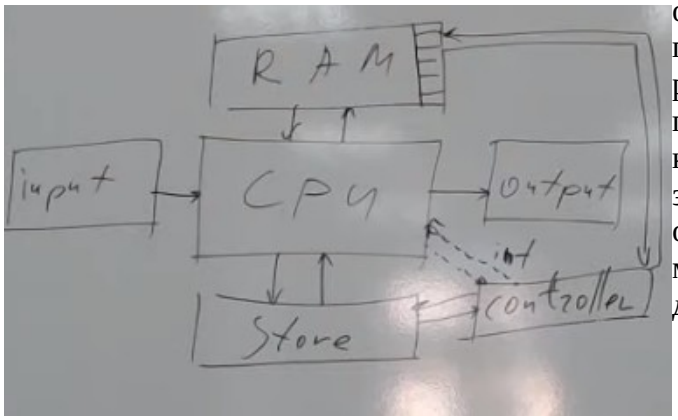
1. Функции и механизмы ОС, появившиеся на этапе программ-диспетчеров, предшественников операционных систем.

На этом этапе еще не существовало понятие OS.

1) Повторное использование кода, автоматизация загрузки и линковки. Выделим участок в памяти, где будем хранить код часто используемых программ и будем их вызывать когда надо. Появилась программа диспетчер, которая управляла другими программами, линковкой.

2) Задача оптимизации взаимодействия хранения и in out вывод. Все действия происходят через процессор, чтобы не было аварий и проблем. Но хотелось, чтобы данные



обрабатывались параллельно процессору, поэтому появился контроллер, который работал с памятью (пост и нет), а также появился механизм прерывание, это когда контроллер посылал процессору сигнал о завершении работы, тогда процессор остановит свою работу и переключится на микро код, который завершает обработку данных. Все это называется модель SPOOL

3) Писать монолитный код плохая идея, тогда возникает понятие пакета, который хранить константы, компоненты кода. Тогда надо уже загружать пакет в память, но почему бы и не несколько сразу, тогда возникает понятие очереди. Пакеты, которые выполняются быстрее стоит обрабатывать в первую очередь.

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

Каждая задача нагружает процессор и контроллер по-разному, поэтому чтобы никто не простаивал, надо нагружать их обоих разными задачами. Тогда возникает проблема, когда необходимо переключаться между командами в очереди и возвращаться в нужную точку, чтобы в памяти оказывались разные код и данные. Тогда появляется понятие OS.

1) Возникает необходимость делить время процессора на разные задачи. Заметим, что если бы они выполнялись последовательно, то потребовалось бы суммарно меньше времени, чем

так из-за дельт на переключение. Возникает понятие таймер — устройство на аппаратном уровне, которое генерит прерывания по времени к процессору. А уже обработчик прерываний решит проблему с переключением процессора между задачами.

2) Но к задачам надо переключаться, а кто где находится в памяти неизвестно, поэтому появляется понятие виртуальной памяти, когда каждая задача имеет «свою» память, а OS потом пересчитает как надо.

3) Появляется необходимость защищать область памяти каждого приложения. Добавляется еще один контроллер, который делает прерывание, если обращение в памяти идет не туда.

4) Планирование ресурсов

5) Задача универсального доступа к устройству хранения

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

Ввод вывод мб не единственным, а например удаленно можно подключиться, тогда появляется **терминал**, занимающийся вводом выводом. Но как понимать, кто, например, вводит данные?

1) Тогда появляются еще механизмы **идентификация** (появляется в системе информация о пользователе, например, хеш пароля), **аутентификации** (когда пользователь предъявляет свой идентификатор) и **авторизации** (когда, например, по табличке смотрим имеет ли пользователь право на запуск данного процесса).

2) Появляется сетевые OS, это когда мы можем перемещать процессы по сети другим компьютерам и уже там производить вычисления, потому что там больше ресурсов например.

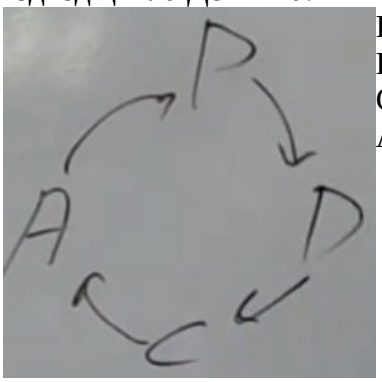
3) Чтобы не переписывать под новый ПК все компиляторы и прочее появляется необходимость в универсальной OS, но она получается должна быть написана на ЯП высокого уровня, но тогда проблема как написать OS на языке, которого в ней нет). Появилась UNICS универсальная система.

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

Функции OS:
1) Управление разработкой и исполнением пользовательского ПО: API, механизм обнаружения и обработка ошибок, мониторинг ресурсов, хранилище

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

1) Появляется несколько критериев для эффективной работы каждого узла, но все они не могут быть максимальны одновременно, поэтому использует сумму взвешенных решений. Это когда **result = A * k1 + B * k2 + C * k3…** где **A + B + C + … = 1** и тогда мы ищем эти **A. B…**, когда **result** будет максимальным. Но бывает важно, чтобы какой-то критерий оставался в определенном диапазоне, тогда это будет **условный критерий**.
2) Данную задачу приходиться всегда рассматривать в разном контексте, поэтому возникает подход цикла Деминга.



P — планирование, формируем правила, как мы распред ресурсы
D — выполняем план
C — проверяем полученные значения, они могли измениться
A — пытаемся исправить ситуацию

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

На архитектуру OS можно посмотреть с разных сторон: **функциональная**, **системная**, **программной архитектуры** и **архитектуры данных**.

Причины возникновения разных архитектур:

1) Производительность, надежность, безопасность. Нельзя все одновременно.
2) OS открыта, контекст применения меняется.

Решение, принятое относительно того, каким будет ядро OS является основным для отличия этих OS.

Ядро OS — это та часть ее кода, которая отличается 2 хара-ами и присущими только ей: **резидентность** (код ядра находится в RAM всегда и в неизменных адресах) и **привилегированный режим**.

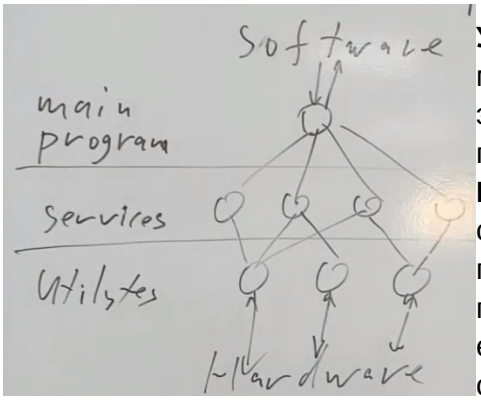
Принципы организации OS:

1) Принцип модульной организации
2) Принцип функциональной избыточности. OS обладает большим функционалом чем надо.
3) Принцип функциональной избирательности. OS позволяет нам выбирать функции, какие нам будут доступны и нет.
4) Принцип параметрической универсальности. OS как можно больше выносит своих параметров управления во внешнюю среду (конфиг файлы, реестр)
5) Концепция многоуровневой системы

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

Все процедуры OS сконцентрированы в ядре и тогда все резидентно и прив режим.

Можно выделить 3 слоя:

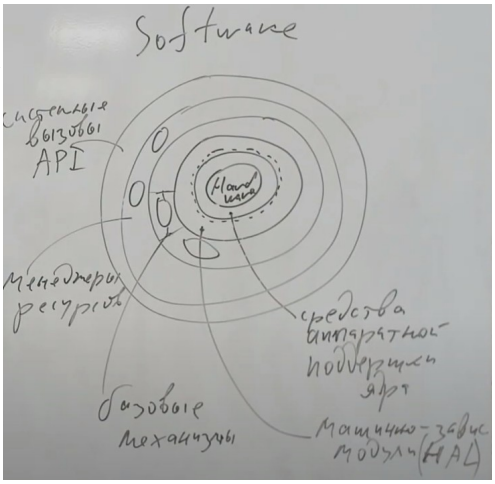


Утилиты — это драйвера, они абстрагированы, чтобы мы могли заменить их в зависимости от железа. **Сервисы** — это код, который не зависит от железа и связаны с принятием решений (планировщики, алокаторы памяти...). **Main program** — это точка входа, то что обрабатывает системный вызов. Приложение помещает в стек параметры системного вызова и инициирует его, потом процессор переходит в прив режим и вызывает единственный адрес (начало main program, единственный с точки зрения безопасности), потом main program читая параметры, дергает сервисы.

Большой минус мон арх в том, что при изменении, например аппаратного обеспечения, придется пересобирать ядро, а это долго. А также занимает RAM, поэтому думали о микроядерной архитерктуры.

Но функционал OS обширный, поэтому хотелось расширить сервисы и тогда появилась концепция **многослойной архитектуры**.

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



1) Средства аппаратной поддержки ядра — все реализовано аппаратно (прерывания, таймер, поддержка прив режима и вирт памяти, защита памяти…)

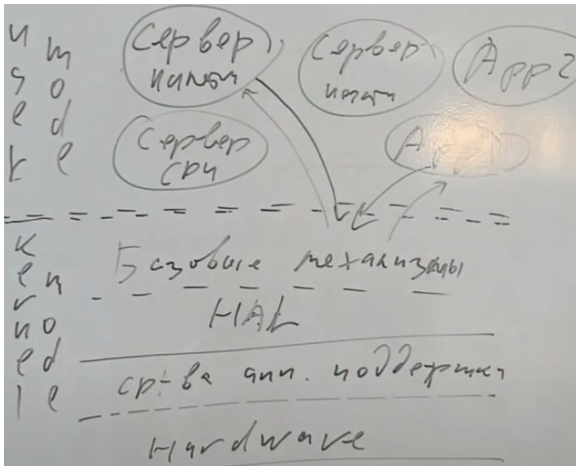
2) Машинно-зависимые модули (HAL) — соответствует утилитам из монол арх (см с 1 п.)

3) Базовые механизмы ядра — исполнение принятых решений

4) Менеджеры ресурсов — задачи принятия решений (см вместе с 3 п.), удобно для замены

5) Системные вызовы (API)

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



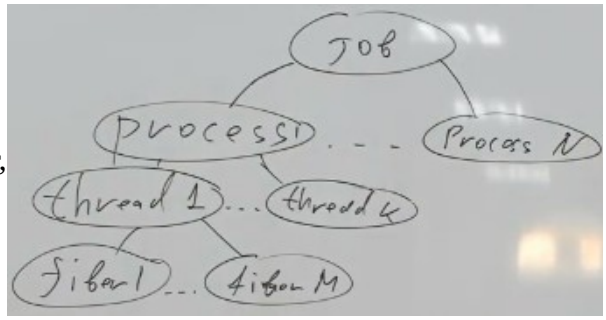
Преимущества: существенно меньше памяти нужно для OS в RAM, серверы не обязаны находиться в фиксированных областях памяти, удобно строить распределенные системы

Недостатки: больше переключений между kernel и user мод, ниже надежность системы (например пользовательское приложение получит больше приоритет, чем планировщик и оно будет работать вечно), безопасность ниже

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

Процесс — совокупность выполняющихся команд и ассоциирующими с ним контекстом и ресурсами, находящиеся под контролем OS. Вместе с процессом создается ее дескриптор.

Но мы можем распараллелить, например, обработку картинки, тогда логично поделить на потоки.



Поток — процесс включает в себя множество потоков. Каждый из потоков определенного процесса имеет отдельный набор команд и контекст, но все они имеют доступ к общему адресному пространству процесса. Находится под контролем OS.

Если мы написали какое-то многопоточное приложение, то возникает 2 проблемы: чтобы переключиться на другой поток, нужно перейти в режим ядра и обратно, а также может

получиться так, что OS будет давать предпочтения потокам не так, как мы рассчитывали и тогда многопоточное решение может быть даже хуже однопоточного.

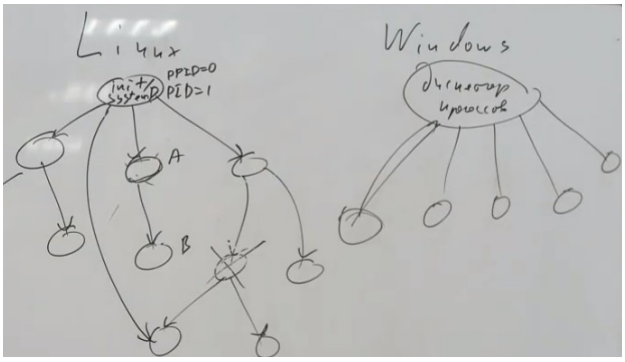
Тогда появилась идея сделать инструментарий для многопоточности на пользовательском уровне, тогда появился **fiber**, на котором мы можем написать код для переключения потоков.

Также несколько процессов объединяются в группы (**job**).

11. Функции подсистемы управления процессами. Краткая характеристика каждой функции.

1) **Создание** (создать процесс, один порождает другой)
2) **Обеспечение ресурсами**
3) **Изоляция** (изолировать процессы друг от друга)
4) **Планирование исполнение потоков** (в каком порядке предоставлять процессу доступ к ресурсам)
5) **Диспетчеризация** (исполнение плана, переключение процессов между различными состояниями)
6) **Организация межпроцессорного взаимодействия** (взаимодействие процессов между собой)
7) **Синхронизация** (разрешение конфликтов процессов за ресурсы)
8) **Уничтожение**

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



В Linux Процесс содержит PID, PPID, UID, статус, историю. Хранятся они в виде дерева. Дочерний процесс делается путем форка родителя и он имеет те же права и возможности, что и родитель. При уничтожении процесс сообщает результат родителю. Если закрывается родитель, то сначала он посылает сигнал детям и они сначала закрывается. Если родитель сдох неожиданно, то дети просто подвяжутся к корню дерева.

В Windows корнем является диспетчер процессов и чтобы создать потомка, нужно обратиться к нему, то есть он контролирует всё. Значит может быть, например, что потомок имеет больше прав, чем родитель.

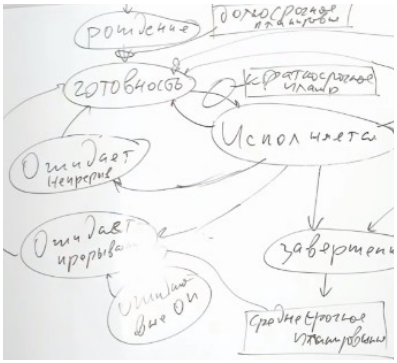
13. Модель жизненного цикла процесса: состояния процесса, правила переходов между состояниями.

Базой является *ожидает* и *исполняется*, но это плохо тем, что в *ожидает* мб процессы, которые прерваны таймером и те, которые ожидают, например I/O, то есть, попадя в *исполняется*, он может сразу же вернуться в *ожидает*, поэтому добавляется *готовность*.
Рождение тоже непросто, как и *завершение*, которое, например, может отправить сигнал завершения родителю, которые в *ожидает*.
Ожидает можем поделить на *ожидает непр* (ждет I/O), процесс в котором нельзя прерывать, например, на обработку сигнала, потому что эта обработка может, например, изменить важный контекст, и *ожидает пр*.
Zombie мб когда, например, родитель в *ожидает непр* и не может обработать сигнал

завершения. *Остановлен*, когда, например, процесс начинает жрать внезапно невообразимо много памяти.

14. Виды планирования и их место в жизненном цикле процесса.

1) **Краткосрочное** (простые алгоритмы)
2) **Среднесрочное** (например, мы знаем, что ближайшее время процесс в *ожидает непр* не получит доступ к I/O, тогда можем скинуть его в подкачку, и тогда мб родится даже новый процесс)
3) **Долгосрочное** (бывает что выгоднее не дать процессу родиться, пока что-то не выполнится)
4) Устройства I/O имеют свою очередь



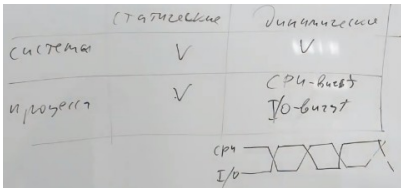
15. Критерии эффективности и свойства методов планирования процессов, параметры планирования процессов.

Критерии:
1) Критерий справедливости (гарантировать каждому процессу равную долю процессорного времени, но противоречит с 2 п.)
2) Критерий эффективности (максимально эффективно использовать ресурсы)

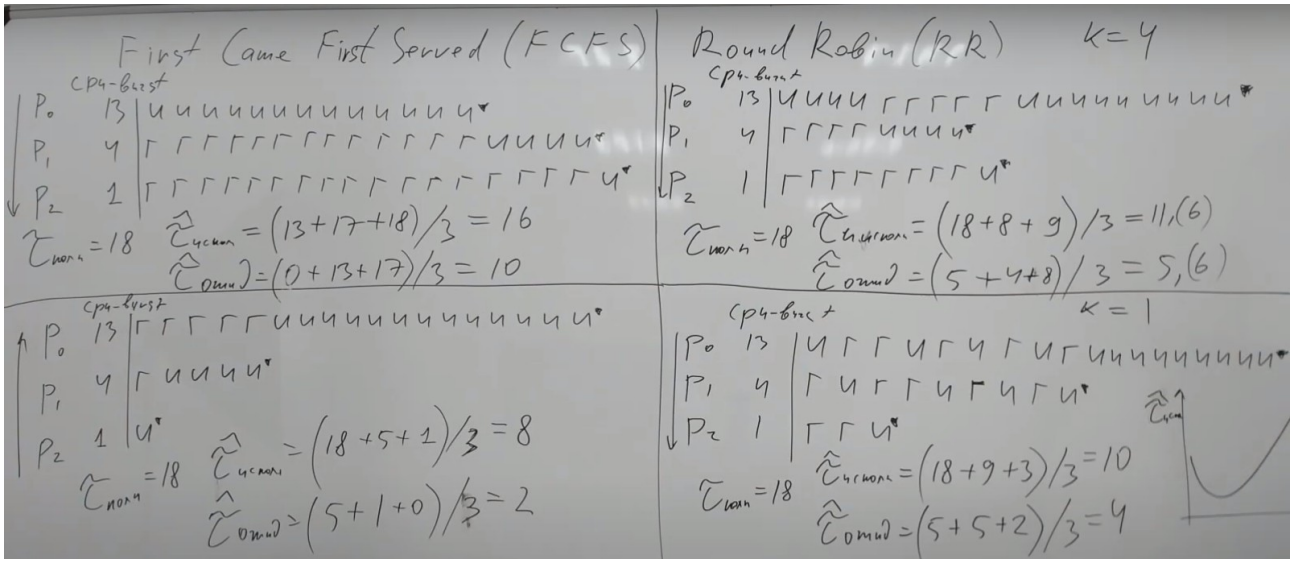
Это абстрактные, вот численные:
1) Сокращение полного времени выполнения
2) Сокращение времени ожидания (пример с размерами корзины S M L, память осв. быстрее)
3) Сокращение времени отклика (на самолетах важно)

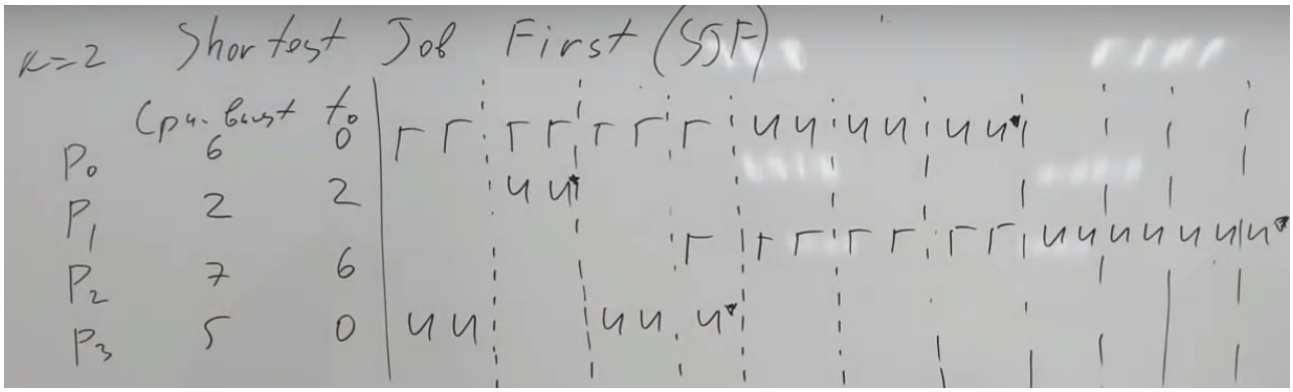
Свойства алгоритмов планирования:
1) Предсказуем (на одних и тех же данных получим один и тот же результат или близкий)
2) Имеют минимальные накладные расходы
3) Должен быть расширяемым

Параметры планирования:
1) СС (предельные значения ресурсов, память, ядер…)
2) ДС (текущее состояние OS, исп. памяти, нагрузка)
3) СП (права, UID, важность процесса)
4) ДП (сколько будет выполняться процесс, если без прерывания)

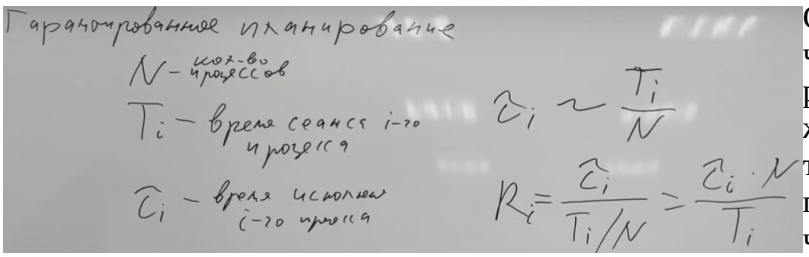


16. Методы планирования без внешнего управления приоритетами (FCFS, RR, SJF), гарантированное планирование. Описание каждого метода, их достоинства и недостатки.



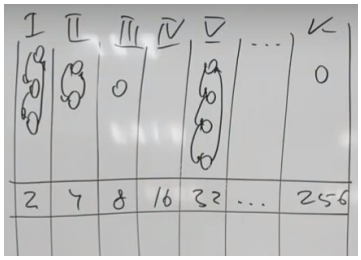


Но у всех этих алгоритмов все равно есть минус, а именно могут быть голодающие процессы, тогда попробуем сделать **гарантированный** алгоритм:



Сортируем очередь по R, заметим, что у всех процессов знаменатель растет одновременно на одно и то же число, а числитель растет только у исполняющихся процессов. **Проблемы**: дробные числа, не эффективный, взломать.

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



Пользователь будет сам ставить внешне приоритет, тогда пусть будет **k** очередей с приоритетом. **Проблема** в том, что мб голодающий процесс. Тогда давайте введем таймаут, за который если процесс ни разу не выполнился, то повышаем уровень приоритета данного процесса на 1, но как только он выполнился, то снова отбрасываем его в конец. Все еще остается **проблема** с эффективностью, если мы неправильно расставили изначально приоритеты, то процессы простаивают. Тогда введем ограничение на выполнения для каждой очереди (горизонтальная линия), если процесс успел выполниться за данное время, то он здесь и остается, если нет, то сбрасываем его в очередь приоритетом ниже на 1. Но все равно возможны голодающие процессы.

18. Организация планирования процессов в ОС семейства Microsoft Windows

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

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

Взаимодействие процессов — это когда мы пытаемся обеспечить возможность процессу обменяться управлением или данными (например pipe)

Например, первый процесс дал свою часть данных порту принтера, потом после таймера какой-то другой процесс начал свои данные посылать и в итоге печатается фигня. Проблема в том, что они не знают друг о друге.

Условия для обеспечения безопасного взаимодействия процессов:

1) **Взаимоисключение** (если один процесс имеет доступ к какому-то неразделяемому ресурсу, то никакой другой не может получить к нему доступ. Никакие два процесса не могут одновременно находится в крит секции относительно одного и того же ресурса)

2) **Отсутствие голодания**

3) **Прогресс** (есть свободный ресурс и готовый его испол процесс, но он его не использует)

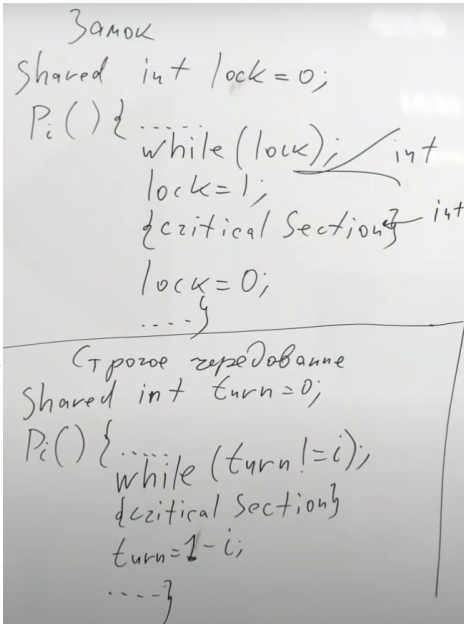
4) **Отсутствие тупиков** (например p1 взял ресурс R1 и p2 взял R2, потом p1 понадобился R2, не отпуская при этом R1 и также для p2, получился замкнутый цикл, такого быть не должно)

Критическая секция — это та часть кода, которая и взаимодействует с неразделяемым ресурсом.

21. Алгоритмы реализации взаимоисключений. Формальное описание алгоритмов, их недостатки.

1) **Аппаратная поддержка взаимоисключений путем перехода в одно программный режим:**

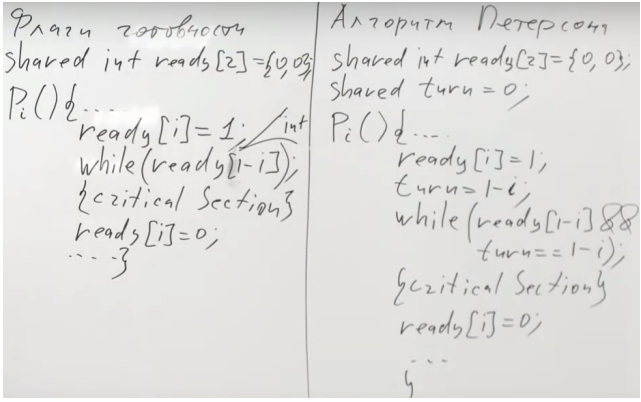
Перед входом в крит секцию процесс дает команду OS и она выключает обработку прерываний пока не выполнится секция. Проблема в том, что система становится неуправляемой, например, если процесс сдох во время секции, то так как нет прерываний его спасти никто не сможет. Но такой подход используется, например, при смене статуса у процессов, когда их меняет планировщик.



2) **Программное решение:**

2.1) **Замок** — просто лочим ресурс. Проблема: мы можем прерваться после while потом другой процесс залочит, снова прерывание и первый процесс продолжил — нарушение взаимоисключения.

2.2) **Строгое чередование** — p0 выполнился, отдал команду исполнения p1, а он спит, потом p0 снова хочет этот процесс, но прав у него нет из-за p1 — нарушение прогресса.

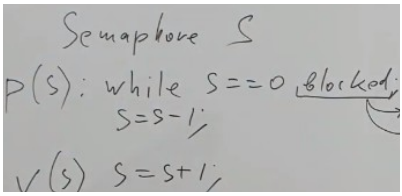


2.3) **Флаги готовности** — есть массив готовности. Проблема: если произойдет такое прерывание перед while, то гарантированно будет тупик, все встанут.

2.4) **Алгоритм Петерсона** — например 2 человека хотят пройти в дверь, 1-ый предлагает 2-ому, а тот опять 1-ому, тогда 1-ый точно проходит. Проблема: если много процессов, то предложение пройти будет переходить по всем желающим. Также при возникновении нового процесса, снова надо проходить по всем процессам.

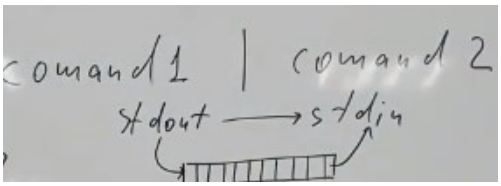
В итоге именно эффективного программного решения не нашли и вернулись к аппаратному решению, но только запретили прерывания у лока в том месте, где образовалась проблема и добавили такую команду на уровне процессора. (Mutex)

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



Семафор — это целая неотрицательная переменная, над которой разрешены только 2 атомарные операции: p(s) и v(s). Дейкстра занимался ими, потому что хотел как-то избежать постоянной проверки, например, можно ли выделить в текущий момент еще поток. Потом это стало применяться для всех примеров, использующих объект конечной емкости.

Проблемы «производитель-потребитель»:

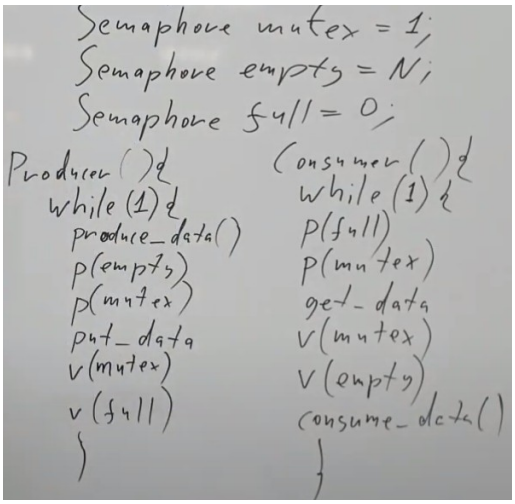


1) Одновременное чтение и запись

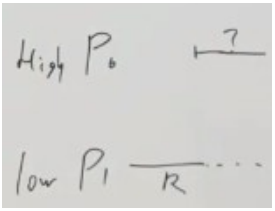
2) Переполнение буфера

3) Попытка чтения из пустого буфера

Решение:



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



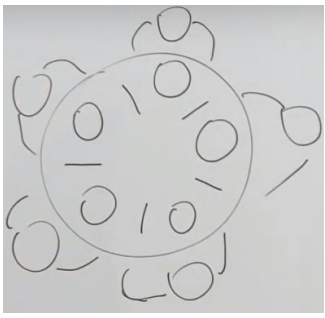
Может быть такое, что p1 с низким приоритетом захватил ресурс, а высокий p0 требуют его, но не сможет получить, так как p1 не будет возобновлен.

Решение:

1) **Спинлок** — когда есть общая переменная и она проверяется в while, получается активно используется процессорное время. Если крит секция небольшая, то норм его использовать.

2) **Mutex** — это бинарный семафор, только при этом он ведет запись того кто держит этот mutex и при блоке также повышается приоритет того процесса у кого этот mutex до текущего.

Проблема обедающих философов:



Есть 5 философов за столом. Три состояния у каждого мб: думает, голодает и ест. Есть можно только двумя вилками (одна слева и справа) макароны.

Если каждый по кругу возьмет левую вилку, то тупик. По другому, возьми левую, если не удалось взять правую, то положи левую обратно и попробуй снова, тоже тупик, только постоянно что-то делаешь. Можно попробовать ждать рандомное время между взятием вилок и это в целом решение, но все равно вероятность одновременного взятия всеми вилок остается. Тогда остается сделать наблюдателя, который должен разрешать и запрещать им есть.

24. Тупики. Условия возникновения и методы борьбы с тупиками.

Условия возникновения тупиков (должны быть одновременно):

1) **Mutual Exclusion** (должно быть условие взаимоисключения, неразделяемости ресурсов)

2) **Hold and Wait** (процесс может захватить ресурс и не отпуская его может требовать другой)

3) **No Preemtion** (у нас нет возможности забрать ресурс у процесса, которому мы выделили)

4) **Circular Wait** (процессы относительно друг друга встали в кольцевое ожидание)

Методы решения:

1) **Игнорировать тупики** (довольно малая вероятность их возникновения)

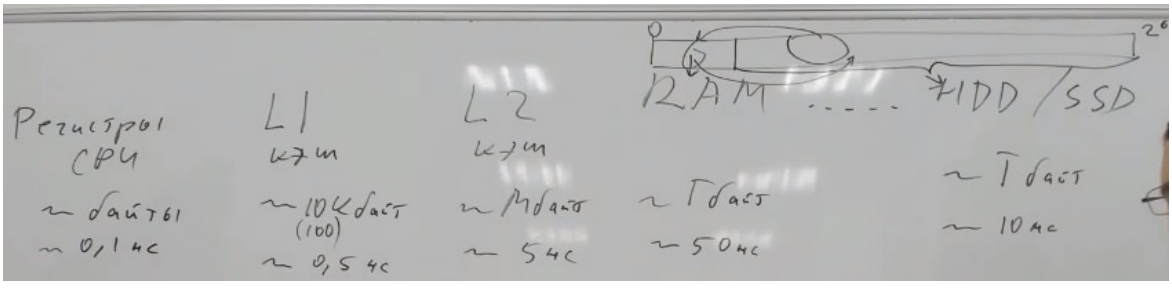
2) **Пытаться тупики предотвращать** (пытаться не допускать условия их возникновения)

3) **Обнаружение тупиков** (очень дорого)

4) **Возобновление работы после тупика**

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

По Фон Неймону код и данные должны быть однородные, но есть данные или код, которые используются редко, а память ограничена и бесконечно расширятся без потери, например, скорости доступа не не может, тогда стоит подумать об иерархии памяти.



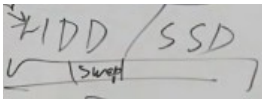
То, что мы редко используем мы храним на HDD, но не в виде файлов, а в виде продолжения RAM, то есть есть общее адресное пространство, пронумерованное 0..2^64 например. Тогда небольшой кусок в нем будет реальная память, а остальное HDD и будем перебрасывать данные между этими двумя кусками - **свопинг** (см картинку). Это все называется **виртуализацией**.

Свопинг можно делать путем перебрасывания всего адресного пространства процесса, тогда сохранится целостность данных и сохранить адресацию данного процесса (изменятся на константу). Или можно частями перекидывать, то будет сложнее пересчет адресов, целостность данных и безопасность хуже, можно, например, взломать данные на диске и потом они подгрузятся в RAM. Но зато не надо гонять туда сюда неиспользуемые данные.

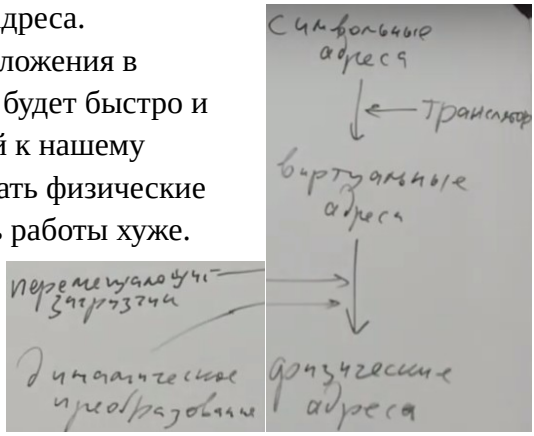
Обеспечить доступ к данным на HDD можно с помощью:

1) **Файл подкачки** (Windows). Раз это файл, то свопинг происходит через стандартные механизмы файловой системы. Плюсы: не надо писать спец код, легко меняется размер файла. Минусы: работаем через абстракцию файлов, но не используем его (накладные расходы), тогда падает производительность, при этом падает надежность если сдохнет файловая система.

2) **Раздел подкачки** (Linux) — мы на диске выделяем раздел и делаем специфичную файловую систему. Плюсы: надежность и безопасность и производительность, так как свой раздел и ф.с. Минусы: переразметить место для раздела сложно



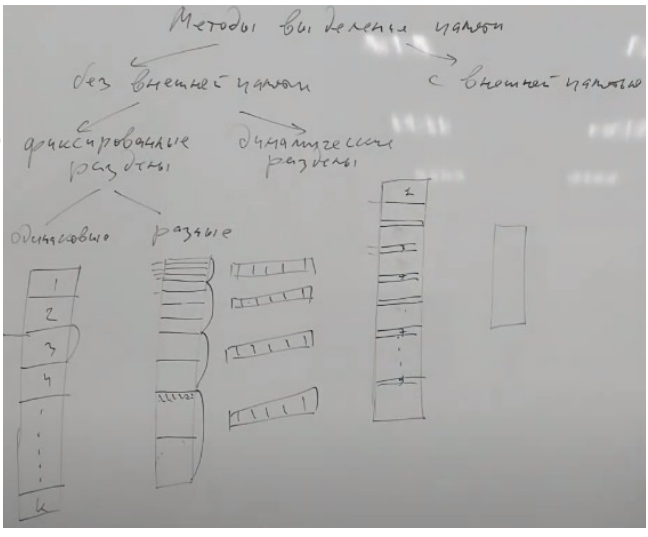
Преобразование адресов:
Символьные адреса — названия переменных, массивов… в яп высокого уровня.
Виртуальные адреса — транслятор переменных в виртуальные адреса.
Физические адреса — мы можем сразу перевести все адреса приложения в физические, тогда загрузка приложения будет долгая, но работать будет быстро и при этом безопасность есть, так как мы знаем границы обращений к нашему приложению в памяти. Или же мы можем динамически высчитывать физические адреса, то есть изначально останутся виртуальные, тогда скорость работы хуже.



26. Методы распределения оперативной памяти без использования внешней памяти.

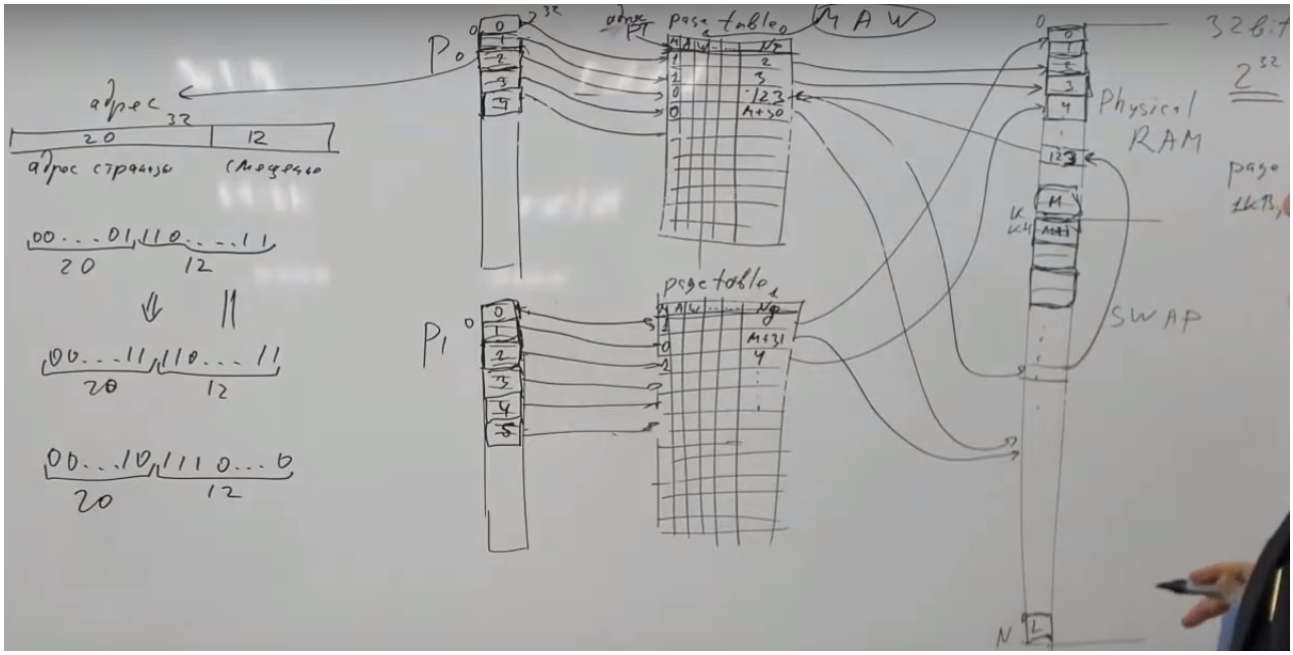
С внешней памятью — это когда еще где-то располагаются процессы, кроме RAM. (свопинг)

Одинакового размера разделы — смещения быстрые от виртуальных к физическим адресам, но непонятно какого размера эти разделы и неэффективно используется память в разделах.
Разного размера разделы — выделим сколько-то разного размера разделов, тогда смещения уже не такие быстрые, так как мы уже полноценно складываем адреса,



а не просто меняем в них бит, но все еще неэффективное использование памяти. Можно добавить очередь на какого-то размера раздел, но тогда возможно голодание.
Динамические разделы — появился процесс и сразу выделили сколько ему надо и т.д. Тогда память будет использоваться эффективно, но со временем возникает много «дырок» и тогда придется перефрагментировать, при этом будут же блокироваться переносимые процессы

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



Разбиваем пространство памяти на страницы размером обычно 4Кб, виртуальные адреса у процессов тоже из страниц. Для каждого процесса есть своя Page Table, в которой есть сколько-то специальных битов (обычно это M, A, W) и физический адрес. Страницы процесса линейно, то есть друг за другом не пересекаясь отображаются в эту таблицу построчно. Адрес страницы состоит из 2 частей: адрес страницы и смещение в ней.

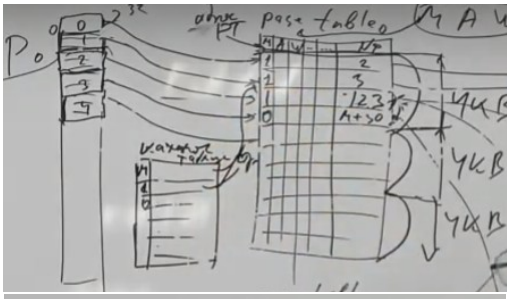
M — указывает, находится ли текущая страница в RAM или нет. Если есть (1), то продолжаем работу. Иначе происходит **страничное прерывание**: делается прерывание и процесс переходит в ожидание, потом происходит страничный обмен между RAM и Swap и продолжаем работу процесса. Однако, страничное прерывания происходит не быстро

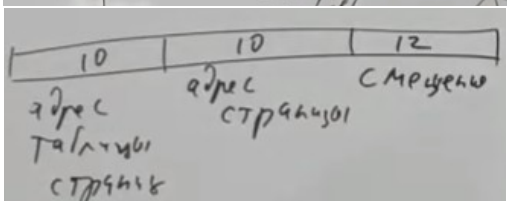
A — если к странице, перемещенной из Swap в RAM никто не обращался, то 0. Это нужно, чтобы при страничном обмене делать более выгодный обмен (типа если к странице никто не обращался, мб и дальше не будут). В Linux есть активные и неактивные списки. Типа если было обращение, то данная страница переезжает в активный список и если в течение некоторого времени не было обращений, то она сбрасывается обратно в неактивный.

W — если с последнего страничного обмена произошли изменения в данной странице, то 1. Если при обмене из RAM в Swap бит равен 0, то есть не было изменений, то можно только изменить физический адрес в таблице, не перенося сами страницы из RAM.

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

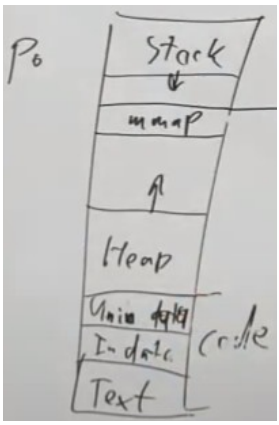
Размер Page Table: 1 + 3 = 4 байта на одну страницу, а всего страниц в таблице мб 2^32, тогда размер 4 * 2^32 = 4 мб. Это очень дорого, поэтому давайте заметим, что мы зачастую работает в определенной локации таблицы.





Тогда давайте поделим таблицу на части размером по 4Кб, таких частей получится 2^10, и заведем для каждого процесса **каталог таблиц страниц** (в ней также есть бит M только указывающая на часть таблицы), где будут храниться адреса этих частей. При обращении к странице мы будем по первым 10 битам смотреть нужную нам часть таблицы страниц и уже по этой части будем брать как и раньше физический адрес. Таким образом, нам достаточно хранить для каждого процесса 4 + 4 = 8Кб вместо 4 Мб, но мы теряем немного в производительности, возможны 2 прерывания. Дополнительно мы будем кешировать адреса страниц в кеше **TLB**.

Сегментно-страничная организация:

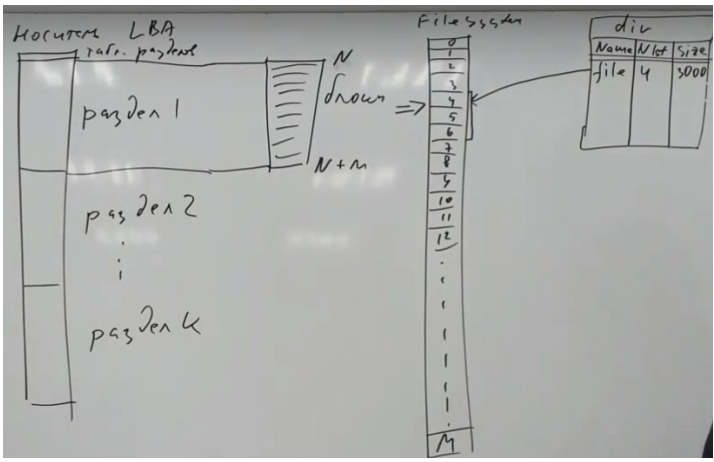


Мы делим адресное пространство процесса на сегменты: стек, куча, код (ин и неин данные и инструкции), mmap. Нам же нужно хранить страницы для ин и неин данных, с которыми мы работаем. Стек постоянно растет и очень быстро может, поэтому для него есть ограничение. Mmap — это сегмент связанный с открытыми файлами. Сегментация делается, потому что нам требуется хранить активную страницу для стека, кучи, кода. То есть для каждого сегмента мы держим активную страницу, так как мы с ними работаем.

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

Мы делим пространство нашего носителя на блоки обычно размером 1Кб.

Непрерывная последовательность блоков:



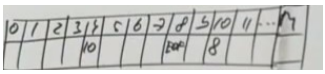
У нас будет каталог dir (таблица), в которой мы храним название файла, номер блока начала его данных и размер. Тогда По размеру мы получаем кол-во блоков, округляя там вверх и по остатку знаем сколько в последнем блоке надо прочитать. Плюсы: простая реализация. Минусы: будет перефрагментация, например при редактировании и увеличении размера файла придется искать другую более большую последовательность блоков.

Связные списки:

Теперь мы не будем выделять последовательные блоки для файла, а будем в последние 4 байта каждого блока хранить адрес на следующий блок данного файла. Плюсы: решена

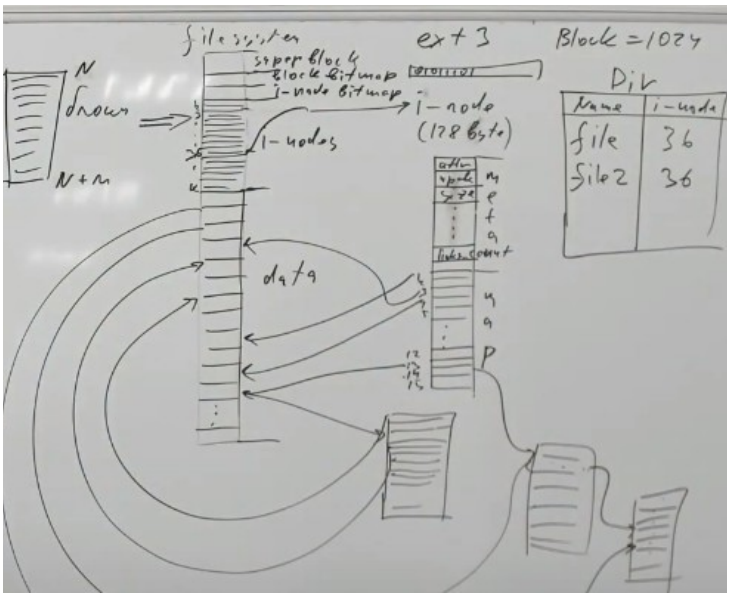
проблема посл блоков с перефрагментацией. Минусы: потеряется один блок, потеряется весь файл, производительность плохая (нельзя сразу начать с 40 минуты фильма), обе части блока не кратны степени двойки!!!

Таблица размещения файлов (FAT):



Дополнительно к каталогу будем хранить вектор из M (число блоков) записей. В каталоге берем первый блок и потом прыгаем по блокам из вектора. FAT-32 означает, что адрес блока состоит из 32 бит. Минусы: надежность и безопасность если потерять этот вектор, а также дорого хранить вектор в RAM (надо так как надо быстро прыгать по вектору). (Если у нас объем носителя > 2^32 * 1Кб, то надо делать блоки большего размера, но тогда увеличивается объем возможного неиспользуемого места в блоках. Тогда можно попробовать поделить блоки на группы блоков, но и там не все так радужно)

30. Методы организации хранения данных в файловых системах: индексные дескрипторы.



Поделим раздел на суперблок, block bitmap, i-node bitmap, i-nodes (блоки 128 байт) и обычные блоки. В ноде хранятся мета данные и карта размещения.

Первые 12 адресов в карте размещения хранят реальные адреса на блоки данных, 13-ый адрес хранит адрес блока, в уже котором хранятся адреса на блоки данных (**косвенная адресация**), 14 и 15-й просто увеличивают вложенность адресации как у 13-ого. Но такая архитектура плохо сказывается на производительности, поэтому **ext2** и **ext3** системы удобны для работы с маленькими файлами.

Но как узнать свободен ли блок или i-node? Мы храним битовые карты block bitmap и i-node bitmap, в которых мы храним бит, указывающий свободен ли блок/нода.

Это все было в ext2, а в ext3 добавилась журналируемость.

31. Журналируемые файловые системы. Назначение и виды журналов.

Журнал — это план того, чего я хочу сделать. Сначала я выполняю операцию, потом стираю строчку из журнала. Если произошел сбой, то либо в журнале осталось что-то недоделанное либо оно уже внесено на диск, тогда, проанализировав состояние диска и журнала можно внести нужные изменения. Но журнал увеличивает расходы, есть 3 вида журнала:

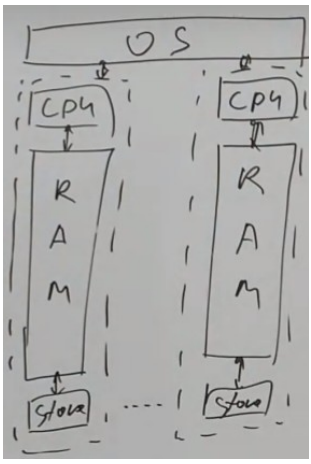
1) **Writeback** — регистрирует только метаданные в журнале. Однако, этот режим записывает данные в файловую систему независимо от метаданных журнала. Другими словами, система может записывать данные в файловую систему либо до, либо после того, как она внесет метаданные в журнал.

2) **Ordered** — регистрируются только метаданные в журнале и мы выполняем действия записи в следующем порядке:
а) записывает данные в блоки назначения в файловой системе

b) записывает метаданные в журнал
c) позже, когда система примет решение, она передаст метаданные в файловую систему

3) **Journal** — самый защищенный, сначала полностью прописываем метаданные и блоки данных тоже в журнал, а потом выполняем эти действия, то есть по факту делаем двойную работу.

32. Обоснование необходимости и принципы построения распределенных ОС.



Например, при увеличении запросов на сервер становится не хватать чего нибудь: памяти, процессора… А бесконечно увеличивать объемы памяти или скорость процессора в силу технического развития невозможно. Тогда мы можем вспомнить о сетевой операционной системе, которая образует канал данных, по которому передаются данные между OS. Это позволяет распределить нагрузку, но у нас появляются накладные расходы (упаковать передать распаковать), а также проблема, если 2 процесса используют одни и те же данные, короче плохо.

Тогда сделаем общую OS над другими. Таким образом мы получаем распределение сил и надежность, а также можно распределить систему территориально по Земле, что будет хорошо для пользователей.

Принципы построения:

1) Ни один узел не имеет полной информации о состоянии всей OS (прям всю нельзя, обобщенные хар-ки можно)

2) Узлы способны принимать решения на основе только локальных данных

3) Узлы принимают решения таким образом, что отказ какого-то узла не должен приводить к невозможности принять решение (например не должно быть централизованных узлов или данных)

4) Не должно быть явного или не явного предположения о существования глобальных часов (создание глобальных часов может привести к точки централизации)

Принципы прозрачности:

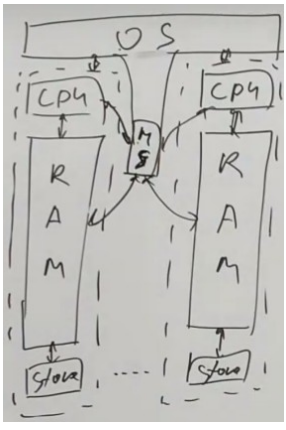
1) **Расположения** (приложение не должно знать где конкретно лежат его ресурсы, данные)

2) **Миграции** (приложение не должно знать о перемещении его ресурсов)

3) **Размножения** (приложение не знает сколько есть копий ресурса, с которым оно работает)

4) **Конкуренции** (приложение не должно знать что оно конкурирует с другими за ресурсы)

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



Алгоритмы:

1) Пусть будет **центральный сервер**, в котором будет единое адресное пространство на все участки памяти системы и через который процессоры и будут брать нужные им данные. Здесь ломается принцип нецентрализованности, но это можно сделать, если мы сумеем сделать очень хорошую производительность, чего невозможно сделать, а также появляются накладные расходы

2) Если в текущей RAM не оказалось текущей страницы и мы каким-то образом знаем в какой OS лежит эта страница, то давайте сделаем типа свопа. Плюсы: данные существуют в едином экземпляре и когда кому-то они понадобятся, то эти узлы встанут в очередь за ними. Минусы: будет подкачка (процесс уйдет в своп), возможно борьба нескольких процессов за страницу

3) Метод размножения для чтения. Если понадобится страница, то также ищем ее и во время свопа мы именно копируем у себе страницу. Мы можем сделать дополнительные механизмы, которые будут удалять ненужные копии со временем. Плюсы: производительность. Минусы: храним дубликаты данных

Но как найти эти страницы?

1) Вариант с центральной таблицей страниц. Минусы: централизация

2) Будем помнить куда мы отдаем страницу, тогда мы будем искать среди соседей тех, кто знает где эта страница мб (типа связный список). Минусы: O(n)

3) Будем делать массовую рассылку всем узлам сразу. Минусы: таким образом мы нагружаем все узлы

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

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