# Семинары

## 20-02

### Загружаемые модули ядра

Линукс и Юникс предоставляют в распоряжение разработчиков  ЗМЯ.
с помощью загружаемых модулей ядра разраб может **добавить(!)** в ядро нужную ему функциональность.

Значение имеет версия ядра, так как постоянно система переписывается
Версия 2.6 — водораздел и т. д. (9, 12…)

Нужно использовать мануал.

Любой загружаемый модуль ядра должен содержать ДВА обязательных макроса. Это точки входа. Когда имеем дело с ядром надо понимать что наши модули имеют много точек входа. Мы используем функции не так, как в приложении (типа напиали фунцию и вызываем в main) . Тут иначе : функцию описали, но нигде ее не вызываем. В макросах ф-ции могут не вызываться. Это значит, что эти функции будут вызываться по определенному событию в системе.
init_module(void)

```c
//модуль init и модуль exit

#include <linux/module.h>
#include <linux/Kernel.h>
//KERN_ALERT <1>
//<4> лучше не использовать - KERN_WARNING
int init_module(void)
{
    //пишет в файл ядреных сообщений
    printk(KERN_INFO "Hello, World!\n");
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_INFO "GoodBye!\n");
}
```

Напишем свои функции и передадим их макросам.
Макрос init вызывается когда наш модуль загружен и начинает работать , а модуль exit когда выгружаем модуль .

Чтобы с этим работать должен быть файл .c и makefile
module_init(hi)
module_exit(bye)

```c
#include <linux/module.h>
#include <linux/Kernel.h>
#include <linux/sched.h>

static int hi(void)
{
    printk(KERN_INFO "Hi, Module has been loaded.\n");
    printk(KERN_INFO "Current process ID is %i\n", current->pid);
}

static void bye(void)
{
    printk(KERN_INFO "Bye\n");
}

module_init(hi); //при подключении модуля к ядру (желательно в названии hi использовать init)
module_exit(bye); //при отмонтировании модуля из ядра (вынимании)

MODULE_AUTHOR("Guru linux");
MODULE_LICENSE("Dual BSD/GPL"); //так принято
```

<module.h> обязательная либа
<kenel> так как используем printk и приоритеты KERN_ALERT \ INFO и т. д.

Компиляция выполняется при определенных условиях:
необходимо находиться в директории , в которой находится makefile, и утилита make

Загрузить созданный модуль (мы можем загрузить только откомпилированный модуль) и начиная с ядра 2.6 откомпилированый модуль имеет расширение <имя.ko>

Загружается наш модуль командой **sudo insmod  <имя.ko>**
insmod – insert module

Обычно макрос подставляется как текст, а здесь не перекомпилируем ядро поэтому вставляем такие же откомпилированные

**sudo rmmod <имя>**

**<sys/init_module >** выделяет в памяти ядра, причем эта память выделяется в функции **vmalloc**()

При указании insmod вызывается модуль init, а при вызове нее вызывается sys_init_module(), внутрии нее вызывается vmalloc(), чтобы была память для размещения этого модуля

Посмотреть сообщения , которые пишет printk можно использовать **dmesg** .
Информация выводится в журнал **var/log/message**

**MAKEFILE**
утилита make

Смотрит на время создания объектного файла и время редактирования исходного файла.
Если первое позже второго, тогда перекомплиировать не надо.

Если вызываем на одной машине, тогда проблем со временем возникнуть не может.

В ЗМЯ мы используем API ядра. Про них надо знать:
функции реализованы в ядре и несмотря на то что имеется совпадение многих из этих фукнций по форме с вызовами станд либы си или с сист вызоваи из user_ mode по форме это совершенно другие функции. Заголовочные файлы для фукнции простарнства пользователя располагаются /user/include , а для API ядра это каталог /lib/modules/uname/build/include

makefile содержит описания требуемых действий:
как надо компилировать и собирать программу

Простой makefile состоит из правил (rules) следующего вида:
target (цель) … : реквизиты (prerequisite) …
    команда
    команда
    …
последовательность команд: каждая на отдельной строке
команда начинается с символа табуляции

target – имя файла (цель), который генерируется в процессе работы утилиты make – это могут объектные или исполняемые файлы собираемой программы

цель может быть именем некоторого действия, которое нужно выполнить - например clean

Реквизиты — файл , который используется как исходные данные для порождения цели
Очень часто цель зависит сразу от нескольких файлов

Команда — действие , которое выполняется утилитой make

Rule – какие надо выполнять файлы, указанные в качестве цели

Простой makefile может состоят только из правил
Для удобочитаемости в makefile используется «\» . Он используется для переноса на новую строку — перевод каретки. Чтобы с помощью  makefile создать исполняемый файл следует вызвать make. Для того чтобы удалить из директории .exe и .o надо выполнить команду make clean

Пример easy_make

```
obj - m += LoadableKernelModule
all:
    make -c /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -c /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

В качестве переменных используются следующие:
objects, OBJECTS, objs, OBJS, obj, OBJ

Пример (на переменные)

```
objects = main.o kbd.o search.o
edit: $(objects)
    cc -o edit $(objects)
main.o: main.c defs.h
    cc -c main.c
kbd.o: kbd.c defs.h command.h
    cc -c kbd.c
command.o: command.c defs.h command.h
    cc -c command.c
clean:
    rm edit $(objects)
```

Стандартные цели . Краткий список
1) all  - стандатраня цель по умолчанию. при вызовее make ее можно явно не указывать
2) clean – очистить каталог от всех файлов, полученных в результате компиляции
3) install
4) unistall

Фективные цели **phony:**
для того чтобы утилита make не искала с только что перечисленными именами их определяют в makefile  при помощи директивы

Пример

```
.PHONY : all, clean, install, unistall
```

## КАК СОЗДАТЬ ПРОСТУЮ ФС

ОС линукс поддерживает огромное количество ФС. Линукс предоставляет оч простой интерфейс для разработчиков, пишуших свои ФС. Подчеркивается что написать свою ФС можно с мин усилиями.
БОльшая часть read_only

Разрабы Линукс создали очень простой интерфейс. Цель — возможность перепроектировать существующие ФС , написав их только для чтения.

Интерфейс VFS экспортируется и поэтому все ФС линукс реализуются как загружаемые модули.

ФС имеют разные форматы хранения данных — в лиункс для этого определена структура  **struct file_system_type**

Сущ 4 основные структуры : супер блок, dentry, inode, file (с лекций) — интерфейс для работы с файлами у ОДНОЙ системы

Сейчас рассматриваем интерфейс для работы С РАЗНЫМИ ФС

Чтобы создать новую ФС — требуется заполнить эту структуру и зарегистрировать ее в VFS . Регистрация выоплняется фукнцией
**register_file_system();**
Структура struct file_system_type объявлена в **linux/fs.h**

```
struct file_system_type
{
    const char *name;
    int fs_flags;
    //требуется блочное устройство
    #define FS_REQUIRES_DEV 1
    //монтируемые данные являются бинарными
    #define FS_BINARY_MOUNTDATA 2 // 2.6.5
    struct dentry *(*mount)(struct file_system_type *, int, const char* , void*)
}
```

Семинар 06-03

Создание своей фс
пишем фукнции которые будут являться точками входа(их несколько)
Допишем несколько полей в структуре:

```
struct file_system_type
{
    const char *name;
    int fs_flags;
    //требуется блочное устройство
    #define FS_REQUIRES_DEV 1
    //монтируемые данные являются бинарными
    #define FS_BINARY_MOUNTDATA 2 // 2.6.5
    struct dentry *(*mount)(struct file_system_type *, int, const char* , void*);

    void (*kill_sb)(struct super_block *);
    struct module *owner;
    struct file_system_type *next;
};
```

Объявлено 7 средств взаимоисключения. Приводим 1 пример.

Dentry – в современной версии 4.10
в **Старой** версии вызывалась ф-ция **read_super**

Поле **name**. Имя должно быть уникальным. Для модулей имя указывает на адресное пространство модуля.

Дальше указано два флага.
Первый флаг указывает что для ФС требуется блочное устройство.
Второй флаг …

Функция mount (read_super) – считывает super_block в процессе монтирвания (то есть функция вызываетя при монтировании). Эта фукнция дложна быть определена обязательно. Если фукнция отсутствует , то операция монтирования независимо оттого из пользовательского вы монтируете или из ядра, всегда будет неудача.

**Пример** определения полей структуры file_system_type

Поля mount и kill_sb содержат указатели на функции
1-ая вызывается при монтировании ФС , а 2-ая при размонтировании

Достаточно реализовывать только 1 функцию. вместо 2-ой можно использовать функцию kill_superblock (), которую предоставляет ядро .

Поле owner нужно для организации счетчика ссылок на модуль. Счетчик ссылок нужен чтобы модуль не был выгружен, когда ФС примонтирована. Это может привести к краху . Счетчик ссылок не позволит выгрузить модуль , если он используется. То есть до тех пор пока ФС не будет отмонтирована.

```
struct file_system_type my_fs_type =
{
    .owner = THIS_MODULE,
    .name = "my", //enc_fs
    .mount = my_mount,
    .kill_sb = my_kill_superblock,
    .fs_flags  = FS_REQUIRES_DEV,
};
```

Подмонтировано может быть несколько ФС например ext2. Но тип один

ФС создаюся в виде загружаемых модулей ядра. Каждая примонтированная ФС на диске занимает раздел. (если это дисковая ФС).

ФС описывает super_block . То есть он хранит информацию о физическом расположении той информации которая нужна для работы с ФС.

**Регистрация и дерегистрация ФС**

Ядро было полностью переписано на версии 2.6 (НАПОМИНАЛКА)

Смотри версию , дебил

Это (kern_err kern_info)называется log_level – уровень вывода сообщений ядра . Они определены в /kennel.h

```c
static int __init my_init(void)
{
    int ret;

    //передаем адрес нашей структуры (описана ранее)
    ret = register_filesystem(& my_fs_type);
    if( likely(ret == 0))
        printk (KERN_INFO "Succes \n");
    else
        printk(KERN_ERR "Error %d\n", ret);

    return ret;
}

static void __exit my_cleanup(void)
{
    int ret;

    ret = unregister_filesystem(& my_fs_type);

    if( likely(ret == 0))
        printk (KERN_INFO "Succes \n");
    else
        printk(KERN_ERR "Error %d\n", ret);
}
```

```c
module_init(my_init);
module_exit(my_cleanup);
```

module_init(my_init)
module_exit(my_cleanup)
Функция маунт является точкой входа и вызывается при монтировании ФС. Выполняет основные действия но не сама.

**Static struct dentry *my_mount()**

```c
static struct dentry *my_mount(struct file_system_type *fs_type, int flags, const char *dev_name, void *data)
{
    struct dentry *ret;
    ret = mount_bdev(fs_type, flags, dev_name, data, my_fill_super);
    if(IS_ERROR(ret))
        printk(KERN_ERR "Error %d\n", ret);
    else
        printk (KERN_INFO "SUcces mounted %s \n", dev_name);
    return ret;
}
```

Bdev – block device

**my_file_super** указатель на функцию которая вызывается из mount_bdev для инициализации супер_блока

**my_mount** возвращает указатель на структуру dentry . Эта структура представляет полное имя файла. Но состоит она из частей. Каждое состовляющее — каталог.
/bin , /vim – разные экземпляры структуры dentry , представляющие участки пути . Первый слэш — корневой каталог. Bin/ vim

Ядро поддерживает кэш этих структур, что позволяет быстро искать inode по полному имени файла . Соответственно функция my_mount должна вернуть dentry, которая предсатвляет корневой каталог нашей ФС . И этот корневой каталог создается именно функцией my_fill_sb

**Раздел жесткого диска с ФС ext2**
ФС ЗАНИМАЕТ НА ДИСКЕ **РАЗДЕЛ** (!!!!)

ФС начинается с суперблока. Он содержит всю основную инфу,чтобы получать доступ к файлам данной ФС.

Пояснения к схеме:

Группа блоков 1 начинается с **резервной** копии суперблока.

Суперблок **имеет** заголовок, версия. Число монтирований. Размер блока. Число свободных блоков и inode , первый inode , struct dentry

В системе два типа айнодов (на диске и в ядре)
у нас тут дисковый inode . В этой структуре имеются атрибуты файла (к атрибутам относятся типы файлов (7), права доступа, владелец, размер, время) и адреса физических блоков (адреса блоков, в которых хранятся данные, записанные в файл)

ФС ext2 поддерживает большие файлы. Инфа хранится вразброс . Надо чтобы у нас хранились эти физ адреса.

В прямой адресации участвует 11 блоков.
В косвенной участвует 12-ый блок
В двойной косвенной 13-ый
В тройной косвенной 14-ый

Дисковый inode ext1 содержит 15 блоков указателей. Прямая адресация. Каждый содержит адрес блока данных.
12-ый блок содержит адрес блока размером 4к и это 1024 указателя на блоки данных.
13-ый блок содержит 1024, где каждый содержит еще 1024, и из них уже по одному.
Тройная косвенная аналогично.

* Косвенная увеличивает время доступа

Супер блок содержит информацию необходимую для монтирования и управления ФС.  В каждой ФС ОДИН суперблок и располагается он в начале

раздела. Но также имеется резервная копия суперблока для обеспечения надежности работы ФС.

Суперблок считывается в память ядра при монитровании ФС и находится там до ее демонтирования.

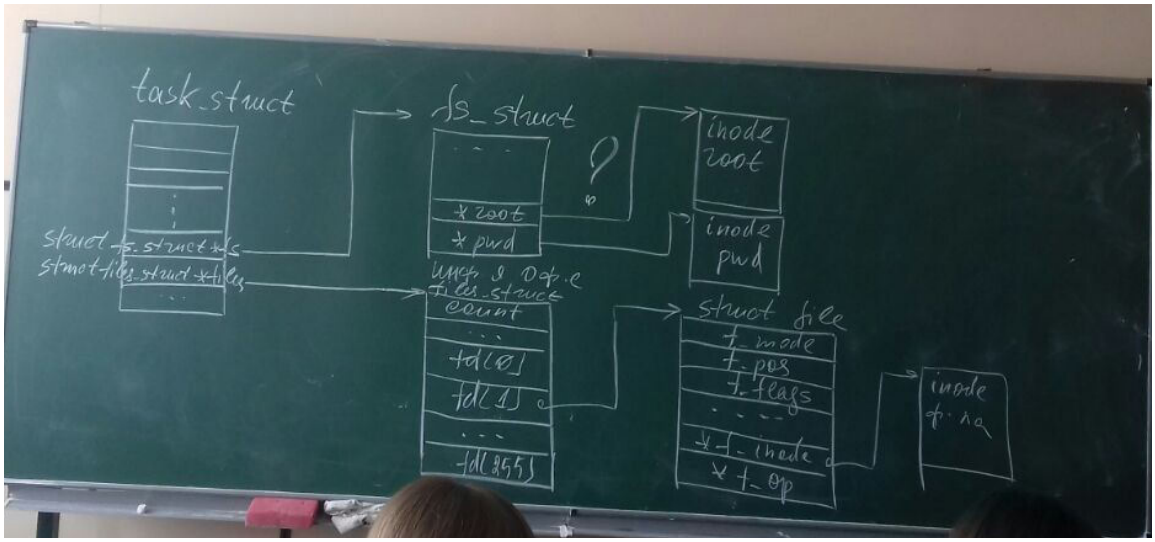
**Битовая карта занятости айнодов.**

Ext2 следующим образом определяет супер операции
**static struct super_operations ext2_sops**

```
static struct super_operations ext2_sops =
{
    .write_inode = ext2_write_inode,
    .put_super = ext2_put_super,
    .write_super = ext2_write_super,

};
```

**<u>Семинар 20-03</u>**

Сегодня должны рассмотреть информацию об открытых файлах




Важнеший момент при создании ФС
Основная структура file-system_type и superblock
file_system_type описывает новую ФС

```
static struct file_system_type lfs_type = {
    .owner          = THIS_MODULE,
    .name           = "lwnfs",
    .get_sb         = lfs_get_super,
    .kill_sb        = kill_litter_super,
};
```

**static struct file_system_type lfs_type**

Выше описанная структура нужна для того, чтобы показать разницу в версиях
Здесь вметое **mount** идет **get_sb**
Основная фукнция my_mount – mount_bdev

Создание ФС в ядре (важный файлик в инете)
ФС называется **aufs**
**static struct dentry *aufs_mount()**

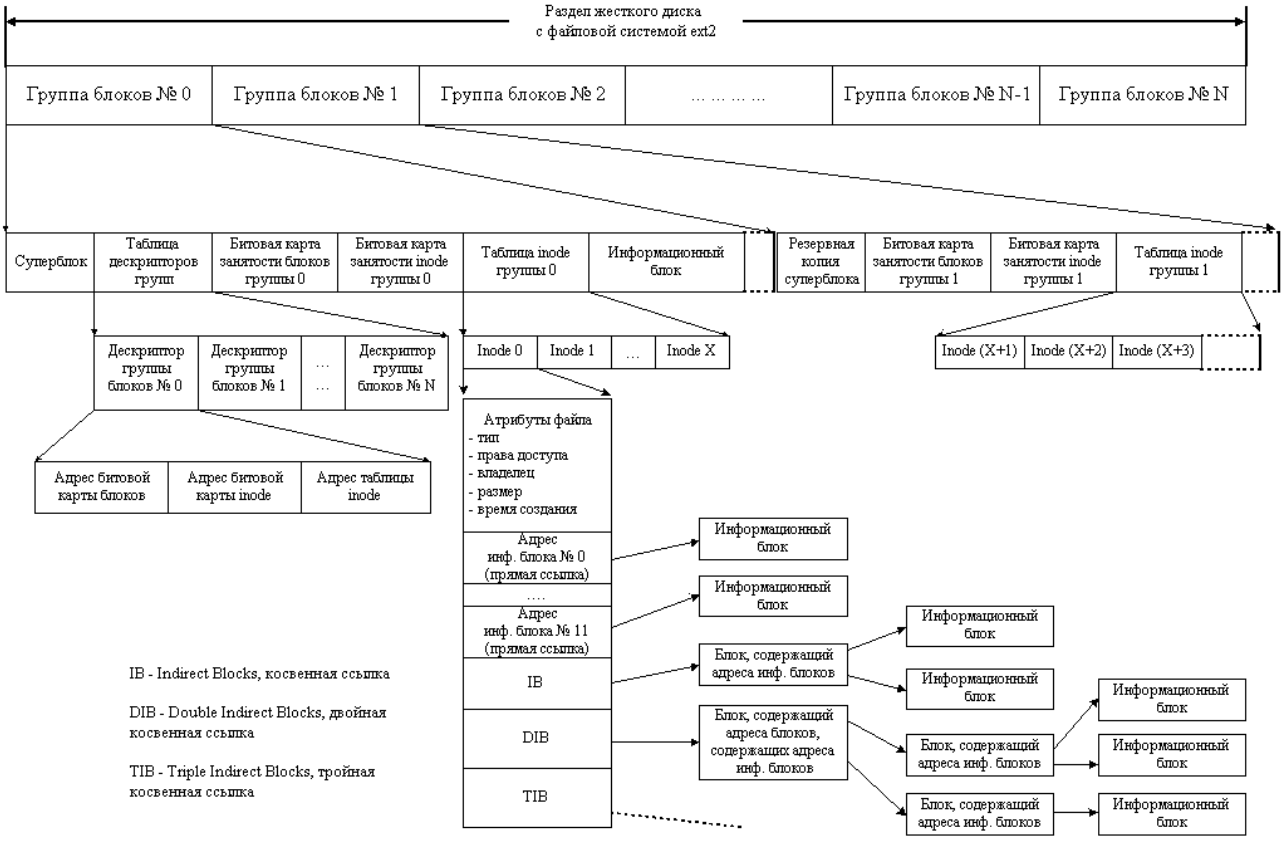
```
static struct dentry *aufs_mount(struct file_system_type *type, int flags,
                                 char const *dev, void *data)
{
    struct dentry *const entry = mount_bdev(type, flags, dev,
                                            data, aufs_fill_sb);
    if (IS_ERR(entry))
        pr_err("aufs mounting failed\n");
struct dentry *mount_bdev(struct filesystem_type *fs_type, int flags, const char *dev_name,
    void *data, int(*fill_super)(struct superblock* ,void *, int))
{};
    }
```

**static dentry *mount_bdev**

Мы передаем m_bdev в фукнцию fill_super
она описана в том файе который мы рассматриваем сейчас

```c
static int aufs_fill_sb(struct super_block *sb, void *data, int silent)
{
    struct inode *root = NULL;

    sb->s_magic = AUFS_MAGIC_NUMBER;
    sb->s_op = &aufs_super_ops;

    root = new_inode(sb);
    if (!root)
    {
        pr_err("inode allocation failed\n");
        return -ENOMEM;
    }

    root->i_ino = 0;
    root->i_sb = sb;
    root->i_atime = root->i_mtime = root->i_ctime = CURRENT_TIME;
    inode_init_owner(root, NULL, S_IFDIR);

    sb->s_root = d_make_root(root);
    if (!sb->s_root)
    {
        pr_err("root creation failed\n");
        return -ENOMEM;
    }

    return 0;
}
```

Понятно что суперблок явл основной. Она содержит метаданные о системе и тд…
Суперблок:




Из всех операций на суперблоке определена операция **put_super**
Также здесь приводится makefile ( в статье) , вызов ins_mode , ls_mode

```
sudo insmod ./aufs.ko
lsmod | grep aufs
sudo rmmod aufs
```

В конце написано нужно создать кталог
После этого нужно примонтировать свою ФС sudo mount и тд
Чтобы размонтировать sudo umount

```
mkdir dir
sudo mount -o loop -t aufs ./image ./dir
sudo umount ./dir
```

Исходники лежат на гите
У КОГО НЕ ПОЛУЧАЕТСЯ С ВФС , ТОТ СОЗДАЕТ ПО ГИТХАБУ

Давайте найдем ранее рассмотренные структуры : file_system_type , super_block, inode, inode_operations, file_operations

Смотрим файл
struct file_operations
Определяем три фукнции open read write
**static ssize_t lfs_read_file**

```
static ssize_t lfs_read_file(struct file *filp, char *buf,
                             size_t count, loff_t *offset)
{

    atomic_t *counter = (atomic_t *) filp->private_data;
    int v, len;
    char tmp[TMPSIZE];

    v = atomic_read(counter);
    if (*offset > 0)
        v -= 1;
    else
        atomic_inc(counter);
    len = snprintf(tmp, TMPSIZE, "%d\n", v);
    if (*offset > len)
        return 0;
    if (count > len - *offset)
        count = len - *offset;
    if (copy_to_user(buf, tmp + *offset, count))
        return -EFAULT;
    *offset += count;
    return count;

}
```

обращаем внимание на функции copy_to_user и copy_from_user (в write)

На что нужно обратить внимание в файле:
определены ли здесь основные структуры ?
super_block, inode, dentry, file

```
 static struct inode *lfs_make_inode(struct super_block
*sb, int mode) {
        struct inode *ret = new_inode(sb);

        if (ret) {
         ret->i_mode = mode;
         ret->i_uid = ret->i_gid = 0;
         ret->i_blksize = PAGE_CACHE_SIZE;
         ret->i_blocks = 0;
         ret->i_atime = ret->i_mtime = ret->i_ctime =
CURRENT_TIME;
        }
        return ret;
 }
```

**static struct inode *lfs_make_inode**
Анализируя read write здесь реализовано взаимодействие с пользовательской программой , то есть взаимодействие с уровнем пользователя

Дальше видим фукнцию create_file – указатель на struct dentry
create_dir
crete_files

В первом случае передается root , создаетя директория
затем создается поддиректория

fill_super

и вместое get_super используем mount

Дальше идет register_file_system, unregister_file_system

Вся наша рабта — изучить структуры и функции (и видимо собрать все эти куски работать )
Убираем GOTO, код надо структурировать и привести его к современной версии ядра

Есть бессмертное творение чувака, где он пишет что вот  я запустил код и конкретно то-то не работает

## ОТКРЫТЫЕ ФАЙЛЫ

Аналогичной разницы между программой и процессом  . Процесс — это абстрация системы, означает выполняемую программу, она частично агружена в оперативную память и частично на диске, даны ресуры. В общем две большие разницы. Программа — файл (ни в коем случае не машинный код). Можем запускать только исполняемый. Также сущ разница между просто файлом и открытым файлом.
Открытый связью с файлом . Мы создаем \ открываем связь с файлом.
Чтобы прочесть или записать в файл, надо его открыть.

Создание и открытие файла функцией **open**
Библиотека **fcntl**

Функция open – третий аргумент не обязательный и появляется только если установлен flag create.
Например
**int open(const char *pathname, int oflag, ...)**

```c
#include <fcntl.h>
// ISO C - указывает, что остальные аргументы и их типы могут варьироваться
int open(const char *pathname, int oflag, ...)

fd = open("myfile", ORDWR | O_CREAT  | O_EXCL, 0644);
```

Второй аргумент формируется одной или несколькими константами

excl – exec control – используется для проверки правильности использования константы create
если файл уже существует, то возникнет ошибка
кроме того комбинация этих флагов выполняет проверку существования и его создания атомарно (сущ монопольный доступ к соотв структурам )

системный вызов open  возвращает файловый дескриптор — целое число . Этот дескриптор применяется как идентификатор файла и исплоьуется затем в системных вызовах read \ write . Получить файловый дескриптор можно с помощью функции **fileno** из библиотеки stdin

#include <stdin.h>
int fileno(FILE *fp);

Открыть файл может процесс, поэтому в структуре процесса есть указатели на соотв структуры, позволяющими процессам работать с открытыми файлами

в линуксе есть структура **task_struct**  и отдельный блок информаци
//**file_system_info**

**//file_system information**
**struct fs_srtuct *fs;**

**//open file information**
**struct files_struct *files;**

**//здесь же инфомрация об адресном пространстве процесса — защищенное и виртуальное**
**(КАМЕНЬ ПРЕТКНОВЕНИЯ)**
**//memory manager info**
**struct mm_struct *mm;**

Хорошие статьи есть у Тима Джонса.

Процесс оперирует 3 структурами : files_struct, fs_struct и namespace (??)

Рассмотрим **struct fs_struct , files_struct**

```
struct fs_struct
{
    int users;
    //...
    int umask;
    int in_exec;
    struct path root;
    struct path pwd;
};
```

```
struct files_struct
{
    //счечтик использования структуры
    atomic_t count;
    // written part on a separate cache line SMP
    //smp архитектура? - симметричная мультипроцессорная

}
```

**struct path**
{
struct vfs_mount *mnt;
struct dentry *dentry;
}
**TODO** дописать структуру files_struct + обсудить ее и связать между собой структуры ядра открытых файлов ( и не только ядра)

Семинар 03-04

Система должна предоставлять процессу инфомрацию о ФС к которой он относится.
Что значит процесс относится к ФС? экзешник — файл. То где он создан

Процесс описывается структурой task_struct (рис из начала прошлого семенара).
files_struct - информация о файлах, открытых процессом.
**Struct files_struct**

```
struct files_struct
{
    count;
    close_tds_init_1;
    fd[0];
    fd[1];
    fd[2]
    //...
    fd[255];
}
```

```
struct files_struct
{
    atomic_t count;
    bool resize_in_progress;
    wait_queue_head_t resize_wait;
    struct fdtable fdtab;
    //fdtable:
    //1-ое поле - макс количество файловых объектов
    //через поле файловые дескрипторы , которые должны закрываться при вызове exec
    //след поле -указатели на дескрипторы открытых файлов
}
```

СМП — симметричная мультипроцессорная архитектура
(равноправные процессоры, обращающиеся к общей памяти )

В системе имеется системная таблица открытых файлов.

```
struct files_struct
{
    atomic_t count;
    bool resize_in_progress;
    wait_queue_head_t resize_wait;
    struct fdtable fdtab;
    //fdtable:
    //1-ое поле - макс количество файловых объектов
    //через поле файловые дескрипторы , которые должны закрываться при вызове exec
    //след поле -указатели на дескрипторы открытых файлов
    struct file_rcu *fd_array [NR_OPEN_DEFAULT];
}

struct file
{
    f_mode;//права доступа к файлу
    f_pos; //текущая позиция файла
    f_flags;//
    f_count;
    f_owner;
    *f_inode;//ЗДЕСЬ ЯВНО УКАЗАТЕЛЬ НА АЙНОД
    *f_op;
}
```

**Struct file, files_struct**

ФС Proc – ВФС. Не является монтируемой ФС. Находится только в памяти. Создается когда линукс стартует и удаляется когда компьютер выключается. Фактически представляет собой API ядра, позволяющий приложениям читать и изменять данные в адресном пространстве других процессов. Управлять процессами , получать инфомрацию о процессах и ресурсах , которые используются процессами . При этом инф-ция предоставляется в формате ФС для простого доступа.

Total : Простой интерфейс для доступа

По умолчанию запись и чтение файлов в ФС proc разрешены только для владельцев. Данные о каждом процессе хранятся в поддиректории **/proc/<PID>**
Чтобы процесс мог получить данные о себе, надо сначала взять айди — фукнция **getpid().**
Имя поддиректории это числовое значение идентификатора процесса ( НЕ ГОВОРИТЬ ЦИФРА, ОНЛИ ЧИСЛОВОЕ ЗНАЧЕНИЕ!!!!!!!!!!!!)
В поддиректории процесса находятся файлы и поддиректории, содержащие данные о процессе.

| **Элемент** | **Тип** | **Содержание** |
|---|---|---|
| Cmdline | Файл | Указатель на директорию процесса |
| Cwd | Символьная ссылка | Указатель на |
| Environ | файла | Список окружающих процессов |
| exe | Символьная ссылка | Указатель на образ процесса (на его файл) |
| fd | директ | Ссылки на файлы , используемые процессом ( файлы, открытые проц) |
| root | Симв ссылка | Указ на корень файловой системы проц |
| start | файл | Информация о проц |

/proc/self

Информация об окружении процесса . Программа режима пользователя. Строки в файле envron разделены не символом новой строки (код 10 или A) а нулями. Поэтому чтобы создать удобочитаемый формат, здесь добавляется перевод строки.

```c
#include <stdio.h>
#define BUF_SIZE 0x100

int main(int argc, char *argv[])
{
    char buf[BUF_SIZE];
    int len, i;
    FILE *f;
    f  = fopen("/proc/self/environ", "r");
    while((len = fread(buf, 1, BUF_SIZE - 1,f)) > 0)
    {
        for(i = 0; i < len; i++)
            if(buf[i] == 0)
                buf[i] = 10;
        buf[len] = 0;
        printf("%s", buf);

    }
    fclose(f);
    return 0;
}
```

**/proc/pci** – о подключенных устройствах
**/proc/apm** – температура ноута

Модули используют структуру proc_dir_entry.
**Struct proc_dir_entry**

```
struct proc_dir_entry
{
    unsigned int low_ino;
    unsigned short namelen;
    const char *name;
    mode_t mode;    // права доступа
    nlink_t nlink;
    uid_t uid;
    gid_t gid;
    const struct inode_operations *proc_iops;
    const struct file_operations *proc_fops;
    struct proc_dir_entry *next, *parent, *subdir;
    void *data;// указатель на локальные данные
    //в этой стр определены две операции : read_proc, write_proc
    read_proc__t *read_proc;
    write_proc-t *write_proc;
    //...

};
```

proc_create
proc_create_data

```
#define proc_create(name, mode, parent, proc_fops)({NULL;})
#define proc_create_data(name, mode, parent, data)({NULL})
```

**Пример**

```c
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/asm/uaccess.h>
#include <asm/types.h>

struct proc_dir_entry *proc;
int len, temp;
char *msg;
int read_proc(struct file *filp, char *buf, size_t count, lofft_p *offp)
{
    char *data;
    data = PDE_DATA(file_inode(filp));
    if(!(data))
    {
        printk(KERN_INFO "Null data");
        return 0;
    }
    if(count > temp)
        count = temp;
    temp = temp - count;
    copy_to_user(buf, fata, count);
    if(count == 0)
        tem = len;
}
```

struct file_operations proc_frops

```c
struct file_operations proc_fops =
{
    read : read_proc;
}
```

Должны закончить на create_new_proc_entry

**Семинар 17-04**

Организация ввода вывода
На лекциях 5 моделей — это программная реализация. Ввод вывод с точк зреня программиста.
Сейчас с точки зрения аппаратуры.

Возможно три способа выполнения ввода вывода
1) программируемый ввод \ вывод — опрос == poling
2) ввод\ вывод с использованием прерываний
3) прямой доступ к памяти (ПДП) — direct memory access – DMA

**Программируемый ввод \ вывод**
связан с опросом. Когда приложение запрашивает ввод\вывод, система выполняет, например, библиотечную команду. В результате с ТЗ аппаратуры контроллер получает команду по ШИНЕ ДАННЫХ , выполняя которую он управляет внешним устройством. В тексте библ функции находится системный вызов,а в его тексте находится прерывание — для виндоус это INTE2 – для линукс INT 8H который перевод систему из режима задачи в режим ядра. В режиме ядра параметры запроса ввода \ вывода сохраняются и менеджер ввода вывода передает их драйверу устройства. Драйвер устройства на оснвоании полученных данных формирует команду в формате понятном устройству . Формат ? - например передается слово , в этом слове каждые биты за что-то отвечает — это и есть формат. В наших системах устройствами управляют функциональные драйверы — драйверы нижнего уровня. Фукнциональные драйверы, как правило, пишут разработчики устройства. Только они знают формат данных , понятные устройству. Только они знают какие сигналы должны получать устройства. Такие драйверы поставляются вместе с устройством или по соглашению в системе есть набор драйверов на выбор(как в винде при подключении у-ва).
Есть иерархия драйверов. В винде есть **стек драйверов.**

После того как команда поступила в регистры контрллера устрйства, процессор начинает опрашивать биты состояния устройства ввода \ вывода с целью определения момента завершения операции ввода \ вывода. То есть прцоессор все время занят организацией ввода вывода, при этом необходимо чтобы в системе были все ужные команды : команда управления, которая инициализирует работу внешнего устройства, команды анализа состояния внешнего устройства, команды передачи информации ( read write).
Такой способ возможен , но он неэффективен, так как процессорное время используется неэффективно.

**Ввод \ вывод с прерываниями**
При вводе выводе с прерываниями процессор передает контроллеру команду ввода вывода ПО ШИНЕ ДАННЫХ. После этого переходит на выполнение другой работы. Контроллер выполняет управление операцией ввода вывода и когда она завершает формирует сигнал который поступает на контроллер прерывания ( в совр системах это не так).

(мы рассматривали трехшинную архитектуру )
Когд аустройство завершает управление оно посылает сигнал прерывания на контроллер прерывания и он формирует сигнал прерывания и посылает его на ножку процессора по ишне упрвления. В конце каждой команды процессор проверяет сигнал прерывания на кадждой выделенной ножке. Когда поступает сигнал прерывания то вызывается обработчик прерывания который должен передать считать данные из регистров контроллера в буфер ядра и затем уже драйвер устройства должен передать например считанные данные в буфер

приложения, при этом мы знаем (2 часть по таймеру) устройство блокированое на вводе выоде получает приращение приоритета. В винде процесс блокрванный на мыши и клавиатуре получает приращение сразу 6. Процесс блокированный на звуковой карте сразу 8. В результате такие процессы смогут вытесниить процессы которые в текущий момент времени выполняются, то есть поулчили квант. Необходимо как можно быстрее передать данные программе которая запросила их. Если программа запросила не ввод а вывод она все равно получает информацю о том что вывод завершился успешно. В программе можно вызвать МАКРОСЫ которыые помогут анализировать ситуацию ввода вывода.

Рассмотрим с ТЗ аппаратуры:
**с ТЗ контроллера**

В  начале контроллер получает по ШД команду например read – считать и выполняя эту команду переходит к считыванию данных из связанного с ним перифирийного устройства. Как только данные поступают в регистры контроллера, контроллер формирует сигнал который поступает на контроллер прерывания и ждет когда процессор запросит данные. Когда процессор запрашивает данные контроллер передает данные по ШД. После чего переходит в состояние готовности к выполнению следующих операций ввода вывода.

С ТЗ процессора процессор выполняя библиотечную функцию делает то что уже было сказано. В итоге система переходит в режим ядра. Параметры команыд должны быть сохранены, программа запросившая ввод вывод должна быть заблокирована. То есть процесс переводится в состояние блокировки, при этом сохраняется как минимум аппаратный контекст. Затем вызывается драйвер , в драйвере вызывается соответсвующая фукнция , данные переводятся в формат понятный устройству и передается по ШД в регистр контроллера устройства. После этого процессор переходит на выполнение других каких то действий , то есть он отключается от управления операцией ввода \ вывода. В конце цикла выполнения каждой команды процессор проверяет наличие сигнала прерывания, в результате выполняющаяся программа прерывается , то есть сохраняется ее аппаратный контекст, процессор переходит на выполнение обработчика прерывания.

Задача обработчика прерывания считать данные из регистра контроллера и поместить их в буфер ядра. После того как завершается выполнение обработчика прерывания, разблокируется программа выдавшая запрос ввода вывода и она может продолжить свою работу получив необходимые данные. Ввод вывод с прерывания значительнее эффективнее опроса, но все еще требует много процессорного времени , так как каждое слово которое передается из памяти на устройство или из устройства в память проходит через регистры процессора. Это можно считать недостатком прерываний. (прерывание это зло, но зло неизбежное).

Это все еще затратный способ.

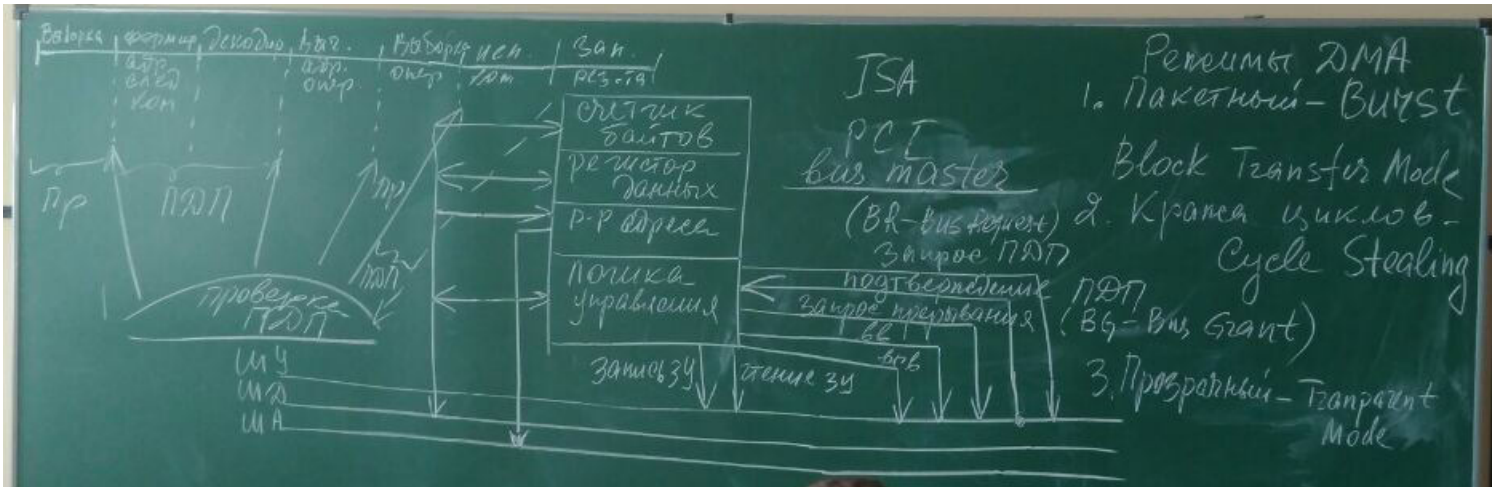
Рассмотрим принтер. Процесс запросивший печать строки на принтере блокируется на все время печати строки. Когда принтер напечатает символ и готов к приему следующего символа он формирует прерывание . Это прерывание прервет выполнение текущего процесса, при этом будет сохранен контекст этого процесса. После того как процессор передаст следующий символ работа прерванного процесса может быть продолжена. То есть на каждом символе прерывание или сохранение контекста. Огромное количество действий!

**<u>Прямой доступ к памяти (ПДП)</u>**
Процессор непосредственно управляет вводом выводом с прерыванием предоставляя свои регистры для передачи данных, прии этом для передачи байта или слова необходимо выполнить некоторое количество команд

DMA – метод доступа внешних устройств к памяти, в котором передача данных осуществляется не центральным процессором, а специальным устройством, которое называется контроллером DMA(ПДП).

Быстрые режимы DMA – ультра DMA.

Контроллер DMA:




(тут только схема нужна)

Контроллер получает от процессора начальный адрес куда записать и количество байтов ( размер блока ).
Через DMA в основном работают жесткие диски.
Контроллер ДМА выставляет на шину адреса — адрес.
Процессор адресует все устройства в системе. Существует два адресных пространста — АП оперативной памяти и АП портов ввода вывода (in \ out).
Внешние устройства также адресуются.

В сигналах управления есть запрос прерывания.
Запрос ПДП — BusRequest – BR
BusGrant – BG

Прерывание все равно возникает , но после того как передан блок данных. Не на каждом символе, а при передаче блока.

Процесс пересылки каждого байта состоит из двух этапов:
мин адресуемая единица памяти - байт

1) Контроллер ПДП поставляет на ША адрес байта ОП.
На ШУ доставляется сигнал readMem . После этого контроллер по ШД считывает из памяти данные и помещает их в свой регистр.

2) на шину адреса выставляется адрес устройства . На ШУ сигнал вывод.И передает инфу из своего регистра данных в регистр данных контроллера устройства.

**Режимы DMA:**
1) Пакетный — Burst Block Tansfer Mode
Получив доступ к системной шине контроллер DMA передает непрерывно байт за байтом весь блок данных , на это время процессор отключается от системной шины. (не может ниче делать, так как он постоянно обращается к ОП) . Не оч хороший способ

2) Кража циклов — Cycle Stealing
Режим используется если процессор не может блокироваться на длительное время. Для доступа к системной шине используются те же сигналы что и в пакетном режиме BR и BG
После сигнала BusRequst передается один байт и контроль над системной шиной возвращается процессору сигналом BusGrant
Чередуется процесс передачи данных под упр контроллером ДМА и работой процессора - выполнения команд .

3) Прозрачный — Transparent Mode
Контрллер ДМА передает данные в те моменты когда процессор не использует системную шину . В результате система вообще не замедляет свою работу. Но данный режим требует чтобы аппаратно определялись моменты когда процессор не использует системную шину, что усложняет аппаратуру.
Достоинством является то что в отличие от прерваний в пределах цикла выполнения команды имеется несколько точек, где контроллер ДМА может захватить шину

(тут левая схема из картинки выше)

Достоинством ПДП является то что ПДП **не требует переключение контекста**

**ISA PCI**

bus master

ISA - industry standart architecture

Это шина использовалась в старых системах с одним Мб памяти . Был один контроллер ДМА. Это медленная шина, но до сих пор поддерживается. Это связано с запросами американской промышленности. Которая продолжает использовать старые интерфейсы.

Более быстрый способ — шина PCI (express). Сейчас осущ переход на хабовую архитектуру — процессор Intel i8.

В архитектуре PCI – нет центрального контрллера DMA. Любой компонент PCI может запросить управление шиной — BusMaster. Если несоклько устройств одновременно запрашивают контроль над шиной , то орбитр южного моста

южный мост - функциональный контроллер, также известен как контроллер-концентратор ввода-вывода

северный мост - контроллер (чип), являющийся одним из элементов чипсета материнской (системной) платы и отвечающий за работу центрального процессора (CPU) с ОЗУ (оперативной памятью, RAM) и видеоадаптером.

использовалась в большинстве систем.

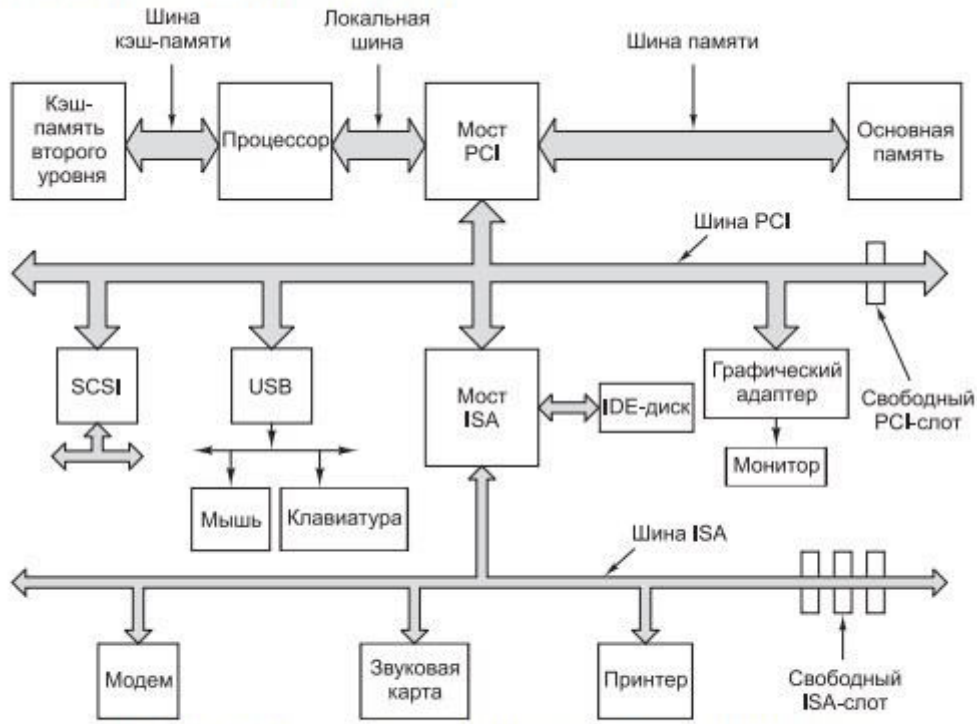



**Рис. 3.48.** Архитектура типичной системы первых поколений Pentium. (Толщина линий шины обозначает ее пропускную способность. Чем толще линия, тем выше пропускная способность)

Если несколько устройств запршивают доступ, то орбитр южного выбирает мастера шины. Устройство успешно осуществившее захват шины самостоятельно выставляет сигналы адреса и управления и управляет передачей данных по шине. Некоторые страые усройства PCI например звуковвые карты Sound Bluster используют старый контроллер DMA для шин ISA.
Очевдно что такой подход явл устаревшим , но поддерживается для полной совместимости ПО.С драйверами версии Sound Bluster для шины ISA. Такая поддержка наз Distributed DMA .

**PCI DMA Обмен**
Диски IDE (integrate drive electronics) с поддержкой DMA. Имеют встроенный интерфейс накопителей и PCI DMA обмен начинается по инициативе IDE контроллера.

**DMA и виртуальная память —** проблема
Для решения этой проблемы требуется выявление физических страниц на которые отображается линейное адресное пространство процесса к котормоу мы обратились, ведь смысл перекачки данных связан или с файлом к которому обр процесс или со сичтыванием данных или с процессом. Для того чтобы решить эту проблему используется так называемый **scatter – gather-list SGL.**

**ДРАЙВЕРЫ**
имеют много точек входа. Также как и обработчик аппаратного прерывания. В системе в любой сущ несколько уровней драйверов. Драйвера которые загружаются при загрузке системы это драйвера нижнего уровня . Виндоус прямо декларирует стек драйверов. На нижнем уровне находится фукнциональный драйвер. Такое название встр не часто. Встр название дарйвер нижнего уровня. Если про стек, то есть функциональный, драйвер фильтр верхнего и нижнего уровня. В Линукс есть драйвера нижнего уровня. Есть возмонжость загрузить при работе системы свой драйвер в виде загружаемого модуля ядра plug and play и с помощью такого модуля драйвера изменить функцкию устройства. Функциональные драйверы пишут разработчики устройства. Только они знают досконально форматы передаваемых данных. В виндоус есть мастер установки оборудования. (принтер например). В системе есть уже драйверы разных производителей. Если нет подходящеого то ищем в инете. Фильтр верхнего и нижнего уровня это драйвер который загружается как plug and play регистрируется в реестре и в результате можно изменить функицональность устройства. В Линукс это загружаемые модули ядра.

**Линукс**
Существует низкоуровневая структура struct device

struct device

```
struct device
{
    //предок (шина или хост контроллер)
    struct device *parent;
    //первоначальное имя устройства
    const char *initname;
    //тип устройства
    const struct device_type *type;
    //может исп для взаимоисключения при вызове устройств
    struct mutex mutex;
    //тип шины
    struct bus_type *bus;
    //описывает драйвер устройства
    struct device_driver *driver;
}

//msi - message signal interrupt
#ifdef CONFIG_GENERIC_MSI_IRQ_DOMAIN
struct irq_domain *msi_domain;
#endif
```

irq_domain
**struct device**

//dma
direct memory access
заинтересованы устройства в прямом доступе к памяти
struct device_dma_parameters ***dma_parms;**

**Параметры дма**
структура низкого уровня представляет устройство в модели устройств линукс. С этой структурой драйверы не часто работают напрямую. И эта структура нужна особенно когда используется основной уровень дма. Обычно можно найти эту структуру закопанной (похоренной) внутри специфической шины, которая опиывает устройство. Например, эту шину можно найти как поле dev в struct pci_device или в struct usb_device.
**Struct device_driver**
Рассмотрим struct **device_driver *driver** – базовая структура драйвера.

```c
struct device_driver
{
    //имя
    const char *name;
    //шина, к которой подключено устройство
    struct bus_type *bus;
    //исп для встроенных модулей
    struct module *owner;
    const char *mod_name;
    int  (*probe)(struct device *dev);
    int  (*remove)(struct device *dev);
    void (*shutdown)(struct device *dev);
    int  (*suspend)(struct device *dev, pm_message_t state);
    int  (*resume)(struct device *dev);
    //...
};
```

**Probe** вызывается для запроса существования конкретного устройства если этот драйвер может работать с ним и связывает драйвер с конкретным устройством. Вызывается когда устанавливается устройство . Вызыввается функция probe и в результте этого вызова должно быть установлено имеет ли драйвер к этому устройству или нет.

**Suspend** - приостановка . Переводит устройство в состояние сна. В режиме низкого энергопотребления
**Resume –** вывод из сна.

**USB драйверы**
архитектура совр ЭВМ в татенбауме (power pc с шиной pci)

должны ЗАПОЛНЯТЬ
Структура содержит call back фукнции — явл точками входа в драйвер, а также параметры для работы с usb ядром — usb core.

```c
struct usb_device
{
    //должно быть уникальным среди драйверов и таким же как имя модуля
    const char *name;
    //вызывается чтобы узнать готов ли драйвер управлять опр интерфейсом устройства
    //еси готов, то возвр 0, иначе ENODEV
    //елси возникли ошибки ввода вывода, то посылается в errno
    //usb driver связывется с интерфейсом
    int (*probe)(struct usb_interface *intf, const struct usb_device_id *id);
    //отключить отсоединить вместо удалить (remove)
    //вызывается если interface больше недоступен,
    // обычно потмоу что утсрйоство было отключено или моудль драйвера был выгружен
    void (*disconnect)(struct usb_inerface *intf);
    //ctl - control, исп в драйверах которые взаимодействуют с пр-вом пользователя через ФС usbfs(!)
    int (*unlocked_ioctl)(struct usb_interfacr *intf, unsigned  int code, void *buf);
    int (*suspend)(struct usb_interface *intf, pm_message_t  message);
    int (*resume)(struct usb_interface *intf);
    //структура поддерживает список различных типов usb устройств, которые поддерживает конкретный драйвер
    //этот список используется usb ядром (core)чтобы решить какой драйвер соотв устройству и по горячим скриптам
    //решить какой драйвер автоматически загрузить, когда конкртеное устройство подключено к системе
    const struct usb_device_id *id_table;
}
```

**Struct usb_device**

Если в драйвере отсутствует id_table то функция обратного вызова callback probe не будет вызвана. Таблица исп для поддержки hot plugging

Таблица экспортируется макросом **MODULE_DEVICE_TABLE**(usb, …)

Структура usb_device_id содержит набор таких макросов :
USB_DEVICE (vendor, product)
USB_DEVICE_VER (vendor, product, lo, hi)
USB_DEVICE_INFO(class, subclass, protocol)
USB_INTERFACE_INFO(class, subclass, protocol)

Если разработчик хочет чтобы его драйвер вызывался для любого usb устройства то надо создать структуру котоаря устанавливает поле driver_info

static struct usb_device_id **usb_ids** [] =
{
	{driver_info = 42},
	{} // обязательно
}
static struct usb_device_id **usb_mouse_id_table [**] =
{
	{USB_INTERFACE_INFO(	USB_INTERFACE_CLASS_HID,
				USB_INTERFACE_SUBCLASS_BOOT,
				USB_INTERFACE_PROTOCOL_MOUSE)},
	{}
}

**Регистрация usb драйвера**
Заполнить поля структуры usb_driver

**static struct usb_driver pen_driver**

```
static struct usb_driver pen_driver =
{
    .name = "pen_driver";
    .id_table = pen_table;
    .probe = pen_probe;
    .disconnect = pen_disconnect;
}
```

Все функции должны быть описаны заранее (ясен пончик).
Элементы id_table находятся в библиотеке <linux/usb.h>
Обычно исп макрос USB_DEVICE(vendor – производитель, product – номер серии)

**static struct usb_device_id pen_table**

```
static struct usb_device_id pen_table [] =
{
    //для какого-то конкретного transcend
    {USB_DEVICE(0x058f, 0x6387)},
    {}
}
//для экспорта структуры
MODULE_DEVICE_TABLE(usb, pen_table);
```

Для регистрации функция usb_register
pen_init
pen_exit

```
static int _init pen_init(void)
{
    return usb_register(&pen_driver);
};

static int _exit pen_exit(void)
{
    return usb_deregister(&pen_driver);
};
MODULE_INIT(pen_init);
MODULE_EXIT(pen_exit);
```

Устройство — файл, поэтому для описания операций исп структура struct file_operations
**static int pen_probe**
**static void pen_disconnect**

```c
static int pen_probe(struct usb_interface *interface, const struct usb_device_id *id)
{
    printk(KERN_INFO "Pen Driver(%04x : %04x) plugged\n", id->idVendor, id->idProduct);
    return 0;
}

static void pen_disconnect(struct usb_interface *interface)
{
    printk(KERN_INFO "Pen Driver disconnected\n");
}
```

**Для курсача код нулевого уровня привилегий**
приложение может быть но должно взаимодействовать с кодом 0 уровня
также нельзя использовтаь report - ы
usb_interface_info :
посл слово HID – human interface
этот класс драйверов (кроссплатформенный) это драйвера которые написаны для устройства обеспеивающих взаимодействие пользователя с ПК

report – структура которую устройство передает (драйвером выполняется) hid ядру