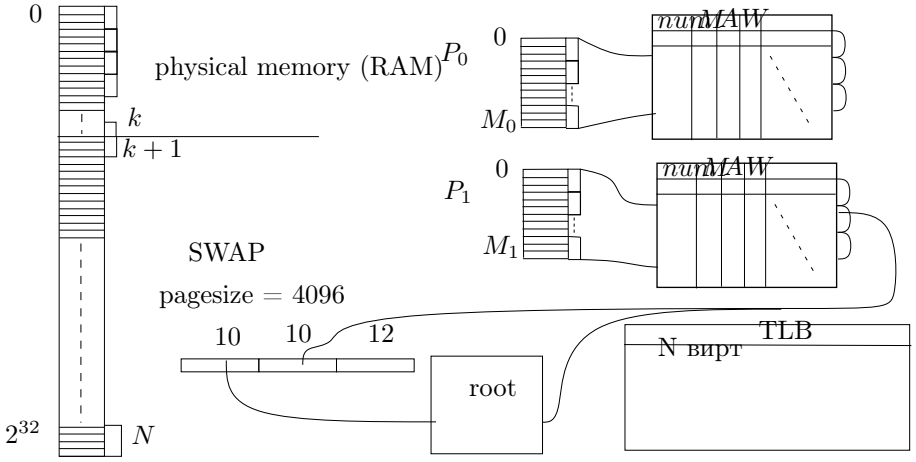


Рис. 1.11: address-space

Делим память на страницы. Внутри страницы блоки нумеруются заново с виртуального нуля конкретной таблицы.

Адрес – начало таблиц + смещение. Казалось бы нужна память – обращаешься по pagetable ищешь реальный адрес и идёшь туда.. но нет. Данные могут лежать в подкачке и их может не быть в физической памяти. Если данных там нет, то стучаться нет даже смысла. Чтобы не стучаться без смысла есть флаг M, который 1 если в физической памяти лежат данные и 0, если нет. Если нет, то процесс вызывает страничное прерывание, его в это время не запустить.

Очередь активных и неактивных страниц. Сначала все страницы добавляются в конец неактивной очереди. Как только мы какую-то берёт, ставим флаг A(ccess) в 1 и добавляем её в активные. По кд обнуляем бит активноти последней активной странице и помещать её в очередь неактивных.

бит W нужен, чтоб проверять меняли ли мы файл. Если нет, то при отсылании мы не будем перрезаписывать, а просто объявим изначальную копию снова верной. Если нет, то запишем в конец после положенного размера swap (для $2^{32}$).

Теперь мы через рутовую табличку можем смотреть на одну страницу. Но вообще нам могут быть нужны код, данные, константы из какого-то файла, .. хочется объяснить ОС, что всё из этого нужно хранить рядом.

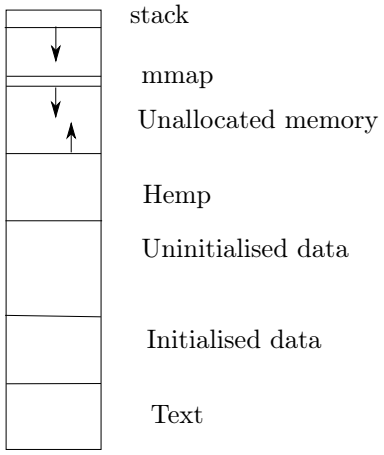


Рис. 1.12: stacknstuff

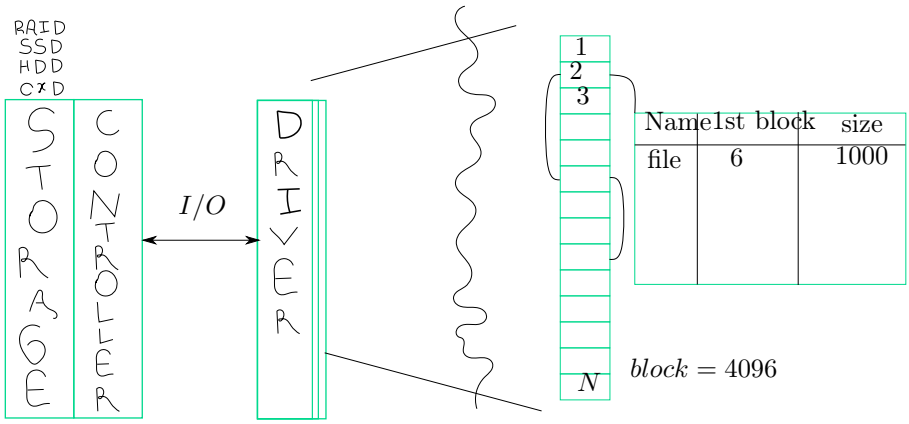


Рис. 1.13: storage

1. Писать непрерывной последовательностью блоков. Такие существуют. Например CD/DVD-диск. Если в блоке хранить следующий адрес, то он сколько-то занимает и либо полезное пространство не степень двойки, либо размер блока. (Тем не менее это используется для хранения свободной памяти как одного большого файла-связного-списка.)

2. Хранить адреса-переходы в отдельном массиве. FileAllocationTable / FAT, FAT32. С фрагментированностью проблем нет. Проблема: все яйца в одной корзине. Если потерять эту таблицу, то всё.

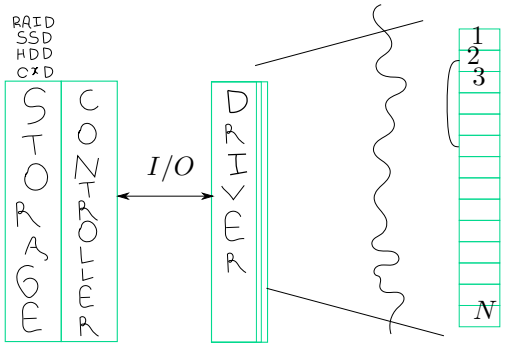


Рис. 1.14: storage2

## 1.11 Распределённые ОС

Предпосылки:

1. Развитие технологий построения вычислительных узлов притормозилось и кажется приблизилось у своему порогу. Частота процессора (теплоотведение это боль), память (random access, больше ячеек, сложнее всё адресовать), .. Запрос не уменьшаются при этом, скорее наоборот.

Ну давайте считать на многих.

**Пример.** Веб-ресурс. Мы делаем запрос и там в относительно реальном времени хотим ответ. Т.е. процесс живёт доли секунды, такие можно аккуратно балансировать.

Если процессы большие, сложнее. Сам процесс перенест это там пид, регистровый контекст, ну килобайта хватит. Но есть ещё память, порты и они все остались там.

Ещё отдельная проблема: географическая распределённость. Задача: приблизить контент к потребителю.

CDN – Content Delivery Network. Статические данные тянутся из близкого сервера например. Но задачи есть не только в вебе.

Принципы прозрачности:

1. Прозрачность расположения – процесс не имеет возможности узнать на каком узле он выполняется.

2. Прозрачность миграции – Когда меня переместят, я даже не узнаю сам этот факт.
3. Прозрачность размножения – процесс не может знать сколько его копий сейчас существует. (не предсказуемо, поэтому можно запустить например 5 разных версий и убить те, которые медленные, заодно найдя узел, в котором быстро)
4. Прозрачность конкуренции – я не должен знать, что я с кем-то конкурирую за ресурсы.
5. Прозрачность распараллеливания – если моё решение использует параллельное программирование, то управление этим распараллеливанием для меня прозрачно,я о нём ничего не знаю.

Не должно быть централизованных структур данных. Например табличка пидов

Также не должно быть распределённх алгоритмов..

4 свойства:

1. Не один узел не должен иметь полной информации о состоянии всей системы.
2. Узлы принимают решения только на основе локальной информации.
3. Выход из строя одного любого узла не должен приводить к выходу из строя всей системы.
4. Не должно быть явного или неявного предположения о существовании глобальных часов.

### 1.11.1 Распределение оперативной памяти

DSM – distributed shared memory

Решения:

1. Давайте плюнем на децентрализацию. Будет memory server, который хранит все страницы. Когда процесс захочет обратится к памяти, он идёт на memory server и просит там память.

   Звучит неплохо, но память memory server это ограничение все системы, что убивает масштабируемость.

2. Миграция страниц. От предыдущего сохраняется, что одна страница в системе ровно в одном экземпляре. Хотим страницу – просим у “соседей” в графе. Можно дальше подбирать топологию этого графа.

   +: консистентность

   -: время, затраты могут быть большими, что даже эффекта от распределённости может не быть

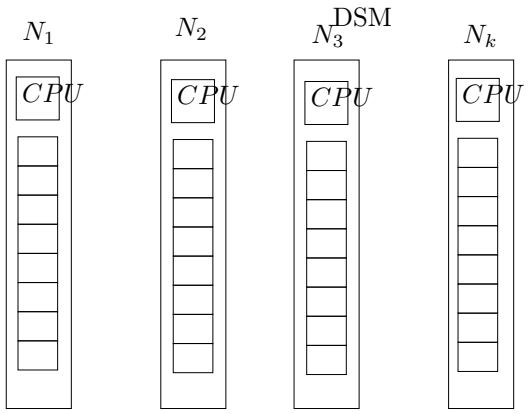


Рис. 1.15: dsm

Хм, но читать то можно параллельно

3. Перемещаем копии, если запрос на чтение. Но теперь кошмар с записью. Надо всех просить сказать, что они удалили у себя эту страницу
4. Полное размножения: пусть все меняют у себя, но друг другу об этом сообщают. Оговорка: не должно создаваться противоречащих изменений (например только добавление). Но теперь важен порядок изменений. Введём сервер синхронизации обратно. Ему дают кучу запросов, он их нумерует и выдаёт всем, чтобы они совершали изменения. Узкое место – сервер синхронизации, но мы хотя бы не потеряем всю память, уже хлеб.

На самом деле оптимального решения объективно найти не получается.

Может с файловой системой полегче?

### 1.11.2 DFS

Две подзадачи:

1. обмен данными
2. поиск файла

размножение файлов на чтение – сразу нет. Они не равного размера, не кратны $2^n$, обеспечивать синхронизацию это слишком болно.

Можно мигрировать файл, но это выгодно, если мы что-то серьёзное делаем. Если там нужно одно поле изменить, не надо там гигабайтную бд таскать туда сюда.

Можно сделать драйвер, который ведёт себя как локальный, но на самом деле шлёт запрос на изменение туда, где файл лежит. В жизни оно ± так и работает, допустим решили.

А что же с поиском?

Решения:

1. Двухуровневый адрес: ip + путь. Кто-то должен знать об узлах. Поиск медленный

2. Линуксовый механизм, монтируем всё в одну корневую систему.

   Проблема: монтировать всех ко всем плохо. А они разные, появляются, исчезают. Каждый узел должен знать информацию о всех ухлах, нарушаем свойство.

   Ходить по соседям. Ходим дальше через них.. Провоцируем кольца и контролировать их сложно.

3. Централизованное пространство имён. Но тогда все минусы централизации.

Как только мы что-то хотим сделать, упираемся в синхронизацию. син хронос.. время.. нужны какие-то очень точные часы, потому что если например часы отстают на секунды за 10 лет, мы порадуемся, а для машины пары дней достаточно, чтобы накосячить с синхронизацией.

1978 – Лампорт заинтересовался синхронизацией. Давай-те так: Нет глобального таймера. Нас в целом волнует только последовательность.Нам не нужно точное время.

В двух ситуациях можем понять последовательность:

- Действия произошли в одном процессе (а процесс это последовательность)

- Одно действия это посылка, а другое приём. Понятно, что первое произошло до второго.

Можно подгонять своё время. Например если присылают что отослали в 25, а у нас 20, то мы подгоняем, что приём был после посылки

Проблема один, с которой разобрались довольно быстро: время может сопасть, а нам нужна последовательность. Добавим всем временам какую-то уникальную вещественную константу.

Но мы так постоянно разгоняем время. Решения есть делать стоп-синхронизацию, когда все времена откатываются к какому-то разумному решению.

## 1.12 Виртуализация

virtual machine – абстракция приложения от ресурсов. Сами приложения не знаю о том, что они конкурируют за ресурсы, считают, что всё принадлежит им.

И хотя они все такие изолированные, но они неявно влияют друг на друга. Например может возникнуть тупик или кто-то может не отдать ресурс аварийно завершившись.

Задачи виртуализации:

1. Поддержка устаревших операционных систем и приложений
2. Повышение надёжности и отказоустойчивости
3. Создание сред для тестирования
4. Консолидация серверов.
5. Повышения управляемости сетевой инфраструктуры

4 технологии:

1. Эмуляция аппаратуры. Device (node) -> Host OS (execution environment) -> App1, App2, ..., Appt, vmenv -> Cntrol App, vemulator -> Guest OS (execution environment)

   Проблемы: два разных деления на страницы от хостящей и гостевой ОС. С произвоительностью проблемы, с кешем проблемы.

   Модель классная для исследований, когда не важна скорость, важно что оно работает

2. Полная (нативная) виртуализация, паравиртуализация. Devices (node) -> Hypervisor (execution environment) -> Virtual Partition Hosts $\rightrightarrows$ Host OS, Guest OS 1..k

   Проблемы: хочется эффективно использовать железо, т.к. оно уже всё настоящее.

   Паравиртуализация – поддержка на уровне ядра

3. Виртуализация уровня ядра ОС. Device (node) -> Host OS (execution environment) -> Container 1..k $\rightrightarrows$ App 1..l 1..k

ЧТо мы хотим виртуализировать:

1. Виртуализация преставлений – конечный пользователь считает, что у него полностью его среда. По факту у пользователей одни и те же приложения
2. Виртуализация рабочих мест – рабочее место это виртуальная машина. Для пользователя разницы нет. Но теперь он не конкуррирует за общие ресурсы, кроме как с другими виртуальными машинами, если

их нсколько. Сбой в работе одной из машине не повлияет на дргих (в отличие от прошлого) Зато обновления придётся делать независимого для всех по-отдельности

3. Виртуализация серверов
4. Виртуализация приложений – portable версия например.

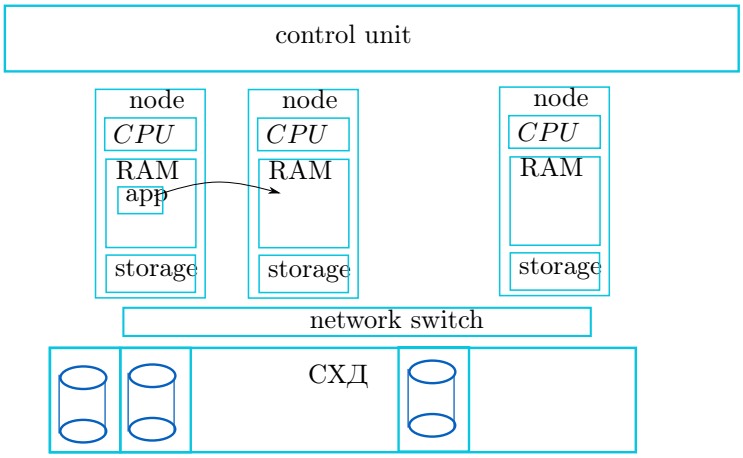


Рис. 1.16: clouds

Проблемы переноса процесса в новую машину:

1. Нужно перенести ресурсы и поток команд. Накладные расходы. Но это не главное.
2. В изначальной машине у процесса есть родители, о нём знаетпланировщик, он там социолизирован и тут его посылают в чужую среду. Очень много накладных расходов, чтобы его реинтегрировать.

Оказалось проще передать адресное пространство, там окажутся только телекоммуникационные расходы. Но storage так не передать, его передавать может минуты и сутуация из-за которой мы решили вообще начать перенос могла 5 раз поменяться.

СХД – система хранения данных. Со своей оперативной памятью, ОС специально заточенной под это со своими алгоритмами и т.д. Отдельно есть network Switch, который обрабатывается запросы к лунам в СХД.

Так скорость доступа к СХД получается быстрее, чем к памяти сервера. Сервера, который без памяти (только ссд-шечка для ОС и всё) называются blade'ами, потому что они получились очень тонкими, т.к. память рала на себя существенную часть толщины.

AMQP – Asynchronous Message Queueing Protocol.

Уровни принятия решений:

1. Узел – можно засыпать и пробуждать по надобности, но это десятки секунд ожидания
2. VM – миграция + остановка/архивирование + поднятие/создание новой из шаблона
3. Container – шалоны приложений в контейнерах
4. Приложение – шаблоны приложений самих по себе
5. Запрос. Есть банасировщик, который получает запросы с веба и кидает по виртуальным машинам

Сверху вниз уменьшаются накладные расходы, но также уменьшается уровень серьёзности операции.

Теперь вспоминаем о пользователях. Они могут иметь виртуальные рабочие места. экран 2 мегапикселя, 180 МБ, даже если сжать 60, но если таких пользователей много, то время network switch приходит к концу

SPICE – повышение требования к рабочему месту, но зато очень существенное ускорение для нашей системы.

Fog computation – размещать центры ближе к пользователю. Более сложная система миграции, уже не один СХД.