Су Меню

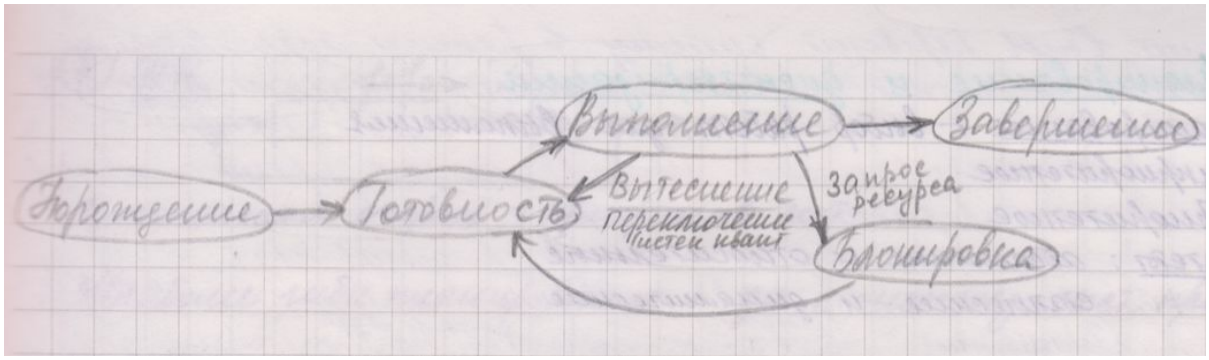
1. Понятие процесса - процесс как единица декомпозиции системы. Диаграмма состояний процесса в Unix. Потоки - определение, типы потоков.Планирование и диспетчиризация. Процессы Unix.
2. Подсистема ввода-вывода: задачи подсистемы ввода-вывода, место в ОС. Система прерываний: типы прерываний и их особенности, прерывания в послед ввода-вывода - обсл запроса процесса на ввод-вывод. Контроллер прерывания - задачи, структура.
3. Управление вводом-выводом: опрос (polling): вводы-вывод с исп прерываний, прямой доступ к памяти (direct memory access). Контроллер ПДП - конфигурирование ПДП, режимы ПДП (пакетный, кража циклов, прозрачный) - особенности.
4. ОС - определение, место ОС в системе ПО ЭВМ. Ресурсы пользовательской системы.Режимы ядра и задачи - переключение в режим ядра - классификация событий. Процесс как единица декомпозиции системы. Диаграмма состояний процесса.
5. Класификация операционных систем и их особенности. Виртуальная и иерарх системы - декомпозиции системы на уровне иерархии, ядро системы - определение.
6. Процессы: бесконечное откладывание, зависание, тупиковая ситуация - анализ на примере задачи об обедающих философах. Считающие и множественные семафоры. Мониторы: монитор кольцевой буфер.
7. Виртуальная подсистема proc - назначение, особенности, файлы, поддиректории и ссылка self. Структура proc_dir_entry: функции для работы с элементами /proc. API - интерфейсы для доступа к адресному пространству пользователя.
8. Классификация ядер ОС. Особенности ОС с микроядром: реализация взаимодействия процессов по модели клиент-сервер. Три состояния процесса при передаче сообщений. Достоинства и недостатки микроядерной архитектуры. Многопоточное ядро; взаимоисключение в ядре – спин - блокировки.
9. Управление устройствами: физические принципы управления устройствами. буферизация ввода-вывода - управление буферами. Буферный пул, кеширование.
10. Взаимодействие процессов. Семафоры. И ещё что то про семафоры.
11. Подсистема ввода-вывода: программные принципы управления устройством. Специальный файл - идентификация устройства в ОС Юникс, точки входа в драйверы.
12. Виртуальная память: распределение страницами по запросам, свойство локальности, анализ страничного поведения процессов, рабочее множество. Стратегии выделения памяти; фрагментация.
13. Физические принципы устройств, структура контроллера, I/O mapping, memory mapping, гибридная схема, общая шина, выделенная шина, как с ними работать.
14. Unix: команды - fork(); wait(); exec(); pipe(); signal().
15. Терминал.управляющий терминал.терминальный ввод-вывод. Псевдотерминал. назначение псевдотерминала.как взаимодействуют процессы в пседотерминале кажется. доп вопросы:как представлен псведотерминал в системе,что эмулирует ведущее устройство, зачем нужны псевдотерминалы
16. Орг монопольного доступа с помощью команды test-and-set, алгоритм Деккера.

17. Сокеты - определение, виды сокетов. DGRAM сокеты.
Примеры(программирование). Сетевые сокеты, сетевой стек, порядок байтов.
18. Unix: разделяемая память (shmget(), shmat()) и семафоры (struct sem, semget(), semop()). Пр. использования
19. Аппаратные прерывания в Linux: нижние и верхние половины и особенности работы softirq и tasklet в SMP-системах
20. Взаимодействие процессов: монопольное использование - программная реализация взаимоисключения, взаимоисключение с помощью семафоров; сравнение - достоинства и недостатки. Мониторы - определение, пример: монитор "кольцевой буфер".
21. Пять моделей ввода-вывода в Linux : диаграммы, особенности.
22. Тупики - определение, условия возникновения тупиков, методы обхода тупиков - алгоритм банкира, семафоры(вроде читающие и ещё какие-то…)
23. ФС Unix, vfs/vnode, inode, адресация больших файлов.
24. Три режима работы компьютера на базе процессоров Intel. Прерывания в защищенном режиме (таблица ID). преобразование адреса при страничном преобразовании в процессорах Intel.
25. Примеры спец. файловых систем. Файловая система Linux -Интерфейс vfs/vnode. (в интерфейсе отвечал в общем про файловую систему и vnode, главное надо было показать struct superblock и struct dentry; не надо писать про ФС в Windows. А также в ответе должны обязательно присутствовать описания структур dentry и superblock.)
26. Файловые системы. Иерархическая структура файловой системы в Unix, задачи уровней. Открытые файлы - структуры описывающие файл. Особенности использования open() и fopen().
27. процессы unix: демоны, примеры системных демонов, правила создания демонов
28. Сокеты - определение, типы, семейства, сетевой стэк(порядок вызова функций - схема), преобразование байтов (little-big endian), мультиплексирование
29. Псевдотерминалы - использование процессами псевдотерминалов, псевдотерминалы в Linux, программирование псевдотерминалов - функция pty_fork().
30. Модель ввода-вывода с мультиплексированием. Сетевые сокеты с мультиплексированием, установление соединения, сетевой стэк
31. Файловая система proc. Функции для передачи данных (между процессом и ядром). Модули ядра, суть, пример.
32. Защищенный режим: системные таблицы – GDT, IDT, теневые регистры. перевод компьютера из реального режима в защищенный и обратно. XMS, линия A20 – адресное заворачивание. Conventional, HMA, UMA, EMA.
33. Прерывание от системного таймера в реальном и защищенном режиме: функции. (Объединены два вопроса)
34. Задача читатели-писатели, решение с использованием семафоров Дейкстра для ОС Unix, синхронизация потоков в Windows, Consumer

1. Понятие процесса - процесс как единица декомпозиции системы. Диаграмма состояний процесса в Unix. Потоки - определение, типы потоков.Планирование и диспетчиризация. Процессы Unix. (8-18)

Процесс - программа в стадии выполнения **Процесс** - единица декомпозиции системы, потому что под него выделяются ресурсы. Процессы конкурирует в системе за ее ресурсы **Ресурс** - любой из компонентов системы, предоставляющий ее возможности **?)**

В многопроцессорных ОС одновременно существует большое количество процессов которые конкурируют **Состояние процесса** - абстракция, которой оперируют разработчики.

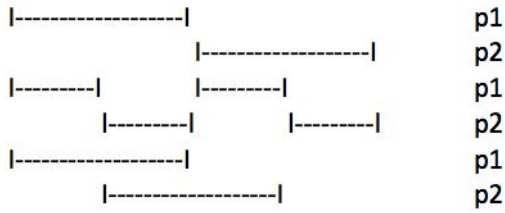


Главная системная таблица - таблица процессов - массив структур

1) Что бы начать выполнение программы, процесс должен быть идентифицирован В unix процесс описывается структурой proc Каждому запущенному процессу выделяется структура (дескриптор) или строка в системной таблице (номер строки - идентификатор процесса)
2) Необходимо определить первый ресурс : объем ОП в соотв с концепцией хранимой программы
3) В состоянии готовности находятся многие процессы - из них получаем очередь готовых процессов
4) Переходим к выполнению после выделения процессорного времени
5) Завершение или запрос доп ресурсов (блокировка) - заканчивается выделением ресурса
6) Из блокировки - в очередь готовых процессов. Процессорное время получает процесс с наивысшим приоритетом
7) Если поступил процесс с большим приоритетом чем тот, который выполняется -> вытеснение (в пакетной обработке) + (система реального времени)

В системе разделенного времени - если истек квант процессорного времени, то процесс возвращается в очередь -> переключение по вытеснению и переключение в блокировку, приводят к переключению контекста процесса. Необходимо в системе сохранить инф, которая позволит продолжить вып процессов - содержимое регистров (аппаратный контекст) + информацию о выделенных ресурсах

Уровни наблюдения (Последовательное, квазипараллельное, параллельное (на рисунке))

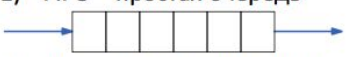


Планирование - выбор работы для выполнения

1) Приоритеты: абсолютные и относительные, статические и динамические
   a) Безприоритетные b) Приоритетные
2) С переключением или без, с вытеснением или без (4 вида)

Алгоритмы планирования

1) FIFO – простая очередь

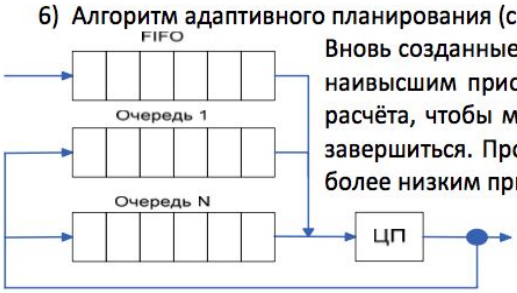


Без переключения, без приоритетов.

2) SJF – Shortest Job First – наикратчайшее задание – первое. М.б. бесконечное откладывание. Д.б. априорная инф. о времени выполнения программы, объеме памяти. Статич. приор., без переключ.
3) SRT – Shortest Remaining Time – с вытеснением. Выполняющийся процесс прерывается, если в очереди появляется процесс с меньшим временем выполнения (меньше оставшегося до заверш. времени).
4) HRR – Highest Response Ratio (наиболее высокое относительное время ответа) – с динамич. приоритетами. Используется в UNIX. $priority = \frac{t_W - t_S}{t_S}$, t_W – время ожидания, t_S - запрошенное время обслуживания. Чем больше ожидает, тем больше приоритет. Позволяет избежать бесконечного откладывания.
5) RR – RoundRobin-алгоритм (циклическое планирование) – с переключ., без вытеснения, без приоритетов

Разл. процессы м. иметь разл. величину кванта.

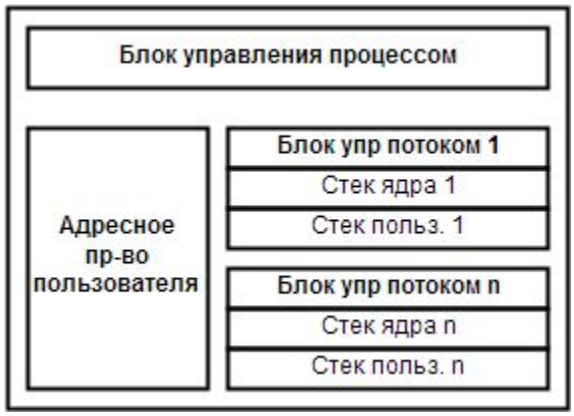
6) Алгоритм адаптивного планирования (с многоуровневыми очередями).



Вновь созданные процессы и процессы после завершения IO попадают в очередь с наивысшим приоритетом (FIFO). Квант времени для этой очереди выбирается из расчёта, чтобы максимальное количество процессов успело выдать запрос IO или завершиться. Процессы, не сделавшие ни того, ни другого, переходят в очередь с более низким приоритетом по алгоритму RR. В очереди N «крутится» idle-процесс (холостой) и там же оказываются все вычислит. процессы.

Адаптивно-рефлективное планирование - дополнительно оценивается текущий объем памяти, занимаемой процессом, а также может ли процесс получить необходимую физическую память для дальнейшего выполнения

Поток - некоторая непрерывная часть кода, которая может выполнятся с другими частями кода программы. Потоки возникли для уменьшения затрат при переключении контекста.Поток не имеет собственного адресного пространства, т. е. выполняется в адресном пространстве процесса. Ресурсы закрепленны за процессом не смотря на то, что они были выделены под потоки. Потоки разделяют код и сегмент данных. Поток владеет аппаратным контекстом, счетчиком команд.



3 типа нитей (thread'ов) 1)ядра

2) пользователя 3)легковесные процессы

2. Подсистема ввода-вывода: задачи подсистемы ввода-вывода, место в ОС. Система прерываний: типы прерываний и их особенности, прерывания в последовательности ввода-вывода - обслуживание запроса процесса на ввод-вывод. Контроллер прерывания - задачи, структура.

Обмен данными между пользователями, приложениями и периферийными устройствами компьютера выполняет специальная подсистема ОС – подсистема ввода-вывода. В работе подсистемы ввода-вывода активно участвует диспетчер прерываний. На подсистему ввода-вывода возлагаются следующие функции:

- организация параллельной работы устройств ввода-вывода и процессора;
- согласование скоростей обмена и кэширование данных;
- разделение устройств и данных между процессами (выполняющимися программами);
- обеспечение удобного логического интерфейса между устройствами и остальной частью системы;
- поддержка широкого спектра драйверов с возможностью простого включения в систему нового драйвера;
- динамическая загрузка и выгрузка драйверов без дополнительных действий с операционной системой;
- поддержка нескольких различных файловых систем;
- поддержка синхронных и асинхронных операций ввода-вывода.

Для ПК операции ввода-вывода могут выполняться тремя способами.

1. С помощью программируемого ввода-вывода. В этом случае, когда процессору встречается команда, связанная с вводом-выводом, он выполняет ее, посылая соответствующие команды контроллеру ввода-вывода. Это устройство выполняет требуемое действие, а затем устанавливает соответствующие биты в регистрах состояния ввода-вывода и не посылает никаких сигналов, в том числе сигналов прерываний. Процессор периодически проверяет состояние модуля ввода-вывода с целью проверки завершения операции ввода-вывода.

2. Ввод-вывод, управляемый прерываниями. Процессор посылает необходимые команды контроллеру ввода-вывода и продолжает выполнять текущий процесс, если нет необходимости в ожидании выполнения операции ввода-вывода. В противном случае текущий процесс приостанавливается до получения сигнала прерывания о завершении ввода-вывода, а процессор переключается на выполнение другого процесса. Наличие прерываний процессор проверяет в конце каждого цикла выполняемых команд.

3. Прямой доступ к памяти (direct memory access – DMA). В этом случае специальный модуль прямого доступа к памяти управляет обменом данных между основной памятью и контроллером ввода-вывода. Процессор посылает запрос на передачу блока данных модулю прямого доступа к памяти, а прерывание происходит только после передачи всего блока данных.

Прерывания (типы)

- программные (системные вызовы) – вызываются искусственно с помощью соответствующей команды из программы(int), предназначены для выполнения некоторых действий ОС (фактически запрос на услуги ОС), являетсясинхронным событием.
- аппаратные – возникают как реакция микропроцессора на физический сигнал от некоторого устройства (клавиатура,системные часы, мышь, жесткий диск и т.д.), по времени возникновения эти прерывания асинхронны, т.е. происходят в случайные моменты времени.
  Различают прерывания: От таймера От действия оператора От устройств вв/выв
- исключения – являются реакцией микропроцессора на нестандартную ситуацию, возникшую внутри микропроцессора во время выполнения некоторой команды программы (деление на ноль, прерывание по флагу TF (трассировка)), являются синхронным событием.
    - Исправимые – приводят к вызову определенного менеджера системы, в результате работы которого может быть продолжена работа процесса (пр.: страничная неудача с менеджером памяти)
    - Неисправимые – в случае сбоя или в случае ошибки программы (пр.: ошибка адресации). В этом случае процесс завершается.

Механизм реализации аппаратных прерываний Когда устройство заканчивает свою работу, оно инициирует прерывание (если они разрешены ОС). Для этого устройство посылает сигнал на выделенную этому устройству специальную линию шины. Этот сигнал распознается контроллером прерываний. При отсутствии других необработанных запросов прерывания контроллер обрабатывает его сразу. Если при обработке прерывания поступает запрос от устройства с более низким приоритетом, то новый запрос игнорируется, а устройство будет удерживать сигнал прерывания на шине, пока он не обработается. Контроллер прерываний посылает пошине вектор прерывания, который формируется как сумма базового вектора и No линии IRQ (в реальном режиме базовый вектор = 8h, в защищенном – первые 32 строки IDT отведены под исключения => базовый вектор = 20h). С помощью вектора прерывания дает нам смещении в IDT, из которой мы получаем точку входа в обработчик. Вскоре после начала своей работы процедура обработки прерываний подтверждает получение прерывания, записывая определенное значение в порт контроллера прерываний. Это подтвержд. разреш. контроллеру издавать новые прерывания.

3. Управление вводом-выводом: опрос (polling): вводы-вывод с использованием прерываний, прямой доступ к памяти (direct memory access). Контроллер ПДП - конфигурирование ПДП, режимы ПДП (пакетный, кража циклов, прозрачный) - особенности.

Задача системы вв/выв – обеспечить взаимодействие процессора с внешними устройствами. **Синхронный** – приложение, запросившее операцию вв/выв. блокируется до завершения этой операции. **Асинхронный** – приложение, подавшее запрос на вв/выв, может какое-то время продолжать выполнение

1) ***Программируемый***. Когда процу встречается команда вв/выв, выполняя эту команду, процессор вызывает библиотечную функцию, кот. переводит процессор в режим ядра, осуществляет передачу команды вв/выв. Контроллер выполняет необходимые действия, затем устанавливает соответствующие биты в регистрах состояний вв/выв. Процессор д. периодически проверять состояние этих битов, т.е. опрашивать соответствующие регистры контроллера. На процессор возложено непосредственное управление вв/выв: опрос битов, пересылка команд чтения/записи, передача данных. Т.о. необходимо иметь следующий набор команд: команды управления (исп. для инициализации внешних устройств, указывают им, что делать); команды анализа состояния (проверка состояний флагов вв/выв); передача данных (для чт/зап команд или данных из/в внешн. устр. в/из регистры процессора);

Недостаток: проц пост опрашивает флаги – остается мало времени для остального.

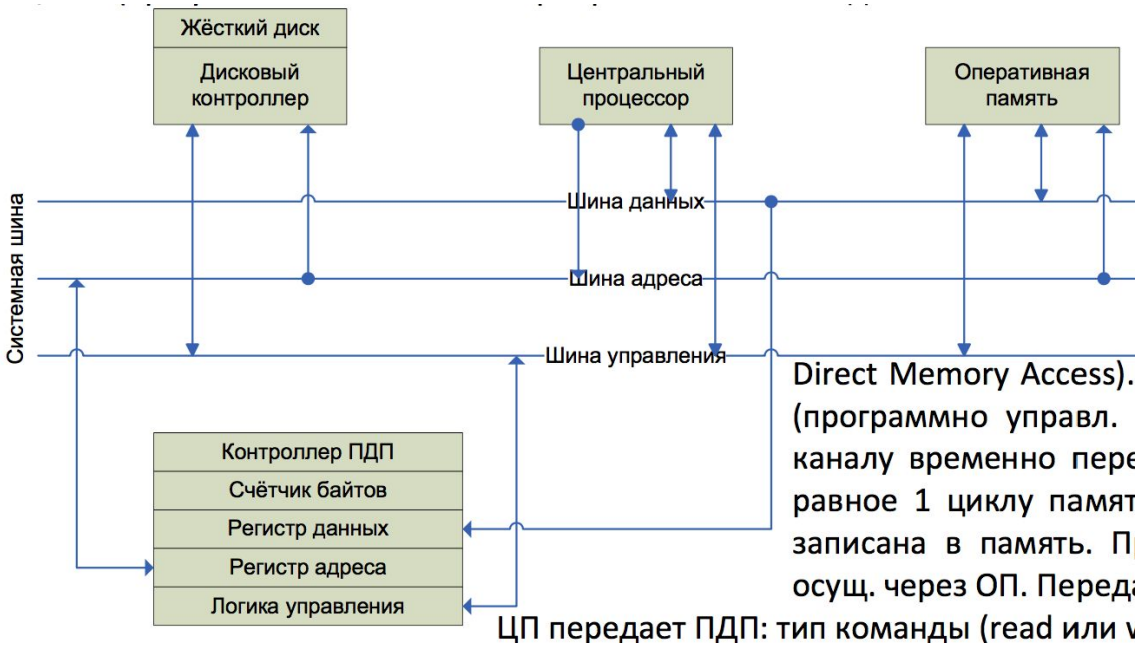
2) ***С использованием прерываний.*** Прерывание от устройства вв/выв поступает, когда оно завершило процесс вв/выв. Процессор освобождается от проверки флагов и м. переключиться на др. работу. Метод требует включ. в состав ОС контроллера прерываний. Контроллер прерываний посылает по шине управления сигнал. В конце выполнения каждой команды процессор проверяет входной сигнал с шины управления (если прерывания не замаскированы в ОС). Если получен сигнал – посылается ответный сигнал контроллеру прерываний, в ответ контроллер прерываний формирует и посылает вектор прерывания. Вектор передается по шине данных. Полученный вектор используется для процедуры обработки прерыв.

Недостатки Кажд. прерыв. вызывает остановку выполнен. текущих процессов и сохранение аппаратного контекста, любое слово, передающ. между памятью и устройством, д. пройти через регистры процессора. Но все равно намного эффективнее программируемого, т.к. искл. активное ожидание на процессоре.

3) *Прямой доступ к памяти* (ПДП, DMA – Direct Memory Access). Для управления ПДП подключ. контроллер ПДП (программно управл. устр. – спецпроцессор). Устройству вв/выв или каналу временно передается управление памятью (обычно на время, равное 1 циклу памяти), благодаря чему слово или группа слов м.б. записана в память. Процессор своей памяти не имеет, поэтому все осущ. через ОП. Передача – через регистры процессора. ЦП передает ПДП: тип команды (read или write); адрес устройства; начальный адрес блока в ОП, используемого для чтения или записи; размер передаваемого блока.

+ Сокращение числа прерываний

– Контроллер ПДП отстаёт по быстродействию от ЦП. Поэтому, если контроллер ПДП не может поддерживать текущую скорость IO, то ПДП не используется, а используются 2 предыдущих способа.



Если требуется заполнить ячейки памяти, расположенные по подряд идущим адресам, используется **«пакетный»** (англ. *burst*) режим работы шины: размер данных записывается в регистр контроллера DMA; первый цикл используется для передачи адреса первой ячейки; последующие циклы используются для пересылки данных указанного размера.

В режиме идентификации состояния процессора (или “**прозрачном**” ПДП) контроллер ПДП занимает шину тогда, когда процессор выполняет внутренние действия по преобразованию данных и не обращается к шине. Процессор (или дополнительная схема) идентифицирует такие промежутки времени для контроллера ПДП специальным сигналом, означающим доступность шины. Производительность процессора в таком режиме не уменьшается, (процессор “не замечает” что происходит ПДП, ПДП “прозрачный”), но контроллер ПДП оказывается жестко “привязанным” к схеме процессора, а сами передачи носят нерегулярный характер, что ведет к уменьшению скорости передачи данных.

В режиме с **кражой циклов** контроллер ПДП при необходимости сигналом запроса ПДП **DMA REQ** (Direct Memory Access REQuire) или **HOLD** заставляет процессор отключиться от системной шины на несколько тактов. После восприятия запроса ПДП процессор завершает выполнение текущего цикла доступа к памяти, приостанавливает свои действия и информирует об этом контроллер ПДП сигналом разрешения прямого доступа к памяти **HLDA** (HoLD Acnowlege). Задатчиком на системной шине становится контроллер ПДП, он передает между памятью и портом внешнего устройства один элемент данных - байт или слово. Внутренние операции процессора могут совмещаться с передачами ПДП. За одно инициирование ПДП может передаваться и блок данных, например, сектор диска. Таким образом, передачи ПДП осуществляются путем пропуска тактов в выполняемой программе.

4. ОС - определение, место ОС в системе ПО ЭВМ. Ресурсы пользовательской системы.Режимы ядра и задачи - переключение в режим ядра - классификация событий. Процесс как единица декомпозиции системы. Диаграмма состояний процесса.

ОС – комплект программ, которые совместно управляют ресурсами вычислительной системы и процессами, которые используют эти ресурсы в вычислениях.

Классификация ОС:

1) Однопрограммная пакетной обработки – в оперативной памяти м.б. только 1 прикладная программа.
2) Мультипрограммная пакетной обработки – в оперативной памяти одновременно много программ. Загрузка – перфокартами.
3) Мультипрограммная с разделением времени – в оперативной памяти одновременно большое число программ, процессорное время – квантуется (чтобы обеспечить фиксированное время ответа <= 3 с).
4) Реального времени – главное – время отклика. Организует работу вычислительной системы в темпе, обеспечивающем обслуживание некоторого внешнего процесса не зависимо от вычислительной системы. Время отклика системы <= время поступления запроса на ответ. В ситемах реального времени используется система абсолютных приоритетов.
   a. Жесткая система (строгое обеспечение времени ответа).
   b. Гибкая система (цифровые аудио и мультимедийные средства).
5) Сетевые (серверные).
6) Многопроцессорные.
7) Встроенные (в ТВ, микроволновки) – Windows CE.
8) ОС для смарт-карт.

Ресурс – любой из компонентов вычислительной системы и предоставляемые им возможности.

Ресурсы: процессорное время, объем ОП, внешние устройства, объекты ядра, реентерабельные коды ОС (коды, не модифицирующие сами себя (Реинтерабельность - свойство повторной входимости).

Процесс – единица декомпозиции системы, потому что ему выделяются ресурсы системы.

Процесс – программа в стадии выполнения.

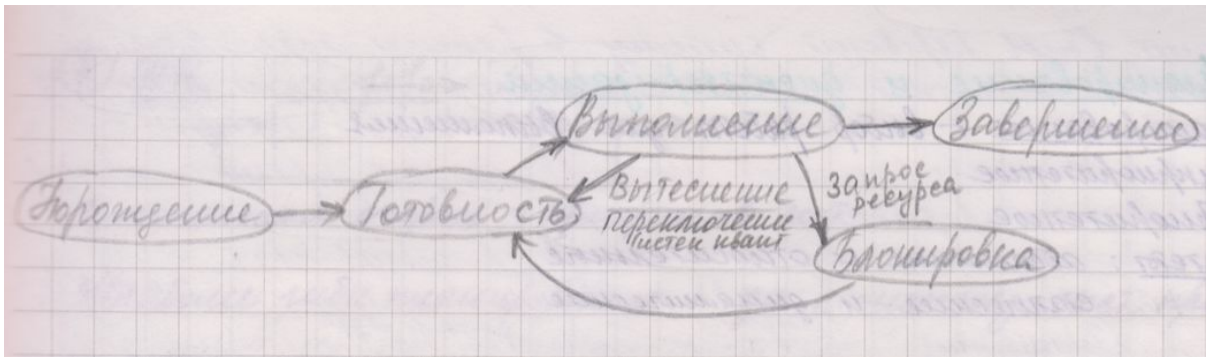
Пользовательский режим - наименее привилегированный режим, поддерживаемый NT; он не имеет прямого доступа к оборудованию и у него ограниченный доступ к памяти.

Режим ядра - привилегированный режим. Те части системы, которые исполняются в режиме ядра, такие как драйверы устройств и подсистемы типа Диспетчера Виртуальной Памяти, имеют прямой доступ ко всей аппаратуре и памяти. Различия в работе программ пользовательского режима и режима ядра поддерживаются аппаратными средствами компьютера (а именно - процессором).

Переключение процесса в режим ядра

Существуют 3 типа событий, которые могут перевести ОС в **режим ядра**:

1) системные вызовы (программные прерывания) – software interrupt – traps
2) аппаратные прерывания (прерывания, поступившие от устройств) - interrupts (от таймера, от устройств I/O, прерывания от схем контроля: уровень напряжения в сети, контроль четности памяти)
3) исключительные ситуации - exception



5. Классификация операционных систем и их особенности. Виртуальная и иерархическая системы - декомпозиции системы на уровне иерархии, ядро системы - определение.

ОС – комплект программ, которые совместно управляют ресурсами вычислительной системы и процессами, которые используют эти ресурсы в вычислениях.

Классификация ОС:

1) Однопрограммная пакетной обработки – в оперативной памяти м.б. только 1 прикладная программа.
2) Мультипрограммная пакетной обработки – в оперативной памяти одновременно много программ. Загрузка – перфокартами.
3) Мультипрограммная с разделением времени – в оперативной памяти одновременно большое число программ, процессорное время – квантуется (чтобы обеспечить фиксированное время ответа <= 3 с).
4) Реального времени – главное – время отклика. Организует работу вычислительной системы в темпе, обеспечивающем обслуживание некоторого внешнего процесса не зависимо от вычислительной системы. Время отклика системы <= время поступления запроса на ответ. В ситемах реального времени используется система абсолютных приоритетов.
    c. Жесткая система (строгое обеспечение времени ответа).
    d. Гибкая система (цифровые аудио и мультимедийные средства).
5) Сетевые (серверные).
6) Многопроцессорные.
7) Встроенные (в ТВ, микроволновки) – Windows CE.
8) ОС для смарт-карт.

Иерархическая машина – ОС разбивается на функции и определяется место этих функций по удаленности от аппаратной части.

Виртуальная машина - ОС работающая на определенной конфигурации аппаратной части предоставляет в распоряжение пользователя некоторую виртуальную машину. Скорость работы зависит от конфигурации.
ВМ - программная и/или аппаратная система, эмулирующая аппаратное обеспечение некоторой платформы
Ядро́ (kernel) — центральная часть операционной системы (ОС), обеспечивающая приложениям координированный доступ к ресурсам компьютера, таким как процессорное время, память, внешнее аппаратное обеспечение, внешнее устройство ввода и вывода информации
Существует два типа структур ядер:
1. **Монолитное ядро**
- ядро – это все, что выполняется в режиме ядра
- ядро представляет собой единую программу с модульной структурой (выделены функции, такие как планировщик, файловая система, драйверы, менеджеры памяти)
- при изменении к.-л. функции нужно перекомпилировать все ядро
- такие ОС делятся на две части – резидентную и нерезидентную

Пример – Unix, хотя она имеет минимизированное ядро (часть функций вынесены в shell).
2. **Микроядро** – имеет блочную структуру, сост. из п/п. Модуль ОС, обеспечив. взаимод. между процессами ОС, и взаимод. пользоват. процессов между собой и с процессами ОС. Также вкл. функции самого низкого уровня – выделение аппаратных ресурсов. Драйверы имеют многоуровневую структуру. Драйвер-фильтр позвол. менять функциональность, не трогая драйвер нижнего уровня. Общение между компонентами происходит с помощью сообщений через адресное пространство микроядра. Микроядерная архитектура основана на модели клиент-сервер (например, ОС Mach, Hurd и Win2k(но не в классическом понимании)). (К серверам ОС относятся: сервер файлов, процессов, безопасности, виртуальной памяти)

6. Процессы: бесконечное откладывание, зависание, тупиковая ситуация - анализ на примере задачи об обедающих философах. Считающие и множественные семафоры. Мониторы: монитор кольцевой буфер.

Процесс – единица декомпозиции системы, потому что ему выделяются ресурсы системы. **Процесс** – программа в стадии выполнения.

Асинхронные процессы – процессы, каждый из которых выполняется с собственной скоростью. Все процессы в системе являются асинхронными, то есть нельзя сказать, когда процесс придёт в какую-то точку.

1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критичские секции - нормально
2. **Бесконечное откладывание (зависание)** – ситуация, когда разделённый ресурс снова захв. тем же процессом.
3. **Тупик** (deadlock, взаимоблокировка) – ситуация, когда оба процесса установили флаги занятости и ждут. Т.е. каждый ожидает освобождения ресурса, занятого другим процессом.

Проблема обедающих философов. Существуют только 3 схемы действия философов:

1. пытается взять обе вилки сразу. Если удается, он может есть. Бесконечное откладывание
2. берет левую вилку и пытается взять правую (левую держит в руке) Тупик
3. берет левую вилку, и если не удается взять правую, то кладет левую вилку обратноOk

Семафор – неотрицательная защищённая переменная S, над которой определено 2 неделимые операции: P (от датск. passeren - пропустить) и V (от датск. vrygeven - освободить). Защищенность семафора означает, что значение семафора м. изменяться только операциями P и V.

1. Операция P(S): S = S – 1. Декремент семафора (если он возможен). Если S = 0, то процесс, пытающийся выполнить операцию P, будет заблокирован на семафоре в ожидании, пока S не станет больше 0. Его освобождает другой процесс, выполняющий операцию V(S).
2. Операция V(S): S = S + 1. Инкремент S одним неделимым действием (последовательность непрерывных действий: инкремент, выборка и запоминание). Во время операции к семафору нет доступа для других процессов. Если S = 0, то V(S) приведёт к S = 1. Это приведёт к активизации процесса, ожид. на семафоре.

P(S) и V(S) есть неделимые (атомарные) операции.

Суть: процесс пытающий выполнить операцию P(S) блокируется, становится в очередь ожидания данного семафора, освобождает его другой процесс, который выполняет V(S). Таким образом исключается активное ожидание.

Семафоры поддерживаются ОС-ой. Семафоры устраняют активное ожидание на процессоре.

Семафоры бывают: бинарные (S принимает значения 0 и 1), считающие (S принимает значения от 0 до n), множественные (набор считающих семафоров). Все семафоры 1 набора м. устанавливаться 1 операцией.

В Windows после освобождения на семафоре приоритет процесса повыш. Изменение S м. рассматривать как событие в ОС.

Использование семафоров часто приводит к взаимоблокировке. Понятие «монитор» предложил Хоар.

Монитор – языковая конструкция, состоящая из структур данных и подпрограмм, использующих данные структуры. Монитор защищает данные. Доступ к данным монитора могут получить только п/п монитора. В каждый момент времени в мониторе м. находиться только 1 процесс. Монитор является ресурсом.

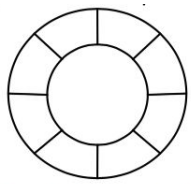
Процесс, захвативший монитор, – процесс **в** мониторе, процесс, ожидающий в очереди, – процесс **на** мониторе. Используется переменная типа «событие» для каждой причины перевода процесса в состояние блокировки.

wait – откр. доступ к монитору, задержив. выполнение процесса. Оно д.б. восстановлено операцией signal др. процесса.

Если очередь перем. к типу усл. <> 0, то из очереди выбирается 1 из процессов и инициализируется его выполн.

Монитор «кольцевой буфер» – решает задачу «производство-потребление», то есть существуют процессы-производители и процессы-потребители, а также буфер – массив заданного размера, куда производители помещают данные, а потребители считывают оттуда данные в том порядке, в котором они помещались (FIFO).

считывают оттуда данные в том порядке, в котором они помещались (FIFO).



```
resource: monitor circle_buffer;
const n = size;
var   buffer: array [0..n-111]]] of <type>;
      pos, write_pos, read_pos: 0..n-1;
      full, empty: conditional;
procedure producer(p: process, data:<
type>)
begin
      if (pos = n) then
            wait(empty);
      buffer[write_pos] := data;
      Inc(pos);
      write_pos = (write_pos + 1) mod n;
      signal(full);
end;
begin       pos := 0;   write_pos := 0;   read_pos := 0;    end.
```

```
procedure consumer(p: process, var data:
<type>)
begin
      if (pos = 0) then
            wait(full);
      data := buffer[read_pos];
      Dec(read_pos);
      read_pos := (read_pos + 1) mod n;
      signal(empty);
end
```

7. Виртуальная подсистема proc - назначение, особенности, файлы, поддиректории и ссылка self. Структура proc_dir_entry: функции для работы с элементами /proc. API - интерфейсы для доступа к адресному пространству пользователя.

Unix поддерживает виртуальную файловую систему proc. В зависимости от версии ядра, м.б. procfs. Не является монтируемой ФС. Это пример виртуально ФС. Это интерфейс ядра, позволяющий приложениям читать и изменять данные в адрес ном пространстве других процессов, управлять прцоессами, получать информацию о процессах и ресурсах используемых процессами путем использования стандартного интерфейса ФС и системных вызовов. Следует, что управление доступом к адресному пространству осуществляется с помощью обычных прав доступа, а именно чтение/ запись/выполнение. По умолчанию запись/чтение файлов proc разрешены только для владельцев. В ФС proc данные о каждом процессе хранятся в поддиректории /*proc*/ < *PID* > В поддиректории процесса находятся файлы и поддиректория, со держащие данные о процессе.

элемент	тип	содержание
cmdline	файл	указатель на dir процесса
cwd	символ ссылка	указатель на dir процесса
environ	файл	список окружения процесса
exe	символ ссылка	указывает на образ процесса (на файл)
fd	директория	ссылки на файлы, открытые процессом
root	символ. ссылка	на корень ф.сист. процесса
stat	файл	информация о прцоессе.

Виртуальная ФС – информация о процессах. Виртуальная – потому что не на диске. Текстовый файл с нужной инфомацией создается, когда выполняется запрос на чтение соответствующей информации. Взаимодейсвтие с ядром заключается в чтении информации из ядра / записи новой информации в ядро.

Примеры:

/*proc*/*pci* - устройства, подключенные в текущий момент к шине PCI /*proc*/*apm* - информация о состоянии батареи ноутбука.

Файлы и поддиректоррии ФС proc могут легко создаваться, регистрировать и прекращать ??? динамически.

Кроме ФС proc есть аналогичные sysfs – ФС имеет сходную функциональ ность и более удачную структуру, т.к. учтен опыт ФС proc. Однако ФС proc является укоренившейся привычкой, и будет оставаться таковой.

debugfs – представляет из себя отладочный интерфейс.

Структура proc_dir_entry:
//используется для работы с файлами и поддиректорями
struct proc_dir_entry {
// номер индекса inode файла.
unsigned int low_ino ;
//имя файла
const char *name ;
//длина имени
unsigned short namelen ;
//права доступа
mode_t mode;
// колво− ссылок на файл
nlink_t nlink ;
uid_t uid ;
gid_t gid ;
//размер
unsigned long size;
struct inode_operations *proc_iops;
struct file_operations *proc_fops;
get_info_t *get_info;
struct module *owner;
//указатели , позволяющие создать связанные списки
struct proc_dir_entry *next, *parent, *subdir;

read_proc_t *read_proc;
write_proc_t *write_proc;
//колво− ссылок
atomic_t count ;
int deleted ;
}

Для удаления файла из ФС proc используется ф-я:

```
1 void remove_proc_entry(const char *name, struct proc_dir_entry *parent);
2 //если parent = NULL, то будет работать с корневым /proc/
```

Таблица 1.2 — Список предопределенных переменных

переменная	путь
`proc_root_fs`	*/proc*
`proc_net`	*/proc/net*
`proc_bus`	*/proc/bus*
`proc_root_driver`	*/proc/driver*

Структура `proc_dir_entry` используется функциями для создания указателя, функцией `create_proc_entry()`

```
1 struct proc_dir_entry create_proc_entry() (const chat *name, mode_t mode,
    struct proc_dir_entry *base);
2 /* принимает:
3 1) имя создаваемого файла;
4 2) права доступа;
5 3)  подкаталог, где будет размещен файл если( = NULL, то в корневом каталоге
    proc);возвращает
6
7 :указатель
8  на структуру. Если неудалось создать файл, будет возвращен NULL.
9 */
```

Начиная с версия ядра 3.10 используется функция:

```
1 struct proc_dir_entry *proc_create_data(const char *name, umode_t mode,
    struct proc_dir_entry *parent, const struct file_operations *proc_fops,
    void *data);
```

Работает с ФС `procfs`. Позволяют создать виртуальный файл в ФС proc.

Для доступа к информации ядра и записи информации в ядро используются ф-и (это не ф-и, относящиеся к структуре `proc_dir_entry`, это ф-и которые позволяют записать информацию в ядро и считать инфо из ядра):

```
1 #include <asm/uaccess.h>
2 //часто вместо copy_to_user используется sprintf, поскольку инфа записывается в
    буфер
3 unsigned long copy_to_user(void _user *to, const void *from, unsigned long
    n);
4 unsigned long copy_from_user(void *to, const void _user *from, unsigned
    long n);
```

8. Классификация ядер ОС. Особенности ОС с микроядром: реализация взаимодействия процессов по модели клиент-сервер. Три состояния процесса при передаче сообщений. Достоинства и недостатки микроядерной архитектуры. Многопоточное ядро; взаимоисключение в ядре – спин - блокировки.

Ядро́ (kernel) — центральная часть операционной системы (ОС), обеспечивающая приложениям координированный доступ к ресурсам компьютера, таким как процессорное время, память, внешнее аппаратное обеспечение, внешнее устройство ввода и вывода информации

ОС разбивается на несколько уровней, причем обращение через уровень невозможно (непрозр. интерфейс).

В качестве ядра выделяется самый низкий уровень распределения аппаратных ресурсов процессам самой ОС (непосредственное обращение с аппаратурой).

Существует два типа структур ядер:

1. **Монолитное ядро**

− ядро – это все, что выполняется в режиме ядра
− ядро представляет собой единую программу с модульной структурой (выделены функции, такие как планировщик, файловая система, драйверы, менеджеры памяти)
− при изменении к.-л. функции нужно перекомпилировать все ядро
− такие ОС делятся на две части – резидентную и нерезидентную

Пример – Unix, хотя она имеет минимизированное ядро (часть функций вынесены в shell).

2. **Микроядро** – имеет блочную структуру, сост. из п/п. Модуль ОС, обеспечив. взаимод. между процессами ОС, и взаимод. пользоват. процессов между собой и с процессами ОС. Также вкл. функции самого низкого уровня – выделение аппаратных ресурсов. Драйверы имеют многоуровневую структуру. Драйвер-фильтр позвол. менять функциональность, не трогая драйвер нижнего уровня. Общение между компонентами происходит с помощью сообщений через адресное пространство микроядра. Микроядерная архитектура основана на модели клиент-сервер (например, ОС Mach, Hurd и Win2k(но не в классическом понимании)). (К серверам ОС относятся: сервер файлов, процессов, безопасности, виртуальной памяти)

Модель клиент-сервер:

Система рассматривается как совокупность двух групп процессов
процессы-серверы, предоставляющие набор сервисов
процессы-клиенты, запрашивающие сервисы

Принято считать, что данная модель работает на уровне транзакций (запрос и ответ – неделимая операция). 3 состояния процесса при передаче сообщения(протокол обмена):

запрос: клиент запрашивает сервер для обработки запроса

ответ: сервер возвращает результат операции
подтверждение: клиент подтверждает прием пакета от сервера
Для обеспечения надежности обмена в протокол обмена могут входить следующие действия:
сервер доступен? (запрос клиента) перезвоните (сервер – недоступен)
сервер доступен (ответ сервера) адрес ошиб. (процесса с No нет в ОС)
+ Высокая степень модульности ядра ОС (упрощает добавление новых компонентов). М. загружать и выгружать нов. драйверы, файловые системы и т. д., т.о. упрощается процесс отладки компонентов ядра.
+ Компоненты ядра *ОС* принципиально не отличаются от пользовательских программ для их отладки можно применять обычные средства.
+ Повышается надежность системы, т.к.
ошибка на уровне непривилег. программы < опасна, чем отказ на уровне режима ядра.
− *Микроядерная архитектура ОС* вносит дополнительные накладные расходы, связанные с передачей сообщений, что сущ. влияет на производительность.
− Для того чтобы микроядерная ОС
по скорости не уступала *ОС* на базе *монолитного ядра*, треб. очень аккуратно проектировать разбиение системы на компоненты, стараясь минимизировать взаимодействие между ними.

9. Управление устройствами: физические принципы управления устройствами. буферизация ввода-вывода - управление буферами. Буферный пул, кеширование.
В кристалле процессора есть кэши. Они содержат актуальную информацию, ту, к которой в последнее время обращался. При вв/выв используется буферизация.
Одинарный буфер – выделяется в пространстве ядра ОС. Перед вв/выв процессу назначается буфер. Сначала данные помещаются в буфер, затем копируются в адресное пространство процесса. Операция вв/выв выполняется быстрее, т.к. процесс не блокиров. – все данные получ. сразу). Опережающее считывание. Исключаются проблемы, т.к. буфер находится в системной (невыгружаемой) области памяти.
Схемы передачи информации: блок-ориентированная (поточно-ориент.), побайтно-ориент. (построчно). Если процесс пытается поместить в буфер строку, а он занят, то процесс будет блокирован (пр-во/потребл.) Используется **двойная буферизация**: 1 буфер передается, др. -использ-ся для заполнения – усложняются действия ОС Проблема буферизации остро встает в ОС реальн. вр. 2 способа перемещения: 1. Копирование – передача последов. байт в обл. памяти. 2. Мэпинг – получение указателя на адресное пространство в ядре ОС. ОС должна сообщать о завершении вв/выв приложению. Классич. Unix не выполняет асинхронного вв/выв, предоставл. прикл. программе чисто синхр. интерфейс. Современный Unix позволяет программисту выбирать отложенной записью (Fire&Forget) и чисто синхр.
Разработчик сам должен разрабатывать нити. Доп. нить должна сообщать основной о завершении операции вв/выв. Чаще всего исп. семафоры, сигналы, спинлоки, мьютексы. «–» усложнение программной логики.
В Windows 200/XP для асинхронного вв/выв исп. порты завершения или др. объекты синхронизации (реже). Порт завершения – механизм сообщения потокам о завершении операции вв/выв. Если файл сопоставить с портом завершения, то по окончании операции вв/выв в очередь порта завершения став. пакет завершения.
Приложение проверяет наличие пакета завершения в порте завершения и т.о. узнает о факте завершения. Независимо от типа запроса операции вв/выв инициализ. драйвером действия в интересах приложения выполняется асинхронно. После выдачи запроса драйвер возвращает управление подсистеме вв/выв. Когда подсистема вв/выв вернет управление зависит от типа запроса.

10. Взаимодействие процессов. Семафоры. И ещё что то про семафоры. (56 - 57 + pdf)

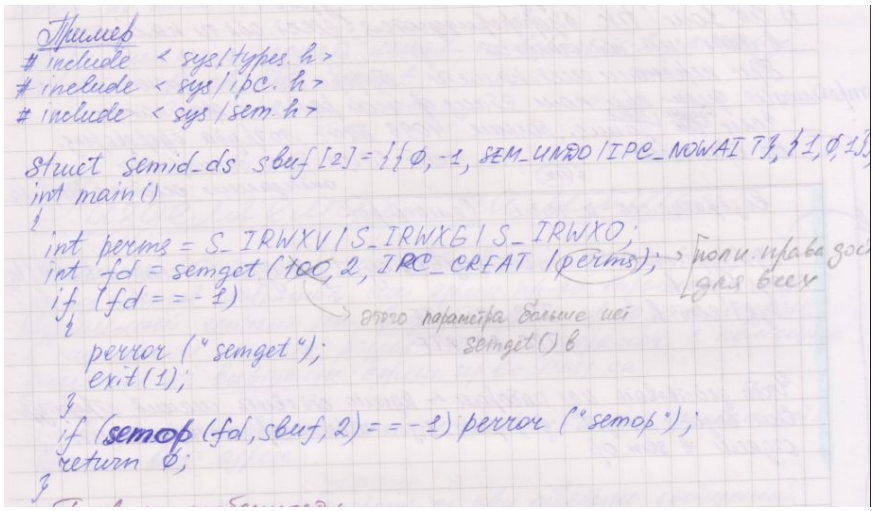
Речь идет о семафорах ядра KERNAL_SEMAPHORES. В обработчиках прерываний семафоры использовать нельзя. В 1 семестре мы изучали семафоры SYSTEMV, там были get_sem и sem_op, но мы говорим о семафорах ядра. Считающий семафор, имеется счетчик заблокированных и очередь потоков, пытающихся захватить данный семафор. На этих семафорах определены 2 операции: down up.

```
struct semaphore {
atomic_t count;
int sleepers;
wait_queue_head wait;
}
//инициализирует счетчик семафора sem−>count заданной величиной v a l.
void sem_init ( struct semaphore *sem , int val );
//пытвается захватить семафор
inline void down( struct semaphore *sem);
inline void up( struct semaphore *sem);
int down_interruptible (&sem);
int down_trylock(&sem);
```

Использование семафоров (по сравнению с спин блокировками) связано с значительными накладными расходами. Если поток ядра заблокирован на семафоре, то ядро может переключиться на выполннеие другой работы, вот и переключение контекста. Когда поток разблокируется, то это связано с переключением контекста. Получается 2 переключения. Поток владеет аппаратным контекстом. Из такого поведения семафоров, связанного с переводом процессов в состояние ожидания, можно сделать следующие интересные заключения: [6] а) Так как задания, которые конфликтуют при захвате блокировки, переводятся в состояние ожидания и в этом состоянии ждут, пока блокировка не будет освобождена, семафоры хорошо подходят для блокировок, которые могут удерживаться в течение длительного времени. б) Так как поток выполнения во время конфликта при захвате блокировки находится в состоянии ожидания, то семафоры можно захватывать только в контексте процесса. Контекст прерывания планировщиком не управляется. в) При захвате семафора нельзя удерживать спин-блокировку, поскольку процесс может переходить в состояние ожидания, ожидая на освобождение семафора, а при удержании спин-блокировки в состояние ожидания переходить нельзя. г) Семафоры не запрещают приемтивность ядра. Код который удерживает семафор может быть вытеснен. Следствием этого является то, что семафоры не влияют на латентность планировщика. (При удержании семафора (хотя разработчик может и не очень хотеть этого) процесс может переходить в состояние ожидания. Это не может привести к тупиковой ситуации, когда другой процесс попытается захватить блокировку (он просто переходит в состояние ожидания, что в конце концов дает возможность выполняться первому процессу).) д) Семафоры позволяют удерживать их любому кол-ву потоков. Кол-во потоков, которые

одновременно могут удерживать семафор определяются счетчиком использования при создании семафора. е) Спин блокировку может удерживать только один процесс/поток. Семафоры в ядре используются мало.

Unix/Linux содерж наборы считающих семафоров. Доступ к отд семафору - по индексу. Семафоры в сист поддерж табл семафоров (1 на систему) в это табл все создав в сист наборы семафоров кажд элемент таблицы содерж: 1) имя - целое число. Др процессы по этому имени могут открыть его 2) Uid создателя набора и идентиф его группы проц. uid совпад с uid созд - можно удалять набор и изменять парам. 3) Права доступа - для user, группы, others. 4) Кол-во семафоров в наборе 5) время измен 1ого или послед измен процессом. 6) время послед измен управл парам 7) указатель на массив семафоров. На семафорах определены сист вызовы: semget() - создание набора, semctl() - измен управл, semop() - измен обращ. Все сист вызовы должны проверяться на -1. 3 операции: semop > 0 - инкремент. проц вып sem_op и в его поле значение освобождает ресурс,sem_op = 0 - проверка освобождения ресурса без попытки захвата, sem_op < 0 - декремент - захват ресурсы. Главная особенность: одной неделимой операцией можно изменить все или часть семафоров.



Флаги: IPCNOWAIT - информ ядру о нежелании проц перехода в режим ожидания. Его наличие объясн стремлением избежать блокиров всех проц, которые могут оказаться в очереди к семафору. SEM_UNDO - указ ядру, что необх отслеживать изменения знач сем semop и при заверш проц сист ликвидир сделанные измен, чтобы проц не были заблок навечно.

11. Подсистема ввода-вывода: программные принципы управления устройством. Специальный файл - идентификация устройства в ОС Юникс, точки входа в драйверы.

(ПЕРВЫЕ ДВЕ СТРАНИЦЫ МОЖНО ВЫЧЕРКУНУТЬ И ДОБАВИТЬ ИНФУ ИЗ 3 ВОПРОСА)

Ввод-вывод – это часть вычислительной системы или деятельность, ориентированная прежде всего на обмен информацией с центральным процессором. Именно система ввода-вывода позволяет получить результаты работы процессора в удобной для человека форме и передавать процессору на выполнение программы и данные на обработку. Система ввода-вывода объединяет аппаратные и программные компоненты, предназначенные для ввода программ и данных, которые затем выполняются и обрабатываются процессором, и вывода информации на различные устройства, подключенные к системе. Система ввода-вывода включает две группы устройств: 1) устройства ввода-вывода и 2) запоминающие устройства.

Часто в качестве основной характеристики быстродействия компьютера указывается частота, с которой работает центральный процессор. Время реакции системы на внешнее воздействие или событие зависит как от быстродействия и организации подсистемы памяти, так и от организации системы ввода-вывода.

Два подхода к проектированию: Первый – процессор выполняет управление вводом-выводом и всю обработку ввода-вывода. Процессор постоянно находится в ожидании “готовности” устройств ввода-вывода. Второй подход связан с включением в систему специальных процессоров, управляющих устройствами ввода-вывода и работающих параллельно с центральным процессором. Производительность вычислительной системы снижается, но значительно меньше, чем в первом случае. Каждый раз, когда процессу требуется какой-то системный ресурс, выполняется обращение с системе ввода-вывода. При управлении параллельными процессами супервизор постоянно обращается к подсистеме ввода-вывода для сохранения на диске контекста вытесняемого или блокируемого процесса и считывает с диска контекст активизируемого процесса. **Задачи подсистемы ввода-вывода**: 1) система должна эффективно управлять всеми устройствами; взаимодействие центрального процессора с внешними устройствами, при котором минимизируется время, затрачиваемое ЦП на управление периферией, и растет время параллельной работы ЦП и периферийных устройств; 2) система должна обеспечивать доступ к устройствам ввода-вывода множества параллельно выполняющихся задач, 3) скрывать от пользователя детали процесса ввода-вывода между программами и внешними устройствами; **Способы управления вводом-выводом:** 1. Упорядоченный опрос, заключающийся в периодической проверке всех регистров состояний занятых устройств. Процессорное время тратится нерационально. 2. Прерывание вызывается устройством ввода-вывода, когда оно снова готово к пересылке информации. При использовании прерываний процессор посылает на внешнее устройство очередную команду и, не ожидая ее завершения, переходит к обслуживанию других процессов, переведя процесс, выполнивший команду ввода- вывода в состояние блокировки (при синхронных

операциях ввода-вывода). Когда внешнее устройство завершит выполнение операции ввода-вывода, например, передаст прочитанные данные через канал прямого доступа в память, оно генерирует прерывание, которое заставит процессор завершить операцию ввода-вывода и перевести процесс, который был блокирован в ожидании ввода-вывода, обратно в очередь готовых процессов.

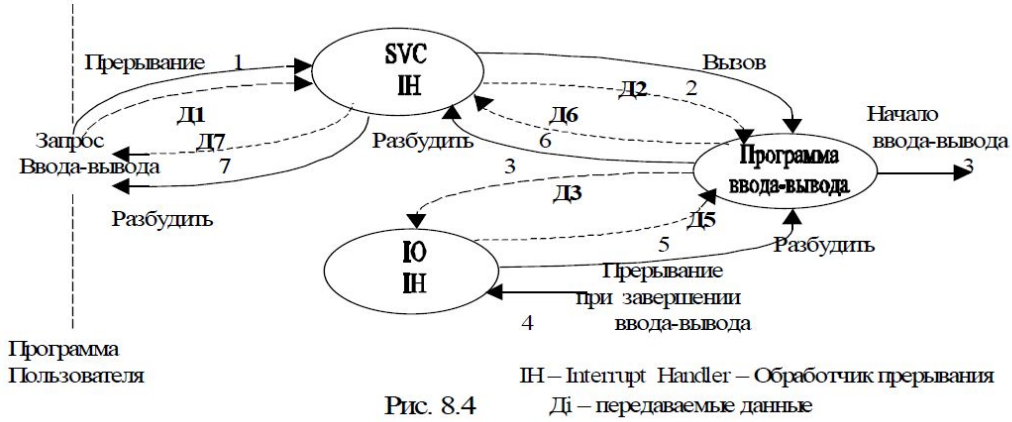


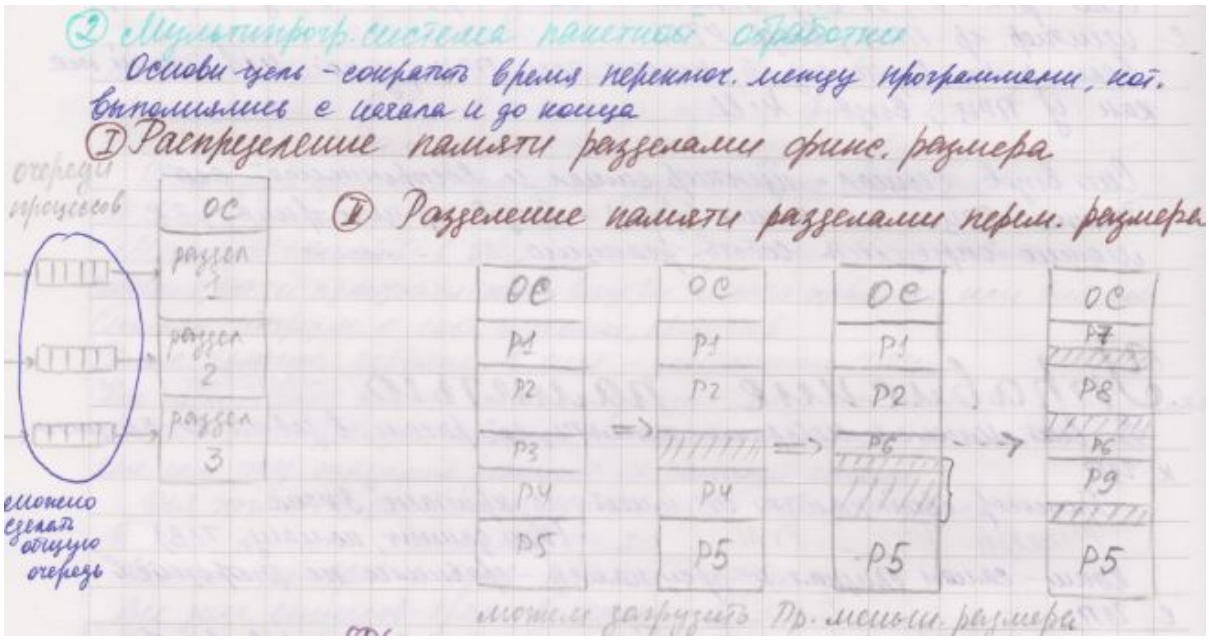
Рис. 8.4

На многих ЭВМ пользовательский процесс запрашивает обслуживание ввода-вывода посредством обращений к ядру ОС вызов супервизора. Такое обращение приводит к прерыванию активного процесса и передаче управления обработчику прерываний. Супервизор вызовет требуемую стандартную программу ввода-вывода и передаст ей параметры запроса, полученные от пользовательского процесса. Программа ввода-вывода передает свой идентификатор обработчику прерываний. Прерывание ввода-вывода, сигнализирующее о завершении операции ввода-вывода, вызовет обработчик прерываний ввода-вывода, который определит нужный процесс ввода-вывода и активизирует его. Процесс ввода-вывода может послать сигнал обработчику прерываний SVC, который вернет управление исходному процессу, запросившему ввод-вывод. Декларация: **в UNIX всё файл**. Следствие этого - работа с файлами и устройствами одними и теми же системными вызовами позволила сократить количество системных вызовов. Монтировать ФС может только привилегированный пользователь. Точка монтирования – каталог. Устройство – специальный файл, с помощью которого система получает доступ к физическому устройству. В командной строке mount. В многопроцессной системе разделения времени на таймер возлагается задача декремента кванта. Если пришло прерывание, то сохраняется аппаратный контекст выполняемой программы. В счетчики команд устанавливается адрес обработчика прерываний. Система переключается в режим ядра. Обработчики прерываний в системе это одна из точек входа драйвера. Это программа, которая в системе управляет работой внешнего устройства до какого то момента. Предназначен, чтобы послать и сформировать команду для устройства. Другая задача драйвера – получить от устройства данные и вернуть их процессу, который запросил эти данные. Драйверы – многовходовые программы. Ни одна программа не может на прямую обратиться к устройствам ввода/вывода. Сделано из соображений

безопасности. Иначе любой процесс может обратиться в любое место ОС, в частности к системным таблицам.
(здесь бы неплохо хорошие фотографии картинок с лекции где драйвера устройств)

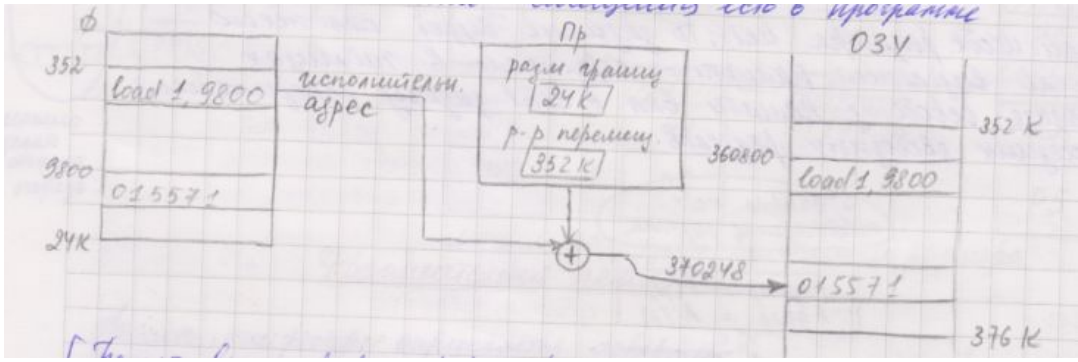
12. <u>Виртуальная память:</u> распределение страницами по запросам, свойство локальности, анализ страничного поведения процессов, рабочее множество. <u>Стратегии выделения памяти; фрагментация</u>.

В системе имеется иерархи памяти, кот. рассматривается в зависимости от близости к ЦП. Процессор своей памяти не имеет, на кристалле 3 кэша. (кэш данных, команд, TLB)
Кэши - самая близкая к проц. память - сравнима по быстродейств.
Вертик. управл - передача данных от уровня к уровню
Горизонт управл - управл внутри одного уровня. Задача файлов. сист - обеспечить доступ к информации. Управл. памятью = управл. ОЗУ



Фрагментация памяти. Память как ресурс определяется иерархией. Кэши содержат актуальную информацию. ОП по быстродейств. остает от быстродейтсв П. П с памятью связан лок. шиной. Все делается под управлением тактовых импульсов.
Этапы управления: 1) одиночное непрерывное распределение. 2) Мультипроцессные ОС. Память на разделы фиксир размера. 2 задачи: 1. управление памятью, чтобы иметь возможность выделять память и инф. о том, где занято, а где свободно.
Если необ размера нет, то задание будет отложено. После выделения памяти отражаем в таблицах. Задача освобождения памяти вкл. в себя задачу объединения соседних свободных разделов.
Стратегии выделения памяти. 1) Первый подходящий по размеру раздел 2) самый узкий (max близкий по разм к программе). 3) Самый широкий (после загрузки остаенется достаточн адресн. пр-ва для еще како-нибудь программы). Все методы требуют сортировки инф-ции о свобод. обл по возр. размера области.

Перемещаемые разделы Введение логических адресов. Каждая программа думает, что начинается с нулевого адреса.

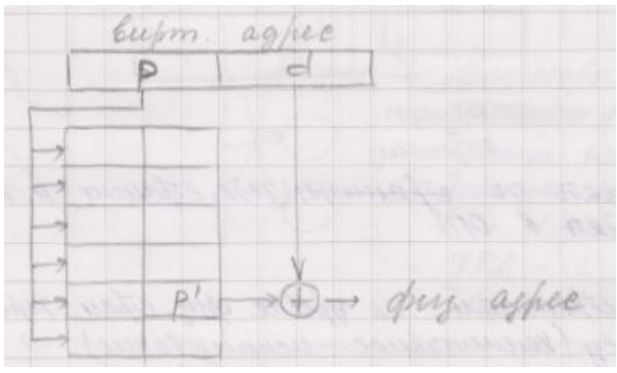


Связанное распределение памяти память выдел прогр непрерывн разделам адр пр-ва, т.е. программа грузится целиком последовательно. Можем отказаться от связанного распредления: поделить погр на логич разделы, выделить сегменты (кода, данных, стека), поделить адр пр-во на блоки одного размера.

Несвязанное распределение Требует преобраз. адресов, доп действий. Не надо целиком грузить целиком программу, чтобы ее выполнить. Нужно хранить только то, что выполн в данный момент.

Виртуальная память. Чтобы создать иллюзию - в сист производятся спец действия. Существует 3 схемы управл вирт. памятью. 1) Страницами по запросам 2) сегментами по запросам 3) сегментами поделенными на страницы по запросу.

“по запросу” - то есть загрузка в память осуществляетс по запросу кот. возникает при обращ прогр к команде или данным кот нет в оп. Если обратиться к диагр процессорв - созд процесса связано с идентификацией. Для начала работы - проц выдел min кол-во страниц с точки входа. В рез-те послед действ. возникает страничное прерывание и тогда память загрузит необх страницу. Дисковое адресное пространство делится на 2 части - большая отведена для хранения файлов и управл файлов системой, а меньш для swapping или paging. (сегменты/страницы). Вирт. память можно представить, как пролонгированную в дисковое пространство. В результате формируется вирт. адрес пр-во процесса - в сист описывается соотв. таблицами (картами трансляции адресов, т.е. таблицы страниц/сегментов.

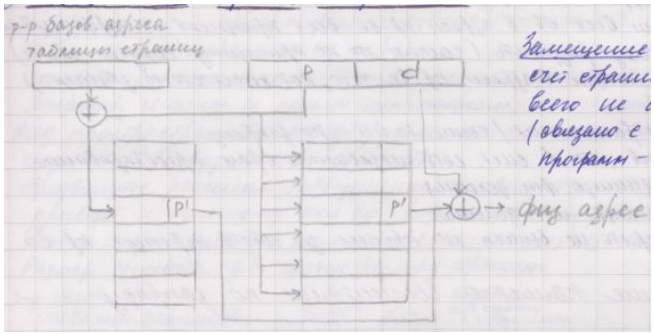


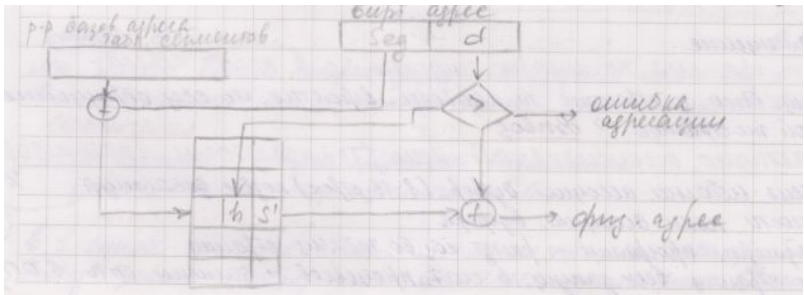
Методы преобразования адресов: 1) Прямое отображение. В процессоре имеется регистр, в котор загруж нек аодес при переключ контекста.

Табл страниц находится в ОП. Их кол-во равно кол-ву процессов. В данном случае постоянно обращ к памяти это не быстрый процесс. Данное преобразование пожирает процессорное время. 2) Ассоциативное отображение. дорогая ассоц память, кол-во элементов удваивается.

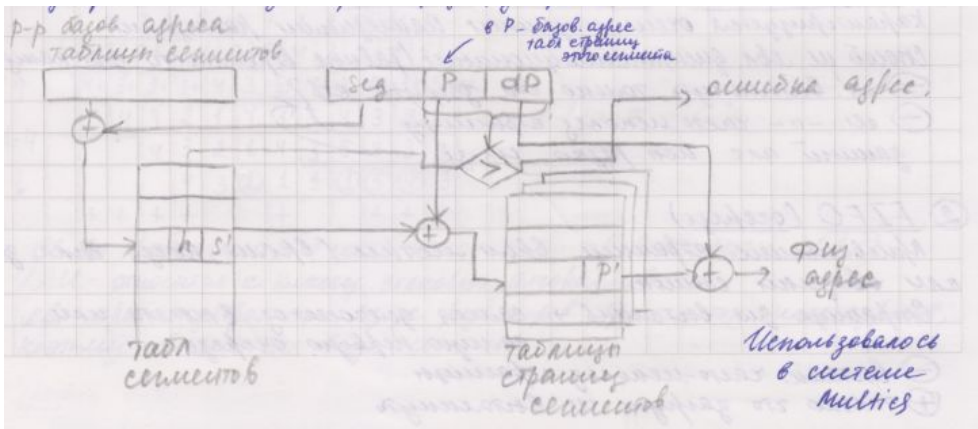
3) Ассоц. прямое отображение. Замещение выполняется за счет страниц, к которой дольше всего не было обращений. Если образ было к странице, то наиболее вероятно, что след обращение будет к этой же странице. Таблицы страниц всех запущенных в сист процессов - должны быть в ОП, но это пожирает все ресурсы.

Управление памятью сегментами по запросу. В 8 байт дискр сегмента - limit и base adress. Организация таблиц сегментов в системе: 1) Единая таблица, локальные таблицы, лок. табл + глоб.





Управление памятью сегментами, поделенными на страницы



Алгоритмы замещения страниц: 1) Выталкивание случайной страницы характеризуется очень низники накладными расходами. Минусы: может вытолкнуть только что загруженные страницы или часто использ. 2) FIFO очередь присваивает

странице врем. метки или связанный список. Выталк самую долгонаход в памяти или первую очередь. Минус вытал часто использ, Плюс только что загруж не вытолкнуть 3) Least recently used page replacement реализ через врем метки или связ список. Но при кажд обращ страницы, врем метка измен или страница переносится в хвост. + никогда не будет вытолкнута часто использ страница. - треб больших наклад расходов. На практике использ NUR (not used). Кажд странице присваивается бит обращения. При обращ к страниц став бит 1, для вытеснен первая с тр с битом 0. 4) Least frequently used - может быть вытолкнута только что загруженная страница, т.к. у нее маленький счетчик 5) Метод связанных пар. Связано с выбором размера стр. Т.е. разм стр. явл крайне важн вопросом. Чем больше страница, тем меньше выполн команд на ней.

Глобальное вытеснение любой страницы любым процессом при необходимости загр страницы для какого-то процесса (выбор из пула всех загр страниц)

Локальное вытеснение страниц для вытеснен выбирается из пула загруж стр данного процесса

Теория рабочего множества

Минимально необходимая память: Страница кода (точка входа), страница сегмента данных, страница сегмента стека. Получив квант процессора, процесс начинает интенсивно подкачивать страницы. За время существования процесс обратится к большей части своих страниц. Этот перегиб связан с тем, что процесс для своего нормального выполнения, без страничных прерываний (увеличивают время выполнения процесса). Если процессу удается загрузить в память все страницы, к которым он обращается, то он будет выполняться без страничных прерываний. Этот факт получил название – теория рабочего множества. Поведение программы, во время выполнения, с точки зрения загрузки страниц, не является стабильным. Страничное поведение процесса не является стабильным. Деннинг 1968 году предложил в качестве локальной меры произвольности взять число страниц, к которым программа обращается за интервал времени.

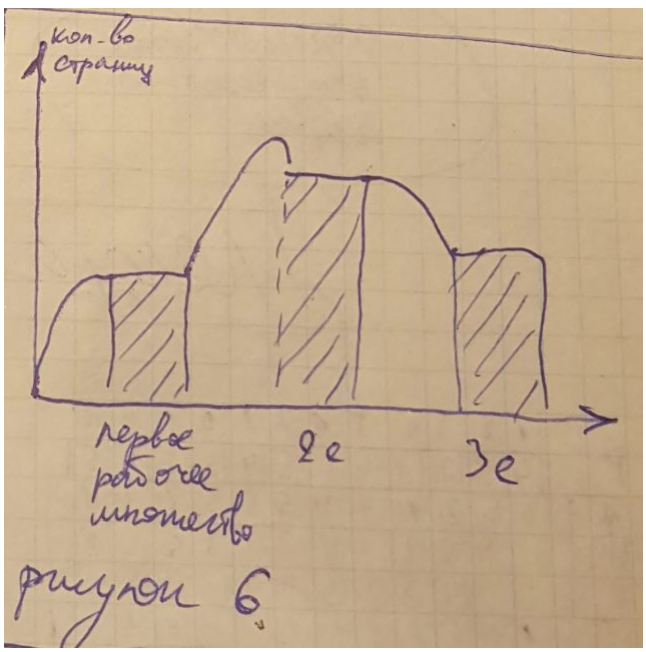


рисунок 6

В процессе жизни процесса он обращается к разным набором страниц. Загрузив рабочее множество, процесс будет выполняться без страничных прерываний.

Затем ему нужно перейти в другое рабочее множество.

13. Физические принципы устройств, структура контроллера, I/O mapping, memory mapping, гибридная схема, общая шина, выделенная шина, как с ними работать.

Каталог */dev* - это не файловая система как */proc*, это каталог специальных файлов: fd0, lp0, null, mem, mouse . . . Специальные файлы обеспечивают доступ к переферийным устройствам. Эти файлы обеспечивают связь между файловой системой и драйверами устройств. Такая интерпретация спец файлов обеспечивает доступ к внешним устройствам, как к обычным файлам. Аналогично обычному файлу с использованием тех же системных вызовов файл устройства может быть открыт/закрыт, в него можно писать/читать. Каждому внешнему устройству UNIX/Linux ставит в соответствие как минимум один специальный файл. Связь имени файла и устройства обеспечивает индексный дескриптор. Подкатолог */dev*/*fd* содержит файлы с именами 0, 1, 2, ... В некоторых системах имеются файлы с именами */dev*/*stdin*, */dev*/*stdout*, */dev*/*strerr* которые эквивалентны */dev*/*fd*/0, */dev*/*fd*/1, */dev*/*fd*/2. Драйвер - это программа, или часть кода кода ядра, которая предназначена для управления конкретным устройством. Обычно драйверы устройств содержат последовательность команд, специфичных для конкретного устройства. Поскольку драйвер предназначен управления устройством, то код должен соответствовать специфике устройства. Обычно это связано с форматом передачи данных от системы к

устройству и обратно. Если не знаем форматы данных , то управлять таким устройством мы не сможем. В линукс драйверы устройств бывают трех типов: а) встроенные в ядро б) реализованные как загружаемые модули ядра в) код драйверов этого типа поделен между ядром и специальной утилитой. Драйверы, встроенные в ядро Соответствующее устройство автоматически обнаруживаются системой и становятся доступными приложениям. Примером таких устройств являются: VGA, IDE контроллеры, мат плата, последовательные и параллельные порты. В наших системах имеется 2 адресных простарнства: адресное простарнство оперативной памяти и портов ввода/вывода. Порт – это адрес. Точно также как минимальной адресуемой ядиницей памяти является байт, порта - тоже байт. Внешние устройства используют 2 схемы: memory mapping (адресация устройства выполняется аналогично адресации памяти, т.е. используются команды работы с оперативной памяти. К таким устройствам относят видеокарты), input/output mapping (внешние устройства подключаются к системе через порты ввода ввода. Две команды in/out). драйверы, реализованные как загружаемые модули ядра Файлы модуля ядра располагаются в */lib*/*modules*. Обычно, при инсталяции системы задается список модулей, которые будут автоматически подключаться на этапе загрузки. Список загружаемых модулей хранится в файле */etc*/*modules* код драйверов этого типа поделен между ядром и специальной утилитой

Например в драйвере принтера ядро отвечает за взаимодействие с параллельным

портом, а формирование управляющих сигналов осуществляет демон печати lpd, который использует для этого специальную программу фильтр. Еще пример драйвера – драйвер модема. В линукс файлы устройств идентифицируются двумя положительными числами: а) major device number б) minor device number major device number - старший номер устройства. Индентфиицирует тип устройства (чаще говорят драйвер устрйоства). Например звуковая плата, жесткий диск. Текущий список старших номеров устрйоств можно найти в файле /*usr*/*include*/*linux*/*major*.h старший номер тип устройства 1 оперативная паямть 2 гиб диск 13 мышь 14 звуковая карта Файлы устройств одного типа имеют одинаковые номера и отличаются по номеру, который добавляется в конец имени. Все файлы сетевых плат имеют имена Ethernet->eth: eth0, eth1,. . . minor device number - идентифицируют конкретные устройства в ряду устройств с одим и тем же старшим номером. Если перейти в каталог /dev и выполнить команду *lsl* то в место размера файла в байтах мы увиди два числа, разделенных запятой (старший и младший номера). Листинг 3.1 — выведет файлы символьных устрйоств 1 # l s – l / dev / | grep "^ c" Листинг 3.2 — выведется информация о старших номерах устройств, известных ядру 1 # cat / proc / d e v i c e s
51 В *linux*/*types*.h определен тип dev_t который содержит старшие и младшие номера. Размер соответствующего значения 32бит: 12бит – старший номер, 20бит – младший. В linux/kdec_t.h определены два макроса: MAJOR(dev_t dev) и MINOR(dev_t dev). Если известен старший и младший номера устрйоств, и нужно получить dev_t, то для этого используется макрос MKDEV(int major, int minor) Листинг 3.3 — Печать старшего и младшего номера устройства 1 s tat ic dev_t f i r s t ; 2 pr intk (KERN_INFO "<major , minor> : <%d,%d>/n" , MAJOR(f i r s t) , MINOR(f i r s t)) ;

14. Unix: команды - fork(); wait(); exec(); pipe(); signal().

Unix создавалась как ОС разделения времени. Базовое понятие Unix – процесс (единица декомпозиции ОС, программа времени выполнения). Процесс рассматривается как виртуальная машина с собств. адресным пространством, выполн. пользов. прогр., предоставл. набор услуг.

Когда выполняется системный вызов **fork** создаётся ещё один процесс, называемый потомком. Для процесса потомка создаются собственные карты трансляции адресов (таблицы страниц), но они ссылаются на адресное пространство процесса-предка (на страницы предка). При этом для страниц адресного пространства предка права доступа меняются на only- read и устанавливается флаг – copy-on-write. Если или предок или потомок попытаются изменить страницу, возникнет исключение по правам доступа. Выполняя это исключение супервизор обнаружит флаг copy-on-write и создаст копию страницы в адресном пространстве того процесса, который пытался ее изменить. Таким образом код процесса-предка не копируется полностью, а создаются только копии страниц, которые редактируются. После перехода на новое адресное пространство предку возвращаются права доступа read-write для сегментов данных и стека.

Чаще всего нет смысла в выполнении двух одинаковых процессов и потомок сразу выполняет системный вызов **exec()**, параметрами которого является имя исполняемого файла и, если нужно, параметры, которые будут переданы этой программе. Говорят, что системный вызов exec() создает низкоуровневый процесс: создаются таблицы страниц для адресного пространства программы, указанной в exec(), но программа на выполнение не запускается, так как это не полноценный процесс, имеющий идентификатор и дескриптор. Системный вызов exec() создает таблицу страниц для адресного пространства программы, переданной ему в качестве параметра, а затем заменяет старый адрес новой таблицы страниц.

Системный вызов **wait(&status)** вызывается в теле предка. Предок б. ждать завершения всех своих потомков. При их завершении он получит статус завершения. Все процессы в Unix объединены в группы, процессы одной группы получают одни и те же сигналы (события). Т.к. Unix система разделения времени, то существует понятие терминала. Процесс, запустивший терминал имеет PID = 1. Все процессы, запущенные на этом терминале, являются его потомками.

Системный вызов **pipe()** создает неименованный программный канал. Неименованные программные каналы могут использоваться для обмена сообщениями между процессами родственниками. В отличие от именованных программных каналов неименованные не имеют идентификатора, но имеют дескриптор.

Труба (**pipe**) –это специальный буфер, который создается в системной области памяти. Информация в канал записывается по принципу FIFO и не модифицируется.

Существуют программные каналы двух типов: именованные и не именованные. Поддерживаются, как специальные файлы. Располагаются в системной области

памяти, т.к. адресные пространства процессов защищены и недоступны для других процессов.

Труба буферизуется на трех уровнях:

а) В системной области памяти.

б) В файловой системе помечается буквой p

в) При переполнении системной памяти, буферы, имеющие большое время существования, переписываются на диск, при этом используются стандартные функции работы с файлами. Если процесс пытается записать больше 4096 байт, то труба буферизуется во времени.

Процессы могут порождать, принимать и обрабатывать сигналы. М.б. синхронные и асинхронные. Средства посылки сигнала - 2 системных вызова kill() и signal(). Механизм сигналов позволяет реагировать на события внутри и вне процесса. Как правило полученный процессом сигнал означает призыв завершиться. Реакция процесса на сигнал зависит от того, как сам процесс определил свое поведение на конкретный сигнал (сигнальная маска). Системный вызов сигнал возвращает указатель на предыдущий обработчик сигнала, т. е. его можно использовать для восстановления обработчика сигнала:

```
#include <signal.h>
int main()
{
        void (*old_handler) (int) = signal (SIGINT, SIG_IGN);
        signal(SIGINT, old_handler);
}
```

signal() не является стандартным для Posix1, но он определен в ANSI C, поэтому имеется во всех UNIX системах. Не рекомендуется использовать signal() в переносимых приложениях.

15. Терминал.управляющий терминал.терминальный ввод-вывод. Псевдотерминал. назначение псевдотерминала.как взаимодействуют процессы в пседотерминале кажется. доп вопросы:как представлен псведотерминал в системе,что эмулирует ведущее устройство, зачем нужны псевдотерминалы

Все процессы в Unix объединены в группы, процессы одной группы получают одни и те же сигналы (события). Т.к. Unix система разделения времени, то существует понятие терминала. Процесс, запустивший терминал имеет PID = 1. Все процессы, запущенные на этом терминале, являются его потомками.

Unix поддерживает понятие терминал программно. И Unix и Windows являются системами (реального ???) времени, это когда к компу подключены терминалы.

Мы можем запустить любое количество терминалов. В Windows это консоль. Когда запускаем систему, то запускается процесс init с id = 0. Когда открываем терминал, то создается процесс ???. Терминальный процесс (id = 1) является предком всех процессов, запущенных в данном терминале.

У многих UNIX-подобных систем, включая Linux, есть средство, именуемое псевдотерминалом. Это устройства, очень похожие на терминалы, за исключением того, что у них нет связанного с ними оборудования. Они могут применяться для предоставления терминалоподобного интерфейса другим программам.

Например, с помощью псевдотерминалов возможно создание двух программ, играющих друг с другом в шахматы, не взирая на тот факт, что сами по себе программы проектировались для взаимодействия посредством терминала с человеком-игроком. Приложение, действующее как посредник, передает ходы одной программы другой и обратно. Оно применяет псевдотерминалы, чтобы обмануть программы и заставить их вести себя нормально при отсутствии терминала.

Одно время реализация псевдотерминалов сильно зависела от конкретной системы. Сейчас они включены в стандарт Single UNIX Specification.

Псевдо терминалы не имеют уникального физического разъема на компьютере.

Они используются для эмуляции последовательного порта. Например, если кто-то соединяется через telnet с вашим компьютером по сети, они могут подсоединиться к устройству /dev/ptyp2 (порт псевдотерминала). В XWINDOWS, программа эмуляции терминала xterm использует псевдотерминалы.

Также его используют программы любительского радио. При использовании некоторого прикладного программного обеспечения можно иметь 2 или более псевдотерминала, присоединенных на один и тот же физический последовательный порт.

Псевдотерминалы образуют пары типа ttyp3 и ptyp3. pty... - это хозяин или управляющий терминал, а tty... - подчиненный. ttyq5 - также псевдотерминал, как - ttysc (c - шестнадцатеричная цифра).

Более точно, главные псевдотерминалы - /dev/pty[p-s]n, а соответствующие подчиненные - /dev/tty[p-s]n, где n - шестнадцатеричная цифра.

16. Организация монопольного доступа с помощью команды test-and-set, алгоритм Деккера.

Монопольный доступ осуществляется взаимоисключением, т.е. процесс, получивший доступ к разделяемой переменной, исключает доступ к ней др. процессов.
Аппаратная реализация взаимоисключения (test_and_set). Впервые test_and_set была введена в OS360 для IBM 370. Эта команда является машинной и неделимой, т.е. ее нельзя прервать. Она одновременно производит проверку и установку ячейки памяти, называемой ячейкой блокировки: читает значение лог. переменной B, копирует его в A, а затем устанавливает для B значение True. test_and_set(a, b): a = b; b = true;
В Windows это называется спин-блокировкой. /* спин-блокировкой по-Рязановой, наз. проверка флага в цикле */

flag, a1, a2: boolean;
P1: a1 = 1;
while(a1 == 1)
test_and_set(a1, flag);
CR1;
flag = 0;
PR1;
	P2: a2 = 1;
	while(a2 == 1)
	test_and_set(a2, flag);
	CR2;
	flag = 0;
	PR2;

Flag = true, когда один из процессов – в своем критическом участке. main:
Теоретически не исключено бесконечное откладывание, но flag = 0; вероятность 0 (т.к. test_and_set – машинная неделимая команда parbegin и выполняется очень быстро. P1;
Бесконечное откладывание – ситуация, когда разделённый ресурс P2; снова захватывается тем же процессом. parend; Программная реализация (алгоритм Деккера).
Деккер – голландский математик. Предложил способ свободный от бесконечного откладывания.

flag1, flag2: boolean;
queue: integer;
p1: flag1 = 1;
while(flag2)do
if(queue == 2) then
begin
flag1 = 0;
while(queue == 2)do;
flag1 = 1;
end;
CR1;
flag1 = 0;
queue = 2;
	p2: flag2 = 1;
	while(flag1)do
	if(queue == 1) then
	begin
	flag2 = 0;
	while(queue == 1)do;
	flag2 = 1;
	end;
	CR2;
	flag2 = 0;
	queue = 1;
	end P2;
		main: flag1 = 0;
		flag2 = 0;
		queue = 1; // Или 2
		parbegin
		P1;
		P2;
		parend;

end P1;		

queue – очередь процесса входить в критическую секцию.

Недостаток обоих методов – активное ожидание на процессоре.

Активное ожидание – ситуация, когда процесс занимает процессорное время, проверяя значение флага.

Активное ожидание на процессоре является неэффективным использованием процессорного времени.

17. Сокеты - определение, виды сокетов. DGRAM сокеты. Примеры(программирование). Сетевые сокеты, сетевой стек, порядок байтов.

Абстракция сокетов введена в BSD (Berkeley Software Distribution) Unix. Сокеты создавались для организации взаимодействия процессов, причем безразлично, где они работают, на одной машине или на разных. Т.е. в распределенной системе, единообразно осуществляются взаимодействтие, будто менйфрейм или распределенная система. Сокеты представляют абстракцию конечной точки взаимодействия. Процессы могут использовать единообразный интерфейс сокетов для отправки/по лучения сообщений (данных) как на локальной машине так и по сети, используя разные протоколы. Существует обобщенная иллюстрация, которая подчеркивает тот факт, что осуществляются соответствующие системные вызовы.

```
#include <sis/types.h>
 #include  <sys/socket.h>
int socket ( int family, int type , int  protocol ) ;
```

family − семейство домен( ) − это пространство имен , для него определены следующие константы :

AF_UNIX − межпроцессное взаимодейсвтие на локальном компе

AF_INET − семейство протоколов TCP/IP на основе интернета v4

AF_INET6 − IPv6 13 AF_IPX − семейство протоколов IPX

AF_UNSPEC − неопределенный домен

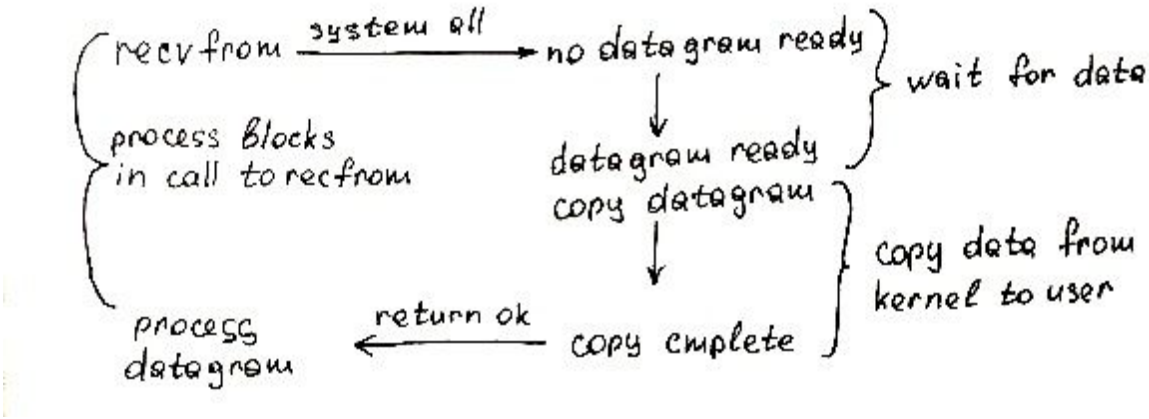
Type: Внутри семейства протоколов ТСПИП/ различают 3 типа :

SOCK_STREAM − потоковые сокеты , определяет ориентированное на потоки , надежное упорядоченное полнодуплексное− логическое соединение между двумя сокетами .

SOCK_DGRAM − определяет ненадежную службу без установления логического соединения , когда пакеты могут передаваться без сохранения порядка , так называемое широковещательная передача данных .

SOCK_RAW − сырой , прямой сокет , низкоуровневый интерфейс .

Протокол выбирается для определенного типа сокетов , если досутпно несколько протоколов для указанного типа . В AF_INET SOCK_STREAM всегда выбирается TCP. ДЛЯ SOCK_DGRAM всегда использует протокол UDP. Обычно аргумент протокол устанавливается в нулевое значение , В результате протокол , для данного типа сокета и пространства имен будет выбран по умолчанию .



PF_PACKET - создано для непосредственного доступа приложений к сетевым ???.
NETLINK_## - используются для обмена между пространством ядра и пространством пользователя. Ядро линукс предоставляет для работы с сокетами один единственный системый вызов в *net*/*socket*.

Данная функция sys_socket_call() реализована как переключатель системных вызовов.
call - типа int определяет номер нужной функции.

```
asmlinkage long sys_socket_call (int call, unsigned long * args){
        int err;
        . . .
        if copy_from_user (a, args,  nargs[call] ){
                return − EFAULT;
        }
        switch(call){
                 case SYS_SOCKET: err = sys_socket(a0, a1, a2); break ;
                // . . .
        }
        return err;
}
```

Если функции sys_socket_call передано параметр call = 1, то будет осуществлен вызов sys_socket (эта функция определена в файле *net*/*socket.c* и в ней вызывается функция sock_create) инициализирует структуру сокета struct_socket

```
#include <linux/net.h>
#defineSYS_SOCKET 1
#define SYS_BIND 2
#define SYS_CONNECT 3
#def ine SYS_LISTEN 4
#def ine SYS_ACCEPT 5
```

Структура сокета:

```
struct socket{
        socket_statestate;
        short type;
    //для синхронизации доступа
        insigned long flags;
   //ссылает на дейсвтие подулюченного протокола
        const struct proto_ops * ops;
   //спиокс асинхронного запуска
        struct fasync_struct * fasync_list;
        struct file* file;
        struct sock * sk;
        wait_queue_head_t wait;
};
```

Сокеты BSD следуют концепции UNIX. Все объекты с доступом чтения/записи отображены как файлы, чтобы можно было работать стандартными функциями. Объекты, которыми манипулируют при операции чтения/записи в контексте транспортных протоколов являются конечные точки коммуникационных отношений (сокеты). Структура сокет содержит указатель на структуру *file*.

Для сокета определено 5 состояний:

а) SS_FREE - не занят

б) SS_UNCONNECTED - не соединен

в) SS_CONNECTING - соединяется в данный момент

г) SS_CONNECTED - соединен

д) SS_DISCONNECTING - отсоединяется в данный момент

Сокет является специальным файлом. Поэтому есть поле struct file и оно ссылается на inode.

Адреса сокетов Сокеты поддерживают множество протоколов, поэтому введена общая структура *sockaddr*. При создании коммуникационных отношений необходимо указать адрес конечной точки коммуникационного партнера.

```
struct sockaddr{
        //определяет семейство адресов
        sa_family_t sa_family;
        char sa_data [14]
}
```

Формат адреса подробно не определен, поэтому для адресов интернета используется другая структура sock_addr_in:

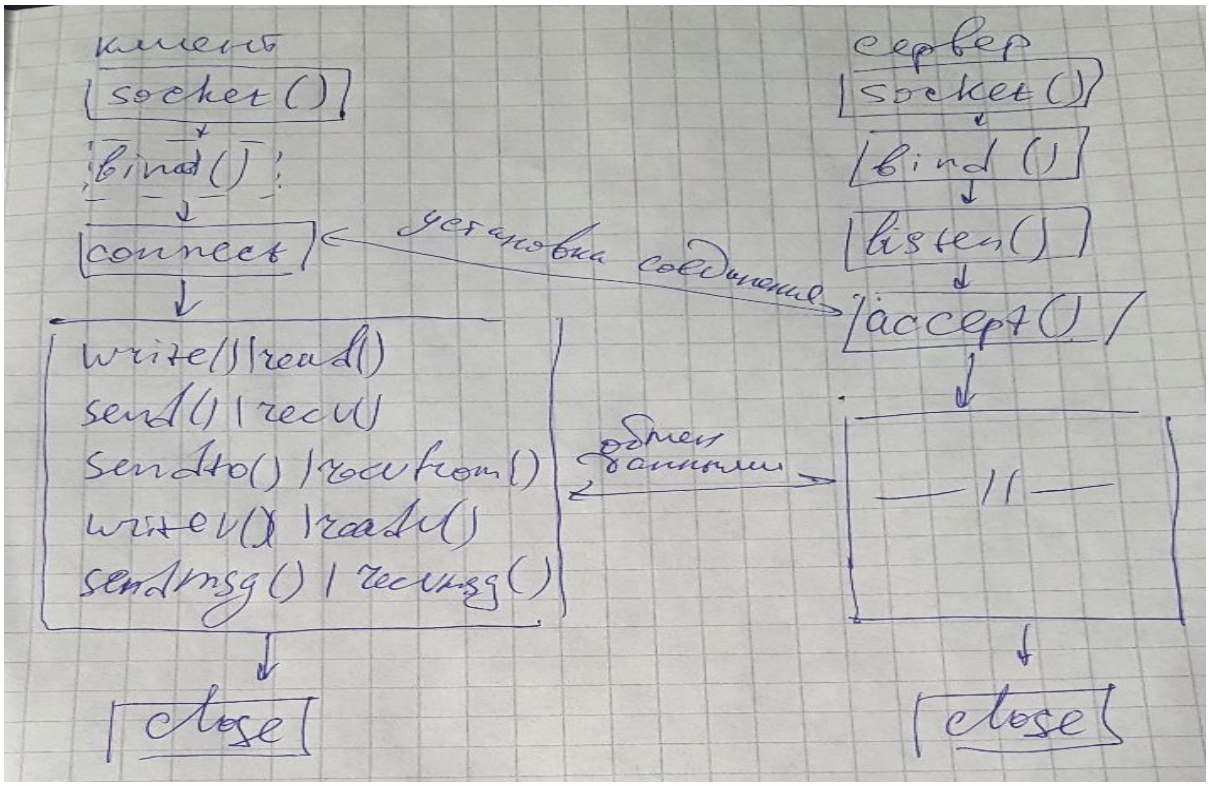
```
#include </usr/include/netinet.in.h>
struct sockaddr_in {
        sa_family_t sin_family;
        unsigned short int sin_port;
```

```
        struct in_addr sin_addr;
        unsigned char sin_zero[sizeof(struct sockadd r) − sizeof(sa_family_t) −
                sizeof(uint16_t) − sizeof(struct in_addr)];
}
```

Адреса и номера портов должны быть указаны в сетевом порядке байтов. В сети и на компьютере может быть разный порядок байтов. Сокеты в файловом пространстве имен. Это сокеты семейства UNIX. Тип DGRAM. Тут взаимодейсвтие осуществляется на локальной машине через файловое пространство имен. Создаем файл
//связываем с именем сокета.
strcpy(srvr_name, sa_data, "socket.soc");

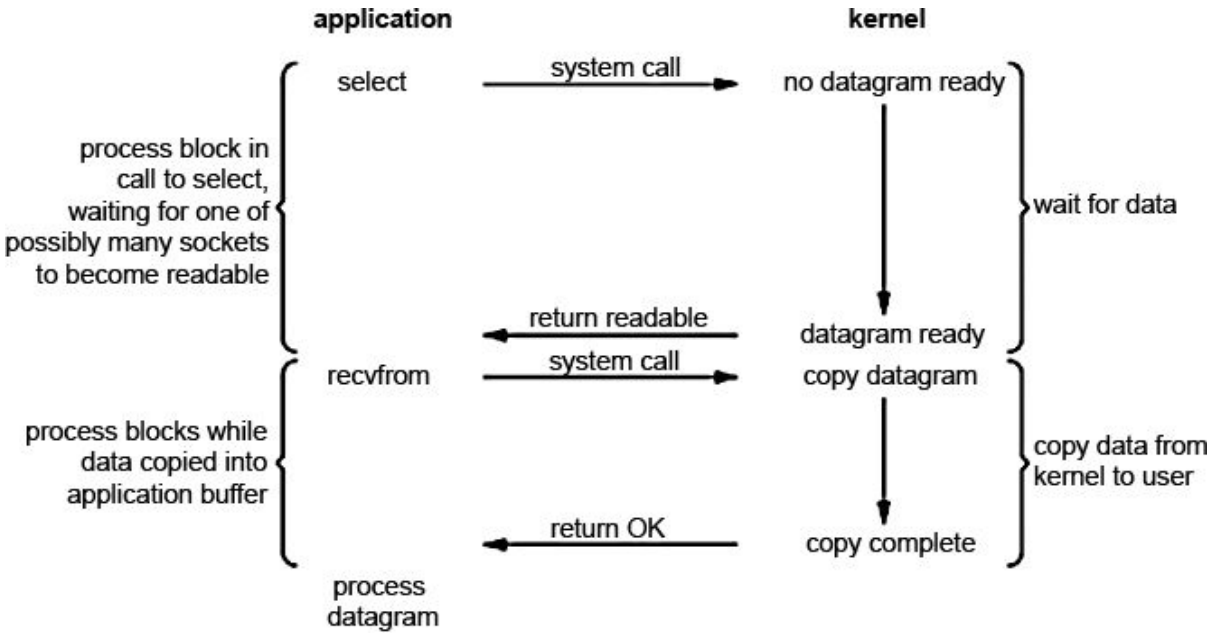
Для семейства AF_UNIX может использоваться только SOCK_DGRAM. Для инер нет домена могут использоваться SOCK_STREAM и SOCK_DGRAM. Для SOCK_RAW исполь зуются протоколы IP и ICMP и т.д.

Сетевые сокеты, которые мы рассматривали AF_INET и SOCK_STREAM. Для них существует сетевой протокол. Взаимодейсвтие процессов через сокеты идет по модели клиент-сервер, кроме парных сокетов, где по модели предок-потомок.



Пунктир - не обязательно.
bind() - для назначения сокету локального адреса. Для сокетов интернета этот ад рес состоит из ip адреса сетевого интерфейса локальной системы и номера порта. Клиенты могу действовать без вызова bind так как их точный адрес не играет ни какой роли и в этом случае адрес назначается автоматически.
listen() используется сервером для информирования операционной системы о том

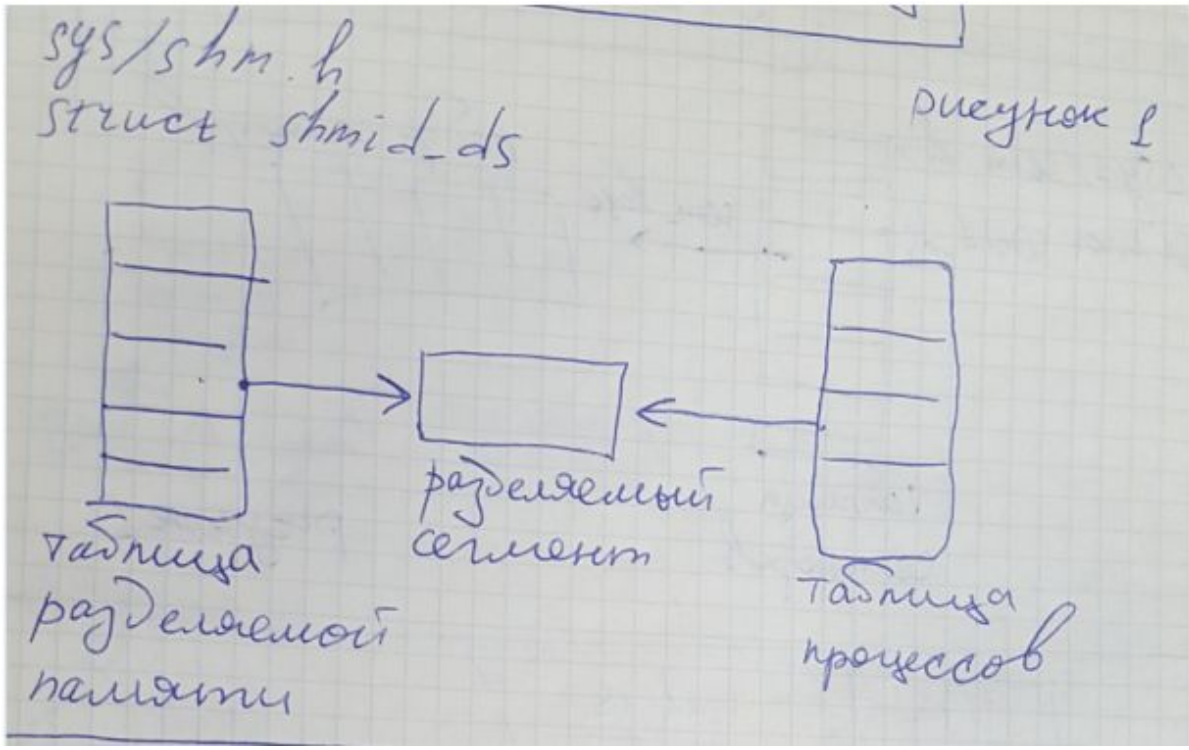
что в сокете нужно принять соединение. Очевидно это имеет смысл ориентирован ных на соединение (в настоящее время это только TCP)
connect() - устанавливает соединение, например TCP по переданному адресу. Для протоколов без установления соединения (UDP) может использоваться для указания адреса назначения для всех передаваемых пакетов.
accept() - используется сервером для принятия соединения при условии, что он ра нее получил запрос на соединение. В противном случае вызов будет заблокрирован до тех пор, пока не поступит запрос соединения. Когда соединение принимается, со кет копируется, а именно первоначальный сокет остается в состоянии listen, а новый сокет в состянии connected. Вызов accept возвращает новый дескриптор файла для следующего сокета. Такое дублирование сокетов в ходе принятия соединения дает серверу возможность продолжить принимать соединения без необходимости предва риательно закрывать предыдущие соединения.
Схематично процесс мультиплексирования представлен на изображении:



18. Unix: разделяемая память (shmget(), shmat()) и семафоры (struct sem, semget(), semop()). Пример использования

Разделяемая память системное средство взаимодействия процессов, позволяет множеству процессов отображать системное адресной пространство выделенное под разделяемый сегмент на своё адресное пространство, т.е. созданный разделяемый сегмент подключается с помощью указателя к виртуальному адресному пространству процесса

РП была реализована как средство повышения производительнсоти при обработки сообщений, как универсальное средство в силу того, что разделяемый сегмент подключается к адресному пространству процесса, при записи сообщения в разделяемый сегмент и при чтении – копирование не выполняется. Так данное средство позволяет осуществить обмен сообщениями быстро, но РП не имеет средств взаимоисключения. Поэтому РП используются совместно с семафорами. В адресном пространстве ядра имеется таблица разделяемых сегментов (памяти).



```c
int *buf;
if (shmid = shmget(IPC_PRIVATE,
    (MAX_SLOTS + 2) * sizeof(int), 0600) == -1)
{
    return fprintf(stderr, "shmget");
}
buf = (int *) shmat(shmid, NULL, 0);
if (buf == (int *) -1) {
    perror("shmat");
    exit(1);
}
```

int shmget(key_t key, int size, int shmflg);

Функция shmget преобразовывает ключ сегмента разделяемой памяти key в идентификатор сегмента разделяемой памяти. Если в параметре shmflg указан флаг IPC_CREAT, создаётся новый сегмент размера size с правами доступа, определёнными в 9 младших битах параметра shmflg. Если дополнительно был указан флаг IPC_EXCL, и сегмент с заданным ключом уже существует, функция shmget завершается с ошибкой EEXIST.

int shmctl(int shmid, int cmd, struct shmid_ds *buf);

Аргумент shmid — идентификатор сегмента разделяемой памяти, полученный с помощью shmget. Если указать в аргументе cmd значение IPC_RMID, а аргументом buf передать NULL, сегмент разделяемой памяти будет удалён.

void *shmat(int shmid, const void *shmaddr, int shmflg);

Функция shmat подключает сегмент разделяемой памяти с идентификатором shmid к адресному пространству процесса.

int shmdt(const void *shmaddr);

Функция shmdt отсоединяет сегмент разделяемой памяти с начальным адресом shmaddr от адресного пространства процесса.

В системе есть ограничения на создание РП.

- SHMMNI Максимальное число разделяемых сегментов, которые могут существовать в системе одновременно
- SHMMIN Минимальный размер разделяемой области памяти в байтах
- SHMMAX Максимальный размер разделяемой области памяти в байтах

Семафоры

int semget(key_t key, int nsems, int semflg);

Функция semget позволяет связать с массивом семафоров с ключом key идентификатор семафора. Этот идентификатор должен использоваться при всех обращения к функциям работы с этим массивом семафоров. Параметр nsems задаёт количество

семафоров в мас- сиве семафоров. Он может быть равен 0, если процесс не пытается создать новый массив семафоров. Параметр semflg определяет флаги создания массива семафоров и права до- ступа к создаваемому массиву семафоров.
IPC_CREATЕсли этот флаг установлен, массив семафоров будет создан.
IPC_EXCL Если установлен этот флаг, и установлен флаг IPC_EXCL, выполнение завершится с ошибкой, если массив семафоров с таким ключом уже существует

int semctl(int semid, int semnum, int cmd, ...);

Аргумент semid определяет идентификатор массива семафоров, над которым следует про- извести операцию. Аргумент semnum задаёт номер семафора в массиве, аргумент cmd задаёт выполняемую команду

int semop(int semid, struct sembuf *sops, unsigned nsops);

Функция позволяет задавать сразу несколько операций над массивом семафоров semid. Операции должны находится в массиве, адрес которого передаётся в аргументе sops. В ар- гументе nsops передаётся количество операций в массиве операций. Каждая операция над семафором задаётся структурой struct sembuf.

```
struct sembuf {
        short sem_num; /* номер семафора (0 - первый) */
        short sem_op; /* операция над семафором */
        short sem_flg; /* флаги операции */
};
```

Поле sem_flg может содержать флаг IPC_NOWAIT, указывающий, что эта операция не должна приводить к блокированию выполнения процесса, а вместо этого функция semop должна завершиться с ошибкой с кодом EAGAIN.
Поле sem_num задаёт номер семафора в массиве семафоров (номер отсчитывается, как обычно, от 0), над которым нужно выполнить операцию. Поле sem_op задаёт операцию:

Если значение sem_op > 0, операция добавляет это значение к значению семафора. Операция не может привести к блокированию процесса. Процесс должен иметь права на запись в массив семафоров.

Если значение sem_op равно 0, операция проверяет значение семафора. Если значение семафора равно 0, операция завершается успешно, и проверяется следующая операция в массиве операций. Если значение семафора не равно 0, и в поле sem_flg указан флаг IPC_NOWAIT, вызов semop завершается с ошибкой EAGAIN, и ни одна из запрошенных операций не выполняется. Если значение семафора не равно 0, и в поле sem_flg не указан флаг IPC_NOWAIT, процесс будет приостановлен до наступления одного из следующих событий: – Значение семафора станет равно 0. – Массив семафоров будет удалён. В этом случае semop завершится с ошибкой EIDRM. – Процесс получит сигнал, обработчик которого вернёт управление процессу. В этом случае semop завершится с ошибкой EINTR.

Если значение sem_op отрицательно, значение семафора уменьшается на заданную величину. Если текущее значение семафора больше, чем величина -sem_op, значение семафора уменьшается на -sem_op.

19. Аппаратные прерывания в Linux: нижние и верхние половины и особенности работы softirq и tasklet в SMP-системах

Верхняя половина - это именно та функция, которая будет вызываться в первую очередь при каждом возникновении аппаратного прерывания. Но это вовсе не означает, что при возврате из этой функции работа по обработке текущего прерывания будет завершена (хотя и такой вариант вполне допустим). Из-за этой «неполноты» такой обработчик и получил название «верхняя половина» обработчика прерывания. Дальнейшие действия по обработке могут быть запланированы эти обработчиком на более позднее время, используя несколько различных механизмов, обобщённо называемых «нижняя половина».

Важно то, что код обработчика верхней половины выполняется при запрещённых последующих прерываниях по линии irq (этой же линии) для того локального процессора, на котором этот код выполняется. А после возврата из этой функции локальные прерывания будут вновь разрешены.

Аппаратное прерывание (IRQ) — это внешнее асинхронное событие, которое поступает от аппаратуры, приостанавливает ход программы и передает управление процессору для обработки этого события. Обработка аппаратного прерывания происходит следующим образом:

- Приостанавливается текущий поток управления, сохраняется контекстная информация для возврата в поток.
- Выполняется функция-обработчик (ISR) в контексте отключенных аппаратных прерываний. Обработчик должен выполнить действия, необходимые для данного прерывания.
- Оборудованию сообщается, что прерывание обработано. Теперь оно сможет генерировать новые прерывания.
- Восстанавливается контекст для выхода из прерывания.

```
#define IRQF_TIMER (IRQF_TIMER | IRQF_NO_SUSPEND | IRQF_NO_THREAD)
// IRQF_NO_SUSPEND − не гарантируется , что это прерывание будет будить системы , не отключать этот запрос прерывания при остановке
// IRQF_NO_THREAD − прерывание не может быть запущено в виде потока
```

3 механизма реализации отложенного дейсвтия необходимы для за вершения обработки прерывания.

а) softirq (гибкие);
б) тасклеты;
в) очереди работ (work queue).

Soft irq определяются статически во время компиляции ядра. Описываются структурой SOFT_IRQ_ACTION. Эта структура представляет один вход.

```
// тут объявлен статический массив
// static struct softirq_action softirq_vec[32]
// те . . таких обработчиков не может быть больше 32
#include <kernel/softirq.c>

struct softirq_action{
        // выполняемая функция
        void (* action) (struct * softirq_action *);
```

```
        // данные для функции
        void * data ;
    }
```

Обработчики Softirq

Индекс	приоритет	комментарий
HI_SOFTIRQ	0	высокоприоритетный тасклет
TIMER_SOFTIRQ	1	таймеры
NET_TX_SOFTIRQ	2	отправка сетевых пакетов
NET_RX_SOFTIRQ	3	прием сетевых пакетов
SCSI_SOFTIRQ	4	блочные устройства подситемы SCSI
TASKLET_SOFTIRQ	5	тасклеты с обычным приоритетом

Чтобы заработал обработчик SOFTIRQ его нужно зарегистрировать. Реги страция происходит с помощью функции OPEN_SOFTIRQ3 парамета):

а) индекс (один из таблицы)

б) функция обработчик

в) значения поля дата

Пример: open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);. Чтобы обработчик обозначить в системе как ожидающий обработки, необхо дима функция raise_softirq(...)

```
static irqreturn_txxx_interrup t (int irq, void *dev_id, struct pt_regs * regs){
        // . . .
        raise_softirq(xxx_SOFTIRQ);
return IRQ_HANDLED;
}
```

do_softirq() // вызывает в своем теле функцию do_softirq() которая реализована просто. Основной цикл этой функции : в системе есть очередь отложенных прерываний, проверяет их все и вызывает ожидающие обработчики .

```
do{
        if ( pending &???) {
                h −> action(h) ;
        }
        h++;
        pending >>= 1;
} while ( pending ) ;
```

В системе имеется демон *KSOFTIRQD* предназначенный для выполнения отложенных действий. При этом обработка отложенных действий типа *SOFTIRQ* и тасклетов осуществляется с помощью набора потоков ядра по одному на каждый процессор. Если набрать #ps −Ahf | head −n12 , то можно наблюдать

UID	PID	PPID	LWP	...	TTY

root … root	1 … 1	0 … 2	1 … 4	...	//sbin/init … [KSOFTIRQ/0]

SOFTIRQ не могут переходить в состояние sleep. Если SOFTIRQ выполняется на процессоре, то его может вытеснить только аппаратное прерывание. Одно и тоже IRQ может параллельно выполняться на разных процессорах. Код должен быть реентабельным, следовательно код меньше простаивает в очереди.

Тасклеты

Тасклет - частный случай SOFTIRQ, реализуется на базе SOFTIRQ. Таскле ты - отложенные прерывания, для которых установлено следующее правило: один и тот же тасклет не может выполняться несколькими процессорами. Тасклеты разных типов могут параллельно выполняться.

Тасклет - компромисс простоты использования и быстроты. Статические макросы создаются в виде макросов в файле *linux*/*interrupt*.h:

а) DECLARE_TASKLET(name, func, data); - создает тасклетЮ у которого счет чик ссылок равно 0.

б) DECLARE_TASKLET_DISABLED(name, func, data); - поле ссылок равно 1 и запрещен

Могут быть созданы динамически с помощью функции TASKLET_INIT

```
struct tasklet * taskl;
tasklet_init(taskl, taskl_handler, data);
```

Чтобы запланировать выполнение тасклета вызывается либо tasklet_shedule() либо tasklet_hi_shedule() и передается 1 параметр на структурау tasklet_struct.

```
struct tasklet_struct{
        struct tasklet_struct * next;

        // текущее состояние
        unsigned long state;

        // счетчик ссылок
        atomic_t count;

        void (* func) (unsigned long);
        unsigned long data;
};
```

Поле *state* может иметь 1 из 2 состояний: TASKLET_STATE_SHED и TASKLET_STATE_RUN. Запланированные на выполнение тасклеты хранятся в связных списках: TASKLET_VEC и TASKLET_HI_VEC.

После планирования тасклет будет запущен 1 раз, даже если добавлен несколько раз, в отличие от *SOF T IRQ*. Для отключения тасклета используются:

а) tasklet_disable() не сможет отменить тасклет, который уже выполняется;

б) tasklet_disable_nosync() сможет.

void tasklet_handler(unsigned long data) - функция обработки тасклета.

Особенностью тасклетов является то, что они не могут блокироваться, по этому в них нельзя использовать семафоры. Если в тасклете используются общие данные с обработчиком прерывания или другим тасклетом, то нужно использовать spin блокировки.

20. Взаимодействие процессов: монопольное использование - программная реализация взаимоисключения, взаимоисключение с помощью семафоров; сравнение - достоинства и недостатки. Мониторы - определение, пример: монитор "кольцевой буфер".

Чтобы не терять значение разделяемой переменной, необходимо обеспечить монопольный доступ (когда процесс получает доступ к критической секции (КС) по разделяемой переменной, то системой больше ни какой процесс не сможет получить доступ к этой же КС.) Монопольльный доступ обеспечивается методами взаимоисключения, делятся на:

а) программные
б) аппаратные
в) с помощью семафоров
г) мониторы.

В общем случае задача взаимоисключения решается для N процессов.

Программная реализация

С использованием флага, который проверяется в цикле.

```
f1, f2 : boolean;
P1:  ….
  while(f2)
    f1 = 1;
    CR1;
    f1 = 0;
```

```
P2: …
  while(f1)
    f2 = 1;
    CR2;
    f2 = 0;
    ...
```

```
begin
  f1 = 0;
  f2 = 0;
  parbegin
     P1; P2;
  end.
```

В второй процесс мог не успеть установить флаг, тогда первый процесс также пройдет цикл while, следовательно сново получаем одновременный доступ.

```
P1: ...
        f1=1;
        while(f2)
                CR1;
                f1 = 0;
```

```
P2: ...
        f2 = 1;
        while(f1)
                CR2;
                f2 = 0;
```

Зацикливание при проверке флага на время всего кванта. Ситуация dead lock (блокировка друг друга)

Блокировка на семафоре

Семафор (0, 1) - бинарный.

Операция P(S) - декремент. Процесс, вызвавший операцию P(S) для S = 0 будет заблокирован на семафоре.

Операция V (S) - инкремент, т.е. процесс разблокирует другой процесс, если было S = 0. Заблокированный процес не выполняет цикл активного ожидания, т.е. сема- форы Дейкстры решили проблему активного ожидания. Но заблокировать/разблоки- ровать процесс может только ядро. Вызов команд P(S) и V (S) приводит к переводу системы в режим ядра. Считающий семафор - принимает целые неотрицательные значения.

//P1: ...
P(S);
CR1;
V(S);
	//P2: . . .
	P(S);
	CR2;
	V(S);

Осуществляется переход в режим ядра при захвате и освобождении семафора, т.е. происходит переключение контекста. Считающие семафоры могут принимать не отрицательные значения. Современные ОС поддерживают наборы считающих семафоров. По своей семантике наборы семафоров похожи на массивы (массивы семафоров). У набора семафоров есть важное свойство: одной неделимой операцией можно изменить значения всех семафоров или части семафоров. Если процесс использует большое кол-во разных семафоров то сложно уследить за состоянием семафоров, то такие программы могли висеть в тупиках. Другие средства взаимоисключения: мьютекс.

Между семафорами и мьютексами есть серьезные различия:

а) Мьютекс имеет хозяина, только процесс, захвативший мьютекс, может его освободить. Семафор может освободить любой другой процесс, зная идентификатор.

б) Чтобы обеспечить для мьютекса свойство владельца, если мьютекс пытается захватить более приоритетный процесс, то временно повышается приоритет процесса хозяина.

в) Мьютекс не может быть случайно удален, т.е. если он захвачен кем то, то удалить процесс–хозяин – нельзя.

Мониторы

Цель монитора – структурировать средства взаимоисключения.

Идея – создание механизма, который бы соответствующим образом унифицировал взаимодействие параллельных процессов. Монитор представляет синтаксическую конструкцию. Монитор содержит данные и процедуры, которые могут изменять эти данные. Говорят, что монитор защищает свои данные, так как изменить данные монитора можно только с помощью процедур монитора. Монитор может предоставляться языком, а может быть системным средством. Монитор гарантирует, что в каждый момент времени процедуры монитора могут использовать только один процесс. Если процесс успешно вызвал процедуру монитора, то говорят, что такой процесс находится в мониторе, остальные процессы ставятся к монитору в очередь. Классические мониторы используют 2 системных вызова wait и signal, эти системные вызовы определены на переменной типа «условие»

Монитор кольцевой буфер

На нем реализована задача производства-потребления.

1) Синхронизания процессов - какие-то действия процессы должны делать вместе, т.е. один подождет другого, чтобы сработать вместе
2) Взаимоисключение процессов - если один процесс в критицеской ситуации, то больше никто не зайдет - нет никакого ожидания

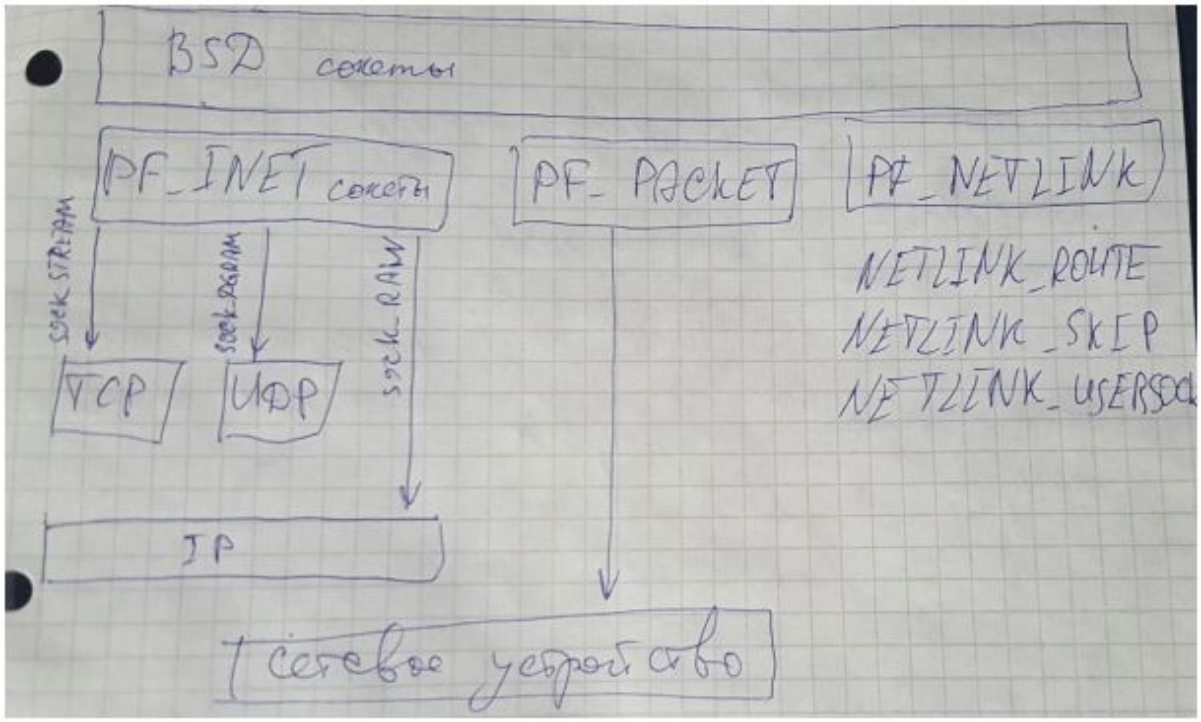
```
RESOURSE: MONITOR
var
    bcircle: array[1..n] of <type>;
    pos: 0..n;
    wp: 0..n-1; //записать в позицию
    rp: 0..n-1;//чтение из pos
    buf_empty, buf_full: conditional;

procedure producer(data: <type>);
begin
    if (pos) then
        wait(buf_empty);
    bcircle[wp] := data;
    pos := pos + 1;
    wp := (wp + 1) mod n;
    signal(buf_full);
end;

procedure consumer(var data: <type>);
begin
    if (pos = 0) then
        wait(buf_full);
    data := bcircle(rp);
    pos := pos - 1;
    rp := (rp + 1) mod n;
    signal(buf_empty);
end

{начальные условия}
begin
    pos = 0;
    wp = 0;
    rp = 0;
end;
```

Механизм кольцевого буфера используется для организации работы spooler. SPOOL – Simultaneous Peripheral Operations On Line – для вывода текста на принтер

21. Пять моделей ввода-вывода в Linux : диаграммы, особенности.

	блокирующий	неблокирующий
синхронный	read/write	read/write (O_NONBLOCK)
асинхронный	i/o multiplexing (select/poll)	AIO

Синхронное-Блокирующее: процесс разблокируется и процесс в результа те премещения данных из ядра в соответсвующий буфер. Один из видов buttom half - завершение прерывание, посылка данных куда надо и процесс разблокируется. Картинка первая модель блокирующий ввод/вывод. По всем канонам явля ется синхронным

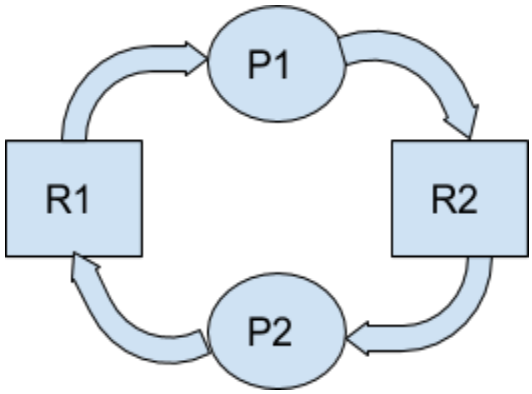


Все модели ввода/вывода в привязаны к сокетам.
а) модель №2: неблокирующий ввод вывод. Это синхронная модель. Связана эта модель с опросом polling. В соответствии с этой моделью, процесс выполняет системный вызов многократно. Это менее эффективный вариант синхронной блоки ровки, можно сказать, что это синхронный не блокирующий ввод/вывод. Очевидно, операция запрос ввода/вывода не может быть удовлетворен немедленно, и возвраща ется ошибка, что приводит к повторному запросу. Процесс вынужден многократно выполнять запросы, чтобы получить нужные данные.
б) модель №3: с мультиплексированием - ассинхронная блокирующая. Процесс блокируется на мультиплексере.

в) модель №3А: модель похожая на мультиплексирование, но связана с созда нием потоков(thread) каждый поток блокируется, но блокируется только один поток. Это дорогой путь.
г) модель №4: ввод/вывод управляемые сигналы SIGIO. Это асинхронный и рассматривается он как неблокирующий. В подчеркивается, что процесс блокируется, пока данные блокируются в приложении. Информирование процессов управля ется сигналами, пока сигнал не получен, процесс может выполнять вскую работу.
д) модель №5: асинхронный не блокирующий. Асинхронный - это когда инфо мирование приходит не зависимо от того, что в данный момент процесс выполняет. Все сигналы - асинхронные события, так же как и прерывания и связаны с преры ваниями. СОответственно смысл неблокирования в том, что процесс заинтересо ван в получении данных, он может обрабатывать большое колличества данных. Для увлечения эффективности, хорошо, при обработке одних данных, получать другие данные.
https://habrahabr.ru/post/111357/

22. Тупики - определение, условия возникновения тупиков, методы обхода тупиков - алгоритм банкира, семафоры(вроде читающие и ещё какие-то…)

Тупик – ситуация, возникающая в результате монопольного использования ресурсов, когда процесс, владея ресурсом, запрашивает другой ресурс, занятый непосредственно/через цепочку запросов процессом, который ожидает освобождения ресурсов, занятых первым процессом



Условия возникновения тупика:

а) взаимоисключение, когда процессы монопольно используют предоставляемые им ресурсы;

б) ожидание, связано с тем, что процесс удерживает занятые им ресурсы и ожидает предоставления доп. ресурсов для возможности продолжения выполнения;

в) неперераспределяемость, у процесса нельзя отобрать ресурсы до завершения процессов или до того момента, когда процесс сам освободит занимаемые ресурсы;

г) круговое ожидание, возникает замкнутая цепь запросов процессов к ресурсам, в котором каждый процесс занимает ресурс, необходимый следующему процессу в цепочке, для продолжения выполнения.

Борьба с тупиками (3 метода)

1) недопущение, создание в системе такой ситуации, когда тупики в принципе невозможны. Одна из причин должна быть устранена. Ханвендер в своей работе доказал, что если устранить хотя бы одну причину, то тупик не возникнет. Согласно этой стратегии существует три основных подхода: Если процесс сразу запрашивает все необходимые ему ресурсы. Так было в первых системах. Бесконечное откладывание!
2) обход, тупики возможны, но в системе прикладываются усилия, чтобы их обойти; - обнаружение тупиков, они возникают, и задача – обнаружить, а после – восстановить работоспособность системы;
3) Обнаружение тупиков

Стратегия Хандевера

1. Процесс должны запрашивать все необходимые им ресурсы (в современных системах это невозможно)
2. Упорядочиваем ресурсы. Ресурсы делятся на классы. Каждому такому классу присваивается номер и выполняется следующее правило: процессы могут запрашивать ресурсы из классов с номерами большими, чем номера классов, к

которым принадлежат классы, которые уже удерживают. Если он запрашивает ресурс с но- мером меньше, чем те, которые он уже удерживает, то он должен освободить все, что удерживает и запросить всё заново. Так он вырождается в первый способ (ему нужно всё опять запросить разом).

3. У процессов можно отбирать ресурсы, устраняет свойство неперераспределимости ресурсов. Если процесс, в процессе запроса, не может получить ресурс, то он должен освободить уже занятые ресурсы. Этот подход может привести к запросу и освобождению одних и тех же ресурсов (трешинг).

P1: P(S1); P(S2); ... V(S2); V(S1);	P2: P(S2); P(S1); ... V(S1); V(S2);

Обход тупиков. Алгоритм Банкира

Классическим подходом к обходу тупиков является алгоритм банкира, предложенный Дейкстрой. Банкиры дают заём. Задача банкира – дать заём тем, кто может их вернуть, но часто заемщик не может вернуть заем сразу, да и то, ему нужны еще заёмы. Банкир – менеджер системы. Заемы – процессы, делающие заявки на ресурсы, заявка процесса должна отображать макси- мальную потребность в ресурсе каждого класса. В процессе выполнения, процесс не может затребовать ресурсов больше, чем указал в заявке. Кол-во процессов – фикси- ровано. В системе должна быть информация, для этого определяются специальные структуры, при этом процесс запрашивает ресурсы по единице. МР гарантирует, что в системе тупик не возникнет. Каждый запрос проверяется по отношению к кол-ву ресурсов данного типа, имеющихся в системе. Анализируя ситуацию, МР ищет та- кую последовательность процессов, которая с учетом максимальных потребностей в ресурсе данного типа может завершиться. Если такая последовательность есть, то система не находится в тупике и состояние системы надежно, в результате запрошенные единицы ресурса выделяется процессу.

процессы	текущее распределение	свободные единицы	заявка
P1 P2 P3	1 3 5	 Всего 2	4 5 9

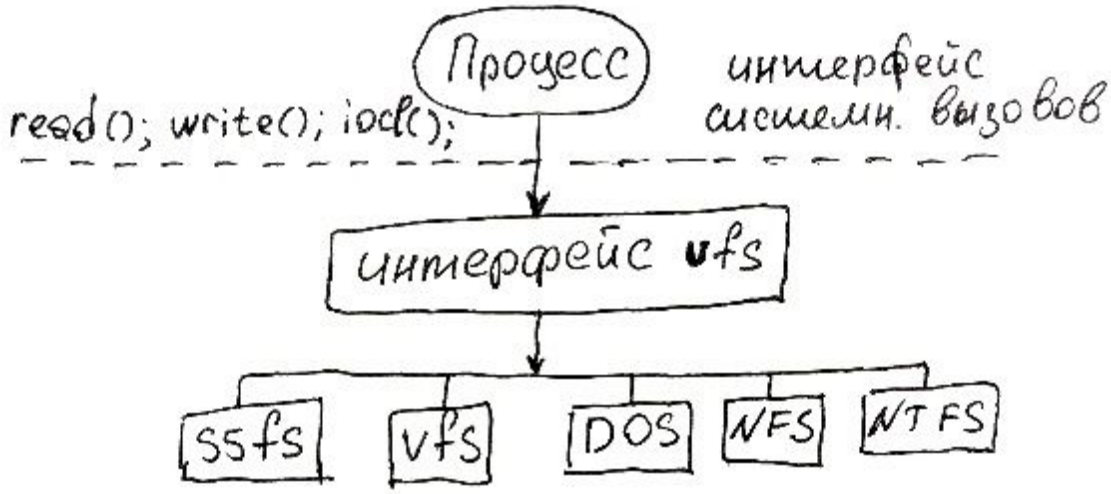
Данная последовательность является надежной и безопасной относительно тупика, так как в таблице имеется последовательность процессов, которая может завершиться. Если текущее состояние системы надежно, то это НЕ значит, что все последующие состояния системы будут надежными.

процессы	текущее распределение	свободные единицы	заявка
P1 P2 P3	2 3 5	 Всего 1	4 5 9

Такое состояние не является надежным, т.к. нет последовательности процессов, которая может завершиться.Но процессы могут и не запросить максимально заявленное кол-во ресурсов. Состояние системы является безопасным, если существует последовательность процессов такая, что первый процесс в последовательности обязательно завершится, так как если он запросит максимально заявленное кол-во ресурсов, то в системе хватит ресурсов. Второй процесс может завершиться, если первый процесс завершится и вернет занимаемые им ресурсы, и в сумме этих ресурсов будет достаточно для удовлетворения потребностей. И так далее. Когда процесс делает новый запрос, менеджер ресурсов должен найти последовательность успешно завершаемых процессов, и только в этом случае запрос будет удовлетворен. Запрос процесса, переводящий систему в ненадежное состояние – откладывается. Так как система постоянно поддерживается в надежном состоянии, то за конечное время все запросы будут завершены и все процессы смогут завершить свою работу, но каждый раз требуется исследовать n! последовательностей, что предполагает большие затраты на выполнение данного алгоритма. Таким образом, алгоритма банкира имеет теоретическое значение. Это важно для систем реального времени, которые управляют внешними по отношению к системе процессами. Чаще всего это системы автопилота, ставятся в автомобили. Не нужно путать систему реального времени с системой разделения времени.

23. ФС Unix, vfs/vnode, inode, адресация больших файлов.

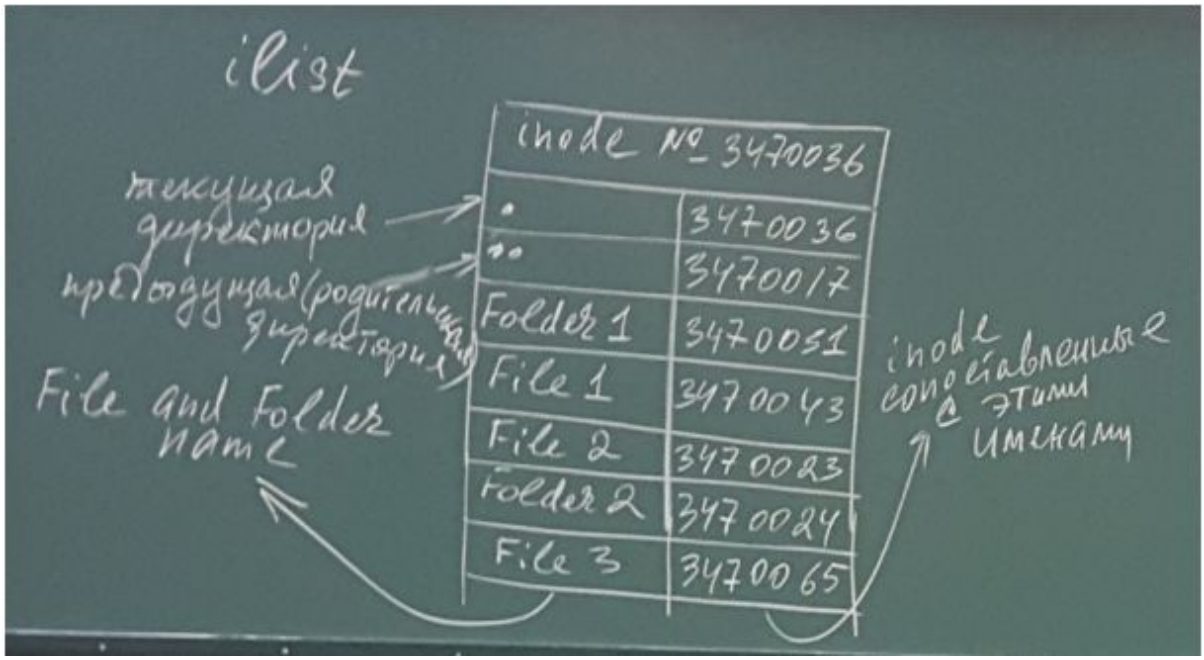
Декларация: в UNIX всё файл. Следствие этого - работа с файлами и устрой ствами одними и теми же системными вызовами позволила сократить количество системных вызовов. В Unix можно установить любое количество файловых систем. Эта идея полу чила название *vnode*/*vfs*. Этот интерфейс имеет объектно ориентированный подход. На основе 1 класса можно породить 1 или более новых классов наследников. Класс наследник может стать базовым для его наследников.

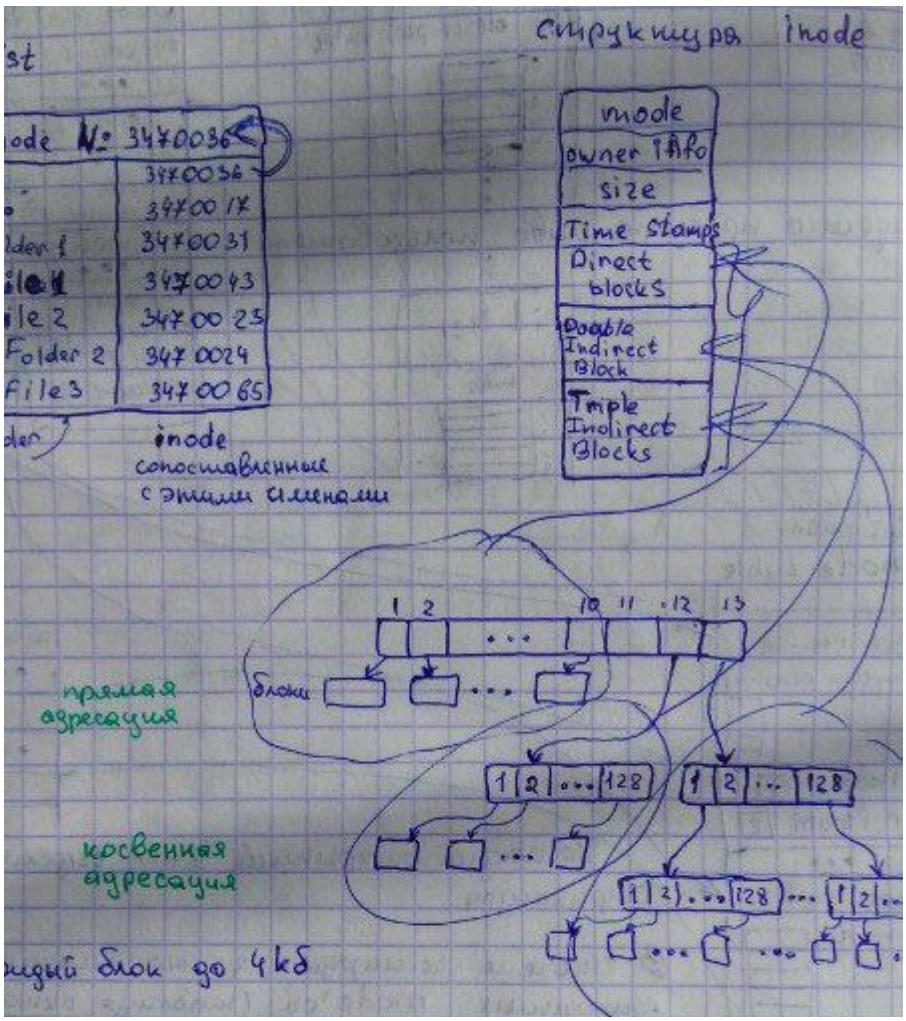


VFS - специальный интерфейс ядра linux для поддержки файловых систем. Virtual Filesystem Switch (vnode/vfs) – производный от SVR4.

inode

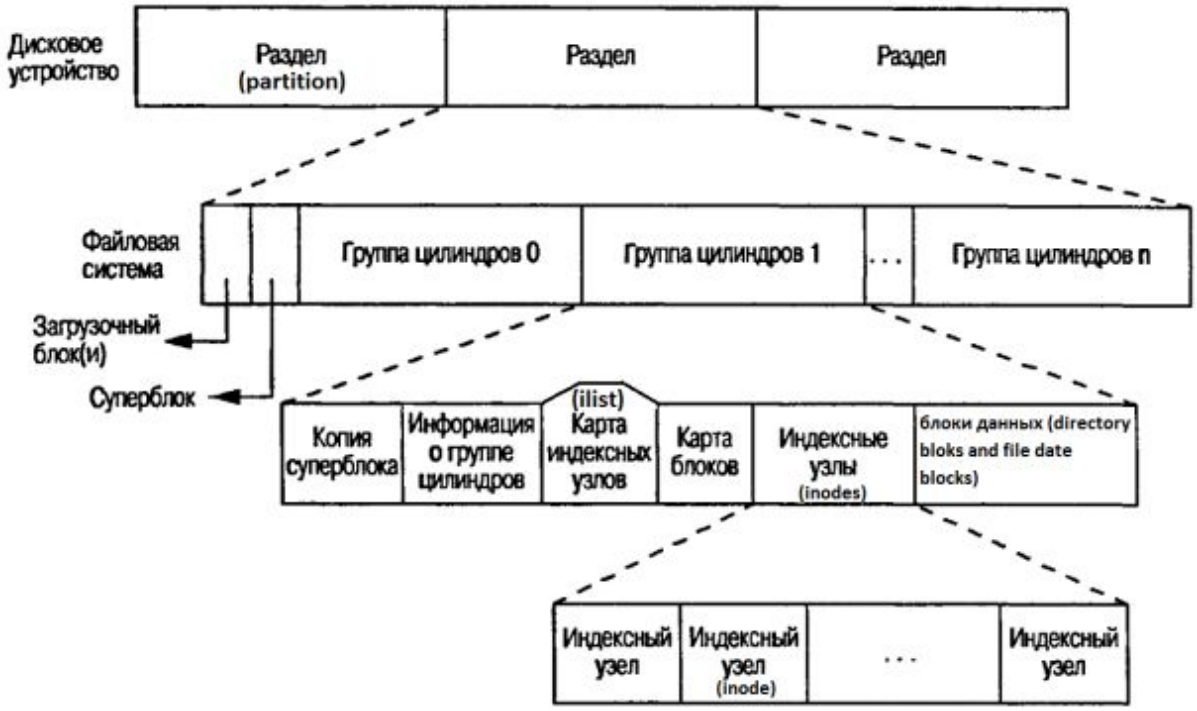
inode – это указатель, в котором хранится информация о физическом расположения файла на диске. inode хранятся на диске, но в системе кэшируется информация об inode открытых файлах (таблица активных inode). Эта таблица находится в области данных ядра системы. Все inode хранятся на диске в массиве фиксированного размера, который называется ilist. Доступ к фай лу осуществляется по номеру inode. Информация о дескрипторах файла (об inode) часто называется мета-данными. Мета данные – это данные о данных. inode иногда называют номером или индексом (индекс лучше, т.к. это смещение в таблице).



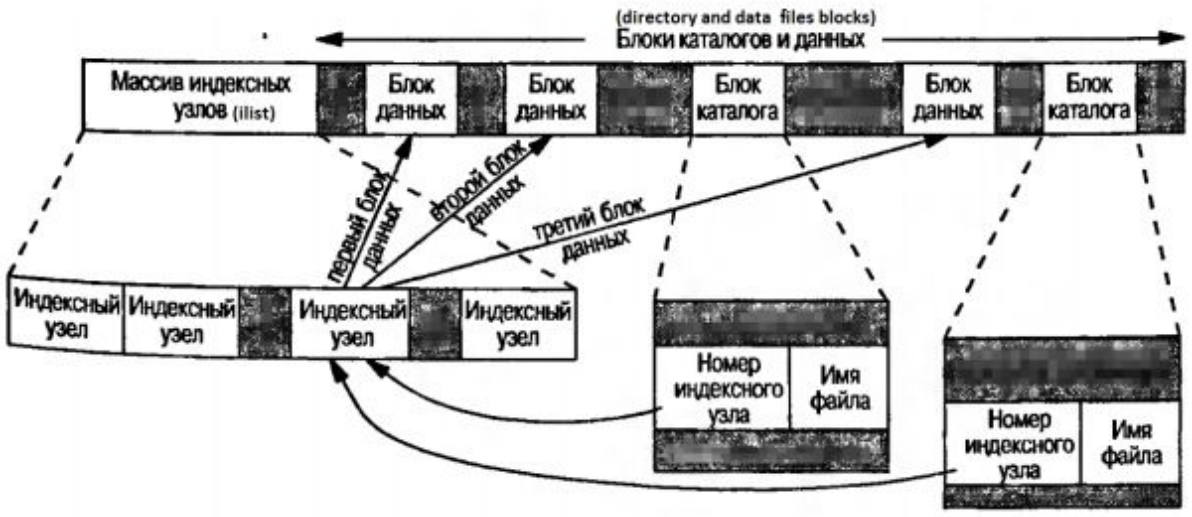


Через косвенную адресацию позволяет адресовать большие файлы.
диски делятся на partition (разделы), а файловая система занимает partition.

Дисковое устройство, разделы и файловая система



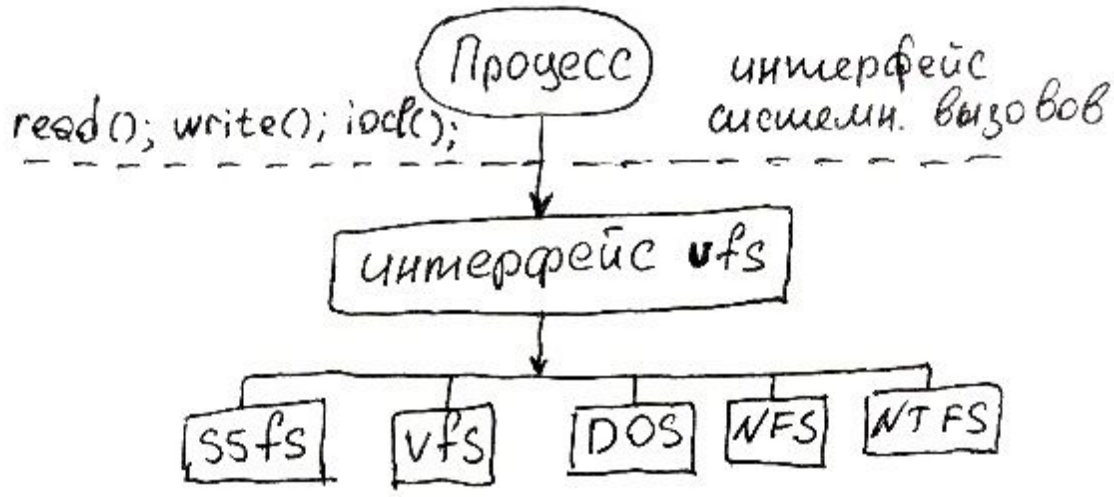
Часть группы цилиндров с индексными узлами и блоками данных более детально. Индексный узел - это inode.



24. Три режима работы компьютера на базе процессоров Intel. Прерывания в защищенном режиме (таблица ID). преобразование адреса при страничном преобразовании в процессорах Intel.

25. Примеры спец. файловых систем. Файловая система Linux -Интерфейс vfs/vnode. (в интерфейсе отвечал в общем про файловую систему и vnode, главное надо было показать struct superblock и struct dentry; не надо писать про ФС в Windows. А также в ответе должны обязательно присутствовать описания структур dentry и superblock.)

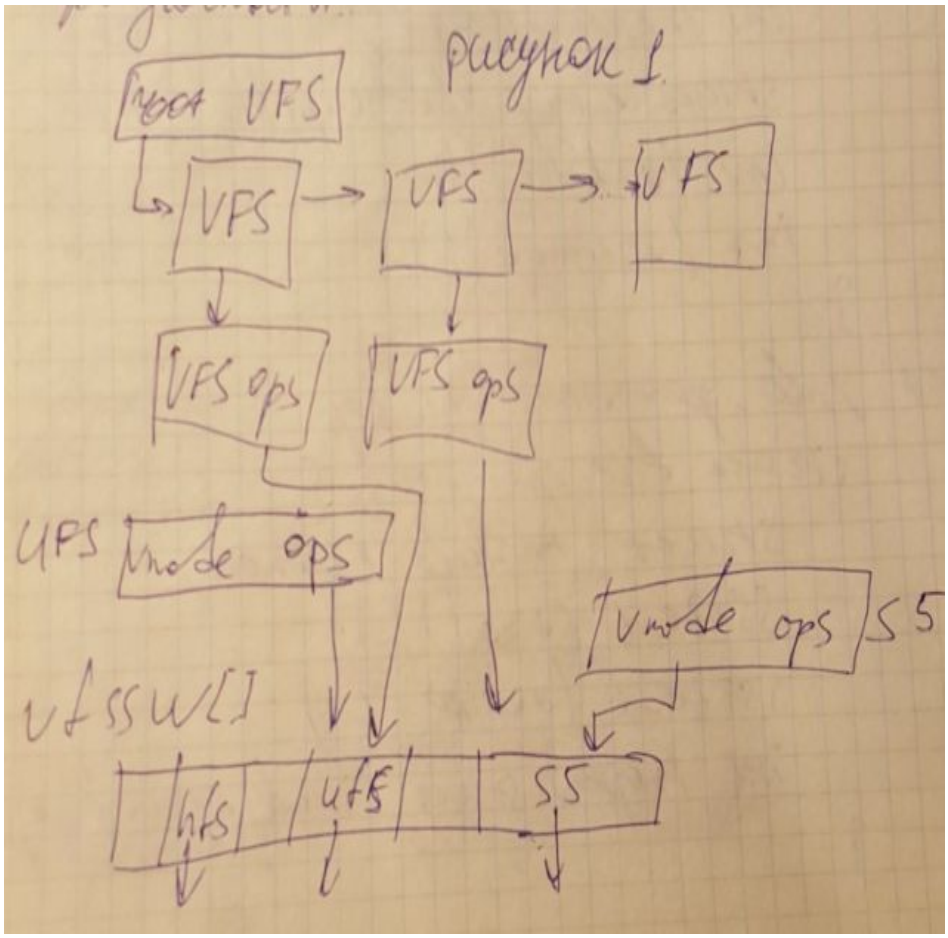
В Unix можно установить любое количество файловых систем. Эта идея получила название vnode/vfs. Этот интерфейс имеет объектно ориентированный подход.
На основе 1 класса можно породить 1 или более новых классов наследников. Класс наследник может стать базовым для его наследников.



Даже ФС в разделе ЖД нужно монтировать. Отметим, что при инсталляции Linux и создании на ЖД раздела Linux, автоматически конфигурируется на монтировании основных ФС-м при каждом запуске.

Монтировать ФС может только привилегированный пользователь. Точка монтирования – каталог. Устройство – специальный файл, с помощью которого система получает доступ к физическому устройству.

VFS - специальный интерфейс ядра linux для поддержки файловых систем.
Virtual Filesystem Switch (vnode/vfs) – производный от SVR4.



struct vfssw vfssw [ ] = {
{0 ,0 ,0 ,0}
{ "spec", specinit, &spec_vfsops, 0}
{ "ufs", ufsinit, &ufs_vfsops, 0}
}
	struct vfs {
	struct vfs * vfsnext;
	struct vfs * vfs_op ;
	// vnode mounted on
	struct vnode * vfs_vnode_covered
	struct v f s * vfs_hash ;
	}

Суперблок – начальная точка любой ФС, содержит инфу о ФС. Хранится в специальном секторе диска и имеет структуру super_block.

struct super_block {
// инфа для ФС
union {}
семафоры ;
struct l i s t _ h e a d _ l i s t ;
// опрееляет операции , специфичные для конкретной ФС
struct s u p e r _ o p e r a t i o n s s_op ;
//каталоги точки монтирования

```
struct d e n t r y * s_root ;
 //всех подмонтированных подсистем
 struct l i s t _ h e a d s_mount ;
 //список индексов
 struct l i s t _ h e a d s_inode ;
//список измененных индексов
 struct l i s t _ h e a d s _ d i r t y ;
 // список всех открытых файлов в данном суперблоке
 struct l i s t _ h e a d s _ f i l e s ;
}
```

Доступ к суперблоку должен быть монопольным.

```
struct s u p e r _ o p e r a t i o n s {
 struct i n o d e * ( * a l l o c _ i n o d e ( struct super_block * sb ) ) ;
 void ( * d e s t r o y _ i n o d e ) ( struct i n o d e * ) ;
 void ( * read_inode ) ( struct i n o d e * ) ;
 void ( * w r i t e _ i n o d e ) ( struct i n o d e * , int ) ;
 int ( * s y n c _ f s ) ( struct super_block * sb , int w a i t ) ;
}
```

26. Файловые системы. Иерархическая структура файловой системы в Unix, задачи уровней. Открытые файлы - структуры описывающие файл. Особенности использования open() и fopen().

По функциональности, ядра почти не различаются: поддерживают многопроцессность, многопоточность, виртуальную память. Управление памятью происходит страницами по запросам. Основное отличие ОСей – файловые системы. Обычно, файловая система - иерархическая.

Таблица 1.1 — Уровни

1 символьный верхний, для именования файлов

2 базовый позволяет обратиться к inode

3 проверки прав доступа возможен только на уровне inode

4 логический объясняет, как получить доступ к отдельному блоку/байту

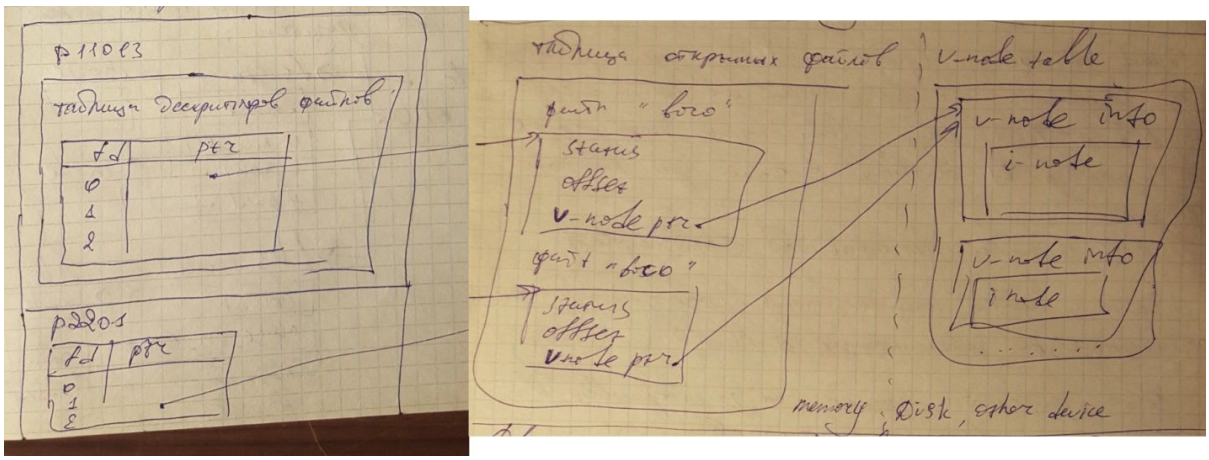
5 физический уровень устройства, подсистема ввода/вывода

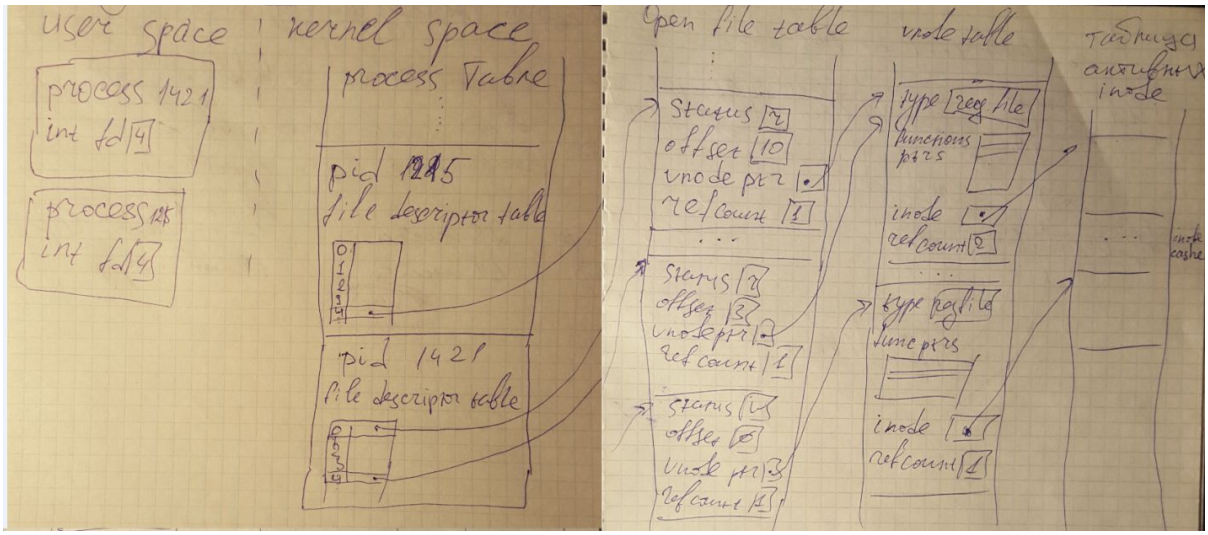
Декларация: в UNIX всё файл. Следствие этого - работа с файлами и устрой
ствами одними и теми же системными вызовами позволила сократить количество системных вызовов.

В Unix можно установить любое количество файловых систем. Эта идея полу
чила название vnode/vf s. Этот интерфейс имеет объектно ориентированный подход. На основе 1 класса можно породить 1 или более новых классов наследников. Класс наследник может стать базовым для его наследников.

Файл – любая поименованная совокупность данных, размещенная на запоми
нающих устройствах.Когда запускается процесс, он заинтересован в чтении/записи файлов. Библиотечные функции написаны для минимизации системных вызовов, т.к. системные вызовы переводят ОС в режим ядра.

Любой запущенный процесс имеет дескриптор в таблице процессов. Все от
крытые файлы описаны в таблице открытых файлов, которая одна на систему.





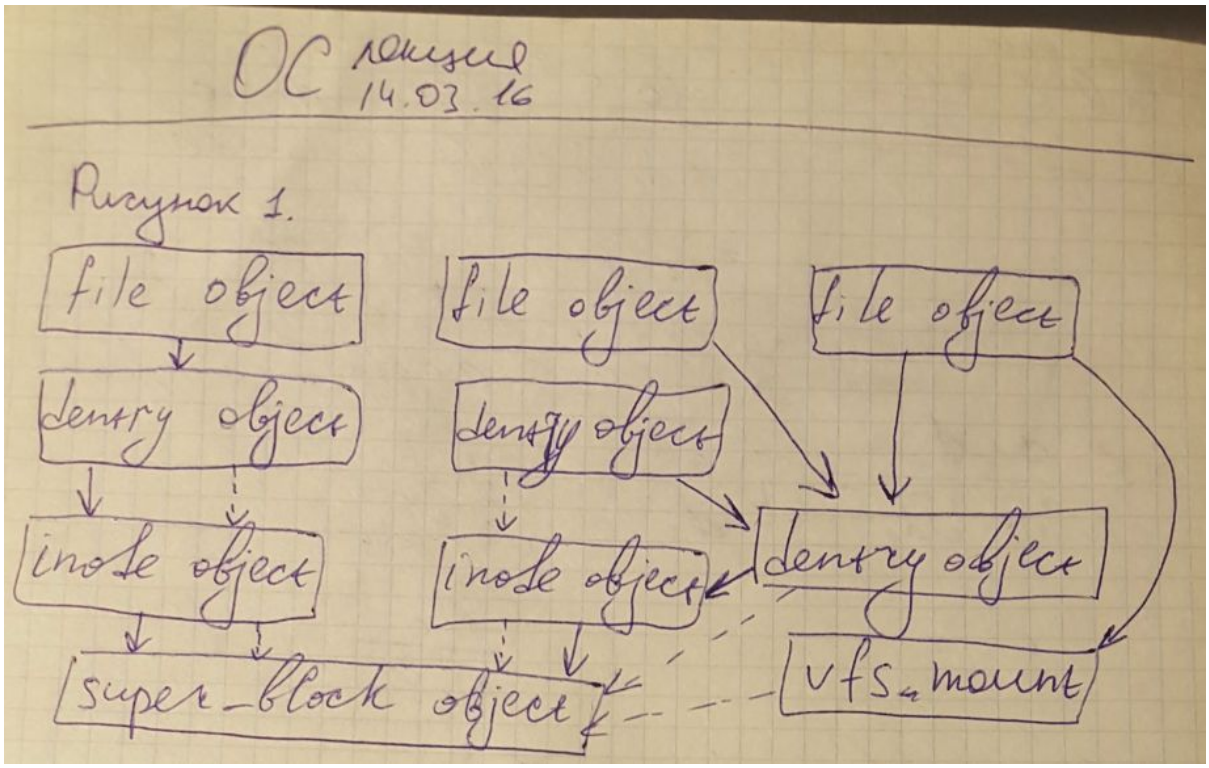
//описывает каждый открытый файл , сокет , . . .
struct f i l e {
mode_tfmode ;
// текущая позиция в файле для операции чтения в файле
loff_tf_pos ;
...
struct i n o d e * f_inode ;
// описывает операции , определенные на файле в системе
struct f i l e _ o p e r a t i o n s * f_op ;
}
	struct f i l e s _ s t r u c t {
	atomic_t count ;
	//Для взаимоисключения в ядре . Построено на команде t e x t a n d s e t , это активное ожидание . Ядро часто не может блокироваться , поэтому используется активное ожидание . Это поле помогает защитить структуру .
	spinlock_t f i l e _ l o c k ;
	...
	int next_fd ;
	//массив файловых объектов , максимум 32 , если больше , то создается новая таблица
	struct f i l e ** f d ;
	struct f i l e * fd_array [NR_OPEN_DEFAULT] ;
	}

inode

inode – это указатель, в котором хранится информация о физическом расположения файла на диске. inode хранятся на диске, но в системе кэшируется информация об inode открытых файлах (таблица активных inode). Эта таблица находится в области данных ядра системы. Все inode хранятся на диске в массиве фиксированного размера, который называется ilist. Доступ к фай лу осуществляется по номеру inode. Информация о дескрипторах файла (об inode) часто называется мета-данными. Мета данные – это данные о данных. inode иногда называют номером или индексом (индекс лучше, т.к. это смещение в таблице).

Системный вызов **open** реализован в виде функции fs/open.c:sys_open, но основную работу выполняет функция fs/open.c:filp_open(), которая разбита на

две части:
а) open_namei(): заполняет структуру nameidata, содержащую структуры dentry и vfsmount.
б) dentry_open() с учетом dentry и vfsmount, размещает новую структуру file и связывает их между собой; вызывает метод f_op->open() который был уста новлен в inode->i_fop при чтении inode в open_namei() (поставляет inode через dentry->d_inode).



Объект dentry не соответствует какой либо структуре данных на жестком диске. Создает на основании строкового имени пути.

Следует использовать библиотечный вариант fopen() по причине минимизации системных вызовов по сравнению с open().

27. Процессы unix: демоны, примеры системных демонов, правила создания демонов

Демоны – это долгоживущие процессы. Зачастую они запускаются во время загрузки системы и завершают работу вместе с ней. Так как они не имеют управляющего терминала, говорят, что они работают в фоновом режиме. В системе UNIX демоны решают множество повседневных задач.

и сессиями. Команда ps(1) выводит информацию о процессах в системе. Эта команда имеет множество опций, дополнительную информацию о них вы найдете в справочном руководстве. Мы запустим команду

```
ps -axj
```

```
 PPID   PID  PGID   SID TTY TPGID UID   COMMAND
    0     1     0     0 ?    -1   0   init
    1     2     1     1 ?    -1   0   [keventd]
    1     3     1     1 ?    -1   0   [kapmd]
    0     5     1     1 ?    -1   0   [kswapd]
    0     6     1     1 ?    -1   0   [bdflush]
    0     7     1     1 ?    -1   0   [kupdated]
    1  1009  1009  1009 ?    -1  32   portmap
    1  1048  1048  1048 ?    -1   0   syslogd -m 0
    1  1335  1335  1335 ?    -1   0   xinetd -pidfile /var/run/xinetd.pid
    1  1403     1     1 ?    -1   0   [nfsd]
    1  1405     1     1 ?    -1   0   [lockd]
 1405  1406     1     1 ?    -1   0   [rpciod]
    1  1853  1853  1853 ?    -1   0   crond
    1  2182  2182  2182 ?    -1   0   /usr/sbin/cupsd
```

Из данного примера мы убрали несколько колонок, которые не представляют для нас особого интереса. Здесь показаны следующие колонки, слева направо: идентификатор родительского процесса, идентификатор процесса, идентификатор группы процессов, идентификатор сессии, имя терминала, идентификатор группы процессов терминала (группы процессов переднего плана, связанной с управляющим терминалом), идентификатор пользователя и строка команды.

Демон cron (crond) производит запуск команд в определенное время. Множество административных задач выполняется благодаря регулярному запуску программ с помощью демона cron. Демон cupsd – это сервер печати, он обслуживает запросы к принтеру.

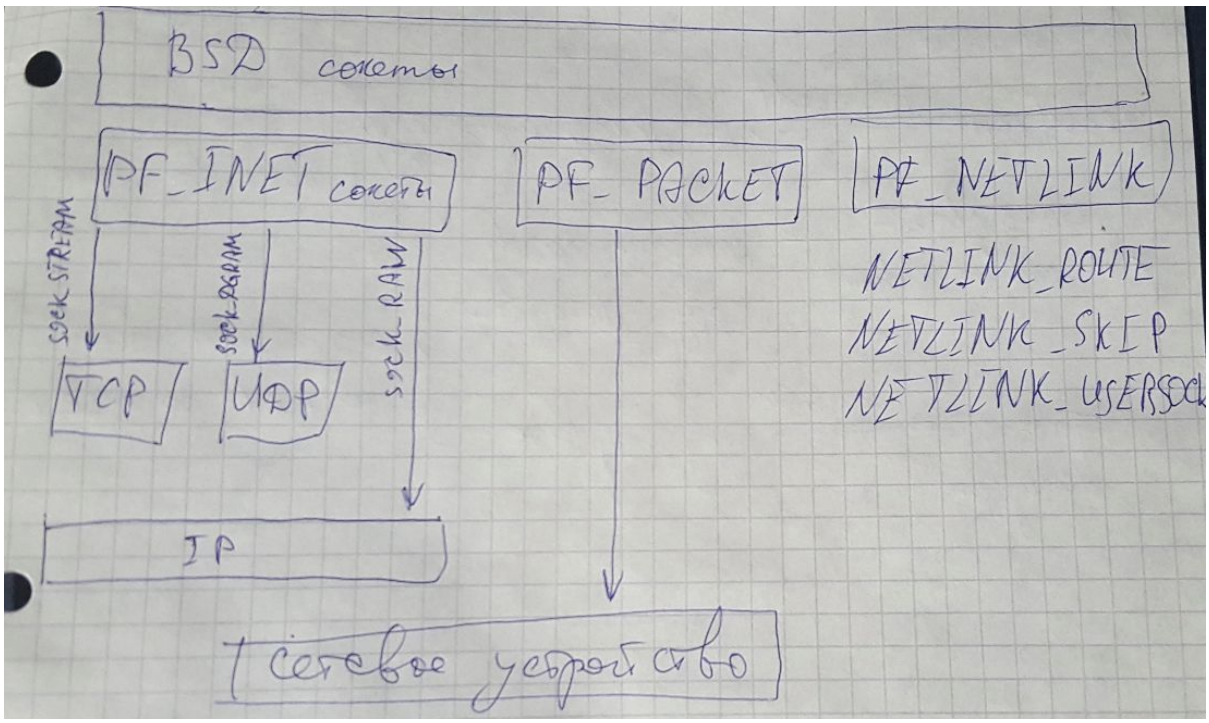
Обратите внимание: большинство демонов обладают привилегиями суперпользователя (имеют идентификатор пользователя 0). Ни один из демонов не имеет управляющего терминала – вместо имени терминала стоит знак вопроса, а идентификатор группы переднего плана равен –1. Демоны ядра запускаются без управляющего терминала. Отсутствие управляющего терминала у демонов пользовательского уровня – вероятно, результат вызова функции setsid. Все демоны пользовательского уровня являются лидерами групп и лидерами сессий, и они являются единственными процессами в своих группах процессов и сессиях. И, наконец, обратите внимание на то, что родительским для большинства демонов является процесс init.

Правила программирования демонов:
1) вызвать umask для сброса маски создания файлов
   - маска наследуется и может маскировать биты прав доступа (запись, чтение)
2) вызвать fork() и завершить предка
   - чтобы командная оболочка думала, что команда была выполнена
   - чтобы новый процесс гарантированно не был лидером группы, что позволит вызвать setsid (у дочернего процесса id отличный от родителя, а pgid наследуется), чтобы процесс СТАЛ лидером группы
3) создать новую сессию, вызвав setsid, тогда процесс станет:
   - лидером новой сессии
   - лидером новой группы процессов
   - лишится управляющего терминала (TTY = ?)
4) сделать корневой каталог текущим рабочим каталогом
   - если рабочий каталог на смонтированной файловой системе, то её нельзя будет отмонтировать, так как процессы-демоны обычно живут, пока система не перезагрузится
5) закрыть все ненужные открытые файловые дескрипторы, которые процесс-демон может унаследовать и препятствовать их закрытию (для этого нужно сначала получить максимальный номер дескриптора (см. код))
6) такой процесс не связан ни с каким терминальным устройством и не может взаимодействовать с пользователем в интерактивном режиме, даже если он был запущен в рамках интерактивной сессии, он все равно будет переведен в фоновый режим (некоторые процессы-демоны открывают файловые дескрипторы 0 1 и 2 на dev/null - "пустые" stdin, stdout, stderr, что позволяет вызывать в них фунции стандартного ввода вывода, не получая при этом ошибок)

28. Сокеты - определение, типы, семейства, сетевой стэк(порядок вызова функций - схема), преобразование байтов (little-big endian), мультиплексирование (ДОПОЛНЯЮ 17 ВОПРОС)
(TODO мультиплексирование и little big endian)

30. Модель ввода-вывода с мультиплексированием. Сетевые сокеты с мультиплексированием, установление соединения, сетевой стэк

-	блокирующий	неблокирующий
синхронный	read/write	read/write (O_NONBLOCK)
асинхронный	i/o multiplexing (select/poll)	AIO



(TODO Илюха использует какую-то книгу, уточнить является ли она по мнению Рязановой достоверным источником, а так инфы больше нет кроме этих картиночек дальше ересь какая то)

31. Файловая система proc. Функции для передачи данных (между процессом и ядром). Модули ядра, суть, пример.

Unix поддерживает виртуальную файловую систему proc. В зависимости от версии ядра, м.б. procfs. Не является монтируемой ФС. Это пример виртуально ФС. Это интерфейс ядра, позволяющий приложениям читать и изменять данные в адресном пространстве других процессов, управлять прцоессами, получать информацию о процессах и ресурсах используемых процессами путем использования стандартного интерфейса ФС и системных вызовов. Следует, что управление доступом к адресному пространству осуществляется с помощью обычных прав доступа, а именно чтение/запись/выполнение. По умолчанию запись/чтение файлов proc разрешены только для владельцев. В ФС proc данные о каждом процессе хранятся в поддиректории /proc/ < PID > В поддиректории процесса находятся файлы и поддиректория, содержащие данные о процессе.

элемент	тип	содержание
cmdline	файл	указатель на dir процесса
cwd	символ ссылка	указатель на dir процесса
environ	файл	список окружения процесса
exe	символ ссылка	указывает на образ процесса (на файл)
fd	директория	ссылки на файлы, открытые процессом
root	символ. ссылка	на корень ф.сист. процесса
stat	файл	информация о прцоессе.

Виртуальная ФС – информация о процессах. Виртуальная – потому что не на диске. Текстовый файл с нужной инфомацией создается, когда выполняется запрос на чтение соответствующей информации. Взаимодейсвтие с ядром заключается в чтении информации из ядра / записи новой информации в ядро.

Системными прогерами ФС proc широко используется и её возможности демонстрируются загружаемыми модулями ядра (Loadable Kernel Module (LKM)). Драйверы устройств компилируются в виде модулей, ядро имеет модульную струк туру, компилируется в виде загружаемых модулей. Он загружается когда в нем воз никает потребность (динамически). Если драйвер скомпилирован как часть ядра, то его код и статические данные, даже когда драйвер не используется. Если драйвер скомпилирован как загружаемый модуль, то он будет занимать память, только если в нем возникла необходимость и он загружен в ядро. Отмечается, что заметные потери производительнстои при иссползовании загружаемых модулей ядра не происходит. ЗМЯ являются средставм адаптируемым к конкретным подключенным устройствам.

Листинг 1.11 — Структура proc_dir_entry

```
1  //используется для работы с файлами и поддиректорями
2  struct proc_dir_entry {
3
4      // номер индекса inode файла.
5      unsigned int low_ino;
6
7      //имя файла
8      const char *name;
9
10     //длина имени
11     unsigned short namelen;
12
13     //права доступа
14     mode_t mode;
15
16     // колво- ссылок на файл
17     nlink_t nlink;
18
19     uid_t uid;
20     gid_t gid;
21
22     //размер
23     unsigne long size;
24
25     struct inode_operations *proc_iops;
26     struct file_operations *proc_fops;
27     get_info_t *get_info;
28     struct module *owner;
29
30     //указатели, позволяющие создавать связанные списки
31     struct proc_dir_entry *next, *parent, *subdir;
32
33     read_proc_t *read_proc;
34     write_proc_t *write_proc;
35
36     //колво- ссылок
37     atomic_t count;
38
39     int deleted;
40 }
```

С версии ядра v2.6 поддерживается сборка модулей с использованием make файла. Для сборки соответствующего модуля необходимо создать make файл, содержащий 1 строку:

```
1  obj-m += simple-lkm.o
```

Загружаемые модули ядра пишутся по определенным правилам: в текст мо дуля обязательно должны быть включены два макроса (module_init module_exit: они регистрируют нашу функцию). Макросы вставляются в текст. В отличии от функций, которые имеют собственное адресное пространство, макросы являются вставками в текст какой то программы, мы не перекомпилируем ядро, но тем не менее это вставка. В разных примерах по разному регистрируются call-back функции. Необязательно использовать copy_to_user (пример - когда используется mem_copy()). Существуют разные способы взаимодействия с виртуальными файлами: есть ФС proc, есть ФС procfs. Кроме copy_to_user() copy_from_user() упоминаются еще get_user() put_user().

22. Задача читатели-писатели, решение с использованием семафоров Дейкстра для ОС Unix, синхронизация потоков в Windows, мьютекс, Consumer.

23. Три режима работы компьютера на базе процессоров Intel. Прерывания в защищенном режиме (таблица ID). преобр адреса при страничном преобразовании в процессорах Intel.

1.

Реальный режим (или режим реальных адресов) - это название было дано прежнему способу адресации
памяти после появления 286-го процессора, поддерживающего защищённый режим.
Реальный режим поддерживается аппаратно. Работает идентично 8086 (16 разрядов, 20-разрядный адрес –
сегмент/смещение). Минимальная адресная единица памяти – байт.
2 20 = FFFFF = 1024 Кб = 1Мб (объем доступного адресного пространства).
Компьютер начинает работать в реальном режиме. Необходим для обеспечения функционирования программ,
разработанных для старых моделей, в новых моделях микропроцессоров.
1-проц. режим под управлением MS-DOS (главное – минимизация памяти, занимаемой ОС => нет
многозадачности).
0-256 б.:
таблица векторов прерываний

резидентная часть DOS

место резидентных программ

сегменты программы

pool или heap (куча)

транзитная часть DOS

Резидентная часть DOS

1 Мб

2.

Защищенный режим – многопроцессный режим. В памяти компьютера одновременно находится
большое число программ с квантованием процессорного времени с виртуальной памятью. Управляет
защищенным режимом ОС с разделением времени (Windows, Linux). В защищенном режиме 4 уровня
привилегий, ядро ОС – на 0-м. Создан для работы нескольких независ. программ. Для обеспечения совместной
работы нескольких задач необходимо защитить их от взаимного влияния, взаимод. задач д. регулироваться.
Разработан фирмой Digital Equipments (DEC) для 32-разрядных компьютеров VAX-11. Формирование таблиц
описания памяти, которые определяют состояние её отдельных сегментов/страниц и т. п.
3.
Специальный режим защищенного режима (V86) – как задачи выполняются ОС реального режима, в
кажд. из кот. выполн. по 1 прогр. реального режима. Многозадачный режим с поддержкой виртуальной памяти.
процессор фактически продолжает использовать схему преобразования адресов памяти и средства
мультизадачности защищённого режима. В виртуальном режиме используется трансляция страниц
памяти. Это позволяет в мультизадачной операционной системе создавать несколько задач, работающих в
виртуальном режиме. Каждая из этих задач может иметь собственное адресное пространство, каждое
размером в 1 мегабайт. Все задачи виртуального режима обычно выполняются в третьем, наименее
привилегированном кольце защиты. Когда в такой задаче возникает прерывание, процессор автоматически
переключается из виртуального режима в защищённый. Поэтому все прерывания отображаются в
операционную систему, работающую в защищённом режиме.
VMM (Virtual Machine Manager) – для запуска виртуальных машин реального режима.

В процессоре i386 компания Intel учла необходимость лучшей поддержки реального режима, потому что
программное обеспечение времени его появления не было готово полностью работать в защищенном режиме.
Поэтому, например, в i386, возможно переключение из защищенного режима обратно в реальный (при

разработке 80286 считалось, что это не потребуется, поэтому на компьютерах с процессором 80286 возврат в
реальный режим осуществляется схемно - через сброс процессора).
В качестве дополнительной поддержки реального режима, i386 позволяет задаче (или нескольким задачам)
защищенного работать в виртуальном режиме — режиме эмуляции режима реального адреса (таким образом в
переключении в реальный режим уже нет необходимости). Виртуальный режим предназначается для
одновременного выполнения программы реального режима (например, программы DOS) под операционной
системой защищенного режима.
Выполнение в виртуальном режиме практически идентично реальному, за несколькими исключениями,
обусловленными тем, что виртуальная задача выполняется в защищенном режиме:
•виртуальная задача не может выполнять привилегированные команды, потому что имеет наинизший
уровень привилегий
•все прерывания и исключения обрабатываются операционной системой защищенного режима (которая,
впрочем, может инициировать обработчик прерывания виртуальной задачи)