15.03.2019

в лабораторной работе №3 по директориям вызываем функцию stat
нужно детализировать ошибки возвращаемые
с помощью свич-кейса

!!!константы записывать большими буквами!!!
ошибка начинается с E (Error)

возвращаемые значения:
0 - успех
-1 - при ошибке и в errno записывает код ошибки
- EACCES
- EBADF - плохой файл
- EFAULT - неверный адрес
- ELOOP - ошибка при переходе, в пути встречается слишком много символических ссылок
- enametoolong - имя файла слишком длинное
- ENOENTR - какой-то компонент пути не существует или является пустой строкой
- ENOMEM - недостаточно памяти ядра
- ENOTDIR - компонент пути не является каталогом

обрабатывать все ошибочные ситуации
при ошибке выводить ее и выходить из программы

если программа завершается аварийно системой - ВЫ ПЛОХОЙ ПРОГРАММИСТ

лабораторная: загружаемые модули ядра
1 часть:
загружаемый модуль ядра «Hello world»

2 часть: 3 загружаемых модуля

стандартный вид модуля ядра

```
#include <linux/module.h>
#include <linux.kernel.h>
#include <linux/init.h>
MODULE_LICENSE(«GPL»);  // указывается лицензия
MODULE_AUTHOR(«I am»);
MODULE_DESCRIPTION(«Lab»);
static int__init my_modyle_init()
{
   printk(KERN_INFO «Ab\n»);
   return 0;
}
static void __exit my_module_exit()
{
   printk(KERN_INFO «Exit\n»);
}
module_init(me_module_init);  // обязательны для модуля ядра
module_exit(me_module_init);  // обязательны для модуля ядра
```

данный модуль сообщает ядру, что он хочет получить доступ ко всем символам ядра

модуль ядра в какой-то момент станет частью ядра, т.е. будет выполняться в kenal

printk - пишет информацию в syslog

KERN_INFO - уровень протоколированная сообщения

module_init - вставляет наш код в код ядра
module_exit - этот код удаляет

существует 8 уровней:

Уровень	**Константа**	**Описание**
7	KERN_DEBUG	отладочное сообщение
6	KERN_INFO	информационное сообщение
5	KERN_NOTICE	еще не предупреждение, но еще не информационное сообщение
4	KERN_WARNING	предупреждение что-то может пойти не так
3	KERN_ERR	уже пошло не так
2	KERN_CRIT	критическая ошибка
1	KERN_ALERT	скоро рухнет
0	KERN_EMERG	рухнуло

на примере функции printk
библиотечные функции режима пользователя определены в посикс(стандарт)
это не касается функций ядра

если используем функции ядра, то нужно брать исходники в том ядре для которого пишется программа

версия ядра очень важна

в ядре тоже имеются коды ошибок (они определены define):
#define EPERM 1 - операция невозможна
ENOENTRY 2 - какой-то компонент пути не существует или является пустой строкой
ESRCH 3 - no … process
EINTR 4 - прерван
EIO 5 - IO error

для работы с модулями в системе есть соответствующие команды:
загрузка модуля ins_mod ?
выгрузка модуля irem_mod ?

tail используется для сокращения выводимой информации и указывается количество выводимых строк

1 часть:
вывести в syslog номер текущего процесса
current -> pid «%i» - формат вывода

написать 2 printk (hello world)

найти библиотеку которая позволит выводить информацию
понять результаты какого процесса выводим
вопрос: зачем использовать двойное подчеркивание? __

чтобы посмотреть загруженные модули ядра eles_mod ?
наши модули в начале списка

если не используем кровь приоритета сообщений
то будет использоваться приоритет DEFAULT_MESSAGE_LOGLEVEL

если приоритет меньше, чем int console_loglevel, то сообщение выводится на наш текущий терминал

если оба демона, а именно syslogd и klogd выполняются, то сообщение будет также добавляться /var/log/messages
в независимости от того выведено ли оно на консоль или нет

ядро у нас вытесняемое

makefile для простейшего модуля ядра
odj -m += module.o

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

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

-C - опция смены каталога
/lib/modules/ - каталог исходных кодов ядра
M= - опция возвращения в директорию исходных кодов модуля

2 часть:
используем код из Цилюрика

md1.c
```
#include <linux/init.h>
#include <linux/module.h>
#include "md.h"
MODULE_LICENSE("GPL");
char *md1_data = "A6";
extern char* md1_proc(void)
{
        ertirn md1_data;
}
static char* md1_local(void)
{
        return md1_data;
}
extern char* md1_noexport(void)
{
        return md1_data;
}

EXPORT_SYMBOL(md1_data);
EXPORT_SYMBOL(md1_proc);
static int __init md_init(void)
{
        printk("+ module md1 start\n");
        return 0;
}
static void __exit md_exit(void)
{
        printk("+ module md1 unloaded\n");
}

module_init(md_init);
module_exit(md_exit);
```

прочитать какая функция у плюса

md.h
```
extern char* md1_data;
extern char* md1_proc(void);
```

md2.c
```
#include <linux/init.h>
#include <linux/module.h>
#include "md.h"
MODULE_LICENSE( "GPL" );
MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
static int __init md_init( void ) {
   printk( "+ module md2 start!\n" );
   printk( "+ data string exported from md1 : %s\n", md1_data );
   printk( "+ string return md1_proc() is : %s\n", md1_proc() );
   return 0;
}
```

```
static void __exit md_exit( void ) {
   printk( "+ module md2 unloaded!\n" );
}
module_init( md_init );
module_exit( md_exit );
```

в лабораторной работе дополнительно мы должны
в md1.h в ф-ии локал
из md2.h должны это (что?) вызвать

md3.h

```
#include <linux/init.h>
#include <linux/module.h>
#include "md.h"
MODULE_LICENSE( "GPL" );
MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
static int __init md_init( void ) {
   printk( "+ module md2 start!\n" );
   printk( "+ data string exported from md1 : %s\n", md1_data );
   printk( "+ string return md1_proc() is : %s\n", md1_proc() );
   return -1;
}
static void __exit md_exit( void ) {
   printk( "+ module md3 unloaded!\n" );
}
module_init( md_init );
module_exit( md_exit );
```

выполняем загрузку с -1 и смотрим какое сообщение получаем
БУДЕТ СПРАШИВАТЬ КАКОЕ!!!

имеет ли значение порядок загрузки модулей
и почему

пункт 2 в книге
модуль md2 использующий экспортируемое имя связан с этим именем по прямому адресу
поэтому если сначала загрузим md2 то мы не сможем получить этот адрес
поэтому надо сначала загрузить md1 все свяжется и можно будет загружать md3

ПУНКТ 3 НАДО ПРОЧИТАТЬ И ПРОДЕМОНСТРИРОВАТЬ В ЛАБЕ
ПУНКТ 5 ПРОЧИТАТЬ
ПРОЧИТАТЬ ПРО ЮНИКОД

23.03.19

виртуальная файловая система proc создана самими разработчиками linux

информацию получаем в виде файлов

тк это файлы, то для них устанавливаются права доступа как для файлов - читать писать исполнять

в файловой системе proc существуют поддиректории на каждый процесс /proc/<pid>

о процессе
система предоставляет информацию и доступ к этой информации обеспечивается соответствующими поддиректориями
все сведена в таблице

элемент	тип	описание
cmdline	файл	указывает на директорию процесса
cwd	символическая ссылка	указывает на директорию процесса
environ	файл	содержит список окружения процесса
exe	символическая ссылка	указывает (содержит путь) на образ процесса
fd	директория	директория, которая содержит ссылки на файлы открытые процессом
root	символическая ссылка	указывает (содержит путь) на корень файловой системы процесса
stat	файл	содержит информацию о процессе

образ процесса - файл
пока программу не запустили
она лежит во вторичной памяти и является файлом
после того как запустили она становится процессом
но продолжает находиться во вторичной памяти

любой файл в системе принадлежит какой-то файловой системе

родной файловой системой unix/linux является ext

пример
запускаем программу с флешки
флешка имеет свою файловую систему
и запущенный процесс имеет файловую систему флешки

как процесс может получить свой идентификатор? - getpid() /proc/self
процесса предка? - getppid()

пример ()

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

int main(int argc, char *argv[])
{
	char buf[BUF_SIZE];
	int l, i;
	FILE *f;
	f = fopen("/proc/self/environ", "r");

	while ((len = fread(buf, 1, BUF_SIZE, f)) > 0)
	{
		for (int i = 0; i < len; i++)
			if (buf[i] == 0)
				buf[i] = 10;  // строки разделены нулям, а нам надо осуществить
переход на новую строку и мы добавляем код 10 (0x0A)
			buf[len] = 0;
		print("%s", buf);
	}
	fclose(f);
	return 0;
}
```

почему устройство тоже файл? - для того же чтобы с устройствами работать как с файлами, чтобы не размножать систему, а сводить все к одному и тому же

если система предоставляет информацию о процессах
что в системе неразрывно связано с процессами? - ресурсы
proc позволяет получать информацию о ресурсах в системе

в системе все ресурс)
возможности которые предоставляет система - тоже ресурсы

какую информацию мы можем получить используя возможности файловой системы proc
/proc/pci об устройствах, подключенных к шине pci
шина pci - устаревшая шина, но система поддерживает совместимость снизу-вверх и в частности система поддерживает интерфейс шины pci

информация о текуцем cpu /proc/cpuinfo
информация о загруженных, работающих драйверов устройств /proc/devices
информация об источниках прерываний и частоте возникновений этих прерываний /proc/interrupts
получить информацию о состоянии батареи ноутбука /proc/apm

iocte() - посмотреть что это

struct proc_dir_entry // хз что это // надо загуглить
{…}

на версии ядра 4.10

ядро линукс стало полностью вытесняемым если выполняется код ядра, его выполнение может быть прервано

архитектура SMP - многопроцессорная система в которой все процесса симметричные (равноправные) работают с общей памятью

прочитать рихтера - windows для профессионалов

```
typedef int(read_proc_t) (char *page, char **start, off_t off, int count, int *cof, void data);
typedef int (write_proc_t) (struct file *file, const char __user *buffer, unsigned long count, void
*data);
struct proc_dir_entry
{
        unsigned int low_ino;  // номер inode  // метаданные
        unsigned short namlen;  // длина
        const char *name;  // имя
        mode_t mode;  // права доступа и информация и типе файла
        n_link_t nlink;  // количество жестких ссылок на файл (кол-во имен файла в системе)
        uid_t uid;
        gid_t gid;
        loff_t size;  // loff_t - обертка типа unsigned long
        const struct inode_operations *proc_iops; //struct inode_operations - операции на inode
        const struct file_operations *proc_fops;  // file_operations пеерчисляет все действия,
которые определены в системе на файлах (read, write, open, …)
        struct proc_dir_entry *next, *parent, *subdir;
        read_proc_t *read_proc;       ll их то включают в эту структуру
        write_proc_t *write_proc;     ll то исключают из нее, почему?  потому что
разработчики могут их выкинуть по ненадобности
        atonue_t count;
        .....
}
```

struct inode - дескриптор файла
если файл находится на диске
то у этого файла будет дисковый inode

если файл на диске не находится то у него будет inode оперативной памяти

доступ к inode выполняется по его номеру

чтобы открыть файл в файловой системе proc предоставляются функции
extern struct proc_dir_entry *proc_create_data(const char*, umode_t, struct proc_dir_entry*, const struct file_operations*, void*);
static inline struct proc_dir_entry *proc_create(const char *name, umode_t mode, struct proc_dir_entry *parent, const struct file_operations *proc_fops);
{
	return proc_create_data(name, mode, parent, proc_fops, NULL);
}

для того чтобы записать информацию в ядро, считать информацию из ядра
нельзя использовать библиотечные функции
scanf позволяет считывать из стандартного потока ввода (связан с клавиатурой и мышью)
printf -//- выводить -//- (связан с монитором)

у ядра и клавиатуры разные адресные пространства и защищенные

для того чтобы взаимодействовать используется несколько функций
line uaccess.h
copy_from_user()
copy_to_user() -> sprintf()

причиной использования copy_from_user() является то, что в линукс память сегментирована
это значит что указатель сам по себе не ссылается на уникальную позицию в памяти
а только на соответствующую позицию в сегменте

тоже самое только другими словами:

у процесса свое адресное пространство защищенное
для процесса создается виртуальное адресное пространство
у него есть последовательность адресов
и отсчитывается смещение от базового адреса сегмента и получаем виртуальный адрес

ос это тоже коробочка
привилегированная
у нее есть тоже своя система адресов
напрямую мы не можем обратиться к адресному пространству системы

поэтому используются специальные функции

copy_to_user()
информация записана в буфер ядра и из буфера ядра может быть записана в адресное пространство процесса (-ора?)

static const struct file_operations my_file_fops =
{
	.owner = THIS_MODULE,
	.open = my_file_open,
	.write = my_file_write,
	.read = my_file_read
};

6.04.19

4я лабораторная

для каждого процесса в программе выводим символьное имя и идентификатор предка

```
struct task_struct
{
        …
        struct task_struct *next_task, *prev_task;
        …
        int pid;
        …
        char comm[16];
        …
        // parent
        …
}
```

init -> init_task

```
struct task_struct *task = &init_task;

do
{
        …
} while ((task = next_task(task))!= &init_task);
```

linux/sches.h

у нас загружаемый модуль ядра => нам доступны структуры ядра (task_struct)

5 лабораторная (открытые файлы)
в виде письменного отчета

демонстрируются проблемы буферзованного ввода-вывода

проблемы могут привесит к потери данных л к выводу данных не в том порядке, в котором рассчитывали

open read write - функции буферзации

ОТКРЫТЫЕ ФАЙЛЫ
vfs поддержвает 4 главные структуры:
struct superlock
struct inode
struct dentry
struct file

как
не
сдохнуть

суперблок предоставляет нам информацию о смонтированной файловой системе

1 блок занимает 4к

когда процесс открывает файл
для этого процесса создается файл
т.е. система хранит информацию о всех открытых файлах в struct file
эта структура формирует системную таблицу открытых файлов

один и тот же физический файл мб открыт несколько раз
при каждом открытии этого файла
для этого файла будет создаваться объект file
т.е. запись в таблице

```
struct file
{
        ...
        struct path f_path;
        struct inode *f_inode;
        const struct file_operations *f_op;  // указатель на таблицу файловых операциий
(перечисление операций определенных на файле)
        spinlock_t f_lock;  // frnbdyjt j;blfybt
        ...
        atomie_long_tf_count;
        unsigned int f_flags;
        fmode_t f_mode;  // режим доступа к файлу
        long offset type;
        struct mutex f_pos_lock;  // защищает поле pos // поле pos - смещение в файле
(логическое смещение)
        loff_t f_pos;  // смещение в файле  // у каждого открытого файла будет дескриптор и
в каждом дескрипторе будет поле pos
        struct fown_struct f_owner;
        ...
}
```

мьютекс - это не активное ожидание, а блокировка
освободить мьютекс может только процесс который захватил мьютекс
па
МА
ги
ТИ

```
/*protects f_ep_links, f_flags
 Must not be taken from IRQ contecst*/
```

информация о владельце нужна, чтобы обрабатывать соответствующие сигналы

в фортунках (лаба) мы определили свои функции read write

система позволяет переопределить стандартные функции работы с файлами (и функции работы и inode и суперблоками)

```
struct path
{
```

```
        struct vfsmount *mut;
        struct dentry *dentry; //
}
```

чтобы обратиться к файлу нужно последовательно обратиться ко всем элементам пути

С
Д
О
Х
Н
У
Т
Ь

```
struct file_operations
{
        struct module *owner;
        loff_t (*llscak)(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 *);
        int (flush)(struct file *, fl_owner_t id);
        int (*release)(struct inode *, struct file *);
        ...
}
```

в этой структуре перечислены все операции на открытых файлОх

```
ssize_t (*read)(struct file *filp, char __user *buffer, size_t count, loff_t offp);
ssize_t (*write)(struct file *filp, const char __user *buffer, size_t count, loff_t *offp);
```

char __user *buffer // указатель на буфер режима пользователя, куда/откуда перемещаются данные

filp - указатель на дескриптор файла
count - размер запрошенных перемещаемых данных

различные перемещение:
перемещения данных от устройства (девайса) в буфер пользователя

offp - указатель типа long offset type который указывает смещение файла

действия не могут осуществляться через указатели обычным путем или через memcpy()

адреса пространства пользователя не могут использоваться напрямую в режиме ядра по ряду причин, основное из которых: память пространства пользователя свапируемое (swapped out) (выгружаемое)

когда ядро обращается к указателю режима пользователя
связанная с ним страница может не находиться в памяти
в результате генерируется страничное прерывание (исключение)
при этом процесс пытающийся сделать такое действие будет отправлен в состояние сна (заблокирован)

до того момента
пока страница не будет загружена в память
поэтому в сердце (в глубине) любой функции read/write
закопаны (находятся) функции
unsigned long copy_to_user(void *to, const void *from, unsigned long count);
unsigned long copy_from_user(void *to, const void *from, unsigned long count);

в лабораторной №4 мы переопределяем в загружаемом модуле ядра стандартные функции ядра

для разработчиков драйверов важно понимать что
функции которые обращаются к пространству пользователей
должны быть реентерабельны
и должны позволять параллельные выполнения с другими функциями драйвера
для этого как правило в таких функциях используются семафоры

struct file формирут таблицу эта таблица пользователю недоступна
но эта структура связана со следующими файлами структуры

```
struct task_struct
{
        ...
        struct files_struct *files;
        struct fs_struct *fs;
        ...
}
```
struct files_struct *files; - структура формирует таблицу открытых файлов процесса
т.е. каждый процесс имеет собственную таблицу открытых файлов

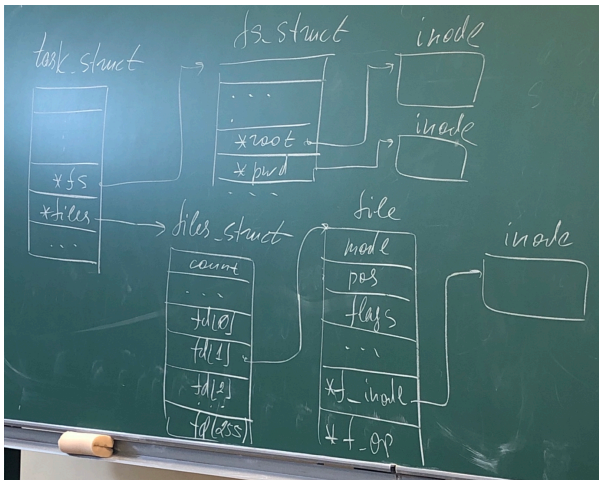
fs_struct представляет файловую системы к которой относится процесс

процесс принадлежит конкретной файловой системе

```
struct files_struct описывает дескриптор (ы) таблицы открытых файлов процесса
{
        atomic_t count;  // счетчик использования структуры // атомарный - неделимый
        spinlock_t file_lock;  // spinlock который защищает эту структуру  // spinlock (активное ожидание) это цикл проверки возможности выполнения дальнейшей работы
        ...
        int next_fd;  // число открытых процессом файлов
        struct file **fd;  // массив всех файловых объектов который содержит информацию о файлах открытых процессом
        ...
        struct file *fd_array[NR_OPEN_DEFAULT];  // NR_OPEN_DEFAULT =32 // массив файловых объектов открытых процессом и если процесс пытается создать больше чем 32 файловых объектов, то ядро создает новый массив в итоге процесс может открыть 256 файлов
}

struct fs_struct
{
        atomic_t count; // счетчик использования структур
        rwlock_t lock;  // блокировка read-write  // защищает структуру
        ...
        int umask;
        struct dentry *root;  // указатель на корневой каталог файловой системы
```

struct dentry *pwd; // указатель на текущий каталог
…
struct vfsmount *rootmnt; // структура которая описывает смотнтированную файловую систему // файловая система одного и того же типа может быть смонтирована много раз и такая смонтированная структура имеет struct vfsmount ????
} структура которая описываей файловую систему …

эти таблицы связаны следующим образом



*root -> struct dentry -> inode
*pwd -> struct dentry -> inode
это можно не указывать т.к. …. хз

20.04.19

СИСТЕМНЫЙ ВЫЗОВ OPEN

2 раза вызывает системный вызов open в результате создается 2 дескриптора …… (для лабы)

у
м
и
р
а
ю

ну почему именно первая пара

рисуем алгоритм на бумаге КАРАНДАШОМ => надо прожарить перед тем как к ней идти

системный вызов open сразу вызывает системный вызов sys_open

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

mode - битовая маска которая определяет права доступа

```
S_IRWXU -> 007700
S_IRWXG -> 00070
S_IRWX0 ->00007
```

sys_open выполняет всю работу

```
int sys_open(const char *filename, int flags, int mode)
{
        char *tmp = getname(filename); // передает имя из user mod в kernel mod  // в результате получаем имя файла
        int fd = get_unused_fd();  // ищет первый неиспользуемый дескриптор  // get_unused_fd возвращает первый неиспользуемый дескриптор
        struct file *f = filp_open(tmp, flags, mode);  // должна вернуть дескриптор открытого файла процесса  // внутри функции получаем указатель открытого файла внутри таблицы открытых файлов системы
        fd_install(fd.f);
        putname(tmp);
        return fd;
 }

int get_unused_fd(void)
{
        struct files_struct *files = current->files;  // указателю files присваивается значение файла текущего процесса current (указатель на пользовательский task struct текущего выполняемого задания)
        …
        files->next_fd() = fd + 1;  // найден последний занятый и к нему добавляется 1
        return fd;
}
```

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

struct nameidata
{
        strict dentry *dentry;  // описывает имя файла  (inode + путь чтобы найти файл)
        struct vfsmount *mnt;  // описывает файловую систему процесса
        struct qstr last;  // qstr - имя и размер имени
}

struct qstr
{
        union
        {
                HASH_LEN_DECLARE;
        }
        u64 ????? hash_len;
        const unsigned char *name;
}

struct nameidata
{
        struct path;
        struct qstr last;
        struct path root;
        struct inode *inode;
        …
}  сделано это для ускорения доступа тчообы миновать какие-то структуры


книга Understanding Linux Karnel

open_namei(const char *pathname, int flag, int mode, struct nameidata *nd)
{
        if (!(flag & O_CREAT))
        {
                if (*pathname == '/')
                {
                        nd->mntget(current->fs->rootmnt);
                        nd->dentry = dget(current->fs->root);
                }
                else {
                        nd->mnt = mntget(current->fs->pwdmnt);
                        nd->dentry = dget(current->fd->pwd);
                }
                path_walk(pathname, nd);
                ….
        }
        return 0;
}
```

проверяется корневой каталог или рабочая директория
mntget
path_walk - связывает с dentry (сначала пытается найти заданный dentry в кеше dentry
в кеше находятся dentry которые использовались в последнее время
если не нашли то внутри path_walk вызывается функция real_lookup() которая обращается
к файловой системе, т.е.. обращается к структурам валовой системы например struct inode,
struct suoeblock и пытается найти соответствующий суперблок
если он в памяти то ищется в памяти по соответствующим структурам иначе ….)

29.04.19

```
struct file_system_type
{
        count char *name;  // название файловой системы
        int fs_flags;
        #define FS_REQUIRES_DEV 1  //
        #define FS_BINARY_MOUNTDATA 2
        #define FS_HAS_SUBTYPE 4
        #define USERS_MOUNT 8
        #define FS_USERS_DEV_MOUNT 16
        #define FS_RENAME_NAME_DOES_D_MOVE 32768

        struct dentry *(*mount)(struct file_system_type *, int, const char*, void*);  // точка входа
файловой системы // вызывается когда монтируем файловую систему из командной
строки
        void (*kill_sb)(struct super_block *);  // размонтирование файловой системы  //
вызывается когда мы отмонтируем файловую систему
        struct module *owner;  // владельцем является загружаемый модуль ядра
        struct file_system_type *next;
        struct hlist_head fs_supers;

        // s - супеблок
        // i - inode
        struct lock_class_key s_lock_key;  //
        struct lock_class_key s_umount_key;
        struct lock_class_key s_vfs_rename_key;  // если ктого захочет переименовать
файловую системы, а ее кто-то использует по старому имени  // хз зачем я это пишу
        struct lock_class_key s_writers_key [SB_FREESE_LEVELS];
        struct lock_class_key i_lock_key;
        struct lock_class_key i_mutex_key;
        struct lock_class_key i_mutex_dir_key;
}
```

каждая смонтирована файловая система будет иметь суперблок

дисковый inode хранить информацию об адресах блоков памяти которые занимает вторичный файл

чтобы смонтировать файловую систему у нее должен быть корневой каталог root
у root должен быть inode

nmount - отмонтирование

перед тем как использовать файловую систему мы должны ее зарегистрировать

```
// C99
struct file_system_type my_fs_type =
{
        .owner = THIS_MODULE,
        .name = "mmy",
        .mount = my_mount,
        .kill_sb = my_kill_superblock,  // регистрируем mount
        .sb_flags = FS_REQUIRES_DEV,
};
```

```
// другой стандарт заполнения структур
struct file_operations fops =
{
	read : device_read,
	write : device_write,
	open : device_open,
};
```

функции переопределяются разработчикам и драйверов

```
ststic int __init my_init(void)
{
	int re;
	my_inode_cachep = nmem_cache_create("mmy_inode_cachep", sizeof(struct my_inode),
0, (SLAB_RECLAIM_ACCOUNT I SLAB_MEM_SPREAD), NULL);
	if (!my_inode_cachep)
	{
		return _ENOMEM;
	}
	ret = register_filesystem(&my_fs_type);
	if (likely (ret == 0))
		printk(KERN_INFO "SUCCESS\n");
	else
		printk(KERN_ERR «FAILED.ERROR-%d\n", ret);
	return ret;
}

static void __exit my_cleanup(void)
{
	int ret;
	ret = unregister_filesystem(&my_fs_type);
	kmem_cache_destroy(my_inode_cachep);
	if (likely (ret == 0))
		printk(KERN_INFO "SUCCESS\n");
	else
		printk(KERN_ERR «FAILED.ERROR-%d\n", ret);
}


ststic struct dentry *gfsmount(struct file_system_type *fst, int flags, const char *dev_name, void
*data)
{
	struct dentry *ret = mount_nodev(fst, flags, data, gfsfillsuper);  // возвращает точку
монтирования
	if (IS_ERR(ret))
	{
		printk(KERN_ERR "GFS ERROR\n");
	}
	else
	{
		printk(KERN_INFO "mount"\n);
	}
	retunr ret;
}
```

// mount_nodev - стандартная функция, которой передается функция которая в стандарте называется fillsuper
она заполняет поля суперблока и создает inode roota

каталог это файлы => следователь каталог должен иметь inode

struct dentry *mount_ns(struct file_system_type *fs_type, int flags, void *data, int *fill, void *data, int (*fill_super)(struct super_block *void, *int));
extern struct dentry *mount_bdev(struct file_system_type *fs_type, int flags, const char *dev_name, void *data, int (*fill_super)(struct super_block *, void *, int));

extern struct dentry *mount_nodes(struct file_system_type *fs_type, int flags, void *data, int *fill, void *data, int (*fill_super)(struct super_block *void, *int));

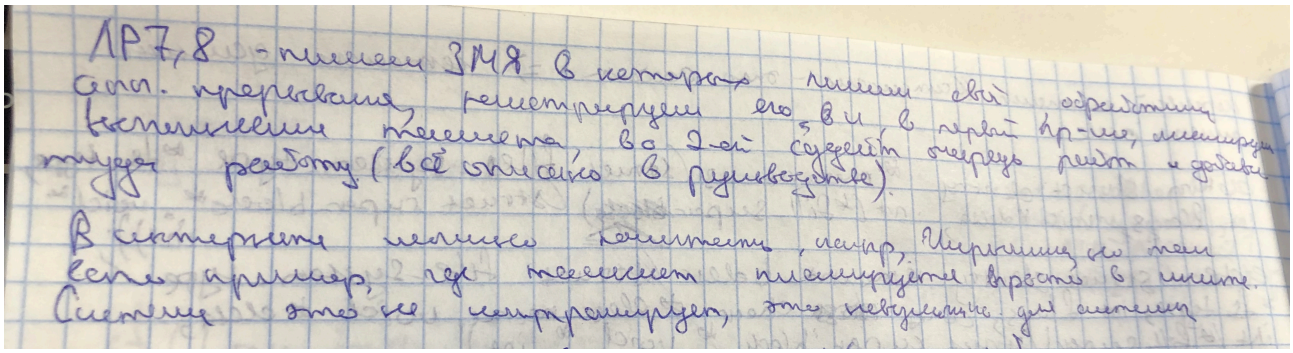
vfsmount - эта структура используется для представления конкретного экземпляра файловой системы (точки монтирования)

```
struct vfsnount
{
	struct dentry *mnt_root; // точка монтирования
	struct super_block *mnt_sb;  // указывает на sb
	int mnt_flags;  // MNT_NODEV - этот флаг запрещает доступ к файлам устройств
}


static int dfsfillsuper(struct super_block *sb, void *datam int silent)
{
	sb->s_blocksize = PAGE_SIZE;
	sb->s_blocksize_bits = PAGE_SHIFT;
	sb->s_op = &gfssbop;
	sb->s_magic = GFS_MAGIC_NUMBER; // магическое число используется для идентификации

	struct inode *rootinode = gfsgetinode(sb, NULL, S_IFDIR, NULL);
	...
	struct dentry *rootdentry = d_make_root(rootinode);
	sb->s_root = rootdentry;
	return 0;
}
```

simple функции - функции содержании минимально необходимый набор действийа

В 6Й ЛАБЕ ДОБАВИТЬ КЕШИ

18.05.19



typedef irqreturn_t(*irq_handler_t)(int, void *);
int request_irq(unsigned int irq, irq_handler_t handler, iunsigned longg flags, const char*name^
void *dev);
extern void free_irq(unsigned int irq, void *dev);
char *name - имя которые можно увидеть в системе
void *dev - * значит что сюда можно передать все что угодно , присутствует тк надо удалить конкретный обработчик прерываний

IRQ ЭТО СВЯТОЕ!!!!!!!!!
система устроена так что линии запроса прерывания могут …. с помощью флага IRQF_SHARED
ps/2 1irq - клавиатура

```
irqreturn_t irq_handler(int irq, void *dev_id, struct pt_regs *regs)
{
        if(irq == 12)
        {
                …
                return IRQ_HANDLER;
        }
        else
                return IRQ_NONE;
}
```

КОМПЬЮТЕР НЕ МОЖЕТ ДУМАТЬ

```
int init_module(void)
{
        return request_irq(12, (irq_handler_t) irq_handler, IRQF_SHARED,
                        «text_mouse_irq_handler», (void*)(irq_handler));
}
```

(void*)(irq_handler) - передали requestirq в обработчик
если в это поле будет поставлен null, то тогда невозможно идентифицировать кто установил прерывание, т.е. невозможно установить источник для того чтобы это прерывание отключить

в лабе тасклет устанавливать в обработчике прерываний

segg_file_interface - один из способов вывода информации

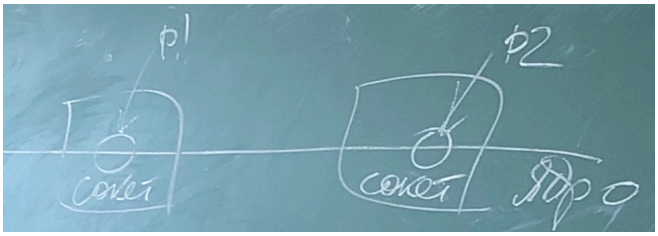
СОКЕТЫ (9-10 лаба)

IPC systemV (5 five)

сигналы - базовый механизм взаимодействия

в unix bsd средства взаимодействия параллельных процессов - сокеты

сокеты создаются системным вызовом сокет



сокет - это абстракция конечной точки взаимодействия

порт - это адрес

в asm есть in и out прочитать про это

memory mapping - отображение на адресное пространство физической памяти

int socket(int family, int type, int protocol);
family - семейство или домен

сокеты бывают (виды):
PF_INET - с которыми мы будем работать
PF_PACKET - сокеты более низкого уровня созданные для непосредственного доступа приложения к сетевым устройствам
PF_NETLINK - средство получения информации о ядре, о объектах ядра

netlink_firewall

sys_socket() - заполняет поля структуры struct socket
<sys/types.h>
<sys/socket.h>

```
struct socket
{
        socket_state state;
        short type; // тип сокета
        unsigned long flags; // флаги используются для синхронизации доступа
        const struct proto_ops *ops; // указатель который ссылается на действия связанные с подключенным протоколом
        struct fasync_struct *fasync_list;  // список асинхронного запуска
        struct file *file;  // явное указание на то, что сокет это специальный файл
        struct sock *sk; // связный список сокетов
        wait_queue_head_t wait; // очередоь ожидания
}
```

состояния сокета:
- SS_FREE
- SS_UNCONNECTED
- SS_CONNECTING
- SS_CONNECTED
- SS_DISCONNECTING

тип сокета:
- SOCK_STREAM
- SOCK_DFRAM
- SOCK_RAW
- SOCK_RDM
- SOCK_SEQPACKET
- SOCK_PACKET (не используется)

параметры family и type являются определяющими

пространство имен family:
domen
все константы определенные в пространстве имен начинаются с букв AF - Adress family
AF_UNIX - домен unix, сокеты для межпроцессного взаимодействия на локальном компьютере
AF_INET_IPV4 - домен интернет, т.е. любая ip сеть (сокеты этого домена предназначены для взаимодействия процессов в сети и работают на основе протокола tcp ip)
AF_INER_IPV6
AF_IPX - domain IPX
AF_ANSPEC - семейство не специфицированных сокетов

sock_stream - определяют ориентированную на потоки надежное логическое полнодуплексное соединение между 2мя сокетами

sock_dgram - определяют ненадежную службу dgram без надежного установления логического соединения где паркеты могут сохраняться (широковещательная передача данных)

sock_raw - низкоуровневый интерфейс по протоколу ip

3й параметр - протокол
определяется на основании 1 и 2го
если в этом параметре 0
то протокол будет установлен по умолчанию на основании первых 2х полей

дизайн сокета беркли следует парадигме unix, а именно в идеале отобразить все объекты в которым осуществляется доступ для чтения или записи на файле
чтобы с ними можно было бы работать с использованием функций записи и чтений

т.е. объектами которыми манипулируют при операции чтении и записи, в контексте транспортных протоколов, являются конечные точки коммуникационных отношений, т.е. сокеты

АДРЕСА СОКЕТОВ
была определена общая структура адреса struct sockaddr

```
struct sockaddr
{
        sa_family sa_family;  // семейство адресов AF_xxxx
        char sa_data[14];  // 14 байт адреса ппротокола
}
```

для адресов интернета используется другая структура

адреса и номера портов должны быть указаны в сетевом порядке байтов

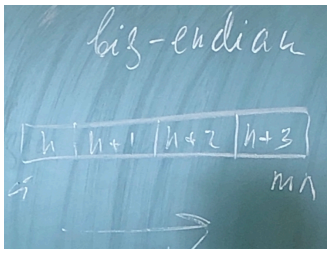
```
struct in_addr
{
        _u32 s_addr;
};

struct sockaddr_in
{
        sa_family sin_family;  // семейство адресов AF_INEX*;
        nnsigned short int sin_port;
        unsiggned char sin_zero[sizeof(struct sockeaddr) - sizeof(sa_family_t) - sizeof(uint16_t) -
sizeof(struct in_addr)];
        // должен еще быть sin_addr
}
```

ПОСМОТРЕТЬ МУЛЬТИК ГУЛИВЕР В СТРАНЕ ЛИЛИПУТОВ
СПРАШИВАЕТСЯ НА ЭКЗАМЕНЕ
ПОСЕТИТЬ ВЫСТАВКУ РЕПИНА!!!!!

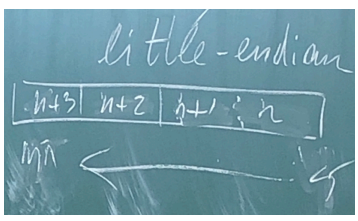
little-ending
big-ending

если тип занимает несколько байтов
то имеет значение порядок байтов

big_ending это network byte oder



little-ending это hort byte oder



ОС	Процесс	порядок
FreeBDS5.2.1	Intel Pentium	little
linux 2.4.22	-//-	-//-
mac os x 10.3	power pc	big
solaris9	sun spare	-//-

для некоторых платформах аппаратный порядок байтов совпадает с сетевым но все равно надо использовать эти функции
это обеспечивает переносимость по …

htons ntohs
htonl ntohl