

9.02.2019

## ФАЙЛОВЫЕ ПОДСИСТЕМЫ

уровень управления данными - уровень обеспечения, хранения долговременных данных, доступ к данным

файлы и временные файлы - рабочие файлы

для чего создаются в программах файлы?

для долговременного хранения информации

для этого используются специальные внешние устройства, которые используют долговременное хранение информации

управление файлами осуществляется частью ОС - файловой системой (файловой подсистемой) File System

файловая система - это часть ОС, которая отвечает за возможность хранения информации и доступа к ней

если файл предназначен для хранения данных, то файловая система управляет процессом хранения и обеспечивает последующий доступ к этим данным

т.е. файл в файловой системе является единицей хранения информации (данных)

файл - обычные файлы хранятся во вторичной памяти - каждая индивидуально  
индексированная совокупность данных  
pipe (буферы) - хранятся в оперативной памяти (ОП)

файл - каждая поименованная совокупность данных хранящаяся во вторичной памяти называется обычным файлом (определение Рязановой)

когда говорят о файловой подсистеме речь идет об обычных файлах

файловая система

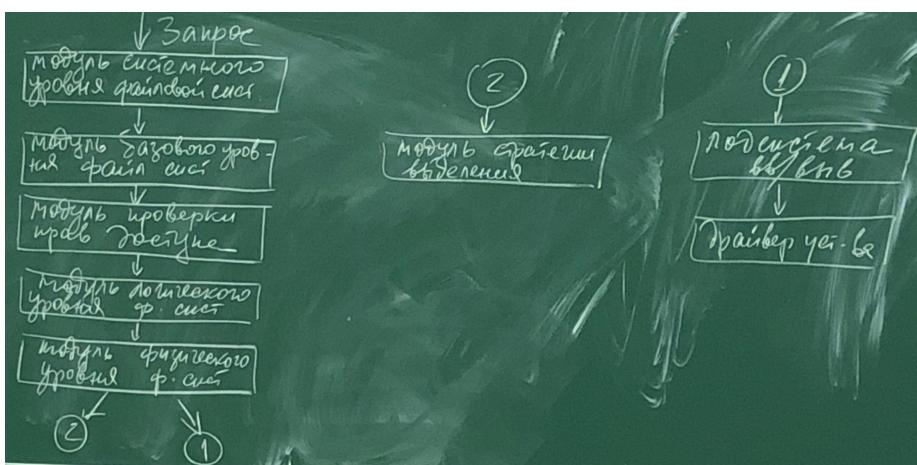
- а) определяет формат сохраненных данных и способ их физического хранения
- б) связывает формат физического хранения и API для доступа к файлам

файловая система не интерпритирует данные хранящиеся во вторичной памяти

вторичная память энергии независимая

«ЛИБИСТРИЧЕСТВО» Рязанова (электричество)

рассмотрим обобщенную модель файловой подсистемы  
она имеет иерархическую организацию



пример  
структурирования  
файловой системы

вопросы структурирования информационной системы при управлении информацией являются чуть ли не самыми важными при разработки системы

такое структурирование должно учитывать задачи, которые должна решать файловая система

задачи файловой системы:

1. обеспечение удобного доступа пользователей к файлам как часть задачи именования файла
2. собственно именование файлов, т.е. 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 файла 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 (unix, windows) должны обеспечивать возможность существования в системе нескольких файлов с одинаковыми именами, возможность доступа к одному и тому же файлу по разным именам  
поэтому в системах должна существовать соответствующая информация (справочник)

в развитых системах такой справочник состоит из двух отдельных справочников:  
символьный и базовый

символьный уровень - уровень именования файлов удобный для пользователей (уровень каталогов и имен файлов которыми оперирует приложение)

удобным способом структурирования, объединением различных файлов является директория (каталоги)

базовый уровень - уровень идентификации файла

файловая подсистема - программа, которая работает по определенным принципам =>  
любая переменная должна быть идентифицирована

файл должен иметь уникальный идентификатор и файл должен быть описан в системе,  
чтобы можно было с ним работать

модуль логического уровня

любая программа считает, что она начинается в нулевого адреса

такое адресное пространство называется логическим  
с файлами тоже самое

когда создаем файл, указатель файла стоит фактически на нулевой отметке, т.е. файл имеет логическое адресное пространство, которое начинается в нулевого адреса

т.е. логический уровень позволяет обеспечить доступ к данным в файле в формате  
отличном от формата их физического хранения

логическое адресное пространство начинается в нулевого адреса и представляет собой  
непрерывную последовательность адресов

обычно файловая система не накладывает ограничений на структуру данных файла  
и никак их не интерпретирует (не полностью записано)

структурой данных управляет пользователь

может существовать разные хранения данных которые известны только программам (приложениям)  
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 (конец строки)  
поэтому они называются текстовыми

для работы с текстовыми файлами существуют текстовые редакторы  
в текстовых файлах информация хранится посимвольно (1 символ - 1 байт)

в ОС (windows, unix, linux) существуют байт ориентированные и блокориентированные файлы

в байтотриентированных файлах информация хранится в байтах и в бауториентируемых устройствах информация передается байтами

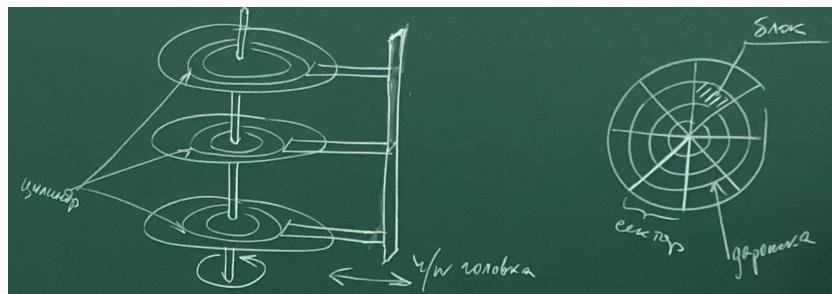
в блокориентированных файлах информация хранится в блоках фиксированных размеров

в современных системах блокориентированными устройствами являются только диски

с блокориентированными файлами могут работать только блокориентируемые программы

физический уровень хранения файлов

наименьший адресуемый участок конкретной пластины определенной дорожки называется сектором



на пластинах концентрические дорожки  
совокупность одних и тех же дорожек разных пластин - цилиндр  
диски постоянно вращаются

адресное пространство диска файла выделяется блоками  
система обеспечивает адресацию каждого выделяемого блока

если связное распределение адресного пространства, то файл на диске занимает смежные сектора

если несвязное распределение, то в разброс это означает что система должна обеспечивать хранение адреса каждого блока выделенного каждому конкретному блоку

т.о. это представление о файловой системе позволяет обеспечить общий интерфейс для любой файловой системы

это возможно потому, что ядро реализует слой абстракции над своим низкоуровневым интерфейсом

VFS (Virtual File System)/vnode unix интерфейс

linux изменил этот интерфейс отказавшись от vnode => VFS (Virtual File System)

VFS (Virtual File System) предоставляет общую файловую модель, которая способна отображать общие возможности и поведение любой мыслимой файловой системы

уровень абстракции VFS работает на основе базовых концептуальных интерфейсов и структур данных

каждая файловая отдельная система определяет особенности того, как файл открывается, как выполняется обращение к файлу, как считывается информация из файла и т.п.

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

большинство файловых систем запрограммировано так, что API которое предоставляют  
файловые системы рассматриваются как абстрактный интерфейс, который ожидаем и  
понятен VFS

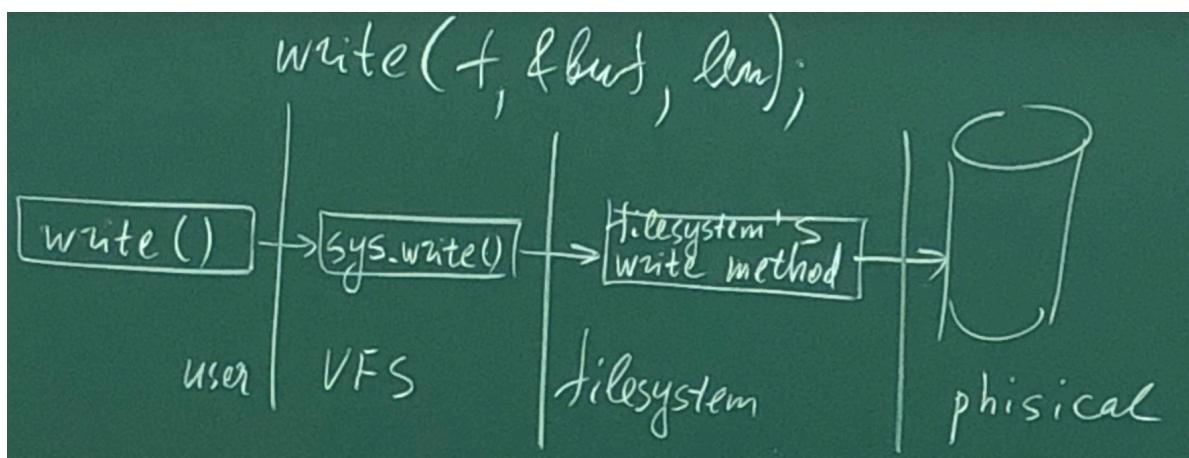
например рассмотрим запрос приложения

`write(f, fbuf, len);`

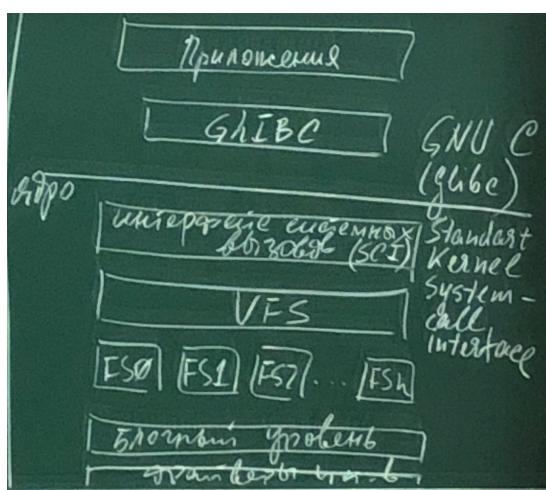
в данном случае данное API записывает `len` байт, которые хранятся в `fbuf`, в текущую  
позицию файла

речь идет о логическом адресном пространстве файла

этот системный вызов обрабатывается в системе по следующей цепочке



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



последнее - драйверы устройств

файловые системы которые поддерживают unix:  
`ext2, ext3, ufs, ntfs` и т.д.

## ВНУТРЯННЯЯ ОРГАНИЗАЦИЯ VFS (Virtual File System)

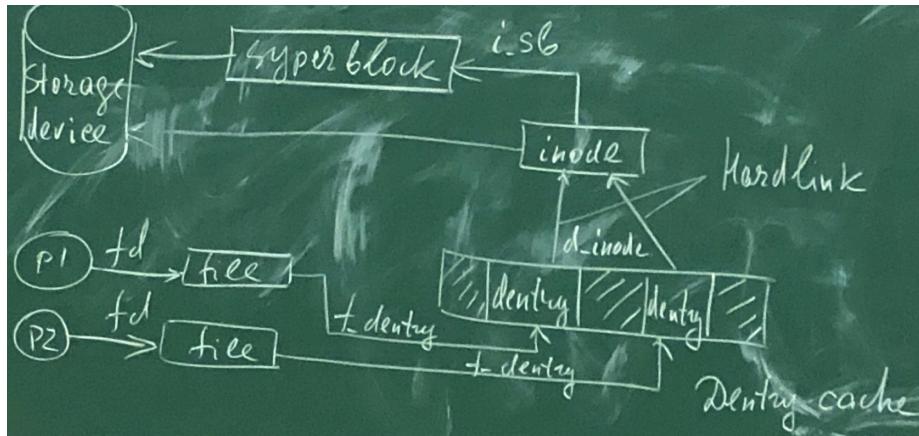
базируется на 4х структурах:

struct

-----

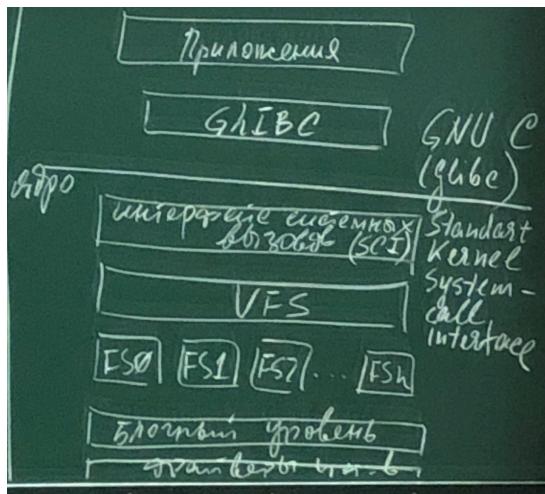
- superblock
- dentry (directory entry)
- inode
- file

между ними существует предельная взаимосвязь



## 23.03.19

чтобы иметь возможность поддерживать большое количество файловых подсистем реализовано  
в unix VFS/vnode  
в linux VFS



VFS представляет уровень абстракции отделяющий posix api от подробностей работы конкретной файловой системы

ключевым моментом здесь является то, что системные вызовы (read, write, ...) работают одинаково независимо от того какая файловая система 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:

1. суперблок
2. индексный узел
3. запись каталога
4. файл

суперблок содержит высокоровневые метаданные о файловой системе

задание: посмотреть что означает приставка 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)

такая структура должна определять параметры для управления файловой системы (суммарное число блоков, корневой inode, ...)

суперблок существует не только на диске

суперблок на диске предоставляет ядру системы информацию о структуре файловой системы (но это не точно)

суперблок в памяти предоставляет информацию необходимую для управления смонтированной файловой системой

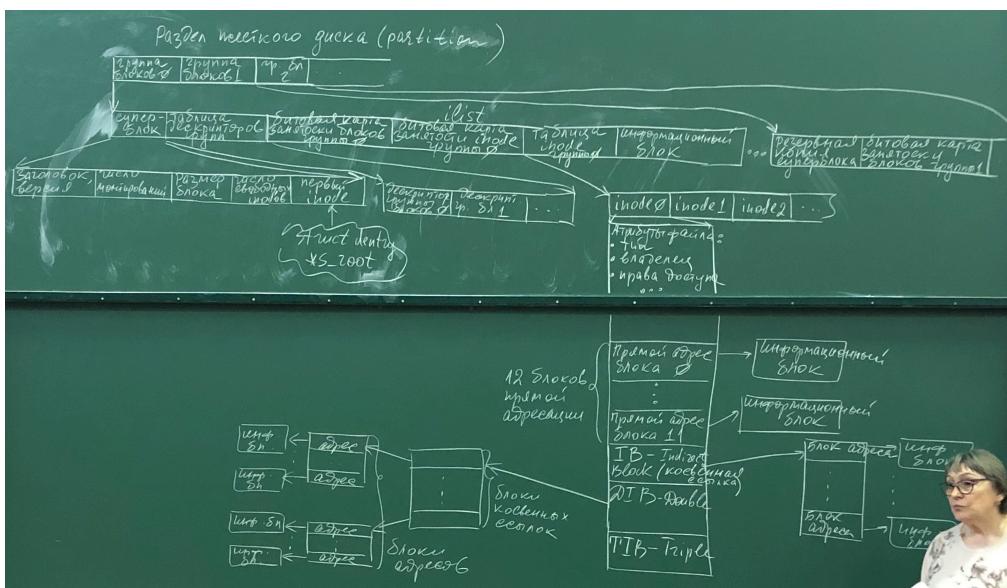
```
<linux/fs.h>
struct super_block
{
    struct list_head s_list; // список суперблоков
    kdev_t s_dev; // указывает устройство на котором находится файловая система
    unsigned long long s_blocksize; // размер блока в байтах
    unsigned char s_dirt; // флаг показывающий, что суперблок был изменен
    unsigned long s_max_byte; //
    struct file_system_type *s_type; // тип файловой системы
    struct super_operations *s_op; // перечислены действия определенные на суперблоках
    ...
    unsigned long s_magic; // магическое число файловой системы
    struct dentry *s_root; // точка монтирования файловой системы // каталог
    монтирования файловой системы
    ...
    int s_count; // счетчик ссылок на суперблок
    ...
    struct list_head s_inodes; // все inode
    const struct dentry_operations *s_d_op; // определяет dentry_operations для
    директорий
    struct list_head s_dirty; // список измененных индексов
    ...
    struct block_device *s_bdev; // драйвер
    ...
    char s_id[32]; // строка имени
    ...
}
```

тк линукс поддерживает большое количество одновременно смонтированных файловых подсистем

то будет большое количество блоков  
и они хранятся в struct list\_head - список суперблоков

суперблок описывает смонтированные файловые системы  
но каждая файловая система описывается структурой struct file\_system\_type

раздел жесткого диска (partition)



файловая система занимает раздел жесткого диска  
поделена на группы блоков  
такая файловая система описывается структурой суперблока  
очевидно  
суперблок описывает файловую систему  
задача файловой системы хранить информацию - файлы  
чтобы иметь доступ к ним  
они должны быть нумерованные  
нумеруются inode'ами  
описывается в struct inode  
эффективнее всего хранить это в виде битовой карты

в соответствии со структурой struct superblock  
там видим dentry \*s\_root (указатель на корневой каталог)  
в системе все является файлом  
все файлы в системе имеют inode

entry - directory entry  
struct dentry создается на лету  
она нигде не хранится  
и она создается на основе информации из inode

дисковый inode хранит информацию о физическом файле и позволяет получить доступ к информации записанной в файле

ОС linux поддерживает файлы очень больших размеров

чтобы получить доступ к блоку необходимо хранить его адрес

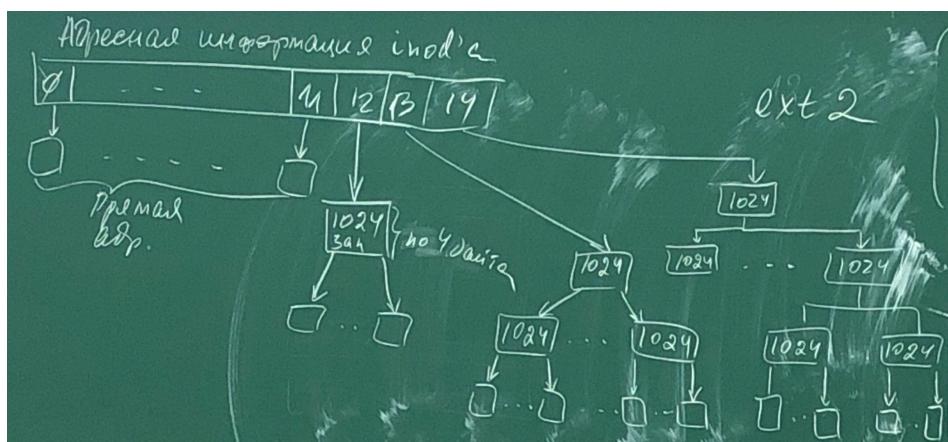
существует 12 блоков прямой адресации  
блок indirect - этот блок ссылается на блок, в котором хранятся адреса блоков по тому же принципу

ghjcnfz rjcdtyyfz flhtcfwbz

видим DIB - двойная косвенная адресация  
TIB - тройная косвенная адресация

Адресная информация inode'a

EXT2



такая косвенная адресация увеличивает время доступа к файлу пропорционально количеству ссылок

операции определенные на суперблоке struct super\_operations

```
<linux/fs.h>
struct super_operations
{
    struct inode *(alloc_inode) (struct super_block *sb); // функция создает и
инициализирует новый объект inode, связанный с данным блоком
    void (*destroy_inode) (struct inode *); // функция удаляет объект inode файла
    void (*dirty_inode) (struct inode *, int flags); // функция вызывается подсистемой vfs,
когда вносится изменение // журналируемые файловые системы (ext3) используют эту
функцию для обновления журнала
    int (*write_inode) (struct inode *, struct writeback_control *wbe); // записывает inode на
диск и одновременно помечает как грязный
    int (*drop_inode) (struct inode *); // сбрасывание // вызывается, когда исчезает
последняя ссылка и vfs ее просто удаляет
    ...
    void (*put_super) (struct super_block *);
    int (*freeze_super) (struct super_block *);
    ...
    int (*remount_fs) (struct super_block *, int *, char *);
}
```

когда файловая система нуждается в выполнении операций 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 суперблок  
и возвращает указатель на новый суперблок или возвращает null, если выделение  
суперблока завершилось аварийно

суперблок описывает информацию необходимую для того чтобы система могла управлять  
смонтированными системами  
любая файловая система описывается структурой file\_system\_type

в этой структуре есть поле

```
struct file_system_type
struct super_block *(get_sb) (struct file_system_type *, int , char *, void *, struct vfsmount);
fstype->get_sb()
```

эта функция вызывается во время монтированная файловой системы  
ядро вызывает эту функцию  
ссылаясь на эту функцию на соответствующий тип  
и эта функция инициализирует поля данных и устанавливает суперблок

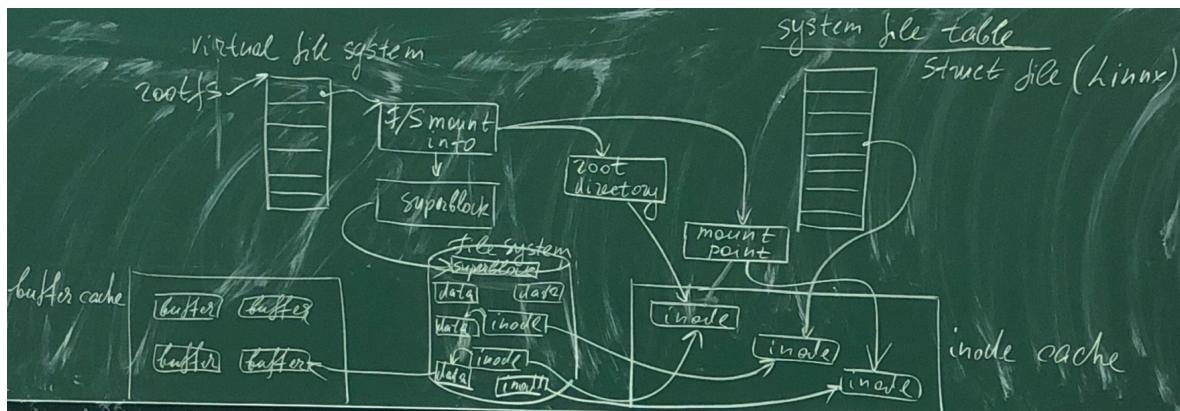
### 30.03.19

в vfs может существовать только 1 структура для каждого типа file\_system\_type  
но систем может быть подмонтировано сколько угодно

inode- дескрипторы физических файлов

если не предпринять определенных действий в системе, а каждый раз обращаться к диску  
то время получения информации из файлов и о файлах будет значительно  
т.к. диск - медленно действующее устройство (с точки зрения быстродействующей  
системы)

информация о файлах кешируется  
иначе доступ к файлам будет слишком долгим



в системе должна быть системная таблица открытых файлов

в системе существует одна таблица всех открытых файлов struct file

открытый файл - это значит обратиться к файлу на диске

struct file - структура, которая позволяет организовать работу с файлом  
inode обеспечивает доступ к данным, которые находятся на физическом носителе

в системе существует в 2x ипостасях:

файл, который лежит на диске (можем посмотреть что файл есть)

в системе существует одна таблица всех открытых файлов

чтобы обеспечить производительность системы вся информация кешируется

система требует чтобы многие файловые структуры были сохранены с ядре резидентно  
(постоянно)

данные сохраняются в различных таблицах

существует file\_system\_type

get\_sb() - в книге у Роберта лава  
сейчас поменяли название этой функции на mount()  
она вызывается когда выполняется монтирование файловой системы  
монтируем файловую системы из командной строки

при монтируем файловой системы создается суперблок  
мы определяем поля, определяем super\_operations  
но все это начинает выполнятся, когда мы начинаем монтирование файловой системы  
(точка входа)

инициализируются соответствующие поля struct superblock  
например

инициализируется обращение к операциям на суперблоке (операции с inode)

в результате будут заполнены поля связанные с конкретными физическими файлами  
и можно начинать работу с файлами, доступ к которым обеспечивает данная файловая  
система

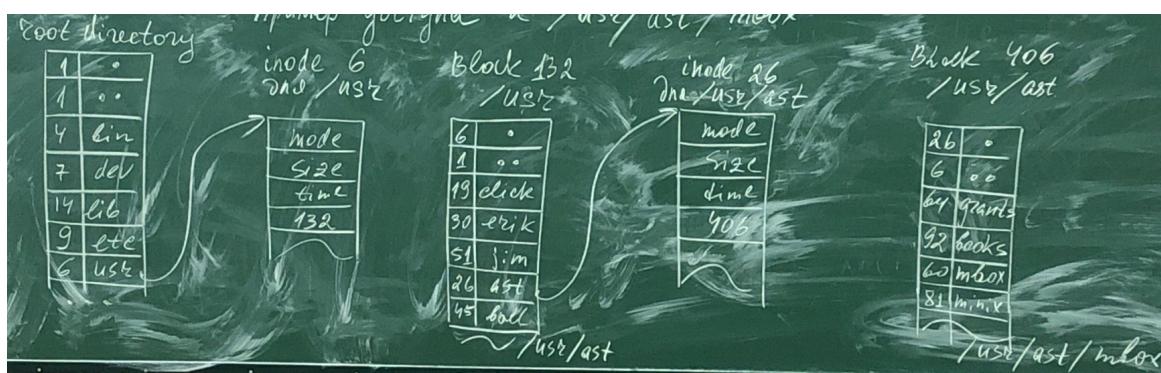
структура struct inode

такие устройства как CD, DVD, флэшки - системы рассматриваются как файлы  
их мы можем увидеть в директории dev  
это устройства, но для системы это файлы

файлы могут иметь имена в виде символьных строк  
имя файла в их родных файловых подсистемах (ext2) не является идентификатором, им  
является я номер inode  
номер inode - метаданные

система для того чтобы получить доступ к файлу  
ищет номер inode в таблице «таблица inode'ов»

пример доступа к /usr/ast/mbox



вся информация в системе находится в соответствующих inode'ах  
действие начинается с корневого каталога  
существует информация, которая позволяет сопоставить имя файла с номером inode  
без этой информации нельзя ничего сделать  
для этого в системе существует директория dentry

```
struct inode
{
    umode_t i_mode;
    unsigned short i_opflags;
    kuid_t i_uid;
```

```

kgid_t i_gid;
unsigned int i_flags;
...
const struct inode_operations *i_op; // указатель на структуру inode_operations
struct super_block *i_sb;
struct address_space *i_mapping;
...
// stat data, not accessed from path walking
unsigned long i_ino;
...
loff_t i_size;
struct timespec i_atime; // обращение
struct time_spec i_mtime; // изменение
struct timespec i_ctime; // изменение параметров управления
...
struct heist_node i_hash;
...
const struct file_operations *i_op;
}

```

```

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 *, umode_t, bool);
    int (*link)(struct dentry *, struct inode *, struct dentry *); // link - операция связанная с
inode'ом
    int (*unlink)(...);
    int (*symlink)(...);
    int (*mkdir)(...);
    int (*rmdir)(...);
    int (*mknod)(...);
    int (*rename)(...);
    ...
}

```

любой системный вызов связан с последовательным вызовом других функций

## 6.04.19

---

объект dentry не сопоставляется ни с какой структурой на диске  
виртуальная файловая система создает этот объект на лету  
из строкового представления пути к файлу, т.е. pass\_name

struct dentry эта структура работает с именами файла

каждая поддиректория это файл  
т.е. интерфейс vfs представляет каталоги как файлы

какая-то страшная информация

объект dentry - это определенный компонент пути (к файлу)  
причем все объекты dentry это компоненты пути включая обычные файлы

компоненты пути (элементы пути) 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

рассорим структуру dentry

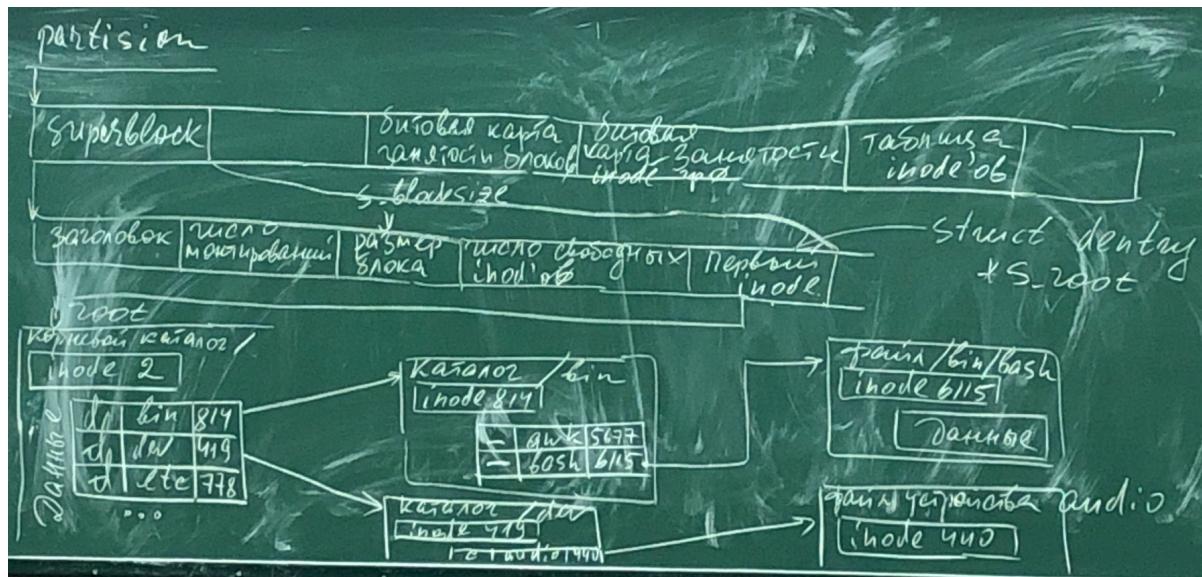
RCU - read-copy-update

```
struct dentry
{
    // RCU lookup touched fields
    unsigned int d_flags; // protected by d_lock
    seqcount_t d_seq; // per dentry seqlock
    struct dentry *d_parent;
    struct qstr d_name; // dehnytry name
    struct inode *d_inode; // связанный inode, если - NULL, то neqative
    unsigned char d_iname[DNAME_INLINE_LEN]; // короткое имя
    ...
    const struct dentry_operations *d_op;
    struct super_block *d_sb; // корневой каталог (root) дерева каталогов файловой
    системы (фс)
    unsigned long d_time;
    void *d_fsdta; // данные спец для фс
    struct list_head d_lru; // список LRU
    ...
    struct list_head d_subdirs;
}
```

объект dentry должен содержать указатель на соответствующий inode

актуальный объект dentry может находиться в одном из 3х состояний:

- used
- unused
- negative



после того как виртуальная файловая система прошла по пути (спустилась поному имени файла) и для каждого элемента пути создан объект dentry  
причем этот путь пройдет от начала до конца

в результате полученная по пути информация сохраняется в dentry кэше (dcache)

dcache состоит из 3х частей:

- список используемых объектов dentry, которые связаны с inode'ами полем d\_inode

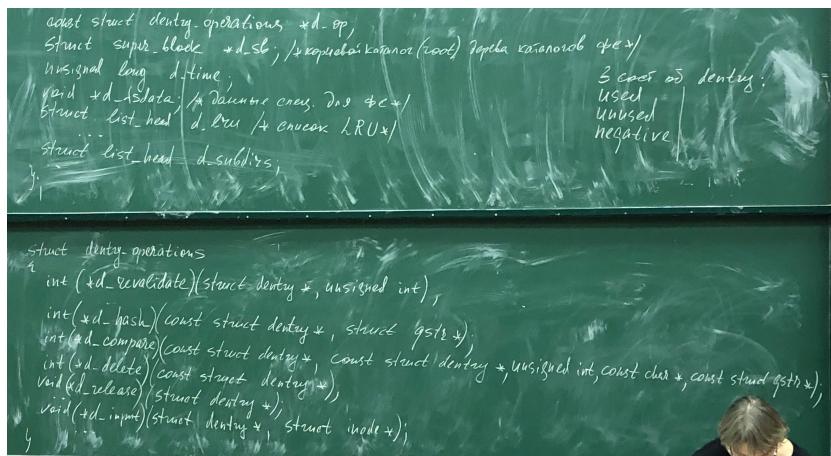
т.к. отдельно взятый inode может иметь много link, то может быть много объектов dentry связанных с этим inode

- двухсвязный список LRU - это список неиспользуемых и отрицательных объектов dentry  
этот список отсортирован по времени  
соответственно в начале этого списка будет самый новый объект dentry

- hash\_table - хеширует функции для быстрого определения пути ....

эта таблица реализована в виде dentry\_hash\_table массива  
каждый элемент является указателем на список dentrys хешированному по тому же самому имени

### dentry\_operations



d\_hash задает величину hash для заданной....

много текста который я не писала)

большинство файловых систем определяют ее как default  
которая выполняет простое сравнение строк

## inode-кэш

представляет из себя

1. глобальный хеш-массив inode-hashtable в котором каждый inode хешируется по

значению указателя на суперблок и 32x разрядному номеру inode

при отсутствии суперблока inode добавляется к списку анонимных inode  
примерами таких inode может служить сокеты

2. глобальный список inode-in-use

в этом списке содержится допустимые inode с i\_count > 0 и i\_nlink > 0

в этот же список записываются вновь созданные объекты inode

3. глобальный список inode\_unused

который содержит допустимые inode с i\_count = 0

4. список для каждого суперблока (sb->s\_dirty)

который содержит inode с i\_count > 0 и i\_nlink > 0

т.е. измененные inode

5. SLAB cache inode\_cachep

объекты inode могут вставляться, освобождаться и изыматься из SLAB кеша

по отношению другим кешам любой объект inode может находиться только в одном из этих кешей

## распределения памяти SLAB в unix

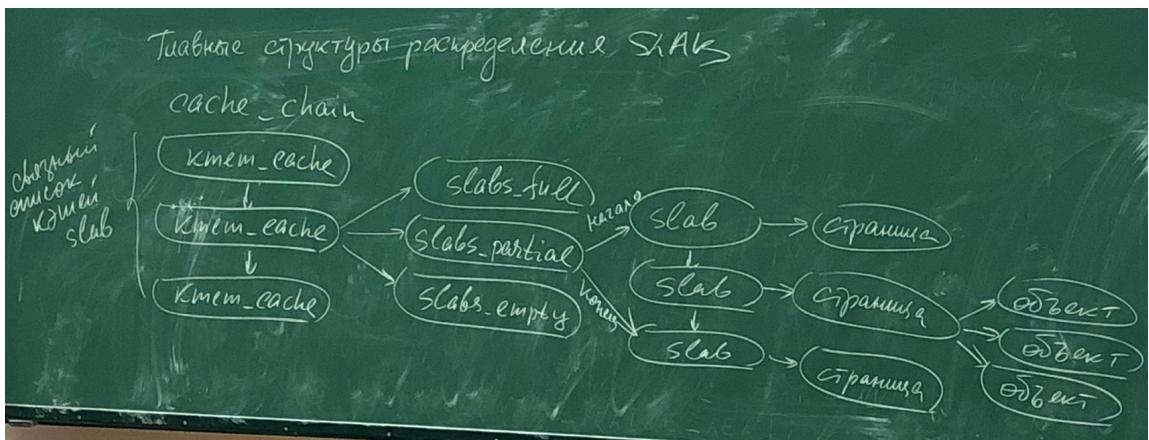
SLAB аллокатор - это кеширующий аллокатор который позволяет выделять блоки памяти (одного и того же размера для данных одного и того же типа)

идея джера Бонвика базируется на том

что количество времени необходимое на время инициализации обычного объекта в ядре превышает количество времени которое необходимо для его выделения и освобождения

если надо выделить участок памяти для объекта inode мы можем воспользоваться уже проинициализированным участком памяти того же размера

главные структуры распределения SLAB



### 13.04.19

---

в адресном пространстве оперативной памяти выделяются блоки одного и того же размера  
для того чтобы не инициализировать объекты заново

в случае распределения slab участки памяти подходящие для размещения объектов данных определенного типов и размера  
определенны заранее

распределитель аллокатор slab хранит информацию о размещении этих участков, которые известны как кеши

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  
в следствии этого  
обращение к менеджеру памяти выполняется реже  
а запросы пользователя удовлетворяются быстрее  
но самым главным выигрышем является не скорость выделения памяти, 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, и т.д.

при освобождении объекта скорее всего имеют правильные значения  
и в силу этого при повторном выделении не нуждаются в повторной инициализации части полей

в настоящее время линут имеет 3 вида slab аллокаторов:

- SLAB
- SLUB
- SLOB

интерфейс они предоставляют одинаковый

рассмотрим некоторые функции slab аллигатора

к лабораторной работа ОТКРЫТЫЕ ФАЙЛЫ составить алгоритмы системного вызова open  
графически

```
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_once);
    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 *struct kmem_cache *, insigned long), void (*dtor)(void *struct
kmem_cache *, insigned long))
```

\*name - имя кэша (можно посмотреть в файловой системе proc для идентификации /proc/slabinfo)

align - выравнивание

ctor/dtor - определяют необязательные объекты конструктор/деструктор  
callback функции

\*kmem\_cache\_create - не выделяет память кэшу

```
void aufs_inode_cache_destroy(void)
{
    rCU_barrier(); // RCU - read/-copy-update (распространенный в ядре механизм
    синхронизации)
    kmem_cache_destroy(aufs_inode_cache);
    aufs_inode_cache = NULL;
}
```

void kmem\_cache\_dastroy(struct kmem\_cache \*cachep);

перед вызовом функции destroy кэш должен быть пуст

барьер здесь поставлен для безопасного освобождения памяти  
для lock\_free алгоритмов

kmem\_cache\_dastroy уничтожает slab аллокатор

rcu\_barrier()  
блокирующая функция  
эта 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 *count i=(struct aufs_inode *) kmem_cache_alloc(anfs_inode_cache,
GFP_KERNEL); // флаг указ, что надо выделить память из RAM ядра
    if (!i)
        return NULL;
    return &i->ai_inode;
}
```

? void kmem\_cache\_alloc(struct kmem\_cache \*cachep, gft\_t flags);

kmem\_cache\_alloc - функция возвращает объект из кэша  
если кэш пуст, то функция может вызвать функцию cache\_alloc\_refill()

освобождение inode  
void aufs\_inode\_free(struct inode \*inode)
{
 call\_rcu(&inode->i\_rcu, aufs\_free\_callback);
}

lock\_free алгоритмы (свободные от блокировок)

## БЛОКИРОВКИ - ЗЛО

они замедляют выполнение кода

проблемы lock\_free алгоритмов:

без блокировок 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

функция call\_rcu откладывает выполнение функции aufs\_free\_call до тех пор пока это не станет безопасно

```
#cat/proc/slabinfo
```

```
...
inode_cache 77005 14792 480 1598 1849 1
```

77005 - число активных объектов

14792 - общее число объектов

480 - доступные объекты

1598 - размер каждого объекта в байтах

1849 - число страниц относящихся по крайней мере к одному активному объекту

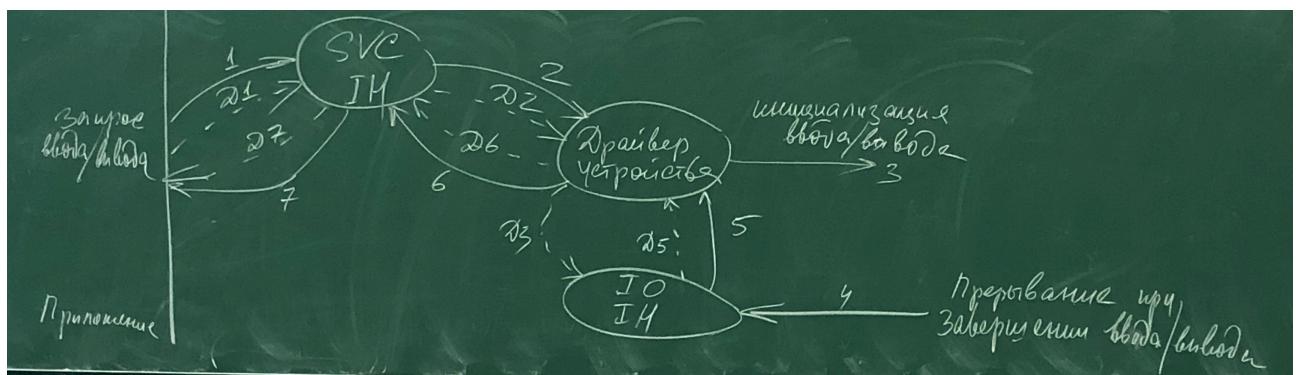
1 - количество страниц выделенных данному slab

```
dentry_cache 5469 5880 128 183 196 1
```

## ПРЕРЫВАНИЯ

аппаратные прерывания

диаграмма из книги шоу



SVC - super visor call

IH - interrupt handler

в результате запроса ввода вывода система переходит в режим ядра

процесс в синхронном блокирующем вводе-выводе будет блокирован

процесс блокируется  
задействуются разные части кода ядра системы

контроллер - устройство в составе некоторого устройства  
адаптер - устройство на материнской плате

на каждом этапе данные получают определенный вид  
драйвер получает определенный формат данных который должен преобразовать в  
команду понятную контроллеру устройства  
получив команду контроллер начинает управлять устройством  
**по завершении ввода-вывода формируется сигнал прерывания**

сигнал прерывания который приходит на контроллер  
может быть замаскирован (игнорируется)  
если не замаскирован то контроллер формирует сигнал который .... на один из входов ...

в конце цикла выполнения каждой команды контроллер проверяет наличие сигнала  
прерывания на своем входе

если сигнал поступил, то процессор переходит на выполнение обработчика прерывания и  
это обработчик аппаратного прерывания

все прерывания ввода/вывода - аппаратные + прерывания от системного таймера

все аппаратные прерывания выполняются на очень высоком уровне приоритета

аппаратные прерывания делятся на 2 части:

- top half(верхняя половина)
- bottom half(нижняя половина)

в windows эта же идея реализованная с помощью DFC (отложенный вызов процедуры)

аппаратные прерывания принято делить на быстрые и медленные

быстрые прерывания выполняются как единое целое

в современном линуксе быстрым прерыванием является только прерывание от системного  
таймера

нижняя половина прерываний выполняется как отложенное действие

фактически аппаратным прерыванием является верхняя половина

в ее задачи входит:

- получение данных из регистра устройства
- помещение этих данных в буфер ядра

завершается аппаратное прерывание инициализацией отложенного действия

например путем подстановки соответствующей нижней половины в очередь на выполнение  
для этого вызывается функция планирования

## 20.04.19

---

механизм аппаратных прерываний - естественный асинхронный параллелизм

пребывания от нижних устройств

общая модель обработки аппаратных прерываний является принципиально архитектурно зависимой

(

например в ms dos использовался программируемый контроллер, который формировался из 2x последовательно соединенных микросхем  
и в этой схеме формировался вектор прерывания для адресации вектора прерывания по схеме адресации 4 сегмента

т.е. номер аппаратного прерывания использовался в качестве смещения

в 32x разрядных системах использовался апиг (пиг - программируемый контроллер прерываний)

аспид формировал вектор прерываний который являлся смещением к 8ми байтовому дескриптору прерываний

)

ВСЕ ЧТО В СКОБКАХ ХРЕНЬ КАКЯ-ТО ОНА ПОСТОЯННО ПЕРЕДУМЫВАЛА ИСПРАВЛЯЛА ФРАЗЫ И ГОВОРИЛА ОТРЫВКАМИ)

основная функция аппаратных прерываний - позволить преферируемым устройствам взаимодействовать с процессором информируя его о завершении некоторых действий или о возникновении ошибочных ситуаций или каких-то других событий которые требуют внимания со стороны системы

в результате возникновения прерываний используя соответствующую аппаратно зависимую схему обработчика

очень часто такой обработчик называется ISR (процедура обслуживания прерывания)

обработчик прерываний выполняется в режиме ядра в системном контексте т.к. .... не имеет никакого отношения к возникшему в системе пришиванию

его обработчик не должен обращаться ....

поэтому 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 тики текущего процесса и поэтому нуждается в доступе к его структуре

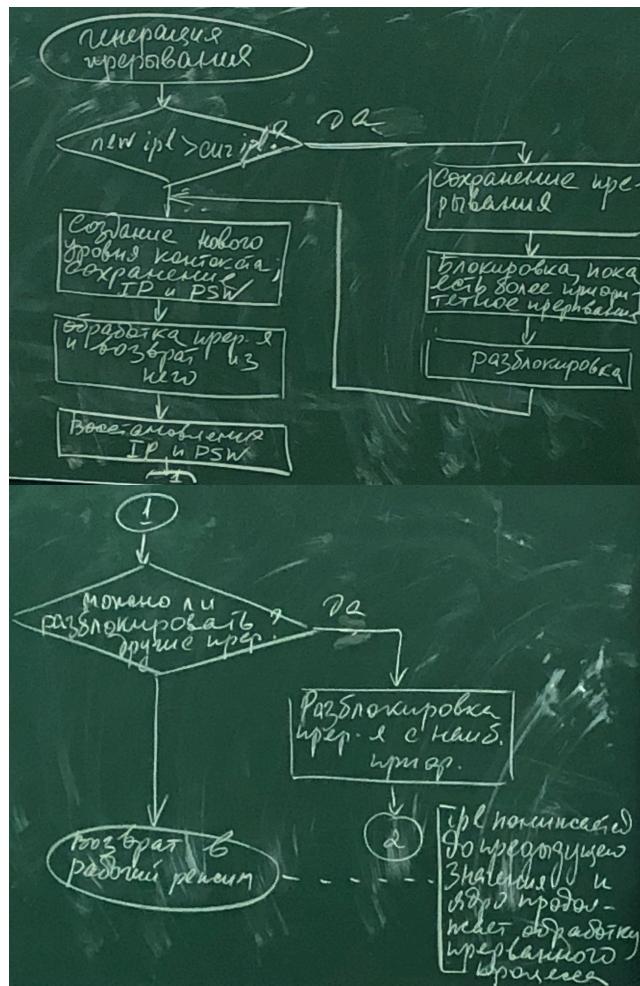
например в юникс бсд это struct proc

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 существуют программные прерывания (системные вызовы) и исключения

очевидно что разные типы прерываний имеют разную важность для работы системы кроме того они могут иметь разный объем кода (большой)

под большим объемом понимается объем который требует вычислений в несколько тиков в силу этого в юникс/линукс вводятся уровни приоритетов прерываний irq



аппаратные прерывания в любой системе имеют очень высокие уровни приоритета от системного таймера имеет наивысший приоритет

пребывания от внешних устройств являются высокоприоритетными  
но от клавиатуры и мыши еще более высокий уровень привилегий

каждое устройство имеет 1 драйвер  
если это устройство использует прерывания этот драйвер регистрирует 1 обработчик прерывания

драйверы могут регистрировать обработчик прерывания и разрешить определенную линию прерывания для обработки 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);
```

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

важно чтобы эта функция освобождала обработчик прерываний именного того устройства драйвер которого установил

unsigned int irq - номер прерывания  
для некоторых устройств таких как системный таймер и клавиатура это величина обычно устанавливается аппаратно  
для большинства других устройств она определяется программно и динамически  
irq - запрос на обработку прерываний

\*handler - указатель на обработчик прерываний который обслуживает данное прерывание именно эта функция будет вызвана когда возникнет соответствующее прерывание возвращает значение типа irgreturn\_t

```
enum irgreturn
{
    IRQ_NONE = (0 << 0),      // ghthsdfybt ,skj yt jn cjjndtnnde.otuj ltfqcf bkb ghthsdfybt yt
    IRQ_HANDLER = (1 << 0), // прерывание было обработано соответствующим
    IRQ_WAKE_THREAD = (1 << 1), // обработчик запрашивает пробуждение нити
    // обработчика
};

typedef enum irgreturn irgreturn_t
#define IRQ_RETVAL(x) ((x) ? IRQ_HANDLER : IRQ_NONE)
```

unsigned long irqflags - старое название SA\_\* он изменился на IRQF\_\*

флаги:

- IRQF\_SHARED - этот флаг разрешает разделение линии прерывания, т.е. совместное использование линии IRQ разными устройствами
- IRQF\_PROBE\_SHARED - устанавливается абонентами, т.е. вызывающими данное действие, если они предполагают возможность нестыковок при совместном использовании линии IRQ
- IRQF\_TIMER - флаг маскирующий данное прерывание как прерывание от таймера
- IRQF\_PERCPU - прерывание закрепленное монопольно за конкретным процессором

const char \*devname - ASCII текст представляющее устройство связанное с данным прерыванием  
эта строка используется в /proc/irq  
а также в /proc/interrupts для связи с пользователем

void \*dev\_id - используется для разделения линий прерываний  
когда обработчик прерывания освобождается dev\_id обеспечивает уникальные cookie-файлы  
чтобы выполнить удаление только нужного обработчика прерывания с соответствующей линии прерывания  
обычно используется указатель на структуру специфичную для устройства

int request\_irq в случае успеха возвращает 0  
ненулевая величина означает ошибку  
и в этом случае обработчик прерывания не регистрируется  
обычная ошибка EBUSY которая означает что данная линия прерывания уже используется  
или вы не указали флаг IRQF\_SHARED

быстрые прерывания - процедура обработки короткая, поэтому прерывает текущую активность на короткий промежуток времени

при выполнении быстрых 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 быстрое прерывание является единственным обозначаемым флагом IRQF\_TIMER

медленное прерывание делится на 2 части:

- верхняя половина (top half) (обработчик аппаратного прерывания - быстрое прерывание для которого характерны все свойства быстрых прерываний) (блокирует всю активность на локально процессоре (выполняется при запрещенных прерываниях))
- нижняя половина (bottom half) (отложенное действие которое выполняется сразу после того как завершилась нижняя половина) (может быть прервана любым обработчиком аппаратного прерывания)

в отличии от реально обработчика быстрого прерывания top half перед своим завершением должна инициализировать последующее выполнение bottom half  
а именно должна поставить соответствующий обработчик в соответствующую очередь бла бла бла

сам обработчик верхней половины заканчивается interrupt return

в настоящее время обработчики нижних половин бывают 3х видов:

- softirq (мягкие прерывания) - определяются статически во время компиляции ядра. в файле linux/interrupt.h в структуре
- ```
struct softirq_action
{
```

```
    // ф-я которая должна выполняться
    void (*action)(struct softirq_action *);
```

```
};
```

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

```
static struct softirq_action* softirq_vec[NR_SOFTIRQS], где NR_SOFTIRQS = 32
```

имеется возможность создать 32 обработчика softirq

softirq определяется статически, т.е. при запуске системы

| индекс               | приоритет | значение                         |
|----------------------|-----------|----------------------------------|
| HI_SOFTIRQ           | 0         | высокоприор softirq таймеры      |
| TIMER_SOFTIRQ        | 1         | таймеры                          |
| NET_TX_SOFTIRQ       | 2         | отправка сетевых пакетов         |
| NET_RX_SOFTIRQ       | 3         | прием сетевых пакетов            |
| BLOCK_SOFTIRQ        | 4         | блочное устройство               |
| BLOCK_IOPOLL_SOFTIRQ | 5         | мультиплексирование ввода/вывода |

| индекс          | приоритет | значение    |
|-----------------|-----------|-------------|
| TASKLET_SOFTIRQ | 6         |             |
| SHED_SOFTIRQ    | 7         | планировщик |

- tasklet
- workqueue (очереди работ)

## 27.04.19

---

только рююююмка водки на столе  
которой мне так не хватает :)

когда ядро выполняет обработчик отложенного прерывания вызывается функция action с указателем на соответствующую структуру softirq\_action в качестве аргумента

например если переменная xxx\_softirq содержит указатель на элемент массива softirq\_vec,  
static struct softirq\_action отображает индекс softirq на имя

```
softirq_vec[NR_SOFTIRQS]
char *softirq_to_name[NR_SOFTIRQS] = {«HI», «TIMER», «NEX_Tx», «NET_RX», «BLOCK»,
«BLOCKIOPOLL», «TASK», «SCHED», «NRTIMER», <> «RCU»};
```

если добавляется новый softirq, то обновляется softirq\_to\_name в <kernel/softirq>  
в результате ядро вызовет функцию «обработчик» соответствующего отложенного  
прерывания  
в следующем виде

```
xxx-> softirq->action(xxx_ softirq);
```

передача функции action всей структуры а ен конкретного значения может показаться  
странный

но этот трюк обеспечивает возможность дополнения структуры без необходимости  
внесения изменений в каждый обработчик softirq

обработчик softirq может получить только значения если ему требуется только  
разыменование и считывание данных

посмотреть softirq в работающей системе можно используя следующую команду  
#cat/proc/softirq

добавить новый уровень обработчика xxx\_SOFT\_IRQ можно только перекомпилировав  
ядро

отложенное прерывание с меньшим номером выполняется раньше, т.е. его приоритет выше

softirq - отложенные действия, действия которые завершают ... соответствующего  
прерывания

для создания нового уровня softirq нужно:

1. определить новый индекс отложенного прерывания вписав его в константу в виде  
xxx\_SOFT\_IRQ в перечисление. оно должно иметь уровень хотя бы на единицу меньше  
tasklet\_softirq, иначе зачем переопределять новый уровень если можно переопределять  
tasklet

2. во время инициализации модуля должен быть зарегистрирован обработчик  
отложенного прерывания с помощью вызова open\_softirq() в следующем контексте:  
void xxx\_soft\_handler(void \*data)  
{...}

```
void __init roller_init()
{
    ...
    request_irq(irq, xxx_interrupt, "xxx", NULL);
    open_softirq(XXX_SOFT_IRQ, xxx_soft_handler, NULL);
}
```

т.е. функция open\_softirq устанавливает обработчик

xxx\_soft\_handler - обработчик  
NULL - значение поля данных

очевидно, что функция обработчик softirq должна соответствовать правильному прототипу  
void xxx\_soft\_handler(unsigned long data);

3. зарегистрированное softirq должно быть поставлено в очередь на выполнение  
для этого оно должно быть возбуждено с помощью функции raise\_softirq(); - генерация  
отложенного прерывания

обычно обработчик аппаратного прерывания (top half) перед возвратом управления  
возбуждает свой обработчик отложенного прерывания

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

проверка ожидающих выполнения обработчиков прерывания и их запуск осуществляется в  
следующий случай:

1. при возврате из аппаратного 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 синевой системе)

независимо от метода вызова softirq его выполнение осуществляется в функции do\_softirq()  
rjnjhffz d wbrkt ghjdthztn yfkbxbt jnkj;tyys[ ghthsdfybq

несмотря на то что у softirq есть приоритеты  
softirq никогда не вытесняет другой softirq

единственное событие которое может вытеснить softirq - 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 реентерабельный  
а если есть критический участок. то он должен быть защищен

демон ksoftirqd - это поток ядра каждого процессора, т.е. per-cpu

когда машина нагружена мягкими прерываниями (softirq) 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 приходит очень быстро одно за  
другим и в результате ОС не может закончить обслуживание одного до прихода другого  
например это может произойти, когда сетевая карта с высокой скоростью получает  
пакеты в течении короткого промежутка времени и ОС не может с этим справиться и  
создает очередь для последовательной обработки softirq с помощью специального  
процесса  
который и называется ksoftirqd

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

## TASKLET тасклеты

тасклеты - частный случай реализации softirq  
но тасклеты - это отложенные прерывания для которых обработчик(тасклет) не может выполняться одновременно на нескольких процессорах в отличии от softirq

отсюда для тасклетов не требуется использовать средства взаимоисключения

тасклеты проще всего понимать как простые в использовании

разные тасклеты могут выполняться параллельно на разных процессорах  
но 2 тасклета одного типа параллельно выполняться не могут  
в силу этого тасклет является компромиссом между производительностью и простотой использования

тасклеты могут быть зарегистрированы в системе как статически так и динамически

```
<linux/interrupts.h>
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state; // текущее состояние
    atomic _t count; // счетчик ссылок на тасклет // если = 0 то тасклет разрешен и
может выполняться если он помечен как ждущий выполнения, иначе тасклет запрещен и
выполняться не может
    void (*func)(unsigned long); // обработчик
    unsigned long data;
};

enum:
0
TASKLET_STATE_SCHED // тасклет запланирован
TASKLET_STATE_RUN // тасклет выполняется
```

видим что реализуется связанный список тасклетов  
у тасклета есть состояние оно определяется как enum

для того чтобы запланировать тасклет на выполнение должна быть вызвана функция tasklet\_schedule() эта функция вызывается в конце соответствующего обработчика аппаратного прерывания

для оптимизации тасклет выполняется на том процессоре на котором выполнялся обработчик прерывания который его запланировал

аналогично softirq планирование тасклета на выполнение выполняет interrupt\_handler()

j,hf,jnxbr ghthsdfybz dsspsdftncz если возникает прерывания от нижнего устройства

статически тасклеты создаются ... это DECLARE\_TASKLET(name, func, data);  
DECLARE\_TASKLET\_DISABLED(name, func, data)

макросы статически создают экземпляр структуры тасклета  
с именем name вызываемой функции func с какими-то данными

первый макрос создает тасклет у которого поле count = 0 => этот тасклет разрешен  
2й создает count = 1 => такой тасклет запрещен

тасклеты могут создаваться динамически  
при таком создании объявляется указатель на структуру  
`struct tasklet_struct *mytasklet;`

для инициализации тасклета вызывается функция `tasklet_init(mytasklet, tasklet_handler, data);`

`tasklet_handler` имеет следующий прототип

```
// прототип tasklet_handler
void tasklet_handler(unsigned longdata);
```

в обработчиках тасклетов нельзя использовать семафоры, т.к. тасклеты не могут блокироваться

если в тасклете используются общие с обработчиком прерываний или другим таскаемом данные, то они должны быть защищены спин-блокировкой

планирование тасклетов

тасклеты могут быть запланированы на выполнение с помощью 2x функций

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

этим функциям передается единственный аргумент - указатель на структуру `tasklet_struct`

например

```
tasklet_schedule(&irq_tasklet);
```

запланированные на выполнение тасклеты находятся в 2x связанных списках  
обычные тасклеты будут находиться в связанном списке `tasklet_vec`  
а высокоприоритетные в связанном списке `tasklet_hi_vec`

оба списка состоят из экземпляра структуры `tasklet_struct`

после того как тасклет запланирован он будет запущен только 1раз  
даже если он был запланирован на выполнение несколько раз

для отключения заданного тасклета используется функция `tasklet_disable();`  
`tasklet_disable_nosync();`

1я функция не сможет отменить тасклет которая уже выполняется

2я сможет прервать выполнение тасклета  
для активизации тасклета используется функция `tasklet_enable()`

на тасклетах определены специальные функции блокировки  
`tasklet_trylock()`  
`tasklet_unlock()`

не смотря на то что тут написано lock  
внутри этой функции написана команда ....

```
static inline int tasklet_trylock(struct tasklet_struct *t) {
    return !test_and_set_bit(TASKLET_STATE_RUN, &(t)->state);
}

static inline void tasklet_unlock(struct tasklet_struct *t)
{
    smp_mb_before_atomie();
    clear_bit(TASKLET_STATE_RUN, &(t)->state);
}
```

## 11.05.19

быстрое прерывание - от системного таймера (выполняется от начала до конца)  
медленное - от внешних устройств (сначала выполняется аппаратное прерывание, потом ....)

IRQ !!! особенно девочкам выучить что это такое (было в прошлом семестре) (на экзамене спрашивается)

dpc

в системах различаются 3 типа отложенных действий:

1. soft\_irq
2. tasklet
3. workqueue (очереди работ)

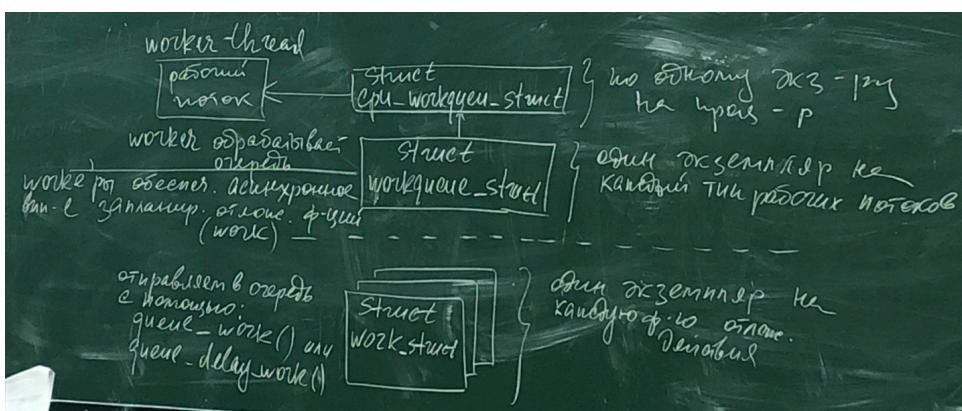
### WORKQUEUE (очереди работ)

отличие от тасклетов

1. тасклеты выполняются в контексте прерывания и в результате код тасклета должен быть атомарным (неделимым); в отличие от тасклетов код очередей работ выполняется в контексте специального потока ядра  
в результате очереди работ могут блокироваться
2. тасклеты всегда выполняются на процессорах которых они были запланированы  
очереди работ по умолчанию выполняются точно также
3. код ядра может запросить чтобы выполнение функции очереди работы было отложено на неопределенное время  
тасклеты выполняются всегда после выполнения аппаратного прерывания

т.к. очереди работ выполняются в контексте специального рабочего ядра => очереди работ не должны быть атомарным

с очередями работ связано несколько структур:



```
struct work_struct
{
    atomic_longg_t data;
    struct list_head entry;
    work_funct_t func;
    #ifdef CONFIG_LOCKED struct lockder_map;
    #endif
};
```

```

struct workqueue_struct // определена в <linux/workqueue.h>
{
    unsigned int flags;
    union{
        struct epu_workqueue_struct_perepu *perpu;
        struct epu_workqueue_struct *single;
        unsignd long v;
    } epu wq; //
    struct list_head list; // список всех очередей работ
    struct mutex flush_mutex; // защищает wq flushing // flush связывается с файлами
    int work_color; // СТРАШНАЯ ТАЙНА: СИСТЕМА НЕ РАСОЗНАЕТ ЦВЕТА!!!! ОНА НЕ
ДУМАЕТ!!!!!
    int flush_color;
    atomic_t nr_cwqs_to_flush;
    struct wq_flusher *first_flushes;
    struct list_head flusher_queue;
    struct list_head flusher_overflow;
    mayday_mask_t mayday_mask;
    struct worker *rescuer;
    int nr_drainers;
    int saved_max_active; // связано с количеством работ
    ...
    char name[];
} 'nj jxthtlm r ghjwtccjhe
в очереди работ находится то что нужно выполнить

```

очереди работ должны быть явно созданы до использования

для создания очередей работ используется функция  
int alloc\_workqueue(char \*name, unsigned int flas, int max\_active); начиная с ядра v2.6.36  
переписывание кода связано с увеличением работы пропускной системы  
начиная с этой версии workqueue не имеет выделенных потоков и служит контекстом для  
представления выделенных задач (хрень какая-то мб неравильно)

не создаются потоки которые используют это имя

char \*name - имя очереди  
unsigned int flas - определяют особенности выполнения  
int max\_active - количество задач которые могут одновременно выполняться на сри

```

struct cpu_workqueue_struct
{
    spinlock_t lock;
    long remove_sequence;
    long insert_sequence;
    struct list_head worklist;
    wait_queue_head_t more_work;
    wait_queue_head_t done;
    struct workqueue_struct *wq;
    task_t *thread;
    int run_depth;
};

```

```

workqueue flas
enum
{
    WQ_UNBOUND, // указывает, что workqueue не привязывается к конкретному сри
    WQ_FREEZABLE, // такая очередь работ может быть приостановлена (заморожена)
    WQ_MEM_RECLAIM, // используется для того чтобы возвращать обратно память
    WQ_HIGHPRI, // высокий приоритет // если установлен этот флаг, то очередь
помещается в высокоприоритетный рабочий пул (workerpool), который обслуживает
рабочие потоки с повышенным приоритетом
    WQ_CPU_INTENSIVE, // имеет смысл только для привязанных очередей и означает
отказ от дополнительной организации параллельного выполнения
    WQ_SYSFS, // что-то что я не поняла)

    ...
    WQ_NON_REENTRANT, // этот флаг больше не существует, т.к. все workqueue не
реентерабельны, т.е. любой рабочий элемент гарантированно будет выполняться не
более чем одним вокером в любой момент в ревени во всей системе в целом
    // хз зачем про него написали если его больше нет

    ...
}

```

очередь которая запускает задание как часть процесса при остановке возобновления не должна использовать флаг WQ\_FREEZABLE

для того чтобы поместить задачу (task) в очередь работ (workqueue) нужно заполнить структуру work\_struct

это может быть сделано во время компиляции статически макросом  
`DECLARE_WORK(name, void (*function)(void *));`

в старых версиях был еще параметрах (void \*data)  
`*function - yb;uzz jkjdbyf j,hf,jnxbrf gghthsdfybz`  
name - имя структуры которое было объявлено

work\_struct может быть задана динамически в процессе выполнения  
для этого используются макросы:

- `INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);` // используется если структура work\_struct определяется в первый раз
- `PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);` // делает почти ту же работу но не инициализирует указатели, используемые чтобы связать work\_struct с workqueue // используется если нужно только изменить структуру

существует 2 функции для отправки работы в очередь работ:

- `int queue_work(struct workqueue_struct *queue, struct work_struct *work);` //
- `int queue_delay_work(struct workqueue_struct *queue, struct work_struct *work, unsigned long delay);` // используется для того чтобы отложить выполнение какой-то работы на заданный интервал времени

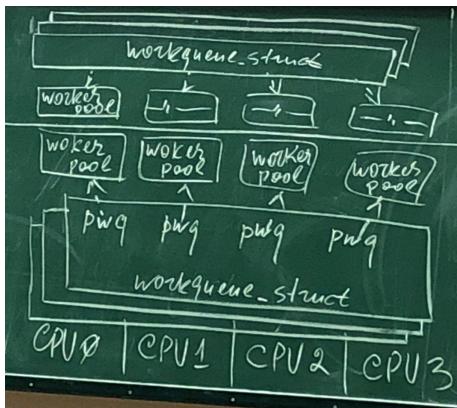
эти функции ставят соответствующий задания (функции) в определенную очередь

функция будет выполняться в контексте потока worker и в силу этого она может спать (если надо), но разработчик должен представлять, понимать как этот сон может повлиять на какие-то другие задачи в этой же очереди работ

несмотря на то что в указанных функциях в качестве параметра указана очередь на самом деле работы «кладутся» не в сами очереди работ, а в список-очередь структуры worker-pool

эта структура является главной в организации механизма очередей работ хотя пользователь может о ней ничего не знать

эту связь можно представить с помощью иллюстрации:



для каждого сри выделяется статически worker

в то время как бла бла бла

когда создается workqueue для каждого сри выделяется служебный пул workqueue rwq  
каждый такой pool workqueue ассоциирован с workqueue  
который выделен на том же сри и соответствует приоритету типу очереди

через них workqueue взаимодействует с worker\_pool

workerы выполняют work из worker pool без разбора не различая каким worker pool они  
принадлежали изначально (мб неправильно я запуталась)

для непривязанных очередей worker pool выделяется динамически

все очереди можно разбить на классы эквивалентности по их параметрам  
и для каждого такого класса создается worker pool

доступ к ним осуществляется с помощью специальной хэштаблицы

где ключ - специальный параметр

значение - worker pool

для освобождения работы (для завершения) используются функции:

для принудительного завершения:

- int/bool flush\_work(struct work\_struct \*work);
- int/bool flush\_delay\_work(struct delayed\_work \*dwork);

для отмены работы (work):

- int/bool cancel\_work\_sysnc(struct work\_struct \*work);
- int cancel\_delay\_work(struct work\_struct \*work); // вернет не 0, если вход был отменен до начала выполнения при этом ядро гарантирует что выполнение данного входа не будет инициализировано после cancel\_delay\_work; если возвращает 0 , то функция начала выполняться и может еще выполняться после вызова cancel\_delay\_work

после того как завершаются действия связанные с очередью работ от нее можно  
освободиться используя функцию

void destroy\_workqueue(struct workqueue\_struct \*queue)

**P.S.** функция (отложенное действие) не может получить доступ к пространству  
пользователя, т.к. она выполняется внутри потока ядра.

## 18.05.19

порт - это адрес

io mapping - отображение устройства адресное пространство ввода вывода

memory mapping- одно устройство (видеокарта) которое отображается на адресное пространство

взаимодействие процессов с помощью сокетов всегда происходит по модели клиент-сервер

сокеты поддерживаются файловой системой

каждый сокет это открытый файл

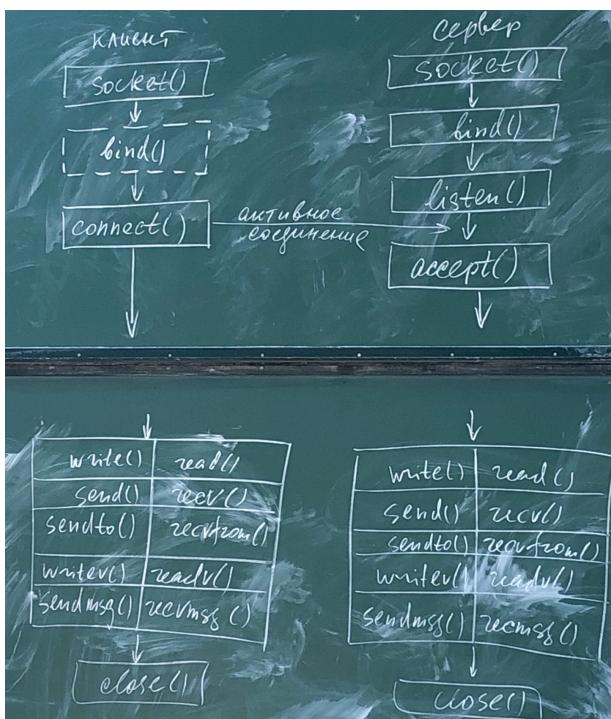
описывается структурой struct file => любой сокет имеет inode, но он не дисковый

все функции которые в качестве аргумента принимают файловые дескрипторы могут принять дескрипторы сокетов

установление локального или удаленного соединения с помощью сокетов требует соответствующих системных вызовов

их принято представлять в виде сетевого стека (есть в книге Стивена)

действия на стороне сервера и клиента могут отличаться



пунктир - необязательный клиентский вызов

bind - связывает сокет с заданным адресом  
его необходимо вызывать на стороне сервера

для интернет сокетов этот адрес состоит из ip адреса и 16 битного номера порта  
клиенты могут не вызывать этот системный вызов, т.к. их точный адрес часто никакой роли не играет  
в этом случае адрес им назначается автоматически

`int bind(int sockfd, 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 (tcp)

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

connect - системный вызов устанавливает соединение, например в tcp по переданному адресу

для протоколов без установления соединения таких как ... может быть использована для установления адреса на значение для передаваемых пакетов

```
int connect(int socket, 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 пропустит запрос соединения

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

когда соединение принимается сокет копируется  
таким образом  
что исходный сокет остается в состоянии LISTEN  
а копия в состоянии CONNECT  
т.е. этим вызовом возвращается новый дескриптор файла для следующего сокета  
такое дублирование файлов в ходе принятия соединения  
дает серверу возможность продолжить принимать новые соединения  
без необходимости закрыть предыдущие соединения

типы сокетов:

1. d cbcntvt ggjllth;bdf.ncz gfhyt cjrtts  
job zdkz.ncz fyfkjuj пайпов с дуплексной связью  
для них используются специальные функции
2. сокеты unix
3. сокеты предназначенные для удаленного взаимодействия INET

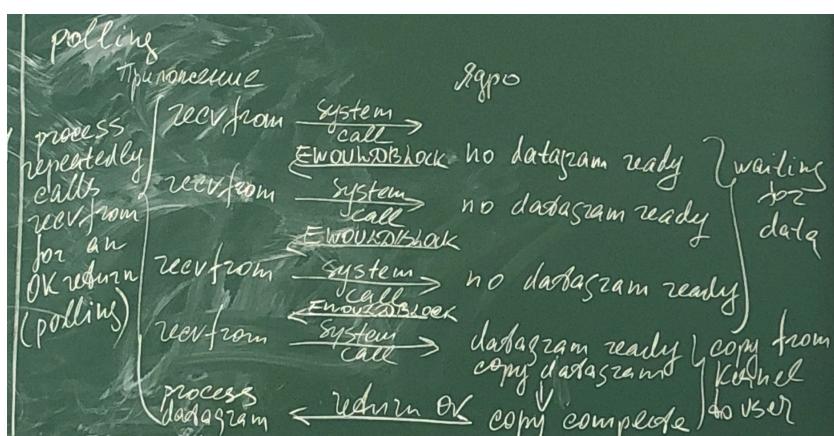
## 5 МОДЕЛЕЙ ВВОДА-ВЫВОДА (первоисточник: стивен) рассмотрим модели ввода/вывода с точки зрения программиста

### 1. блокирующий ввод/вывод Blocking I/O

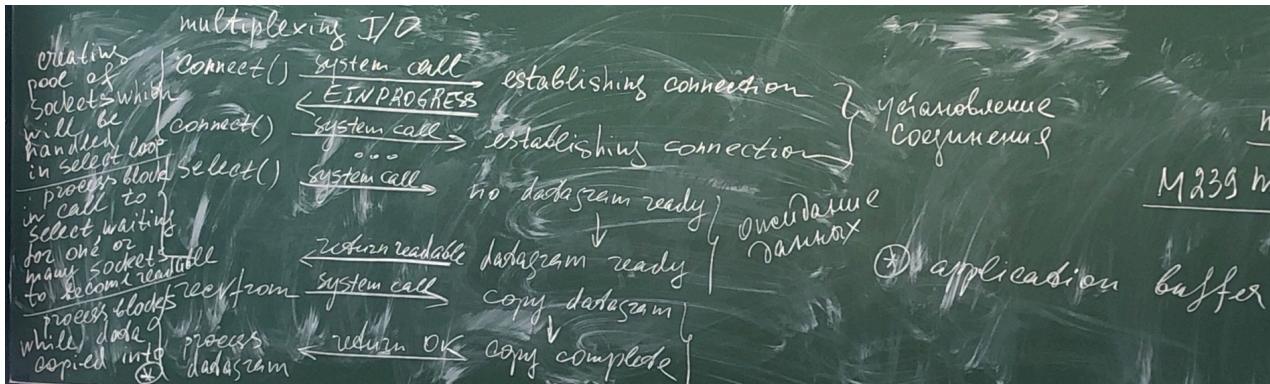


### 2. не блокирующий ввод/вывод Polling (опрос)

речь идет о готовности данных



### 3. мультиплексирование ввода/вывода



мультплексер - устройство которое bla bla bla определение из словаря

мультплексер (multiplexing)- процесс совмещения нескольких сообщений передаваемых одновременно в одной физической или логической среде

существует 2 вида мультиплексирования:

1. временное
2. частотное

когда выполняется системный вызов select процесс блокируется

время ожидания меньше

в цикле проверяются все сокеты и берется 1й готовый

пока обрабатывается 1й сокет

могут подоспеть другие

в результате снижается время простоя

если эту же проблему решать путем обычной блокировки

то случайным образом выбирается соединение из которого чтение будет выполняться первым

если данные не готовы то будем сидеть в этом ожидании

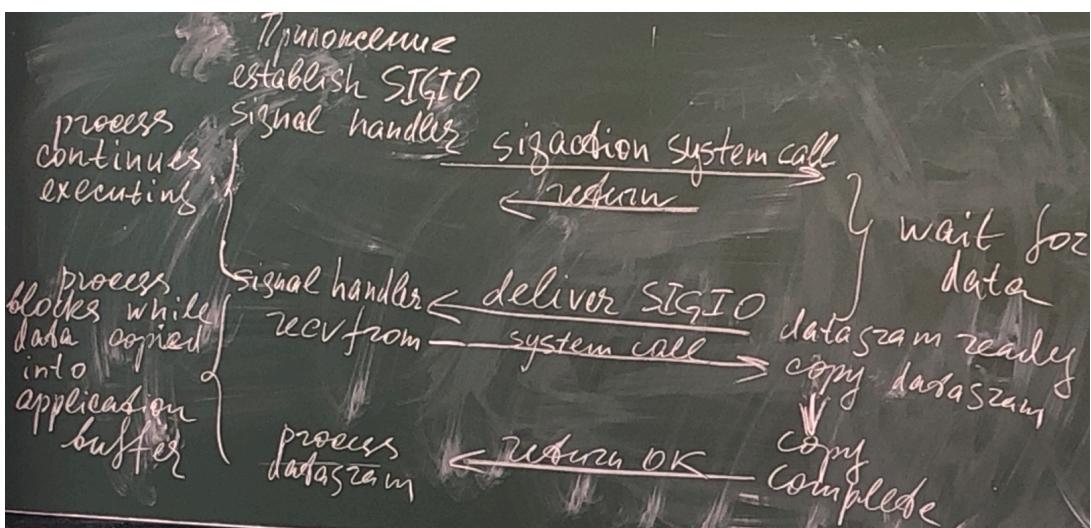
и т д

#### 3.5.

недостатки:

- 1) дорогой подход - требует больших накладных расходов
- 2) в питоне есть GIL (накладывает некоторые ограничения на поток), а именно нельзя использовать несколько процессоров одновременно => в каждом процессоре может выполняться только 2 поток

### 4. модель ввод/вывод управляемый сигналами



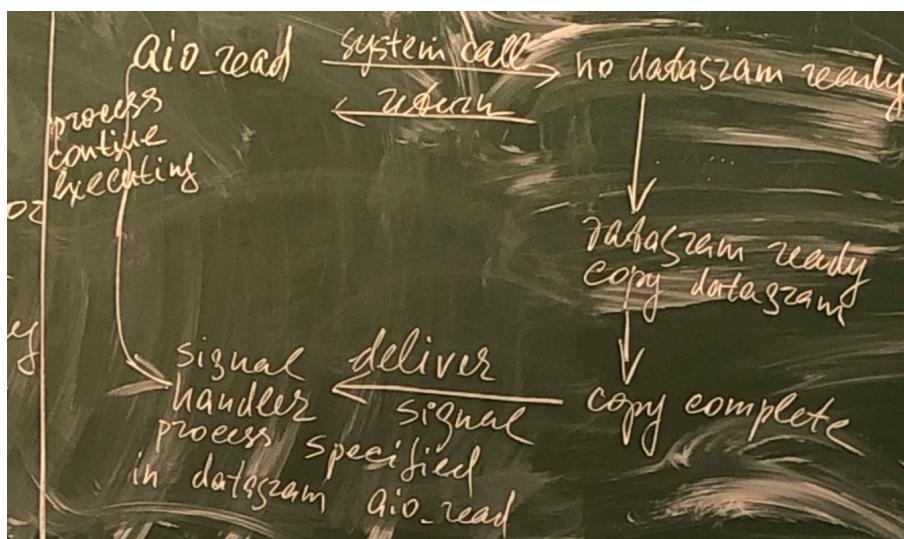
обработчик сигнала устанавливается системным вызовом dig\_option

всю работу берет на себя ядро  
оно отслеживает когда данные будут готовы  
после этого посыпает сигнал sig\_io  
который вызовет обработчик сигналов

сам системный вызов можно выполнить  
либо а обработки сигналов либо в основном потоке  
при этом сигнал для каждого процесса может быть только 1  
в результате за 1 раз можно работать только с 1м файловым дескриптором

на время обработчика сигнала  
данный сигнал блокируется

5. асинхронный ввод/вывод  
выполняется с помощью специальных системных вызовов



2 последние модели связаны с идеей асинхронного ввода вывода  
идея преследует возможность не блокировать процесс вплоть до получения данных

в чем выигрыш мультиплексирования?

возникновение соединения с несколькими клиентами будет время установления  
соединения будет меньше чем если устанавливать соединение с клиентами в некотором  
порядке

25.05.19

классификация моделей способов ввода/вывода:

|              | Blocking                          | Non-Blocking |
|--------------|-----------------------------------|--------------|
| Synchronous  | read/write                        | опрос        |
| Asynchronous | I/O multiplexing<br>(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

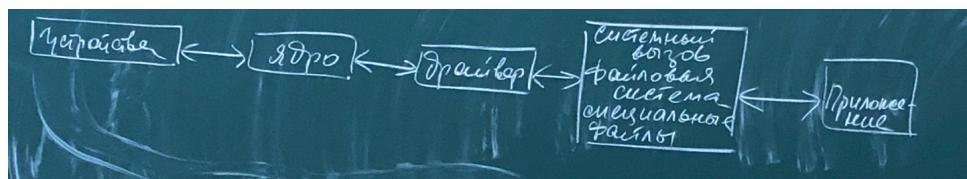
```
в struct inode
struct inode
{
    ...
    dev_t i_rdev; // это поле содержит фактический номер устройства
    struct list_head i_devices;
    union
    {
        struct pipe_inode_info *i_pipe;
        struct block_device *i_bdev;
        struct cdrom *i_cdev;
        char *i_link;
        unsigned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

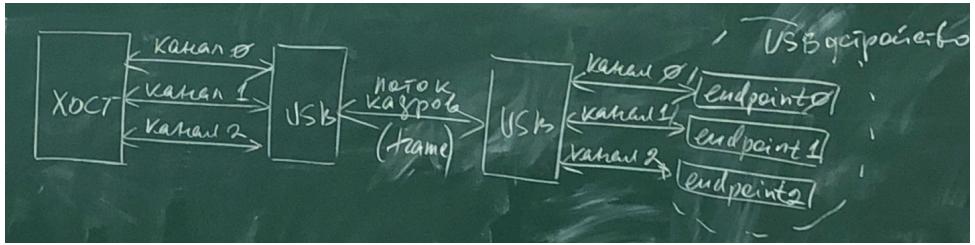
## USB ДРАЙВЕРЫ

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

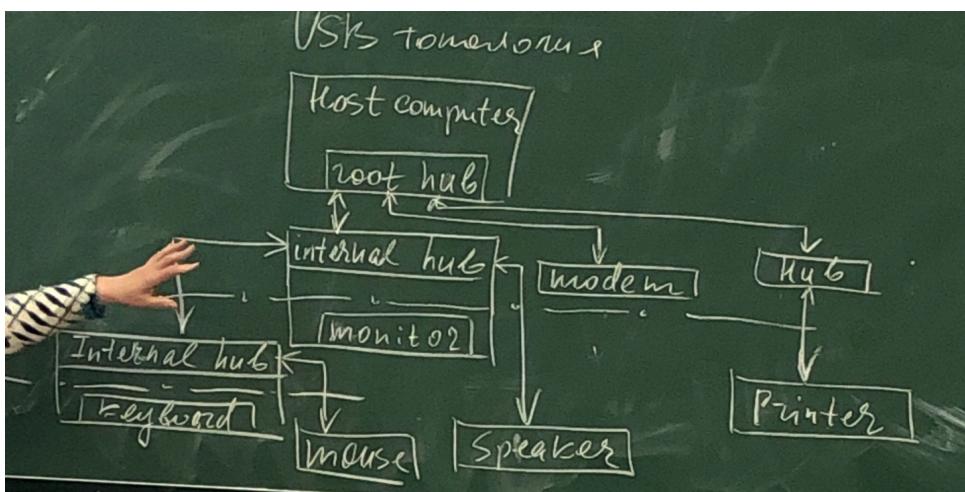


pipe это логическое соединение между хостом и конечной точкой устройства  
 при этом потоки данных имеют определенное направление in/out передачи данных  
 перед отправкой данные собираются в пакет

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

## 1.06.19

usb порты работают по принципу горячего подключения

hub (активное электронное устройство) - имеет одно соединение которое называется up stream port

многие устройства имеют встроенные hub

по usb шины состоит из 2х частей:

- верхняя часть (драйвер usb устройств) - используют функции usb для установления соединения с устройствами которые они контролируют
- нижняя (функции usb)

передача данных между host компьютером и устройством

она осуществляется с помощью конечных точек

она выполняется в форме пакетов

используемый тип передачи зависит от типа канала по которому выполняется передача  
тип канала определяется usb устройством подключаемым

на логическом уровне usb устройство поддерживает транзакции приема и передачи  
данных

т.е. прервать такую передачу нельзя

каждый пакет каждой транзакции содержит номер конечной точки (end point) на  
устройстве

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

совокупность каждой конечной точки и структур данных ядра называется каналом

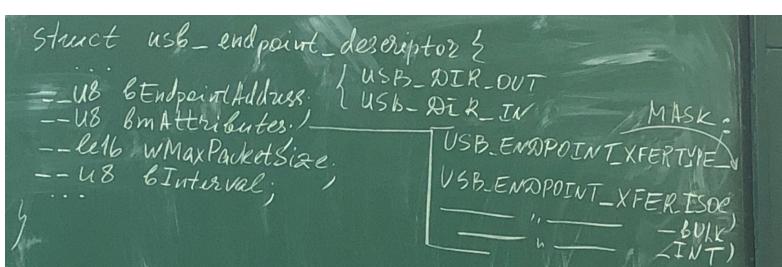
существует 4 режима передачи (типа каналов передачи)

| тип передачи | описание                                                                                                                                                                                                                                                                                                                                                                                                  |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| control      | определяет гарантированную доставку данных<br>канал является двунаправленным и<br>предназначен для обмена с устройством<br>короткими пакетами типа вопрос-ответ<br>используется системным по usb для выдачи<br>команд на usb устройство<br>позволяет ос прочитать информацию об<br>устройстве (коды производителя, модели)<br>PID/VID<br>передача типа контрол обычно осуществляется<br>конечной точкой 0 |

| тип передачи | описание                                                                                                                                                                                                                                                                                                                                                                                                   |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| isochronous  | имеет гарантированную пропускную способность<br>пропускная способность считается к количеству пакетов на 1 период шины (скорость доставки) т.к. скорость доставки определена => определена время задержки<br>передача осуществляется без подтверждения приема<br>используется для приложений реального времени таких как передача аудио и видео информации<br>этот канал осуществляет непрерывную передачу |
| interrupt    | речь идет о коротких (до 64 байт) упаковках срежиссированных ggtlfb в блоки по 8 байт на низкой скорости<br>например при вводе символов или координат используется такими устройствами как клавиатура и мышь<br>прерывания носят случайный характер и должны обслуживаться не медленно чем того требуют устройства (обслуживаться до появления следующего прерывания)                                      |
| bulk         | сплошная/поточная передача<br>используется устройствами отправляющими/принимающими большое кол-во данных<br>канал не имеет определенной пропускной способности и времени задержки<br>этот тип передачи имеет самый низкий приоритет<br>занимают всю свободную полосу пропускания шины                                                                                                                      |

end points (конечные точки) описываются структурой

```
struct usb_endpoint_descriptor
{
    ...
    __u8 bEndpointAddress; // адрес конечной точки // u8 - 8 бит // определяет
    // направление конечной точки // в это поле могут помещаться битовые маски USB_DIR_OUT
    // и USB_DIR_IN
    __u8 bmAttributes; // указывается битовая маска которая определяет какой
    // используется тип передачи // USB_ENDPOINT_XFERTYPE_MASK:
    // USB_ENDPOINT_XFER_ISOC, USB_ENDPOINT_XFER_BULK, USB_ENDPOINT_XFER_INT //
    // для типа interrupt это поле определяет время между ... конечной точки
    __le16 wMaxPacketSize;
    __u8 bInterval;
    ...
}
```



взаимодействие со всеми usb устройствами осуществляется с помощью urb

USB Request Block - блок запроса usb

блок описывается структурой struct urb мб найден в файле /include/linux/usb.h

используются для приема/передачи данных в/из конечной точки в асинхронном режиме

драйвер устройства может для 1й конечной точки или выделять несколько блоков или повторной выделять один и тот же блок

каждая конечная точка может обрабатывать очередь блоков

структура urb имеет следующий вид

```
struct urb
{
    ...
    struct usb_device *dev; // указатель на структуру в которую послан этот urb // структура которая описывает данное usb устройство // эта структура должна быть заранее проинициализирована драйвером
    ...
    unsigned int pipe; // информация конечной точки для указанного устройства для проинициализированной структуры usb_sevice
    ...
    int status;
    unsigned int transfer_flags;
    ...
}
```

urb создаются и уничтожаются

структуря struct urb не может быть создана статически в драйвере или в какой-то другой структуре

она должна быть создана строго динамически вызовом usb\_alloc\_urb();

когда завершается работа драйвера с urb необходимо проинформировать ядро об этом с помощью функции usb\_free\_urb()

в драйвере необходимо определить конечную точку прерывания

с помощью следующей функции

функция инициализация urb которой пос лается конечная точка прерывания

usb\_fill\_int\_urb();

отправка urb

после того как urb создается, проинициализирован usb драйвером  
должен быть отправлен в usb с помощью функции usb\_submit\_urb();

есть еще функция отключения/уничтожения и т д для urb

драйвер

есть просто usb драйверы а есть usb hid (USB Human Interface Device) драйверы  
это класс hid драйверов который присутствует в любой системе

эти драйвера управляют работой интерактивных устройств (мышь, клавиатура)

сам класс usb hid и его базовые функции пописаны в документации usb\_if

в любой операционной системе структура драйвера определена изначально

usb\_if означает USB implements forum

со временем скорость передачи информации по шине увеличивается в стандарте 3.1  
скорость 10ГБ в секунду

для работы с usb устройствами динамически генерируется вфс являющаяся частью  
файловой системы proc

это вфс находится в /proc/bus/usb

генерация этой файловой системы выполняется при загрузке системы

если открыть данный каталог то можно увидеть сколько в системе usb контроллеров

мы должны четко понимать какой драйвер мы пишем usb или usb hid

цитата из вахалии про драйверы

драйвер единственный модуль ядра который может взаимодействовать с устройством

драйверы не взаимодействуют друг с другом

преимущества:

1. участки кода который зависят от конкретных устройств можно размещать в отдельных модулях-драйверах
2. в систему легко добавлять новые устройства
3. производители могут создавать драйверы устройств без использования кода ядра
4. ядро «видит» все устройства одинаково (ядро взаимодействует с любыми устройствами одинаково) и может обращаться к ним посредством одного и того же устройства одинаково (речь про вызов драйверов)

в системе выделяется подсистема ввода/вывода которая обслуживает внешние устройства

если речь о usb драйверах (не hid) он определяется структурой struct usb\_driver

эта структура должна быть заполнена usb драйвером

в ее состав входит набор функций обратного вызова call back и параметрами usb core

```
struct usb_driver
{
    const char *name; // должно быть уникальным среди драйверов и должно быть
    таким же как имя модуля
    int (*probe)(struct usb_interface *intf, const struct usb_device_id *id);
    void (*disconnect)(struct usb_interface *intf);
    int (*unlocked_ioctl)(struct usb_interface *intf, unsigned int code, void *buf);
    int (*suspend)();
    int (*resume)
}
```

функция probe вызывается чтобы узнать готов ли драйвер управлять определенным интерфейсом устройства

в параметрах указано usb\_interface т.к. взаимодействие выполняется с интерфейсом  
если готов то prob возвращает 0 и использует usb\_set\_infdata чтобы связать данные  
характерные для драйвера с интерфейсом

вместо указанной функции (usb\_set\_infdata) может использована usb\_set\_interface для того  
чтобы указать нужный allsetting

если не готов то возвращается -ENODEV (с минусом)

если действительно возникли ошибки ввода/вывода то в errno возвращается соответствующее значение

\*disconnect вызывается если интерфейс больше не нужен (чтобы он стал недоступным) обычно интерфейс становится недоступным когда интерфейс отключается от системы или выгружается модуль драйвера

unlocked\_ioctl используются драйвера которые хотят взаимодействовать через файловую систему используя usbfs (не уверена что про это речь)

const struct usb\_device\_id \*id или \*id\_table

эта структура нужна для поддерживания горячего подключения hotplugging

таблица struct usb\_device\_id содержит список всех различных видов usb устройств которых драйвер может распознать

если данный параметр не определен то функция probe никогда не будет вызвана

любая структура определяется в системе какой-то таблицей

```
static struct usb_driver usb_miuse_driver = {
    .name = "usb_mouse_mouse",
    .probe = usb_mouse_probe,
    .disconnect = usb_mouse_disconnect,
    .id_table = usb_mouse_id_table,
};
```

```
#define USB_VENDOR_ID_LOGITECH 0x046d
#define USB_DEVICE_ID_LOGITECH 0xc52f
static const struct usb_device_id usb_mouse_id_table[] = {
    {USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HID,
                       USB_INTERFACE_SUBCLASS_BOOT,
                       USB_INTERFACE_PROTOCOL_MOUSE)},
}
```

```
MODULE_DEVICE_TABLE(usb, usb_mouse_id_table);
```

часто в это структуре (таблице) указывается сразу product и vendor как написано ниже

```
static struct usb_device_id pen_table[] =
{
    {USB_DEVICE(0x058f, 0x6387)},
}
```

```
MODULE_DEVICE_TABLE(usb, pen_table);
```

функции для регистрации драйвера

usb\_register\_dev(); используется для привязки драйвера к устройству

usb\_deregister\_dev(); функциям передается struct usb\_interface и usb class драйвер указатель (это предложение звучит как каша и кажется неправильным)

в курсами который она использовала используется input\_register\_device() и input\_

input\_register\_device() вызывается в call back функции prod  
input\_unregister\_device() вызывается в call back функции disconnect

module\_usb\_driver(usb\_mouse\_driver);

в init вызывается usb\_register() передается указатель на соответствующий драйвер

а exit вызывается usb\_deregister() передается указатель на соответствующий драйвер