

# Рязань, 1 лекция, 02.10.19

## Файловые подсистемы.

Мы рассматривали иерархическую машину Медника Донована. Она построена относительно процессов, на самом верхнем уровне находился уровень управления данными, или по-другому, это уровень файловых подсистем, уровень долговременного хранения данных, доступа.

Временные файлы называются рабочими. Все файлы рабочие. Временные файлы, создаваемые для обслуживания системы. Существует масса вариаций в определении файловых систем, которые описывают одну суть. В отношении файлов такая вариация отсутствует, если понимать о чем идет речь. Для долговременного хранения информации, существуют специальные внешние устройства.

Управление файлами осуществляется частью ОС, которую принято называть файловой системой или файловой подсистемой (FileSystem). Файловая система - часть ОС, которая отвечает за возможность долгого и надежного хранения и доступа инфо (создание, чтение, запись, именование, переименование, удаление, изменение прав доступа и т.д.). Если файл предназначен для хранения информации (данных), то фс управляет процессом хранения и обеспечивает последующий доступ к этой информации. То есть файл, фактически, в файловой системе является единицей хранения информации. Для обычных файлов важно подчеркивать, что они хранятся во вторичной памяти. Бывают специальные файлы (программные каналы), они создаются в оперативной памяти.

### Определение файла (UNIX-овое)

Файл - каждая индивидуальная индексированная совокупность информации называется файлом. Каждая индивидуально идентифицируемая единица информации называется файлом.

Каждая поименнованная совокупность данных, хранящаяся во вторичной памяти называется файлом. Речь идет об обычных файлах

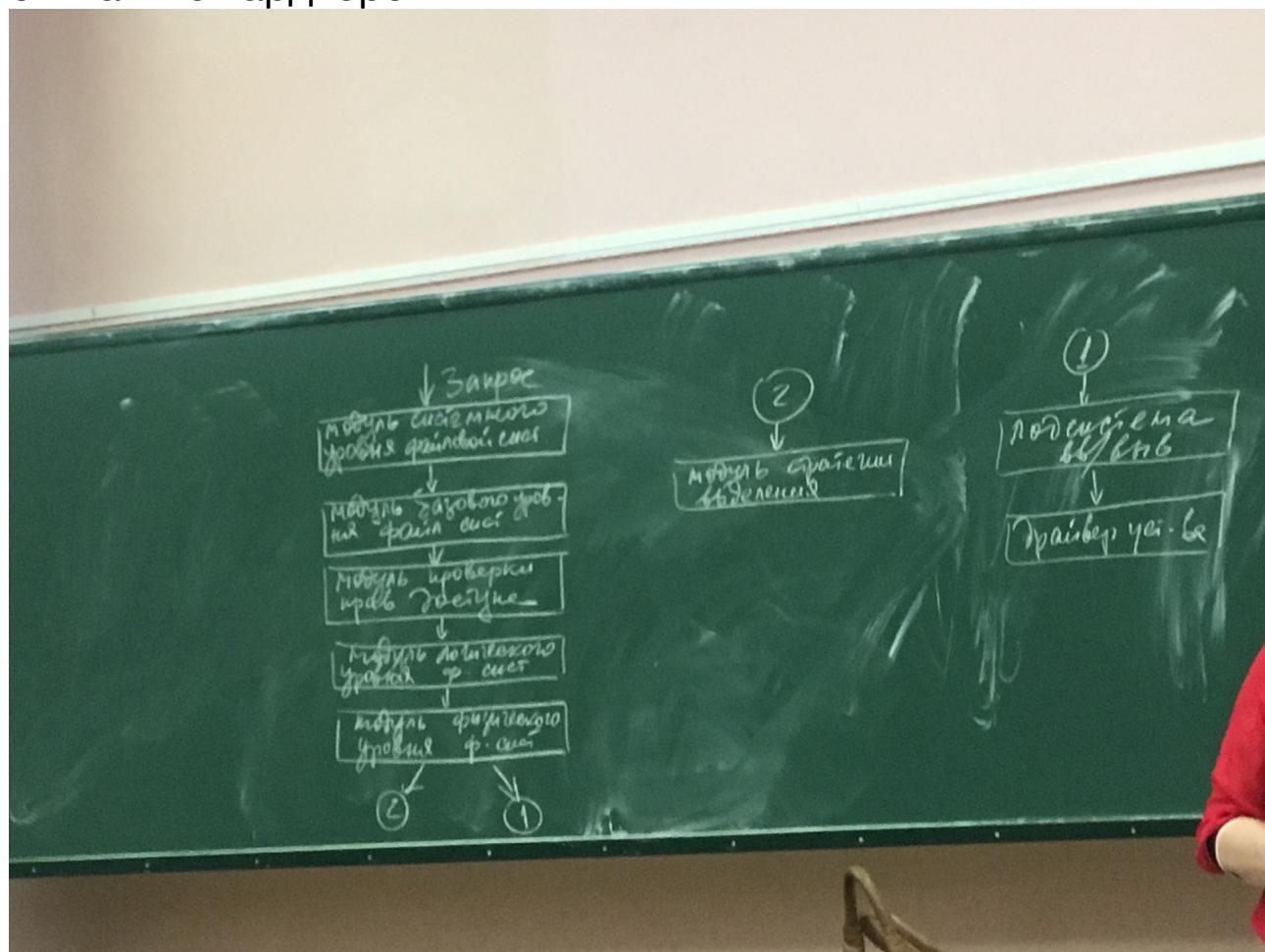
(Это определение Рязановой)

Когда говорят о фс, то речь идет об обычных файлах.

Файловая система определяет: а) формат сохраняемой информации и способ ее физического хранения, б) связывает формат физического хранения и API для доступа к файлам.

Файловая система не интерпретирует информацию, хранящуюся во вторичной памяти. Этим занимаются соответствующие программы. Вторичную память, профессионально определяют как энерго-независимую.

Рассмотрим обобщенную модель файловой подсистемы и рассмотрим отдельно ее уровни, фс как правило имеет иерархическую организацию, в ней определены уровни, эти уровни от вышележащего к нижележащему. Верхние уровни связаны с работой с файлами пользователей. Нижние уровни связаны с хардвером.



Это пример структурирования файловой системы. Вопросы структурирования ос при управлении информации, является едва ли не самыми важными при разработке системы. Очевидно, что такое структурирование должно учитывать задачи, которые должна решать файловая система.

### **Задачи фс:**

1. Обеспечение удобного доступа пользователей к файлам как часть задачи именования файлов.
2. Собственно, именование файлов, то есть присвоение файлам уникальных идентификаторов, с помощью которых можно обеспечить доступ к файлам.
3. Обеспечение программного интерфейса для работы с файлами пользователей и приложений.
4. Отображение логической модели или логического представления файлов на физическую организацию хранения данных на соответствующих носителях.
5. Обеспечение надежного хранения файлов, доступа к ним и обеспечение защиты от несанкционированного доступа.
6. Обеспечение совместного использования файлов.

### **Системный уровень** - это символьный уровень.

Развитые файловые системы (win, unix/linux), должны обеспечивать возможность существования в системе нескольких файлов с одинаковыми именами, обеспечивать возможность доступа к одному и тому же файлу по разным именам. Поэтому, в системах должна существовать соответствующая информация (справочник). Такой справочник состоит из двух частей, или состоит из двух отдельных справочников (символьные и базовые уровни)

**Символьный уровень** - уровень именования файлов удобный для юзера, уровень именования каталогов, уровень оперирования пользователем и приложением файла.

Удобным способом стркутурирования различных файлов, являются директории. Директории определяются как каталоги (папки).

**Базовый уровень** это уже уровень идентификации файла. Очевидно, что и файловая подсистема это программа, и как любая программа, она работает по определенным принципам. Понятно, что файл должен иметь уникальный id. А поскольку система должна давать возможность называть файл разными именами (hardlink) и имеет несколько файлов с одним именем. Файл должен быть описан в системе, чтобы можно было с ним работать. Модуль проверки прав доступа - это кэп.

Модуль логического уровня. Мы говорили о том, что любая программа считает, что она начинается с нулевого адреса, что называется логическим адресным пространством, с файлами также. Когда мы создаем файл указатель стоит на начало файла, файл имеет логическое адресное пространство, которое также начинается с нуля. То есть логический уровень позволяет обеспечить доступ к данным в формате отличном от формата их физического хранения. Это непрерывная последовательность адресов.

Обычно, файловая система не накладывает никаких ограничений на структуру данных файла и не интерпретирует их. Структурой данных управляет пользователь. Существуют разные форматы хранения данных которые известны только программам, которые создают файлы и работают с ними. Например, текстовые файлы формируются в виде последовательности символов, которые объединяются в строки произвольной длины (условно) и строка заканчивается специальным символом конца строки. Поэтому они и называются текстовыми, потому что для программиста это набор строк. Для работы с текстовыми файлами предназначены специальные программы - текстовые редакторы. Информация в текстовых файлах хранится посимвольно. Один символ - 1 байт. Можно написать программу, которая будет работать с любым текстовым файлом.

В операционных системах существует два типа файлов - байт-ориентированные и блок-ориентированные. В системе существует два типа устройств - символьные и блочныe.

В байт-ор. Файлах информация хранится по байтам, соответственно в байт устройствах передача данных по байтам.

Блок-ориентированные файлы, в них информация хранится в блоках одинакового размера.

В современных системах блок-ориентированных устройствами являются только диски.

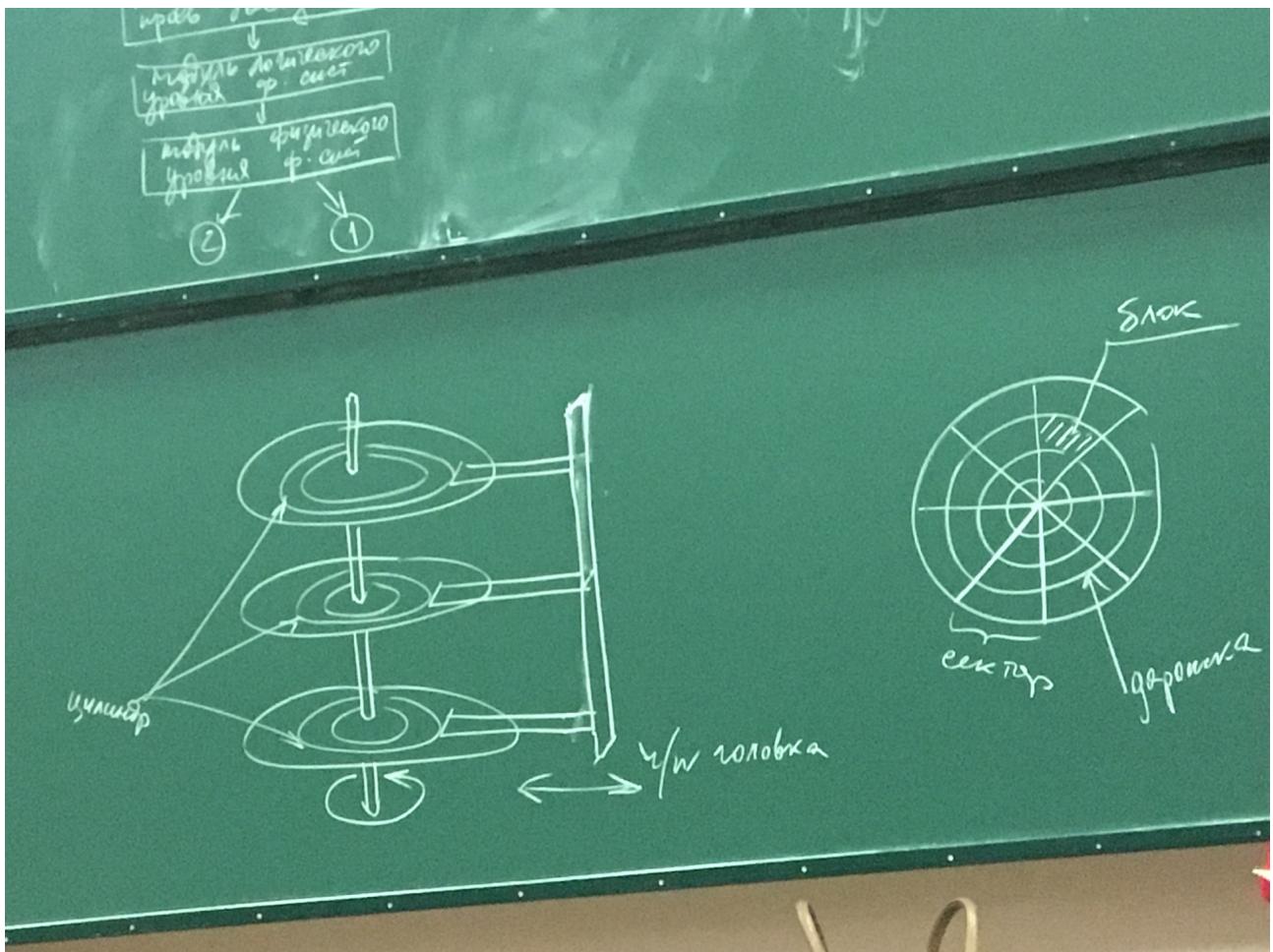
Соответственно, языки, такие как си и паскаль поддерживают такие файлы, в паскале это текстовые и типизированные, а в си это текстовые и бинарные.

Физический уровень хранения файлов.

БОЯН НАХУЙ, давайте глянем.

Наименьший адресуемый участок конкретной пластины определенной дорожки называется сектор. На пластинках концентрические дорожки. Совокупность одних и тех же дорожек разных пластин называется цилиндром.

Read-Write головка.



Если связное распределение, то файл на диске занимает непрерывное адресное пространство - непрерывные сектора.

Несвязное - сектора выделяются в разброс, но при этом система должна обеспечивать возможность адресовывать каждый отдельный блок, выделенный конкретно файлу. Позволяет более эффективно использовать адресное пространство диска.

Таким образом, данное представление о файловой системе позволяет обеспечить общий интерфейс для любой файловой системы. Все файловые системы могут быть описаны таким образом и именно это позволяет создать некоторый общий интерфейс для любого типа файловой системы. Это возможно потому, что ядро реализует слой абстракции над своим низкоуровневым интерфейсом. Именно этот подход используется в файловой подсистеме UNIX, которая реализует

интерфейс, названный VFS/vnode (Virtual File System/ virtual node).

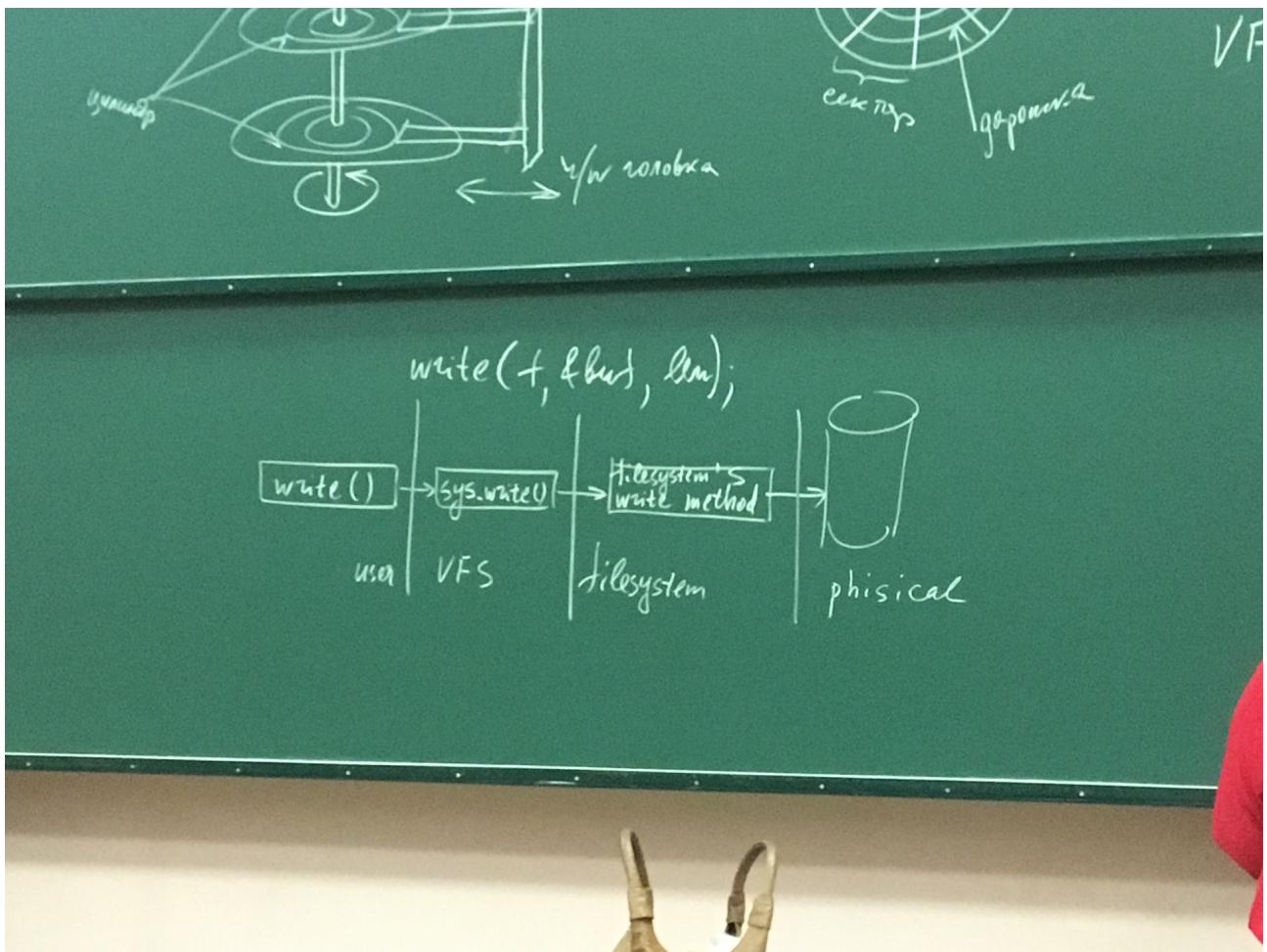
Linux изменил этот интерфейс, отказавшись от vnode, и называется VFS (Virtual File System.). VFS представляет общую файловую модель, которая способна отображать общие возможности и поведение любой мыслимой файловой системы.

Уровень абстракции VFS работает на основе базовые концептуальных интерфейсов и структур данных. Каждая отдельная файловая система определяет особенности того, как файл открывается, как выполняется обращение к файлу, как считается информация из файла и т. п.

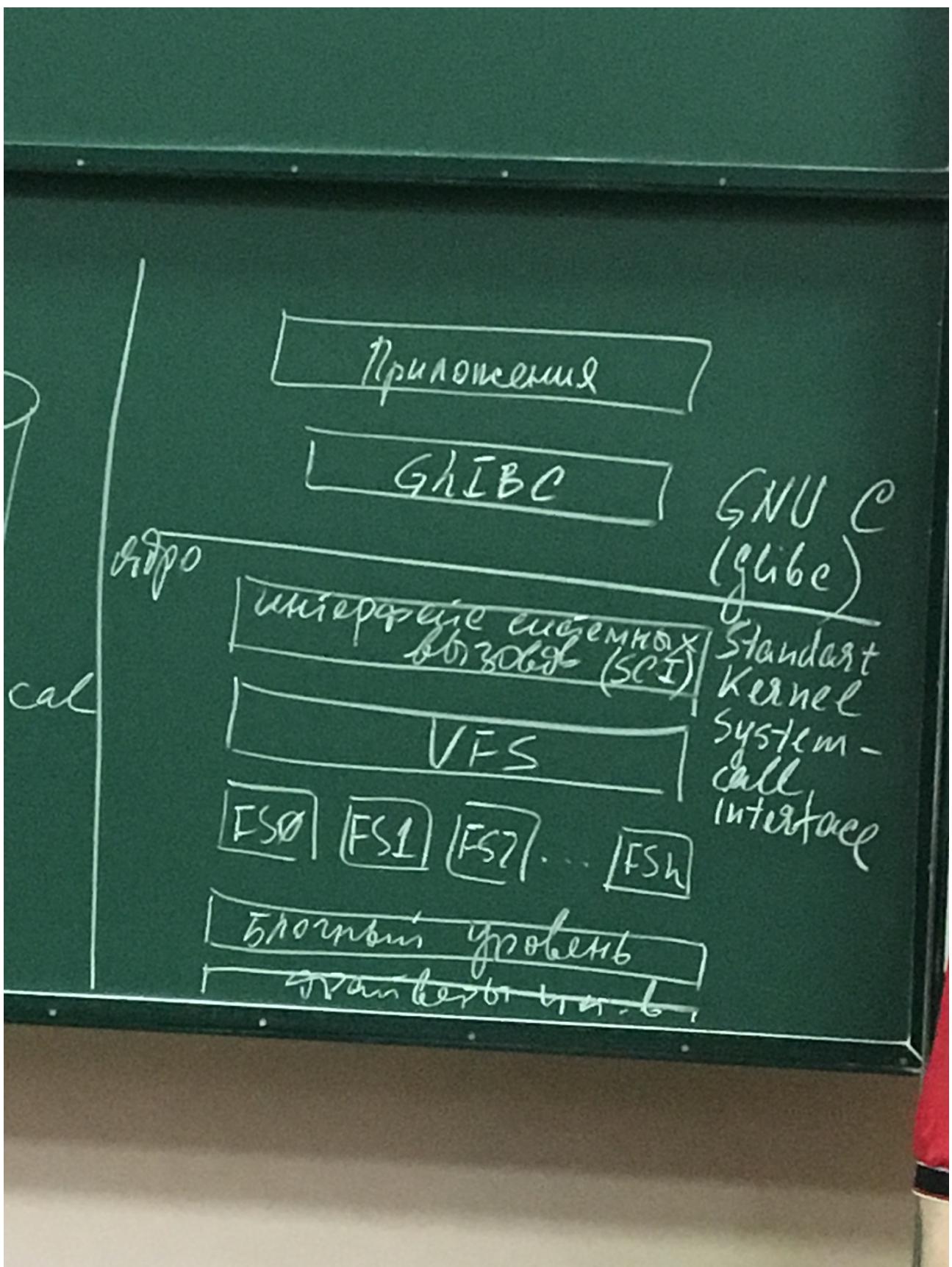
Фактический код файловых систем скрывает детали реализации, однако все фс поддерживают такие понятия как файлы, каталоги, поддерживают такие действия как создание файла, удаление файла, переименование, открытие файла, чтение/запись, закрытие файла и т. д.

Большинство файловых систем запрограммировано так, что API, которые предоставляют файловые системы рассматриваются как абстрактный интерфейс, который ожидаем и понятен VFS.

Например, рассмотрим запрос приложения `write(f, buf, len);` В данном случае данное API записывает `len` байт, которые хранятся в `buf`, т. е. У приложения есть адрес, в текущую позицию файла. Речь идет о логическом адресном пространстве файла. Этот системны вызов обрабатывается в системе по следующей цепочке:



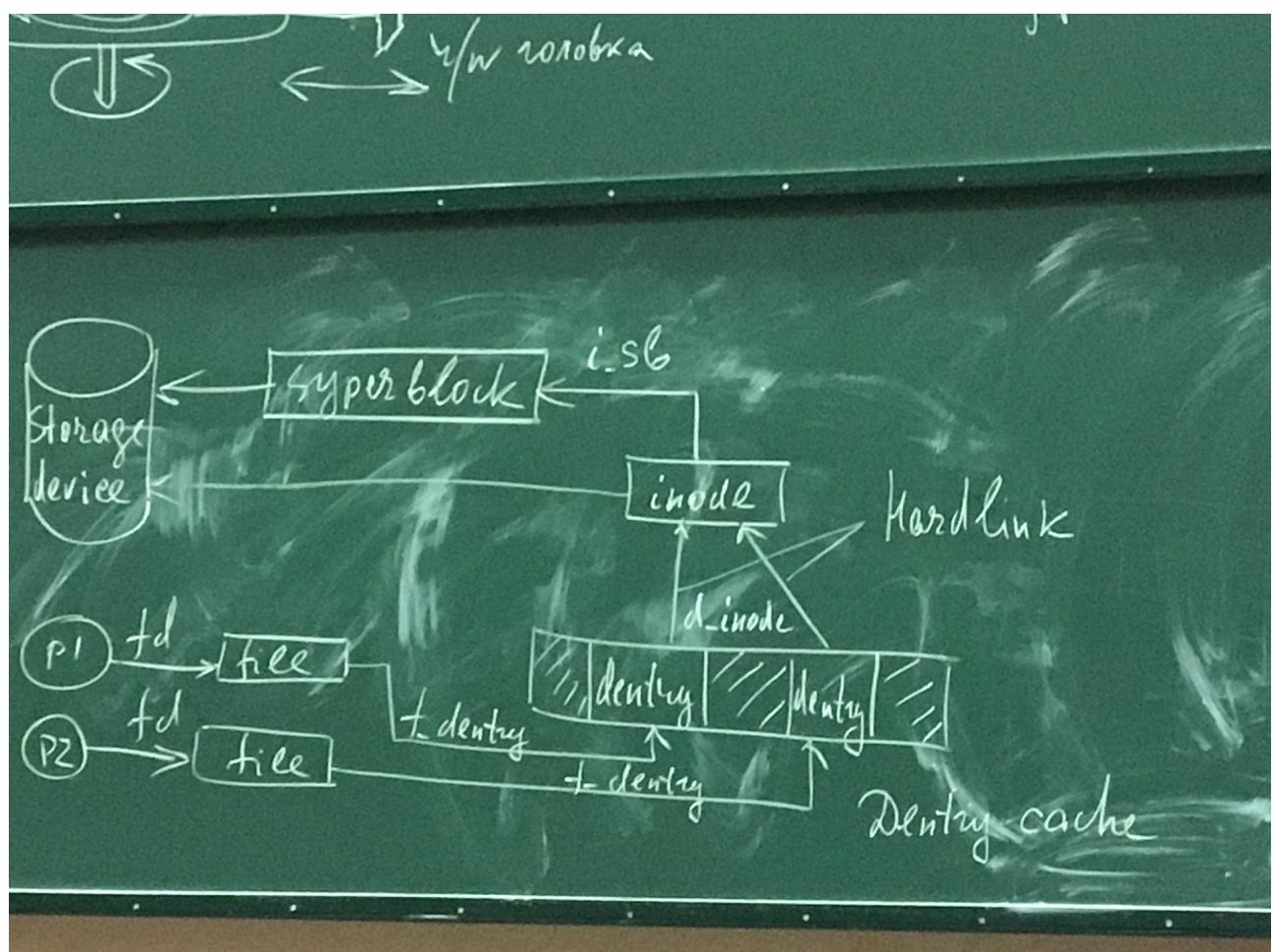
Системный вызов `write` сначала обрабатывается общим системным вызовом VFS `sys.write()`, затем, осуществляется вызов соответствующего системного вызова конкретной файловой системы, в данном случае, для записи данных в файл.



ФС, которые поддерживает UNIX/LINUX: EXT2, EXT3, UFS, NTFS, APFS, MS-DOS, FAT, FAT32, HPFS.

Внутренняя организация файловой системы. VFS базируется на 4х структурах:  
superblock, dentry (directory entry), inode (index node), file.

Безусловно, между этими структурами и объектами, которые описывают эти структуры, существует связь.



# Рязанова лекция. Файловая система UNIX/LINUX.

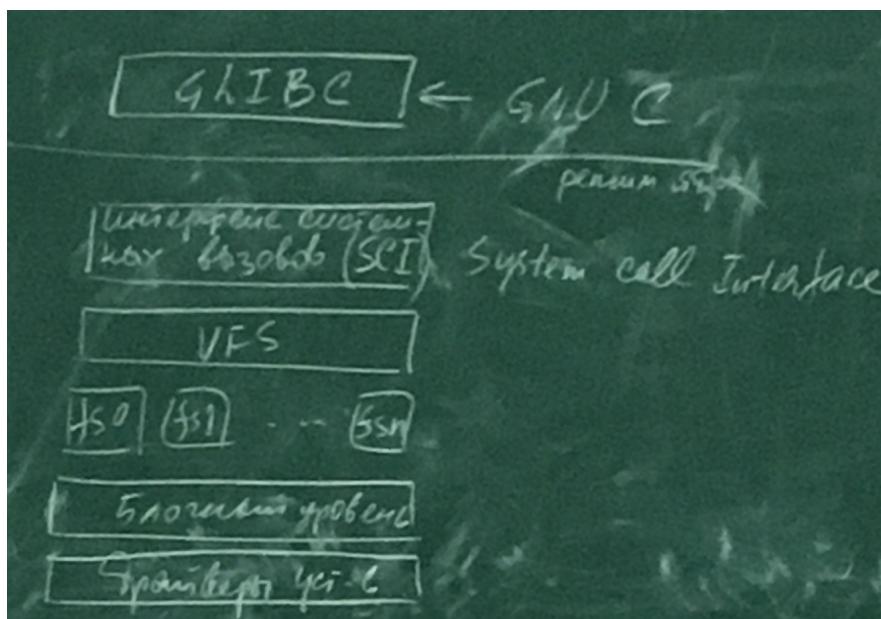
Всегда подчеркивают, что это многофайловая система. Для того, чтобы иметь возможность поддерживать такое большое количество фс в ос реализован специальный интерфейс. В UNIX он называется VFS/vnode,

в LINUX просто VFS.

LINUX не

декларируют vnode, виртуального узла в нем нет.

VFS представляет уровень абстракции, отделяющий POSIX API от подробностей работы конкретной файловой системы. Ключевой момент - системные вызовы, такие как open, read write работают



одинаково, независимо от того, какая фс располагается ниже. VFS представляет собой общую файловую модель, которую наследуют низлежащие файловые системы, фактически реализующие действия различных POSIX API.

В основе работы VFS лежат 4 базовые структуры - superblock, inode, запись каталога (dentry), файл.

## Superblock.

Суперблок содержит высокоуровневые метаданные о файловой системе. Задание - посмотреть в толковом словаре значение приставки мета. **Суперблок - это структура, которая содержит информацию, необходимую для монтирования и управления файловой системой.**

**Суперблок это структура, которая находится на диске, каждая файловая система имеет один суперблок, но на диске суперблок находится в нескольких экземплярах для надежности хранения данных, т. к. Суперблок есть ключевая структура файловой системы.**

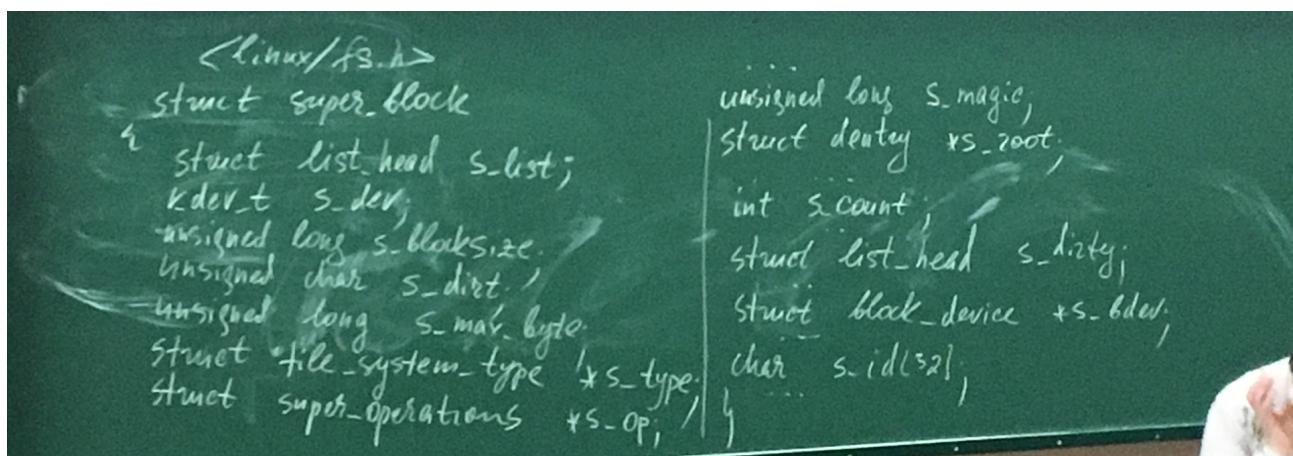
Сб, 23 марта

Очевидно, суммарное число блоков, свободное число блоков, корневой inode и тд.

Суперблок существует не только на диске. Он предоставляет системе информацию и файловой системе для ядра, чтобы ядро могло работать с фс. Суперблок в памяти предоставляет информацию, нужную для работы с монтированной фс.

```
// там где в описании интерфейса суперблока три точки упущено поле  
// struct list_head s_inodes; - это все inode'ы.
```

```
// struct dentry_operations *s_d_op - определяет dietary operations для  
директории
```



Поскольку линукс поддерживает одновременно большое количество фс, значит в системе будет соответствующее количество суперблоков, и они хранятся в списке `struct list_head`, т. е. Это список суперблоков.

`s_dev` - указывает устройство, на котором находится файловая система.

`blocksize` - размер блока в байтах.

- `s_dirty` - флаг, показывающий, что суперблок был изменен

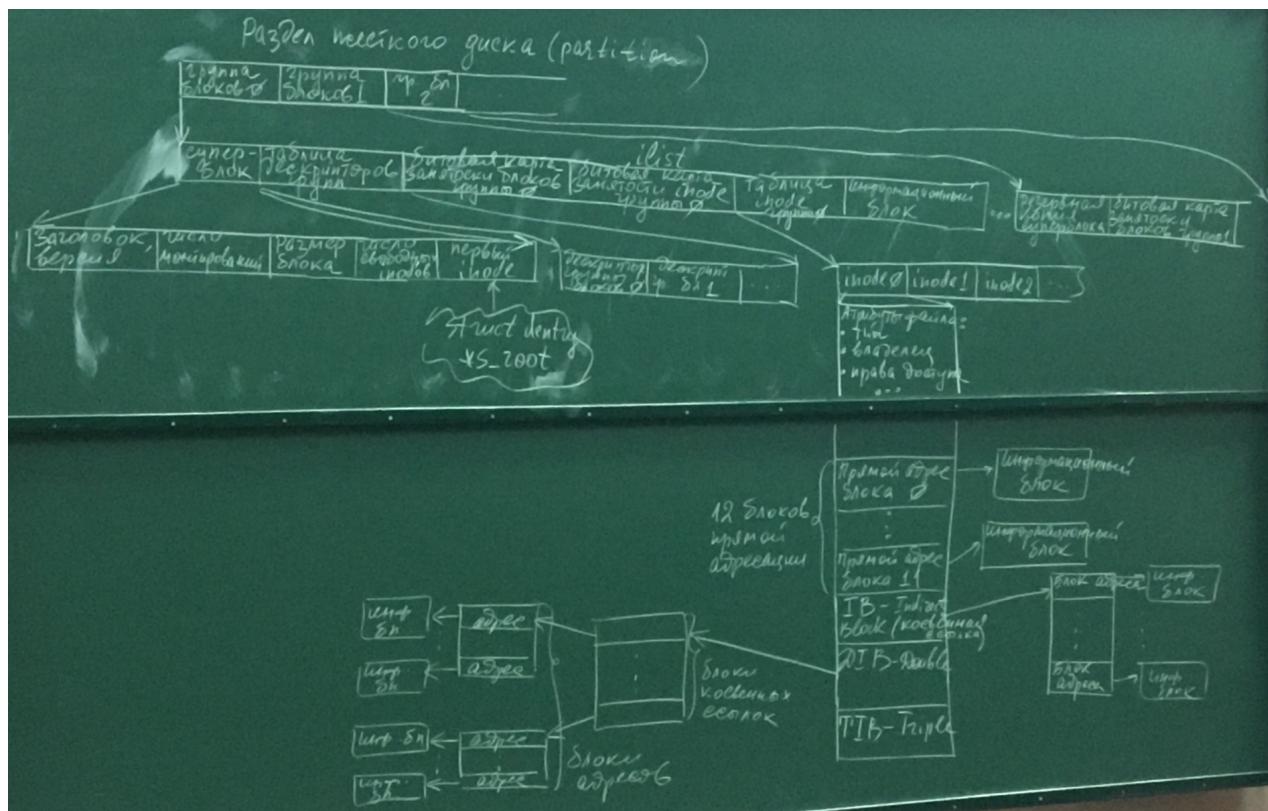
Суперблок описывает смонтированные фс, но каждая фс описывается структурой `struct file_system_type`. Это тип файловой системы, соответственно тип в системе может быть один. Мы можем смонтировать несколько фс ext, но тип у этих фс будет один.

- `struct super_operations` - действия, определенные на суперблоках.
- `s_magic` - магический номер файловой системы

Сб, 23 марта

- s\_root - точка монтирования файловой системы или каталог монтирования фс.
  - s\_count - счетчик ссылок на суперблок. (it appears to be the mounting count???)
  - list\_head s\_dirty - список измененных индексов.
  - Struct block\_device \*s\_bdev - драйвер соответствующего блочного устройства
  - s\_id - строка имени.

## **Раздел жесткого диска.**



Файлы в системе именуются номерами айнодов. Чтобы разделять адресное пространство диска, надо иметь информацию, которую эффективнее всего хранить в битовой карте - о занятых и свободных айнодах.

В соответствии с описанной структурой суперблок, что в суперблоке хранится информация, а в частности struct dentry\_root - это указатель на корневой каталог. **В системе LINUX все файл. ВСЕ ФАЙЛЫ В СИСТЕМЕ ИМЕЮТ АЙНОДЫ.**

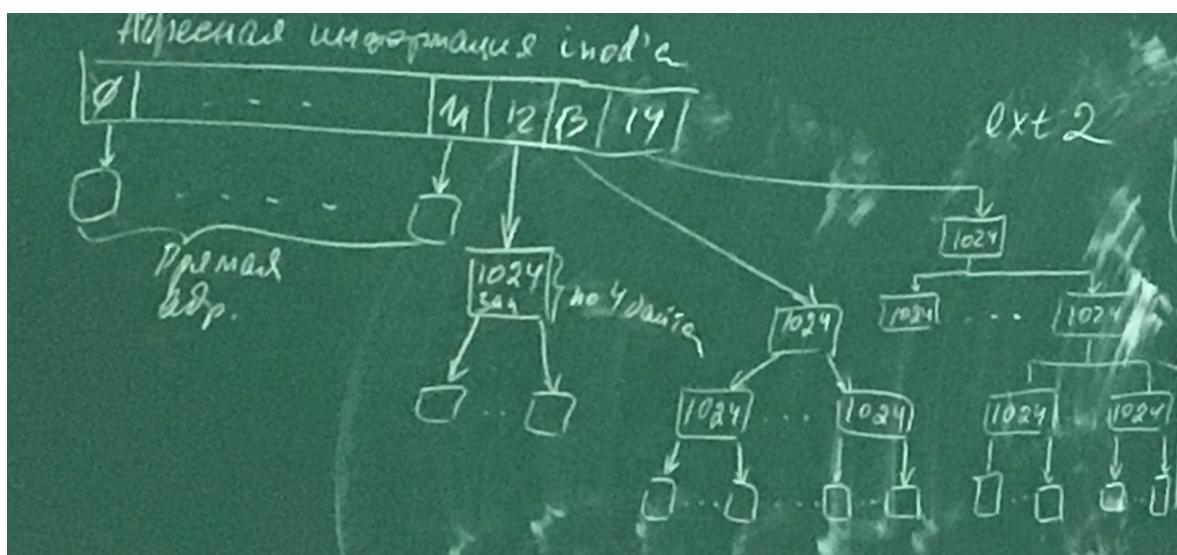
Сб, 23 марта

Struct dentry (directory entry) создается на лету, создается на основе информации, которая хранится в айнодах. Файлы ищутся по их полным именам, то есть сначала рассматривается путь к файлу, то есть все составляющие пути к файлу должны быть просмотрены. Dentry кэшируются.

**Дисковый inode хранит информацию о физическом файле и позволяет получить доступ к информации, записанной в файле. LINUX поддерживает файлы очень больших размеров.**

Блок косвенной адресации ссылается на блок в котором хранятся адреса блоков по тому же принципу.

Double Indirect Block - двойная косвенная адресация, дальше тройная косвенная.



Struct super\_operations - операции, определенные на суперблоке.

Struct Super\_operations - linux/fs.h

↳ struct inode *(alloc_inode)(struct super_block *sb), void (*destroy_inode)(struct inode *), void (*dirty_inode)(struct inode *, int flags), int (*write_inode)(struct inode *, struct writeback_control *wbc), int (*drop_inode)(struct inode *), void (*put_super)(struct super_block *);	int (*reize_super)(struct super_block *), int (*remount_fs)(struct super_block * sb, int flags),
---	---

Сб, 23 марта

dirty\_inode - функция вызывается, когда в файл вносятся изменения.

Журналируемые фс, например, EXT3 используют эту функцию для обновления журнала.

write\_inode - записывает inode на диск и одновременно помечает его как грязный

drop\_inode - сбрасывает inode.

Функция drop\_inode вызывается, когда исчезает последняя ссылка на индекс, и vfs ее просто удаляет.

Когда файловая система нуждается в выполнении операции над суперблоком, то она следует за указателем на желаемый метод объекта суперблока.

Например

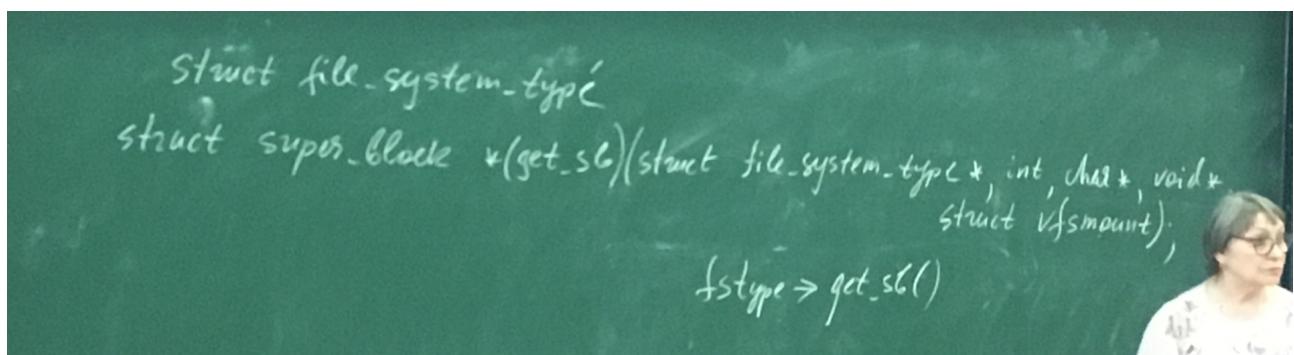
sb -> s\_op -> put\_super(sb);

Код для создания управления и ликвидации объектов суперблока находится в файле fs/super.c

Объект суперблок создается и инициализируется функцией alloc\_super(). Эта функция выделяет и инициализирует новую структуру суперблока, и возвращает указатель на новый суперблок, и null если выделение суперблока завершилось аварийно.

Суперблок описывает информацию, которая нужна системе чтобы управлять смонтированной фс. Любая фс описывается структурой file\_system\_type. В этой структуре есть поле

Struct super\_block \*(get\_sb)



Сб, 23 марта

Эта функция вызывается вовремя монтирования файловой системы.  
Ядро называет эту функцию <нижняя строчка фотки>

И эта функция инициализирует поля данных и устанавливает суперблок

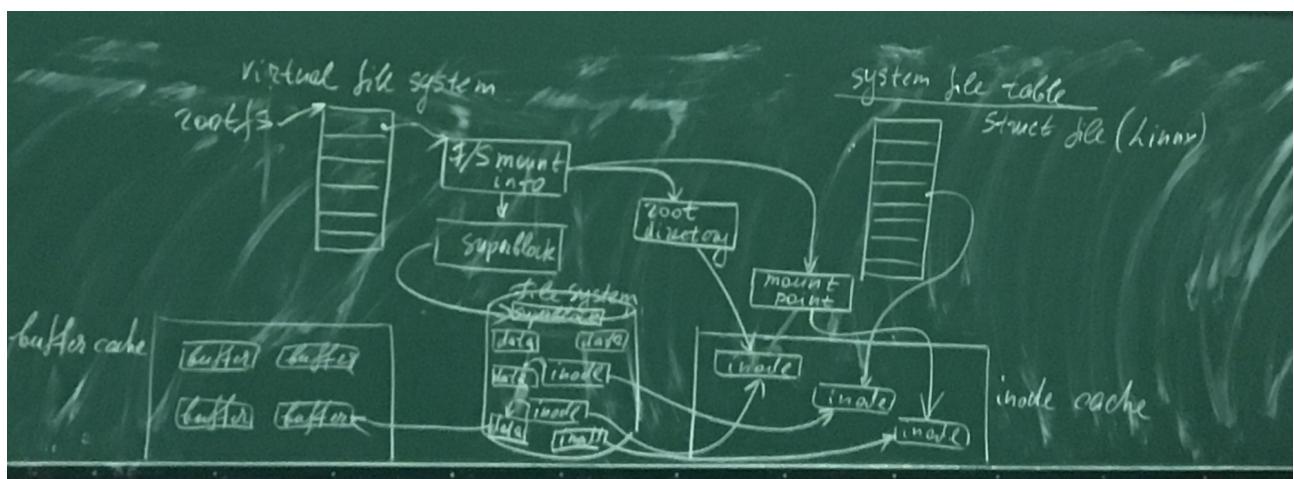
# Рязанова, лекция. Продолжение. Файловая система UNIX/LINUX.

Поле struct super\_operations - структура, которая определяет действия, которые мы можем выполнять над суперблоками.

Имеется также ссылка на file\_system\_type, она определяет тип файловой системы, в файловой системе может быть только один тип конкретной файловой системы, в VFS может существовать только одна структура file\_system\_type, но систем, которые могут иметь такой тип может быть сколько угодно.

Очевидно, что если не предпринять определенных действий в системе, а каждый раз обращаться к диску, то время получения инфы в файлах и о файлах будет значительным, потому что диск это внешнее устройство, медленно-действующее с точки зрения быстродействия системы.

Информация о файлах кэшируется, иначе доступ к файлам будет слишком долгий.



Все в системе есть файл, следовательно у всего есть inode.

Struct file описывает открытые файлы. Файл существует в двух эпостасях - файл, который лежит на диске, и мы можем посмотреть, что он есть.

30 марта, сб

В системе существует одна таблица всех открытых файлов - struct file table. Открыть файл значит обратиться к файлу на диске, эта структура позволяет организовать работу с открытым файлом. **А сам файл описывается inode'ом. Inode обеспечивает доступ к данным, которые находятся на физическом носителе.**

Чтобы обеспечить производительность обращения к файлам, для того чтобы процесс обращения к файлам не занимал слишком много времени, вся информация кэшируется. В значительно упрощенном виде происходит все кэширование.

Файловые структуры должны быть в ядре резидентно (постоянно). В inode кэше сохраняют соответствующие inode, данные и тд. Кроме того, эти данные сохраняются в различных таблицах.

File\_system\_type:

(Роберт Лав - про разработку ядра, в сети есть инфа из этой книжки Зе издание 2012, структуры, которые приводятся в этой книге устаревшие.)

get\_sb() - ее название теперь mount(). Она вызывается когда выполняется монтирование фс, которое происходит из строки. При монтировании файловой системы создается суперблок. Мы определяем поля, определяем super\_operations, это все должно быть описано, но все это начинает выполняться, когда мы выполняем монтирование, т е на самом деле это точка входа. Мы пишем модуль ядра, в котором описываем тип фс, описываем свою функцию mount, но срабатывает она когда мы монтируем эту фс. Инициализируются соответствующие поля struct super\_block. Например, инициализируются обращения к операциям на суперблоке. В результате будут заполнены соответствующие поля, связанные с конкретными физическими файлами. И уже можно начинать работу с файлами, доступ к которым обеспечивает данная фс.

## **Struct inode. В UNIX/LINUX все доступно как файл. ВСЕ НАХОДЯЩИЕСЯ ФАЙЛЫ.**

Такие устройства как жесткий диск, как оптические диски, флешки системой рассматриваются как файлы. В директории def мы видим перечень устройств, но для системы это файл.

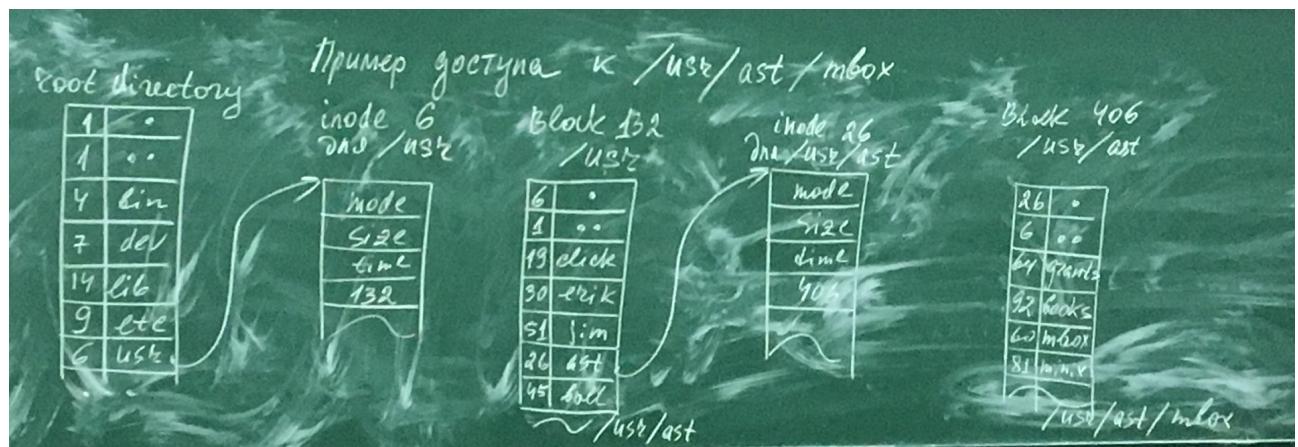
Еще раз повторим, что UNIX/LINUX поддерживают символьные имена в виде символьных строк. Такое именование файла удобно для пользователя. Имя файла в UNIX/LINUX в их родных файловых

30 марта, сб

подсистемах (EXT2) не является идентификатором. На самом деле идентификатором файла является номер inode, который принято называть метаданными.

Система, для того, чтобы получить доступ к файлу ищет его номер inode в таблице, которая называется таблицей inode'ов.

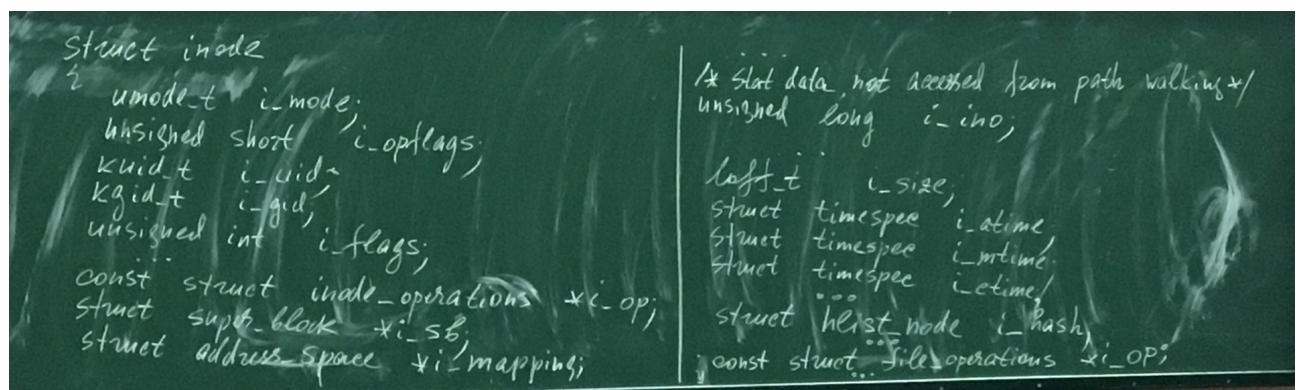
Давайте те же, дети мои, обратим очи на сию *МАНУЮ* схему:



Тут показывается доступ, начиная с корневого каталога.

В системе существует структура dentry, которая создается, когда выполняется обращение к файлам/директориям. Очевидно, что каждый раз просматривая путь, мы фактически спускаемся по пути начиная с корневого каталога.

## Структура inode:



30 марта, сб

Мы видим, что в этой структуре присутствует указатель на struct inode\_operations

В этой же структуре находится информация об устройствах.

### struct inode\_operations:

The diagram shows the definition of the struct inode\_operations. It starts with the keyword 'struct' followed by the identifier 'inode\_operations'. A brace '{' indicates the start of the structure body. Inside, there is a list of function pointers, each preceded by 'int (\*' and followed by ')'. The functions listed are: lookup, get\_link, readlink, create, link, unlink, and symlink. To the right of the list, there are seven corresponding function names enclosed in brackets: 'int (\*lookup)(...)', 'int (\*get\_link)(...)', 'int (\*readlink)(...)', 'int (\*create)(...)', 'int (\*link)(...)', 'int (\*unlink)(...)', and 'int (\*symlink)(...)'. A vertical line connects the brace '{' to the first function name, and another vertical line connects the last function name to the closing brace '}'.

```
struct inode_operations
{
    struct dentry *(*lookup)(struct inode *, struct dentry *, unsigned int);
    void *(*get_link)(struct dentry *, struct nameidata *);
    int (*readlink)(struct dentry *, char __user *, int);
    int (*create)(struct inode *, struct dentry *, char __user *, int);
    int (*link)(struct dentry *, struct inode *, struct dentry *, umode_t, bool);
    int (*unlink)(...);
    int (*symlink)(...);
}
```

link - это операция, связанная с inode'ом, в частности можно рассмотреть, что делается в системе, когда вызывается link.

Любой системный вызов связан с последовательным вызовом других функций.

# Рязанова, лекция.

## Struct dentry.

В отличие от объектов superblock и inode, dentry это конкретная структура, выделенная на directory entry, не сопоставляется ни с какой структурой на диске. Виртуальная фс создает этот объект как говорят на лету (by fly) из строкового представления пути к файлу (pathname).

Dentry, как и все структуры ядра, претерпевает изменения. Эта структура работает с именами файлов.

Каждая директория, или каждая поддиректория в UNIX/LINUX это файл, специальный файл. То есть интерфейс VFS представляет каталоги как файлы. Но это специальные файлы, на которых определены специальные действия, и у них есть свои задачи, главная из которых обеспечить возможность по символьному имени файла обеспечить доступ к физическому файлу. Но обратиться к файлу мы можем по номеру inode'a, но для человека удобнее использовать какие-то названия (имена) - строка символов, которая разбирается, но по этому пути в конечном итоге надо получить номер inode'a. То есть объект dentry - это определенный компонент пути (к файлу). Причем, все объекты dentry это компоненты (элементы) пути, включая обычные файлы, которые могут включать в себя точки монтирования. Поскольку объекты элементов каталога не хранятся на физическом носителе, например на диске, структура struct dentry не имеет флагов, которые указывали бы на то, что объект был изменен (dirty) потому, что объект dentry не надо переписывать на диск, а отредактированный inode надо, если сопоставить с объектом inode.

6 апреля, сб

## Рассмотрим структуру struct dentry:

Замечание: это код из ядра версии 3.14

**RCU** - специальный механизм синхронизации (read-copy-update), который обеспечивает монопольный доступ. Синхронизация предполагает, что какой-то процесс заинтересован в действиях другого процесса (сформирует какое-то сообщение, получив которое, процесс сможет продолжить свое выполнение).

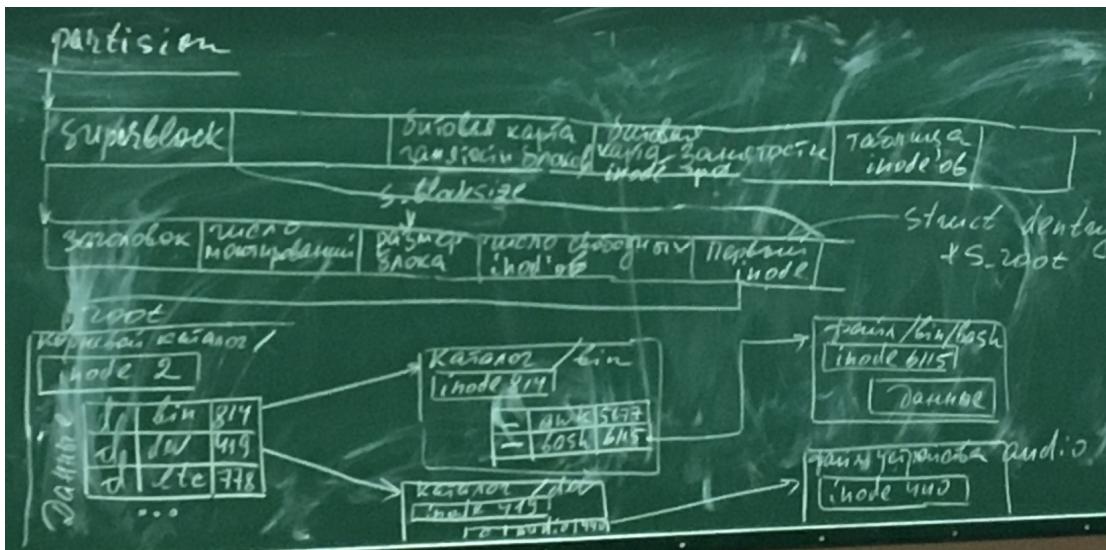
## STRUCT DENTRY:

```
Struct dentry
{
    /* RCU lookup touched fields */
    unsigned int d_flags; /* protected by d_lock */
    seqcount_t d_seq; /* per dentry seqlock */
    struct hlist_node d_hash; /* list of hash table entries */
    struct qstr *d_parent;
    struct inode *d_inode; /* dentry name */
    unsigned char d_iname[DNAME_INLINE_LEN]; /* короткое имя */

    /* struct dentry_operations */
    struct super_block *d_sb; /* корневой каталог (root) списка каталогов */
    unsigned long d_time;
    void *d_fsdata; /* данные слуз. для fs */
    struct list_head d_lru; /* LRU */
    struct list_head d_subdirs;
}
```

Актуальный объект dentry может находиться в одном из трех состояний - used, unused, negative (используемый, неиспользуемый, противоречивый).

Имеется указатель на корневой каталог (struct super\_block \*d\_sb).



Dentry кэшируются.

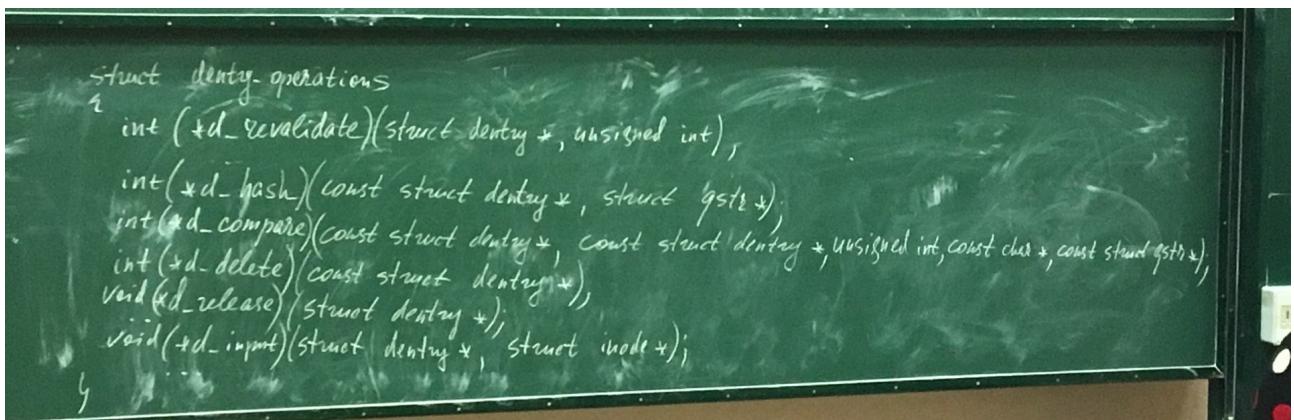
После того, как виртуальная фс прошла по пути (спустилась по пути (полному имени файла)), и для каждого пути создан объект dentry (каждый путь пройдет от начала до конца), то это очень большая работа.

Очевидно, что в системе эту работу не выбрасывают. Вместо этого полученную в результате разбора информацию сохраняется в dentry кэш, который состоит из 3х частей - список используемых объектов dentry, которые с inode'ом полем d\_inode. Т к отдельный inode может иметь много линков, то может быть много объектов dentry, связанных с этим inode, корреспондирующих этот inode. Второй - двусвязный список list recently used - список неиспользуемых объектов dentry. Соответственно удаление элементов из данного списка выполняется по алгоритму lru.

Третий список - hash table - кэширует функции для быстрого определения заданного пути в ассоциированный объект dentry. Hash table реализована в виде hash\_dentrytable массива. Каждый элемент является указателем на список dentries, хэшированных по тому же самому имени.

6 апреля, сб

## **DENTRY OPERATIONS:**



d\_revalidate - эта функция работает, если заданный dentry объект существует. Виртуальная фс вызывает ее, чтобы подготовить dentry в хэше. Большинство фс устанавливают его в 0, так как они всегда доступны.

d\_hash - создает значение хэш, величину хэш для заданного объекта dentry. VFS вызывает эту функцию для того, чтобы добавить эту функцию в hash\_tripple

d\_compare - сравнивает два имени файла. Большинство файловых систем определяют ее как found, впизду.

## **Inode cache.**

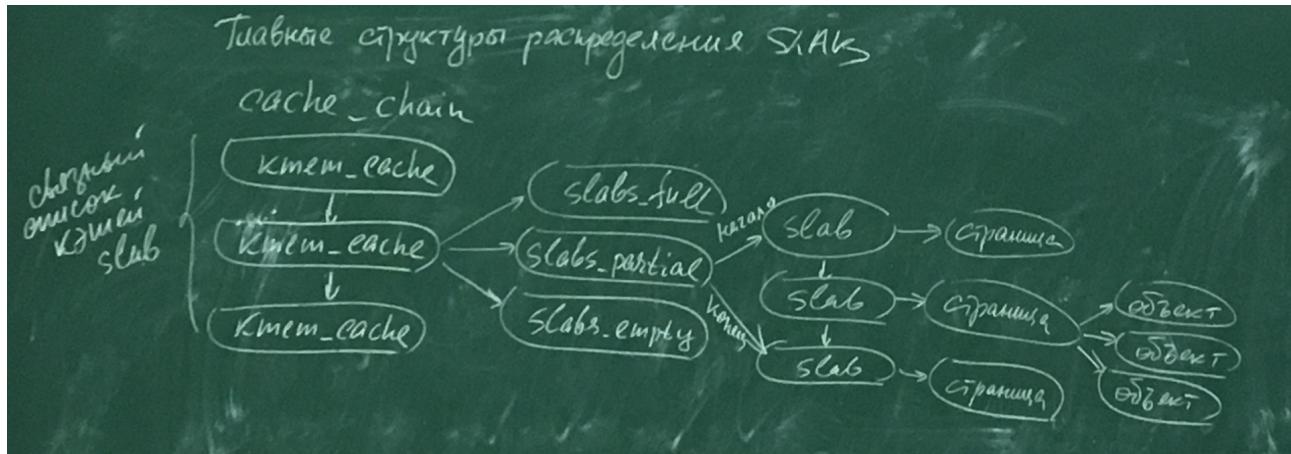
Inode кэш представляет из себя:

1. Глобальный хэш-массив inode\_hashtable, в котором каждый inode хэшируется по значению указателя на суперблок и 32х разрядному номеру inode. При отсутствии суперблока, inode добавляется к списку анонимных inode'ов. Примером таких анонимных inode'ов могут служить сокеты, вызванные sock\_alloc().
2. Глобальный список inode\_in\_use. В этом списке содержатся допустимые inode'ы (с i\_count > 0 и i\_nlink > 0). В этот же список записываются вновь созданные inode'ы (объекты inode).
3. Глобальный список inode\_unused, который содержит допустимые inode'ы с i\_count равным нулю.
4. Список для каждого суперблока (sb -> s\_dirty), которые содержат inode'ы с i\_count > 0 и i\_nlink > 0 и состоянием i\_dirty, то есть измененные inode'ы
5. Флаг кэш inode'a SLAB cache inode\_cachep. Объекты inode могут освобождаться вставляться и изыматься из SLAB кэшей. Любой

6 апреля, сб

объект inode может находиться только в одном из этих списков. SLAB был введен Джефом Бонвиком для операционной системы SunOS. (SLAB аллокатор) - это кэширующий аллокатор, позволяющий выделять блоки памяти одного и того же размера, но для данных одного и того же типа. SLAB распределение стало использоваться для того, чтобы упростить управление памятью и выделение памяти. Дело в том, что в ядре значительные объемы памяти выделяются на ограниченный набор объектов таких как дескрипторы файлов, inode'ы и т. п. Идея Бонвики базируется на том, что количество времени, нужное для инициализации обычного объекта в ядре, превышает время, которое необходимо для его выделения и освобождения. Вместо того, чтобы возвращать память системе, оставлять ее в проинициализированном состоянии, рассчитывая на то, что в будущем она будет использована для тех же целей. Если надо выделить участок памяти на объект inode, то будем юзать уже выделенную. Например, если память выделена для mutex, то функцию init\_mutex надо выполнить только один раз, когда память выделяется впервые. Последующие распределения памяти под mutex не нужны, так как она уже выделена в результате инициализации и последующего освобождения.

## Главные структуры распределения SLAB



Алгоритм best-fit - самый подходящий.

Сб, 13 апреля

# Рязанова, лекция 5.

Очевидно, что если постоянно обращаться к диску, то это очень большие временные затраты, что приводит к торможению приложений. Разработчики системы учитывают эти моменты - кэширование.

В силу того, что SLAB кэширование построено на принципе хранения данных одного размера, доступ к этим данным ускоряется.

В случае распределения SLAB участки памяти, подходящие для размещения объектов данных определенного типа и размера определены заранее. Распределитель (allocator) SLAB хранит информацию о размещении этих участков, которые известны также как кэши. Очевидно, что за счет того, что выделяются блоки памяти одного размера, ускоряется выделение памяти. SLAB аллокатор запрашивает у ОС большие участки памяти и выделяет из них небольшие участки по запросу. Вследствие этого обращение к менеджеру памяти выполняется реже, а запросы пользователя удовлетворяются быстрее. Но самым главным выигрышем является не скорость выделения памяти при использовании SLAB'ов, а сокращение времени на инициализацию такой памяти. Это связано с тем, что аллокатор часто используется для выделения объектов одного и того же типа, что позволяет пропустить инициализацию некоторых полей структур при повторном выделении такого участка памяти. Например, мьютексы, спинлоки, inode'ы и т. п. при освобождении объекта, скорее всего, имеют правильное значение. В силу этого при повторном выделении не нуждаются повторные инициализации части полей.

В настоящее время LINUX имеет три вида SLAB аллокаторов: SLAB, SLUB, SLOB. У них есть различия, но интерфейс у них один и тот же.

Рассмотрим некоторые функции SLAB аллокатора:

Эти функции применены для inode'ов

```
int aufs_inode_cache_create(void)
{
    aufs_inode_cache = kmem_cache_create("aufs_inode", sizeof(struct aufs_inode), 0, SLAB_RECLAIM_ACCOUNT | SLAB_MEM_SPREAD, aufs_inode_init_one);
    if (aufs_inode_cache == NULL) return -ENOMEM;
    return 0;
}

struct kmem_cache *kmem_cache_create(const char *name, size_t size, size_t align,
                                     unsigned long flags, void (*ctor)(void *vaddr))
{
    struct kmem_cache *cache;
    unsigned long flags, void (*ctor)(void *vaddr);
    void (*ctor)(void *vaddr);
}
```

Сб, 13 апреля

const char \* name - имя кэша которое используется виртуальной файловой системой proc, которые можно посмотреть в /proc/slabinfo

sizeof(struct aufs\_inode) - размер объекта

size\_t align - выравнивание

Это callback функции, которые пишутся разработчиком

Функция kmem\_cache\_create не выделяет память кэшу

```
void aufs_inode_cache_destroy(void)
{
    rcu_barrier();
    kmem_cache_destroy(aufs_inode_cache);
    aufs_inode_cache = NULL;
}

void kmem_cache_destroy(struct kmem_cache *cache);
```

rcu\_barrier - распространенный механизм синхронизации, он поставлен для безопасного освобождения памяти, для так называемых lock-free алгоритмов

k\_mem\_destroy удаляет SLAB аллокатор

rcu\_barrier - блокирующая функция, ждет пока все отложенные действия над защищенными rcu данными завершатся, после этого возвращает управление коду, который ее вызвал.

```
struct inode *aufs_inode_alloc(struct super_block *sb)
{
    struct aufs_inode *const i = (struct aufs_inode *) kmem_cache_alloc(aufs_inode_cache,
        GFP_KERNEL);
    if (!i) return NULL;
    i->ai_inode = NULL;
    void *kmem_cache_alloc(struct kmem_cache *cache, gfp_t flags),
        flag указывает надо брать память из RAM сразу
```

Функция возвращает объект из кэша. Если кэш пуст, то функция может вызвать функцию cache\_alloc\_refill(), чтобы добавить в кэш память

```
void aufs_inode_free(struct inode *inode)
{
    call_rcu(&inode->i_rcu, aufs_free_callback);
}
```

Освобождение  
inode'a с  
использованием rcu.

Блокировки это зло.

Проблема lock-free  
алгоритмов  
заключается в том,

что без блокировок нет гарантий, что мы получим правильный результат, так как у нас нет гарантий, что этот же объект не используется каким-то другим параллельным потоком выполнения. Поэтому, lock-free алгоритмам приходится заботиться о безопасном освобождении памяти. Именно такие средства предоставляет rcu.

Rcu-call откладывает выполнение функции aufs\_free\_callback

```
#cat /proc/slabinfo
...
inode_cache 7005 14792 480 1598 1849 1
dentry_cache 5469 5880 128 183 196 1
```

Описание числовых полей на запись выше слева направо.  
Число активных объектов, общее число объектов, доступные объекты, размер каждого объекта в байтах, число страниц с относящихся по крайней мере к одному объекту, количество страниц, выделенных слабу.

---

Сб, 13 апреля

## Прерывания.

Шоу, “Логическое проектирование операционных систем”, там указана

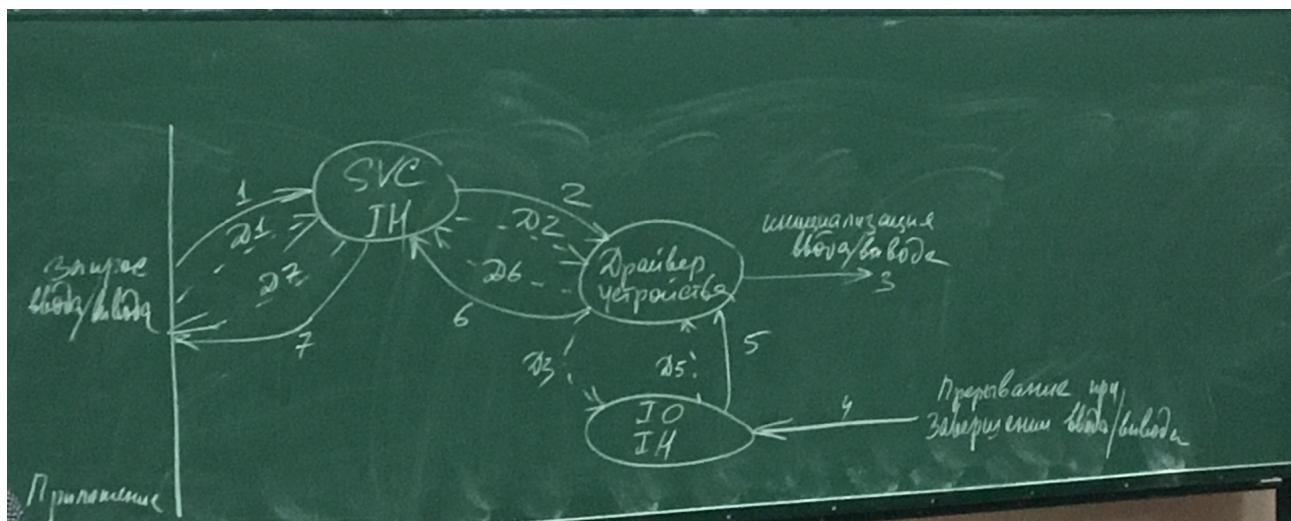


схема прерываний, приведенная ниже.

SVC - supervisor call, операционная система с стадии выполнения  
IH - interrupt handler

При обработке ввода/вывода система переводится в режим ядра. Ни одна ос не позволяет приложениям обращаться к устройствам ввода/вывода напрямую. В противном случае, такую систему нельзя было бы защитить.

Если мы вызываем какую-то библиотечную функцию ввода вывода, то в ее тексте обязательно будет системный вызов, выполняя которую система перейдет в режим ядра и начнет выполняться код ядра. Процесс в синхронном блокирующем вводе/выводе будет блокирован.

Речь в данном случае идет о синхронном блокирующем вводе/выводе. Блокировка процесса - непростой процесс для системы. Из диаграммы видно, что действуются разные части кода ядра системы.

Безусловно, операционная система имеет соответствующие модули, которые часто называются подсистемой ввода/вывода. Эта подсистема определяет какой драйвер использовать, чтобы дать данные(?) в понятном виде для пользователя.

На каждом этапе данные преформатируются, то есть они получают определенный вид. Драйвер получает определенный формат данных, который должен будет преобразовать команду, которую получит контроллер устройства, который начнет управление устройством по

Сб, 13 апреля

получении этой команды. При получении сигнала прерывания по окончанию ввода вывода. Соответственно, сигнал прерывания, который приходит на контроллер может быть замаскирован, то такой сигнал будет игнорироваться. Если не маскирован, то контроллер сформирует сигнал прерывания, который поступит на один из входов процессора. В конце цикла выполнения каждой команды, контроллер проверяет наличие сигнала прерывания на этой своей выделенной ножке (входе). Если сигнал поступил, то процессор переходит на выполнение обработчика прерывания и это обработчик аппаратного прерывания. IO IH на схеме. Все такие прерывания приходят от устройств - аппаратные, плюс прерывания от системного таймера тоже аппаратные. Все аппаратные прерывания выполняются на очень высоком уровне приоритетов, фактически когда они выполняются никакая другая работа в системе выполняться не может, пока работа прерывания не будет завершена. С точки зрения системы это нехорошо, поэтому аппаратное прерывание делится на две части. В UNIX/LINUX их принято называть **top half** и **bottom half** - верхние и нижние половины. В Windows эта же идея реализована с помощью dpc different procedure call - отложенный вызов процедуры.

Аппаратные прерывания принято делить на быстрые и медленные. Быстрое не делится на две половины, выполняется как одно целое. В современном LINUX быстрым прерыванием являются только пребывания от системного таймера. Нижняя половина прерывания выполняется в UNIX/LINUX как отложенное действие. То есть, фактически аппаратным прерыванием является верхняя половина, задачи которой следующие: получение данных из регистра устройства и помещение их в буфер ядра. Аппаратное прерывание, как правило, завершается инициализацией отложенных действий, например путем установки соответствующей нижней половины в очередь на выполнение, для этого например вызывается функция *scheduler* (планирования). Если вернуться к диаграмме выше, то в любом случае, или команда чтения, или записи, все равно необходимо передать информацию об успешном завершении операции вывода.

Соответственно в любом случае процесс должен получить информацию об успешном завершении операции. 5 стрелка и стрелка D5 с помощью соответствующих функций драйвера, драйвер должен инициировать передачу данных из буфера ядра в буфер приложения, а для этого процесс должен быть разблокирован. Обеспечение ввода вывода - одна из самых сложных задач, которые решает операционная система.

# Рязанова, лекция #6

## Аппаратные прерывания.

Понятно, что для разработчика имеют значение функции, которые вызываются, но также важно понимать особенности этого процесса, процесса завершения получения данных от устройства, аппаратные прерывания от устройств приходят на контроллер прерываний в итоге контроллер формирует сигнал прерывания, поступающий на процессор и процессор начинает обрабатывать это прерывание.

Механизм аппаратных прерываний - это естественный асинхронный параллелизм.

Общая модель обработки аппаратных прерывания является принципиально-архитектурно-зависимой. Например, в MS-DOS использовался контроллер прерываний peek, из которого формировался вектор прерывания для адресации смещения по схеме 4 байта. Номер аппаратного прерывания использовался в качестве смещения в таблице векторов, которая начиналась с нуля. В 32-разрядных системах использовался apeek, который формировал вектор прерывания, который является смещением к 8 байтовому дескриптору прерывания в таблице дескрипторов пребываний (IDT).

Основная функция аппаратных прерываний заключается в том, чтобы позволить периферийным устройствам взаимодействовать с процессором, чтобы информировать его о завершении некоторых задач, или о возникновении ошибочных ситуаций или каких-то других событий, которые требуют внимания со стороны системы. В результате возникновения прерываний, используя соответствующую схему адресации обработчиков прерываний, такой обработчик от конкретного устройства начинает выполняться. Очень часто такой обработчик называют (ISR) Interrupt Service R???. Обработчик прерывания выполняется в режиме ядра в системном контексте, так как прерванный процесс, обычно, не имеет никакого отношения к возникшему в системе пребыванию, его обработчик не должен обращаться к контексту процесса. По этой причине он не обладает правом блокировки. Однако, прерывание оказывает некоторое влияние на выполнение текущего процесса. Время, потраченное на обработку прерывания, является частью кванта, выделенного процессу. Например, обработчик прерывания от системного таймера использует

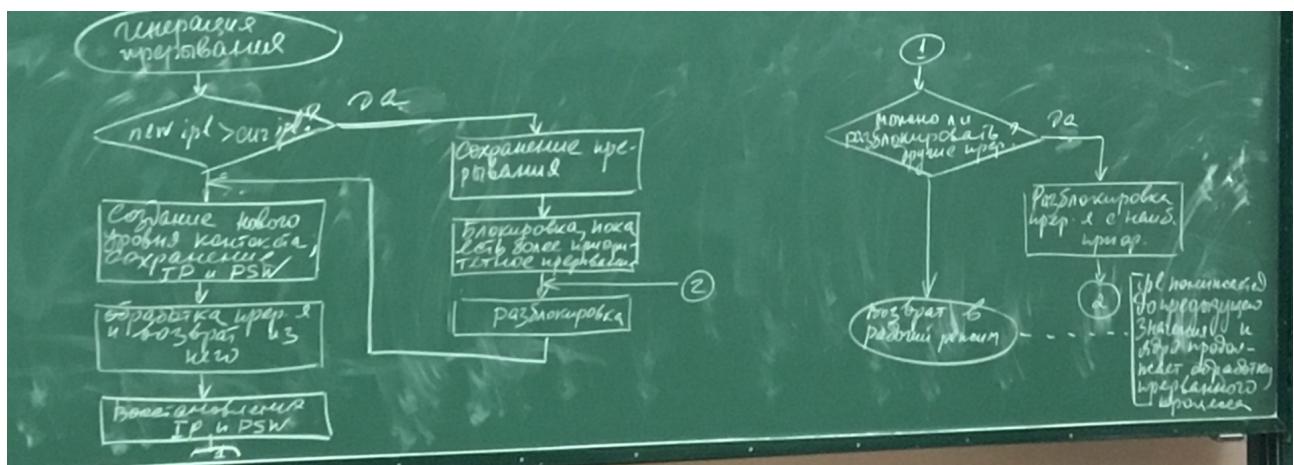
тиki текущего процесса, и поэтому нуждается в доступе к его структуре. В UNIX это struct proc.

Важно отметить, что контекст процесса не полностью защищен от обработчика прерывания. Поэтому неверно написанный обработчик может нанести вред любой части адресного пространства процесса.

Кроме аппаратных прерываний в системе также существуют программные прерывания (системные вызовы) и исключения (юниксоиды говорят программные и системные прерывания).

Очевидно, что разные типы прерываний имеют разную важность для работы системы. Кроме того, они могут иметь разный объем кода, часто довольно большой. Под большим объемом понимается объем, который требуется для вычислений в течение нескольких тиков системного таймера. В силу этого в UNIX/LINUX вводятся уровни приоритетов прерываний (ipl) interrupt priority level.

Рассмотрим общий алгоритм обработки прерываний, предлагаемый юниксоидами: (в нашем ГОСТе нет у стрелок писать не положено)



Аппаратные прерывания имеют очень высокий уровень приоритетов, из которых прерывание от системного таймера имеет наивысший приоритет.

Очевидно, что прерывания от внешних устройств являются высокоприоритетными, так как должны быть обработаны быстрее всех, клавиатура и мышь имеют наиболее высокий уровень привилегий для того, чтобы их обрабатывали сразу, потому что это интерактивные устройства.

Говоря о прерываниях, речь идет о драйверах. Каждое устройство имеет один драйвер и если устройство использует прерывание, то этот драйвер регистрирует один обработчик прерывания. Драйверы могут

регистрировать обработчик прерывания и разрешить определенную линию обработки прерывания, посредством функции `request_irq`

`int request_irq(unsigned int irq, irqreturn_t (*handler)(int, void*, struct pt_regs*), unsigned long irqflags, const char *devname, void *dev_id)`  
- регистрирует обработчик прерывания

Можно обратиться к книжке Alessandro Rubni, Yonathan Corlet Linux Device Drivers

`extern void free_irq(unsigned int irq, void *dev)` - освобождает обработчик прерывания, чтобы он больше не выполнялся.

Первый параметр `irq` определяет номер прерывания. Для некоторых устройств, например, для legacy device, таких как системный таймер и клавиатура, эта величина устанавливается аппаратно (hard coded). Для большинства других устройств она устанавливается динамически.

Второй параметр `handler` - указатель на обработчик прерывания, который обслуживает данное прерывание. Именно эта функция будет вызвана, когда возникнет соответствующее прерывание.

```
enum irqreturn {
    IRQ_NONE = (0<<0), // прерывание было не от соответствующего
                         // девайса или прерывание не было обработано
    IRQ_HANDLED = (1<<0), // прерывание было обработано
                          // соответствующим устройством
    IRQ_WAKE_THREAD = (1<<1) // обработчик запрашивает
                           // пробуждения нити обработчика
};

typedef enum irqreturn irqreturn_t;
#define IRQ_RETVAL(x) ((x) ? IRQ_HANDLER: IRQ_NONE)
```

Важно, чтобы функция `free_irq`, освобождала именно то устройство, драйвер которого установил этот обработчик. `void *dev` - указатель на девайс для этого, это поле добавили в более поздних версиях LINUX.

Возвращаясь к `request_irq`: третий параметр `irqflags`.

Замечание: Начиная с 2.6.19 версий ядра, все флаги были заменены, старое название было SA\_\*, -> IRQF\_\*, это сигнализирует о актуальности литературы.

Важнейшим флагом, который мы будем использовать в лабах - IRQF\_SHARED.

### Флаги:

- IRQF\_SHARED - разрешает разделение линии прерывания, то есть совместное использование линии IRQ разными устройствами
- IRQF\_PROBE\_SHARED - устанавливается абонентами, вызывающими данные действия, если они предполагают возможность нестыковок при совместном использовании использования линии IRQ
- IRQF\_TIMER - флаг, маскирующий прерывание как прерывание от таймера
- IRQF\_PERCPU - прерывание, закрепленное монопольно за конкретным процессором

Четвертый параметр dev\_name, ASCII текст, представляющие связь устройства с прерыванием, например может иметь значение keyboard, эта строка использует название /proc/irq, а также /proc/interrupts.

Пятый параметр dev\_id, используется прежде всего для разделения (share) линий прерываний. Когда обработчик выполняется, dev\_id обеспечивает уникальные cookie-файлы (смысл в том), чтобы выполнить удаление только нужного обработчика прерывания соответствующей линии прерываний. Указатель dev\_id имеет тип void, то есть он может указывать на что угодно, но обычно используется указатель на структуру, специфичную для устройства.

В случае успеха request\_irq возвращает 0, ненулевая величина означает ошибку, и в этом случае обработчик прерывания не регистрируется. Обычная ошибка EBUSY значит, что данная линия пребывания уже используется или не был указан флаг SHARED.

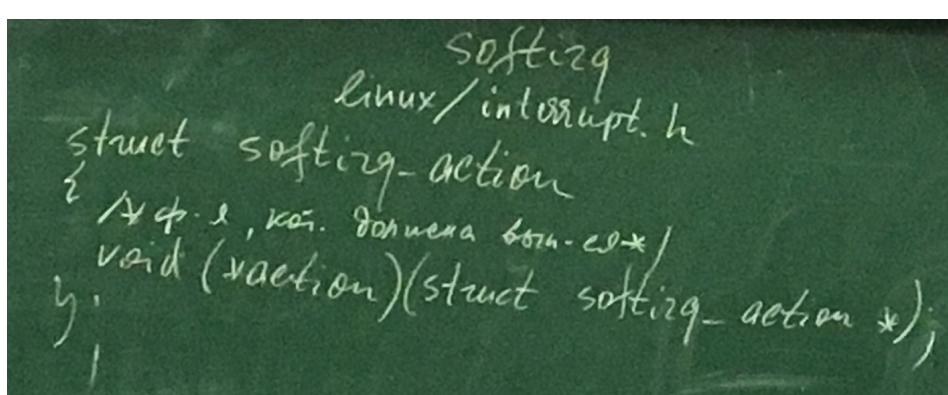
Процедура обработки быстрых прерываний очень короткая, поэтому прерывает текущую активность недолго. При выполнении таких прерываний, запрещены все прерывания на локальном процессоре, а также запрещены прерывания по данной линии прерываний. В старых версиях ядра такие обработчики прерываний регистрировались с флагом IRQF\_INTERRUPT. В новых версиях ядра быстрым прерыванием является единственное - IRQF\_TIMER.

Медленные прерывания делятся на две части: верхнюю и нижнюю половины. Это историческое название - top & bottom half. Фактически, top half это быстрое прерывание со всеми вытекающими, то есть верхняя половина должна заканчиваться как можно быстрее, так как она блокирует всю активность на локальном процессоре, то есть выполняется при защищенных прерываниях.

Нижняя половина это отложенное действие, которое как правило идет сразу после верхней, но имеет другой уровень приоритета и может быть прервана кем угодно, если называть вещи своими именами.

Однако, в отличие от реального обработчика быстрого прерывания, top half делит свои завершения, должна инициализировать последующее выполнение bottom half, а именно должна поставить соответствующий обработчик в очередь на выполнение, причем в соответствующую очередь, в соответствии с типом отложенного действия. Заканчивается классическим IR\_RET'ом.

В настоящее время обработчики нижних половин бывают трех видов: softirq, tasklet, workqueue. Мягкие или гибкие пребывания - tasklet, workqueue'ы. Softirq прерывания определяются статически во время компиляции ядра. В файле linux/interrupt.h определена структура struct softirq\_action. В версии ядра 4.9 в этой структуре определена только функция которая должна выполняться.



В версии ядра  
2.??  
существовало  
поле данных

В файле  
kernel/softirq.c  
определен  
массив из 32  
экземпляров  
этой структуры.

NR\_SOFTIRQS - число задействованных номером - 32. Очевидно, что имеется возможность создать 32 softirq обработчика, в настоящее время определено десять.

# Рязанова, лекция

....///  
//////

Добавить новый уровень разработчика - XXX\_SOFT\_IRQ, можно только перекомпилировав ядро.

Отложенное прерывание с меньшим номером выполняется дальше, то есть его приоритет выше.

Для создания нового уровня soft\_IRQ нужно

1. Определить новый тип прерывания, вписав его в константу XXX\_SOFT\_IRQ в перечислении
2. Вовремя инициализации модуля должен быть зарегистрирован обработчик отложенного прерывания с помощью вызова open\_softirq

soft\_irq - действия, которые завершают действие прерывания.

```
void xxx_soft_handler(void *data)
{
    void __init rollers_init()
    {
        request_irq(irq, xxx_interrupt, 0, "xxx", NULL);
        open_softirq(XXX_SOFT_IRQ, xxx_soft_handler, NULL);
    }
}
```

Функция open\_softirq имеет три параметра: XXX\_SOFT\_IRQ - индекс, второй параметр обработчик, третий параметр значение поля data.

Очевидно, что функция обработчик soft irq должна соответствовать правильному прототипу

3. Зарегистрированная soft irq должна быть поставлена в очередь на выполнение. Для этого оно должно быть возбуждено с помощью функции raise\_softirq - имитация отложенного прерывания. Обычно обработчик аппаратного прерывания, то есть верхняя половина перед возвратом управления возбуждает свой обработчик отложенного прерывания

```
/* the interrupt handler */
static irqreturn_t xxx_interrupt(int irq,
                                 void *dev_id)
{
    /* Mark softirq as pending */
    raise_softirq(XXX_SOFT_IRQ);
    /* Then IRQ_HANDLER. */
}
```

Проверка ожидающих выполнение отложенных прерываний и их запуск осуществляется в следующих случаях:

1. При возврате прерывания
2. В контексте ksoftirqd
3. В любом коде ядра, в котором явно проверяются и запускаются ожидающие обработчики отложенных прерываний, как это делается например в сетевой подсистеме

Независимо от метода вызова soft\_irq его выполнение осуществляется функцией do\_soft\_irq, которая в цикле проверяет наличие отложенных прерываний. Несмотря на то, что у softirq есть приоритеты, softirq никогда не вытесняет другой softirq. Единственное событие, которое может вытеснить softirq, это аппаратное прерывание. При этом на другом процессоре может выполняться другой обработчик этого же softirq. В результате для softirq остро стоит проблема взаимоисключения. Это означает, что функция должна быть реинтегрируемой, а если есть критический участок, то он должен быть защищен. Демон ksoftirqd это поток ядра каждого процессора, то есть percpu.

Когда машина нагружена мягкими прерываниями, которые, как правило, обслуживаются по возвращении из аппаратного прерывания, то возможна ситуация когда такое softirq переключается значительно быстрее чем может быть обслужено. Это связано с тем, что возможны ситуации, в которых IRQ приходят очень быстро, одно за другим, и в результате он не может закончить обслуживание одного до прихода другого. Например, это может произойти когда сетевая карта с высокой скоростью получает пакеты в течение короткого промежутка времени. В результате операционная система не может с этим справиться и создает очередь для последовательной обработки softirq с помощью специального процесса, который называется ksoftirqd.

Сб, 27 апреля

Если ksoftirqd занимает больше чем какой-то достаточно небольшой процент процессорного времени, это говорит о том, что машина находится под большой нагрузкой прерываний.

## Tasklet.

Отмечается, что тасклеты это частный случай softirq, но это отложенные прерывания, для которых обработчик, то есть tasklet не может выполняться одновременно на нескольких процессорах, в отличие от softirq. Для них по этой причине не нужно реализовывать средства взаимоисключений.

Tasklet'ы проще всего понимать как простые softirq. Разные Tasklet'ы могут выполняться параллельно на нескольких процессорах, но одного типа не могут. Поэтому tasklet'ы являются компромиссами по производительности и простотой. Tasklet'ы могут быть зарегистрированы в системе как статически, так и динамически.

The handwritten code shows the definition of the `struct tasklet_struct` from the `linux/interrupt.h` header file. The structure includes fields for a linked list pointer (`*next`), a state variable (`unsigned long state`), a reference count (`atomic_t count`), a function pointer (`void (*func)(unsigned long)`), and a data pointer (`unsigned long data`). A brace on the right side groups the `state`, `count`, and `func` fields, with annotations above them: `TASKLET_STATE_SCHED` above `state` and `TASKLET_STATE_RUN` above `func`. There is also a small circle with a zero above the brace.

```
<linux/interrupt.h>
Struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state; /* текущее состояние */
    atomic_t count; /* счетчик ссылок */
    void (*func)(unsigned long); /* адрес */
    unsigned long data;
};
```

`TASKLET_STATE_SCHED` - запланирован, `__RUN` - выполняется.

Count счетчик ссылок на tasklet, если 0, то tasklet, разрешен и может выполняться если он запланирован, иначе он запрещен и выполняться не может.

Для того, чтобы запланировать tasklet на выполнение должна быть вызвана функция `tasklet_schedule()`. Аналогично softirq планирование tasklet'ов на выполнение выполняет `interrupt_handler`. Для оптимизации tasklet обрабатывается на том процессоре, на котором его запланировал обработчик прерывания.

Сб, 27 апреля

Статические тасклеты создаются с помощью двух макросов, которые определены в linux/interrupts.h - DECLARE\_TASKLET(name, func, data); DECLARE\_TASKLET\_DISABLE(name, func, data). Оба макроса статически создают экземпляр структуры struct tasklet. Соответственно с именем name, вызываемой функцией func и какими-то данными data. Первый макрос создает tasklet, у которого поле count равно нулю и соответственно этот tasklet разрешен, второй создает экземпляр с count равным 1, и, соответственно такой tasklet будет запрещен.

При динамическом создании tasklet'а объявляется указатель на структуру.

```
struct tasklet_struct *mytasklet;
```

Для инициализации tasklet'а вызывается функция:  
tasklet\_init(mytasklet, tasklet\_handler, data);

//tasklet\_handler prototype:

```
Void tasklet_handler(unsigned long data);
```

В обработчиках tasklet'ов нельзя использовать семафоры, так как они не могут блокироваться. Если используются одинаковые данные с обработчиком прерывания или с другим tasklet'ом, то они должны быть защищены spin блокировкой.

## Планирование tasklet'ов.

Tasklet'ы могут быть запланированы с помощью следующих функций:

```
tasklet_schedule(); tasklet_hi_schedule();
```

Этим функциям передается единственный аргумент - указатель на структуру tasklet\_struct.

Например tasklet\_schedule(&irq\_tasklet);

Запланированные на выполнение tasklet'ы хранятся в связных списках. Обычные tasklet'ы будут находиться в связном списке - tasklet\_vet, а высокоприоритетные в списке tasklet\_hi\_list, оба списка состоят из экземпляров структуры tasklet\_struct.

После того как tasklet запланирован он будет запущен только один раз даже если он был запланирован на выполнение несколько раз. Для

Сб, 27 апреля

отключения заданного tasklet'а используется функция tasklet\_disable() или функция tasklet\_disable\_nosync(). Первая функция не сможет отменить tasklet, который уже выполняется, вторая может прервать выполнение tasklet'а. Для активации tasklet'а используется функция tasklet\_enable().

На tasklet'ах определены специальные функции блокировки - tasklet\_trylock(), tasklet\_unlock(). Внутри этой функции вызывается команда

```
Static inline int tasklet_trylock(struct  
tasklet_struct *t)  
{  
    if (!test_and_set_bit(TASKLET_STATE_  
RUN, &(t->state)),  
}
```

```
Static inline void tasklet_unlock(struct tasklet_struct *t)  
{  
    smp_mb_before_atomic().  
    clear_bit(TASKLET_STATE_RUN, &(t->state)),  
}
```

# Рязанова, лекция №9

## Сетевой стек, связанный с взаимодействием через сокеты.

Знать, что такое IRQ. Знать, что такое порт. Порт - это адрес.

Два способа взаимодействия с устройствами - io mapping - отображение устройств на адресное пространство ввода/вывода, memory mapping - видеокарта, которая отображается на адресное пространство физической памяти.

Контроллер прерывания, освежить в памяти.

---

**Сокет** - это абстракция конечной точки взаимодействия (точки соединения)

Тема сокетов излучается как продолжение взаимодействия процессов.

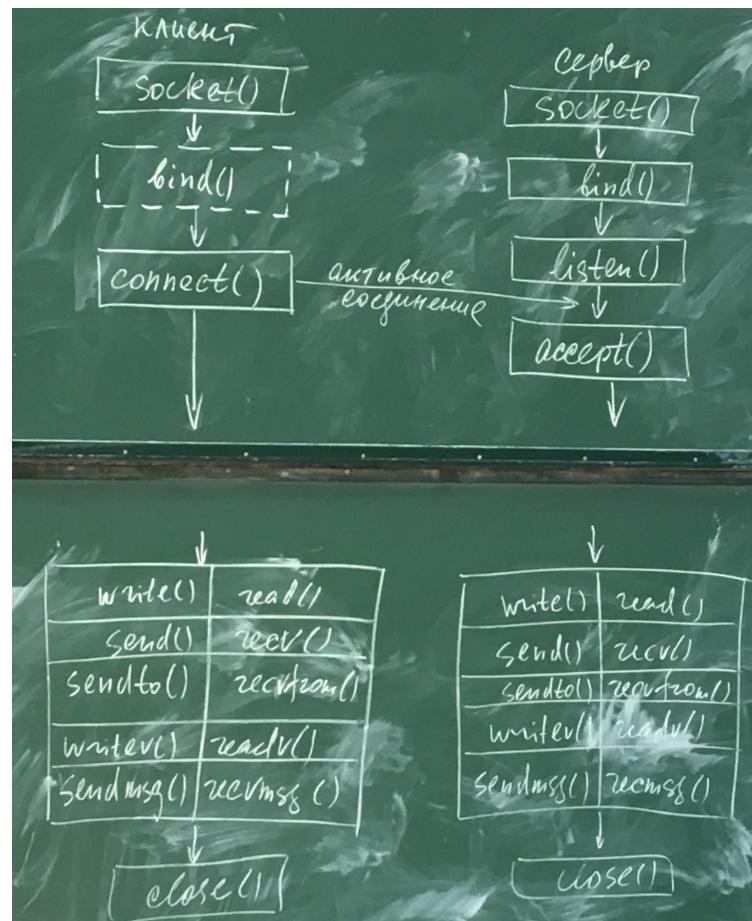
Взаимодействие процессов с помощью сокетов всегда выполняется по модели взаимодействия клиент/сервер. Клиенты обращаются к серверам за обслуживанием. В UNIX?LINUX сокеты рассматриваются как специальные файлы, то есть они поддерживаются файловой системой. Каждый сокет это открытый файл, описывается структурой struct file. И поскольку это файл, то сокет имеет inode, но этот inode не дисковый.

Все функции, которые принимают в качестве аргумента файловые дескрипторы могут принять дескриптор сокета.

Очевидно, что установление локального, или удаленного соединения через сокеты требует использования соответствующих системных вызовов. Их принято представлять в виде так называемого сетевого стека. Сетевой стек можно найти в книге Стивена Network Programming.

Сб, 18 мая

Рассмотрим сетевой стек:



Методы `sendmsg()` и `sendto()` используется для сокета DGRAM.

`Bind()` связывает сокет со связанным адресом, этот метод необходимо вызывать на стороне сервера. На стороне интернет сокетов этот адрес состоит из арифметической системы и номера порта

Клиенты могут не вызывать `bind()`, так как их адрес никакой роли не играет, в этом случае адрес им назначается автоматически.

```
int bind(int socket, struct sockaddr *addr, int addrlen);
```

Системный вызов `listen()` используется сервером для информирования ОС, что в сокете нужно принять соединение. Это имеет смысл только для ориентированных на соединение протоколов - только для TCP.

```
int listen(int sockfd, int backlog);
```

`Connect()`, этот системный вызов устанавливает соединение. Например, в TCP по переданному адресу. Для протоколов без установления

Сб, 18 мая

соединения, таких как UDP может использоваться для указания адреса назначения для передаваемых пакетов.

```
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

Accept() используется сервером для принятия соединения при условии, что ранее он получил запрос соединения. В противном случае, запрос будет заблокирован до тех пор, пока не поступит запрос соединения.

```
int accept(int sockfd, void *addr, int *addrlen);
```

Когда соединение принимается, сокет копируется таким образом, что исходный сокет остается в состоянии LISTEN, а копия в состоянии CONNECT. То есть вызовом accept() возвращается новый дескриптор файла для следующего сокета. Такое дублирования сокетов в ходе принятия соединения дает серверу возможность принимать новые соединения без необходимости предварительно закрыть предыдущие соединения.

## Типы сокетов.

В системе поддерживаются парные сокеты, фактически они являются аналогом pipe, но с дуплексной связью. Для них используются свои специальные функции.

Можно также отделить сокеты, которые используются для взаимодействия на отдельностоящих процессорах, сокеты UNIX.

Сокеты предназначенные для удаленного взаимодействия, сокеты INET

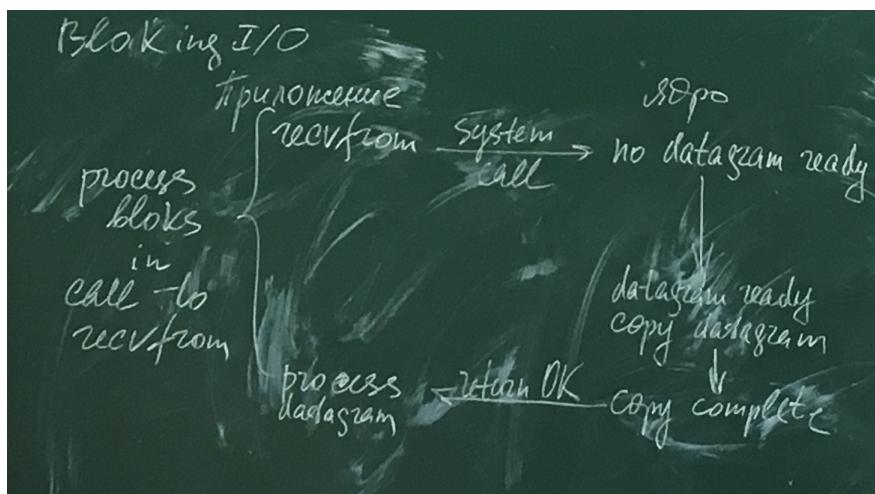
Их API одинаковое, но для сетевых сокетов часть функций неактуальна.

## 5 моделей ввода вывода. Источник - Стивен.

Рассмотрим модели ввода вывода с точки зрения программиста.

В UNIX подобных системах доступно 5 + 1/2 различных моделей ввода/вывода.

### 1) Blocking I/O (блокирующий ввод/вывод)

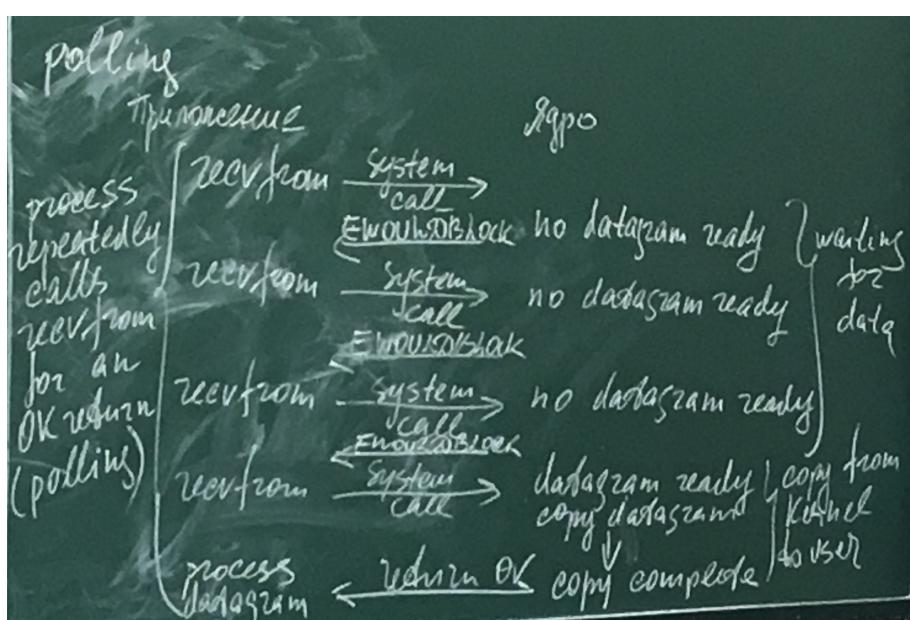


Процесс блокирован до момента, когда он сможет получить данные. Данные нужно скопировать из буфера устройства в буфер ядра, чем занимаются обработчики прерывания, которые затем надо вернуть приложению, что

показано return OK, все это время ожидания, процесс находится в состоянии блокировки

### 2) Polling I/O (неблокирующий ввод/вывод)

Ввод вывод с запросом, если по-другому.



Здесь речь идет о готовности данных.

### 3) Multiplexing I/O. Ввод/вывод с мультиплексированием. Мультиплексирование ввода/вывода.



Системный вызов `select()` есть мультиплексер. Что такое мультиплексер. Из словаря. Мультиплексор - по-другому коммутатор, устройство, которое объединяет информацию по нескольким каналам ввода и выдает ее по одному каналу выхода

Oxford compute tech определение статья m239. Мультиплексирование - процесс совмещения нескольких сообщений, передуваемых одновременно в одной физической или логической среде. Существует два вида мультиплексирования - временное и частотное. Временное time divisor multiplexing, устройству отводятся интервалы времени, которое оно может использовать.

Когда выполняется системный вызов `select()` процесс блокируется, в ожидании поступления данных на один или несколько сокетов (пока не возникнет соединение).

В этом смысл мультиплексирования.

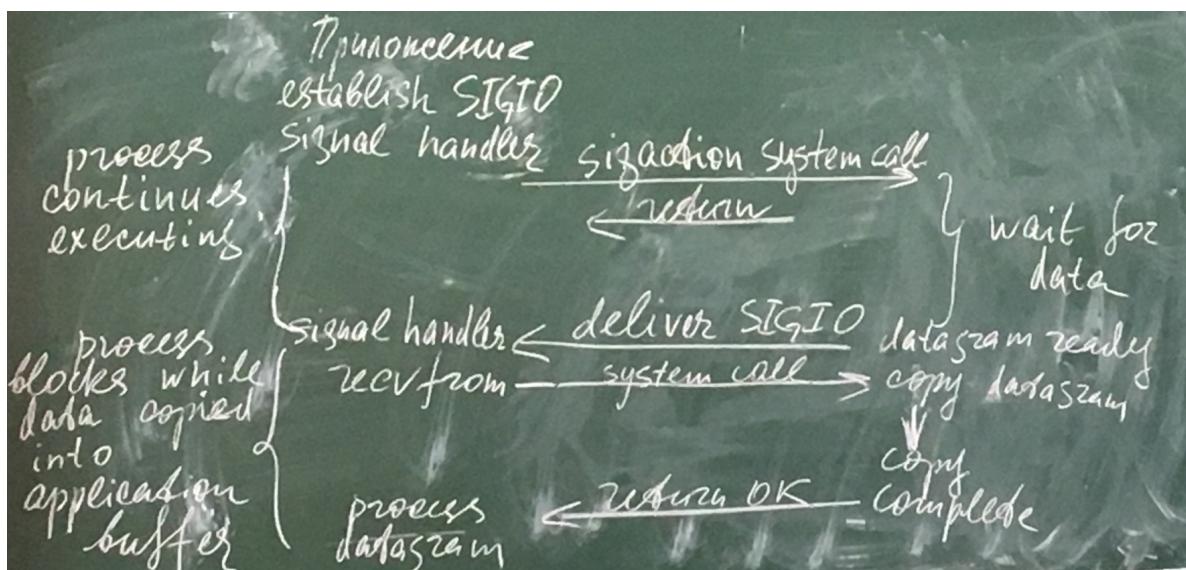
В случае мультиплексирования, в цикле проверяются все сокеты и берется первый готовый. Пока обрабатывается первый сокет, могут подоспеть другие, в результате чего снижается время простоя. Если же эту проблему решать путем обычной блокировки, то случайнм образом выбирается соединение из которого чтение будет выполняться первым. Если данные не готовы, то будем висеть в этом соединении в блокировке, затем выбирается второе соединение, опять может быть блокировано ожидание, и так далее. А соединение могло в этот момент произойти на i-м сокете, а мы заблокированы на первом. Ъ

Мультиплексирование решает эту проблему - блокировка есть, но ее время меньше.

## Этот метод половинка, для решения той же самой задачи вместо мультиплексирования.

Способ, похожий на мультиплексирование, при котором несколько процессов или потоков выполняют блокирующий ввод/вывод, то есть создается несколько потоков, например, каждый из которых работает с одним сокетом. Недостаток этого подхода очевиден - это "дорогой" подход, поскольку потоки LINUX дорогие с точки зрения команд, то есть требуют больших накладных расходов. Второй недостаток связан с питоном. В Питоне существует так называемый GIL (Global Interpreter Lock), которая накладывает ограничения на потоки - а именно нельзя использовать несколько процессоров одновременно. В итоге в каждом процессе может выполняться только один поток. Решение - создавать дочерние процессы в блокирующем стиле и решать проблему взаимоисключения.

### 4) Ввод/вывод, управляемый сигналами.



Сначала устанавливаются параметры сокетов для работы с сигналами, и обработчик сигналов. Обработчик сигнала устанавливается системным вызовом `sigaction()`. Результат возвращается сразу и приложение не блокируется. Вся работа на ядре, а именно ядро отслеживает когда данные будут готовы, после этого, посылает сигнал (когда готовы), который вызовет установленный на этот сигнал обработчик сигнала. Фактически, обработчик сигнала это callback функция. Сам метод `recvfrom` можно выполнить либо в обработчике сигнала, либо в основном потоке программы. При этом сигнал для каждого процесса SIGIO, для каждого процесса будет только один, поэтому в результате за раз можно работать только с одним файловым

Сб, 18 мая

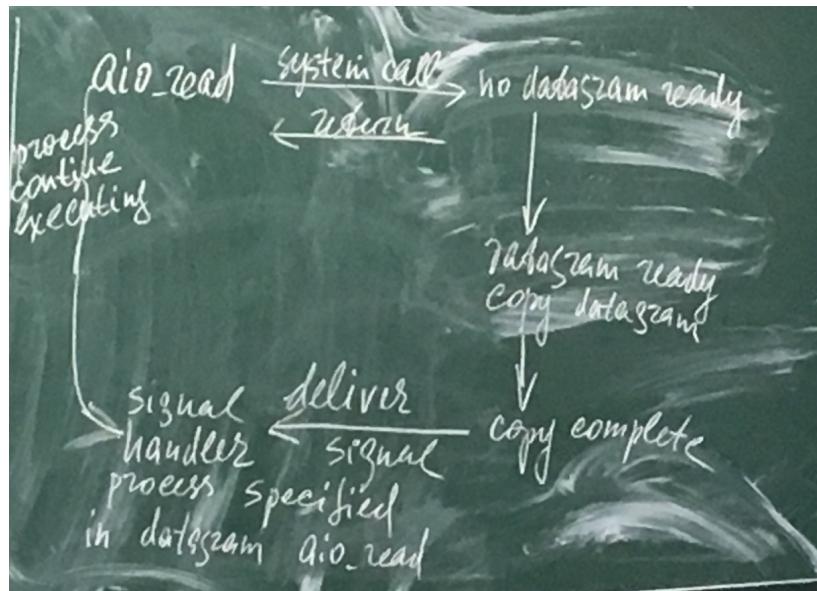
дескриптором. На время обработчика сигнала, сигнал блокируется, если за это время он доставляется несколько раз, то они теряются.

## 5) Асинхронный ввод/вывод.

Такой ввод/вывод выполняется с помощью специальных системных вызовов, которые назовем aio\_read.

Эти две последние модели связаны с идеей асинхронного ввода/

вывода. Блокировки зла, они замедляют выполнение процессов. Но идея такого ввода/вывода (асинхронного) преследует возможность процессу не блокироваться, и процесс не блокируется вплоть до получения данных.



25.05.19

классификация моделей способов ввода/вывода:

	Blocking	Non-Blocking
Synchronous	read/write	опрос
Asynchronous	I/O multiplexing (select/poll)	AIO

1. модель блокирующего синхронного ввода/вывода Blocking I/O  
матрица базовых моделей ввода/вывода

команды(функции read/write) ...

базовый способ работы с обычными файлами  
процесс запросивший ввод/вывод блокируется

все время пока данные не будут готовы для того чтобы быть перемещенными в буфер  
приложения

все это время процесс блокирован

2. модель не блокирующий ввод/вывод Polling (опрос)

процессор постоянно занят тем

что опрашивает готовность устройства

3. мультиплексирование ввода/вывода

блокирующий асинхронный (мультиплексирование)

не смотря на то что процесс блокирован на мультиплекс - это асинхронный ввод/вывод

обработка по мере возникновения соединения

речь идет о советах о специальных файлах

5. асинхронный ввод/вывод

это модель с перекрывающей обработкой ввода/вывода

например запрос read возвращается немедленно показывая что чтение было успешно  
начато

приложение может выполнять другую обработку

в то время как фоновая операция чтения завершается

когда операция чтения возвращает ответ в виде сигнала или в виде call back функции (/  
которая может быть реализована в виде потока)

по модели генерируется сообщение что операция выполнена

процесс не блокируется

асинхронный ввод/вывод - приложение запросив данные продолжает что-то делать  
имеется ввиду единственный поток

запуск асинхронного ввода/вывода:

1. запуск фронтом
2. запуск уровнем

<http://davmac.org/davpage/linux/async-io.html>

## **ВВЕДЕНИЕ В ДРАЙВЕРЫ (управление устройствами)**

unix/linux рассматривает внешние устройства как специальные файлы

### **СПЕЦИАЛЬНЫЕ**

специальные файлы устройств обеспечивают унифицирование к периферийным устройствам

эти файлы обеспечивают связь между файлами системы и драйверами устройств

такая интерпретация специальных файлов обеспечивает доступ к специальным файлам

как и обычный файл устройства может быть открыт/закрыт/из него можно читать/в него можно писать

каждому внешнему устройству ОС ставит в соответствие минимум 1 специальный файл

эти файлы находятся в каталоге /dev корневой файловой системы

подкаталог /dev/fd содержит файлы с именами 0, 1, 2

в некоторых системах имеются файлы с именами /dev/stdin /dev/stdout /dev/stdf (или r? x?)  
что эквивалентно /dev/fd/0

в ОС имеются 2 типа специальных файлов устройств:

1. символьный
2. блочный

тк специальные файлы устройств это файлы => они имеют inode, т.е. описываются в системе соответствующим индексным дескриптором

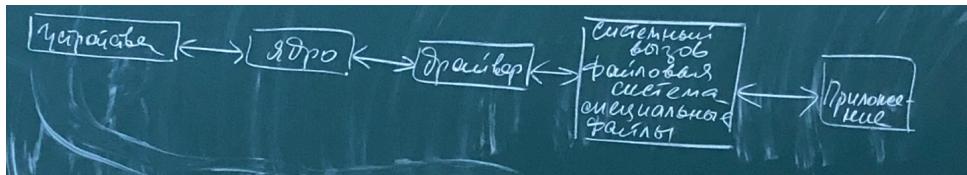
```
в struct inode
struct inode
{
    ...
    dev_t i_rdev; // это поле содержит фактический номер устройства
    struct list_head i_devices;
    union
    {
        struct pipe_inode_info *i_pipe;
        struct block_device *i_bdev;
        struct cdrom *i_cdev;
        char *i_link;
        unsigned i_dir_seq;
    }; // отражает перечисление специальных файлов
    ...
};
```

kdev\_t - device type предназначен для хранения номеров устройств

система должна идентифицировать внешнее устройство  
для этого используется старший и младший номера устройств (major/minor)

каким образом происходит обращение и работа с внешними устройствами

взаимодействие прикладных программ с аппаратной частью системы осуществляется по следующей схеме



драйвер это программа или часть кода ядра которая предназначена для управления обычным конкретным внешним устройством

в линукс драйверы устройств бывают 3х типов:

1. встроенные в ядро (устройство автоматически обнаруживаются системой и становятся доступными приложению) (контроллеры ide, материнская плата, последовательные и параллельные порты)
2. реализованные как загружаемые модули ядра (модули часто используются для управления: SCSI-адапторы, звуковые и сетевые ...) (/lib/modules) (обычно при инсталляции системы задается перечень модулей, которые будут автоматически подключать на этапе загрузки системы) (список загружаемых модулей хранится в файле /etc/modules) (для подключения и отключения модуля есть утилиты: lsmod, insmod, rmmod,)
3. код драйверов 3го типа поделен между ядром и специальной утилитой (например у драйвера принтера ядро отвечает за взаимодействие с параллельным портом, а формирование управляющих сигналов для принтера осуществляется демоном печати lpd rjnjhsg lkz 'njuj bczjkmpetn cgtwbfkmye. ghjuhfve abkmnh) (другим примером такого типа драйвера могут служить драйвера модели)

#### МЛАДШИЕ СТАРШИЕ НОМЕРА УСТРОЙСТВ

если в каталоге /dev задать команду ls -l, томы увидим список специальных файлов устройств

с - устройство символьное ... в этой строке есть 2 числе в конце это и есть старший и младший номера устройств

старший номер идентифицирует драйвер связанный с устройством  
например /dev/null и /dev/zero

оба управляют драйвером 1

а виртуальные консоли и последовательность терминалов управляющих драйвером идут под номером 4

... разрешают многим драйверам разделять старшие номера

что отражает младший номер?

младший номер отражает конкретное устройство

например жесткий диск (устройство и у него 1 старший номер)

но разделы диска будут в системе иметь младшие номера

в результате каждый раздел диска будет иметь 2 номера старший и младший

и старший у всех одинаковый

внутреннее представление номеров:  
не оговаривается поля этого типа  
тип полей определен в <linux/types.h>

некоторые старшие номера зарезервированные для определенных драйверов устройств  
другие страшите номера динамически присваиваются драйверам устройств

когда загружается ос линукс  
например старший номер 94 всегда означает DASD direct access storage device

старший номер может разделяться множеством драйверов устройств  
для того чтобы определить какие старшие номера  
имеются в текущей реализации линукс надо посмотреть /proc/devices

используют младшие номера чтобы определять отдельные физические устройства или  
отдельные логические устройства

например используя  
# ls -l /dev/ |grep «^c» можно получить список файлов символьных устройств

чтобы получить старшую или младшую часть dev\_t используются макросы  
MAJOR(dev\_t dev);  
MINOR(dev\_t dev);  
возвращают unsigned int

если наоборот имеются номера и нужно преобразовать в dev\_t  
MKDEV(int major, int minor);

```
struct stat
{
    st_dev; // для каждого файла хранится номер устройства файловой системы в
    // которой располагается файл и соответствующий ему индексный узел
    st_rdev; // определен только для специальных файлов, т.е. для блочных или
    // символьных устройств аналогично struct inode
}
```

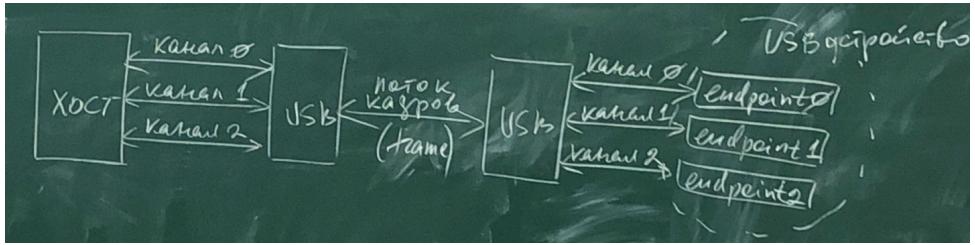
## USB ДРАЙВЕРЫ

в ней главным является хост который начинает все транзакции

1й патек token генерируется хостом для описания того  
как будет выполняться чтение или запись и указывается адрес устройства и номер  
конечной точки endpoint

при подключении устройства драйверы ядра считывают список конечных точек и создают  
управляющие структуры данных для взаимодействия с каждой конечной точкой

совокупность конечной точкой и структурой данных ядра называется каналом pipe

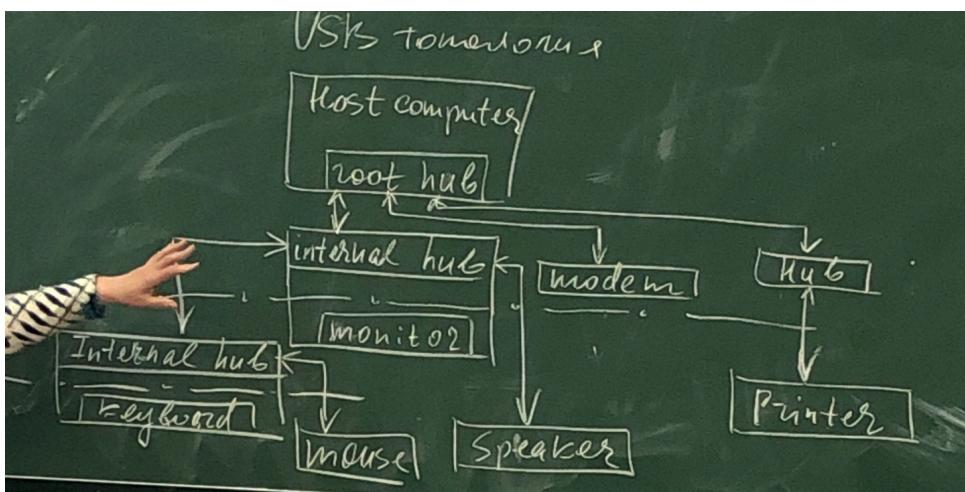


pipe это логическое соединение между хостом и конечной точкой устройства  
при этом потоки данных имеют определенное направление in/out передачи данных  
перед отправкой данные собираются в пакет

4 типа пакетов usb:

- token
- data
- handshake
- SOF - start of frame

usb топология



возможно до 5 уровней слоев  
mouse уже устройство  
на этой схеме 3 слоя