ЛЕКЦИИ ОПЕРАЦИОННЫЕ СИСТЕМЫ
6-ой семестр

Файловая подсистема

Расматривали машину Медника Донавана. На самом верхнем уровне был уровень управления данными.
Упр данными — это и есть файловая подсистема. Важная задача хранить данные. Для хранения используются специальные внешние устройства . В наших машинах винчестеры, состоият из магнитных дисков.
Основная задача хранения — не просто записать , но записать таким образом чтобы потом к этим данным многократно обращаться.
ФПС обеспечивает возможность работы с данными, которые хранятся на внешних носителях.

Важнейшим понятием явл понятие файла.
Файл -любая поименованная совокупность данных. То есть содержание этих данных систему не волнует. Систему интересует возможность доступа к этим данным . Содеражние интересует только приложения или пользователя.

Файловая система (ФС) .
Файл — информация, хранимая во вторичной памяти или во вспомогательной ЗУ
с целью:
а) ее сохранение после завершения после конкретного задания
б) преодоления ограничения связанного с объемом основного ЗУ , рабочие файлы

Пространство жесткого диска делится на неравные части:
1) для хранения файлов (упр ФС)
2) область пейджинга или свопинга, предназначена для организации управления виртуальной памятью

Внешняя память (backing storage, auxiliry memory)
secondary
bulk memory – память большоого размера

ФС — порядок, который определяет способ организации и хранения , именования и доступа к данным на вторичных носителях информации

(коротко)
ФС используется для долговременного хранения данных и доступа к ним.

ФС явл частью ОС. ФС должна обеспечить создание чтение запись удаление переименование именование файлов и т. п.

Если файл хранит информацию , то ФС ***управляет*** процессом хранения или сохранения информации , а также обеспечивает последующий доступ.

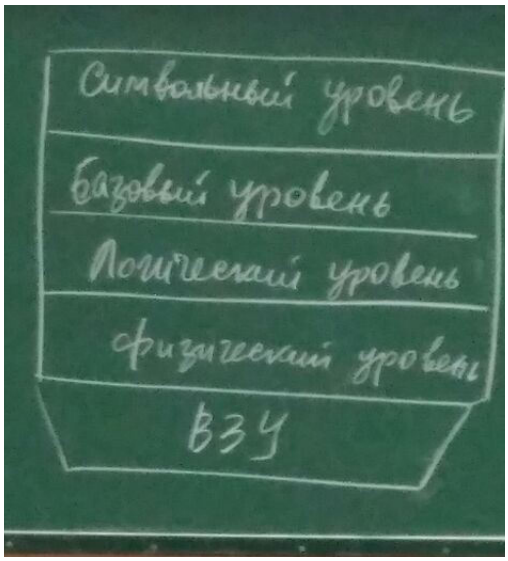
ФПС определяет:
а) формат и способ ее физического хранения
формат не всмысле приложения , а формат с точки зрения самой системы (логическая организация) и с точки зрения физического устройства

б) связывает физический носитель информации и API для доступа к файлам
Именно поэтому в иерархической машине файловая система распологается над уровнем управления устройством . Внешними устройствами поэтому управляет менеджер ввода \ вывода и специальные программы ДРАЙВЕРЫ.

Задача ФС

1. именование файлов с точки зрения пользователя (удобный способ обращения)
2. обеспечение программного интерфейса для работы с файлами пользователей и приложений
3. отображение логической модели или логического представления файлов на физическую организацию хранения данных на соответствующих носителях
4. обеспечение надежного хранения файлов, доступа к ним и защиты от несанкционированного доступа
5. обеспечение совместного пользования файлов

Иерархическая структура ФС
Вопросы структурирования ОС явл оч важными. Но при упр информации явл самыми важными.
В связи с перечисленными пунктами ФС как правило имеет следующую иерархическую структуру.



Символьный уровень — уровеень именования файлов (тот интерфейс котоырй предоставляется системой в распоряжение пользователя)
Разные ограничения. Они касаются как длины имени , регистра , расширения

Линукс более свободно относится к расширениям.
Симв ур предоставляет возможность систематизации файлов.
Класс названия : directory & subdirectory

Для более удобного обращения к файлам сущ **ссылки**. (уже знакомы) либо soft link либо hard link
В windows это иконки (тоже ссылки) .

Базовый уровень — это уровень формирования дескриптора файла . Файл должен быть описан в системе . Для этого система должна иметь соответственные структуры, которые позволяют хранить о файле всю необходимую для работы системы с этим файлом информации.

Логический уровень - лог адресное пространство файла аналогично лог адр пр-ву процесса. То есть начинается с нулевого адреса и представляет собой непрерывную последовательность адресов.

Два вида файла
Byte ориентированные (текстовые, доступ строго последовательный) и block ориентированные (позволяет обращаться к блокам напрямую)

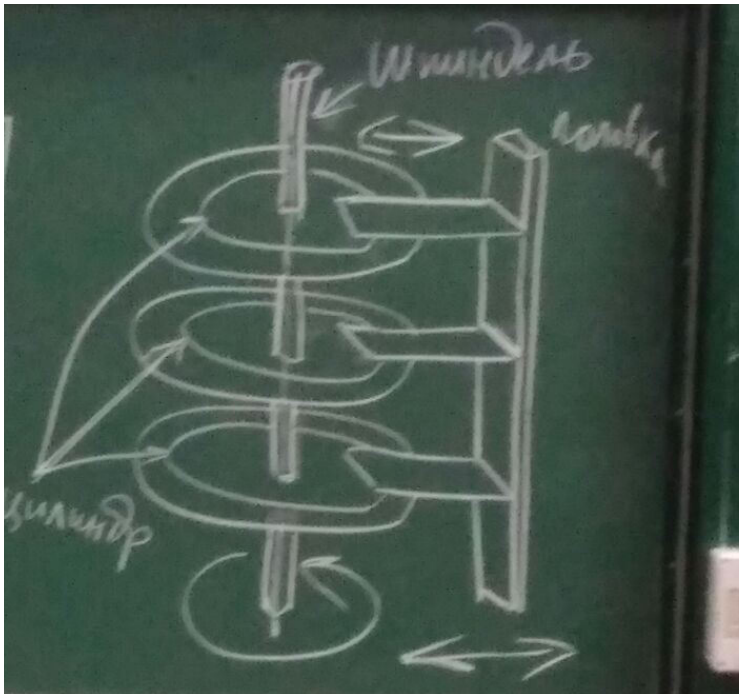
Символьные — определены символы конца строки

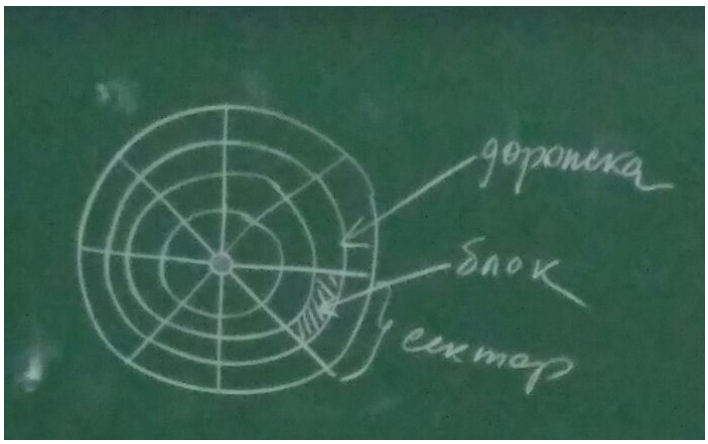
Есть возможность адресовать конкретный блок . Соотв функции fc

Логический уровень позволяет обеспечить доступ к данным, хранящимся в файле в формате, отличном от формата физического хранения . Обычно система не накладывает никаких ограничений на внутреннюю структуру данных, записанных в файл, и никак ее не интерпретирует . Структурой данных управляет пользователь, а не ФС. Могут сущ самые разные форматы данных , котоыре известны только программам, которые создают файлы и работают с ними. Это касается бинарных или типизированных файлов. Можем привести параллель между типами внешних файлов и типами внешних устройств. (Спец блочный и спец символьный файл) .

Физический уровень — задача обеспечить непосрдественный доступ к информации, которая хранится на внешнем носителе.

Устройство винчестера: цилиндр, шпиндель, головка





Один конкретный сектор на пластине по определенной дорожке наз **блоком**. Совокупность дорожек равноудаленных от центра пластины наз **цилиндр**. (связное распределение) Файл может храниться на диске в непрерывной последовательности адресов. Способ явл простым но не гибким. Такой способ имеет массу ограничений.
Файлы удаляются , на их место записываются новые. Таакой способ ведет к фрагментации диска. Когда файлу выделятся непр посл адресов, такй способ называется **связным распределением**.

В современных машинах **несвязное**, когда адр пространство в разброс . Возникают доп задачи адресации каждого такого выделнного файлу учатска диска, а также обеспечения доступа к информации, поскольку инфа хранится в разных блоках. Существуют алгоритмы, которые минимизируют время доступа.

Мы изучаем конкретную **файловую подсистему Линукс.**

В обеих системах сущ уровень абстракции , который получил название Virtual File System (VFS).

Unix	Linux
Vnode \ VFS	VFS (Virtual File System Switch)
	Ex2, ext3, VFS, UFS, NTFS, AFFS, HPFS, ISO9660, MS-DOS, FAT, FAT32 etc.

Такой общий интерфейс для любого типа ФС возможен только потому, что ядро само реализует слой абстракции над своим файловым интерфейсом в файловой системе. Это возможно, потому что VFS предоставляет общую файловую модель, которая способна отображать общие возможности и поведение любой мыслимой ФС .
Уровень абстракции работает на основе базовых интерфейсов и структур данных , которые поддерживаются конкретными ФС
Каждая ФС определяет особенности того, как открыть файл , как найти файл, как прочитать или записать информацию в файл и т. п.

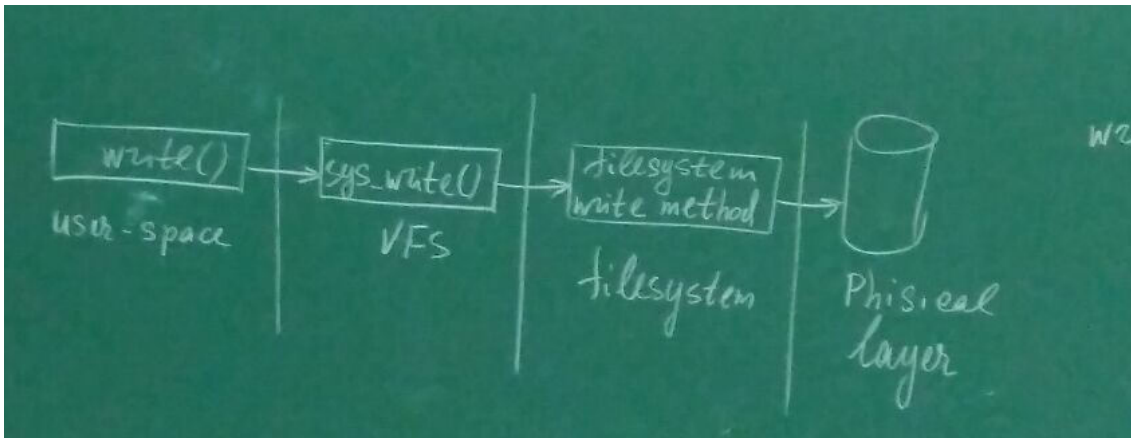
Физический код ФС скрывает детали реализации, однако все ФС поддерживают такие понятия как файлы каталоги. Все они поддерживают такие действия как создание файла, удаление файла, переименование, открытие, чтение запись, закрытие файла и т.п.

ФС запрограммированный этап, что их API и структуры данных рассматриваются как абстрактный интерфейс ожидаемый VFS.

Ядро легко работает с любой ФС ,а экспортируемый интерфейс пользователя успешно работает на любой ФС.
Физически ничто в ядре не должно понимать низкоуровневые детали ФС, кроме самих ФС. Рассмотрим например системный вызов write который выполняется приложением

write(f, &buf, len) – вызов записывает len байт из буфера в текущую позицию файла, который представлен файловым дескриптором f

В системе этот системный вызов будет обрабатываться последующей цепочкой.

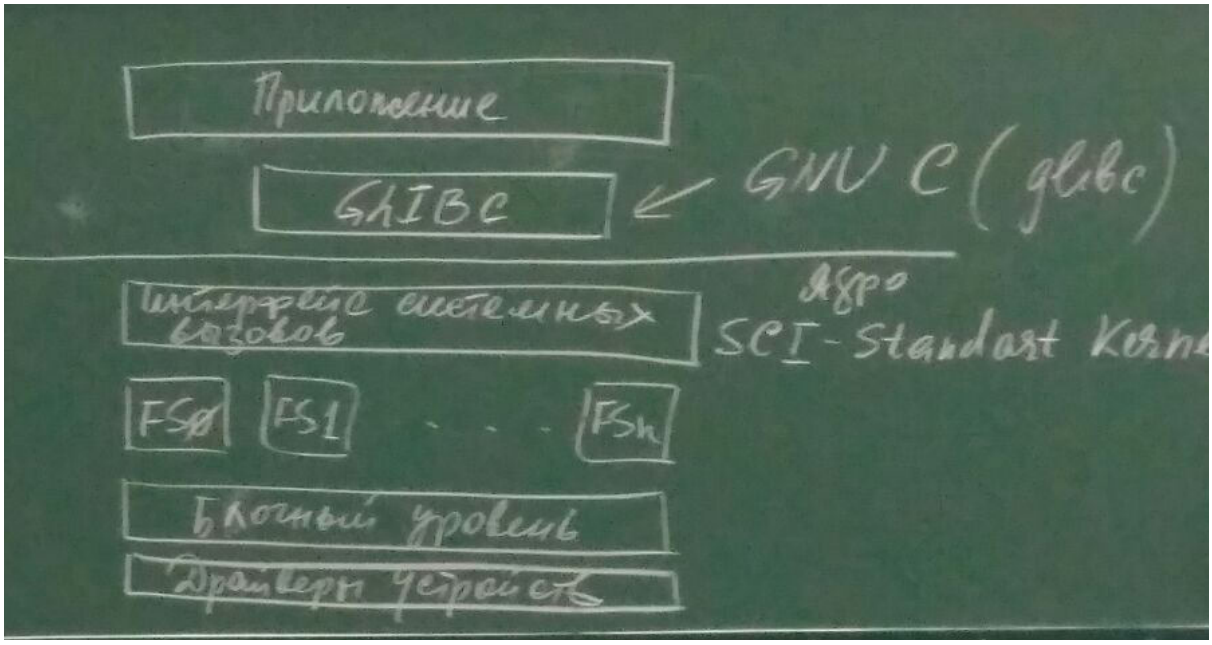


Вызов write

Как видно write обраб sys_wirte который опр фактичекйи способ записи файлов для сисетмы на которой находится файл описываемый дескриптором f
Затем общ сстемный вызов вызывает конкретный метод ФС для записи данных на физический носитель
в ФС Линукс Юникс все ФС представлены с помощью дерева каталогов и файлов. Это дерево может расширяться , т е могут появляться новые ветки путем монтирования новых ФС.
Любая ФС монтируется к каталогу, то есть к т**очке монтирования**.

Все операции с файлами на подмонтированных ФС осущ через интерсфейс VFS.

* SCI – Standart Kernel System – call Interface



GNU C (glibc) Рекурсивный акроним англ GNU’s Not Unix - GNU - идея открытого ПО

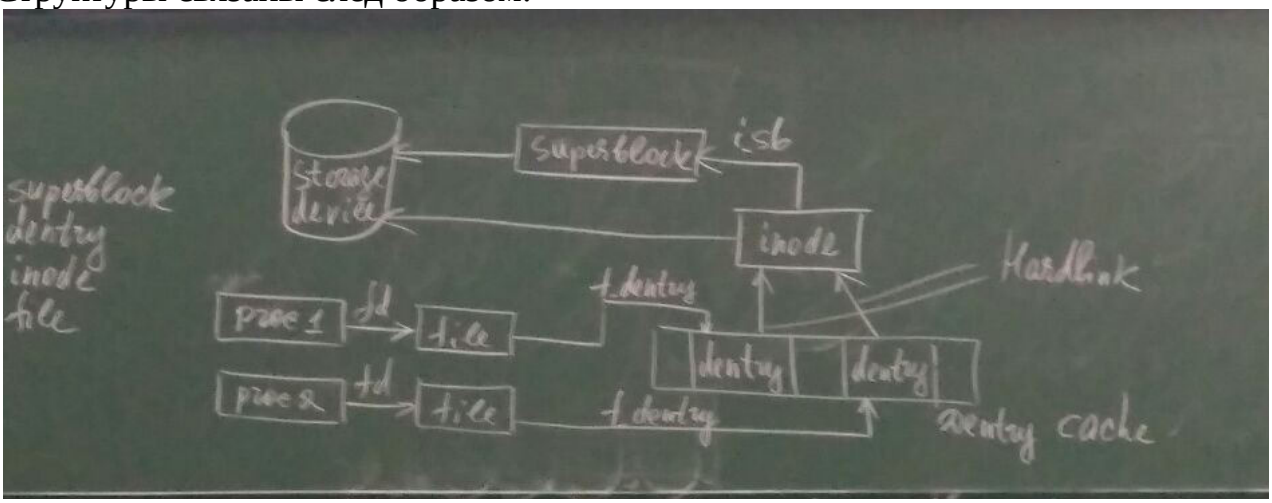
Линукс поддерживает FHS - стандарт определяет структуру каталогов и содержимое каталогов в Linux distributions . (август 93 разработка)
Стандарт иерархии ФС FSSTND был выпущен 14 февраля 94 года.
Убунту поддерживает стандарт ?
Указывается что в ФС обязательно должен быть корневая ФС которая обозначается « / » или корневой каталог , они должны составлять единую ФС расположенные на одном носителе или на диске или на дисковом разделе и т. п.

В нем должны распологаться все компоненты необходимые для старта системы.

Внутренняя организация VFS .

Базируется на 4 основных структурах:
1) superblock
2) dentry
3) inode
4) file

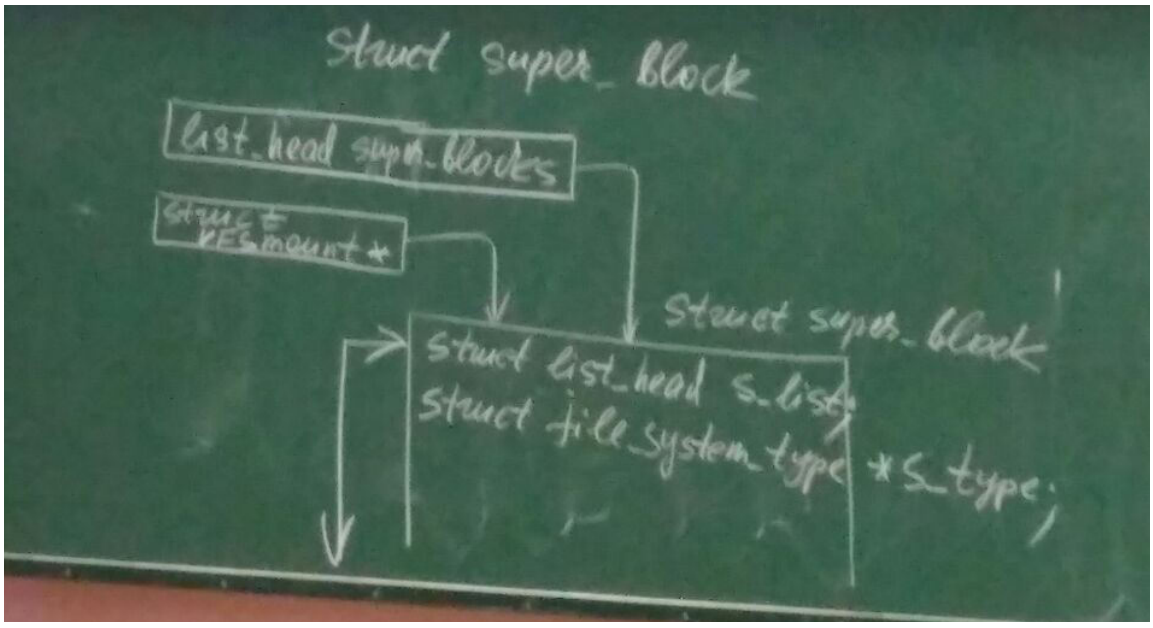
Структуры связаны след образом:



Суперблок описывает конкретную ФС .

Struct super_block
поле s_list организует двусвязный список всех смонтированных ФС

Каждая ФС имеет тип. Описывается структурой file_syste**m**_type. Может быть несколько ФС одного типа



Суперблок описывает конкеретную ФС.

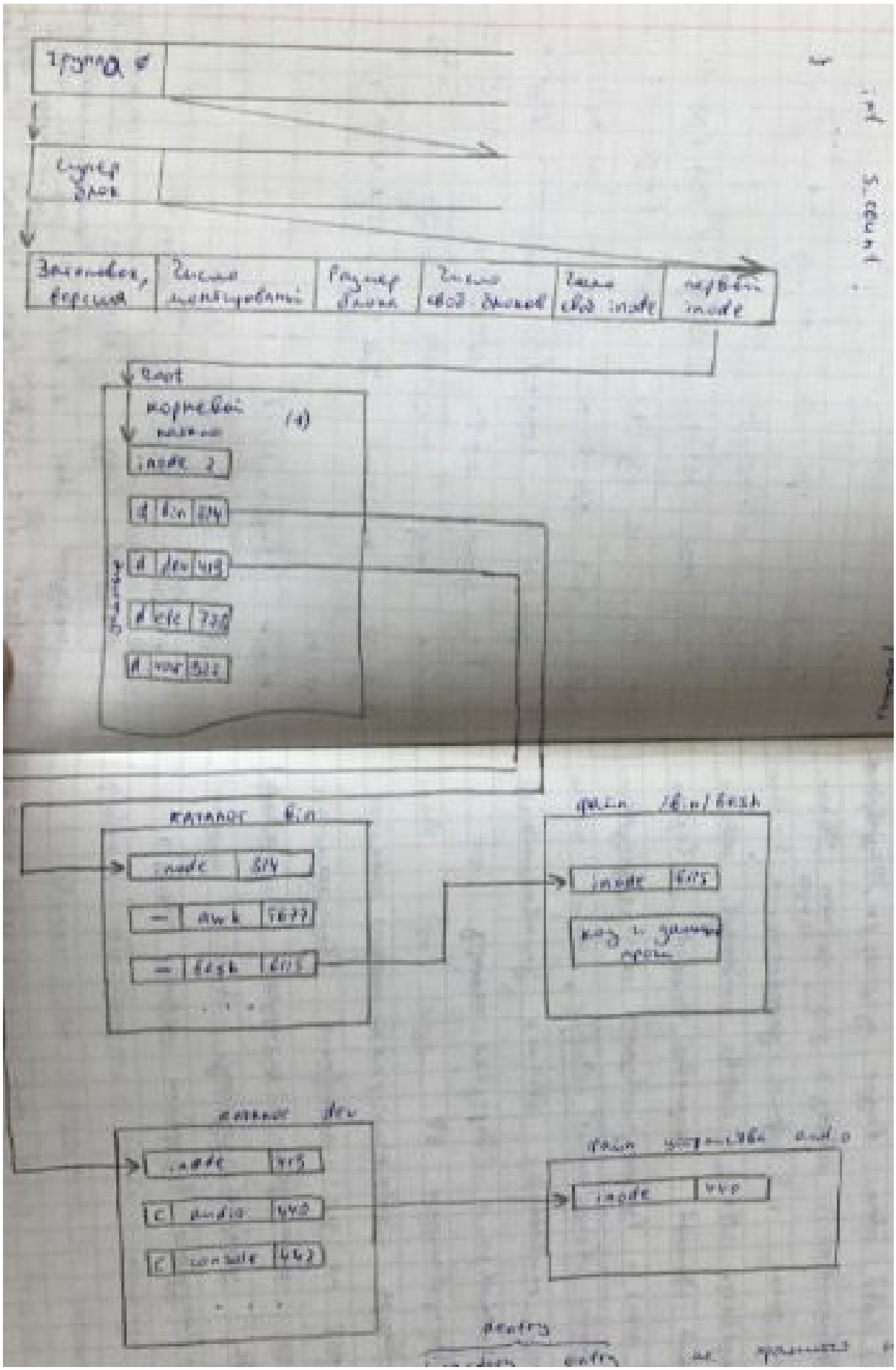
Лекция 27.02

Назначение super block
Содержит общую информацию о ФС
Это контейнер для метаданных высокого уровня, представляющий данные ФС.
ПО решающее задачи при непосредстенном взаимодействии с аппаратурой не использует суперблок.
Описывает информцию более высокого уровня, чем инф кот необходима для доступа к инф хранящейся на диске.

Суперблок хранится на диске.
Он связан с каждой конкртеной ФС. Для того чтобы создать свою ФС, подгрузить,на диске необходимо создать раздел, т е выделить объем вторичной памяти для хранения данных конкретной ФС. Суперблок описывает данный раздел.

Группа 0
\
суперблок
\
заголовок версии \ число монтированрий \ размер блока \ число свободных блоков \ число своб inode \ первый inode →
→ root – корневой каталог inode 2

Двусвязный список всех смонтированных ФС


Root
inode 2
inode
inode
inode

```c
struct super_block
{
    struct list_head s_list;
    kdev_t s_dev;
    unsigned long s_blocksize;//размер блока в байтах
    unsigned char s_dirt;//флаг изменения суперблока
    struct file_system_type *s_type;//описывает тип ФС(всегда один!)
    struct super_operations *s_op;
    //перечисляются операции, определенные на суперблоке
    struct dentry *s_root; //точка монтирования
    int s_count;//
}
```

struct super_block

Как видно из структуры суперблок обеспечивает основу ля работы ФС на диске и определяет управляющие параметры ФС , а именно количество блоков, количество свободных блков, корневой айнод .

Структура суперблок взаимодействует со структурой struct vfsmount*, которая описывает объект управления, называемый vfsmount., и также предоставляет инф о смонтированных ФС.

Суперблок располагается на диске, но для активных ФС эта структура копируется в память, то есть кешируется и чтобы ускорить доступ системы к информации.

<linux/fs.h>

каталог, к которому подмонтирована ФС , называется точкой монтирования

В системе каждый фаайл имеет свой inode.

Например, каталог /proc имеет inode = 1

каталог /root имеет inode = 2

При создании ФС создается таблица inode – ов. Inode bitmap (битовая карта айнодов), в которой хранится инф о том какие inode заняты, а какие свободны. В суперблоке хранится инф о том сколько в данной ФС имеется свободных айнодов. При создании ФС создается заведомо большее количество inode – ов, чем разработчик прогнозирует. При форматировании можно самостоятельно указать сколько айнодов зарезервировать .

Inode хранит в себе полную инф о файле, эту инф можно посмотреть с помощью команды **stat.**

Объект суперблок создается и инициализируется **alloc_super();** эта функция вызывается при монтировании ФС

```c
struct super_operations
{
    // создает и инициализирует новый inode связанный с данным суперблоком
    struct inode *(*alloc_inode)(struct super_block *sb);
    //уничтожает объект inode файла
    void (*destroy_inode)(struct inode *);
    //ф-ция вызывается подсистемой VFS когда в индекс (inode) вносятся изменения
    //журналируемые ФС используют эту ф-цию для обновления журнала
    void (*dirty_inode)(struct inode *, int flags);
    //записывает inode на диск и помечает его как грязный
    int (*write_inode)(struct inode *, struct writeback_control *wbc);
    //ф-ция дроп вызывается VFS когда удаляется последняя ссылка на индекс
    //и в результате VFS его удаляет
    int (*drop_inode)(struct inode *);
    //ф-ция put_super вызывает VFS при отмонтирования
    void (*put_super)(struct super_block*);
    //обновляет суперблок на диске
    void (*write_super)(struct super_block*);
}
```

Struct super_operations

ext2 пример
Struct super_block , VFSmount

```c
struct super_block
{
    struct list_head s_list;
    kdev_t s_dev;
    unsigned long s_blocksize;//размер блока в байтах
    unsigned char s_dirt;//флаг изменения суперблока
    struct file_system_type *s_type;//описывает тип ФС(всегда один!)
    struct super_operations *s_op;
    //перечисляются операции, определенные на суперблоке
    struct dentry *s_root; //точка монтирования
    int s_count;//
}

struct vsfmount
{
    struct dentry *mnt_root;
    struct super_block *mnt_sb;
    int mnt_flags;
};
```

```
static struct super_operations ext2_soops =
{
    ext2_read_inode;
    Null;
    ext2_write_inode;
    ext2_put_inode;
    ext2_write_super;
}

static super_operations encfs_sops =
{
    .destory_inode = encfs_destroy_inode;
}
```

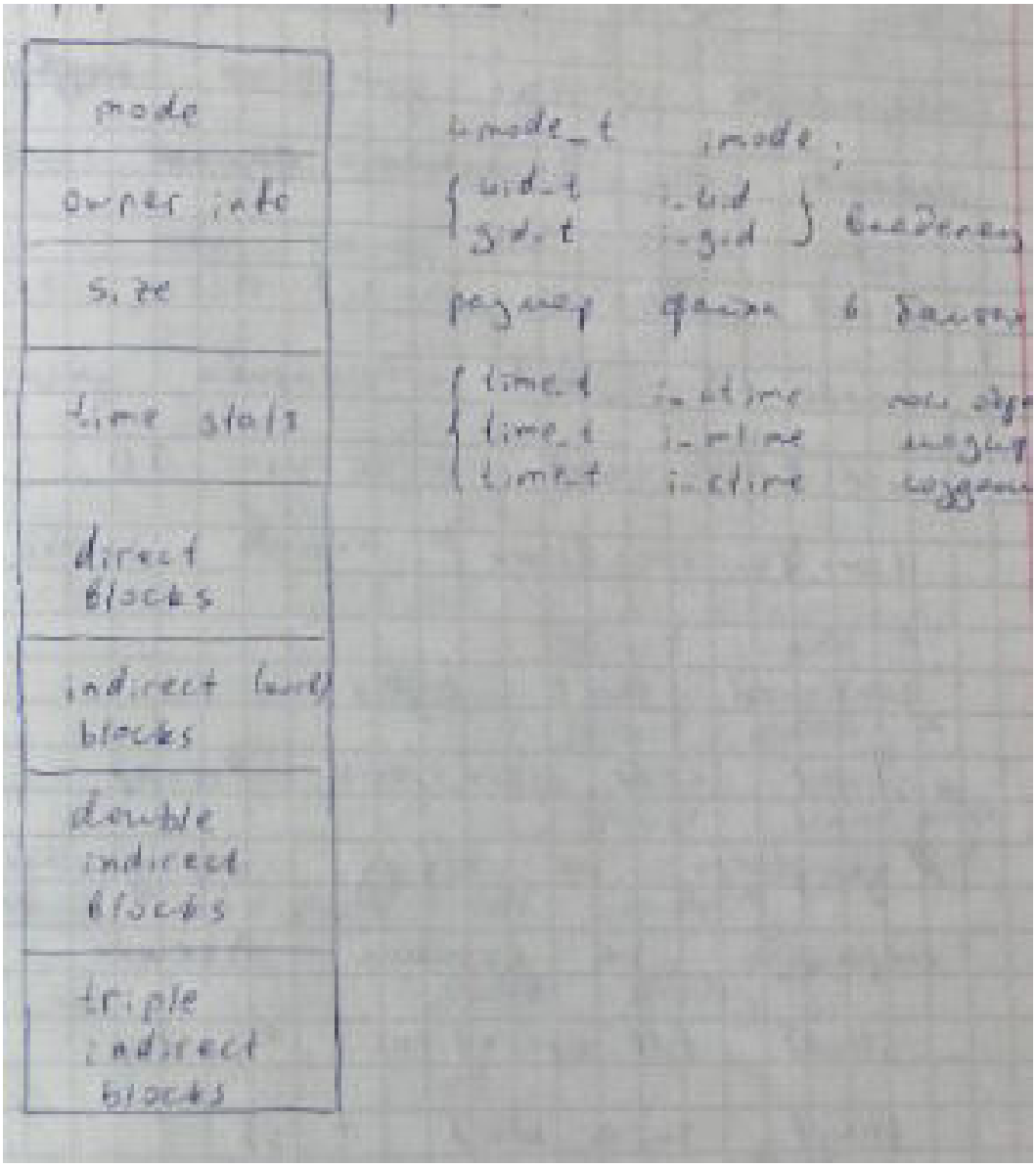
static Struct super_operations ext2_soops, encfs_sops
Когда мы создаем ФС из всего набора операций, который перечислен в sruct operations, мы можемм определить только нужные нам операции. (см выше)
Когда фс нуждается в выполнении операций над ее суперблоком, то говорят она следует за указателем на нужный метод объекта суперблока. Например если необходимо написать в суперблок то будет выполненма следующая строка:

sb → s_op → write_super(sb);
Inode – это структура, которая содержит полную информацию о физических файлах . ОС хранит инф о файле в inode. Инф о файле также как и инф о суперблоке иногда называется **метаданными**.
Другими словами **inode** – метаданные о данных.

Когда пользователь или процесс нуждается в доступе к файлам, ОС ищет точный и уникальный inode в таблице которая называется таблица inode -ов. Чтобы получить доступ к определенному файлу по его имени нужно найти inode данного файла, но для доступа к номеру inode имя файла не требует.

Inode number 3470036	
*	3470036
* *	3470017
Folder 1	3470031
File 1	3470043
File 2	3470023
Folder 2	3470024
File 3	3470065

Соответсвие имен и айнодов
* - текущая
* * - родительский каталог
Чтобы увидеть dot-ы надо ввести ls -a
Структура inode файла:



size - размер файа в байтах
time stams -
{
time_t i_atime \\ attach
time_t i_mtime\\ modify
time_t i_ctime \\ create}
direct blocks
indirect blocks
double indirect
triple indirect

для поддержки больших файлов ext* используется прямая адресация для 12 блоков непоседственно адреса, потом косвенная двойная и тройная.
Очевидно что чем больше косвенных ссылок тем медленнее доступ к нужной информации.

```
struct inode
{
    umode_t i_mode;
    uid_t i_uid;
    time_t i_atime;
    unsigned short i_bytes;
    //указатель на структуру перечисляющую операции на объекте (важна!)
    struct inode_operations *i_op;
    struct file_operations  *i_fop;
    struct super_block *i_sb;
}

struct inode_operations
{
    //связывает между собой соотв информацию
    int create(struct inode *, struct dentry*, struct nameidata *);
    struct dentry*(*lookup)(struct inode*, struct dentry *, struct nameidata);
    int mkdir(struct inode*, struct dentry*, int);
    int (*remove)(struct inode*, struct dentry*, struct inode*);
}
```

Struct inode, inode_operations, file_operations

```
struct file_operations
{
    struct module *owner;
    size_t (*read)(struct file*,  char_user *, loft_t *);
}
```

13.03

Функция линк вызывается из системного вызова линк для создания жесткой ссылки на файл из каталога который указан в качестве первой структуры

mkdir вызывается из системного вызова mkdir для создания нового каталога с указанными правми доступа

mknod вызывается из системнго вызова mknod для создания специального файла именнованного программного канала или сокета. Информация об этом специальном файле хранится в параметре dev_t. Файл создается в каталогге определенным struct_dentry с режимами доступами umode_t.

Struct inode ссылается на struct file_operations

```c
struct file_operations
{
    struct module *owner;
    loff_t (*llseek)(struct *file, loff_t int);
    ssize_t(*read)(struct file*, char _user*, size_t, loff_t *);
    ssize_t(*write)(struct file*, const char _user*, size_t, loff_t *);
    int(*open)(struct inode *, struct file*);
}
```

Первое поле struct module указатель owner
Struct file_operations перечисляет все фукнции, определенные на объекте файл.

Struct file и inode есть потому, что система различает ФАЙЛ и ПРОСТО ФАЙЛ, открытый процессом.

Функция read вызывается из системного вызова. Read(2) читает из файлового дескриптора . Из протоптипа read может прочитать из файла count байтов из fd в буфер buff. (читает не из fd, а из файла , описываемого десктриптором)

```c
#include <unistd.h>

sszie_t read(int fd, void *buff, size_t count);
```

системный вызов read

read начинается со смещения в файле file_offset и смещения файла увеличивается на число байтов, котрое было прочитано. Если смещение приходится на конец или за концом файла, то read вернет 0.

size_t и ssize_t - unsigned и signed integer. Сертифицированы в POSIX_1

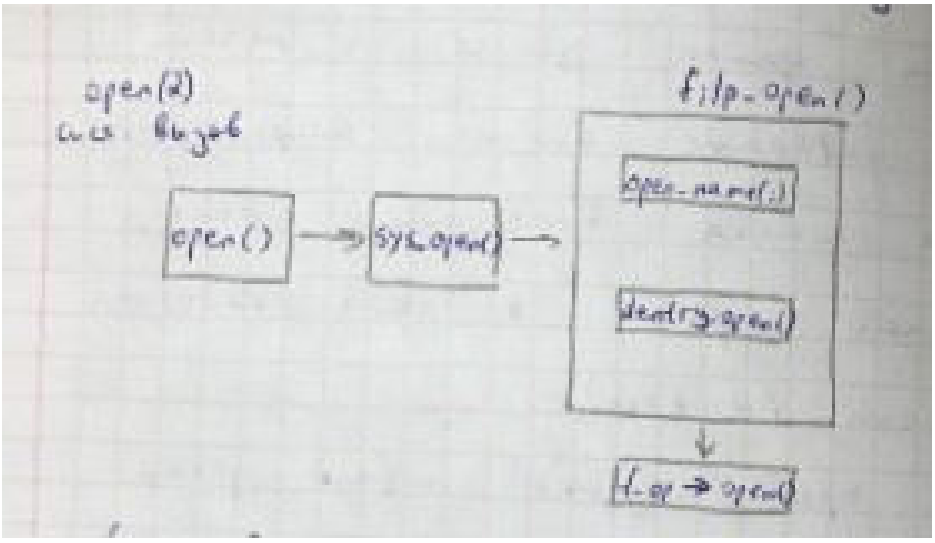
***_t** скрывает детали реализации. Но есть общее согласие на то чтобы не использовать typedef для структур? Ответ : стиль кодирования линукс избегает сипользоваение typedef , это нежесткое правило. И я применил это правило, так как в противном случае он выглядел номральньо но доольно часто используется явный экземлпяр структуры xxx_t . Мне не нравится абстрация типа если она не имеет реальный оснований. И экномия при наборе текста если ваша скорость ввода является основной проблемой когда вы кодируете.

Системный вызов **open** . Мануал open(2) - открывает и возможно создает файл.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fentl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Открывает файл, определенный pathname. Если указанный файл не сущ, флаги установлены как O_CREATE, файл может быть создан. Open возвращает файловый дескриптор — короткое и неотрицательное целое, которое используется в последующих системных вызовах, чтобы ссылаться на открытый файл. Файловый дескриптор возвращается из успешного системного вызова и будет наименьшим файловым дескриптоом еще не открытым процессом. Смещение файла устанавливается в начало файла. Аргумент pathname используется VFS для поиска вхождения dentry для данного файла в кеше каталогов dentry_cash (dcash). Вызов open создает новый дескриптор открытого файла (open_file_descriptor). Дескриптор открытого файла описывает смещение и установленные флаги.

Выводы:
для открытых файлов создается несколько таблиц. Одна таблица дескрипторов файлов, открытых процессом, и информация об открытом файле записывается в системную таблицу открытых файлов.



Sys_open, filp_open

Open реализован с помщью fs/open.c как sys_open.
Sys_open :

```
int sys_open(const char *name, int *flags, int mode)
{
    int fd = get_unused_fd();
    struct file *f = filp_open(name, flags, mode);
    fd_install(fd, f);
    return fd;
}
```

вызывается функция get_unused_fd пытается найти пустой слот в таблице дескрипторов открытых файлов процесса. Если процесс открыл слишком много файлов, то вызов будет fail .В случае успеха будет вызывана фукнция filp_open.

```
struct file *filp_open(const char *name, int flags, int mode)
{
    struct nameidate nd;
    open_namei(filename, namei_flags, mode, &nd);
    return dentry_open(nd.dentry, nd.mnt, flags);
}
```

Struct file *filp_open

Основную работу выполняет filp_open – выполняется в два этапа.
1) Первый — вызывает open_namei для генерации структуры nameidata . Эта структура выполняет связь линк к inode файла , то есть связывает с inode файла. Сначала поиск inode осуществляется в hash таблице под блокировкой inode_lock
2) На втором шаге вызывается функция dentry_open , которой передается информация из структуры nameidata. Фукнция dentry_open размещает новую struct_file и связывает ее с dentry и vfs_mount . Затем вызывает метод f_op → open (), который был установлен inode→i_op при чтеии inode фукнции open_namei. Фукнция detnry_open находит адрес фукнции на уровне ФС. Адрес фукнции на уровне ФС которая может выполнить операцию открытия и вызывается ее передавая inode и структуру файла. Адрес функции находится или в самом айноде или в одном из родительских айнодов.

Вернемся к filp_open. Функция находит директории dentry для конкретного файла. Она делает это спускаясь в соответствии с заданным путем. Начиная с корневой директории или начиная с рабочей директории процесса. В нашей лит-ре используется «текущая» вместо рабочей. Смотря откуда начинается pathname (со слеша или нет) . Спуск осущ до того момента, пока не будет найден файл или пока система не поймет что не получит нужный файл. Если каждый компонент пути уже был кеширован как dentry , то процесс разбора будет быстрым, так как все действия выполняются в памяти .
При этом если айнод прочитан, то увеличивается его счетчик ссылок icount . Если счетчик == 0 и айнод != грязный (информация не была изменена) , то удаляется из списка inode_unuse и вставляется в список inode_in_use. При этом счетчик показывающий неиспользуемые айноды уменьшается. Inode_in_used и inode_unused это кеши. Система для этого поддерживает несколько кешей.

1) Inode_cash в линукс представляет из себя глобальный хеш массив inode_hashtable, в котором каждый айнод хешируется по значению указателя на суперблок и 32-разрядному номеру айнод. Если файл не был объектом доступа

ранее, его айнод не был кеширован. В этом случае айнод будет прочитан с диска на уровне ФС и будет кеширован. Если отсутствует суперблок, то есть поле inode→i_sb == Null . Вместо хеш массива inode добавляется к двусвязному списку anon_hash_chain

К примеру таких анонимных айнодов могут служить сокеты созданные вызовом sock_alloc
2) Глобальный список inode_in_use
3) Глобальный сниок inode_unused, который содержит допустимые айноды с icount == 0.
4) Список для каждого суперблока sb→s_dirty содержит айноды с i_count > 0 и i_nlink > 0

i_state &I_DIRTY
когда айнод помечается как грязный\измененный он добавляется к списку sb→s_dirty (смотри struct super_block) при условии что он был кэширован. Поддержка такого списка уменьшает накладные расходы на взаимоисключения и синхронизацию.

5) inode_cache - является SLAB cache . Распределение памяти основано на введенном джефом бонвиком OS Sun
Смысл идеи: в ядре в значительном бъем памяти выделяется ограниченный набор объектов: дексирпторы файлов и тп . Идея базируется на том что количество времени необходимое для инициализации регулярного объекта в ядре превышает количество времени небходимого для его выделения и освобождения. Идея заключаетя в том что вместо того чтобы возвращать освободившуюся память в системе — оставлять эту память в инициализированном состоянии для возможного последующего использования в тех же целях . Например, если память выделена для мьютекса , то фукнцию init_mutex надо выполнить только один раз, когда первый раз выделяется память для объекта mutex. Последующие распределения памти под мьютекс уже не нужны так как память уже выделена в результате инициализации и последующего освобождения в связи с вызовом деструктора.
Вывод: все структуры взаимосвязаны методы вызываются в ответ на системные вызовы. В результаты в системы задействованы и inode , и dentry, и struct file.

DENTRY

VFS представляет каталоги как файлы, но VFS необходимо выполнять операции характерные для каталогов такие как поиск компнента пути по его имени , проверка существования пути, переход на следующий компонент пути. Для решения этих задач VFS сущ dentry.
Объект dentry _ это определенный компонент пути. При чем все объекты dentry – это компоненты пути включая обычные файлы. Элементы пути могут включать в себя точки монтирования. Объект dentry не соответсвует какой-либо

структуре данных на диске во вторичной ппамяти. Подсистема VFS создает эти объекты by_fly – налету в памяти по строковому представлению имени пути . Поскольку объекты элементов каталога не хранятся на физическом диске структура dentry не имеет флагов, которые указывают что объект был изменен и был записан на диск.

Struct dentry описывает не тлько каталоги , но и все то , что во вторичной памяти может быть описано с помощью inode.

```
struct dentry
{
    struct hlist_bl_node d_hash;
    struct dentry *d_parent;
    struct qstr d_name;
    struct inode *d_inode;
    const struct dentry_operations *d_op;
    struct super_block *d_sb;
}
```

Struct dentry

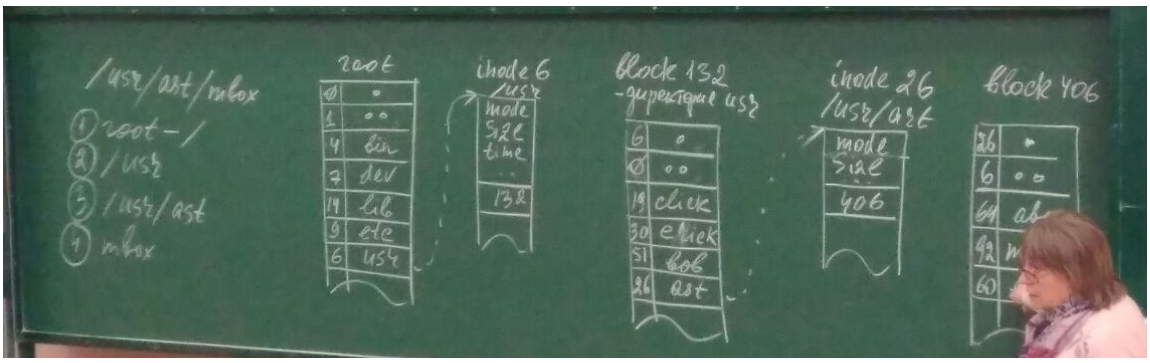
Лекция 27-03

Dentry создается на основае информации о файлах, которая хранятся в системе. Пример показывает доступ к mbox

/usr/ast/mbox
Имена не имеют значения. Важен процесс доступа к файлу.
Используются 4 объекта dentry:
1) root - /
2) /usr
3) /usr/ast
4) mbox

block 406 – содержимое поддиректории /ast

Обратить внимание: одна точка — текущая дир



struct file , file_operations
Одна из 4-ех структур, которые опр как основные структуры для работы с файлами и struct file – структура, которая предсатвляет открытые файлы

Любой открытый файл будет иметь в ядре объект файл

```
struct file
{
    struct dentry *f_dentry;
    struct vfsmount *f_vfsmount;
    struct file_operations *f_op;
    //права доступа
    mode_t f_mode;
    //текущая позиция файла
    loff_t f_pos;
}

//содержит фукнции, которые определены на структуре файл
//для работы с файлами в системе
struct file_operations
{
    struct module *owner;
    loff_t (*llseek)(struct file *, loff_t, int);
    ssize_t (*read)(struct file*, char*, size_t loff_t *);
    //ф-ции ассинхронного ввода\вывода
    ssize_t (*aio_read)(struct kiocb*, char *, size_t, loff_t);
    //...write, flash, open, redir , nmap and etc.
}

/*если мы не определеии функции, то
 gcc сам установит указатели на эти функции в null */
```

СОКЕТЫ

Сокет — специальный фай . Сокеты и 5 моделей ввода — на лекции, а ФС proc и задание — на семе (не сущ в системе, создается на диске; создана, чтобы пользователи могли получать инфу о файлах и их ресурсах).

Сокеты BSD

это абстракция конечной точки. (конечная точка связи или коммуникации). На транспортном уровне сокет описывается 3 параметрами: family, type и protocol (передаются системному вызову socket())

```
int socket(int family, int type, int protocol)
```

1) **family** определяет природу взаимодействия, включая формат адреса (address family == AF) - пространство имен
- AF_UNIX, PF_UNIX , PF_LOCAL, AF_LOCAL – опр сокеты для локальной связи процессов, то есть для связи процессов на одной машине

(см Unix 7 – раздел мануала)
- AF_INET – семейство протоколов TCP/IP (IPv4) — домен интернета
- AF_INET6 – семейство протоколов TCP/IP (Ipv6)
- AF_IPX – семейство протоколов IPX
- AF_UNSPEC - неопределенный домен

2) **type** – определяет тип сокета , который определяет характер взаимодействия
- SOCK_STREAM - потоковые сокеты определяют ориентированные на потоки надежное упорядоченное (!), полно-дуплексное, логическое соединение между двумя сокетами
- SOCK_DGRAM - опр ненадежную службу DGRAM без логического соединения, в котором пакеты могут передаваться без сохранения передачи. Это так называемая широковещательная передача данных
- SOCK_RAW – сырой сокет.
- NetLink – дейта -грамно-ориентированная система обмена сообщениями между ядром и usermode(режим пользователя, пользовательские приложения)

* Любой протокол, не устанавливающий предварительное соединение (а также обычно не контролирующий порядок приёмо-передачи и дублирование пакетов), называется **датаграммным протоколом**.

3) **protocol** – определяет протокол для опр типа сокетов
Если доступно несколько протоколов
Например, в семейство адресов возможным типом сокета type могут быть sock_stream и sock_dgram .
Для sock_stream всегда выбирают TCP
Для sock_dgram UDP

Протоколы обозначаются символами константами с префиксами IPPROTO_*
В качестве 3-его аргумента указывается 0 — говорит о том что протокол будет выбран по умолчанию.

Например,

```c
#include <sys/socket.h>
#include <sys/un.h>

unix_socket = socket(PF_UNIX, type, 0);
/*сокеты созданы системным вызовом socketpair -
 парные сокеты - явл безымянными
 здесь не указывается тип, так как их всего два
 при этом подчеркивается, что UNIX сокеты всегда надежны и не путают
 datagramm */
error = socketpair(PF_UNIX, type, 0, int *sv);
```

Адресация

Прежде чем передать сокет его надо связать с адресом в выбранном домене , то есть нужно идентифицировать процесс, с которым необходимо взаимодействовать.

Идентификация состоит из 2 частей:

1) ид сетевого узла с помощью сетевого адреса

2) ид конкретного процесса с помощью номера службы. Эта процедура обычно называется **именованием сокета**

Для явного связывания сокета с некоторым адресом используется функция **bind()**

Для каждого домена (family) используется свой формат представления адреса.

```c
struct sockaddr
{
    sa_family_t sa_family;
    char sa_data[14];
}
```

Вопрос адреса для сокетов до сих пор явл болезненным. Поскольку форматы сильно отличаются. Существует структура **struct sock_addr**

Иногда используются суффиксы к sock_addr: **_un и _in**

un – unix, in – internet

Порядок байтов

является характеристикой аппаратной платформы и определяет порядок следования байтов в длинных типах данных таких, как целые числа.

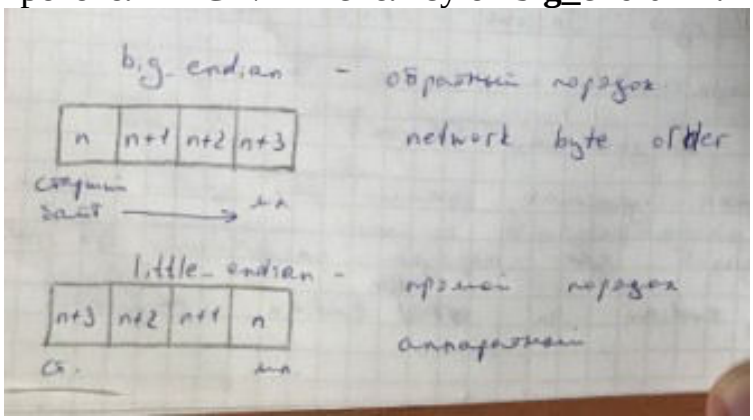
Существует в жизни два порядка следования байтов

big_endian - в старшем адресе располагается старший значащий байт

little_endian - в старшем адресе располагается младший значащий байт

Если на локальной машине, то задумываться о порядке следования байтов нет смысла. Но если взаимодействие идет через интернет, то здесь порядок следования байтов имеет значение.
Например,
процессор intel поддерживает **остроконечный** порядок следования
powerPC и что то еще – **тупоконечный**

Протоколы TCP / IP используют **big_endian** или **network_byte_order**



Преобразования осуществляются с помощью след функции:
htons()

```
#include <arpa/inet.h>
uint16_t - s
htons -> Host to NetWork Short
uint32_t - l
htonl -> Host to NetWork Long
```

Обратно к адресации
В зависимости от наначения программы вместо sock_addr используется sock_addr_суффикс
Например, для PF_INET (ipv4) следующая структура определяет:

```
struct sockaddr_in
{
    short int sin_family; // семейство адресов
    unsigned short int sin_post;
    struct in_addr sin_addr; //IP-адрес
    unsigned char sin_zero[8];
}
```

Лекция 10-04

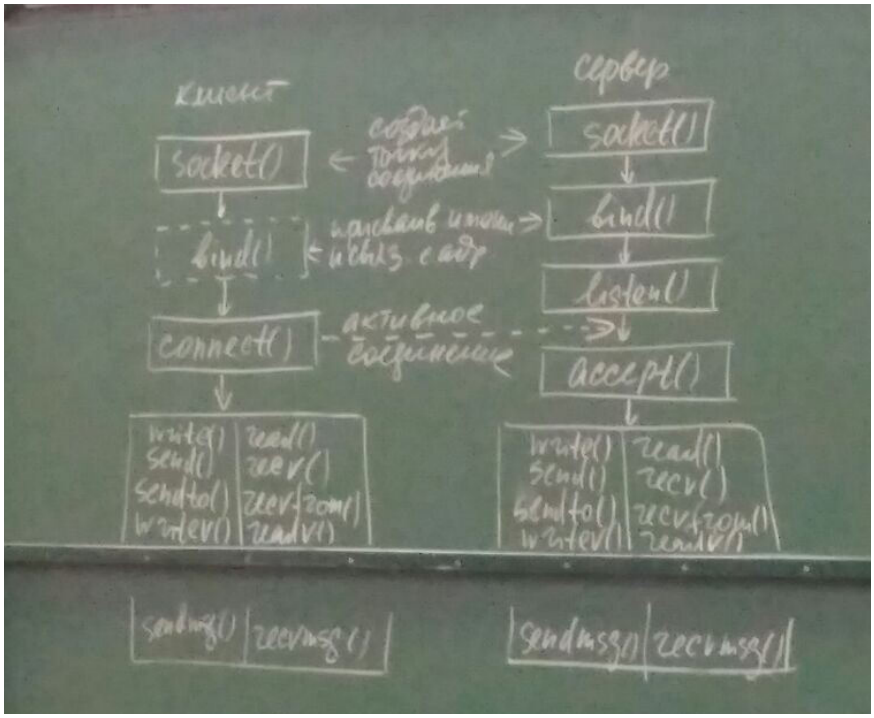
Сокеты определены в системе как файл, но исп обычные фукнции для работы с файлами — нельзя. Для этого есть спец системные вызовы. Сокеты — конечная точка назначчения.

На сокетах изначально была объявлена struct sock_addr. Но она не решает все, потому были объявлены еще. Причем объявлены для сетевых структуры содежат большее количество полей, то есть явл более длинными.

Например

```
struct sockaddr_in
{
    sa_family_t sin_family;
    unsigned short int sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero [sizeof(sockaddr_sizeof(sa_family_t)_
        sizeof(uint16_t)_sizeof(struct in_addr))];
}
```

Рассмотрим системные вызовы в сетевом стеке.
Взаимодейтсие сокетов происходит по модели клиент — сервер. ТО есть один по адресации сервера, другой — адресации клиента.



И в конце там у каждого close().

Системному вызову сокет надо передать 3 параметра: домен тип и протокол. Затем сокет связывает с определенным адресом , причем связывать сокет с адресом необходимо на стороне сервера . Для интенет сокетов этот адрес состоит из ip адреса сетвеого интерфейса локальной системы и номера порта .

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *addr, int addrlen);
int listen(int sockfd, int backlog);
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen )
```

Клиенты могу не вызывать bind , так как их номера не играют роли в большинстве случаев. Если клиент не вызвал bind (), адрес назначается им автоматически. Как видим в системном вызвое bind 1-ым явл дескриптор, а 2-ым структура sockaddr .
Затем на стороне сервера вызывается listen, который используется сервером для информирования системы о том что сокет должен принять соединение. Это имеет смысл только для протоколов , ориентированных на соединения. В наст время только для TCP.
На стороне клиента вызывается connect() , который инциализирует активное соединение (TCP) по указанному в качестве параметра адресу. Для протокола без соединиения UDP системный вызов connect может использоваться для указания адреса назначения всех передаваемых впоследствии пакетов.

Accept используется сервером при соединении, при условии что сервер ранее получил запрос с соединения, иначе запрос будет заблокирован до тех пор пока не поступит запрос соединения. Когда соединение принимается сокет копируется(СОЗДАЕТСЯ КОПИЯ), а именно

а) исходный сокет остается в состоянии listen , а копия в состоянии connect
б) вызовом accept возвращ новый дескриптор файла для следующего сокета .
Такое дублирование сокетов в ходе принятия соединения дает серверу возможность продолжить принимать новые соединения без необходимости предварительно закрывать предыдущие соединения.

Сокет это фукнция которую мы вызываем в приложении. Ядро линукс предоставляет для работы с сокетами один единственный системный вызов. Который включает все перечисленные функции.

```
//////////////////////
#include <net/socket.c>
asinlinkage long sys_socketcall(int call,  unsigned long *args)
{
    int err;
    if(copy_from_user(a, args, nargs[call]))
        return DEFAULT;
    a0 = a[0];
    a1 = a[1];
    switch(call)
    {
        case SYS_SOCKET: err = sys_socket(a0, a1, a[2]); break;
        case SYS_BIND:   err = sys_bind(a0, (struct sockaddr *)a1, [2]); break;
        case SYS_CONNECT:err = connect(a0, (struct sockaddr *)a1, a[2]); break;
        //...
    }
    return err;
}
```

Asinlinkage long sys_socketcall

Функция реализована как переключатель системных вызовов .

Int call – определяет номер нужной фукнции. В файле <include/linux/net.h> определены соответствующие номера.
В ядре 2.6.28 их 18.
#define SYS_SOCKET 1
#define SYS_BIND 2
#define SYS_CONNECT 3
//...
#defin SYS_RECV 10

Мы видим что если фукнции передан парам == 1 будет осуществлен вызов sys-socket, если == 3, то будет вызван sys_connect. Если приложение вызывает функцию socket() , то произойдет вызов sys_socket. Фукнция sys_socket определена в файле <sys/socket.c >.

```
<net/socket.c>
asmlinkage long sys_socket(int family, int type, int protocol)
{
    int retval;
    struct socket *sock;
    //...
    retval = sock_create(family, type, protocol &sock);

    return retval;
}
```

Asinlinkage long sys_socket

Функция инициализирует сокет. Значение retval и есть тот самый дескриптор который возращ фукнция socket.

Дизайн сокетов беркли следует парадигме юникс. В идеале отобразить все объекты которым осуществл доступ для чтения или для записи на файлы , чтобы с ними можно было работать с обычными операиями чтения и записи из файла . Другими словами объектами которыми манипулируют при операциях чтения и записи в контексте транспортных протоколов являются конечные точки коммуникационных отношений , а именно сокеты.
Рассмотрим структуру socket

```
struct socket
{
    socket_state state;
    short type;
    unsigned long flags;
    const struct proto_ops *ops;
    struct fasync_struct *fasync_list;
    struct file *file;
    wait_queue _head_t wait;
}
```

struct socket

Есть поле fasync_list – список асинхронного запуска. Также видим wait_queue_head_t – очередь .

На сокете определено 5 **состояний**.
SS_FREE – НЕ ЗАНЯТ
SS_UNCONNECTED – НЕ СОЕДИНЕН
SS_CONNECTING – СОЕДИНЯЕТСЯ В ДАННЫЙ МОМЕНТ
SS_CONNECTED – СОЕДИНЁН
SS_DISCONNECTING – РАЗЪЕДИНЯЕТСЯ В ДАННЫЙ МОМЕНТ

Здесь нет DISCONNECTED – нет смысла, так как он становится FREE.

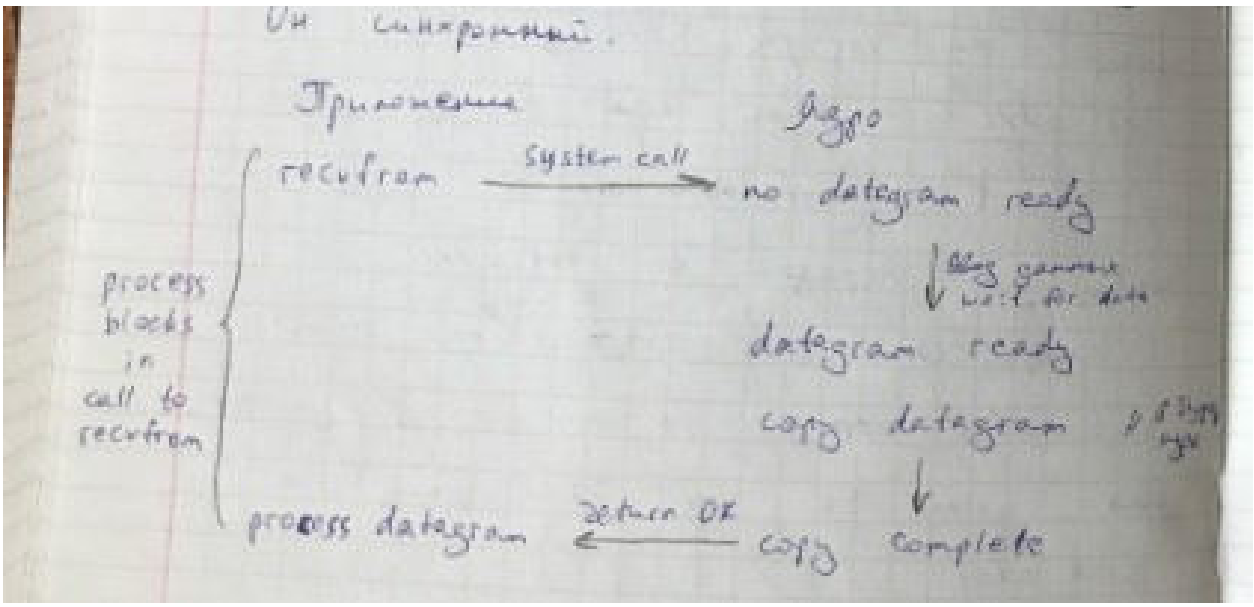
Поле flags используется для синхронизации доступа. Указатель ops ссылаетя на действия подключенного протокола , например TCP или UDP. Сокет в линукс является специальным файлом , поэтому struct socket содержит поле struct file *file. В свою очередь структура file ссылается на inode . Поле type структуры socket служит для хранения второго параметра фукнции socket. Допустимые значения этого параметра определены в include/asm/socket.h

Определены следующие значения

.SOCK_STREAM
.SOCK_DSRAM
.SOCK_RAW
.SOCK_RDM
.SOCK_SEQPACKET
.SOCK_PACKET(не следует использовать)

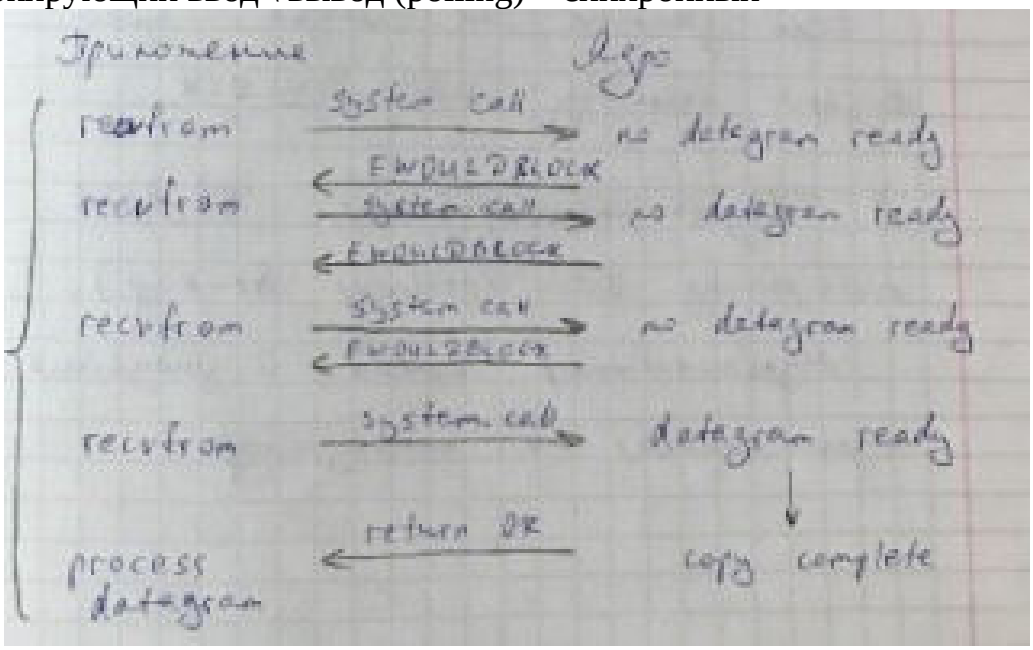
Была определена общая структура sock_addr . Интересно что bind и connect , которые запрашивают указать sock_addr обращаются к данным из указателя если он указывает на структуру большую чем ожидалось . Несомненно, что передается размер стуктуры , но каков фактический синтаксис используемой этими функциями для получения ip – адреса , если указатель указывает на структуру большую чем sock_addr . Ответ : фукнции которые ожидают указател на struct sock_addr , обращаются по указателю, который им указан . Например если им отправляется укзаатель на struct sock_addr storage , то они получают доступ как если бы это был указатель на sock_addr.

Пять моделей ввода \ вывода:
Можно сказать , что эти модели рассматриваются с точки зрения программистов. Аппаратура работает всегда однаково но именно ПО формирует различные способы работы с аппаратурой. Базовым источником явл классик Стивен. Стивен указывает что Unix и им подобным доступно 5+1 модель ввода \ вывода.
1) блокирующий ввод\вывод (blocking IO) - синхронный
Все эти модели рассматриваются с использованием диаграмм



Приложение выполняет запрос на ввод \ вывод. Данные не готовы . Здесь не говорится о распараллеливании фукнций. Упр вводом берет на себя контроллер устройства , ввод данных очевидно занимает некоторое время и это обозначается на данной диаграмме стрелочкой вниз обозначается и это все определяется на ожидание данных (wait_for_data) . Дальше аппаратное прерывание. Дальше вызывается обработчик прерывания. В его задаче стоит сохранение данных в буфер ядра. Целью явл передача данных от устройства к приложеию. Когда данные скопированы в буфер приложения, оно может продолжить свое выполнение. Все это вермя процесс блокирован на системном вызове receive_from() . Стивен использовал здесь систм вызов для того чтобы показать что это самый общий способ ввода \ вывод (передачи сообщений).

2) неблокирующий ввод \ вывод (polling) – синхронный



Процесс повторяет системный вызов recvfrom() ожидая получения сообщения о том, что данные получены , то есть copy_complete. Такой режим возможен и с обычным внешним устройством, но это неэффективно так как процессоор занят непроизводительной работой — опросом. Прерывания введены чтобы освободить процессор от опроса. Но при работе с сокетами может быть установлен неблокирующий режим, то есть recvfrom() будет вызываться в цикле до тех пор пока ядро не вернет данные для копирования. Мы видим что если данные не готовы — ядро возвращает ошибку EWOULDBLOCK.

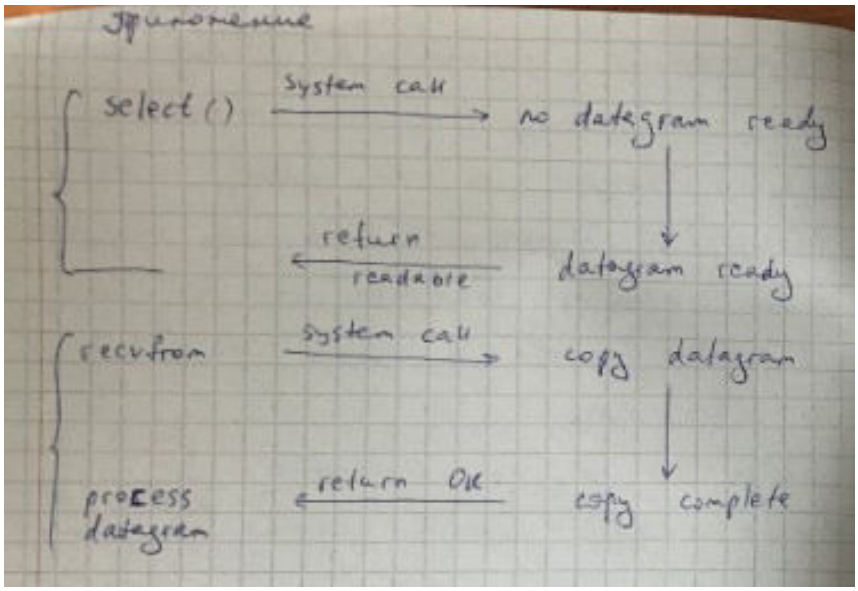
3) мультиплексирование ввода\вывод (multiplexing) – асинхронный блокирующий
Мультиплексор имеет два значения : мультиплексор и коммутатор

Лекция 24-04

Мультиплексер - это устройство, которое объединяет информацию по нескольким каналам ввода и выдает ее только по одному каналу.
Мультиплексирование или объединение - это процесс совмещения нескольких сообщений , передаваемых одновременно в одну физическую или логическую среду.

Виды мультиплексироания:
1) временное time division multiplexing - устройству отводятся интервалы времени , в которые оно может использовать передающую среду
2) частотный

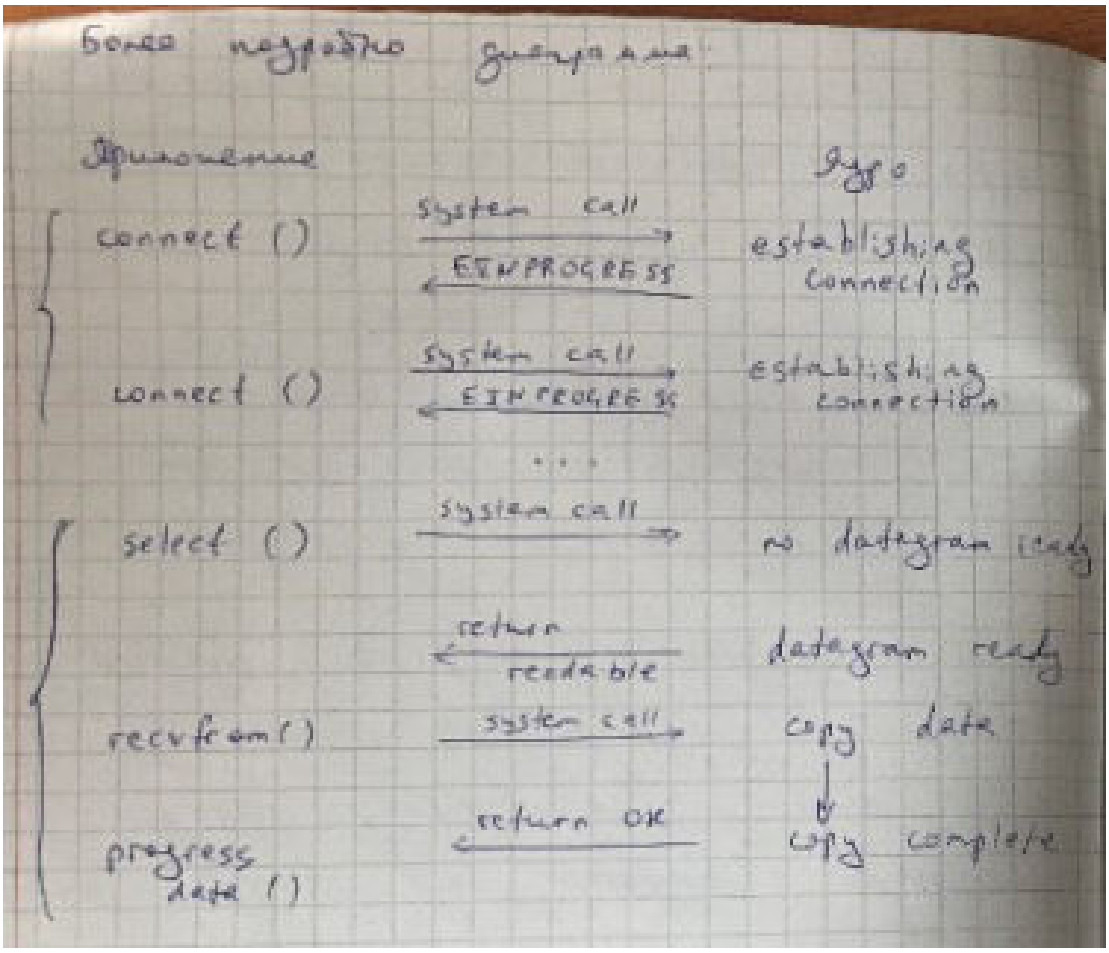
Мультиплексированный IO



для реализации испольуется один из мультиплексеров:
select, poll, pselect, epoll
В результате вызова select процесс блокируется, ожидая готовности одного из многих сокетов. Готовность подразумевается, что на каком-то сокете поступили данные. А мультиплексирование предполагает, что опрашивается много сокетов.

Системный вызов select как раз и выполняет эти действия и сообщает что один из сокетов готов для чтения

Затем после того как поступила информация о готовности сокета дял чтения выполнятеся recvfrom - процесс блокируется на время копирования данных из ядра в адресное пространство процесса . И по данной диаграме возвращение сообщения о том что данные готовы (скопированы) процесс переходит к обработке .
Преимущество мультиплексирования перед блокируемым вводом выводом заключается в том что обрабатыывается не один а сразу много дескрипторов . Процесс блокирован, но время блокировки будет меньше , так как вероятность того что данные посступят на любой из набора дескрипторов выше чем вероятность того что данные поступят на конкртеный дескриптор или на конкретный сокет.



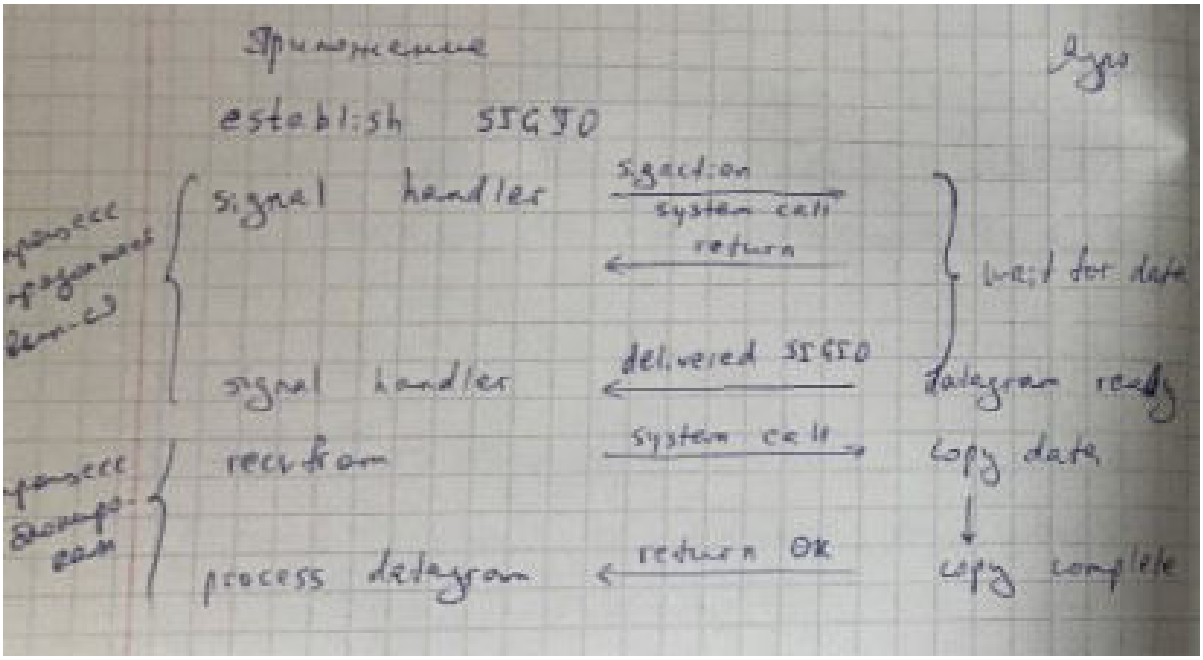
Connect создает пул сокетов, который будет обрабатываться в цикле select . .Select блокирует процесс в ожидании готовности хотя бы одного сокета . После того как возвращено сообщение что данные готвоы для чтения выполняется recvfrom() - процесс блоикруется до момента пока данные будут скопированы в буфер приложения.

При мультиплексировании проверяются все сокеты и берется первый готовый. Пока обрабатываестя первый сокет могут подоспеть и остальные . В результате сокращается время блокировки .

3) Похожий на мультиплексирование способ — обозначим ккак AAA (3_A). В этом способе запускается несколько потоков . В каждом из которых выполняется блокирующий ввод вывод. Недостатки — дорогие потоки в линукс . Дорогие то есть много требует (большое количество накладных расходов).

Если взять python, то в нем есть **GIL** (global interpreter lock) , то есть в кжадом процессе может выполняться только один поток.

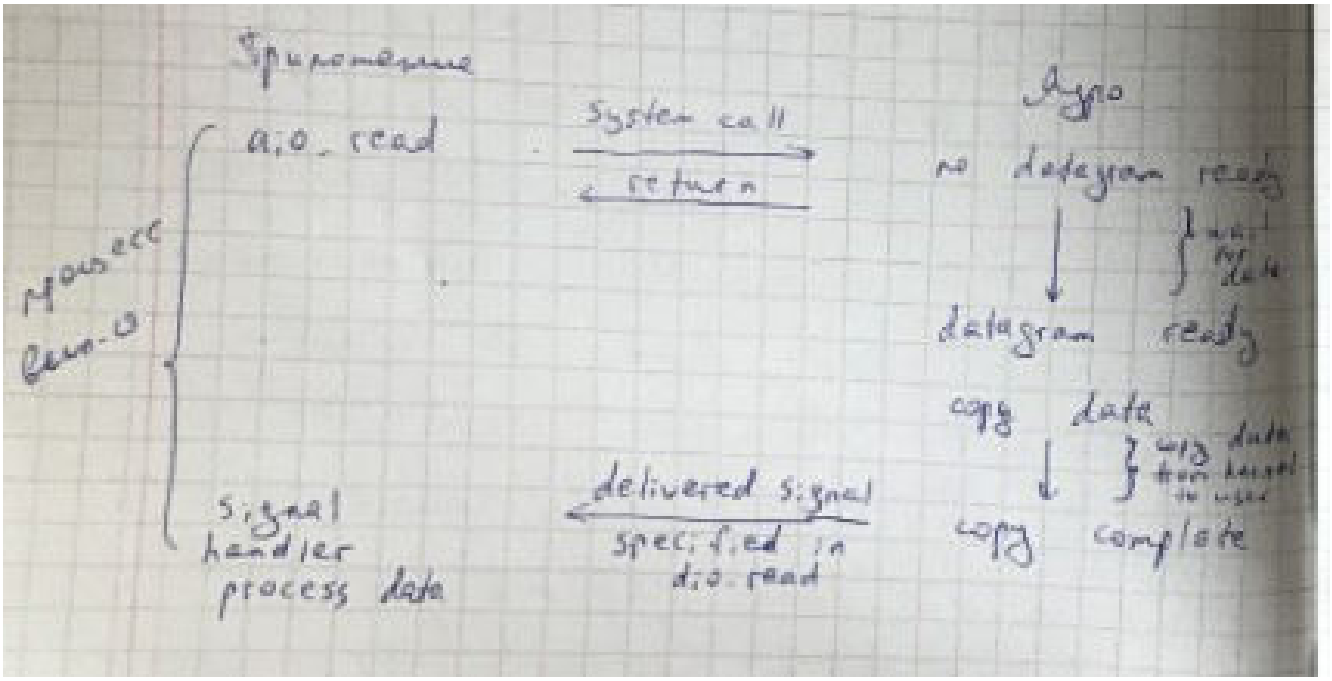
4) ввод \ вывод управляемыми сигналами (signal driving I\O)
Ассинхронный (блок или не блок потом определим)



Как видно из рисуночка необходимо установить обработчик сигнала signal _ handler . При этом используется сигнал SIGIO . Обработчик устанавливается системным вызовом sigaction. Входящим в POSIX 1. Результат выполнения системного вызова sigaction возвращается сразу. Приложение не блокируется, оно продолжает выполняться или лучше сказать может продолжать

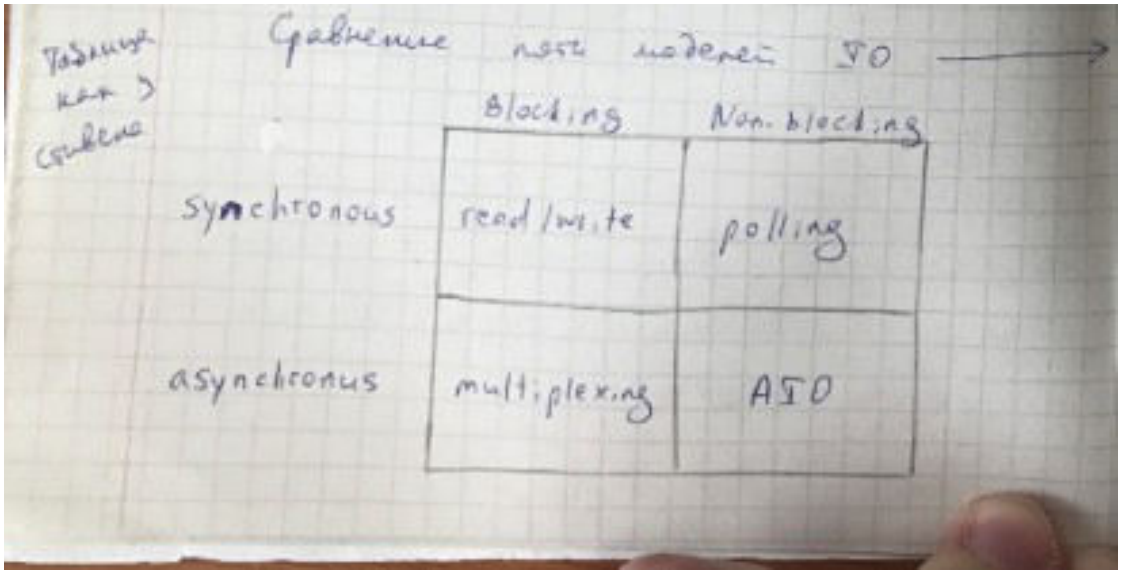
выполняться. Всю работу берет на себя ядро. Ядро отслеживает когда данные будут готовы . После чего посылает сигнал SIGIO. Который вызывает установленный на него обработчик. Это так называемая call_back функция. Вызов recvfrom можно выполнить либо в обработчике сигнала, который сообщит основному циклу, что данные готовы, либо в основном потоке программы. Ожидание может выплняться в цикле, который выполняется в основном потоке . Сигнал SIGIO для каждого процесса может быть только один . В результате за один раз можно работать только с одним файловым дескриптором. Важно отметить, что на время выполнения обработчика сигнала данный сигнал блокируется. Если в период блокировки сигнал доставляется несколько раз , то эти сигналы теряются. Если маска сигнала **sa_mask ==** null, то во время выполнения обработчика другие сигналы не блокируются .

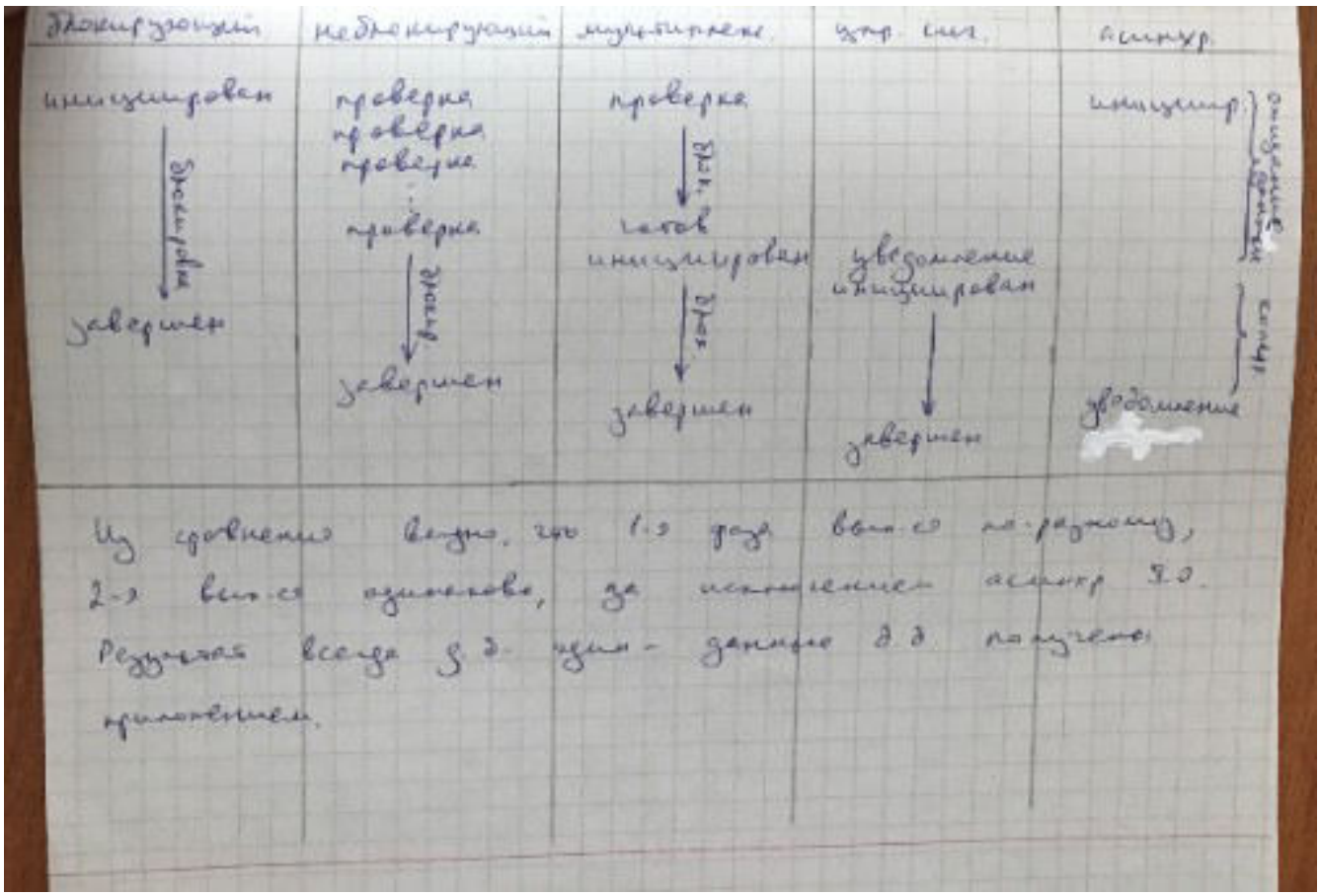
5) Асинхронный ввод \ вывод (неблокирующий)
Осуществляется с помощью специальных системных вызовов. Идея заключается в том , чтобы дать ядру команду начать операцию ввода\вывода ,а затем сообщить процессу или с помощью сигналов или еще каким либо способом , что операция ввода \ вывода завершена.



Из диаграммы видно что процесс может выоплняться., но в этом случае возникает одна и та же проблема . Эта пробелма состоит в том что необходимо получить асинхронное событие синхронно. POSIX определяет асинхронные фукнкциии A_IO или L_IO. Системный вызов A_IO_READ должен иметь параметры. Параметры этого системного вызвва следующие : дескриптор , адрес буфера, размер буфера , то есть параметры такие же , как в системном

вызове read. Стандарт POSIX или спецификация POSIX согласовала различие в фукнциях реального времени, которые появились в разных стандартах, объединив их. В основном эти фукнции работают таким образом , что сообщают ядру о начале операции и уведомляют приложения когда вся операция включая копирование данных из ядра в буфер приложения завершена . Основное отличие этой модели от модели ввода \ вывода управляемого сигналом заключается в том, что в модели управляемой сигналом ядро сообщает приложению когда операция ввода вывода может быть инициирована. А в асинхронном вводе выводе ядро сообщает приложению когда операция ввода вывода завершена.
Приложение пишет человек : Для того чтобы после запроса ввода вывода наше приложние продолжало что-то делать , это должно быть предусмотрено в программе. Здесь же предполагается что запросив данные программа должна продолжать что-то делать, естественно не связанное с запрошенными данными. Нельзя обрабатывать то, чего у нас нет .
Блокировки это зло. Они снижают производительность, увеличивают время выполнения процесса. Есть стремление от них избавиться.





Сравнение 5 моделей ввода вывода.
Из сравнения видно , что 1-я фаза выполняется по-разному. 2-ая выполняется одинаково, за искл асинхронного ввода вывода. Но результат всегда должен быть один — данные должны быть плучены приложением.

Предлагается определить какие способы ввода вывода попадут в клеточки.

По сокетам 2 программы : взаимодейтсвие двух процессов на одной машине (datagram сокеты семейство юникс) , сетевые юниксы с мультиплексированием. Соответственно модель клиент сервер. Мультиплексер выбираем любой.

Драйверы и прерывания

Драйвер может иметь 1 обработчик прерывания. Если устройство использует прерывания, то драйвер устройства региструрует один обработчик прерывания. Для обработки прерывания драйвер может разрешить определенную линию прерывания IRQ. Можно сказать что обработчик связан с определенной линией прерывания.
Драйвер регистрирует обработчик прерывания и связывает его с опр линий фукнцией 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 );
```

1-ый параметр — номер прервания, который будет обрабатывать обработчик.

Для некоторых устройств, например, для **legacy_pc_device** таких как системный таймер или клавиатура эта величина обычно устанавливается аппаратно **hard_coded** . Для большинства других устройств определяется программно и динамически.

2-ой параметр — handler
Это указатель на функцию обработчика прерывания — АКТУАЛЬНОГО обработчика , который обслуживает прерывание

Когда вызыывается этот обработчик?
Мы видим, что обработчик прерывания получает три параметра.

Типичное объявление обработчика :

```
static irqreturn_t intr_handler(int irq, void *dev_id, struct pt_regs *regs);
```

static irqreturn_t intr_handler

1-ый параметр irq – числовое значение прерывания, которое обслуживает обработчик

2-ой параметр dev_id – общий указатель на тот же самый dev_id , что задается request_irq , когда обработчик прерывания был зарегистрирован. Если это значение уникальное единственное , то оно выполняет роль различителя , то есть это значение используется для того чтобы различать множество устройств , которые потенциально могут использовать один и тот же обработчик прерывания. Dev_id может также указывать на структуру, которая используется обработчиком прерывания. Так как структура struct_device уникальна для каждого устройства и может потенциально использоваться обработчиком, то она обычно передается как dev_id

3-ий параметр regs – содержит указатель на структуру ,содержащую регистры процессора и состояние перед обслуживанием прерывания . Этот параметр испоьлзуется редко, обычно для отладки.
Возваращаемое значение тип irqreturn_t – фукнция может вернуть 2 определенных значения IRQ_NONE и IRQ_HANDLED

IRQ_NONE - возвращается когда обработчик прерываия обнаруживает прерывания, которое его устройство не инициировало
IRQ_HANDLED – возвращается когда обработчик прерываания был вызван корректно и его устройство действительно сформировало прерывание.
* конец 2-ого параметр *
=======
продолжаем request_irq
2-ой параметр — irq_flags – могут быть или нулем или битовой маской одной или более флагов
с флагами беда — они переписаны

Они описаны в linux / include / linux / interrupt.h

```
#define IRQF_SHARE
/*указывает что линия прерывания
может разделяться разными обработчиками прерывания
то есть на одну линию можно зарегистрировать несколько обработчиков */
#define IRQF_PROBE_SHARED
/*устанавливается вызывающими абонентами когда  могут иметь место несоответствия */
```

IRQF_SHARE
IRQF_PROBE_SHARED

Лекция 08-05

Регистрация обработчика прерывания request_irq .
С версии 2.6.19 флаги были координально изменены. Суть осталась той же. Например, сущ флаг SA_INTERRUPT – помечались быстрые прерывания. Приставку SA заменили на IRQ_F. Флаг interrupt вообще перестал сущ. Введен флаг IRQ_TIMER , который маскирует прерывания как прерывания от таймера. Быстрые и медленные прерывания это общий подход деления прерываний в системе. Для системы такое деление важно, так как быстрые прерывания выполняются на высоком уровне приоритета и прервать выполнение такого быстрого прерывнаия нельзя. Делается это засчет того что в СМП архитектурах запрещаются на данном процессоре все прерыванрия, на всех процессорах данное прерывнаие. Очевидно что бытрое прерывание по своему название означает что прерыванрие должно выполняться быстро иначе производительность будет снижаться. И таких прерываний не может быть много в системе. В настоящее время единственным быстрым прерыванием является прерывание от системного таймера.
Рассмотрим еще флаги
IRQF_SHARED – разрешает разделение или совместное использование линии IRQ
Данный флаг полезен , так как мы можем на этот флаг повесить собственное прерывание и увидеть наш обработчик прерывания в системе.

IRQF_PROBE_SHARED – флаг устанавливается, если предполагается возможность возникновения проблем при совместном использоании линии IRQ

IRQF_RERCPU – фла указывает, что прерывание закреплено за определенным процессором

IRQF_NOBALANCING – флаг, запрещающий использовать данное прерывание для балансировки IRQ

---

4 параметр в Функции
const char* devname - это ascii текст, представляющее устройство связанное с этим прерыванием
Например, это значение для прерывания от клавиатуры PC_TABLE

Текст имени или строка имени используется в /proc/irq
/proc/interrupts
что удобно для пользователя

5 параметр
void * dev_id - используетя для разделения линии прерывания
когда обработчик прерывания освобождается dev_id обеспечивает уникальные cookie файлы
обратить внимание на тип — может указывать на что угодно , но обычно используется указатель на структуру специфическую для устройства
В случае успеха request_irq функция возвращает ноль
Ненулевая величина означает ошибку
В этом случае обработчик прерывания не регистрируется
Обычная ошибка E_BUSY , которая означает что данная линия прерывания уже используется и либо текущий пользователь или мы не указали IRQF_SHARED

Медленные прерывания
Верхние и нижние половины
top_half & bottom_half in Unix and Linux

То есть обработчик прерывания делится на две части
1) остается обработчиком аппаратного прерывания
2) становится нижней половиной , выполнение которых возможно на более низком приоритете при более простых для системы условиях выполнения
в верхней половине — только самые необходимые функции, в нижней- завершение прерывания.

Быстрые прерывания фактически не дают выполнятсья системе никакой другой работе , поэтому они должны выполнять как можо меньший объем действий

Оставить у обраба только самые необходимые функции

Ззавершение обработки прерывания выполняется нижней половиной

Рассмотрим **например** обработку поступившего в сетевой адаптер пакета. Такая обработка требует несколько тысяч тактов перед тем, каак пакет будет передан в адресное пространство пользователя.
Несмотря на то что задача инициализируется прерыванием , тем не менее она не может быть выоплнена как быстрая процедура обработки . Решением является деление такого обработчика на две части.

Верхняя половина запускается в процессорах x86
Т е в результате возникшего аппартного прерывания будет запущен на выполнение обработчик прерывания и это будет верхняя половина.

Задачами Верхней половины являются:
1) так как обработчик верхней половины выполняется при запрещенных прерываниях, то обработчик возвращает выполнение ядру системы традиционным return

в случае адаптера копируем пакет в ядро. В ядре пакет ставится в буферную очередь . Где ставится в соотв поток и ожидает обработки
2) перед своим завершением должна обеспечить последующее выполнение нижней половины , которая позже завершит работу начатую верхней половиной
Верхняя должна поставить нижнюю половину в очередь на выполнение.
Делается это для разных типов нижних по-разному.

После return ядро завершает взаимодействие с аппаратурой контроллера обработчика
после return обработчика top half завершит взаимоисключение с аппаратурой контроллера, разрешает последующие локальные прерывания
Восстанавливает маску прерываний контроллера командой завершения обработки прерыания и возвращает управление из прерываия уже командой iret

В наст время имеются **три типа нижних половин**:
1) отлложенные прерывания (soft irq)
2) тасклеты
3) очереди работ

1) soft irq определяются статически во время компиляции ядра
В наст время в этой структуре только одна строка
версия ядра 4.10

```
<linux/interrupt.h>
struct softirq_action
{
    void (*action)(struct softirq_action *);
}
```

struct softirq_action

В большинстве источников в этой структуре есть еще одно поле (?)
В файле <kernel/softirq.c> определен массив из 32 экземпляров структуры soft_irq action

```
<kernel/softirq.c>
static struct softirq_action softirq_vec[NR_SOFTIRQS];
```

где NR_SOFTIRQS задействованное число номеров то есть имеется возможность создать 32 обработчиков soft_irq , в наст время определено 10.

Индекс	приоритет	
HI_SOFTIRQ	0	Высокоприор. Отл. Прер.
TIMER_SOFTIRQ	1	таймеры
NET_TX_SOFTIRQ	2	Отправка сетевых пакетов
NET_RX_SOFTIRQ	3	Пример сетевых пакетов
BLOCK_SOFTIRQ	4	Блочные устройства
BLOCK_IOPOLL_SOFTIRQ (нововведение)	5	Тасклеты
TASCLET_SOFTIRQ	6	тасклеты
SHED_SOFTIRQ	7	планировщик
HRTIMER	8	Не используется, но сохраняется для сохранения нумерации
RCU_SOFTIRQ	9	Должен быть последним soft_irq
NR_SOFTIRQ	10	

Когда ядро выполняет обработчик отложенния прерывания то фукнция action вызывается с указателем на соответствующую структуру soft_irq_action в качестве аргумента
Например, если переменная содержит xxx_softirq содержит указатель на элемент массива softirq_vec , то

```
char *softirq_to_name[NR_SOFTIRQS] =
{
    "HI", "TIMER", "NET_TX", "NET_RX", "BLOCK", "BLOCKIOPOLL", "TASKLET",
    "SHED", "HRTIMER", "RCU";
}
```

И массив softirq_to_name отражает индекс softirq на имя softirq

Если возникает необходимость добавить новый softirq , то обновляется softirq_to_name в kernel/softirq.c
Если выполнен вызов и указан индекс softirq , то ядро вызовет фукнцию обработчик соотв отложенного прерывания в виде
xxx_softirq→action(xxx_softirq)

Это значит что функции action передается вся структура, а не конкретное значение. Это может показаться странным
Данный трюк обеспечивает возможность дополнения структуры без необходимости внесения изменений в каждый обработчик softirq
Посмотреть softirq в работающей системе можно следующим образом
#cat /proc/softirqs
Добавить новый уровень обработчика можно только перекомпилировав ядро
Макс число обработчиков не может быть изменено динамически
Отложенные прерывания с меньшими номерами выполняются раньше

Для создания нового уровня softirq нужно :
1) определить новый индекс или уровень отложенного прерывания вписав его константу в перечисления
Очевидно, оно должно иметь уровень хотя бы на 1 меньше tasclet_softirq
Иначе подобные действия смысла не имеют

2) во время инициализации должен быть зарегестрирован обработчик отложенного прерывания с помощью вызова open_softirq

```
/*the bottom half*/
void xxx_soft_handles(void *data) {/**/}
void _init roller_init()
{
    /**/
    request_irq(irq, xxx_interrupt, 0, "xxx", NULL);
    open_softirq(XXX_SOFT_IRQ, xxx_soft_handler, NULL);//устанавл обр-к
}
```

_init roller_init()
фукнци open_softirq принимает 3 : индекс, фукнцию обработчик и значение поля дата
Фукнция обработчик должна сооответствовать правильному прототипу

3) зарегистрованная softirq должна быть поставлена в очередь на выполнение оно должно быть отмечено — используется слово raise

Для этого должна быть вызвана фукнция raise_softirq() - генерация отложенного прерывания
Обр-к верхней половины то есть аппаратное прерывание перед возвратом должно инициализировать выполнение нижней половины, в данном случае об-к возбуждает свой обр-к отложенного прерывания

```
/* The interrupt handler*/
static irqreturn_t xxx_interrupt(int irq, void *dev_id)
{
    /**/
    /*Mask softirq a pending*/
    raise_softirq(XXX_SOFT_IRQ);
    return IRQ_HANDLED;
}
```

static irqreturn_t xxx_interrupt

4) в некоторое время отложенное прер выполняеся. Обр-к отложенного прерывания выполняется при разрешенных аппаратных прерываниях на данном процессоре, но softirq запрещены. На других процессорах softirq могут выполняться, причем если во время выполнения определенного обработчика softirq генерируется такой же softirq, то он может начать выполняться на другом процессоре параллельно, поэтому обработчики softirq должы быть реентерабельными. А критические данные должны использоваться монопольно. Главная причина использования softirq это масштабируемость на многие процессоры, но требования к коду softirq очень серьезные. Проверка ожидающих выполнение обрабтчиков отложенных прерываний и их запуск осуществляется в след случаях :
1) при возврате из аппаратного прерывания
2) в контексте потока ядра **ksoftirqd (процесс демон)**
в любом коде ядра, в котором явно прверяются и запускются ожидающие обработчики отложенных прерываний , как например это делается в сетевой подсистеме
Независимо от вызова softirq его выполнение осущ фукнцией do_softirq, которая в цикле проверяет наличие отложенных прерываний .

Softirq никогда не вытесняет другой softirq . Единственное событие которое может его вытеснить это аппаратное прерывание.

Демон softirq
Это поток ядра каждого процессора PER_CPU , который выполняется кгда в машине запущены отложенные прерывания
Отложенные прерывания обслуживаются по возвращенным из аппаратного.

но возможно что отложенные прерывания переключаются значительно быстрее , чем может быть обслужено
Компьютер обменивается данными с устройствами используя irq , когда возникает прерывание от устройства, ОС прерывает выполнение текущей задачи и начинает адресовать возникшее прерывание. В некоторых ситуациях irq приходит очень быстро . В результате ОС не может закончить обслуживание одного до прихода другого. Такое может произойти например когда сетевая карта с высокой скоростью получает пакеты в течение короткого промежутка времени. Поскольку ОС не может справиться с такой ситуацией создается очередь и управлять этой очередью должен какой то поток — ksoftirqd.
Если демон softirq занимает больше чем небольшой процент процессорного времени, то это указывает на то что машина находится под большой нагрузкой прерываний.

Tasklet
Частный случай реализации softirq
Но тасклеты это отложенные прерывания для которых обработчик не может выполняться одновременно на нескольких процессорах. Название тасклет НЕ связано со словом таск — задание. Тасклеты надо понимать как простые в использовании отложенные прерывания.
Разные тасклеты могут выполнятсья параллельно на разных процессорах
Но два тасклета одного типа одновременно выполняться не могут.
В результате тасклеты являются хорошим компромиссом между производительностью и простотой использования. В отличие от softirq тасклеты могут быть зарегистрированы как статически так и динамичеки. Тасклет описывается структурой

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

struct tasklet_struct

Поле state может принимать одно из двух значений
enum { TASKLET_STATE_SCHED, TASKLET_STATE_RUN}

1) означает что тасклет запланирован на выполнение
2) тасклет выоплняется

count – счетчик ссылок. Если count == 0, то тасклет разрешен и может выполняться. Если он помечен как ждущий выполнения, иначе запрещен и не может выполняться.

Тасклеты могут создавтаься динамически и статически
Для статического создаия тасклета определено два макроса
DECLARE_TASKLET(name, func, data)
объявление тасклета с указанием имени и вызываемой функцией
DECLATE_TASKLET_DISABLE(name, func, data)
разница между макросами следующая — 1 создает тасклет у которого count == 0. соотвтенно он разрешен, 2 создает тасклет со значением count = 1, соответстенно тасклет будет запрещен.

Семинар 15-05

Тасклеты и очереди работ

Тасклет — частный случай реализации softirq. Это отложенные прерывания, но для тасклетов обработчик не может выполняться параллельно. Один и тот же тасклет не может выполняться параллельно, т есть одновременно на нескольких процессорах. Таск надо понимать как простые в использовании softirq. Тасклет явл хорошим компромиссом между производительностью и простотой использования.
Тасклеты могут быть зарегистрированы как статически, так и динамически.

```
<linux/interrupt.h>
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state; //текущее состояние
    atomic_t count; // счетчик ссылок
    void (*func)(unsigned long); // обработчик
    unsigned long data; // аргумент ф-ции обраб-ка
}
```

Тасклет описывается структурой :

Поле state может находиться в одном из двух состояний:
TASKLET_STATE_SCHED
TASKLET_STATE_RUN

1) указывает что тасклет запланирован на выполнение, то есть находится в очереди на выполнение к процессору
2) означает, что тасклет выполняется

Имеет смысл только для СМП архитектур.

Поле count : если == 0, то тасклет разрешен и может выполняться, если он отмечен как запланированный. Иначе тасклет запрещен и выполнятсья не может.

```
struct tasklet_struct name_t = {NULL, 0, ATOMIC_INIT(0), func, data};
```

Инициализация тасклетов
Статически:

```
<linux/interrupt.h>
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data);
```

Оба макроса статически, то есть во время компиляции, создают экземпляр структуры tasklet с именем name . Вызываемая фукнция func, которая передается аргументу data.
1-ый макрос создает тасклет, у которого поле count = 0. Следовательно этот тасклет разрешен на выполнение.
2-ой макрос создает тасклет со значение поле count = 1. Соответственно такой тасклет будет запрещен.

При динамическом создании тасклета объявляется указатель на структуру struct tasklet_struct

```
struct tasklet_struct *task;
tasklet_init(task, tasklet_handler, data);
```

Планирование тасклетов:

```
<linux/interrupt.h>
tasklet_schedule()
tasklet_hi_shedule()
```

Этим фукнциям передается один аргумент — указатель на структуру tasklet struct

Запланированные на выполнение тасклеты хранятся в двух связных списках. Структура tasklet_struct : первое поле next

tasklet_vec - обычные
tasklet_hi_vec - список высокоприоритетных тасклетов

Глядя на структуру taklet_struct эти списки состоят из экземпляров структуры tskletstruct

После того как тасклет запланирован, он будет выполнен один раз. Даже если он был запланирвоан на выполнение несколько раз

Для отключения конкретного тасклета используется фукнция
tasklet_disable() и **tasklet_disable_nosync()**

1-ая ф-ция — не сможет отменить тасклет, который уже выполняется.
2-ая ф-ция — может прервать выполняющийся тасклет

Для активизации тасклета используется ф-ция **tasklet_enable()**

Обработчик тасклета
Прототип
void tasklet_handler(unsigned long data);

В обаботчиках тасклетов нельзя использовать семафоры. Так как тасклеты не могут переходить в состояние блокировки. Если в тасклете используются общие с обработчиком или другим таклетом данные , то необходо реализовать взаимоисключение с активным ожиданием, то есть spinblock.

```
CONFIG_SMP
static inline int tasklet_try_lock(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_atomic();
    clear_bit(TASKLET_STATE_RUN, &(t)->state);
}
```

Отложенные прерывания и тасклеты. Сравнения
Softirq используются для запуска самых важных и критических по времени выполнения нижних половин

В настоящий момент только в двух подсистемах , а именно сетевых и блочных устройств, напрямую используется механизм softirq. Кроме того на основе softirq построены таймеры ядра и тасклеты.

Тасклеты имеют более простой интерфейс и упрощенные правила блокировок. Это связано с тем, что два тасклета одного типа не могут одновременно выполняться на разных процессорах. Но параллельно могут выполняться тасклеты разных типов. В силу указанных различий для задач критичных ко времени выполения и способных обеспечить взаимоисключение лучше использовать отложенные прерывания. Обработчик отложенного прерывания выполняется при разрешенных прерываниях и не может блокироваться. Во время выполнения обработчика отложенного прерывания на данном прцессоре отложенные прерывания запрещаются. Однако на другом процессоре они могут выполняться, причем даже параллельно могут softirq того же типа. Очевидно что для softirq должны применяться очень строгие выверенные способы взаимоисключения, с учетом того что softirq не могут блокироваться, то есть все делается на spinblock-ax.
Softirq масштабируемы. Если нет необходимости в масштабировании на бесконечное число прцессоров, то лучше использоваться тасклеты. Так как тасклет по сути это отложенные прерывание для которого обработчик не может выполняться параллельно на нескольких процессорах.

Очереди работ — третий тип отложенных действий который планируется обработчиками прерываний
Очереди работ существенно отличаются от тасклетов.
1) тасклеты выполняются в контексте прерывания и в результате код тасклета должен быть атомарным. Очереди работ выпоняются в контексте специального потока ядра . Как результат они являются более гибкими и в частности могут блокироваться.
2) Тасклеты всегда выполняются на процессоре , на котором выполнялось аппаратное прерывание. Очереди работ по умолчанию выполняются также. Но правило умолчания можно отменить.
3) Код ядра может запросить чтобы выполнение фукнции очереди работ было отложено на заданный интервал времени. Если сказать коротко, то ключевым отличием таскетов и очередей работ является то , что тасклеты выполняются быстро в режиме неделимости. А очереди работ могут блокироваться и не долнжы быть атомарными. Соответственно выбор того или иного способа реализации отложенных действий возлагается на программиста. Должен выбирать из соображений целесообразности.

Здесь отличаются понятия очереди работ и работа (действие).
Действие описываются структурой **struct work_struct**

```
struct work_struct
{
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
    #ifdef CONFIG_LOCKED
        struct locked_map locked map;
    #endif
};
```

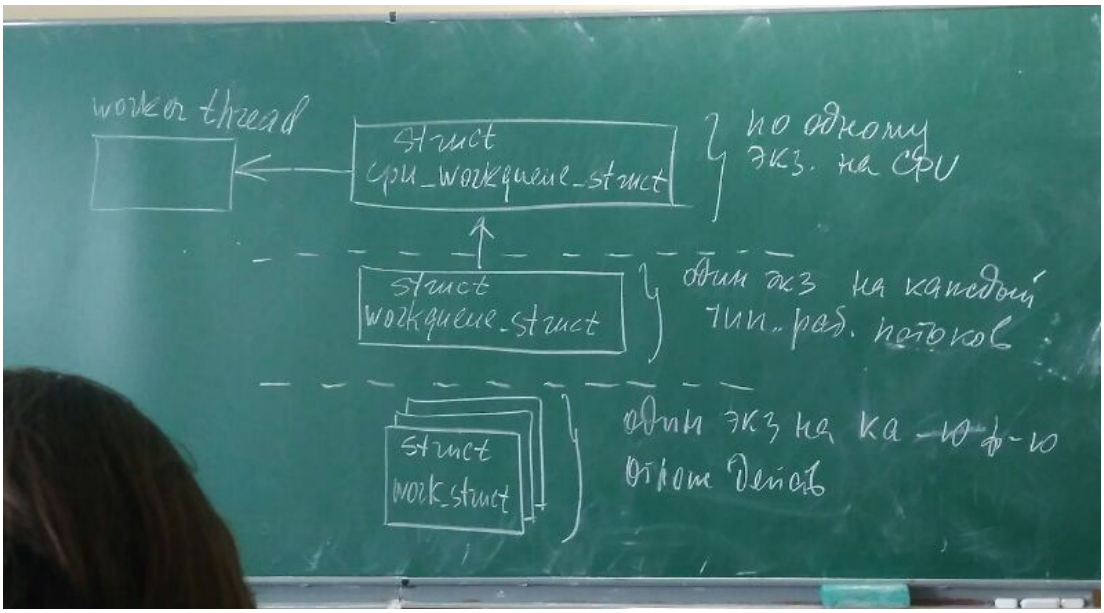
```
//организация очереди работ к процессору
//имеет смысл только для СМП архитектур
//определяет список на каждый процессор
struct workqueue_struct
{
    unsigned int flags;
    union
    {
        struct cpu_workqueue_struct percpu *pcpu;
        struct cpu_workqueue_struct *single;
        unsigned long v;
    }cpu wq;
    struct list_head list;//list of all wokqueues
    //..
    char name[]; // имя очереди работ
};

//так как рабочий поток сущ для каждого процессора в система
//то для каждого рабочего потока сущ такая очередь
```

А очередь работ описывается структурой **work_queue_struct**

```
<kernel/workqueue.c>
struct cpu_workqueue_struct
{
    spinblock_t lock;
    long remove_sequence;//удалить из последовательности
    long insert_sequence;//вставить в посл
    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; //глубина рекурсии ф-ции run_workqueue()
}
```

Поток **worker thread ← struct cpu_workqueue_struct ← struct workqueue_struct ← struct work_struc**

Worker-ами управляет основной планировщик ядра.



Worker-ы обеспечивают асинхронное исполнение запланированных отложенных функций — work-ов .

```
//ядро версии 2.6.36
struct workqueue_struct *create_workqueue(const char *name);
struct workqueue_struct *create_singlethread_workqueue(const char *name);
int alloc_workqueue(char *name, unsigned int flags, int max_active );
```

Flags – определяет как workqueue будет выполняться и параметр max_active ограичивает число задач.

WQ_UBOUND – очередь работ не привязана к конкретному процессору

Почему очереди работ привязывают к конкретному процессору? Это связано с расчетом на лучшее использование кеша.
Для того чтобы поместить работу или задачу в очередь работ нужно заполнить структуру work_struct. То есть инициализировать такую работу и
статически это делается с помощью макроса
DECLARE_WORK (name, void (*func)(void*));
name – struct work_struct *name;
func – фукнция которая вызывается из очереди работ — обработчик нижней половины

Для **динамического** задания структуры work_struct используются 2 макроса:
INIT_WORK(struct work_struct *work, void (*func) (void *), void * data);
PREPARE_WORK(-=-);

INIT_WORK – делает более тщательную работу по инициализации структуры . Этот макрос надо использовать если структура определяется первый раз . PREPARE делает почти ту же работу, но этот макрос не инициализирует указатли, используемые для связи work_struct с work_queue_struct . Если сущ вероятность, что стрктура в данный момент представлена и в work_queue ее нужно изменит, то используется PREPAE_WOKRK . Дл я отправки работы work в очередь работ исп 2 функции:
int **queue_work**(struct workqueue_struct *queue, struct work_struct *work);
int queue_delay_work(-=-, unsigned long delay);

Лекция 22-05
Идентификация УСТРОЙСТВА
идентифицируется так называемым старшим и младшим номером
идентифицируется типом dev_t

устройства бывают 2 типов:
символьные и блочные (рассматриваются системой как специальные файлы, в ОС такие файлы обозначаются С и В)
отдельно выдел сетевые

если в каталоге dev набрать ls -l мы увидим перечень устройств и увидим два номера : 1) старший — идентифицирует драйвер устройства(или драйвер связанный с устройством) 2) младший — идентифицирует конкретное устройство

Например, жесткий диск поделен на размеры
каждый такой раздел может содержать ФС
Диск идентифициуресят старшим номером
В системе есть драйвер жесктого диска
Разделы его будут иметь младшие номера

Коротко, диск имеет старший номер, разделы имеют младшие номера в порядке создания

POSIX_1 определяет существование типа dev_t , но не оговаривает формат полей и их содержание

Для разных версий системы формат полей может различаться
dev_t – 32 = 12 под старший + 20 под младший
dev_t определен в файле <sys/types.h> - определяет ряд зависимостей от типов данных — элементарные системные типы данных

Все объявлены через **typedef**

Также можно отметить тип **sigset_t (набор символов) , time_t (счечик секунд календарного времени)**
Определение типов выполнение таким образом, чтобы при написании программ не было необходимости погружаться в детали конкретной реализации , которая может меяться от системы к системе

Sigdev_t может находиться в /linux

Старшие и младшие номера устройства можно получить с помощью макросов, которые определены в большинстве реализаций
макрос major, minor:

```c
#include <sys/sysmacros.h>
unsigned int major(dev_t dev);
unsigned int minor(dev_t dev);
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

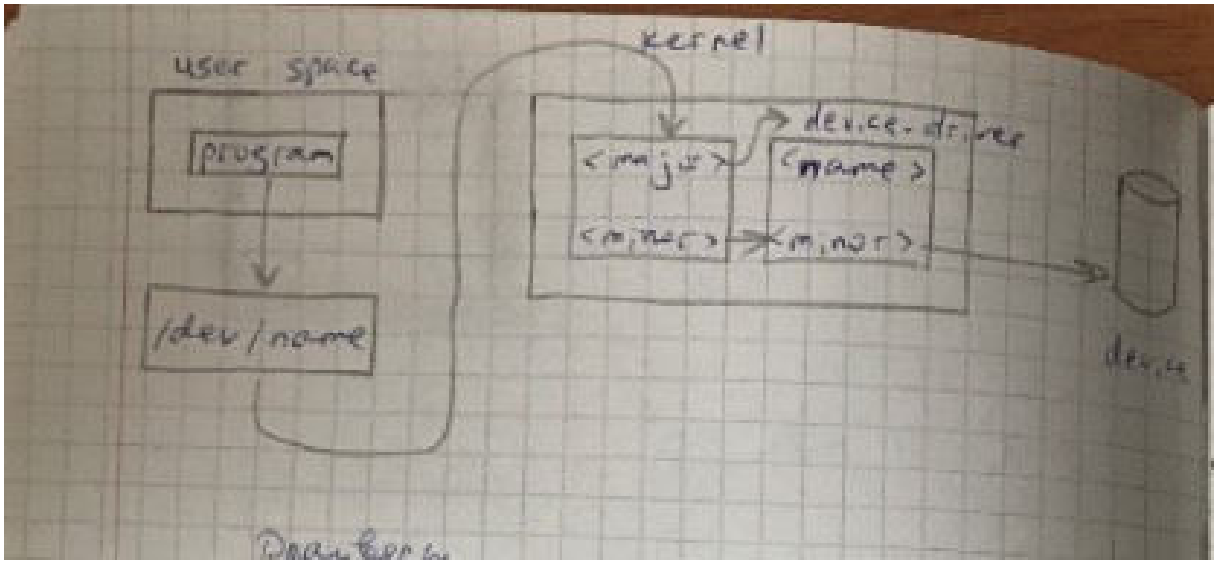
Наличие двух макросов избавляет от необх задумываться о том как хранятся эти два номера

В Л.Р. функция lstat была. Рассмотрим тип **stat**

```c
struct stat
{
    //описывает устройство, на к-ом находится файл
    //идентификатор устройства
    //макросы могут помочь разложить ИД на два числа
    dev_t st_dev;
    ino_t st_ino;
    //7 типов файлов:
    //S_ISREG(m)
    //S_ISDIR(m)
    //S_ISCHR(m)
    //S_ISBLK(m)
    //S_ISFIFO(m)
    //S_ISLINK(m)
    //S_ISSOCK(m)
    mode_t st_mode;
    //...
    //описывает устройство, кот представляет этот файл (inode)
    dev_t st_rdev;
}
```

```c
//для воспроизведения идентификатора устройства
dev_t makedev(unsigned int maj, unsigned int min);
```

user space kernel



ДРАЙВЕРЫ

Драйвер — программа или часть кода ядра, которая предназначена для управлеия кгнкретным устройством . Обычно драйверы устройств содержат последовательность команд специфичных для конкретного устройства. Кроме того это программы , которые имеют много точек входа или разные точки входа. Которые вызываются в зависимости от тех действий котоыре выполняются в данном устройстве .
В Linux драйверы бывают 3 типов:
1) драйверы, встроенные в ядро. Соответствующие устройства автоматически обнаруживаются системой и становятся доступными приложению. Примеры: VGA -controller, мат плата, посл и парал порты
2) драйверы, реализованные как загружаемые модули ядра. Такие драйверы часто используются для управления такими устройствами, как SCSI адаптеры, звуковые и сетевые карты . Файлы модулей ядра располаются в подкаталогах каталога **/lib/modules** . Обычно при инсталяции системы задается перечень модулей, которые будут автоматически подключаться на этапе загрузки. Список загружаемых модулей хранится в файле /**etc/modules.** В файле **/etc/modules.conf** находится перечень опций для таких модулей. Редактировать такие файлы в ручную нинада. Для этого существуют специальные скрипты типа update – modules . Для подключения и удаления к работающей системе имеются специальные утилиты lsmod insmod rmmod modprobe . Modprobe автматически загружает модули . Для того чтобы отобразить текущую конфигурацию всех модулей воспользоваться командой modprobe -c
3) драйверы, код которых поделен между ядром и спец утилитой . Например у драйвера принтера ядро отвечает за взаимодействие с параллельным портом. А формирование управляющих сигналов осуществляет демон печати , который использует для этого спец программу фильтров. Другие примеры — драйверы модемов.

Любое устройство , подключенное к системе регистрируется ядром и ему присваивается спец дескриптор в виде структуры struct device

устройство родитель, к котормоу подключается новое устройство. В большинстве либо шина, либо хост контроллер

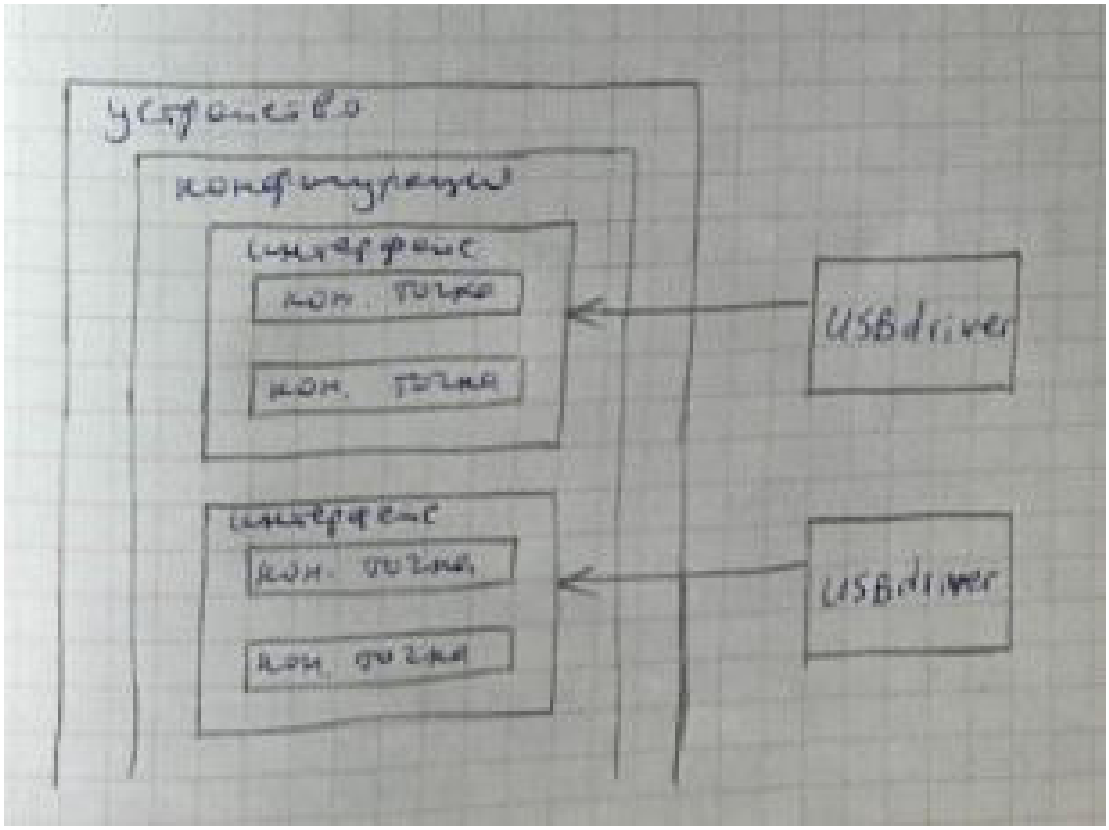
device – представляет модели устройств в linnux – эта структура не та структура с которой драйверы работают напрямую

Обычно можно найти эту структуру как говорят похороненной внутри специфической шины . Например ее можно найти как поле dev в struct_pci_device или struct usb_device .

```
struct device_driver
{
    const char *name;
    struct bus_type *bus;
    struct module *owner;
    const char mod_name;
    //...
    int(*probe)(struct device *dev);
    int(*remove)(struct device *dev);
    void(*shutdown)(struct device *dev);
    int(*resume)(struct device *dev);
    int(*suspend)(struct device *dev);
}
```

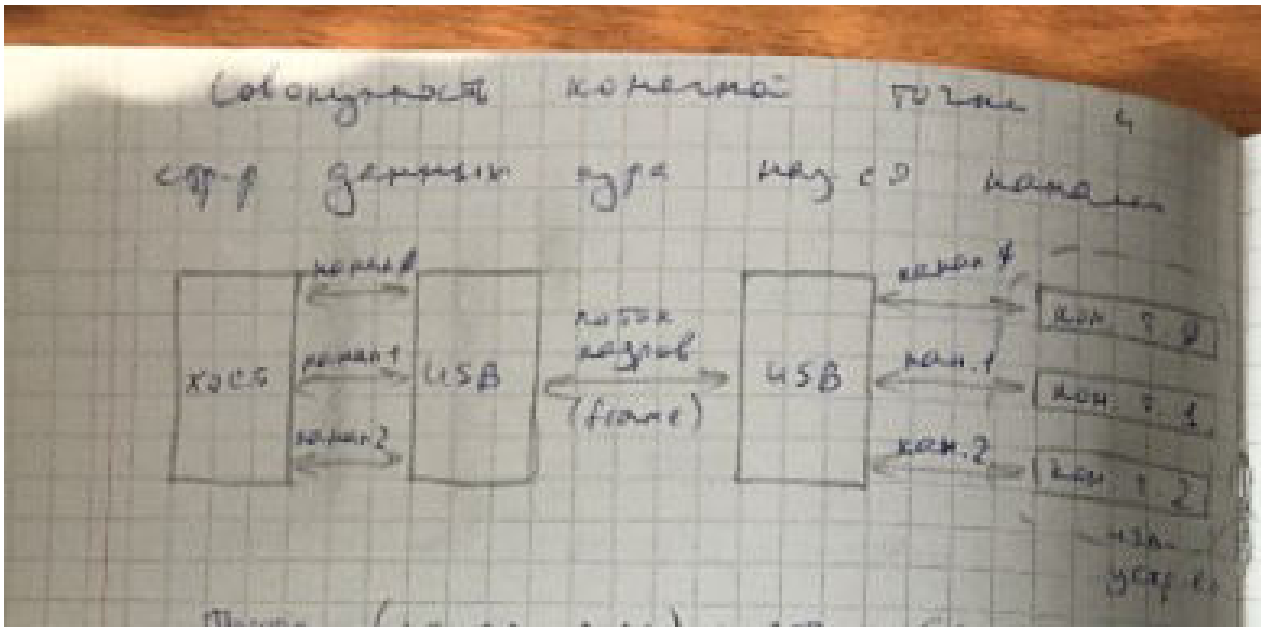
Struct device_driver

Usb_driver – рассматриваем так как USB-шина явл наиболее исплоьзуемой.
Подсистема usb драйверов связана с драйвером usb_core, который называется usb ядро. Usb шина явл шиной, в которой главным явл хост (переводится как главный) .
Хост начинает все транзакции usb шины. Называется она шина но построена она по принципу иерархии tree (дерево). Или иногда называются star_tree
Взаимодействие usb шины или usb хоста с устройствами выполняется всегда состороны хоста. То есть хост работает как мастер ведущий. Хост начинает все транзакции. Первый пакет, который называется токен, генерируется хостом для описания что будет выполняться — чтение или запись, и указывает адрес устройства и номер конечной точки — endpoint. При подключении устройства драйверы ядра ОС считывают с устройства список конечных точек и создают соотв управляющие структуры данных для взаимодействия с каждой конечной точкой устройства. Конечная точка устройства это программная сущность у которой есть свой уникальный идентификатор и которая может иметь буфер с некоторым числом байтов (заданного размера) для приема-передачи информаци и.



Конфигурация устройства

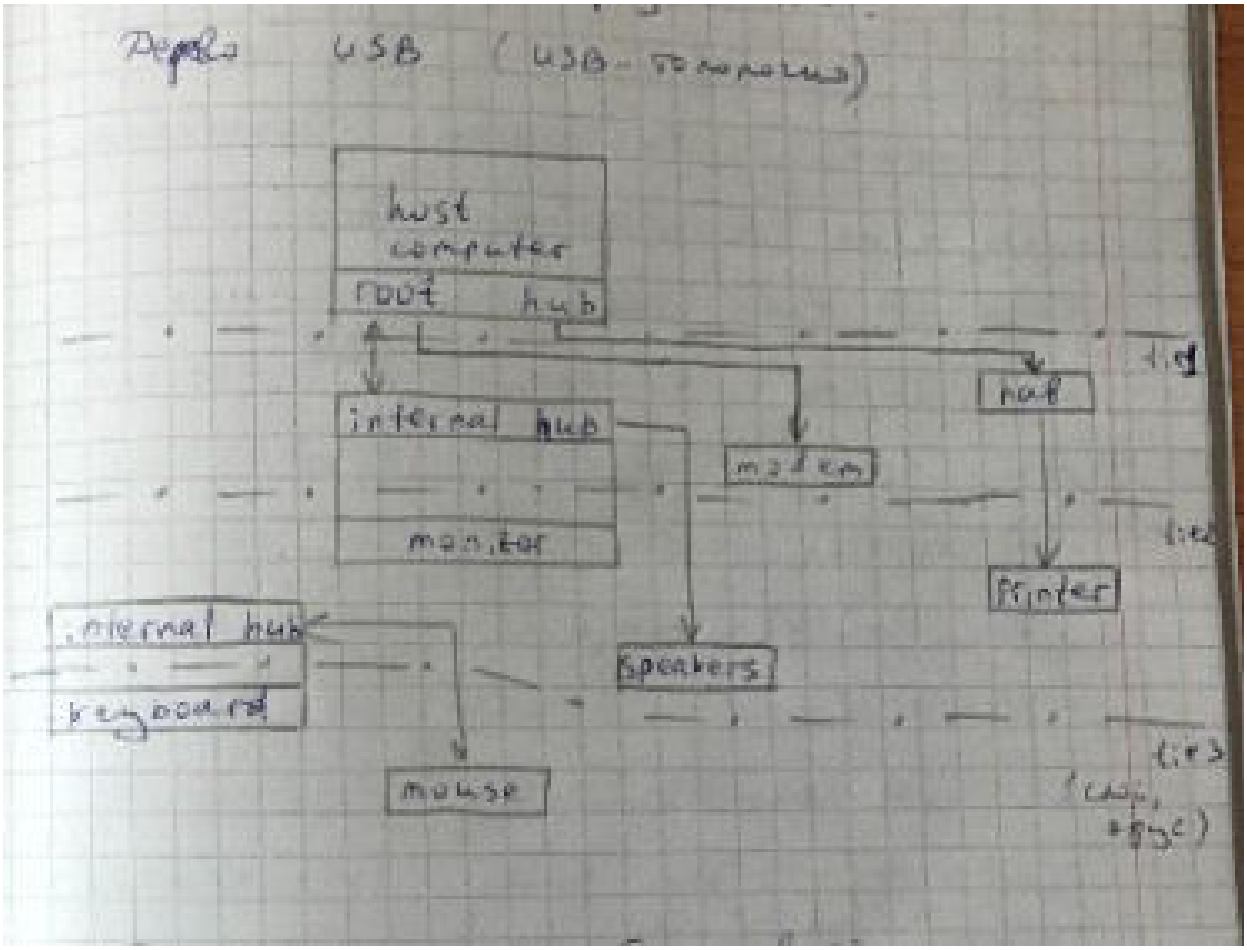
Совокупность конечной точки и структур данных ядра называется каналом — pipe.



хост — usb – usb – usb устройтво

Поток (pipe) - логическое соединение между хостом и конечной точкой. Потоки данных направленные , то есть имеют определенное направление передачи данных In или Out . Перед отправкой данные собираются в пакет .
Существует 4 типа пакетов:
1) token
2) data
3) handshake
4) SOF-start of frame

Usb устройства подключаются к узлам hub-ам . В узлах предусмотрены порты которые заканчиваются разъемами.



Дерево USB

Допускается до 5 уровней . Хост компьютер — корневой узел дерева : содержит неявные возможные implicit узлы (hubs). Фукнция hub распространять сигнал или передавать данные к одному или более портам путем увеличения общего количества функций которые разделяют шину. Hubs – активные электронные устройства . Многие устройства имеют встроенные хабы. На рисунке встроенный хаб имеет клавиатура
Существует 4 типа передачи данных, которые определены в universal serial bus specification:

1) controll
2) interrupt
3) bulk
4) isochronous
В результате конечные точки а значит и каналы относятся к одному из 4 типов. Поточный bulk, управляющий controll , изохронный и прерывания (interrupt)

Упр канал — для обмена короткими пакетами. Любое устройство имеет управляющий канал 0. Кторый позволяет ОС считать информацию об устройстве в том числе коды производителя и модели , которая исп для выбора драйвера , а также список других конечных точек.

Канал прерывания доставляет короткие пакеты in и out без получения на них ответа — подтверждения. Но с гарантией времени доставки . ТО есть пакет будет доставлен не позже чем через n милисекнд. Например этот канл исп в устройстве интерактивного ввода клавиатура мышь джойстики.

Изохронный — доставляет пакеты без гарантия доставки и без ответов — подтверждений. Но с гарантированной скоростью доставки. То есть n пакетов за один период шины. Для передачи видео и аудио информациии.

Поточный bulk – дает гарантию доставки каждого пакета, поддерживает автоматическую остановку передачи данных от устройства например в результате заполнения буфера или опустошения буфера. Но не дает гарантий скорости и времени доставки. Используется в принтерах и сканерах .

Каждое подключенное к машине usb устройство представляется в usb core структурой **struct usb_device** .

```
struct usb_device
{
    int devnum;//номер ус-ва на usb-шине
    char devpath[16];//путь к файлу
    enum usb_device_state state;//состояние уст-ва
    struct usb_host_endpoint ep0;//данные первой кон.т.
    struct device dev;
    char *product;
    char *manufacturer;
    char *serial;
}
```

Лекция 05-06

Управление буферами

Буфер — это область оперативной памяти для промежуточного хранения информации . Обычно используются для выравнивания разности скорости обработки информации в процессе передачи данных. Или между двумя процессами или между процессом и устройством или между двумя устройствами.

Такой буфер как программный канал сам по себе может явлться средством передачи информации. Но этот буфер синхронизирует выполнение чтение и записи .

В отличие от буфера кеш память (кеш буфер) — быстродействующее энергозависимая Оперативная Память, в которую записывается копия команд или данных из более медленной памяти для ускорения процесса их дальнейшей обработки. Исполльзование кеш памяти повышает быстрдействие системы в целом или какой то из ее подсистемы. В базовой подсистеме ввода вывода смешивать понятия буферизация и кеширование не следует. Очень часто содержит единственный набор данных, сгенерированный в системе. (в качестве примера программный канал)

Кеш (по назначению) ВСЕГДА! содержит копию данных, которые хранятся еще где-то в системе.

Cash TLB (адреса страниц к которым были последние обращения)
но эти же адреса в таблице страниц есть
Также могут находиться в ОП
Т.е. в системе может быть несколько копий одних и тех же данных
При этом буфер может содержать единственный экземпляр данных в системе

При рассмотрении различных методв буферизации необходимо учитывать сущетвование двух типов устройств, а именно блок ориентированных (в которых инфа хранится блоками) и байт ориентированных (в которых инфа хранится байтами ,а передача информации выполняется не структурированным потоком байтов) . Здесь слово неструктурированный подчеркивает что нет смыслового объединения байтов (байт за байтом идет).

К блок ориентированным устройствам относятся диски
К байт ориентированнм относятся все остальые (принтеры , мышь коммуникационные порты , терминалы, джойстики, а также большинство устрйоств которые не явл внешними запоминающими устойствами)

Особую роль буферы играют в файловом вводе \ выводе. Для буферизации файлового ввода вывода обычно выделяется не один, а несколько буферов которые называются буферным пулом (пул == объединение).

Для управления буферами создается специальная таблица . Каждый элемент которой описывает один буфер.

Примерно в таком виде
/*
I-ый буфер пула
дескриптор I-ого буфера

таблица упр буферным пулом

адрес начала — рамер — состояние
*/
Буферный пул или отдельный буфер может быть создан
а) статически (во время выделения памяти задаче) такой буфер будет существовать в памяти все время существования задачи.
Слово задача — поразмышлять
б) динамически . Очевидно , что динамическое создание явл более экономным . Буферный пул или буфер создается перед началом обмена, например с внешним устройством и занимаемая область памяти освобождается когда обмен завершается.

Естественно система должна предоставлять средства работы с буферами. (средство pipe – буфер типа fifo. Тут если не устанавливать флаг , то можно создать буфер) . Буф пул может быть сформирован одним из следующих способов в самом общем виде :
1) в программе может быть создан буфер с помощью специальной команды
2) создание буфера с помощью макро команды, например, get pool – free pool (доджны быть парой)
3) получение от системы разрешения создавать буферный пул автоматически при открытии набора данных
Соотвественно буферный пул может быть создан статически в процессе трансляции , динамически в процесе выполнения программного кода или ОС может создавать созданный пул автоматически

Для упр буферами самой ОС используется две схемы управления
1) простая буферизация
2) обменная буферизация (exchange buffering)
Схемы простой буферизации
* картинка *
а)
ввод вывод
входной буфер → рабочая область → выходной буфер
(пересылка для обработки) (пересылка для вывода)

б)

ввод и обработка вывод

входной буфер → выходной буфер

(пересылка для вывода)

в)

ввод обработка и вывод

входной буфер → выходной буфер

(пересылка для обработки и вывода)

При простой буферизации выполняется большое число пересылок данных внутри оперативной памяти , что может привести к значительным временным затратам. Достоинством является простота , идеи, очевидность выполняемых действий, так как связи между буфером и внешним устройством не меняются. Способ простой буферизации применяется в тех случаях, когда объем обрабатываемых данных относительно небольшой и обработка данных в ОП занимает достаточно большое время.

Схемы обменной буферизации

а)

ввод вывод

входной буфер | рабочая область -> выходной буфер

пересылка для вывода

взаимно меняются

без пересылки данных

б)

ввод

входной буфер → раб. Обл. Вых. Буфер

взаимнно меняются

в)

ввод и обработка вывод

входной буфер выходной буфер

взаимно меняются

без пересылки данных

Достоинства обменной

полностью или частично устраняе необходимость пересылки информации из буфера в буфер

Схема пок как меняется буфер и рабочая облась

Данные с внешнего устр считываются во входной буфер по завершении считывания входной буфер объявляется рабочей область, а раб обл становится входным буфером
Когда первая порция данных будет обработана и будет отправлена во входной буфер, раб обл и вх буф снова поменяются ролями

Обменная буферизация требует соответствующих действий со стороны ОС . Она должна следить за сменой функций буферов а также обладать информацией о том, какую функцию в каждый момент времени выполняет каждый из областей. Кроме этого система должна заменять адреса ввода у подрограмм ввода и рабочего массива у обрабатывающего прикладного процесса. Происходит динамическая смена адреса — это должно отражаться. Меняется адрес буфер, то есть меняется ссылка на буфер и тд.

(книга Шоу)
Пул буферов — память выделенная для кеширования страниц.
В самих ОС присутствует название буфер и название кеш.

==========
Обратить внимание (было сказано неверно):
Системные вызовы
Сис выз часто опр как запрос приложения на сервис системы. Но есть более строгое предсатвление о сис вызове. Когда программе ввод вывод , например программа на си, можно юзать либу stdio. Если посмотреть код, то там находится много if-ов. В либе дожен быть системный вызов — до этоо всегда говорили что это read write. В режим ядра систему переводит сис вызов , кот определяется как int
В любой ОС системный вызов , то есть запрос обслуживания, выполняется некоторой процессорной инструкцией (машинной командой), прерывающей последовательность выполнения команд приложения и передающей управление коду режима ядра или суперюзер.
При вызове функции мы переходим по адресу данной функции, то есть на другую область память, но эта обл вс еще usermode и уже в ней системный вызов , который переведет режим пользователя в режим ядра.

Команды типа **int**
Если платформа x86
MS-DOS 21h
Windows – 2Eh
Linux – 80h
QNX – 21h
MINIX3 – 21h

Обычно используются следующие мнемоники:
svc – supervisor call
cmt
trap

Spin блокировки в ядре
В ядре линукс спин блокировки реализуются посредством переменной типа spinlock_t
Данный тип соответствует типу integer
Основной интерфейс spinlock_t содержит два вызова:(внутри test_and_set – блокировка шины памяти — негавтиный момент; while test-and-set заменен на while)
spin_lock(spinlock_t *sl) захват
spin_unlock(spinlock_t *sl) освобождение

Кроме этих двух основных функций еще есть дополнительные:
spin_lock_irqsave(spin_lock_t *sl, unsigned long flags)

Данная штука блокирует вход в критический участок и дополнительно предотвращает выполнение прерываний, но только на локальном процессоре и сохраняет регистр состояния процессора в переменной flags.

spin_lock_irq(spin_lock_t *sl);
она не сохраняет значение регистра состояния процессора - использовать ее не рекомендуется
Но эту блокировку можно использовать, если перед ее захватом прерывания разрешены ,а при освобождении блокировки прерывания можно просто разрешить. То есть чтобы использовать блокировку нужно убедиться что прерывания разрешены.

spin_lock_bh(spin_lock_t *sl);
bh – bottom half
данная команда пытается установить блокировку , но одновременно предотвращает запуск нижних половин обработчиков прерываний

Для освобождения критической секции. Для выхода из блокировки.
spin_unlock_irqrestore(spin_lock_t *sl, unsigned long flags);
Освобождает спин блокировку и разрешает прерывания, если при сохранении регистра состояния процессора прерывания были разрешены. В противном случае прерывание завершается (?). Если при захвате блокировки прерывания были запрешены, то при освобождении блокировки они разрешены не будут.

spin_unlock_irq(spin_lock_t *sl);
Освобождае спин блокировки и разрешает прерывания безусловно.

spin_unlock_bh(spin_lock_t *sl);
Освобождает блокировку и разрешает немедленную обработку нижних половин.

Примерчик
spinlock_t sl

```
#include <linux/spinlock.h>
spinlock_t sl = SPIN_LOCK_UNLOCKED;
// p1
spin_lock(&sl);
spin_unlock(&sl);

//p2
spinlock(&sl);
spinunlock(&sl);
```

try_lock – пытается захватить блокировку. Если уже захвачена. То вместо циклической проверки сразу возвращается ненулвое значение, то есть процесс не переходит в активное ожидание

spin_is_locked – используется для получения текущего значения спин блокировки, при этом состояние спин блокировки не меняется.

Семафоры (-1, 1, 0)

Спинблокировки чтения и записи
В системе часто возникают ситуации структуры данных активно считываются и довольно редко изменяются. Примером может служить список зарегистрированных сетевых устройств **dev_base .** Этот список часто считывается и редко изменяется. Такую ситуацию учитывают блокировки чтения записи. (читатели писатели) Для реализации таких спинблокировок используется тип **rw_lock_t.** Блокировку по чтению могут одновременно захватить несколько потоков, выполняющих чтение. Блокировку по записи может захватить только один поток. При этом чтение данных запрещено.
Система предоставляет следующие функции:
void **read_lock**(rwlock_t *rw);
void **write_lock**(rwlock_t *rw);
Дополнительно для читателя:

void **read_lock_irqsave**(rwlock_t *rw, unsigned long flags);
void **read_lock_irq**(rwlock_t *rw);
void **read_lock_bh**(rwlock_t *rw);

Для записи

void **write_lock_irqsave**(rwlock_t *rw, unsigned long flags);
void **write_lock_irq**(rwlock_t *rw);
void **write_lock_bh**(rwlock_t *rw);
void **write_try_lock**(rwlock_t *rw);

соответвующие unlock-и

* написать соотв код на экзе *

Спинблокировки предполагают активное ожидание, то есть процессы расходуют процессорное время, ожидая освобождение ресурсов. Альтернативой спинблокировок являются средства взаимоисключения , которые переводят процессы в состояние блокировки. Если процесс не может войти в критический участок. В этом случае процессор может начать выполнение другой работы, но для такой реакции системы также характерна издержки, а именно два переключения контекста. Очевидно, что переключение контекста требует выполнение довольно большого объема кода. Который значительно больше тех строчек кода, которые потребовали взаимоисключения. Поэтому разумно использовать спинблокировку если время ее удержания меньше длительности двух переключений контекста. Такая оценка затратна. Поэтому разработчики протос должны учитывать факт активного ожидания. И удерживать блокирвки по возможности максимально короткого времени. В многопроцессорных системах время которые удерживается спинблокировки эквивалетно времени задержки системного блокировщика. В линукс спиблокировки не рекурсивны, что может привести к само блокировке. Если поток войдет в цикл ожидания освобождения блокировки, которую сам захватил ранее. То есть один поток выполняет два раза спинлок и через какое то время спин лок.
Спинблокировки могут использоваться в обработчике прерываний. Семафоры в обаботчике использовать нельзя (речь идет об аппаратных и о тасклетах и softirq , очереди работ могут засыпать) . Если блокировка используется в обработчике прерывания, то перед тем как ее захватить где-то в другом месте (не в обработчике прерывания) необходимо запретить все локальные прерывания на текущем процессоре. В противном случае может возникнуть такая ситуация, когда обработчик прерывания прервет выполнение кода ядра, удерживающего блокировку, а затем снова попытается его захватить. При этом блокировки не освобождаются. В это время обработчик прерывания начнет проверять не освободилась ли нужная ему блокировка. С другой стороны, код

ядра который удерживает блокировку не сможет выполняься до тех пор пока обрбаотчик прерывания не завершит выполнение. Возникает dead_lock . Правило блокировки : нужно защищать данные, а не код. Блокировка должна быть связана с данными и их защищать.