

Лекции (и семинары) “Операционные системы 6 семестр ИУ7”

Файловые подсистемы	2
Загружаемые модули ядра	4
Внутренняя организация VFS.	5
struct super_block	5
struct inode	7
struct dentry	8
struct file	9
Пример создания Ф.С.	10
Структура раздела жд с Ф.С. ext2	12
Виртуальная файловая система /proc	12
read()	15
open()	15
Открытые файлы	17
files_struct	17
fs_struct	18
namespace	18
Сокеты	19
Адресация сокетов	20
Пять моделей ввода/вывода	24
I.Блокирующий ввод/вывод	24
II.Неблокирующий ввод/вывод	24
III.Мультиплексирование ввода/вывода	24
IV.Ввод/вывод, управляемый сигналами	26
V.Ассинхронный ввод/вывод	27
Сравнение пяти моделей ввода/вывода	27
Драйверы и прерывания	28
Медленные прерывания	29
Отложенные прерывания	29
Тасклеты	31

Планирование тасклетов	32
Сравнение отложенных прерываний и тасклетов	32
Очереди работ	33
Устройства	35
Драйверы	36
USB драйвер	37
Регистрация USB драйвера	42
Управление буферами	44
Методы буферизации	44
Простая буферизация	45
Обменная буферизация	46
Системные вызовы	46
Спин-блокировки в ядре Linux	47
Спин-блокировки чтения и записи	47

Файловые подсистемы

Ф.П. - управление данными над уровнем управления устройством.

Задача Ф.П. - хранение данных: записать данные так, чтобы можно было многократно к ним обращаться.

Используются специальные внешние устройства - диски.

Файл - любая поименованная совокупность данных. Их содержание систему не интересует, только возможность доступа к ним.

Файл - информация, хранимая во вторичной памяти или во вспомогательном ЗУ с целью ее сохранения после завершения конкретного задания и в преодолении ограничений, связанных с объемом рабочих ЗУ.

Файловая система - порядок, определяющий способ организации хранения, именования и доступа к данным на вторичных носителях.

- Используется для хранения данных и доступа к ним.
- Часть ОС.
- Должна обеспечивать создание, чтение, запись, удаление, переименование файлов
- Определяет формат хранимой информации и способ ее физического хранения.
- Связывает физ.носитель информации и API для доступа к файлам.

Задачи Ф.С.

- Именованние файлов с точки зрения пользователя
- Обеспечение программного интерфейса для работы с файлами пользователя и приложения
- Отображение логической модели файлов на логическую реализацию хранения данных на соотв. носителях.
- Обеспечение надежного хранения файлов, доступа к ним и защиты от несанкционированного доступа
- Обеспечение совместного использования файлов.

Иерархическая структура Ф.С.

<p>Символьный уровень</p> <p>предоставляет возможность систематизации документов.</p> <p>уровень именования файлов и системат. управление информацией, доступной пользователю.</p>
<p>Базовый уровень</p> <p>уровень формирования дескриптора файлов (описан в системе).</p> <p>должны быть соответствующие структуры, позволяющие хранить необходимую для файла информацию</p>
<p>Логический уровень</p> <p>логическое а.п. файла аналогично логич.а.п. процесса</p> <p>позволяет обеспечить доступ к данным в файле в формате, отличном от формата физ.хранения</p> <p>обычно система не накладывает ограничения на внутр. структуру данных в файле</p>
<p>Физический уровень</p> <p>обеспечение непосредственного доступа к информации на внешнем носителе</p>
<p>Внешние устройства хранения</p>

Хранение файла на диске

Связное распределение: файл может храниться на диске в непрерывной последовательности адресов. (ограничивает размеры файла, фрагментация)

Несвязное распределение: выделяются участки пространства диска вразброс (доп. задачи адресации, обеспечение эффективного доступа к информации)

Ф.С. определяет особенности того, как открывать файл, найти его, прочитать или записать в него информацию.

Физ.код Ф.С. скрывает детали реализации, но все Ф.С. поддерживают понятия: файлы, каталоги

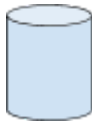
действия: создание, удаление, переименование, открытие, чтение, запись, закрытие

Физически ничто в ядре не должно понимать низкоур. детали Ф.С. кроме самих Ф.С.

```
write(f, fbuf, len);
```

Записывает len байт из fbuf в текущую позицию файла, представленного файловым дескриптором f.

В системе будет обрабатываться:

write()	sys_write()	filesystem write method	
user space	VFS	filesystem	physical layer

Сначала обрабатывается системным вызовом `sys_write()`, который определяет фактический способ записи файлов для Ф.С., где находится файл, описанный фд `f`. Затем вызывается метод конкретной Ф.С. для записи данных на конкретный файловый носитель.

Дерево каталогов и файлов

Новые ветви появляются путем монтирования новых Ф.С. Любая Ф.С. монтируется к каталогу - точке монтирования. Все операции осуществляются через интерфейс VFS. Linux поддерживает интерфейс FHS - стандарт иерархии Ф.С. определяющий структуру каталогов:

- должна быть корневая Ф.С. : /
- корневой каталог и ветви - единая Ф.С. на одном носителе
- в нем располагаются компоненты, необходимые для старта системы

Загружаемые модули ядра

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Gorokhova Irina");
MODULE_DESCRIPTION("Module Printing Hello & Goodbye Messages");

static int __init my_module_init(void) {
    printk("Hello world!\n");
    return 0;
}

static void __exit my_module_exit(void) {
    printk("Goodbye!\n");
}

module_init(my_module_init);
module_exit(my_module_exit);
```

Для работы с загружаемыми модулями ядра чаще всего используются makefile-ы. Загрузка модуля:

```
> sudo insmod <имя.ko>
```

Просмотр очереди сообщений:

```
> dmesg
```

Выгрузить модуль:

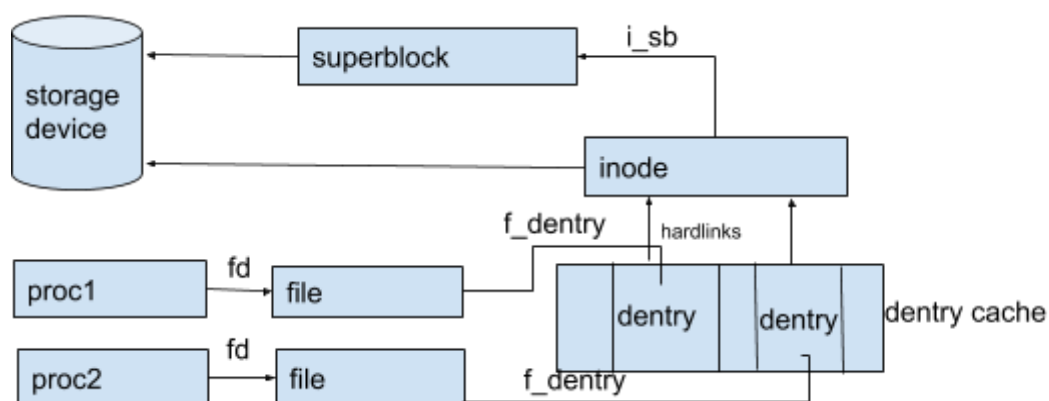
```
> sudo rmmod <имя>
```

- insmod - загружает код модуля и данные в ядро. Чтобы разместить модуль и данные, с которыми он работает, используется функция **vmalloc()**
- rmmod - выгружает код модуля и данные из ядра. Завершится неудачей, если удаляемый модуль используется.
- lsmod - выдает список модулей, загруженных в ядро в данный момент. Работает с виртуальным файлом /proc/module

Внутренняя организация VFS.

VFS базируется на четырех основных структурах:

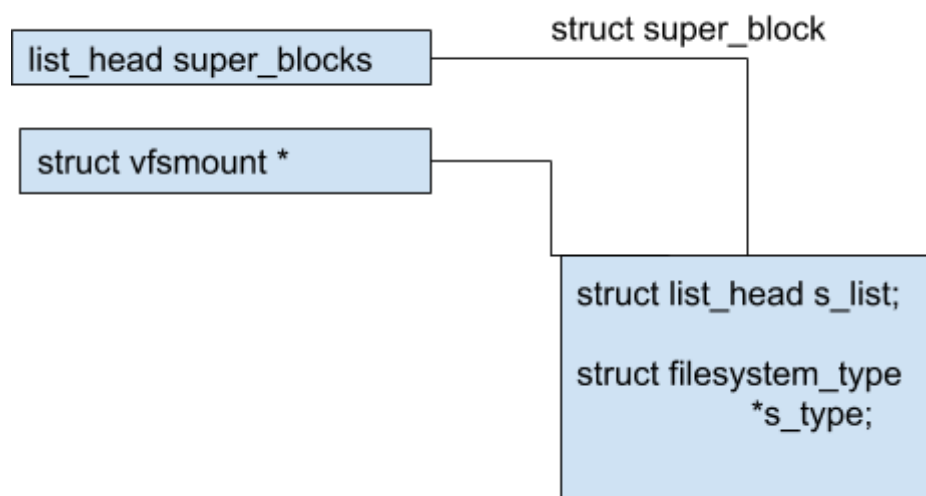
- superblock
- dentry
- inode
- file



struct super_block

- описывает конкретную файловую систему

Организуется двусвязный список всех смонтированных Ф.С.



Superblock содержит общую информацию о Ф.С. - контейнер

Описывает информацию более высокого уровня чем та, которая необходима для доступа к информации на диске. Superblock хранится на диске

Чтобы создать/подгрузить ф.с. на диске необходимо создать раздел. Суперблок описывает этот раздел.

Суперблок определяет управление параметрами Ф.С. - количество блоков, количество свободных блоков, корневой inode.

Объект super_block создается и инициализируется функцией **alloc_super()**.

Вызывается при монтировании Ф.С.

```
#include <linux/fs.h>
struct super_block {
    struct list_head s_list;
    kdev_t s_dev;                //устройство
    unsigned long s_blocksize;   //размер блока в
    байтах
    unsigned char s_dirt;        //флаг изменения
    суперблока
    ...
    struct file_system_type *s_type; //тип файловой
    системы
    struct super_operations *s_op;   //операции, опред.
    на суперблоке
    ...
    struct dentry *s_root;          //точка
    монтирования
    ...
    int s_count;
}
```

Структура super_block взаимодействует со структурой struct vfsmount *, которая описывает объект управления vfsmount. Он тоже предоставляет информацию о смонтированных файловых системах.

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

Структура с операциями. определенными на суперблоке.

```
struct super_operations {
    //создает и инициализирует новый inode
    связанный с суперблоком
    struct inode *(alloc_inode)(struct super_block *sb);
    //уничтожает объект inode
    void (*destroy_inode)(struct inode *);
}
```

```

//вызывается подсистемой vfs когда в
индекс inode вносятся изменения
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);
//вызывается системой vfs, когда
удаляется последняя ссылка на индекс
//тогда vfs просто удаляет inode
void (*drop_inode)(struct inode *);
//вызывается vfs при размонтировании
void (*put_super)(struct super_block *);
//обновляет суперблок на диске
void (*write_super)(struct super_block *);
}

```

Из всего набора операций можно определить только нужные.

Каждый элемент в структуре - указатель на функцию, выполняющую действие с суперблоком (низкоуровневые действия с Ф.С.)

Когда система нуждается в выполнении операции над суперблоком:

```
sb -> s_op -> write_super(sb);
```

struct inode

В Ф.С. каждый файл имеет свой inode. (/proc - 1, / - 2)

При создании Ф.С. создается таблица inode-ов с информацией о том, какие inode заняты, а какие свободны. В суперблоке хранится информация о том, сколько в системе свободных inode-ов.

ОС хранит информацию о файле в inode - метаданные о данных.

Когда пользователь или программа нуждается в доступе к файлу система ищет точный и уникальный inode в таблице inode-ов.

Чтобы получить доступ к файлу по его имени, нужно найти inode данного файла. Для доступа к номеру inode имя файла не требуется. Структура inode:

inode	
owner info	
size	
time stats	
direct blocks	чем больше косвенных
indirect blocks	ссылок, тем медленнее
double indirect blocks	доступ
triple indirect blocks	

```

struct inode {
    umode_t i_mode;
    uid_t i_uid;           //user id
    time_t i_atime;        //attach
    time_t i_mtime;        //modify
    time_t i_ctime;        //create
    ...
    unsigned short i_bytes;
    //определяет набор функций, важных с т.з.
    доступа к файлу
    struct inode_operations *i_op;
    struct file_operations *f_op;
    struct super_block *i_sb;
}

```

Структура с inode-операциями

```

struct inode_operations {
    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*);
    ...
}

```

Информация хранится dev_t. Файл хранится в каталоге, указанном в struct_dentry с правами доступа umode.

Структура с file-операциями (определены на объекте FILE)

```

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*);
}

```

struct dentry

- directory entry

vfs представляет каталоги как файлы, выполняет поиск компонента пути по имени, проверяет существование пути, переход а следующий компонент пути.

Объект dentry - определенный компонент пути. Причем все объекты dentry - компоненты пути, включая обычные файлы. Элементы могут включать в себя точки монтирования.

vfs создает эти объекты на лету по строковому представлению имени пути.

struct dentry описывает не только каталоги, но и то, что м.б. описано как 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;
    ...
}
```

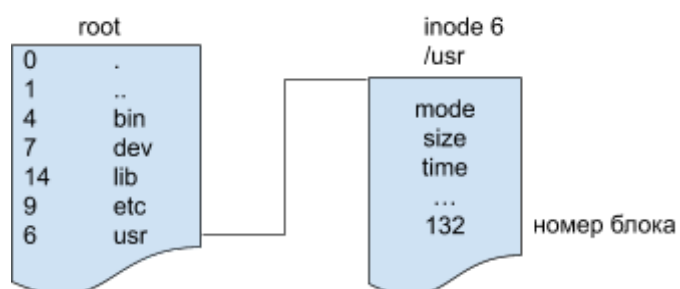
Имена значения не имеют, имеет значение процесс доступа к файлу

Пример:

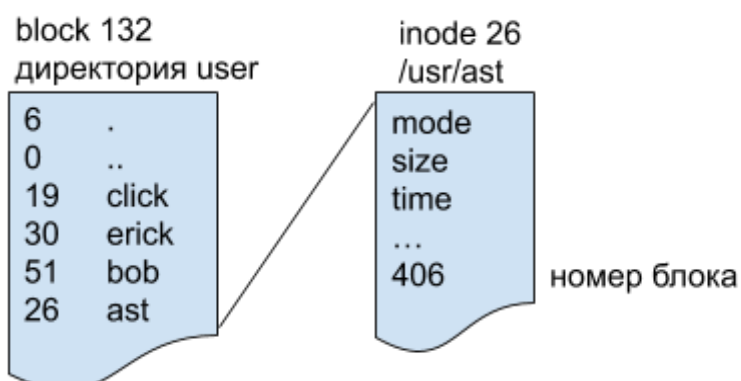
```
/usr/ast/mbox
```

имеет 4 объекта dentry:

- root - /
- /usr
- /usr/ast
- mbox



Говорят, что информация, на которую указывает inode 6 находится в блоке 132.



Говорят, что /usr/ast находится в блоке 406

block 406
содержимое
поддиректории ast

26	.
6	..
64	abc
92	mbox
60	eofg

Для ускорения доступа информация, полученная с помощью dentry кэшируется.

struct file

- одна из четырех структур, определяемых как основные для работы с файлами.
FILE - структура, представляющая открытые файлы.

```
struct FILE {
    struct dentry *f_dentry;
    //указатель на точку монтирования
    struct vfsmount *f_vfsmount;
    struct file_operations *f_op;
    //права доступа для файла
    mode_t f_mode;
    //поле с текущей позицией файла
    loff_t f_pos;
    ...
}
```

Пример создания Ф.С.

Структура file_system_type:

```
struct file_system_type {
    const char *name;
    int fs_flags;
    struct super_block *(*get_sb)(struct file_system_type *,
        int, char *, void *, struct vfsmount *);
    void (*kill_sb) (struct super_block *);
    struct module *owner;
    struct file_system_type *next;
    struct list_head fs_supers;
    struct lock_class_key s_lock_key;
    struct lock_class_key s_umount_key;
}
```

Заполняем структуру struct file_system_type:

```
static struct file_system_type encfs_fs_type = {
    .owner = THIS_MODULE,
```

```

        .name = "encfs",
        .mount = encfs_mount,
        .kill_sb = encfs_kill_block_super,
        .fs_flags = FS_REQUIRES_DEV,
};

```

В функции загрузки модуля вызывается `register_filesystem` с заполненной структурой:

```

static int __init encfs_init(void) {
    ...
    ret = register_filesystem(&encfs_fs_type);
    if (likely(ret == 0))
        printk(KERN_INFO "Successfully registered!\n");
    else
        printk(KERN_ERR "Failed register! Error [%d]\n", ret);
    return ret;
}

```

В функции выгрузки модуля вызывается `unregister_filesystem`:

```

static int __exit encfs_cleanup(void) {
    int ret;
    ret = unregister_filesystem(&encfs_fs_type);
    ...
    if (likely(ret == 0))
        printk(KERN_INFO "Successfully unreg!\n");
    else
        printk(KERN_ERR "Failed unregister. Error [%d]\n", ret);
}

```

Передаем эти функции макросами:

```

module_init(encfs_init);
module_exit(encfs_cleanup);

```

`.mount` будет вызываться при монтировании Ф.С., а `.kill_sb` при удалении суперблока

```

static struct dentry *encfs_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, encfs_fill_super);
    if (unlikely(IS_ERR(ret)))
        ...
    return ret;
}

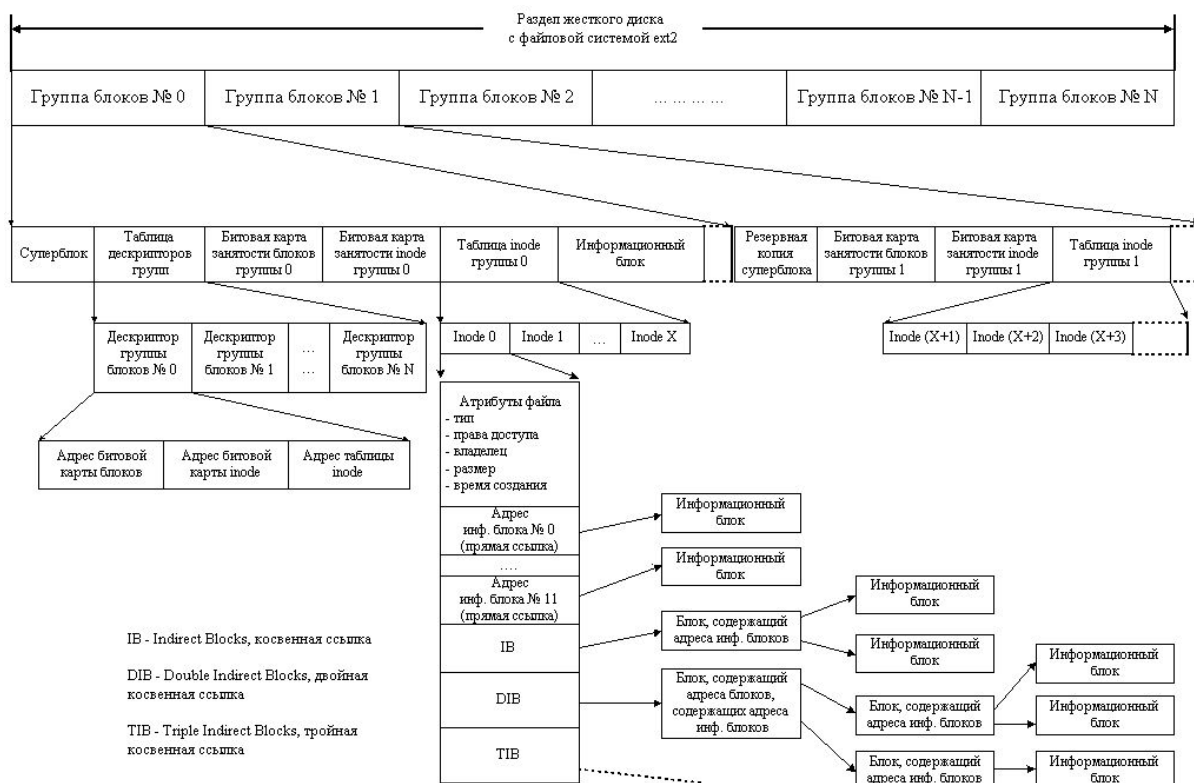
```

```

static void encfs_kill_superblock(struct super_block *vsb) {
    ...
    kill_block_super(vsb);
}

```

Структура раздела жд с Ф.С. ext2



Суперблок считывается в память ядра при монтировании Ф.С. и остается там до отмонтирования. Поэтому есть флаги dirty и пр.

Виртуальная файловая система /proc

Ф.С. proc не является монтированной.

Представляет собой интерфейс ядра, позволяющий приложению читать и изменять данные в адресном пространстве других процессов, управлять процессами, получать информацию о других процессах и ресурсах, используя для этого стандартный интерфейс Ф.С. и системных вызовов.

Управление доступом к адресному пространству осуществляется с помощью read/write/execute.

Информация о каждом процессе хранится в поддиректории /proc/<pid>. В ней находятся файлы и поддиректории, содержащие данные о процессе.

Элемент	Тип	Содержание
cmdline	файл	указывает на директорию процесса
pwd	символическая ссылка	указывает на директорию процесса
environ	файл	список окружения процесса
exe	символическая ссылка	указывает на образ процесса
fd	директория	ссылки на файлы, открытые процессом

root	символическая ссылка	указывает на корень Ф.С. процесса
stat	файл	содержит информацию о процессе

/proc/self - в этой поддиректории каталог с данными

Используется структура proc_dir_entry

```
struct proc_dir_entry {
    unsigned short low_ino;      // номера inode файла
    unsigned short namelen;
    const char *name;           // имя виртуального
    файла
    mode_t mode;                // права доступа
    nlink_t nlink;
    uid_t uid;
    gid_t gid;
    unsigned long size;
    struct inode_operations * ops;
    // функции чтения и записи
    int (*get_info)(char *buffer, char **start,
        off_t offset, int length, int unused);
    void (*fill_inode)(struct inode *);
    struct proc_dir_entry *next, *parent, *subdir;
    void *data;
};
```

Чтобы начать работать с proc нужно создать ссылку на структуру proc_dir_entry с помощью вызовов proc_create или proc_create_data

Пример (создание proc_entry)

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/sched.h>
#include <linux/mm.h>
#include <linux/slab.h>
#include <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, loff_t *offt)
{
    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, data, count);
    if (count == 0)
        temp = len;
    return count;
}

struct file_operations proc_fops = {
    .read = read_proc,
}

void create_new_proc_entry() {
    msg = "Hello!";
    proc = proc_create_data("hi", 0, NULL, &proc_fops, msg);
    len = strlen(msg);
    temp = len;
}

int __init proc_init(void) {
    create_new_proc_entry();
    return 0;
}

void __exit proc_cleanup(void) {
    remove_proc_entry();
}

module_init(proc_init);
module_exit(proc_cleanup);

```

Определены также функции:

- proc_symlink()
- proc_mkdir()
- proc_mkdir_data()
- proc_mkdir_mode()
- extern void *PDE_DATA(const struct inode*)
- proc_remove()
- remove_proc_entry()
- remove_proc_subtree()

Ядреный буфер не доступен никому кроме функций. Можно использовать метаску

Загружаемые модули ядра, если используется Ф.С. прос, позволяют получить доступ к функциям ядра, а прос позволяет получить информацию о процессах и ресурсах которые они используют.

Система предоставляет средства взаимодействия с приложениями:

- copy_to_user
- copy_from_user

(самостоятельные функции)

read()

- вызывается из системного вызова read.

Сис.вызов read читает из файлового дескриптора.

```
#include <unistd.h>
ssize_t read(int fd, void *buff, size_t count);
```

Пытается прочитать count байт из файл.дескриптора fd в буфер buff (байты читаются из файла, описанного fd)

На файлах, которые поддерживают поиск seeking, операция read начинается со смещения в файле и смещение увеличивается на прочитанное количество байтов.

Если смещение на конце файла read вернет 0.

open()

- системный вызов. Открывает и (возможно) создает файл.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Открывает файл, определенный pathame. Если файл не существует, то если установлен флаг O_CREATE, то он м.б. создан.

Флаг O_EXCL приводит к ошибке, если файл уже существует.

Комбинация O_CREATE и O_EXCL проводит проверку существования файла и его создание и делается это атомарно

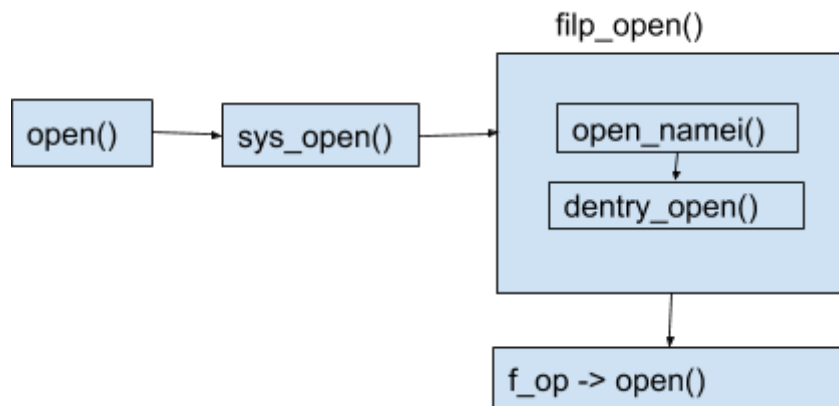
Возвращает файловый дескриптор открытого файла.

Смещение файла - в начале файла.

Создается новый дескриптор открытого файла, запись в системной таблице открытых файлов.

Получить файловый дескриптор можно с помощью функции

```
int fileno(FILE *fp);
```



Системный вызов `open()` реализуется в виде функции `sys_open()`

```

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

`get_unused_fd` ищет пустую строку в таблице дескрипторов файлов.

В случае успеха - вызов `filp_open`:

```

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

- `open_namei` для генерации структуры `nameidata`. Эта структура выполняет связь с `inode` файла. Поиск `inode` осуществляется в хеш-таблице под блокировкой `inode_lock`
- функции `dentry_open` передается информация из структуры `nameidata`. `dentry_open` размещает новую структуру `struct file` и связывает ее с `dentry` и с `vfsmount`. Затем вызывается метод `f_op -> open()`, который был установлен в `inode_i_op` при чтении `inode`.
- функция `dentry_open` находит адрес функции на уровне Ф.С. которая может выполнить операцию открытия, передавая ей `inode` и структуру `file`.

`Filp_open()` находит `dentry` для конкретного файла. Она делает это спускаясь в соответствии с заданным путем от одного компонента к другому начиная с корневой или рабочей директории. Спуск осуществляется, пока не будет получен нужный файл.

Если `inode` прочитан, то увеличивается его счетчик ссылок `icount`. Если счетчик = 0 и `inode` не грязный (информация не была изменена), то удаляется из списка `inode_unused` и вставляется в список `inode_in_use`, при этом счетчик, показывающий неиспользованные `inode` уменьшается.

inode хеш представляет из себя:

- глобальный хеш массив `inode_hashtable`, в котором каждый `inode` хешируется

по значению, указанному на суперблоке и по 32 разрядному номеру inode.

- глобальный список inode_in_use
- глобальный список inode_unused - неиспользуемые inode, содержащие допустимые inode с icount=0
- список для каждого суперблока

sb->s_dirty который содержит inode-ы с icount > 0 и inlink > 0

(когда inode помечен как грязный, он добавляется к списку sb->s_dirty, при условии что он был хеширован. Поддержка такого списка уменьшает расходы на взаимноеисключение).

Открытые файлы

task_struct - структура, описывающая процесс в Linux

В структурах есть два поля с информацией об открытых файлах

```
struct task_struct {
    ... //состояние, поля флагов, поля,
    описывающие адресное пространство
    struct mm_struct *mm; //описывает
    адрес.процесса
    struct mm_struct *active_mm; //описывает
    адрес.предыдущего процесса
    struct files_struct *files;
    struct fs_struct *fs;
    ...
}
```

Каждый процесс имеет свой список открытых файлов.

В VFS 3 структуры использующиеся для представления открытых файлов:

- files_struct
- fs_struct
- namespace
- (*)FILE

files_struct

```
#include <linux/file.h>
struct files_struct {
    atomic_t count; //количество
    пользователей
    spinlock_t file_lock; //средство
    взаимногоисключения
    int max_fds; //макс.количество
    файловых объектов
    int max_fdset; //макс.количество
    файловых дескрипторов
    int next_fd; //номер дескриптора
    следующего файла
    struct file **fd; //массив всех файловых
    объектов
```

```

    fd_set *close_on_exec; //массив файловых
    объектов закрытых exec
    fd_set *open_fds;      //указывает на открытые
    файловые дескрипторы
    fd_set close_on_exec_init;//иниц. файлы, которые
    закрыты exec-ом
    fd_set open_fds_init;
    struct file *fd_array[32];//массив открытых
    файл.дескрипторов
}

```

Если открыто больше 32 файловых объектов, ядро выделяет новый массив и указатель на него.

fs_struct

содержит информацию о файлах и файловых дескрипторах.

```

#include <linux/fs_struct.h>
struct fs_struct {
    atomic_t count;          //количество
    пользователей
    rwlock_t lock;          //лок-защита структуры
    int umask;               //права доступа
    struct dentry *root;     //корневая директория
    struct dentry *pwd;      //текущая директория
    struct dentry *altroot;  //альтернативная корневая
    директория
    struct vfsmount *rootmnt;//объект монтирования
    корневой директории
    struct vfsmount *pwmnt;  //объект монтирования
    текущей директории
    struct vfsmount *altrootmnt;//объект монтир.
    альтерн. корневой дир-рии
}

```

namespace

Структура namespace имеется в дескрипторе процесса (указатель на нее)

Позволяет каждому процессу иметь свое видение смонтированной файловой системы, но это не относится к корневому каталогу

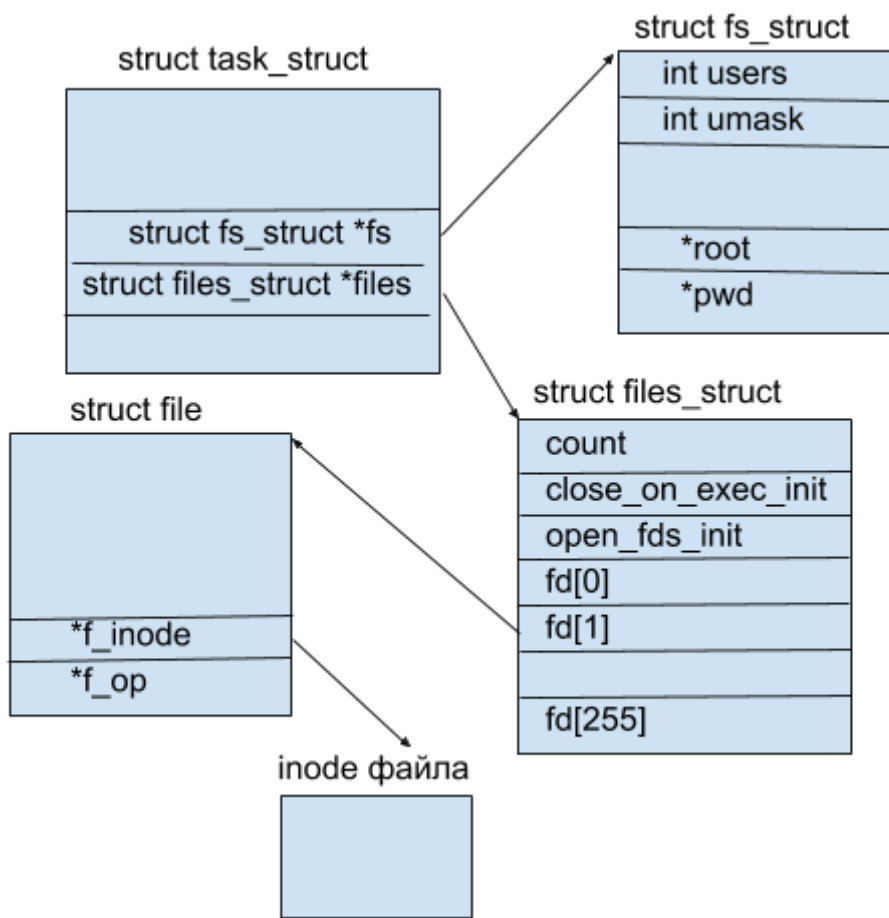
```
#include <linux/namespace.h>
```

```

struct namespace {
    atomic_t count    ;
    struct vfsmount *root;
    struct list_head list; //список точек
    монтирования
    struct rw_semaphore sem;
}

```

FILE - отдельная структура для работы с библиотечными функциями.



Сокеты

Сокет - абстракция конечной точки (конечная точка связи или коммуникаций)

На транспортном уровне сокет описывается тремя параметрами:

- family - определяет природу взаимодействия, включая формат адреса (address family)
- type - определяет тип сокета, который определяет характер взаимодействия
- protocol

Эти три параметра передаются системному вызову `socket()`

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

family:

- AF_UNIX - определяет сокеты локальной связи процессов
- AF_INET - семейство протоколов TCP/IP основаны на протоколе интернета v4 (IPv4)
- AF_INET6 - TCP/IP основаны на протоколе интернета v6 (IPv6)
- AF_IPX - IPX
- AF_UNSPEC - неопределенный домен

type:

- SOCK_STREAM - Передача потока данных с предварительной установкой соединения. Обеспечивается надёжный канал передачи данных, при котором фрагменты отправленного блока не теряются, не переупорядочиваются и не дублируются.

- SOCK_DGRAM - Передача данных в виде отдельных сообщений (датаграмм). Предварительная установка соединения не требуется. Сообщения могут теряться в пути, дублироваться и переупорядочиваться.
- SOCK_RAW - Этот тип присваивается низкоуровневым (т. н. "сырым") сокетам. protocol - протокол для определенного вида сокетов.
 - для SOCK_STREAM - TCP
 - для SOCK_DGRAM - UDP
 - 0 - протокол будет выбран по умолчанию

Пример (сокеты для локальной связи процессов)

```
#include <sys/socket.h>
#include <sys/un.h>
unix_socket = socket(PF_UNIX, type, 0);
error = socketpair(PF_UNIX, type, 0, int *sv);
```

Адресация сокетов

Прежде чем передать данные через сокет, его необходимо связать с адресом в выбранном домене. Т.е. нужно идентифицировать процесс, с которым необходимо взаимодействовать.

Идентификация: идентификация сетевого узла с помощью сетевого адреса
 идентификация конкретного процесса с помощью номера службы

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

Структура struct sockaddr:

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

В зависимости от назначения программы вместо sockaddr используют sockaddr_in и sockaddr_un (internet и unix соответственно):

```
struct sockaddr_in {
    sa_family_t sin_family;      //с е м е й с т в о  а д р е с о в
    unsigned short int sin_port; //н о м е р  п о р т а
    struct in_addr sin_addr;     //I P - а д р е с
    unsigned char sin_zero[8];
}
```

```
struct sockaddr_un {
    sa_family_t sun_family;      //с е м е й с т в о  а д р е с о в
    char sun_path[108];         //и м я  ф а й л а
}
```

Порядок байтов - характеристика аппаратной платформы, которая определяет порядок следования байтов в длинных типах данных, таких как целые числа.

Существуют два порядка следования байтов:

BIG ENDIAN - обратный порядок ($n \rightarrow n+1 \rightarrow n+2$) - network byte order (TCP/IP)

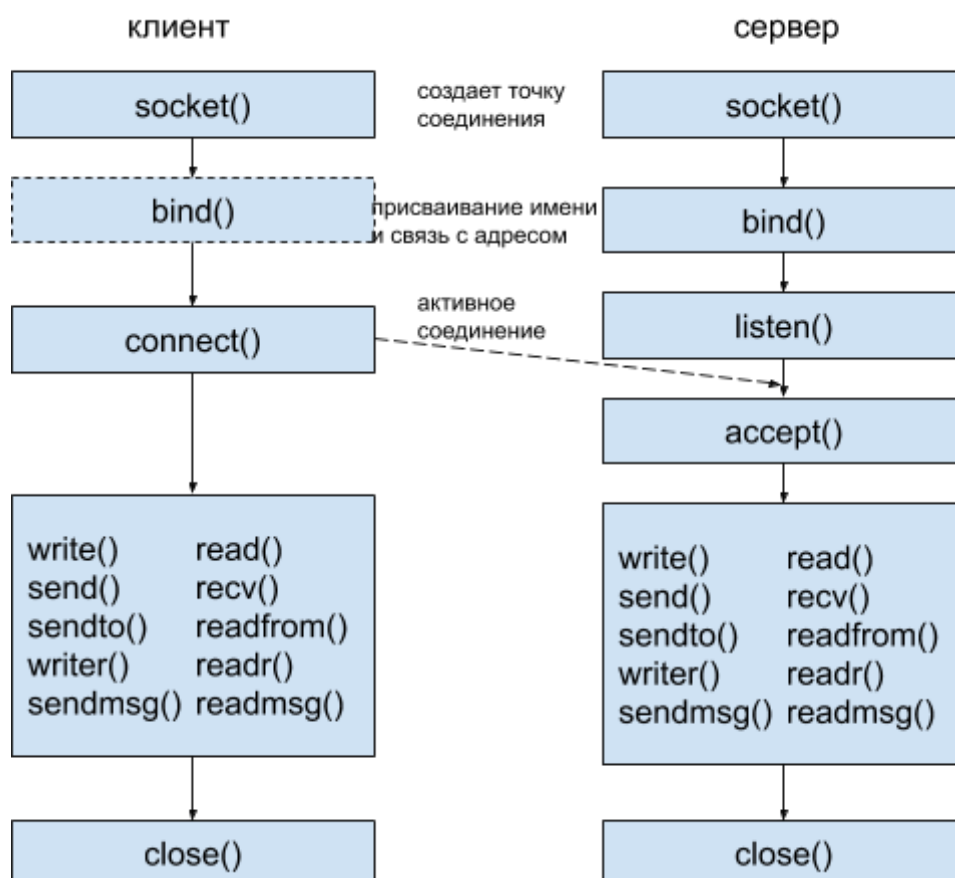
LITTLE ENDIAN - прямой порядок ($n+2 \leftarrow n+1 \leftarrow n$) - аппаратный

Если взаимодействие сокетов организуется на локальной машине - задумываться не надо, если через интернет - порядок байтов имеет значение.

Преобразования выполняются с помощью функций:

```
#include <arpa/inet.h>
htons(); //-> HostToNetworkShort
htonl(); //-> HostToNetworkLong
```

При работе с сокетами различают две роли - клиент и сервер



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

Сокет связывает с определенным адресом (на стороне сервера). Для интернет-сокета - IP адрес сетевого интерфейса локальной системы и номер порта.

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

Клиенты могут не вызывать **bind()** так как их адрес роли не играет. В таком случае адрес назначается автоматически.

Затем вызывается системный вызов **listen()** (сервером) для информирования ОС о том, что сокет должен принять соединение. Это имеет смысл только для протоколов TCP.

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

На стороне клиента вызывается сис. вызов **connect()**, который иницирует активное соединение (TCP) по указанному в качестве параметра адресу.

Для протокола без соединения (UDP) сис.вызов **connect()** может использоваться для указания значения адреса всех передаваемых пакетов.

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

Системный вызов **accept()** (на стороне сервера) используется для принятия соединения при условии, что ранее сервер получил запрос-соединение иначе процесс будет заблокирован до тех пор, пока не поступит запрос-соединение.

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

Когда соединение принимается - сокет планируется:

- исходный сокет остается в состоянии listen
- копия - в состоянии connect
- вызовом accept возвращается новый дескриптор файла для следующего сокета (дает серверу возможность принимать новые соединения без необходимости предварительно закрывать предыдущие соединения)

Ядро Linux предоставляет для работы с сокетами 1 системный вызов, который включает в себя все перечисленные системные вызовы:

```
<net/socket.c>
asmlinkage 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, a[2]);
            break;
        case SYS_CONNECT:
            err = connect(a0, (struct sockaddr*)a1, a[2]);
            break;
        ...
    }
```

```

    }
    return err;
}

```

int call - номер нужной функции

в <linux/net.h> определены соответствующие номера:

```

#define SYS_SOCKET      1
#define SYS_BIND        2
#define SYS_CONNECT     3
#define SYS_LISTEN      4
#define SYS_ACCEPT      5
#define SYS_GETSOCKNAME 6
#define SYS_GETPEERNAME 7
#define SYS_SOCKETPAIR  8
#define SYS_SEND         9
#define SYS_RECV        10

```

Если приложение вызывает функцию socket(), то происходит вызов sys_socket:

```

asmlinkage long sys_socket(int family, int type, int protocol) {
    int retval; // тот самый дескриптор, который
    возвращается
    struct socket *sock;
    ...
    retval = sock_create(family, type, protocol, &sock);
    ....
    return retval;
}

```

Структура сокета:

```

struct socket {
    socket_state state;           // состояние
    short type;
    unsigned long flags;
    const struct proto_ops *ops; // действия,
    подключа. протокола
    struct fasync_struct *fasync_list;
    struct file *file;           // т.к. сокет - спец
    файл
    wait_queue_head_t wait;      // очередь
}

```

На сокете определено 5 состояний:

SS_FREE	не занят
SS_UNCONNECTED	не соединен

SS_CONNECTING	соединяется в данный момент
SS_CONNECTED	соединен
SS_DISCONNECTING	разъединяется в данный момент

Пять моделей ввода/вывода

Эти модели рассматриваются с точки зрения программиста. Именно ПО формирует различные способы работы с аппаратурой.

I.Блокирующий ввод/вывод

блокирующий ввод/вывод (blocking IO) - синхронный

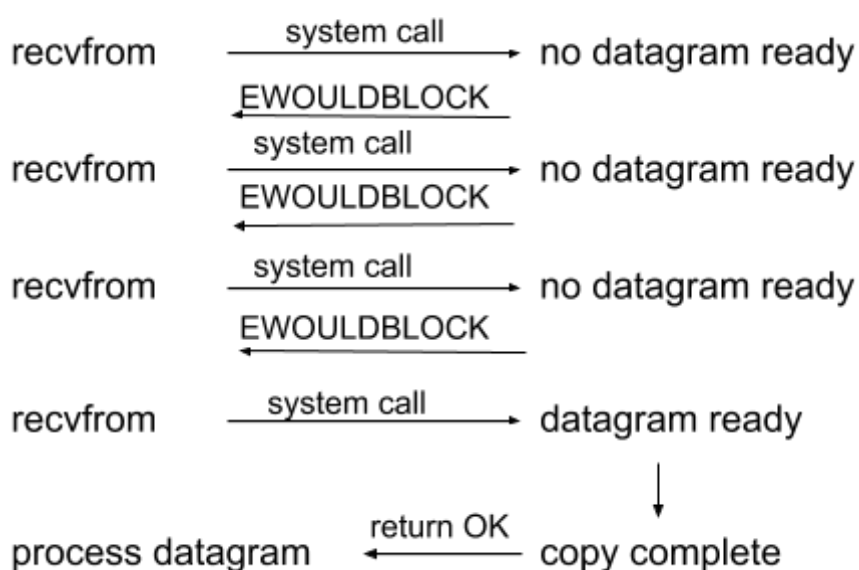


II.Неблокирующий ввод/вывод

неблокирующий ввод/вывод (polling) - синхронный

Приложение

Ядро



Процесс повторяет сис.вызов recvfrom ожидая сообщения о том, что данные получены.

Может быть установлен неблокирующий режим (т.е. recvfrom будет вызываться пока ядро не вернет данные для копирования)

Если данные не скопированы, ядро вызывает ошибку EWOULDBLOCK

III. Мультиплексирование ввода/вывода

Мультиплексирование ввода/вывода - асинхронный, блокирующий

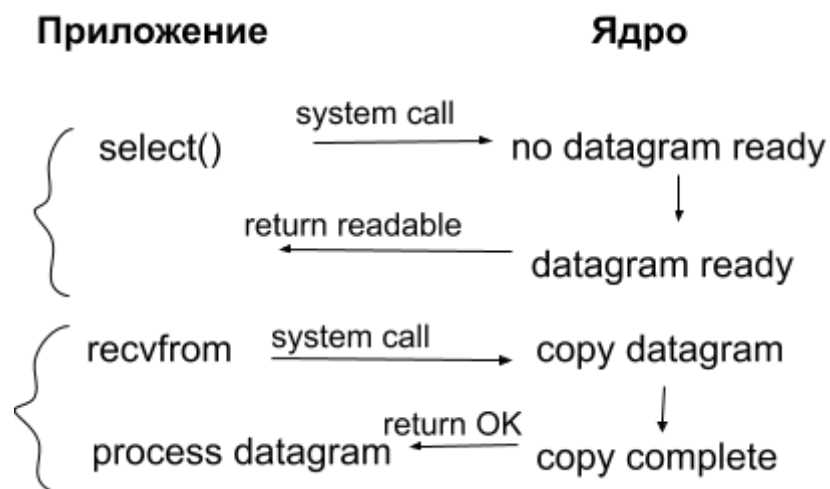
Имеет 2 тех. значения:

- мультиплексор
- коммутатор

Мультиплексор - устройство, объединяющее информацию, поступающую по нескольким каналам ввода и выдает итог по одному выходному каналу.

Два вида мультиплексирования:

1. временное (Time Division Multiplexing - устройству отводятся интервалы времени, в которые оно может использовать управляющую среду)
2. частотное



В результате вызова select процесс блокируется, ожидая готовности одного из многих возможных сокетов. Готовность - на какой-то сокет поступили данные.

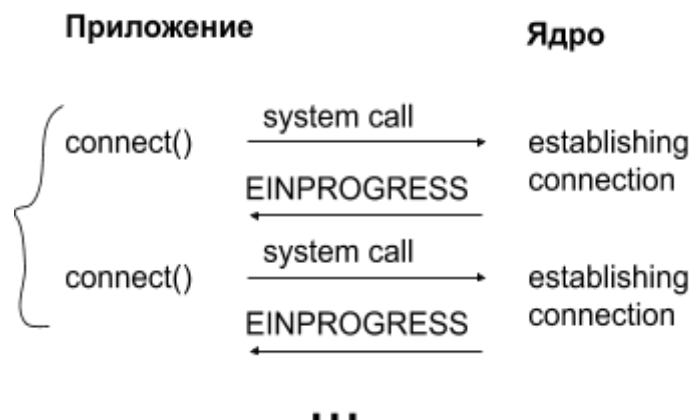
Мультиплексирование подразумевает, что опрашивается много сокетов. Это выполняет системный вызов select.

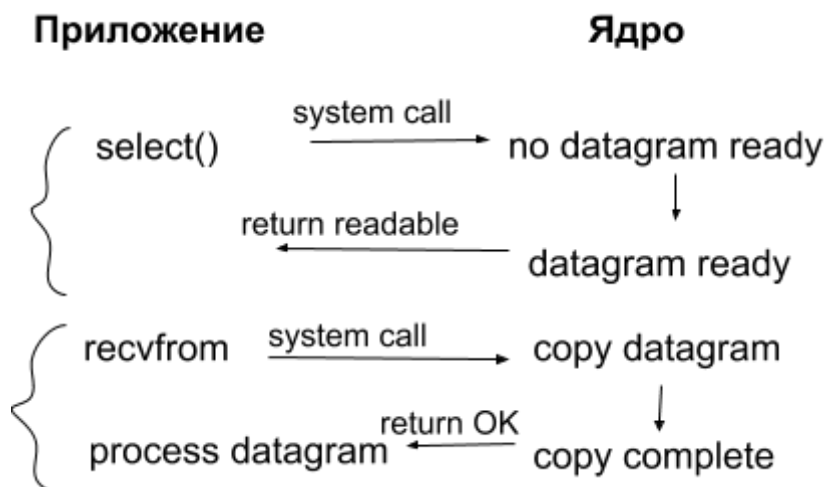
Затем, после того, как поступила информация о готовности сокета для чтения, процесс блокируется на время копирования данных из ядра в адресное пространство процесса.

Возвращается сообщение OK.

Преимущества: обрабатывается не один, а сразу много дескрипторов.

Время блокировки сис.вызовом select меньше.





`connect()` создает пул сокетов, который будет обрабатываться в цикле `select()`.

Системный вызов `select` блокирует процесс в ожидании готовности хотя бы одного сокета.

При мультиплексировании в цикле проверяются все сокеты и берется первый готовый. Пока обрабатывается один, могут подоспеть остальные. В результате сокращается время блокировки.

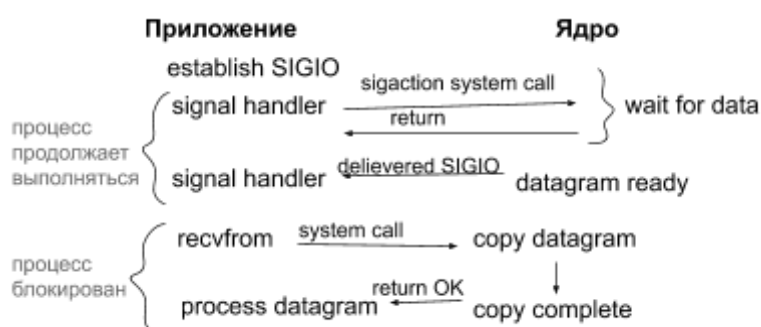
IIIА.

Запускается несколько потоков, в каждом из которых выполняется блокирующий ввод/вывод.

Недостатки дорогие потоки в Linux.

IV. Ввод/вывод, управляемый сигналами

Ввод/вывод, управляемый сигналами - signal driving IO - асинхронный



Необходимо установить обработчик сигнала SIGIO. Обработчик устанавливается сис.вызовом `sigaction`.

Результат возвращается сразу, приложение не блокируется, оно может продолжить выполняться.

Ядро отслеживает, когда данные будут готовы, после чего посылает сигнал SIGIO, который вызывает установленный на него обработчик.

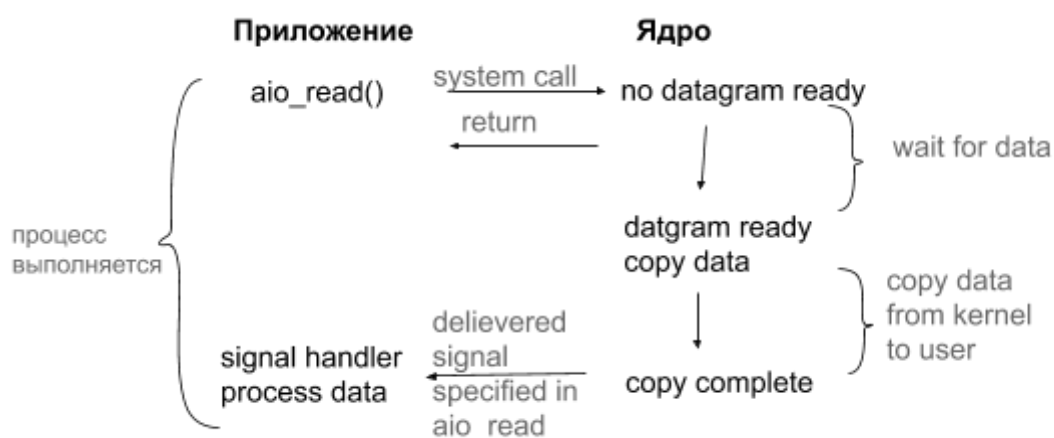
Вызов `recvfrom` можно выполнить либо в обработчике сигнала, который сообщает, что данные готовы, либо в основном потоке программы.

Ожидание может выполняться в цикле в основном потоке. Сигнал для каждого процесса может быть только один.

Во время обработчика сигнала процесс блокируется. Если в период блокировки сигналы доставляются несколько раз, то они теряются.
Если маска сигнала `sa_mask=NULL` то во время выполнения обработчика другие сигналы не блокируются.

V. Ассинхронный ввод/вывод

Ассинхронный ввод/вывод - ассинхронный, неблокирующий
Осуществляется с помощью специальных системных вызовов.



Ассинхронный ввод/вывод предполагает, что процесс может выполняться.
Необходимо получать ассинхронные события синхронно.
POSIX определяет ассинхронные функции ввода/вывода, они имеют приставку `aio` или `lio`
Параметры системного вызова: дескриптор, адрес буфера, размер буфера.
Ядру сообщается о начале операции и уведомляют приложение, когда вся операция из ядра в буфер будет завершена.

Отличие IV и V моделей:

- В модели IV ядро сообщает приложению, когда операция ввода/вывода может быть инициирована
- В V ядро сообщает приложению, когда операция ввода/вывода завершена.

Сравнение пяти моделей ввода/вывода

Блокирующий	Неблокирующий	Мультиплексирование	Управление сигналами	Ассинхронный
инициирован (блокирован) завершен	проверка ... проверка (блокирован) завершен	проверка (блокирован) готов инициирован (блокирован) завершен	уведомление инициирован (блокирован) завершен	инициирован (ожидание данных) (копирование) уведомление

	BLOCKING	NON-BLOCKING
SYNCHRONOUS	(I) blocking IO	(II) polling
ASYNCHRONOUS	(III) multiplexing IO	(V) aio
	(IV) signal driving IO	

Драйверы и прерывания

Драйвер может иметь один обработчик прерывания.

Если устройство использует прерывания, то драйвер устройства регистрирует один обработчик прерывания.

Для обработки прерывания драйвер может разрешить определенную линию прерываний IRQ:

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

irq - номер, прерывания, который будет обрабатывать обработчик.

handler - указатель на функцию обработчика прерывания, который обслуживает прерывание.

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

- irq - численное значение линии прерывания, которую обслуживает обработчик.
- dev_id - общий указатель на тот же самый dev_id, что задается в request_irq, когда обработчик был зарегистрирован.
- regs - структура, содержащая регистры процессора и состояние перед обслуживанием прерывания. Обычно используется для отладки.
- возвращаемое значение: IRQ_NONE - обработчик прерывания обнаруживает прерывание, которое его устройство не инициировало; IRQ_HANDLER - обработчик прерывания был вызван корректно и его устройство действительно вызвало прерывание.

irqflags - битовая маска либо NULL

- IRQF_SHARED - на одну линию прерывания можно зарегистрировать несколько обработчиков
- IRQF_PROBE_SHARED - устанавливается вызывающим, когда предполагается возможное возникновение проблем при использовании линии IRQ
- IRQF_TIMER - маскирует прерывание как прерывание от таймера
- IRQF_NOBALANCING - запрещает использовать это прерывание для балансировки IRQ

dev_name - текст, представляющий устройство, связанное с этим прерыванием. (Используется в /proc/irq и /proc/interrupts)

dev_id - используется для разделения SHARED линии прерывания.

возвращаемое значение -

- 0 - OK
- не 0 - ошибка

- E_BUSY - данная линия прерывания уже используется и не указан флаг IRQF_SHARED.

Медленные прерывания

top half <--> bottom half

Обработчик прерывания делится на 2 части:

- первая часть - обработчик прерывания
- вторая часть - выполнение на более низком приоритете в более простых условиях

Завершение обработки прерывания выполняется нижней половиной

Задачи верхней половины:

1. Так как обработчик верхней половины выполняется при запрещенных прерываниях, то обработчик возвращает управление обычным return
2. Должна обеспечить последующее выполнение нижней половины (поставить ее в очередь на выполнение)

В настоящее время есть 3 типа нижних половин:

- отложенные прерывания (soft irqs)
- тасклеты
- очереди работ

Отложенные прерывания

определяются статически во время компиляции ядра.

В файле <linux/interrupt.h> определена структура

```
struct softirq_action {
    void (*action)(struct softirq_action*);
    // поле с данными в версии ядра 2.6.37)
}
```

В файле <kernel/softirq.c> определен массив из 32 экземпляров структуры softirq_action

```
static struct softirq_action softirq_vector[NR_SOFTIRQS];
```

Индекс	Приоритет	
HI_SOFTIRQ	0	высокоприоритетные отложенные прерывания
TIMER_SOFTIRQ	1	таймеры
NET_TX_SOFTIRQ	2	отправка сетевых пакетов
NET_RX_SOFTIRQ	3	прием сетевых пакетов
BLOCK_SOFTIRQ	4	блочные устройства
BLOCK_IOPL_SOFTIRQ	5	
TASKLET_SOFTIRQ	6	тасклеты
SHED_SOFTIRQ	7	планировщик

HTIMER_SOFTIRQ	8	(не используется)
RCU_SOFTIRQ	9	(д.б. последним)

Когда ядро выполняет обработчик отложенного прерывания, то функция `action` вызывается с `softirq_action` в качестве аргумента.

`xxx_softirq -> action(xxx_softirq)`

(функции `action` передается вся структура, а не конкретное значение. Это обеспечивает возможность дополнения структуры без необходимости внесения изменений в каждый обработчик `softirq`)

Добавить новый обработчик можно только перекомпилировав ядро.

Для создания нового уровня `softirq` нужно:

- определить новый индекс отложенного прерывания, вписав его константу `XXX_SOFT_IRQ` в перечисление, (очевидно, на одну позицию выше `TASKLET_SOFTIRQ`)
- во время инициализации модуля должен быть зарегистрирован (объявлен) обработчик отложенного прерывания с помощью вызова `open_softirq()`, который принимает три параметра: индекс отложенного прерывания, функция-обработчик и значение поля `data`:

```
/* The bottom half */
void xxx_analyze( void *data ) {
    /* Analyze and do ..... */
}
void __init roller_init() {
    /* ... */
    request_irq( irq, xxx_interrupt, 0, "xxx", NULL );
    open_softirq( XXX_SOFT_IRQ, xxx_analyze, NULL );
}
```

- зарегистрированное отложенное прерывание должно быть поставлено в очередь на выполнение. Это называется генерацией отложенного прерывания. Обычно обработчик аппаратного прерывания (верхней половины) перед возвратом возбуждает свои обработчики отложенных прерываний:

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

- обработчик отложенного прерывания выполняется при разрешенных прерываниях процессора (особенность нижней половины), новые отложенные прерывания на данном процессоре запрещаются. Однако на другом процессоре обработчики отложенных прерываний могут выполняться. Обработчики `softirq`

должны быть реентерабельными, а критические данные должны использоваться монопольно.

Проверка осуществляется в следующих случаях: (какая проверка?? хз)

- при возврате из аппаратных прерываний
- в контексте потока ядра ksoftirqd (демона softirq)
- в любом коде ядра в котором явно проверяются и запускаются обработчики отложенных прерываний.

Независимо от метода выполнения используется do_softirq(), которая в цикле проверяет наличие отложенных прерываний

softirq никогда не вытесняет другой softirq
softirq может быть вытеснено только при аппаратном прерывании

Демон softirq - поток ядра каждого процессора, который выполняется когда в машине запущены отложенные прерывания.

Как правило, отложенные прерывания обслуживаются по возвращении из аппаратного прерывания.

Компьютер обменивается данными с устройствами используя IRQ

ОС прерывает выполнение текущей задачи и начинает адресовать прерывание. Когда прерывания приходят очень быстро, ОС не может закончить с одним до прихода другого. Такое может произойти когда сетевая карта получает пакеты в течение короткого времени. Тогда создается очередь, которой управляет ksoftirq

Тасклеты

тасклеты базируются на softirq. простые в использовании отложенные прерывания

Тасклеты - отложенные прерывания, для которых обработчик не может выполняться одновременно на нескольких процессах.

```
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;           //data для обработчика
}
```

state может принимать 2 значения:

- TASKLET_STATE_SCHED - тасклет запланирован на выполнение
- TASKLET_STATE_RUN - тасклет выполняется

count

- = 0 - тасклет разрешен и может выполняться если ждет выполнения
- != 0 - тасклет запрещен и не может выполняться

Тасклеты могут быть зарегистрированы статически и динамически <linux/interrupt.h>

Для статического создания тасклета определено 2 макроса:

```
// создает тасклет с count=0 -> разрешен
```

```
DECLARE_TASKLET(name, func, data)
// создает тасклет с count=1 -> запрещен
DECLARE_TASKLET_DISABLE(name, func, data)
```

При динамическом создании тасклета объявляется указатель на структуру `struct tasklet_struct`. Для инициализации вызывается функция:

```
int tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long),
unsigned long data);
```

`func` - функция обработчик тасклета. Прототип:

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

В обработчике тасклета нельзя использовать семафоры, так как тасклеты не могут переходить в состояние блокировки. Если в тасклете используется общие с обработчиком прерывания или с другим тасклетом данные, то доступ должен быть монопольным с использованием спинлоков.

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

Планирование тасклетов

Для того, чтобы тасклет смог начать выполняться, его надо запланировать с помощью одной из двух функций:

```
tasklet_schedule(struct tasklet_struct *t);
tasklet_high_schedule(struct tasklet_struct *t);
```

Запланированные тасклеты хранятся в 2 связанных списках: `tasklet_vec` и `tasklet_hi_vec` (высокоприоритетные тасклеты). Оба списка состоят из экземпляров структуры `tasklet_struct` т.е. отдельных тасклетов.

После того, как тасклет запланирован, он будет запущен только один раз, даже если он запланирован на выполнение несколько раз.

Определены следующие функции:

```
tasklet_disable(); //tasklet_disable_no_sync();
tasklet_enable();
tasklet_kill(); //ждет пока тасклет завершится,
затем удаляет его из очереди
tasklet_kill_immediate();//удаляет тасклеты из
очереди в любом случае
```


Сравнение отложенных прерываний и тасклетов

Отложенные прерывания	Тасклеты
<ul style="list-style-type: none"> используются для запуска самых критичных нижних половин обработчика прерывания обработчик не может переходить в состояние блокировки 2 обработчика могут выполняться параллельно 	<ul style="list-style-type: none"> имеют более простой интерфейс и упрощенные правила блокировок 2 тасклета не могут выполняться параллельно лучше использовать, если необходимо масштабировать на бесконечное число процессов.

Очереди работ

Тасклеты	Очереди работ
выполняются в контексте прерывания	выполняются в контексте специального процесса ядра - воркера
код тасклета должен быть атомарным	код очереди работ не должен быть атомарным так как код ядра может запросить, чтобы выполнение запланированной очереди работ было отложено на определенный интервал времени
выполняются на процессоре, на котором было выполнено аппаратное прерывание	выполняются на том же процессоре по умолчанию, но это можно поменять

Очереди работ имеют тип (<linux/workqueue.h>)

```
struct workqueue_struct
```

Очереди работ должны быть явно созданы до их использования. Для создания используется:

```
alloc_workqueue(char *name, unsigned int flags, int max_active);
```

name - имя очереди (но не создается потоков, которые его используют)

flags - определяет как будут выполняться очереди работ

max_active - ограничивают количество задач, которые выполняются одновременно на одном сри

Флаги:

- WQ_HIGHPRI - задание помещается в начало очереди
- WQ_UNBOUND - очередь, не привязанная к конкретному сри
- WQ_CPU_INTENSIVE - отказ от организации параллельного выполнения
- WQ_FREEZABLE - указывает, что данная очередь будет заморожена, когда система приостанавливается

- WQ_NON_REENTRANT - гарантирует, что разные задания в очереди будут выполняться параллельно

Чтобы поместить задачу в очередь работ нужно заполнить структуру

```
struct work_struct
```

Это может быть сделано во время компиляции статически с помощью макроса

```
DECLARE_WORK(name, void (*func)(void *)), /* void *data */;
```

name - имя структуры, которое должно быть объявлено как `struct work_struct *name`

func - функция, содержащая обработчик нижней половины

```
//делает более тщательную работу по
инициализации структуры. Если структура
определена в первый раз
INIT_WORK(struct work_struct *work, void (*func)(void *), void *data);
//делает ту же работу, но не инициализирует
указатели. Используется чтобы связать
work_struct с workqueue_struct
PREPARE_WORK(struct work_struct *work, void (*func)(void *), void
*data);
```

Если существует возможность, что структура уже представлена в очереди работ и ее надо изменить, то предпочтительнее `INIT_WORK`

Существуют две функции отправки работы в очередь работ:

```
int queue_work(struct workqueue_struct *queue, struct work_struct
*work);
int queue_delay_work(struct workqueue_struct *queue, struct work_struct
*work, unsigned long delay);
```

Возвращает 0 в случае успеха.

Функции выполняются в контексте потока, который называется `worker`. Такие потоки могут блокироваться, но нужно знать как блокировка повлияет на другие задачи в очереди работ.

Так как функции работают внутри потока ядра, у них нет доступа к адресному пространству пользователя.

Чтобы отменить незавершенный вход в очередь работ, нужно вызвать

```
int cancel_delay_work(struct work_struct *work);
```

Если функция вернет не 0, то вход был отменен до начала выполнения. Это гарантирует, что данный вход не будет иницирован после вызова функции `cancel_delay_work`.

Функция

```
void flush_workqueue(struct workqueue_struct *queue);
```

после возврата из нее, никакая из функций work не будет выполняться нигде в системе.

Ликвидировать очередь можно с помощью

```
void destroy_workqueue(struct workqueue_struct *queue);
```

Устройства

Устройства идентифицируются старшим и младшим номером устройства.

Устройства

- символьные - c
- блочные - b
- (сетевые)

Если в каталоге /dev выполнить команду ls -l, то получим перечень адресов и два номера: первый - старший номер, идентифицирует драйвер устройства; второй - младший идентифицирует конкретное устройство.

Для идентификации устройства в системе имеется тип dev_t (POSIX1), определенный в <sys/types.h>. Формат полей и их содержание не оговаривается. Для разных версий системы формат отличается.

для 32-битной: 12 бит - старший, 20 бит - младший.

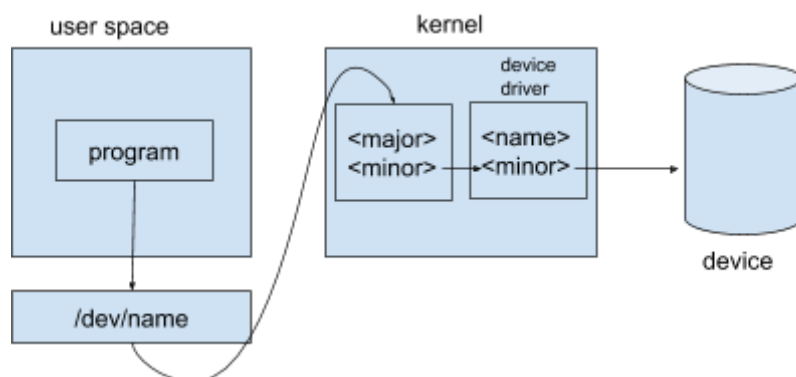
Старший и младший номера устройств можно получить с помощью макросов, что избавляет от необходимости задумываться как хранятся эти два номера:

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

```
struct stat {
    dev_t st_dev;      //описывает устройство, на
    котором находится файл - идентификатор
    устройства
    ino_t st_ino;
    mode_t st_mode;    //S_ISREG, S_ISDIR, S_ISCHR, S_ISBLK, S_ISFIFO,
    S_ISLINK, S_ISSOCK
    ...
    dev_t st_rdev;     //устройство, которое
    представляет этот файл (inode)
}
```

Функция makedev комбинирует номера чтобы произвести идентификатор устройства, который возвращает эта функция.

```
dev_t makedev(unsigned int maj, unsigned int min);
```



Драйверы

Драйвер - программа или часть кода ядра, предназначенная для управления конкретным устройством.

Обычно, драйверы устройств содержат последовательность команд, специфических для данного устройства. Имеют разные точки входа в зависимости от действий, которые выполняются на устройстве.

Драйверы Linux:

1. Драйверы, встроенные в ядро
автоматически распознаются системой и становятся доступными приложению
 - материнская плата
 - последовательные и параллельные порты
 - контроллеры IDE
2. Драйверы, реализованные как загружаемые модули ядра
используются для управления устройствами
 - SCSI адаптеры
 - звуковые и сетевые карты

Файлы модулей ядра располагаются в `/lib/modules`. При инсталляции системы задается перечень модулей, которые будут подключены автоматически на этапе

загрузки. Список хранится в `/etc/modules`. В `.conf` находится перечень опций для таких модулей

3. Драйверы, код которых поделен между ядром и специальной утилитой.
 - у драйвера принтера ядро осуществляет взаимодействие с параллельным портом, а демон печати `lpd` осуществляет формирование управляющих сигналов для принтера, используя для этого специальную программу.
 - драйвера модемов

Любое устройство, подключенное к системе регистрируется ядром и ему присваивается / выделяется специальный дескриптор в виде структуры

```
struct device {
    struct device *parent; //устройство, к которому
    //подключается новое устройство. Обычно шина
    //или хост-контроллер
    const char *init_name; //первоначальное имя
```

```

устройства
    const struct device_type *type;
    struct mutex mutex;    // для синхронизации
вызовов устройств
    struct bus_type *bus;  // тип шины, к которой
подключено устройство
    struct device_driver *driver;
    ...
#ifdef CONFIG_GENERIC_MSI_DOMAIN
    struct irq_domain *msi_domain;
#endif
    ...
    //DMA
    ...
    struct device_dma_parameters *dma_params; // структура
низкого уровня
    ...
}

```

Структура struct device_driver

```

struct device_driver {
    const char *name;
    struct bus_type *bus;
    struct module *owner;
    const char mod_name;    // имя модуля (для
встраиваемых модулей)
    ...
    // точки входа в драйвер
    // вызывается для запроса существования
конкретного устройства, если драйвер может
с ним работать
    int (*probe)(struct device *dev);
    // удаляет драйвер
    int (*remove)(struct device *dev);
    // отключает драйвер
    void (*shutdown)(struct device *dev);
    // переводит драйвер в режим сна
    int (*resume)(struct device *dev);
    // выводит драйвер из сна
    int (*suspend)(struct device *dev);
    ...
}

```

USB драйвер

Подсистема usb драйверов в Linux связана с драйвером **usb_core**, который называется usb ядром.

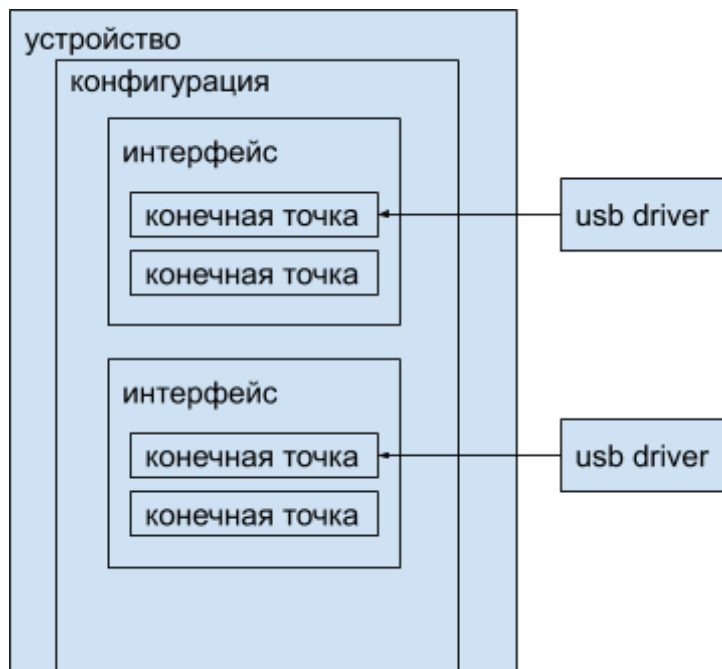
usb шина является шиной, в которой главной является **host**.

Host начинает все транзакции usb шины. Usb-шина построена по принципу иерархии. Взаимодействие usb-шины и usb-хоста выполняется со стороны host-a.

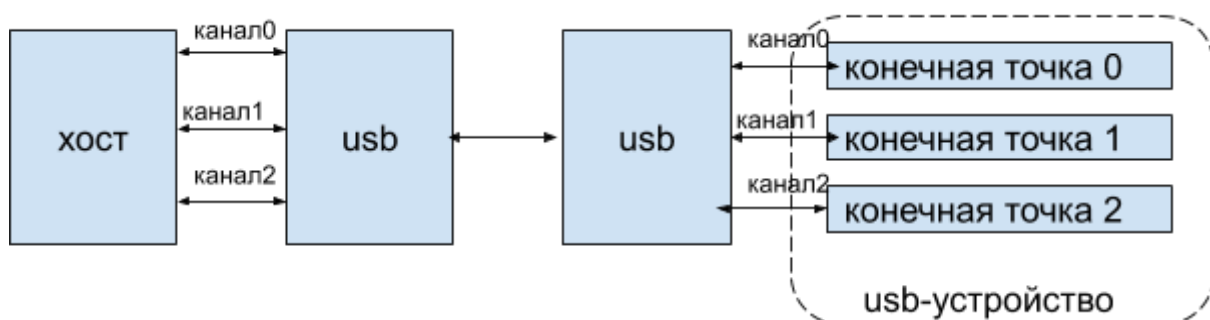
Первый пакет token генерируется хостом для указания что будет выполняться. При этом указывается адрес и конечная точка.

При подключении устройства драйверы считывают с него список конечных точек и создают структуры для взаимодействия с каждой конечной точкой устройства.

Конечная точка - программная сущность с уникальным айди и имеющая буфер с некоторым числом байтов для приема/передачи информации.



Совокупность конечной точки и структур данных ядра - канал (pipe).

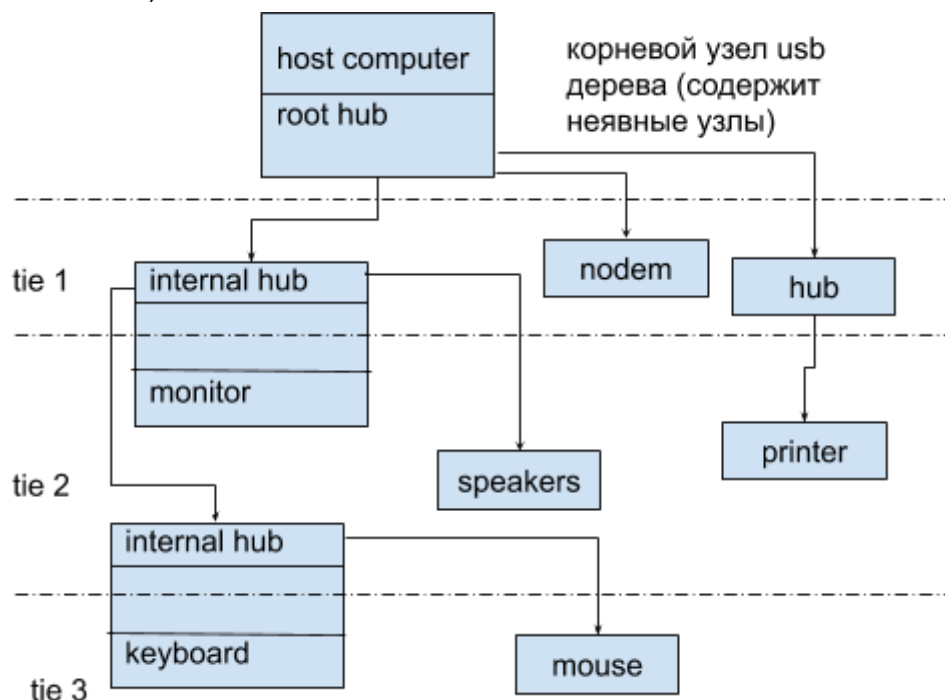


pipe - логическое соединение между хостом и конечной точкой. Потоки данных направленные - имеют определенное направление передачи данных. Перед отправкой данные собираются в пакет.

Типы пакетов

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

Usb-устройства подключаются к хабу (предусмотрены порты, которые заканчиваются разъемами).



Допускается до пяти уровней

Функция хаба - распространять сигнал или передавать данные к одному или более портам.

Хабы - активные электронные устройства. Многие устройства имеют встроенные хабы

Существует 4 типа передачи данных, которые определены в universal serial bus specification:

1. Control
2. Interrupt
3. Bulk
4. Isochronous

В результате конечные точки (а значит и каналы) относятся к одному из четырех типов:

- поточный
- управляющий
- изохронный
- прерывания

- Управляющий канал - для обмена с устройством короткими пакетами. Позволяет системе считать информацию об устройстве (в том числе коды производителя модели)

- Канал прерывания - доставляет короткие пакеты in и out без получения ответа подтверждения, но с гарантией времени доставки (не позже чем n мс). Канал используется с клавиатурой, мышью и пр.

- Isochronous каналы - доставляют пакеты без ответа подтверждения без гарантии времени доставки, но с гарантированной скоростью доставки (n пакетов)

- Поточные - дает гарантию доставки каждого пакета, не дает гарантии скорости и времени доставки. Используется в принтерах и сканерах.

Каждое подключенное к машине usb устройство представляется usb-core структурой

```
struct usb_device {
    int devnum; //номер устройства на usb шине
    char devpath[16]; //путь к файлу
    enum usb_device_state state; //состояние
устройства
    ...
    struct usb_host_endpoint ep0; //данные первой
конечной точки
    struct device dev;
    struct usb_device_descriptor descriptor;
    ...
    char * product; //версия устройства
    char * manufacturer; //производитель
    char * serial; //серийный номер
    ...
}
```

descriptor содержит данные об устройстве согласно протоколу.

Имеет поле с количеством возможных конфигураций. Каждая предоставляет набор операций взаимодействия.

Каждому интерфейсу соответствует драйвер. При подключении устройства usb-core подбирает для каждого его интерфейса соответствующий драйвер. Устройство может иметь несколько драйверов (принтер может работать как принтер и как сканер)

Основная структура, которая заполняется драйвером:

```
struct usb_driver {
    const char * name;
    int (* probe) (struct usb_interface *intf, const struct
usb_device_id *id);
    void (* disconnect) (struct usb_interface *intf);
    int (* unlocked_ioctl) (struct usb_interface *intf, unsigned int
code, void *buf);
    int (* suspend) (struct usb_interface *intf, pm_message_t
message);
    int (* resume) (struct usb_interface *intf);
    ...
    const struct usb_device_id * id_table;
};
```

name - имя драйвера, уникальное. Такое же как имя модуля

точки входа - probe, disconnect, suspend, resume

Если драйвер готов работать с устройством probe возвращает 0. Использует usb_set_infdata чтобы связать с интерфейсом. Если драйвер не соответствует устройству или не готов, то возвращается ENODEV.

Драйвер работает с интерфейсом:

```
struct usb_interface {
    struct usb_host_interface * altsetting;
    struct usb_host_interface * cur_altsetting;
    ...
    int minor;
    ...
    unsigned sysfs_files_created:1;
    ...
    struct device dev;          //является внутренней для
usb_interface
    struct device * usb_dev;
    ...
    struct work_struct reset_ws;
};
```

id_table - поле struct usb_device_id.

struct usb_driver формирует интерфейс. Драйвер использует id_table чтобы поддерживать "горячее подключение".

Эта таблица экспортируется макросом:

```
MODULE_DEVICE_TABLE(usb, ...)
```

Таблица id_table содержит список всех различных видов устройств, которые драйвер может распознать. Если таблица не создана, то функция обратного вызова никогда не будет вызвана.

Если разработчик хочет, чтобы драйвер вызывался всегда, то в этой таблице надо установить только поле driver_info

```
static struct usb_device_id usb_ids[] = {
    {driver_info=42};
    { }...
}
```

В этой структуре также есть поля:

- id_vendor - идентифицирует производителя устройства
- device_class
- device_subclass
- device_protocol
- interface_class
- interface_subclass
- interface_protocol

```

USB_DEVICE(vendor, product)
USB_DEVICE_VER(vendor, product, lo, hi)
USB_DEVICE_INFO(class, subclass, protocol)
USB_INTERFACE_INFO(class, subclass, protocol)

```

Регистрация USB драйвера

вызывается функция `usb_register`, в которую передается указатель на заполненную структуру

```

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

```

id_table:

```

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

```

После заполнения таблицы надо вызвать макрос

```

MODULE_DEVICE_TABLE(usb, pen_table)

```

```

static int __init pen_init(void) {
    return usb_register(&pen_driver);
}
static void __exit pen_exit(void) {
    usb_deregister(&pen_driver);
}
module_init(pen_init);
module_exit(pen_exit);

```

Для привязки драйвера к устройству вызывается функция `usb_register_dev`

```

int usb_register_dev(struct usb_interface *, struct usb_class_driver *);

```

Для отключения связи драйвера с устройством используется

```

void usb_deregister_dev(struct usb_interface *, struct usb_class_driver *);

```

Структура `usb_class_driver` содержит информацию о классе устройств, к которому принадлежит регистрируемое устройство

```

struct usb_class_driver {

```

```

    char *name; //шаблон имени устройства в /dev и
в /sys/devices
    const struct file_operations *fops;
    int minor_base; //начало диапазона младших
номеров устройств данного класса. 0 -
автоматическое выделение диапазона
    ...
}

```

Пример.

```

static struct file_operations fops = {
    .open = pen_open,
    .release = pen_close,
    .read = pen_read,
    .write = pen_write,
};
static struct usb_class_driver class = {
    .name = "usb/pen%d",
    .fops = &fops,
};

```

Вызов usb_register_dev выполняется в функции probe interface_to_usb.dev

```

static int ph_probe(struct usb_interface *iface, const struct
usb_device_id *id) {
    struct usb_device *udev = interface_to_usb.dev(iface);
    ...
    struct usb_host_interface *ifd = NULL;
    ...
    int ret = usb_register_dev(iface, &ph_class);
    if (ret < 0)
        return ret;
    return 0;
}

static int pen_probe(struct usb_interface *interface, const struct
usb_device_id *id) {
    int ret;
    device = interface_to_usbdev(interface);
    if (ret = usb_register_dev(interface, &class) < 0)
        err("Not able to get minor for this device");
    else
        printk(KERN_INFO "Minor obtained: %d\n", interface->minor);
    return ret;
}

```

В disconnect вызывается функция usb_deregister_dev

```
static void pen_disconnect(struct usb_interface *interface) {
    usb_deregister_dev(interface, &class);
}
```

Открытый код hid драйвера мыши лежит в linux/drivers/hid/usbhid/usbmouse.c
Если написан новый драйвер, то без выгрузки старого драйвера новый работать не будет (rm <имя драйвера>)

Управление буферами

Буфер - область оперативной памяти для промежуточного хранения информации. Используется для выравнивания скорости при обработке информации в процессе передачи данных

- между процессами
- между устройствами
- между процессом и устройством

Pipe синхронизирует выполнение чтения/записи.

Кеш-буфер - быстродействующая энергозависимая ОП в которую записывается копия команд или данных из более медленной памяти для ускорения процесса их дальнейшей обработки.

В базовой подсистеме ввода/вывода смешивать понятия буферизация и кеширование не следует!

Буфер содержит один набор данных. Может содержать единственный экземпляр данных в системе.

Кеш по своему назначению всегда содержит копию данных, которые хранятся где-то еще в системе.

Кеш TLB с адресами физических страниц, к которым было последнее обращение, эти же адреса есть в таблицах страниц.

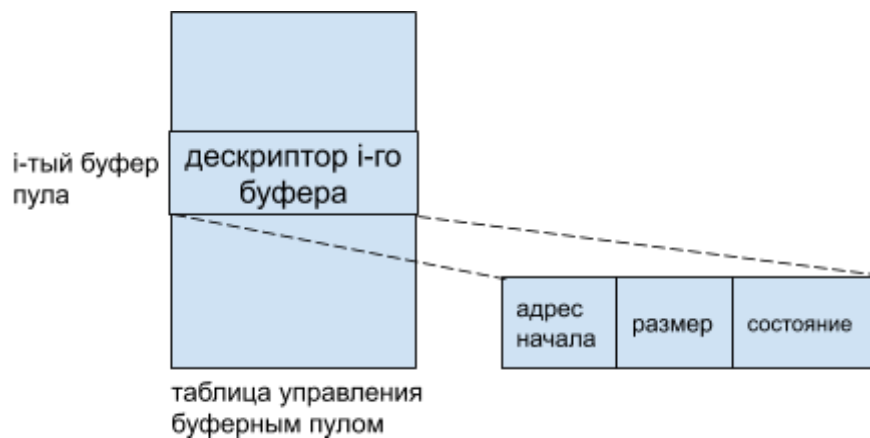
Методы буферизации

2 типа устройств:

- блок-ориентированные (диски)
- байт-ориентированные - передача неструктурированным потоком байтов (терминалы, принтеры, мыши и пр.)

Особую роль буферы играют в файловом вводе/выводе. Для буферизации файлового ввода/вывода выделяются несколько буферов - буферный пул (объединение)

Для управления буферами создается специальная таблица, каждый элемент которой описывает 1 буфер:



Буферный пул или отдельный буфер может быть создан:

- статически (во время выделения памяти задаче). Будет существовать во время выполнения задачи
- динамически (более экономно, создается перед началом обмена с устройством, освобождается когда обмен завершен)

Система должна предоставлять средства работы с буферами.

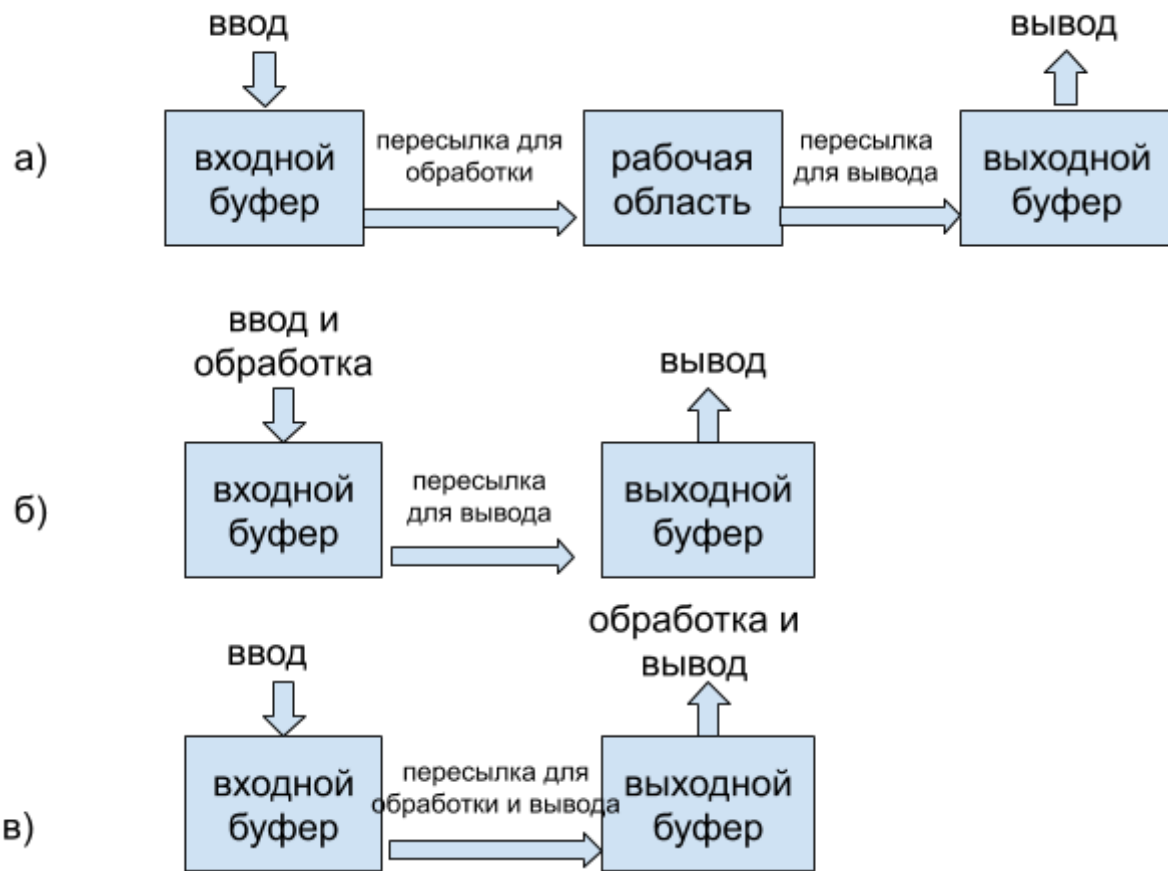
Буферный пул может быть сформирован (в общем виде):

- в программе может быть создан буфер с помощью специальной команды
- создание буфера с помощью макрокоманды (GETPULL, FREEPULL)
- получением от системы разрешения создать буферный пул при открытии набора данных

Для управления буферами самой ОС используются 2 схемы управления:

1. простая буферизация
2. обменная буферизация

Простая буферизация

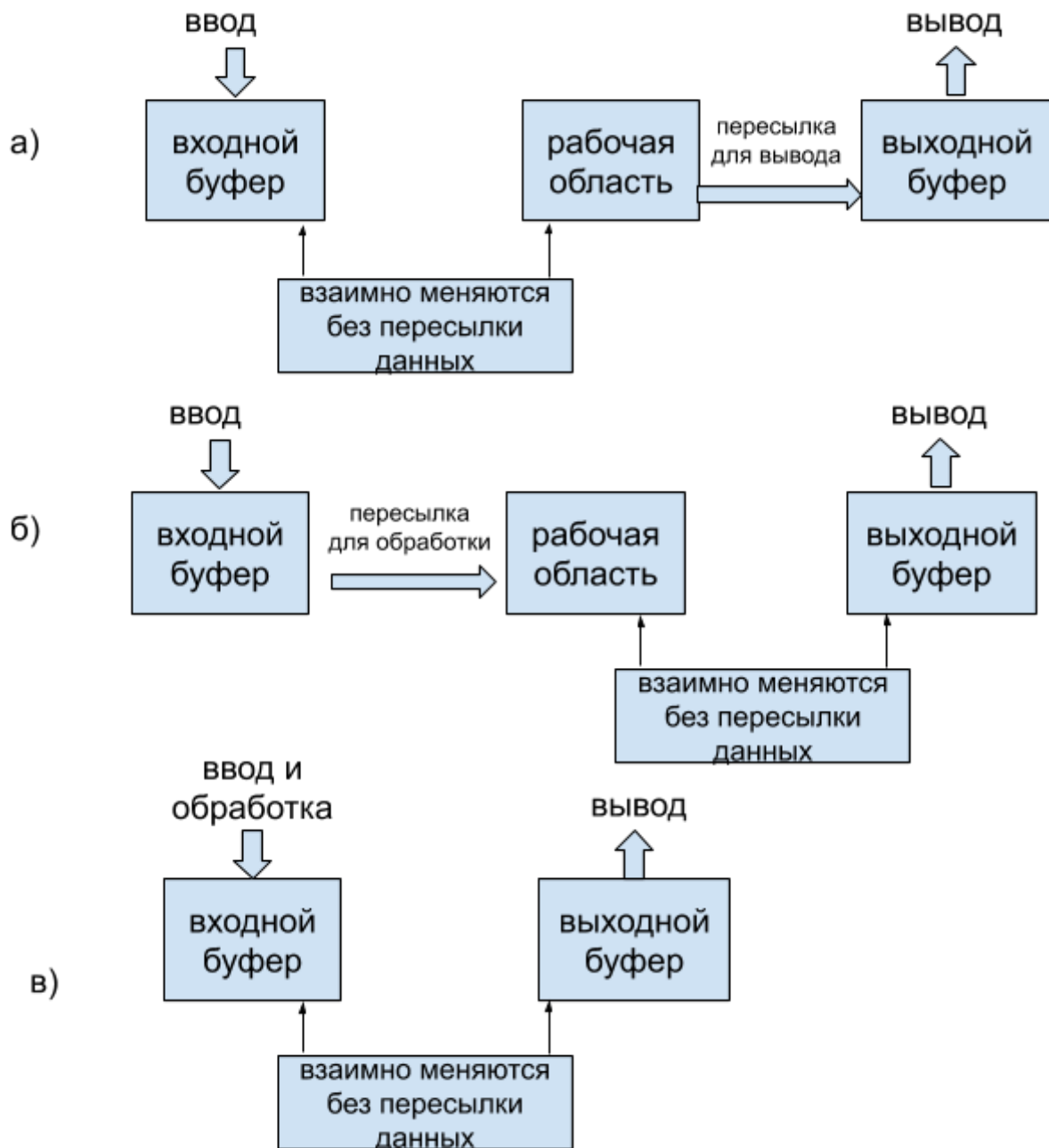


При простой буферизации выполняется большое число пересылок в оперативную память, что может привести к значительным временным затратам.

Связи между буфером и внешним устройством не меняются

Простая буферизация применяется когда объем данных относительно небольшой и обработка данных занимает довольно большое время

Обменная буферизация



Обменная буферизация полностью или частично устраняет пересылку между буферами.

(а - входной буфер принимает данные. после считывания входной буфер станет рабочей областью, а затем входными данными для выходного буфера)

Обменная буферизация требует соответствующих действий со стороны ОС:

- должна следить за обменной функцией среди буферов
- какую функцию выполняет в каждый момент каждая из областей
- динамическая смена адресов

Системные вызовы

часто определяются как запрос приложения на сервис системы.

В режим ядра систему переводит системный вызов INT.

В любой ОС системный вызов INT выполняется некоторой процессорной инструкцией, прерывающей последовательность команд приложения и передающей управление коду ядра (супервизору)

На различных ОС - это всегда команда типа int (платформа x86)

ms-dos	windows	linux	QNX	MINIX3
21h	2Eh	80h	21h	21h

В других архитектурах это не обязательно INT. М.б. svc, cmt, trap

Спин-блокировки в ядре Linux

реализуются посредством переменной типа spinlock_t (соответствует типу int)

Основной интерфейс spinlock_t содержит два вызова:

```
spin_lock(spinlock_t *sl);    // захват
spin_unlock(spinlock_t *sl);  // освобождение
```

Есть еще дополнительные функции:

```
// блокируют вход в критический участок
// только на локальном процессоре. Сохраняет
// регистр состояний в переменной flags
spin_lock_irqsave(spinlock_t *sl, unsigned long flags);
// состояния не сохраняет. Можно
// использовать если перед ее захватом
// прерывания разрешены, а перед
// освобождением их можно просто разрешить
spin_lock_irq(spinlock_t *sl);
// пытается установить блокировку, но
// одновременно предотвращает запуск нижних
// половин обработчиков прерываний
spin_lock_bh(spinlock_t *sl);
// освобождает спин-блокировку и разрешает
// прерывание, если при захвате блокировки
// прерывания были разрешены, в противном
// случае не разрешаются
spin_unlock_irqrestore(spinlock_t *sl, unsigned long flags);
// освобождает спин-блокировку и разрешает
// прерывания
spin_unlock_irq(spinlock_t *sl);
// освобождает блокировку и разрешает
// немедленную обработку нижних половин
spin_unlock_bh(spinlock_t *sl);
```

Есть дополнительные спин-блокировки

```
spin_trylock(); // процесс не переходит в активное
```



```
о ж и д а н и е
spin_islocked(); //получение текущего значения
спин-блокировки
```

Спин-блокировки чтения и записи

В системе возникают ситуации, в которых структуры данных активно считываются и редко изменяются. Например, dev_base - список зарегистрированных сетевых устройств. Этот список часто считывается и редко изменяется.

Такую ситуацию учитывает задача чтения-записи.

Тип **rwlock_t**:

- блокировку по чтению могут захватить несколько потоков
- блокировку по записи только один поток, чтение запрещено

Система предоставляет функции:

```
void read_lock(rwlock_t *rw);
void write_lock(rwlock_t *rw);
```

И дополнительные функции:

```
//read
void read_lock_irqsave(rwlock_t *rw, unsigned long flags);
void read_lock_irq(rwlock_t *rw);
void read_lock_bh(rwlock_t *rw);
//write
void write_lock_irqsave(rwlock_t *rw, unsigned long flags);
void write_lock_irq(rwlock_t *rw);
void write_lock_bh(rwlock_t *rw);
int write_trylock(rwlock_t *rw);

/* соответствующие unlock-и */
```

Спин-блокировки предполагают активное ожидание - процессы расходуют процессорное время, ожидая ресурсов.

Альтернатива - средства взаимного исключения, переводящие процесс в состояние блокировки, если он не может войти в критический участок.

Издержки: два переключения контекста. (переключение контекста требует выполнения большого объема кода, много большего чем объем кода, который потребовалось бы взаимноисключать с:)

Разумно использовать spin-блокировку, если **время ее удержания меньше длительности двух переключений контекста**

В многопроцессорных системах время спин-блокировок ~ времени планирования

В Linux спин-блокировка не рекурсивна, что может привести к самоблокировке.

Spin-блокировки можно использовать в обработчиках прерываний (семафоры использовать нельзя)

Если блокировка используется в обработчике прерывания, то перед ее захватом в

другом месте необходимо запретить все локальные прерывания на текущем процессоре.

Правила блокировки:

- нужно защищать данные, а не код
- блокировка должна быть связана с данными

2018 г. Gorokhova Irina. tlgrm: @irishaa