OS

Шелудченко Анна Демьяновна, Назаров Егор Александрович,
Папикян Сергей Седракович

January 2021

1 Функции и механизмы ОС, появившиеся на этапе программ-диспетчеров, предшественников операционных систем.

На данном этапе термин ОС не существует (но ряд функций, которые стали в последствии функциями ОС появляются на этом этапе).

Проблемы и механизмы решения:
1) Повторное использовние кода; автоматизация загрузки и линковки.
При увеличении сложности программ, стало необходимым повторное использование кода.

Например : переиспользование тригонометрических функций.

Идея : в области RAM фиксируется область, в которую помещаются часто используемые программы. В основной программе - вызов программы из упомянутой области. Необходима система управления для идентифиакции адресов внутри фиксированной области; для передачи параметров программе и корректной обработки результатов при возвращении из области в основную программу.

2) Оптимизация взаимодействия с устройствами хранения, ввода-вывода.
Так как загрузка всех данных в RAM невыгодна (часть данных может быть никогда не использована), необходим иной подход.

Идея : загрузка части данных, их обработка и последующая выгрузка (циклический подход). Необходимо участие CPU как посредника для синхронизации данных этапов. Тогда, к классической архитектуре Фон Неймана необходимо добавить контроллер, который позволил бы совершать эти действия параллельно (например, контроллер отвечал бы за перемещение данных из хранилища в RAM и занимался этим пока CPU обрабатывает другие данные из RAM, а не простаивает в ожидании). Для избежания обработки неготовых данных или замены еще не обработаных или обрабатываемых (избежания коллизий), необходима синхронизация контроллера и CPU. Для этого был создан механизм прерывания (interrupts).

SPOOL - Simultaneous Peripheral Operation OnLine - обеспечение взаимодействия с периферийным устройством параллельно с работой основного вычислительного узла.

Прерывание - сиганл, поступающий от внешнего устройства к CPU, сообщающий о наступлении некоторого события и в результате которого CPU прекращает выполнение текущего набора команд и передает управление программе-обработчику прерываний

3) Однопрограммная пакетная обработка
Наличие множества компонентов программного кода. Появляется необходимость хранения данных(констант, массивов и т.д.), программ для их обработки (и возможно чего-то еще) совместно

Идея : объединение данных, программных компонентов для совместной работы (пакет).
Данное решение позволяет во время работы одного пакета подгружать другой. Как следствие появляется задача однопрограммной пакетной обработки(разные заказчики пакетов, которые, возможно, готовы платить больше чтобы их пакет обработался быстрее). Отсюда появление очереди и необходимости управления ею(приоретизации).

2 Функции и механизмы ОС, появившиеся на этапе мультипрограммных операционных систем.

Отсутствует возможность эффективно использовать ресурсы системы, так как для разных программ в разные моменты времени могут понадобиться различные объемы ресурсов(например одной программе необходимо мало данных для большого количества вычислений, а другой много данных для малого количества вычислений).

Проблемы и механизмы решения:
1) Обеспечение разделения времени процессора
Возможность использования ресурсов процессора несколькими приложениями (переключения между приложениями).

Идея : на аппаратном уровне появление таймера (для еще одного механизма прерывания). Генерация прерывания по таймеру, для решения задачи о переключении. Необходимость корректного возврата в состояние приложения до прерывания (сохранение регистрового контекста - также решается аппаратно).

2) Обеспечение разделения памяти
На этапе разработки неизвестно нахождение адресов используемых приложений в памяти.

Идея : появление механизма виртуальной памяти (каждое приложение имеет собственное виртуальное пространство, отсчитываемое с нуля; физическая память в последствии выполняет пересчет адресов).

3) Защита деятельности программ от деятельности других программ
У одной программы есть возможность обратиться к адресам памяти, которые содержат данные другой программы.

Идея : наличие проверок на аппаратном уровне (наличие еще одного прерывания); избежание обращения к несуществующему участку памяти.

Необходимо расширение с защиты памяти на защиту любых ресурсов.

4) Планирование использования ресурсов(I/O, CPU, RAM и т.д.) и исполнения программ
Усложнение планировщиков (необходимо учитывать алгоритмы действия данной выполняемой программы).

5) Синхронизация работы различных приложений
Невозможность использования неразделяемого ресурса несколькими программами (синхронизация работы приложний относительно ресурсов).

6) Обеспечение универсального доступа к устройствам хранения
При наличии в памяти программ, работающих параллельно, обращение к памяти происходит постоянно (наличие сложной адресации).

Идея : появление файлово-каталожной системы.

Для эффективного использования системы используется большое количество системного кода

Концепция ОС как слоя абстракции между пользовательским ПО и железом. Как следствие концепция виртуальной машины для каждого приложения - оно не знает реальных ресурсов вычислительной системы и "живет в коробке"

3 Функции и механизмы, появившиеся на этапах сетевых и мобильных (универсальных) операционных систем.

Сетевые операционные системы
Узкое место у предыдущих систем - работа с единым терминалом (input -> CPU).

Проблемы и механизмы решения:
1) Необходимость универсальности ОС относительно различных архитектур ЭВМ

Появлиось много ЭВМ и еще большего количество терминалов для доступа к ним. Как следствие появилась необходимость универсальности ОС для объеденения вычислительных узлов различной архитектуры в единую систему(для повышения эффективности и распределения нагрузки)

2) Необходимость идентификации личности

Идея : (аутентификация и авторизация) контроль доступа к ресурсам вычислительного узла из вне.

Универсальные (мобильные, открытые) системы
Разнообразие операционных систем тормозит развитие отрасли информационных технологий (повторное использование кода, сложности с переносимостью между системами).
Система должна стать универсальной, относительно различных архитектур и платформ (поддержка разработки приложений на языке высокого уровня).

Проблемы и механизмы решения:
1) Создание ОС на языке высокого уровня

Идея : 1969 год - Bell Labs - создание языка C и опреционной системы UNIX (UNICS)

4 Задачи и механизмы, реализуемые в рамках функции операционной системы по обеспечению интерфейса между пользовательскими приложениями и аппаратным обеспечением вычислительного узла.

Управление разработкой и исполнением пользовательского ПО

1) API - Application Program Interface - интерфейс для прикладного программирования (инструментарий для разработки); описание способов взаимодействия программ.

2) Механизм для загрузки и исполнения Software (инструмент управления исполнением).

3) Обнаружение и обработка ошибок.

4) Высокоуровневый уровень доступа к устройствам ввода-вывода (для пользователя не важны физические параметры устройств и т.д).

5) Реализация управления доступа к хранилищу (обеспечение корретной обработки файлов вне зависимости от файловой системы, обеспечение прав доступа между файлами).

6) Мониторинг использования ресурсов.

5 Принципы организации эффективного использования ресурсов компьютера. Критерии эффективности. Подходы к решению многокритериальной задачи.

Оптимизация использования ресурсов

Решение многокритериальных задач принятия решений :
(для работы нескольких программ требуется использование и памяти, и ресурсов процессора. Необходимо использовать и то, и другое эффективно) При наличии нескольких критериев (К1, К2, К3 и др.) и при попытке максимизации одного из них, невозможно максимизировать и остальные критерии.
Решения :
1) *Использование взвешенных критериев (свертка критериев).*
К* = a * К1 + b * К2 + c * К3 + ..., где a + b + c + ... = 1 (весовые коэффициенты).
Поиск варианта, при котором значение суперкритерия К* будет максимальным.

Проблема : недопустимость значения одного из критериев.

Так как необходимо обеспечить лучшее время отклика (например, для систем реального времени), вводят условный критерий :
Поиск максимума для одного из критериев, при наличии ограничений на другие критерии.

2) *Использование цикла Деминга (PDCA).*
Plan (формирование коэффициентов или выбор алгоритма планирования) - Do (последовательное выполнение плана) - Check (проверка предсказанных и полученных результатов, достижения целевых показателей) - Act (решение проблем планирования).

6 Виды архитектур ядер операционных систем. Общая характеристика каждого вида, достоинства и недостатки.

Ядро ОС - *часть кода, которая в совокупности обладает следующими характеристиками : ему присуща резидентность (код ядра весь период эксплуатации находится в оперативной памяти в неизменных адресах), и он работает в привелегированном режиме (код не ограничен проверками на доступ к адресам памяти и ему доступны некоторые специальные инструкции процессора).*

Виды архитектур :

1) *Монолитная архитектура* - Вся ОС в ядре.

+ Быстро(все внутри)
+ Безопасно(Одна точка входа для пользовательского ПО)
- Занимает много RAM на постоянной основе
- Проблемы с надежностью(Много кода в привилегированном режиме)
- при замене части кода ядра необходима пересборка

1.1) *Модульная архитектура* - модификация монилитной архитектуры, которая не требует полной перекомпилции ядра при изменении состава аппаратного обеспечения компьютера. Модульные ядра предоставляют механизм подгрузки модулей ядра.

2) *Микроядерная архитектура* - Часть слоев из многослойной архитектуры(базовые механизмы и ниже) остается в ядре, часть - переносится за преде-лы ядра.

+ Снижается размер кода ОС в RAM
+ Удобно для распределенных систем
+ Удобно для переходов между различными архитектурами(переписывается лишь часть кода)
- Много переключений между пользовательским режимом и привилегированным
- Проблемы с безопасностью(у кода многих компонентов ОС нет защиты от вмешательств пользовательского на аппаратном уровне)

3) *Гибридная архитекрура* - ядро, в котором возможна замена части кода с кода ядра на код, выполняемый в пользовательском режиме.

4) *Нано-ядро* - в ядре остается лишь обработка прерываний (иногда - низкоуровневые планировщики).

5) *Экзо-ядро* - попытка построения ОС над оборудованием с различными характеристиками.

Ядро -> принятие решений и межпроцессное взаимодействие.
Вне ядра -> взаимодействие с оборудованием.

7 Монолитная архитектура операционной системы. Подробное описание компонентов (слоев), их назначение и взаимодействие между собой. Достоинства и недостатки монолитной архитектуры ядра.

Монолитная архитектура - схема операционной системы, при которой все компоненты её ядра являются составными частями одной программы, используют общие структуры данных и взаимодействуют друг с другом путём непосредственного вызова процедур (весь функционал реализован в режиме ядра).

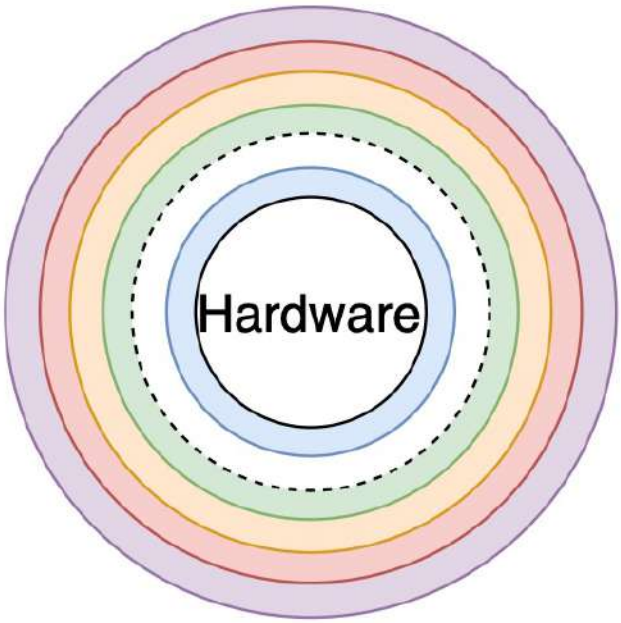
Слои :

Software
⇕
1) Main Program - интерфейс для внешнего пользовательского ПО (получение и переадресация сервису для выполнения системного вызова).
⇕
2) Services - принятие решений и их реализация с помощью утилит.
⇕
3) Utilities - протокол взаимодействия с контроллером соответственного экземпляра ПО.
⇕
Hardware

+ быстрота взаимодействия
+ безопасность (обращение пользовательского ПО только к main program)
- надежность (Много кода работает в привилегированном режиме ⇒ сбой в одном из компонентов может нарушить работоспособность всей системы)
- занимает много RAM;
- при замене части кода ядра необходима пересборка

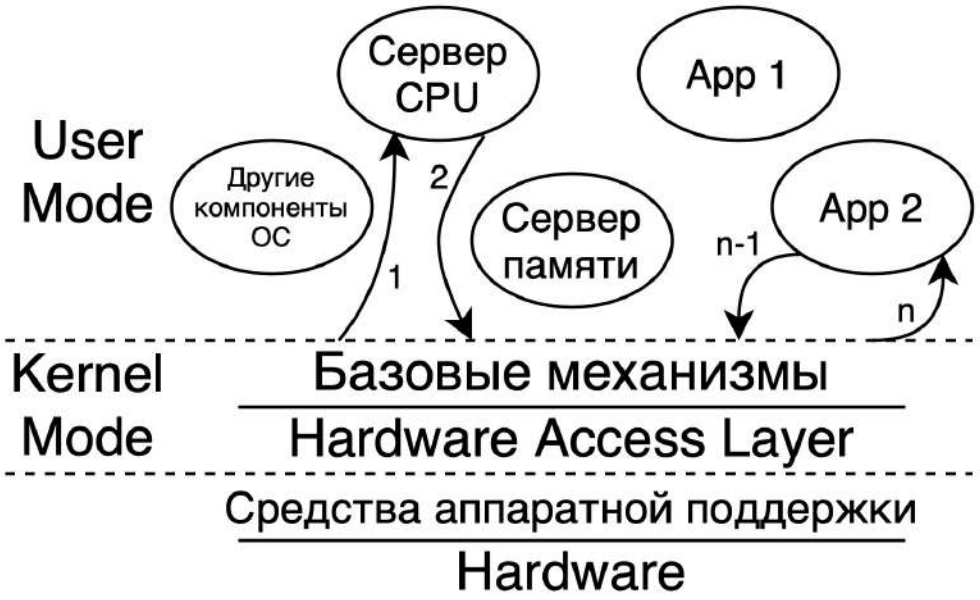
8 Концепция многослойного ядра операционной системы. Подробное описание слоев, их назначение.

Структура ОС, представленная рядом слоев (каждый слой обслуживает вышележащий слой, выполняет для него некоторый набор функций).



9 Микроядерная архитектура операционной системы. Подробное описание компонентов, их назначение и взаимодействие между собой. Достоинства и недостатки микроядерной архитектуры ядра.

Часть слоев из многослойной архитектуры остается в ядре, часть - переносится за пределы ядра.



+ уменьшение объемов памяти, которые выделяются для ОС
+ удобство в построении распределенных систем
+ быстрота и удобство при переходе между различными архитектурами
- количество переключений между user и kernel mode сильно снижают производительность
- проблемы с надежностью и безопасностью (при вынесении кода в user mode снижается надежность системы в целом; сложнее обеспечить безопастность из-за отстутствия аппаратной защиты некоторых компонентов ОС).

10 Понятия процесса, потока, нити, задания. Их определения, назначение и различия между собой.

Process (Процесс) - совокупность набора исполняющихся команд, ассоциированных с ним ресурсов и контекста исполнения, находящиеся под управлением ОС.

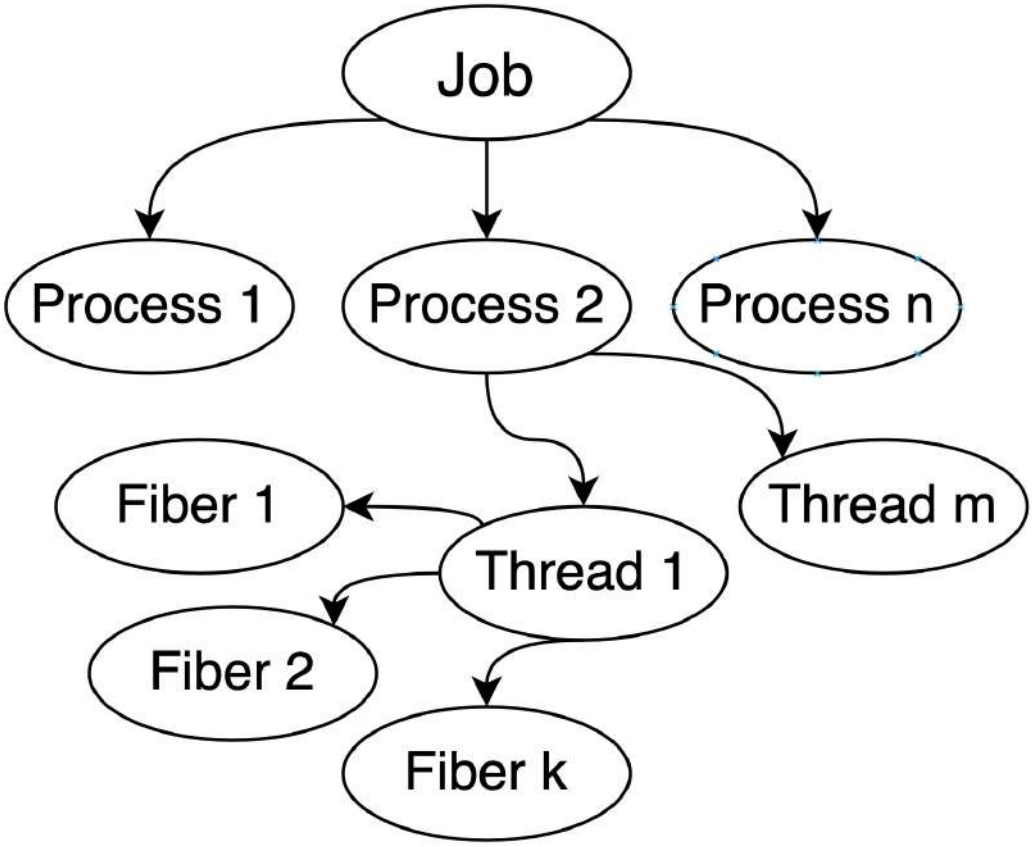
Создание процесса ⇒ создание Process Control Block(PCB или дескриптора процесса) - представления процесса в памяти ОС.

Thread(Поток) - отдельный набор команд и контекст, имеющие доступ к одному адресному пространству (более широко - к ресурсам в целом)

Проблема: Многозадачность потоков в большинстве случаев опирается на механизмы ОС для переключения между потоками, что может повлечь за собой неэффективное использование ресурсов(сама выполняемая задача не имеет управления над процессом своего выполнения)

Волокно (нить) - облегченный поток(в какой-то мере подпоток) - позволяет решить проблему кооперативной многозадачности: само волокно отдает управление либо следущему волокну, либо волокну-диспетчеру. Таким образом выполняемая задача сама управляет своим выполнением

Job (c-group/задание/работа/контрольная группа) - "надпроцесс" - совокупность исполняемых операций над данными, набор квот для ресурсов.



11 Функции подсистемы управления процессами.

1. Создание процессов и потоков

2. Обеспечение ресурсами процессов

3. Изоляция процессов друг от друга

4. Планирование процессов и потоков на разных условиях. Решение в каком доступе и объеме необходимо предоставить доступ к ресурсам

5. Диспетчеризация - переключение процессов между различными исостояниями; движение по жизненному циклу процесса

6. Взаимодействие между процессами

7. Синхронизация - обеспечение одновременного исполнения условий исключений, наличие прогресса, отсутствие голодания и др

8. Уничтожение процессов в конце их работы

12 Методы создания процессов в различных операционных системах. Структуры данных о процессах

Структуры данных:

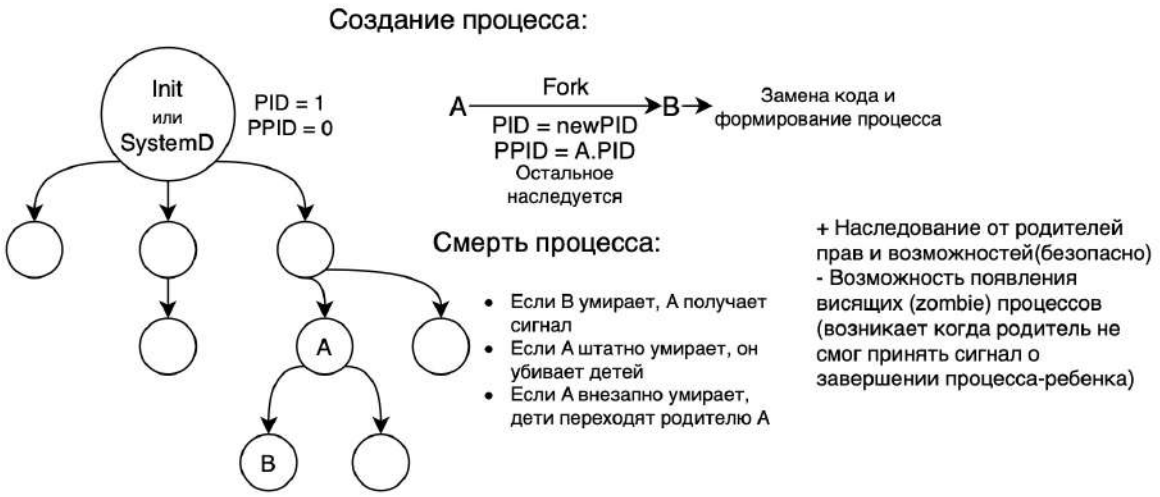
Process Control Block или дескриптор:

1. идентификация процессов (PID, часто PPID - UID и др.)
2. информация о состоянии процесса (статус и контекст)
3. история процесса

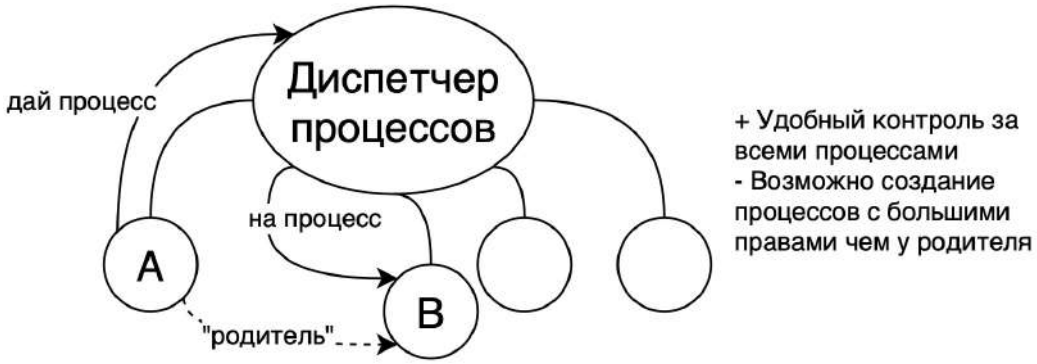
Иерархия процессов(некоторый граф, часто - дерево)

Создание процессов в различных системах:

Linux:

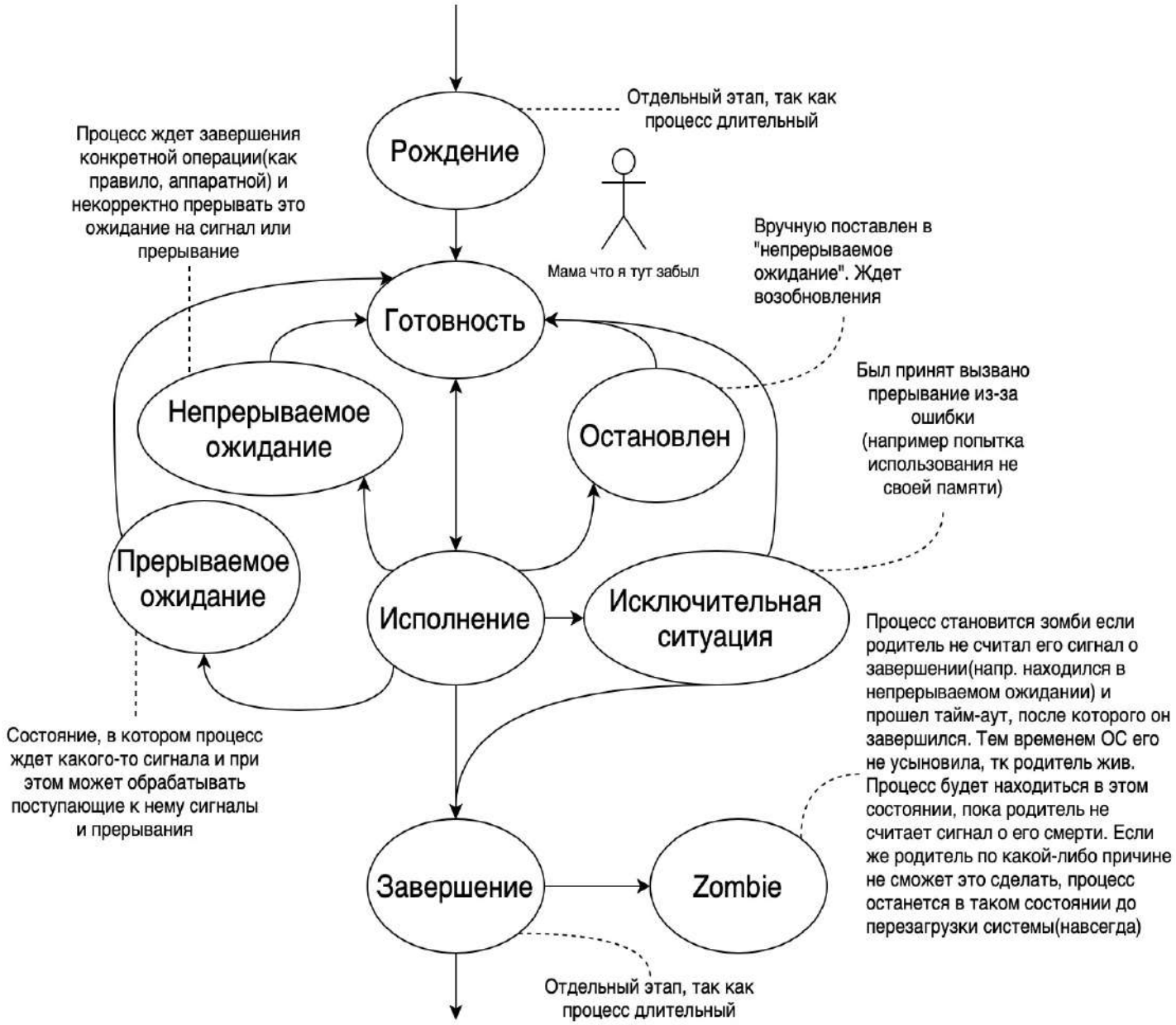


Windows:



13 Модель жизненного цикла процесса: состояния процесса, правила переходов между состояниями

Linux:



14 Виды планирования и их место в жизненном цикле процесса

Планирование - (исторически) распределение CPU-time между процессами при соблюдении требований

Проблема: Есть набор процессов из которых выстраивается очередь, но при этом реальное время исполнения - случайная величина, + система открытая ⇒ ее состояние постоянно меняется ⇒ изначальная очередь(особенно длинная) может стать неэффективной

Горизонт планирования - число процессов, после выполнения которых вызывается прерывание и происходит перепланирование. "Дальновидность"планирования.

Проблема: Планирование занимает время. Часто планировать - эффективно, но дорого. Редко - дешево, но неэффективно.
⇒ *Решение*: Разное планирование в разных местах

- Краткосрочное планирование: быстрое, не самое эффективное, но это не страшно, так как горизонт планирования невелик
  Например: Этапы готовность ↔ исполнение. Удобно, так как кванты времени для каждого процесса на CPU очень малы

- Среднесрочное планирование: не самое быстрое, более эффективное, средний горизонт планирования.
  Например: Этап ожидания (ожидание в RAM ↔ ожидание вне RAM). Если мы знаем, что процесс в ближайшее время будет ждать, мы можем заменить его в RAM на другой процесс

- Долгосрочное планирование: самое эффективное, хоть и долгое, но это не страшно, так как оно происходит не так часто
  Например: Этапы рождение ↔ готовность. Если мы позволим процессу родиться, он будет использовать ресурсы машины долгое время, поэтому здесь планирование должно быть максимально эффективно

15 Критерии эффективности и свойства методов планирования процессов, параметры планирования процессов

Критерии планирования:

- Справедливости - гарантия каждому заданию равной доли CPU-time или другого ресурса системы
- Эффективности - раздача ресурсов по потребности. Максимальная загруженность ресурсов

Критерии эффективности:

  - Сокращение полного времени выполнения(от рождения до смерти)
  - Сокращение времени ожидания
  - Сокращение времени отклика

Свойства методов планирования:

- Предсказуемость - на одинаковых данных одинаковые или схожие результаты
- Минимальные накладные расходы - низкое время выполнения
- Масштабируемость на количество возможных ресурсов

Параметры планирования:

	Статические	Динамические
Системы	Предельные значения ресурсов системы(характеристики CPU, RAM и т.д.)	Сведения о свободных ресурсах(средняя нагрузка на CPU, загруженность канала связи, I/O и т.д.)
Процесса	Права доступа, важность процесса, ограничения в целом	CPU-Burst(Сколько тактов нужно для полного выполнения процесса в изоляции от других), I/O-Burst(Общий объем работы с I/O) и т.д.

16 Методы планирования без внешнего управления приоритетами (FCFS, RR, SJF), гарантированное планирование: описание, преимущества и недостатки

Невытесняющее планирование - если процесс начал выполняться он может прервать себя лишь сам (завершиться или уйти в ожидание)

Вытесняющее планирование - есть механизм прерывания выполнения текущего процесса для начала выполнения другого

$\tau_{\text{полное}}$ - Полное время выполнения всех процессов от начала первого до конца последнего
$\hat{\tau}_{\text{исполнения}}$ - Среднее для всех (время ожидания в готовности + время выполнения)
$\hat{\tau}_{\text{ожидания}}$ - Среднее время ожидания

- First Came First Served(FCFS, кто первый встал того и тапки)

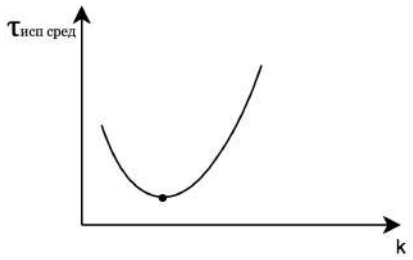
  \+ Не вызывает голодания процессов

  \+ Эффективен если процессы в очереди выстроены в порядке возрастания полного времени выполнения

  \- Максимально неэффективен если процессы в очереди выстроены в порядке убывания полного времени выполнения

  \- По сути эффективность - случайная величина

- Round Robin(RR, Круглый Робин(карусель))



  \+ Прост

  \+ Очень эффективен при правильном кванте

  \- При неправильном кванте вырождается в FCFS (если очень большой), или душит накладными расходами на переключения (если оч малый)

- Shortest Job First (Кто короткий тот вперед).

  Начинает выполнять самый быстрый процесс в течение определенного кванта, затем, если есть более короткий процесс переключается на выполнение его

  \+ Дает кратчайшее время ожидания

  \+ Оптимален для пакетной обработки когда время ожидания не критично

  \- Провоцирует голодание длинных процессов

- Гарантированное планирование

  N - количество процессов
  T_i - время сеанса i-го процесса
  τ_i - время выполнения i-го процесса
  $\tau_i \sim \frac{T_i}{N}$, $R = \frac{\tau_i \cdot N}{T_i}$

  \+ Разделяет ресурсы между процессами поровну

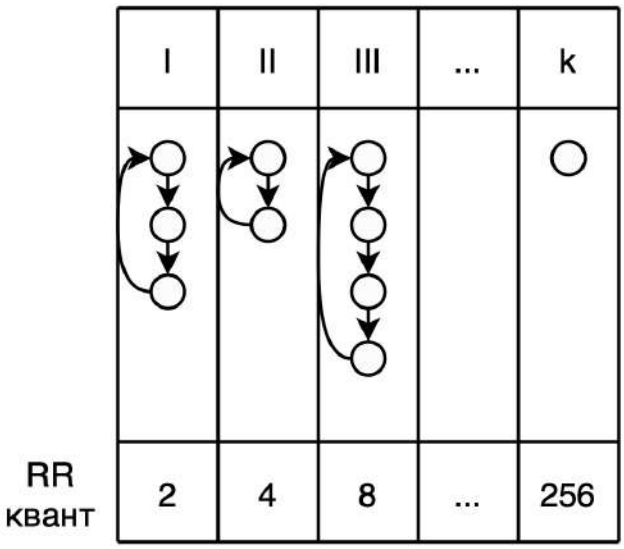
  \- Как следствие часто неэффективен

  \- Сложные вычисления коэфициента справедливости R

17 Приоритетное планирование с внешним управлением приоритетами, многоуровневые очереди. Описание методов, их достоинства и недостатки

Приоритетным планированием - планирование, при котором каждому процессу присваивается определенное числовое значение – приоритет.
Приоритет может задаваться как внутренними критериями системы (например, длительность CPU-Burst для SJF), так и внешними по отношению к системе (например, важность для пользователя).

Многоуровневые очереди:



Несколько очередей с разными (пользовательсими) приоритетами
⇒ Проблема: голодание процессов низкого приоритета
⇒ *Решение*: после таймаутра повышеть приоритет процесса, а затем, после некоторого времени выполнения - отбрасывать назад

Многоуровневые очереди с обратной связью:
Для каждой очереди квант разный. Как только процесс рождается он считается хорошим и попадает в самую приоритетную очередь. Если он успевает выполниться (или уйти в ожидание) за этот квант его приоритет остается. Если нет, его приоритет понижается. При этом с понижением приоритета очереди увеличивается квант выполнения.

18 Организация планирования процессов в ОС семейств Microsoft Windows

Классы приоритетов:

1. Realtime 24
2. High 13
3. Above normal 10
4. Normal 8
5. Below normal 6
6. Idle 4

Уровни насыщения(для управления приоритетами):

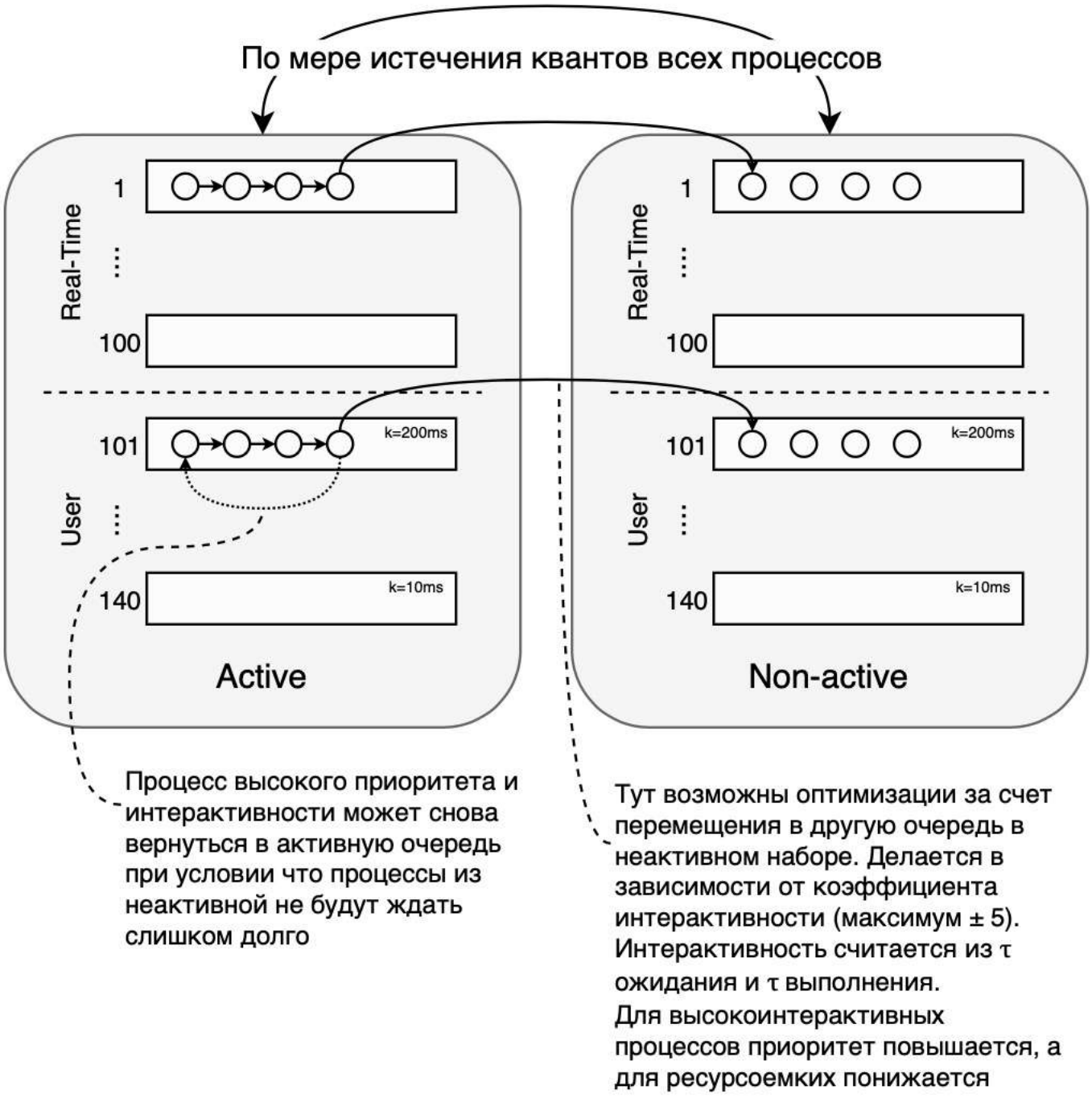
1. Time-critical +15
2. Highest +2
3. Above normal +1
4. Normal ± 0
5. Below normal -1
6. Lowest -2
7. Idle -15

Внешнее управление приоритетом:
Создание процесса $\xrightarrow{\text{система вызовов для потока}}$ изменение приоритета потока при этом между Real-time и Dynamic потоки переходить не могут.

Внутреннее управление приоритетом:
Для Dynamic система может менять приоритеты процессам в зависимости от их статуса. При этом интерактивным процессам повышается приоритет (так как они интерактивные). Если они успевают выполниться или уйти в ожидание за квант в очереди с более высоким приоритетом, они там и остаются. Если нет, их приоритет понижается на 1 до изначального.

19 Принципы работы планировщиков O(1) и CFS в операционных системах GNU/Linux

Linux O(1):



\+ 2 очереди, а значит нет голодания

\- Высокие накладные расходы на расчет интерактивности и других коэффициентов

Linux CFS:
Для каждого процесса существуют две переменные: exec_time(время, которое процесс в сумме выполнялся) и max_exec_time(время, которое процесс будет выполняться в следующий раз, когда попадет на CPU). max_exec_time рассчитывается так, чтобы было "честно"по отношению к другим процессам. Фактически: $\frac{\tau_{ожид}}{N}$

Массив процессов упорядочен по возростанию exec_time. Следующим на выполнение выбирается процесс с наименьшим exec_time. Во время выполнения exec_time растет. После достижения max_exec_time процесс перестает выполняться и улетает обратно в массив в соответствии со своим exec_time (чтобы массив оставался упорядочен).

Внешнее управление приоритетами достигается за счет изменения max_exec_time и более приоритетные процессы просто получают больше времени на выполнение когда придет их очередь.

В силу используемых структур данных (в частности - красно-черного дерева), доставание процесса осуществлсяется за O(1), а вставка и удаление за O(log(n))

Еще существует понятие **гранулированности** - минимальная величина max_exec_time. Таким образом снижение общих накладных расходов на переключения при высоких загрузках.

В обоих планировщиках своя очередь для каждого CPU. Раз в определенный промежуток времени происходит ребалансировка процессов по процессорам

20 Взаимодействие процессов. Условия взаимоисключения и прогресса. Понятие критической секции. Голодание процессов

Внешне-определенное взаимодействие - обеспечение возможности процессам обмениться данными или управлением.

Вынужденное взаимодействие - ситуация при которой процессы начинают конкурировать за неразделимый ресурс.

Необходимые условия которые должны обеспечиваться при взаимодействии процессов:

- Взаимоисключение доступа процессов к неразделяемому ресурсу
  Критическая часть кода относительно ресурса - та часть кода которая непосредственно осуществляет взаимодействие с неразделимым ресурсом
  $\Rightarrow$ Задача: два или более процессов не могут одновременно находиться в критических частях кода относительно одного и того же ресурса
  $\Rightarrow$ *Решение*: Пролог и эпилог - обозначения точек кода начала и конца взаимодействия с неразделимым ресурсом (для сообщения ОС о начале и окончании работы с ним)

- Прогресс
  Не должно возникнуть ситуации когда(одновременно):
  - Ресурс свободен
  - Есть процесс, которому нужен данный ресурс для работы
  - Процесс не может использовать данный ресурс

  Причины: недостатки алгоритмов планирования или механизмов их реализации

- Отсутствие **голодания** - длительного (потенциально бесконечного) отсутствия доступа у процесса к запрашиваемому ресурсу

- Отсутствие тупиков - ситуаций, когда процессы не могут ни получить запрашиваемый ресурс и ни при этом разблокировать уже занятый ими ресурс, чтобы дать возможность другим процессам выполниться и потенциально разблокировать первый ресурс.

21 Алгоритмы реализации взаимоисключений. Формальное описание алгоритмов, их недостатки

Подходы к решению задачи по планированию с соблюдением необходимых условий взаимодействия процессов:

Аппаратный уровень:

- Идея: Дать ресурсу право не отдавать ресурс пока работа с ним не закончена
  Проблема: Вытесняющая многозадачность ⇒ нельзя.
- Идея: Запрет на прерывания во время выполнения критической части кода
  Проблема: Работа неэффективна (нарушается работа планирочщика) + потенциально неуправляемый сломавшийся процесс и вечно заблокированный ресурс

Алгоритмы взаимоисключения:

1. Замок

```
1 shared int lock = 0;
2 P_i() {
3     //Some code here...
4     while (lock); //Wait if
          closed
5     lock = 1;
6     {/*Critical section...*/}
7     lock = 0;
8     //Some code here...
9 }
```

Если произойдет прерываение P_0 на между строками 4 и 5, то P_1 сможет начать выполнение критической секции.
При этом если произойдет его прерывание в ней, в свою критическую секцию перейдет и P_0
⇒ Нарушение взаимоисключения

2. Строгое чередование

```
1 shared int turn = 0;
2 P_i(){
3     // Some code here...
4     while (turn != i); // Wait
          if not its turn
5     {/*Critical section...*/}
6     turn = 1-i; // Change turn
7     // Some code here...
8 }
```

Возможно, что P_0 завершил работу с ресурсом и передал его P_1, но тот ждет и ресурс не использует.
За это время процессу P_0 может снова потребоваться ресурс, но он его не получит, так как P_1 еще с ним не отработал.
⇒ Нарушение прогресса

3. Флаги готовности

```
1 shared int ready[2] = {0,0};
2 P_i() {
3     // Some code here...
4     ready[i] = 1;
5     while (ready[1-i]); // Pass
          if others are ready
6     {/*Critical section...*/}
7     ready[i] = 0;
8     // Some code here...
9 }
```

Если произойдет прерываение P_0 на между строками 4 и 5, то P_1 встанет в готовность, увидит, что есть другие и будет ждать. Когда снова начнет выполняться P_0 он увидит, что есть процессы, которые готовы и так же будет ждать
⇒ Тупик

4. Алгоритм Петерсона

```
1  shared int ready[2] = {0,0};
2  shared int turn = 0;
3  P_i() {
4      //Some code here...
5      ready[i] = 1;
6      turn = 1 - i;
7      while (ready[1-i] && turn == 1-i);
8      {/*Critical section...*/}
9      ready[i]=0;
10     //Some code here...
11 }
```

Все условия выполняются, но долго ⇒ возврат к аппаратному подходу

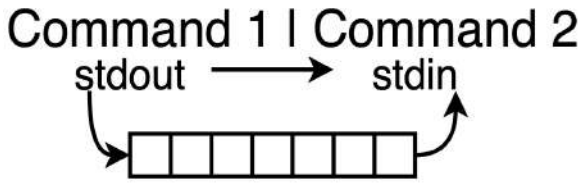
Решение: Создание новой сложной команды команды CPU, так как для решения пробелмы достаточно запретить прерывания на строках 4-5 алгоритма замка.
Новая команда Test&Set опрашивает переменную, и если ее значение равно нулю, то ей присваивается единица и одновременно возвращается true. Иначе возвращается false и изменения переменной не происходит
⇒ Test&Set - аппаратная поддержка условия взаимосиключения

Обновленный замок:

```
1 shared int lock = 0;
2 P_i() {
3     //Some code here...
4     while (!test_and_set(lock)); //Wait if closed
5     {/*Critical section...*/}
6     lock = 0;
7     //Some code here...
8 }
```

22 Семафоры Дейкстра. Решение проблемы «производитель-потребитель» с помощью семафоров

Пусть результат работы одной команды идет на вход другой.



Возникают потенциальные проблемы:

- одновременные запись и чтение (так как они не мгновенны - неконсистентность данных)
- переполнение буфера
- чтение пустого(необновленного) буфера

Семафор - целая неотрицательная переменная
Механизм семафора Дейкстры:

Существуют 2 операции:

```
shared int s = 0;
p(s) { // Probing
    while (s == 0); // The resource stays blocked
    s = s - 1;
}
v(s) { // Increment
    s = s + 1;
}
```

Решение проблемы производителя и потребителя при помощи семафоров:

Объявление семафоров:

```
shared Semaphore mutex = 1;
shared Semaphore empty = N; // number of elements in buffer
shared Semaphore full = 0;
```

```
Producer() {
    while (1) {
        produce_data();
        p(empty);
        p(mutex);
        put_data();
        v(mutex);
        v(full);
    }
}
```

```
Consumer() {
    while(1) {
        p(full);
        p(mutex);
        get_data();
        v(mutex);
        v(empty);
    }
}
```

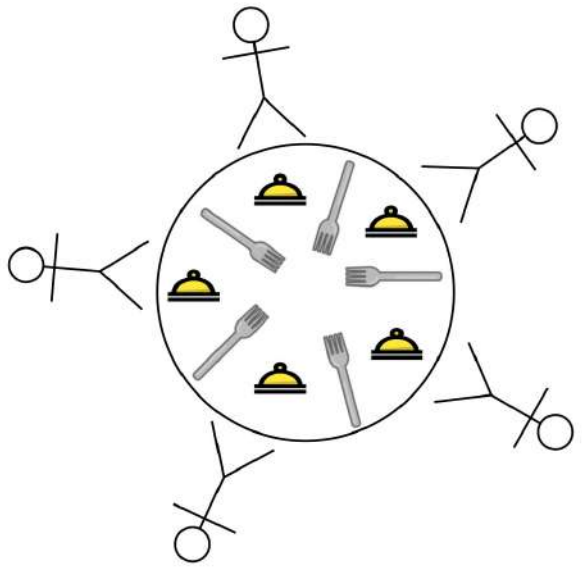
23 Проблемы взаимодействующих процессов. Проблема обедающих философов, проблема писателей и читателей

Проблемы взаимодействующих процессов - обеспечение условий взаимоисключения, прогресса, отсутствия голодания, отсутствия тупиков.

Livelock - что-то делается, но беспродуктивно

Deadlock - вообще ничего не делается

Проблема обедающих философов



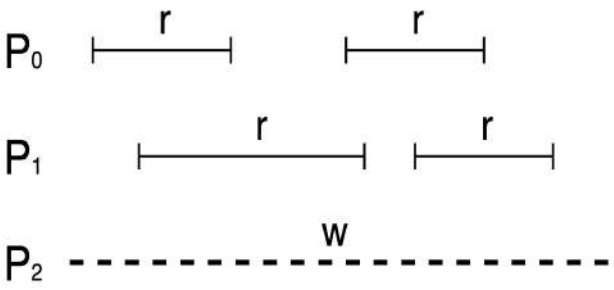
Для того шобы поесть пельмешей надо 2 вилки. При этом вилка - критический ресурс, не все могут одновременно его использовать.

⇒ *Решение*: Взять левую вилку, если правой нет - положить левую и ждать.

Однако даже при случайном времени ожидания есть шанс, что случится lock

⇒ *Решение*: Официант (ОС)

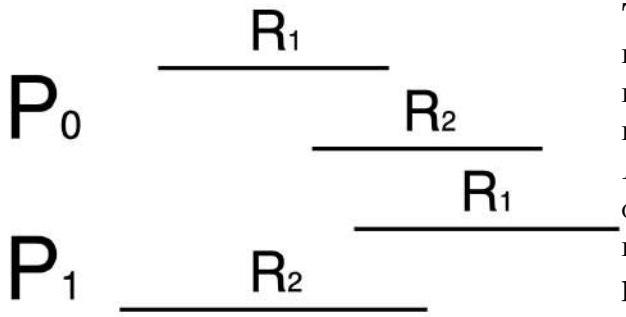
Проблема читателя и писателя



Проблема: Голодание записи
⇒ *Решение*: Единая очередь |r|r|r|w|r|r|r|r|w|
При этом чтения можно производить параллельно
Потенциальная проблема: Частое чередование
⇒ *Возможное решение*: Механизмы перегруппировки

Однако такое решение является частичным, и возможно лишь тогда, когда чтение и запись не связаны

24 Тупики. Условия возникновения и методы борьбы с тупиками



Тупик - ситуация, в которой для продолжения выполнения процессу P_0 нужен заблокированный ресурс R_2, и при этом сам этот процесс держит другой неразделимый ресурс R_1. Процессу P_1, который заблокировал ресурс R_2, для продолжения нужен ресурс R_1, который заблокировал P_0. Таким образом образуется тупик, в котором процессы P_0 и P_1 держат друг другу нужные ресурсы и не отпускают.

Условия возникновения тупиков (по Кофману):

1. Mutual Exclusion
2. Hold&Wait
3. No Preemption (нет возможности отобрать)
4. Circle Await (Deadlock meets Round Robin)

3 пути:

1. Игнорировать → очень дешево и не так страшно
2. Предотвращать → оптимально
3. Обнаруживать и восстанавливать → дорого

Пути предотвращения тупиков:

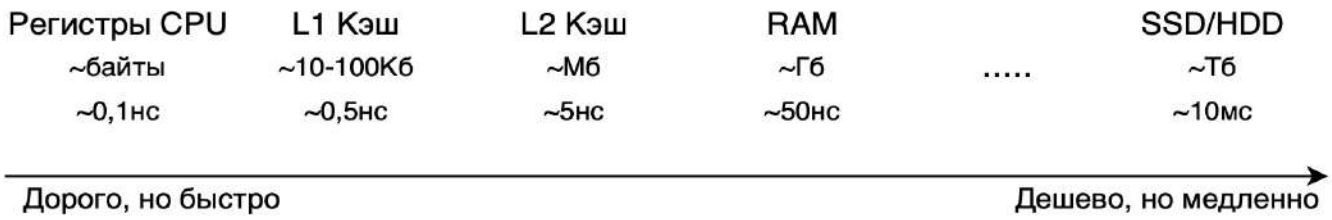
- Буферизация (против Mutual Exclusion)
- Блокировка всех неразделяемых ресурсов (против Hold&Wait)
- Отбитрать у слабых процессов ресурс, забуферизовав его состояние (против No Preemption)
- Нумеровать ресурсы и давать процессу только номером выше тех, что у него уже есть, периодически сдвигая номера (против Circle Await) → дорого

На самом деле все полумеры, хоть и везде частично реализованные. На практике часть тупиков все равно игнорируется.

25 Принципы управления памятью вычислительной системы. Виртуальная память и преобразование адресов

3 свойства компьютерной памяти

1. Чем быстрее тем дороже
2. Чем выше объем тем дешевле
3. Чем выше объем тем медленнее



Идея: **Виртуализация** - Расширение RAM при помощи места в хранилище с созданием единого адресного пространства.

Своппинг - процесс переноса данных между хранилищем и RAM.
Подходы: Переносить все данные процесса или фрагментами?
Windows: Файл подкачки(работает через стандартный механизм ФС)

\+ Стандартные механизмы работы с файлами на диске

\+ Простое управление размером: файл может увеличиваться и уменьшаться

\- Работа через абстракцию файловой системы без ее фактического использования

\- Свойства и атрибуты файла не нужны, они определены заранее и не будут изменены

Накладные расходы и снижение надежности
Linux: Отдельный раздел диска

\+ отдельный раздел (надежно, безопасно, эффективно)

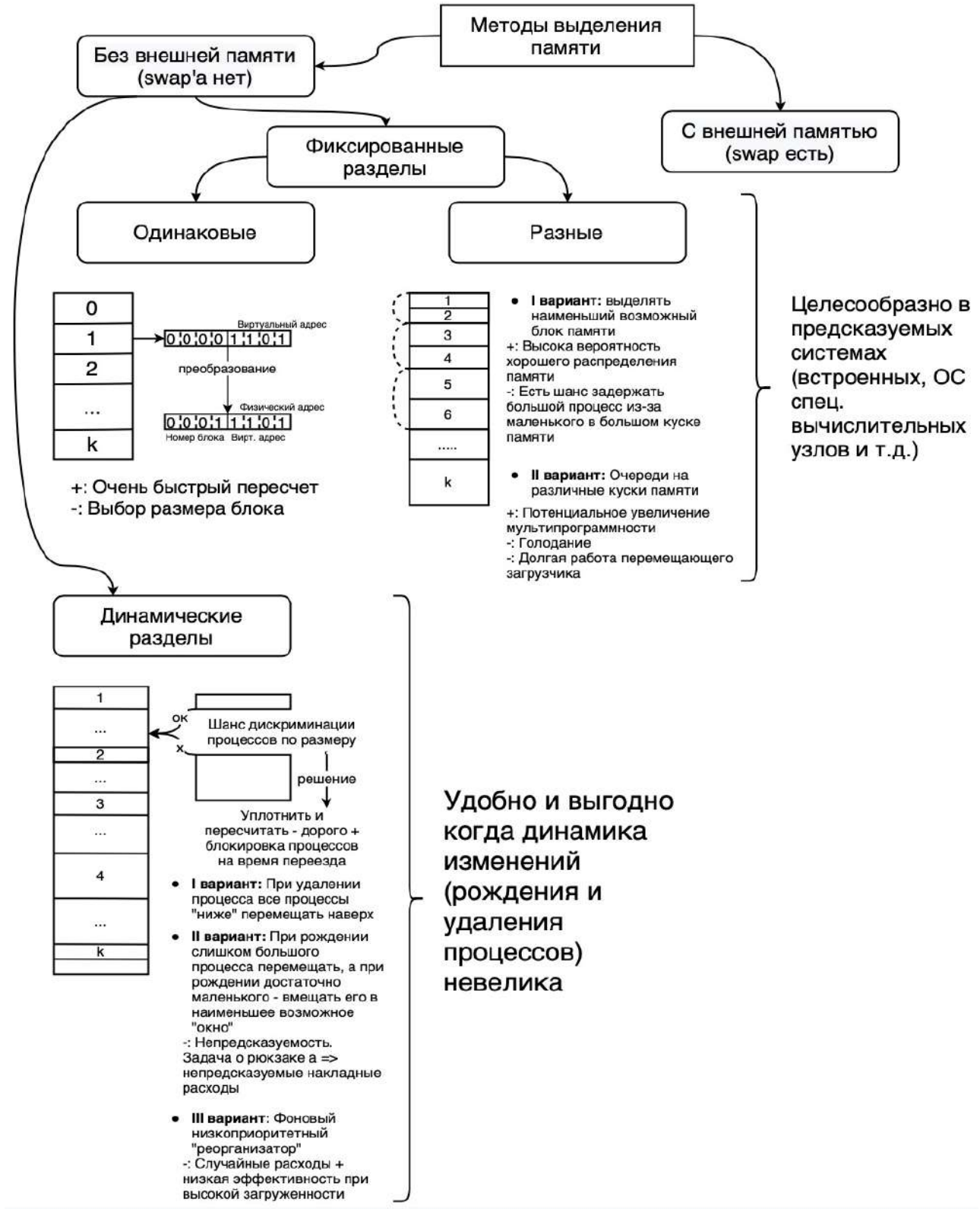
\- размер (разделять или переразделять диск сложно долго и дорого)

Пересчет адресов

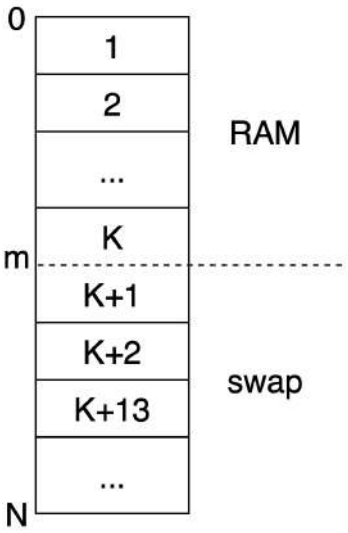
- Перемещающий загрузчик: Расчет адресов на этапе загрузки программы в RAM.
  - \+ Простая защита памяти
  - \+ Быстрое обращение
  - \- Долгая загрузка
  - \- Проблема при расположении фрагментами ⇒ Много (долгих) пересчетов при свопе

  Неудобное решение, если не использовать непрерывное пространство для всего процесса
- Динамическое преобразование: в момент загрузки адреса виртуальные, пересчитываются при обращении.

26 Методы распределения оперативной памяти без использования внешней памяти



27 Страничная организация виртуальной памяти. Вычисление физических адресов при страничной организации виртуальной памяти



Идея: Сегментное выделение как в динамическом разделении без swap'а, только с тем исключением, что сегмент может быть скинут в подкачку

Проблема: Как и в динамическом разделении - сложные расчеты + фрагментированность

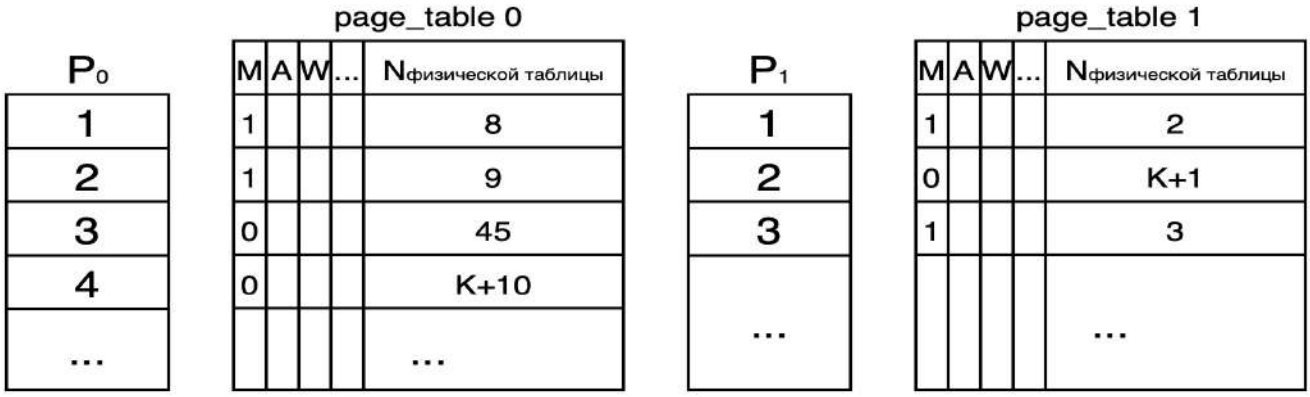
Наиболее общепринятый вариант - страничный подход:

Разделение памяти на страницы одинакового размера (1/2/4/...Кб - кратно степеням 2). Перерасчеты и размещения - тоже страницами.

Проблема: Как найти нужные страницы?

⇒ *Решение*: Использование таблиц для адресации.

Каждая строка таблицы строго соответствует странице в виртуальной памяти процесса



Адрес состоит из адреса страницы и смещения. При обращении адрес виртуально страницы при помощи page_table заменяется на адрес физической страницы.
Для навигации по PT используется лишь ее адрес в памяти и номер страницы в виртуальной памяти, тк строки в ней строго соответствуют

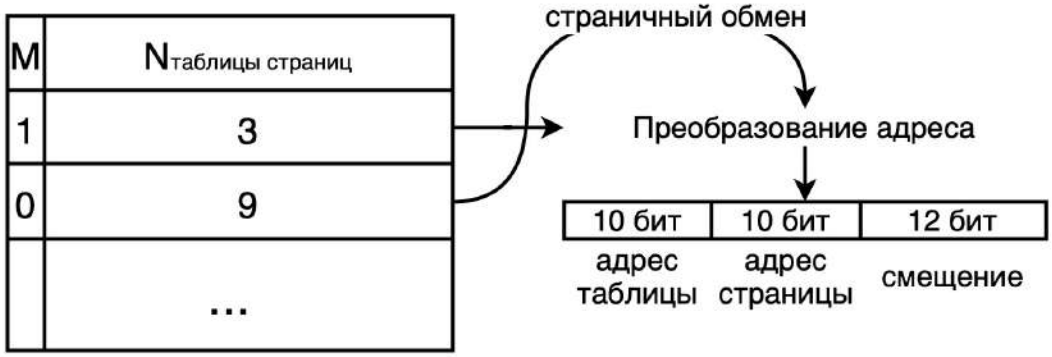
Если страница находится в подкачке, вызывается страничное прерывание, процесс направляется в непрерываемое ожидание, запускается механизм страничного обмена и после его завершения процесс направляется в готовность.

Биты заголовка:

- M - бит памяти. 1 - в RAM, 0 - в подкачке
- A - бит доступа. 1 - если к странице было обращение после страничного обмена. В случае если свободных страниц в RAM не хватает в первую очередь меняются страницы с A = 0. В Linux - списки неактивных страниц и своп их в первую очередь.
- W - write_bit - равен 1 если с момента последнего страничного обмена данные на странице изменились и ее предыдущую копию в свопе, если она есть, при перемещении страницы в него надо перезаписать. Если он равен 0, то страница при необходимости просто заменяется в RAM без долгой перезаписи свопа а вместо этого просто возвращается адрес ее старой копии.

Проблема: Пусть выделено по 4 байта на страницу в таблице. Тогда при 32-битной адресации размер таблицы страниц равен 4Мб. Для 200 процессов это уже Гигабайт ⇒ невыгодно

Идея: Иерархическая организация страниц. Разобьем каждую pt на блоки по 4Кб и будем хранить для каждого такого набора каталог таблиц страниц.



Таким образом мы сокращаем минимальный необходимый объем для хранения информации о страницах с 4Мб до 4Кб для каждого процесса

28 Методы оптимизации потребления ресурсов при страничной организации виртуальной памяти. Сегментно-страничная организация виртуальной памяти

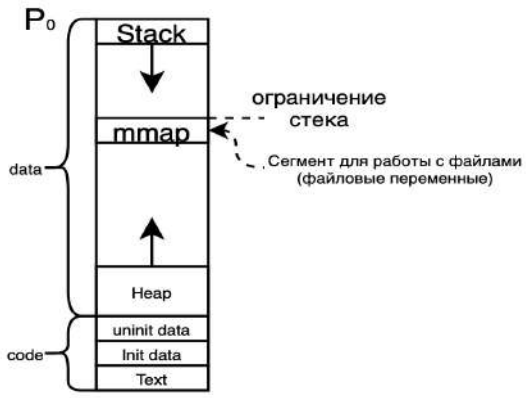
TLB-кэш:

- кэш для сохранения строк из pt к которым последний раз было обращение. Ассоциативный массив (номер виртуальной страницы ↔ строка в pt).

В общем случае CPU может за очень короткое время опросить весь массив TLB и получить необходимый адрес. Сложный расчет адреса лишь если строки в кэше нет. (Не так часто, так как чаще всего код и данные, которые он обрабатывает находятся на одной странице)

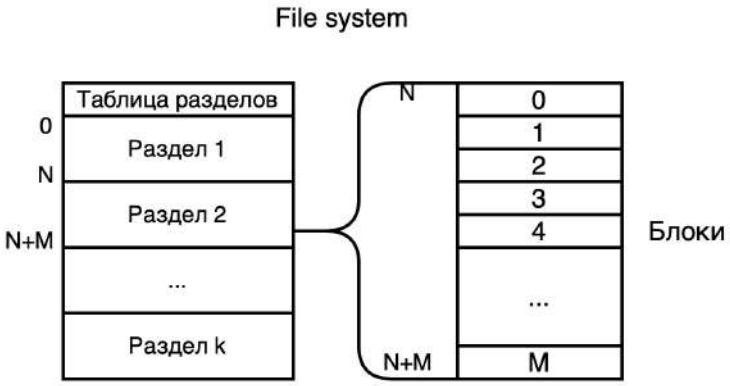
Сегментно-страничная организация:

Для ускорения работы для каждого сегмента держится его активная страница в памяти



Page-cache - набор страниц, в которых находятся данные назначенные для записи в файлы. Сохраняет время, но не очень надежно и безопасно, тк при потере питания часть данных может быть утрачена

29 Методы организации хранения данных в файловых системах: непрерывная последовательность блоков, связный список, таблица размещения файлов



- Непрерывная последовательность блоков

dir		
Name	N	Size
File 1	4	3000
File 2	7	1500

\+ Простая реализация

\- Фрагментация

Пример: CDFS

- Связный список
  Последними 4 байтами блока хранить ссылку на следующий блок
  - \+ Нет фрагментации
  - \- При потере одного блока теряются остальные
  - \- Низкая производительность
  - \- Неделимость размера блока или хранилища данных на 2
- Отдельная структура (Таблица размещения файлов)

Таблица размещения файлов

N	0	1	2	3	4	5	6	...	M
next N	null		4	2	eof	null		...	

dir		
Name	N	Size
File	3	3000

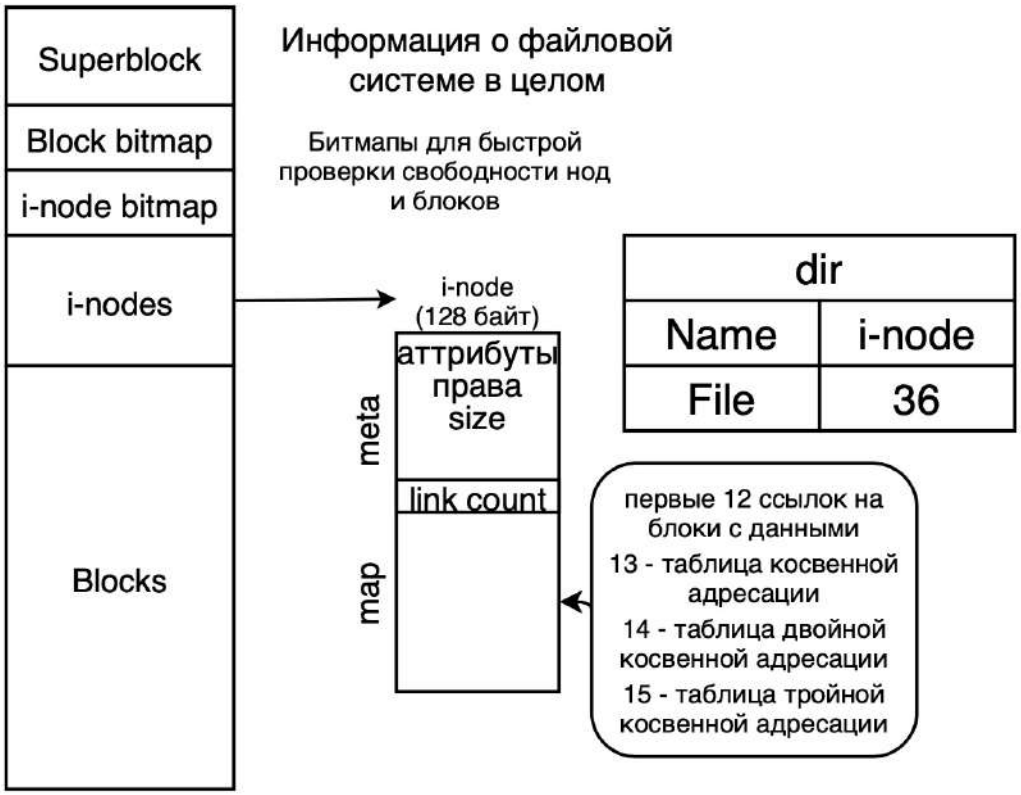
\+ Максимально простая ⇒ поддерживается всеми

\- При потере таблицы теряется все ⇒ резервные копии

\- Необходимо хранить всю таблицу в RAM

30 Методы организации хранения данных в файловых системах: индексные дескрипторы

На примере ext2:



+ Надёжность – потеряли 1 i-node – потеряли 1 файл, а не всё.
+ Каталог хранит только 2 поля: имя и i-node, без размера.
+ Эта модель позволила создать сетевой каталог.
+ Общение с файлами "по ссылке". Изменения в одном файле с одним именем заденут "другой"файл с другим именем в другом месте если их дескриптор одинаков
- Заранее ограничено количество файлов.

31 Журналируемые файловые системы. Назначение и виды журналов

Для того, чтобы при разных сбоях удержать данные, придумали идею составлять план (напоминает журналирование транзакций в базах данных) и по шагам выполнять план, постепенно удаляя строчки выполненных шагов из журнала. Тогда при сбое можно будет вычислить состояние, на котором случился сбой, и с него продолжить выполнять операции и тем самым завершить их успешно.

Журналируемая файловая система – файловая система, в которой осуществляется ведение журнала, хранящего список изменений, в той или иной степени помогающего сохранить целостность файловой системы при сбоях.

3 типа журналов:

- Journal - в журнал помещаются и блоки данных и мета. (высокая надежность но медленная работа)
- Ordered Journal - записывеются только метаданные, то есть информация об изменениях файловой системы, после работы с каждым блоком
- Writeback - записывеются только метаданные, то есть информация об изменениях файловой системы, до работы с каждым блоком

32 Обоснование необходимости и принципы построения распределенных ОС

Обоснования нобходимости распределенных ОС:

1. Производительность:

Проблема: упираемся в необходимоть повышении производительности
$\Rightarrow$ *Решение*: ставим больше процессоров
Проблема: нагрузка на контроллер памяти

2. Надежность:

Проблема: выход из строя единственного сервера ведет к погибели всей системы

3. География:

Проблема: требуется поддержка запросов и откликов с любой точки мира

$\models\Longrightarrow$ *Решение*: Пилим распределенную ОС

Принципы построения распределённых ОС:

1. Ни один узел не имеет полной информации о состоянии всей системы

2. Узлы могут принимать решения только на основе локально имеющейся у них информации

3. Выход из строя какого-либо из узлов не должно приводить к отказу алгоритма управления

4. Не должно быть явного или неявного предположения о существовании глобальных часов

Прозрачность:

Прозрачность означает, что какие-то действия, события, состояния данные недоступны пользователю. Выделяют 4 вида прозрачности которые должны быть в распределенной ОС:

1. Прозрачность расположения - приложение не должно знать, где находится его ресурс
2. Прозрачность миграции - приложение не должно знать о перемещении ресурсов
3. Прозрачность размножения - приложение не должно знать о количестве копий ресурса
4. Прозрачность конкуренции - приложение не должно знать о конкуренции за ресурсы

33 Алгоритмы управления памятью в распределенных ОС. Их преимущества и недостатки

1. Память с центральным сервером. Все разделяемые данные поддерживает центральный сервер.

   + Простота в реализации алгоритма
   - Сервер может стать узким местом

2. Миграция страниц. Страничный обмен с другим узлом в котором, рассполагается необходимая страница.

   + Консистентность данных - данные существуют в одном экземпляре
   - Trashing - два узла могут спорить за одну страницу и всё время перемещать её друг другу.
     $\Rightarrow$ *Частичное решение*: timeout после перемещения страницы

3. Метод размножение для чтения. Страничное копирование с другим узлом в котором, рассполагается необходимая страница

   + Производительность: возможность одновременного доступа по чтению
   - Большой расход памяти: храним дубликаты одних и тех же данных на разных узлах
   - Время: поиск данных.
     $\Rightarrow$ *Решение*: записать в список узы кому отдали страницу.
     Проблема: может быть очень большой связный список.
     $\Rightarrow$ *Решение*: у каждого узла запрашивать о наличии страницы.
     Проблема: дополнительная нагрузка на узлы.

   - Запись: противоречивые данные
     $\Rightarrow$ *Решение*: кидаем всем узлам сообщение убить страницу.
     Проблема: коллизия если два узла делают запись.

4. Метод полного размножение. Разрешим всем процессам вносить изменения и всем сообщать, что мы внесли изменения.

   - Модификации могут быть противоречивые
     $\Rightarrow$ *Решение*: сервер нумерации, который получает модификацию, присваивает номер странице и рассылает всем узлам

34 Методы управление файлами и каталогами в распределенных ОС. Их преимущества и недостатки

Методы управление файлами

1. Модель загрузки и разгрузки.

Файл передается между клиентом (памятью или дисками) и сервером целиком

- Не выгодные операции: перенести файлик 1 Гб ради изменения 1 бита

2. Модель удаленного доступа

Происходит системный вызов на данные из файла в файловую систему. Отдаем узлу страницы которые нам нужны.

- Сложность реализаци
- Тупики

Методы управления каталогами

1. Узел + путь. Все узлы имеют какие-то имена: /hostname/root/... внутри узла

+ Просто реализовать
- Нельзя иметь файлы из разных узлов в одном каталоге
- При перемещении файла нужно всем сообщать, что адрес поменялся

2. Монтирование. Каталог смонтирован так, что все узлы смонтированы в одно дерево в каждом узле.

+ Легче отслеживать копирование и перемещение
- Может быть циклический граф при неаккуратном монтировании

3. Единое пространство имён. Единый каталог, который одинаково выглядит на всех узлах и любые действия синхронизируется в него. Можем построить графое БД или другое NoSQL решение

+ Быстрый поиск
- Очень дорого, т.к нужно содержать БД

35 Синхронизация времени в распределенных системах. Метод Лампорта для синхронизации времени

Глобальные Часы:
На каждом сервере есть кварцевые часы, они достаточно точные, но за год, например, могут отстают на секунду.

Проблема: Процессы работают за микросекунды и отставание может вызвать сбой
$\Rightarrow$ *Частичное решение*: ставить на каждый сервер атомные часы

Проблема: дорого
$\Rightarrow$ *Частичное решение*: синхронизировать Time Server'а

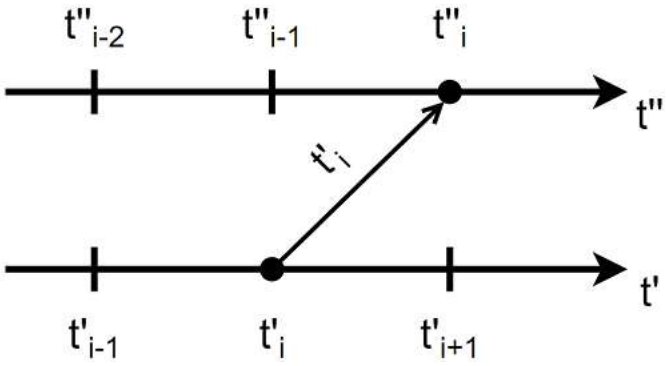
Проблема: сети не могут гарантировать одинаковую скорость доставки сообщения
$\models\Longrightarrow$ глобальные часы не нужны

Метод Лампорта:

Синхронизация времени возможна, используя только логическое время. Важно не астрономическое время, а только последовательность событий.
Событие A $\xrightarrow{\text{предшествует}}$ событие B $= \top \iff$ если B - реакция на A, то A произошла до B истина, когда:

- A и B произошли в пределах одного узла
- A - посылка из одного узла, а B - прием другим узлом

$t_i^n = t_{i-1}^n + \Delta t^n$



- $t_i^{'} > t_i^{''} \Rightarrow t_{i+1}^{''} = t_i^{''} + \Delta t^{''}$
- $t_i^{'} < t_i^{''} \Rightarrow$ Делаем перескок во времени на рассинхронизацию между узлами $t_{i+1}^{''} = t_i^{'} + \Delta t^{'}$

Чтобы $t_i^{'} \neq t_i^{''}$ для каждого узла добавляется уникальная дробная константа

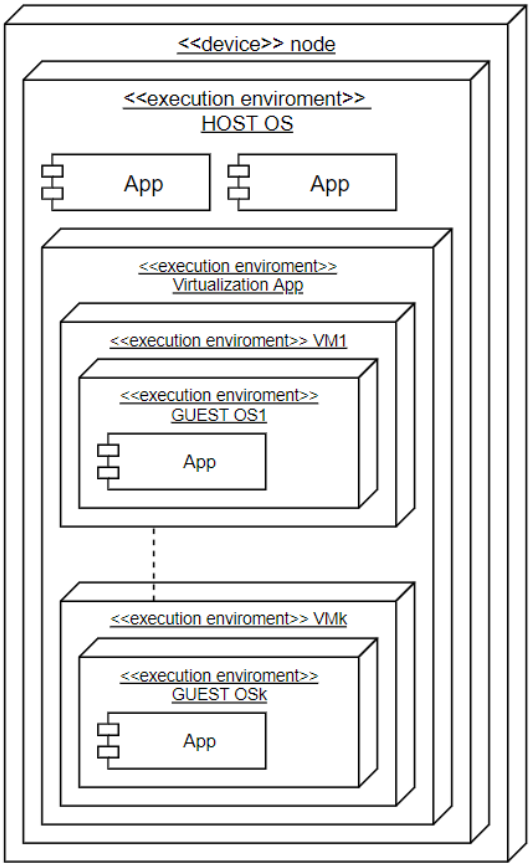
36 Технологии виртуализации. Виды виртуализации: эмуляция аппаратуры, полная виртуализация, паравиртуализация, виртуализация уровня ядра операционной системы. Их достоинства и недостатки

Технологии виртуализации решают задачи распределения pool'а ресурсов между множеством потребителей так, чтобы эти ресурсы были распределены по множеству узлов. Облачные решения основываются на технологиях виртуализации.
Изначальные основные задачи виртуализации:

1. Возможность поддержки устаревших ОС и приложений
2. Повышение отказаустойчивости и надежности
3. Созание средств для тестирования ПО
4. Консалидация серверов
5. Повышение управляемости и надежности сетевой инфраструктуры

Виды виртуализации

1. Эмуляция аппаратуры



А-ля наша машина и виртуалка с centOS

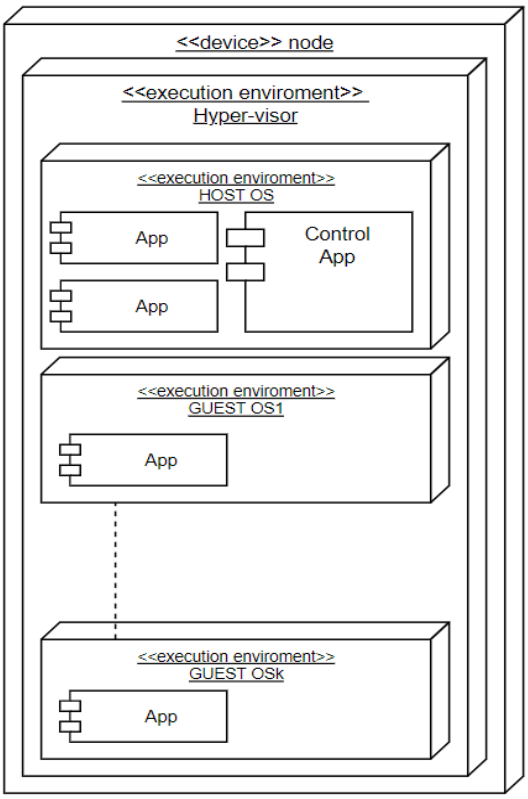
\+ Нет ограничений в архитектуре

\+ Можем работать не модифицируя приложение в GUEST OS

\- Производительность: приложение делает системный вызов в HOST OS

\- Память: пересчёт памяти несколько раз

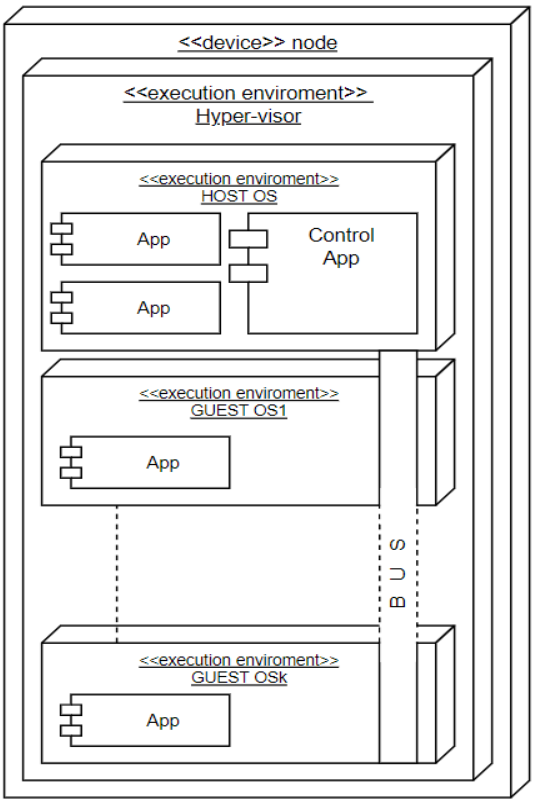
2. Полная виртуализация



HOST OS и GUEST OS с точки зрениия Hyper-visor'а находятся на одном уровне. Hyper-visor - усеченная ОС, реализует распределение рессурсов (RR) между OS под его управлением работают.

+ Нет ограничений для архитектуры
+ Можем работать не модифицируя приложение в GUEST OS
- Потребность драйверов для Hyper-visor'а
- Архитектура HOST OS и GUEST OS должна быть совместима с архитектурой аппаратного узла

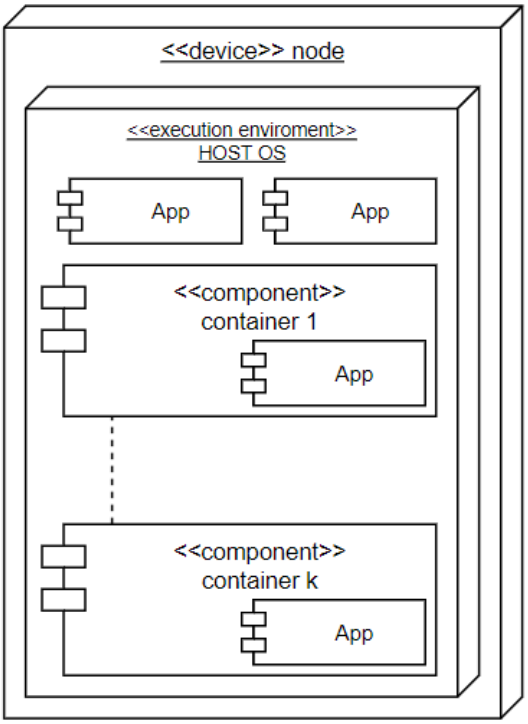
3. Паравиртуализация



Наличие шины между GUEST OS, вместо драйверов

+ Нет ограничений в архитектуре
+ Можем работать не модифицируя приложение в GUEST OS
+ Производительность: более тесная интеграцию с гипервизором за счет шины
- Архитектура HOST OS и GUEST OS должна быть совместима с архитектурой аппаратного узла

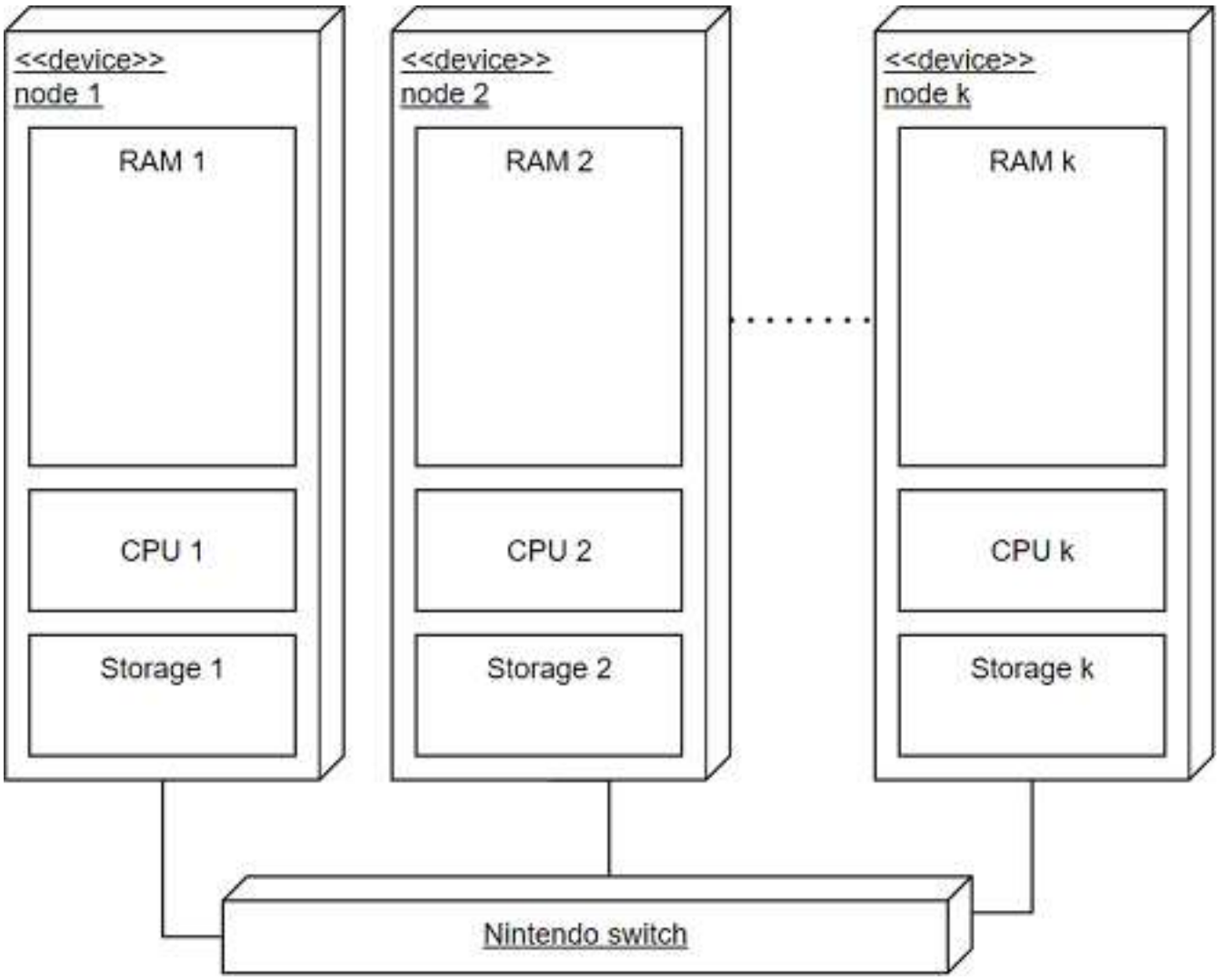
4. Виртуализация уровня ядра (Контейнеризация)



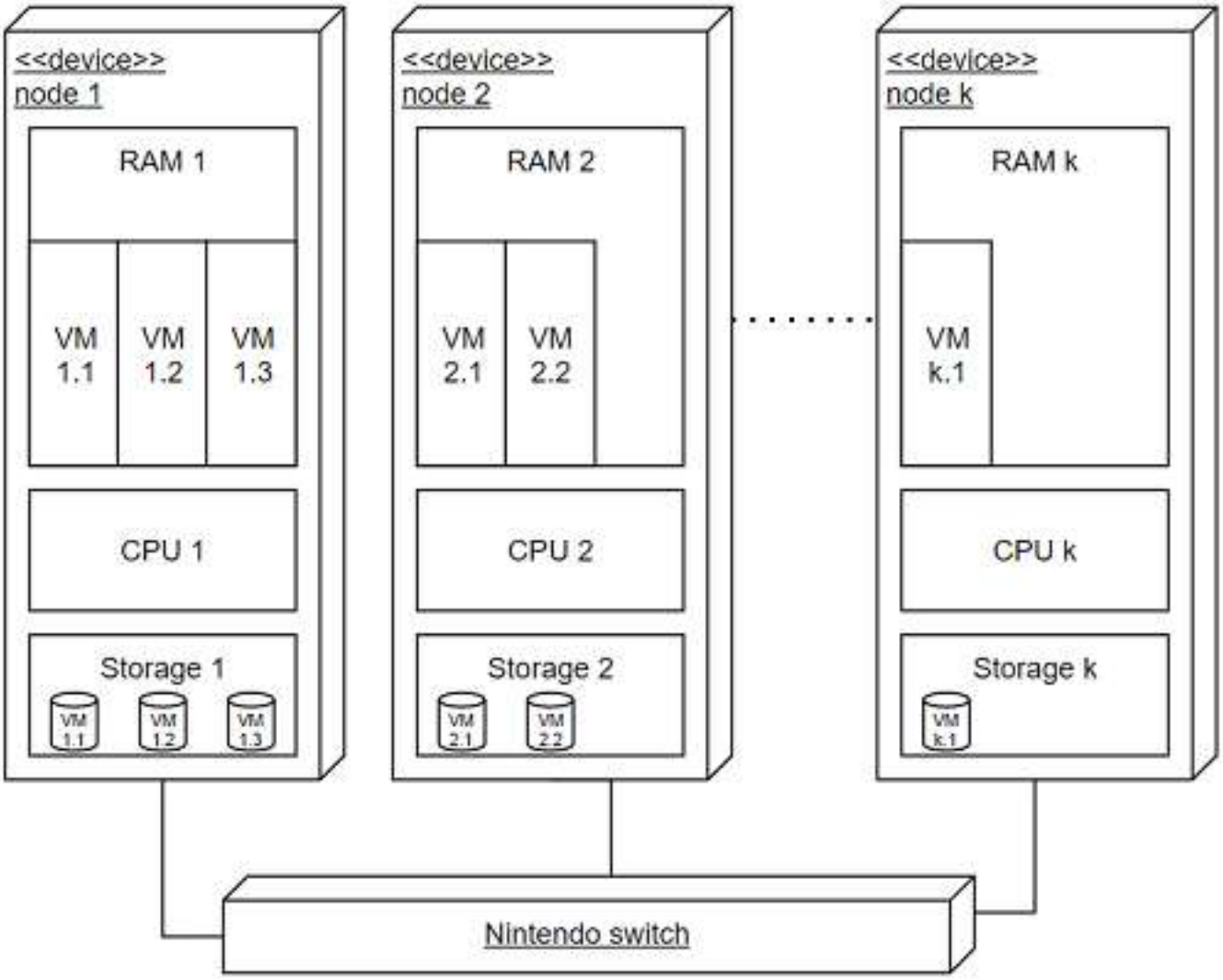
Виртуализация, при которой ядро является одним и тем же. Процессы внутри одного контейнера видят только друг друга и библиотеки

+ Производительность: высокая эффективность использования аппаратных ресурсов
- Сместимость: контейнеры должны быть совместимы по ядру, т.е отсутвует возможность запуска на одном узле ОС разного типа (Windows и Linux)
- Безопастность: сбой на уровне ядра затронет все компоненты

37 Архитектура облачных систем. Основные компоненты, их назначение и способы взаимодействия. Принципы мониторинга и управления производительностью в облачных системах



Проблема: При переносе процесса с одного узла на другой потребуется полность пересоздать структуру процесса, причем с учетом специфики ОС другого узла
⇒ *Решение*: воспользоваться полной виртуализацией/паравиртуализацией



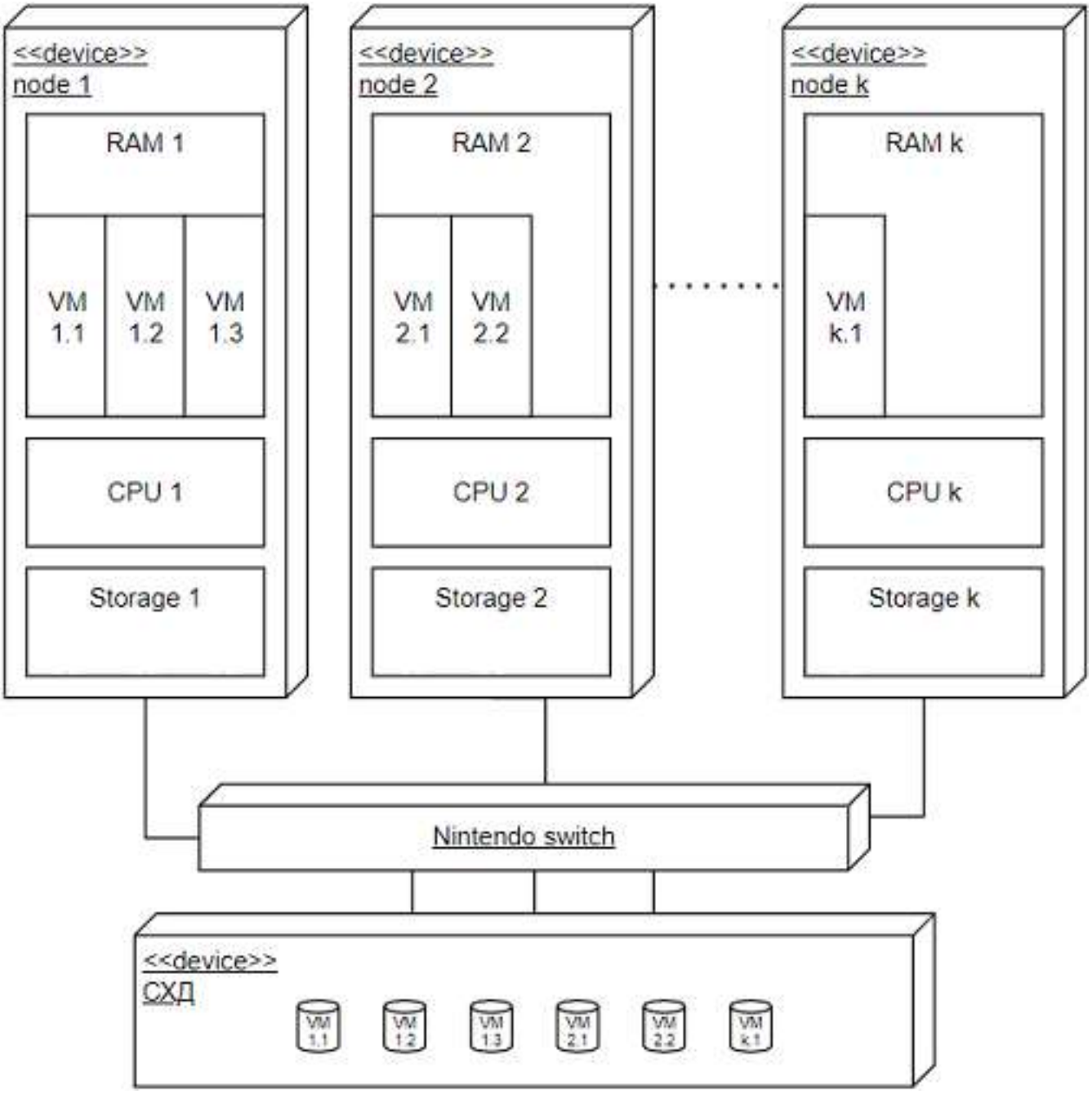
Можем спокойно переносить процессы вместе с адресным пространством с одного узла на другое

Проблема: снижение производительности т.к адресное пространство в одном узле, а образ остался в исходном

⇒ *Решение*: перенести образ системы

Проблема: ooooоччччень долгая передача, почти нереально

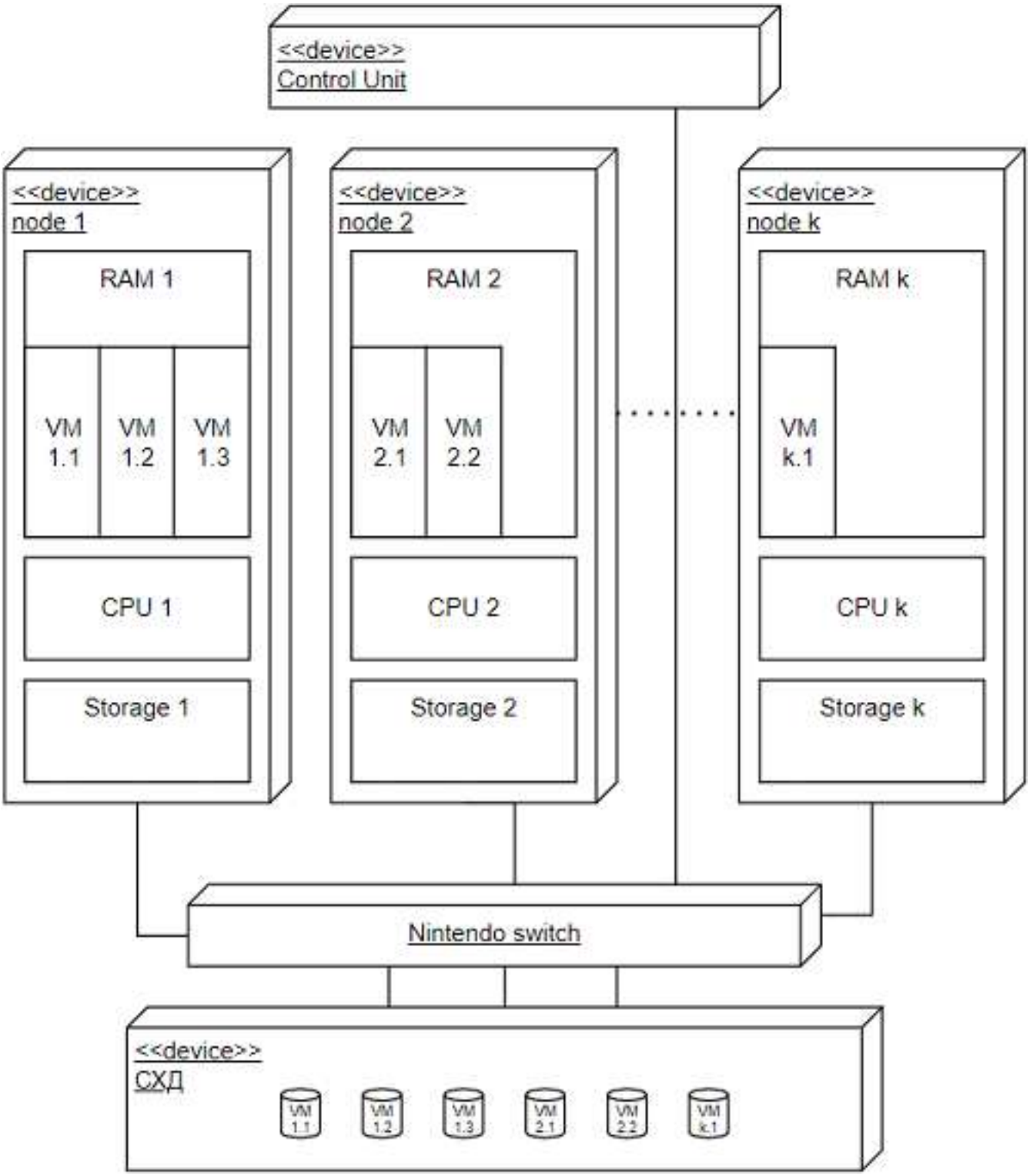
⇒ *Решение*: добавить СХД (Система хранения данных)



СХД обёртка под файловыми системами. Наружу представляется как диски, которые можно использовать и обращаться по сети – все виртуальные образы дисков можно туда перенести. Можно не переносить ничего, а просто говорить гипервизору, куда теперь обращаться.

Живая миграция - перенос виртуальной машины с одного узла на другое без остановки приложений и без прерывания сессий прикладных протоколов.

Для того чтобы корректно отслеживать ресурсы каждого узла и планировать переносы виртуальных машин добавляется аппаратный контролирующий узел



Control Unit Имеет свое СУБД, в котором хранит все сведения о виртуальных машинах, хранит сведения по результатам мониторинга.
Способы опросов узлов:

1. Night Calling - опрос каждого узла раз в ед. времени.

Проблема: нагрузка сети и трата ресурсов на сборку и отправку пакетов ("ну как там с памятью вопрос обстоит?")

2. Интеллектуальный [A]гент - в каждую систему (хостовую и виртуальную) помещается агент, который собирает данные и сообщает о превышении выставленных порогов (изменяемых, в зависимости от ситуации). AMQP протокол - инфраструктура для работы с сообщениями и очередями.