

10 февраля 2023 - Лекция 1

Рейтинговая подсистема

Рейтинг — из английского слова — иерархия, хранящая во вторичной памяти все...

Задача О/С — обеспечивать хранение данных и доступ к извлечённым фрагментам

Рейтинг — это последовательность фрагментов, хранимых во вторичной памяти (возможно, бесконечных).

К нему можно получить доступ (по ссылке).
Нужна иерархичность для обеспечения доступа, т.е. О/С занимает некоторое защищённое адресное пространство.

Рейтинговая система — порядок, определяющий способ организации, хранения, использования и доступа к фрагментам во вторичном носителе информации.

Вторичная память энергозависима.

Все запоминающие устройства являются блочными.

Устройства  символьные
блочные

Управление файлами — программы и процессы, взаимодействие с общим управляемым памятью (размножение файлов, борьба за рабочих).
Обычно основные функции — то ОС дополнительные — файловая подсистема

Доступ — open, read, write, remove, delete, ...

paradigm UNIX

В UNIX всё файл, а если нет, то процесс.

Есть специальные файлы ("больше, чем файлы") — pipe, socket, блоки устройства.

Файловая подсистема работает с регулярными файлами.

Каталог / директория — регулярный файл, содержащий список файлов.

7 типов файлов (no Paro):

- обычный
- d директория
- s softlink
- c special character
- b block
- s socket
- p named pipe

Существует стандарт Filesystem Hierarchy Standard (FHS),

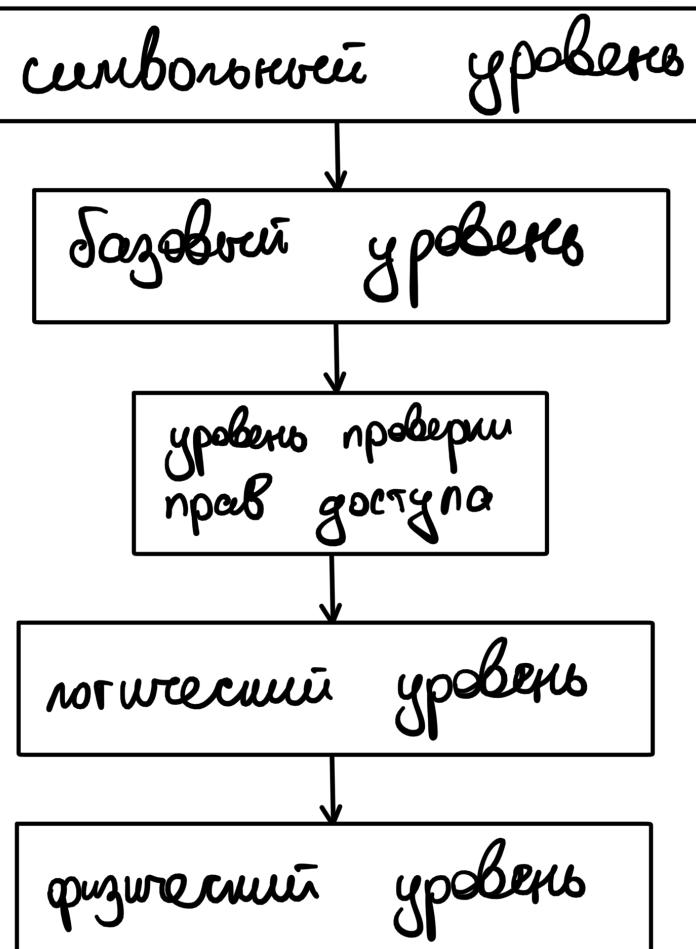
Он определяет структуру и содержание каталогов. Ubuntu его поддерживает

/ — корневой каталог, его ветви составляют единую файловую систему, расположенную по адресом каталога (путь или дисковому разделу). В ней должны находиться все компоненты для запуска системы.

Файл — контейнер индивидуального сценария, содержащая информацию.

Задачи файловой системы

1. Использование файлов
2. Обеспечение программного взаимодействия пользователей с файлами пользователями или приложениями
3. Обработка логической модели на организационную структуру данных по соответствующим правилам
4. Обеспечение надежного хранения, доступа, защиты от несанкционированного доступа.
5. Обеспечение совместного использования файлов



Символический уровень (работа с символами, изменение остатков при помещении символов из-за человеческого оператора), он же уровень изменения остатков.

Стандарт FHS поддерживает иерархию директорий.

Базовый уровень. Имя не является идентификатором (это просто hardlink). Идентификатор — inode (index node). Структура inode содержит всю необходимую системе информацию, а ее головной идентификатор.

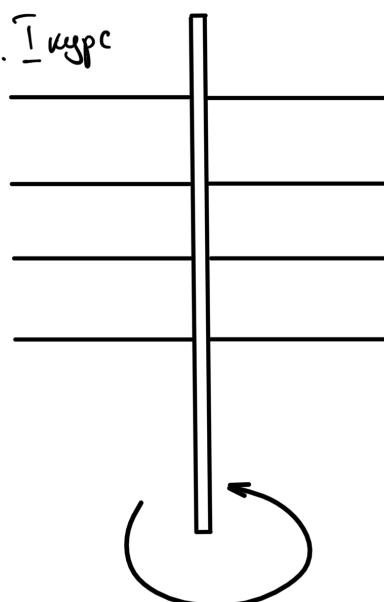
Проверка доступа — по разрешению для всех устройств.

Логический уровень. Работа с файлами (текстовыми, бинарными). Операции создания, записи. Байт — ориентированные файлы
Блок

Район \rightarrow символические = Байт-ориентированные
 \rightarrow блочного (т.к. 1 символ = 1 байт)

Физический уровень — диски, магнитные ленты

см. 1 курс



Про сектора и дорожки, орбита может быть на несколько секторах

Район вращается пространство диска вразброс.

Районовая система Linux

Позволяет работать с гесионами устройствами системами.

	PC UNIX	PC Linux
Технология	VFS / vnode	VFS
Идентификатор файла	Номер inode	Номер inode
	/root	

/bin /boot /etc /usr /var /dev /home

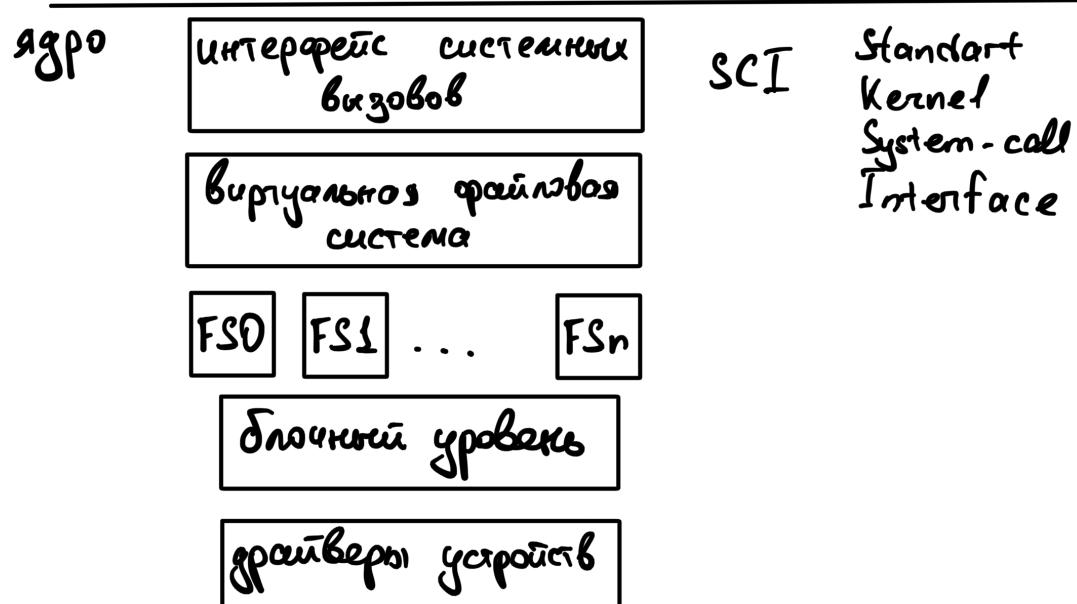
`ext`
`ext2` - встроенные файловые системы

`ext4`, `UFS`, `NFTS`, `XFS`, `MS-DOS`, `FAT`,
`EXT3`, `MPFS`, `NFS`

По сути, `VFS` - интерфейс, позволяющий работать с различными подсистемами.

В основе лежит приложение
идетерминированное

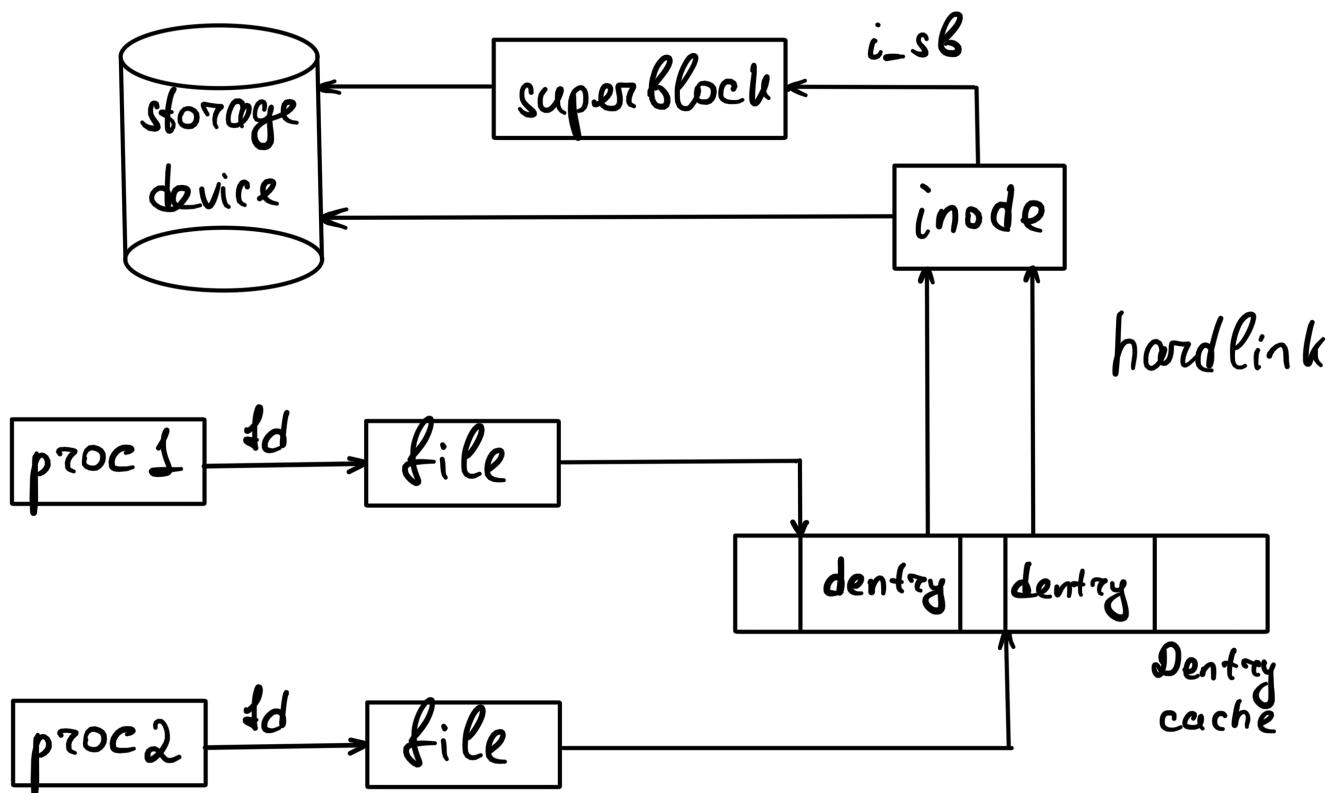
`GLIBC` ← (`GNUC`)
(`glibs`)



Чтобы в основе VFS:

1. Superblock — подмонтированный FS.
2. Dentry (directory entry) — элемент директории
3. inode — descriptor файла помимо дескриптор обеспечивает доступ в ядре
4. file открыт

Демонстрация связей структур:



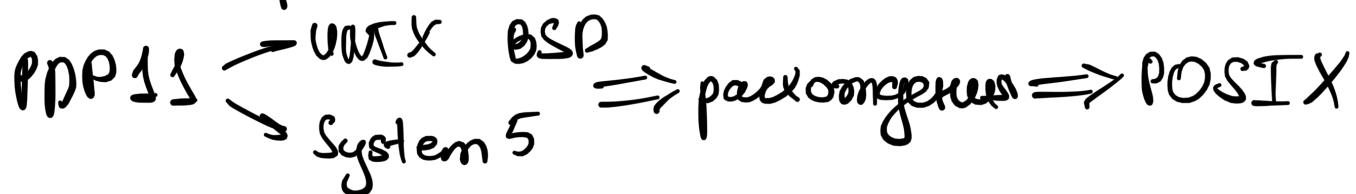
Каждый процесс имеет свою таблицу открытых файлов. До того, как процесс был запущен на стадии выполнения, от логике к диску.

В **struct task_struct** я указателя: **f** — оно программы и указатель на таблицу файлов

Любая ФС монтируется как поддиректория

16 февраля 2023 — Семинар 1 Сокеты

Сокеты — это средства взаимодействия независимых процессов

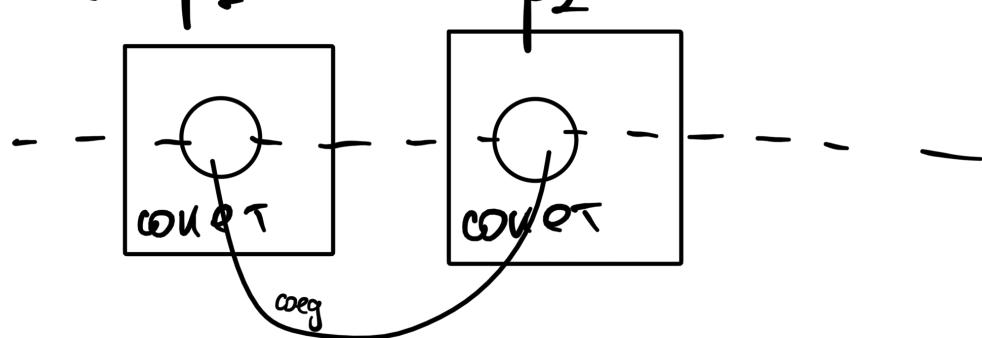


Парные сокеты — оптернатива pipe

У pipe буфер — 1 страница (т.к. передача
стражи оптимизирована)

Сокет — абстракция точки соединения

Типичная структура сокетов:



Сокеты — универсальное средство

взаимодействия независимых процессов.

Сокеты работают по модели "клиент-сервер"

Данные при пересыпале должны быть оформлены

— понятие **пакета**

Главные характеристики — **домен**.

domain

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

Не сигнатура, а синопсис. В технической литературе — прототип. Есть неформальные параметры (≠ фиктивные)

family

AF_UNIX
AF_INET [IPv4]
AF_INET6 [IPv6]
AF_IPX
AF_UNSPEC

type

SOCK_STREAM — сокеты потоков, определяют ориентированное по потокам, упорядоченное международное соединение между двумя сокетами

SOCK_DGRAM — определяют неподчиненную службу datagram где установление соединения, где пакеты могут передаваться без сохранения порядка (шагами вспомогательного пересыпала генерик)

SOCK_RAW — низкоуровневые сокеты

0 — по умолчанию — значение бара

IPPROTO_*,
например IPPROTO_TCP

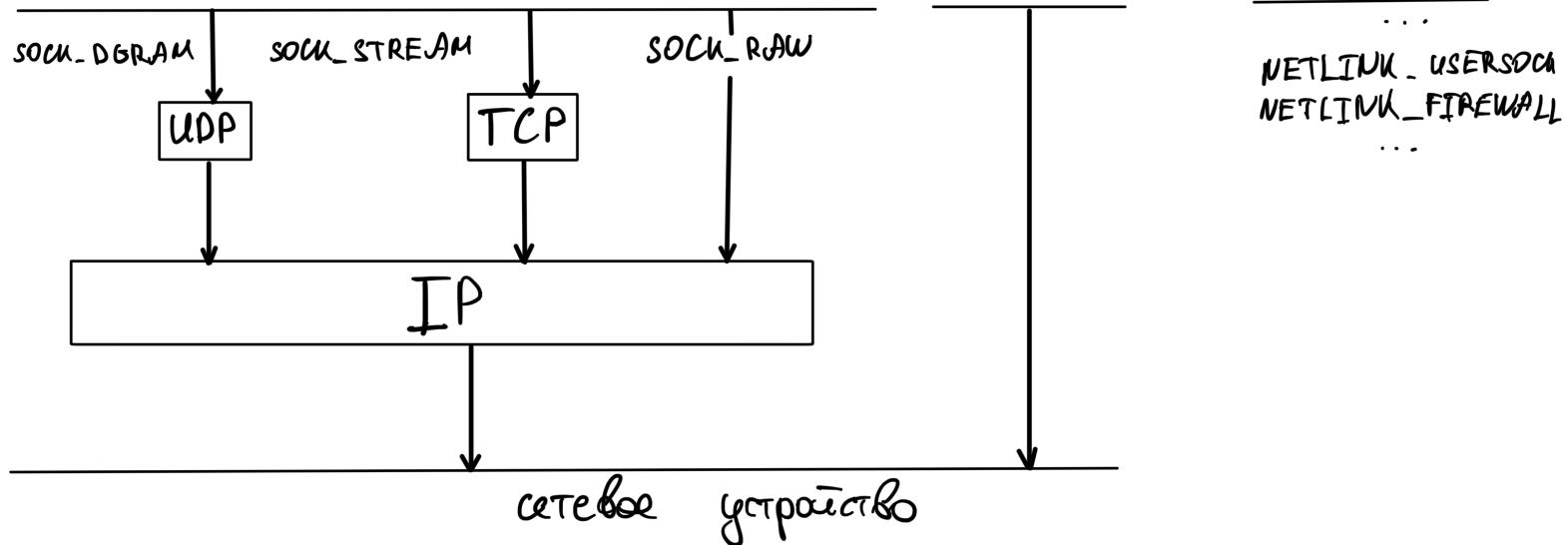
наш пересыпале информации от ядра к ядеру

BSD сокеты

PF_INET

PF_PHCET

PE_NETLINK



```

<net/socket.c>
asm linkage long sys_socketcall(int call, unsigned long *args);
{
    int err;
    ...
    if (copy_from_user(a, args, nargs[call])) {
        return -EFAULT;
    }
    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 = sys_connect((struct sockaddr *) a1, a[2]); break;
    ...
    default: err = -EINVAL; break;
    }
    return err;
}

```

sys_bind
 sys_socket
 sys_connect
 sys_socketcall

- сетевой стек

и. состоятелько по кругу

Перечень констант, связанных с сокетами:

#define	SYS_SOCKET	1
#define	SYS_BIND	2
#define	SYS_CONNECT	3
#define	SYS_ACCEPT	4
	:	

net/socket.c

```

asm linkage long sys_socket(int family, int type, int protocol)
{
    int retval;
    struct socket *sock;
    ...
    retval = sock_create(family, type, protocol, &sock);
    return retval;
}

```

struct socket - базовая структура сокета
 и наследует struct proc
 и task_struct

struct socket

```

{
    socket_state state;           cn. працює
    short type;                  cn. буна
                                гн. циклическим
                                гостини?
    unsigned long flags;          на розсортуванні
                                по в. куркові?
    const struct proto_ops *ops;   cn. асеміхронного
    struct fasync_struct *fasync_list; запуска
    struct file *file;           AF_UNIX номін.
                                - може конф-операції
                                виконувати через
    struct sock *sk;             іноді
    main_queue_head_t wait;      очікування
}

```

State

SS_FREE

SS_UNCONNECTED

SS_CONNECTING

SS_CONNECTED

SS_UNCONNECTING

Адреса сокетів

гн. сетей IPv4

(гн. IPv6 — sockaddr_in6)

```

struct sockaddr
{
    sa_family_t sa_family;        генер. постарано
    char sa_data[14];             умін.
}

```

Дні сетей та розрізки

struct sockaddr_in

```

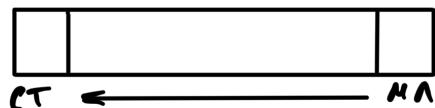
{
    sa_family_t sin_family;
    unsigned short int sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero[sizeof(struct sockaddr) -
                           sizeof(sa_family_t) -
                           sizeof(uint16_t) -
                           sizeof(struct in_addr)];
}

```

Дні сетей ножкою

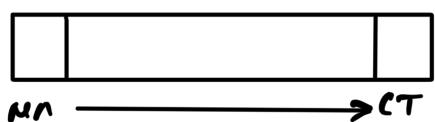
Порядок даних

Аппаратний
(прямий)



-host byte order / little
endian

Сетевий
(обратний)



-network byte order / big
endian

Про лаб. раб. №1

Разработать применение для взаимодействия процессов на одинаковых машинах через сокеты AF_UNIX, тип. SOCK_STREAM, модель "клиент-сервер"

Сокеты AF_UNIX реализуются в пространстве основных имен, т.е. процесс для взаимодействия использует спец. один (см. ls -al) — struct sockaddr

1 вариант — клиент → сервер

2 вариант — клиент ⇌ сервер

SOCKSTREAM
+ connect

2 марта 2023 — Семинар 2

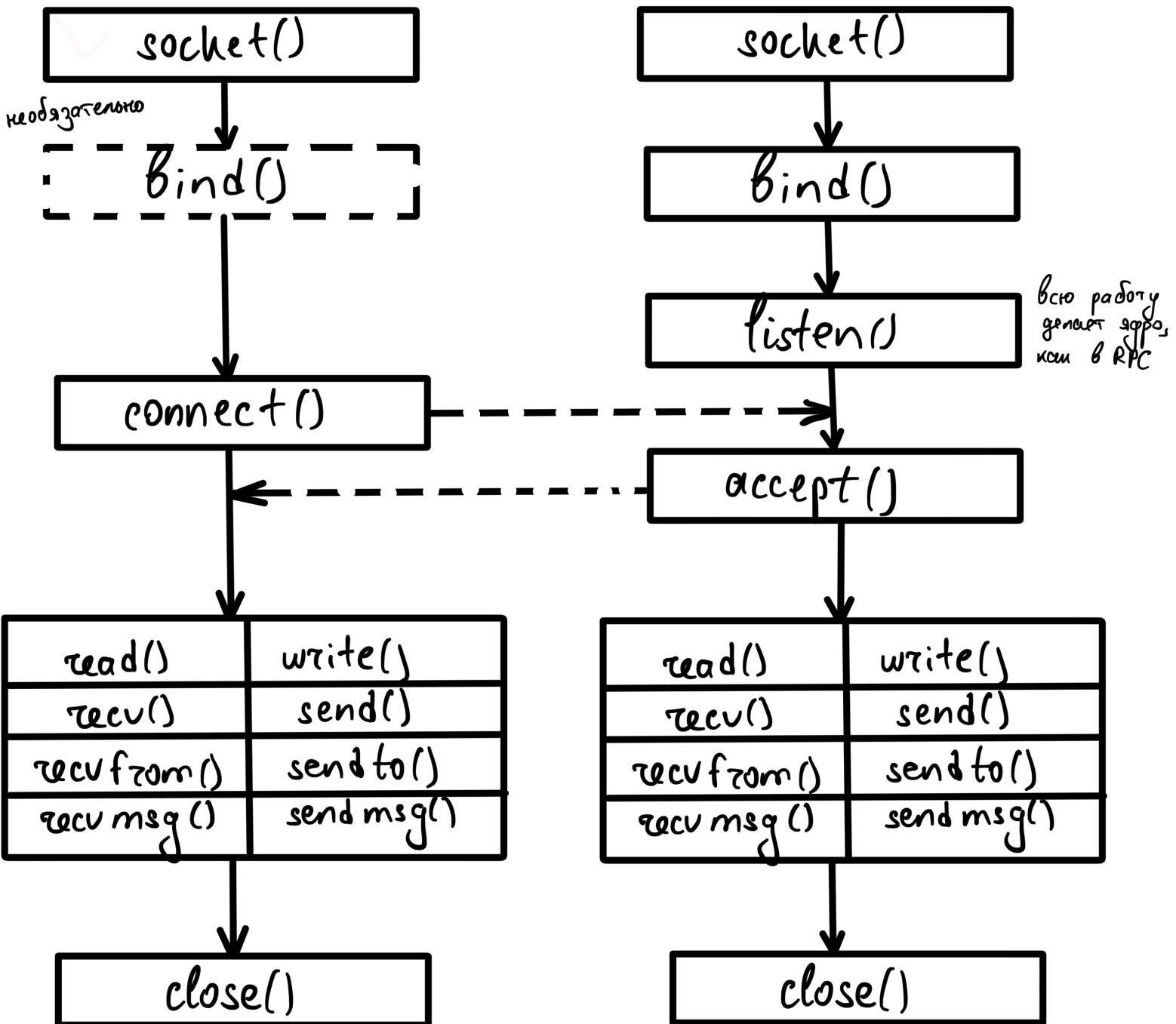
для курсовой

Про сокеты пишут Маргин, Борисов, ibm.com, книга Стивенса "Network" — самая полная по сокетам UNIX

Системные вызовы для сокетов (сетевой стек):

Клиент

Сервер



`socket()` — это создание демонитора открытого файла (это `AF_UNIX`) типа `struct FILE`

`Bind()` — это привязка имени файла / адреса ????

```
int bind ( int sockfd, struct sockaddr *addr, int addrlen )
```

Клиенты могут не вызывать `bind`, т.к. их адреса берутся по умолчанию — тогда их адреса назначаются автоматически.

lis-en — это инкорпорированное слово, что означает **создание**, т.е. что сервер переходит в состояние **насыщенного ожидания**.

listen имеет смысл для TCP. Его прототип:

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

На стороне клиента — активное соединение, которое инициируется **клиентом connect()**

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

Помимо состояний **клиента** (listen), есть состояния протокола (TCP), они же противоречат, а дополняют друг друга. TCP — тройное рукоопомощие — это слабое место (klient не получает ответ, соединение заливается — см. полуотрицательное соединение — сн. запрос на соединение отбрасывается, называемое **DDoS** атакой)

Готовый к соединению сервер вызывает **accept()**, который создаёт **копию** исходного союза, который продолжает слушать соединение, а копия обслуживает клиента.

```
int accept(int sockfd, void *addr, int addrlen)
```

Картина из последней книги (аналогична)

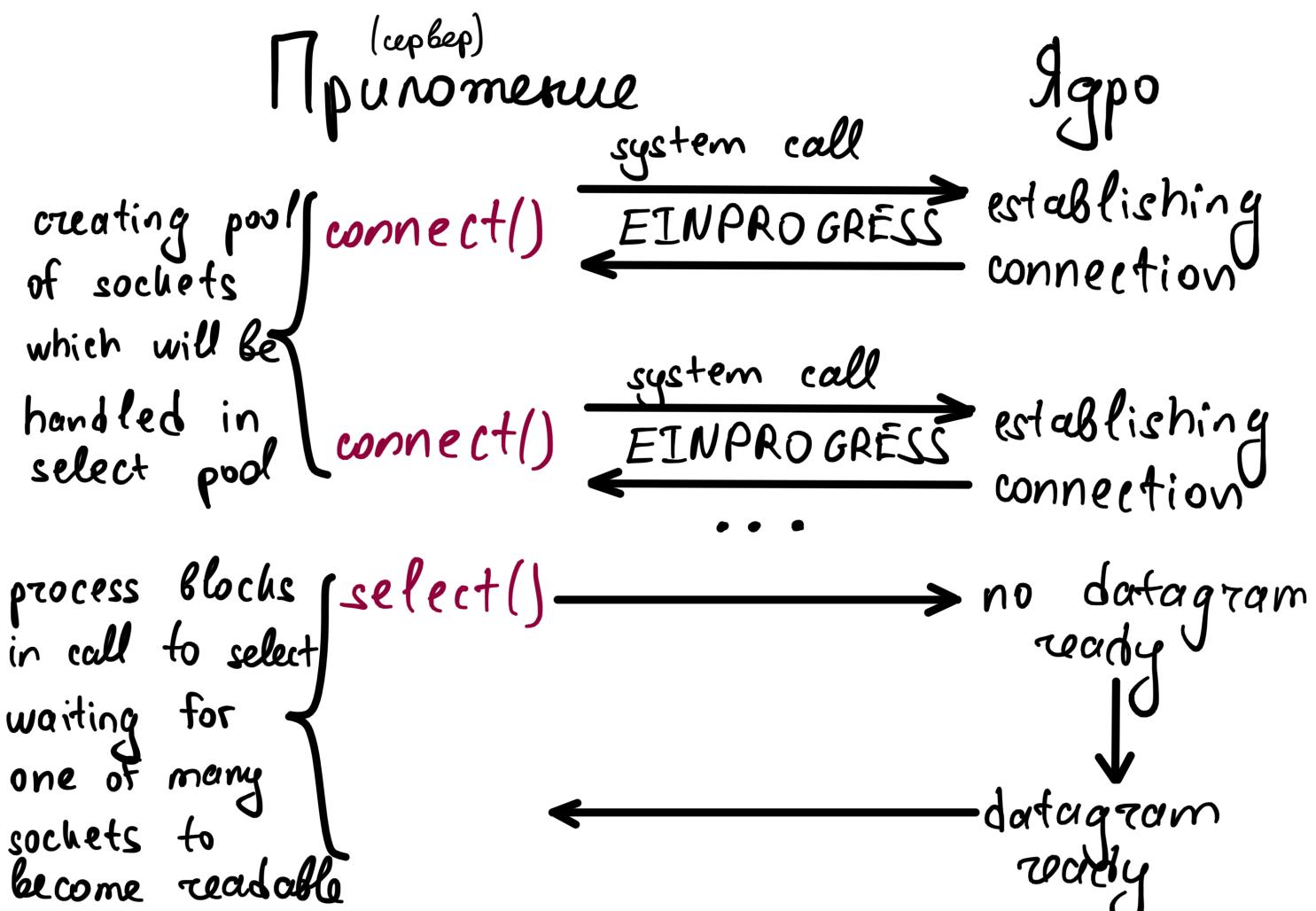
код программы более, то чад и write
в программе нопсигн)

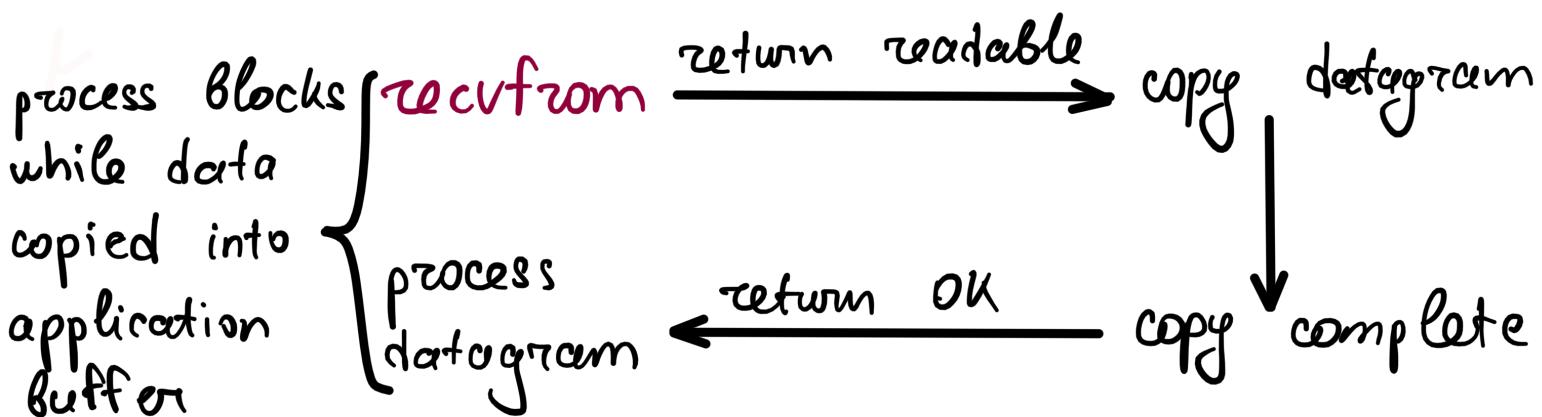
Мультиплексирование (на примере сокетов)

Файлы с интерактивной многопоточностью на сервере

Могут входить-выходить с мультиплексированием:
(последний будет на эзажите)

select	poll	предпочтительнее	pselect(..., NULL)
pselect	epoll	при этом	 select(...)





Пример — у себя можно было сделать запреты на fork:

unp.h — незнакомая функция

```

int main(int argc, char **argv)
{
    int listenfd, connfd;
    pid_t childpid;
    socklen_t clilen;
    struct sockaddr_in cliaddr, servaddr;
    listenfd = accept(AF_INET, SOCK_STREAM, 0);
    Bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET
    servaddr.sin_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT); 9877 — понять и почитать
    bind(listenfd, (SA*)&servaddr, sizeof(servaddr));
    listen(listenfd, LISTENQ);
    for (;;) {
        clilen = sizeof(cliaddr);
        connfd = accept(listenfd, (SA*)&cliaddr, &clilen);
        if ((childpid = fork()) == -1)
            {
                close(listenfd);
                str_echo(connfd);
                exit(0);
            }
    }
}

```

```
        close(connfd);  
    }  
    return 0;  
}
```

Распределенное системное

Всё рассказанное есть в мануале (см. *man proc*)

(изучаюся) т.к. все подмонтируются, всё
 происходит "на лету" при /proc

Виртуальная ОСИЛ. система *proc*

Позволяет приложению и разработчикам

в реальном пользователе получать иког-то

о процессах и ресурсах системы.

Монтируется при запуске системы
(или и другие Виртуальные ОСИЛ. системы,
предназначенные, например для длительного
хранения данных)

Не "создаём структуру" а "заполняем поле"

proc не является монтируемой и представ-
ляет, по сути, интерфейс ядра, который

использовать как основного состава для
того, чтобы можно было использовать
интерфейс стандартных РС.

Необходимо иметь соответствующие права.

По умолчанию запись и чтение файлов
в директории процесса разрешены только
владельцу

Директории:

/proc /<PID>

self

для пады можно
использовать модуль
программу, например,
курсок по КГ

Базовая таблица

10 марта 2023 — Лекция 2

Интерфейс VFS базируется на структурах:

- 1) struct superblock
- 2) struct inode
- 3) struct dentry
- 4) struct FILE

superblock описывает подмонтированную РС (она одна

у которой PC), содержит данные для доступа и описания.

Хранит данные о том, какой блок диска или тег

inode — дескриптор описания. Имеет номер (инодн)

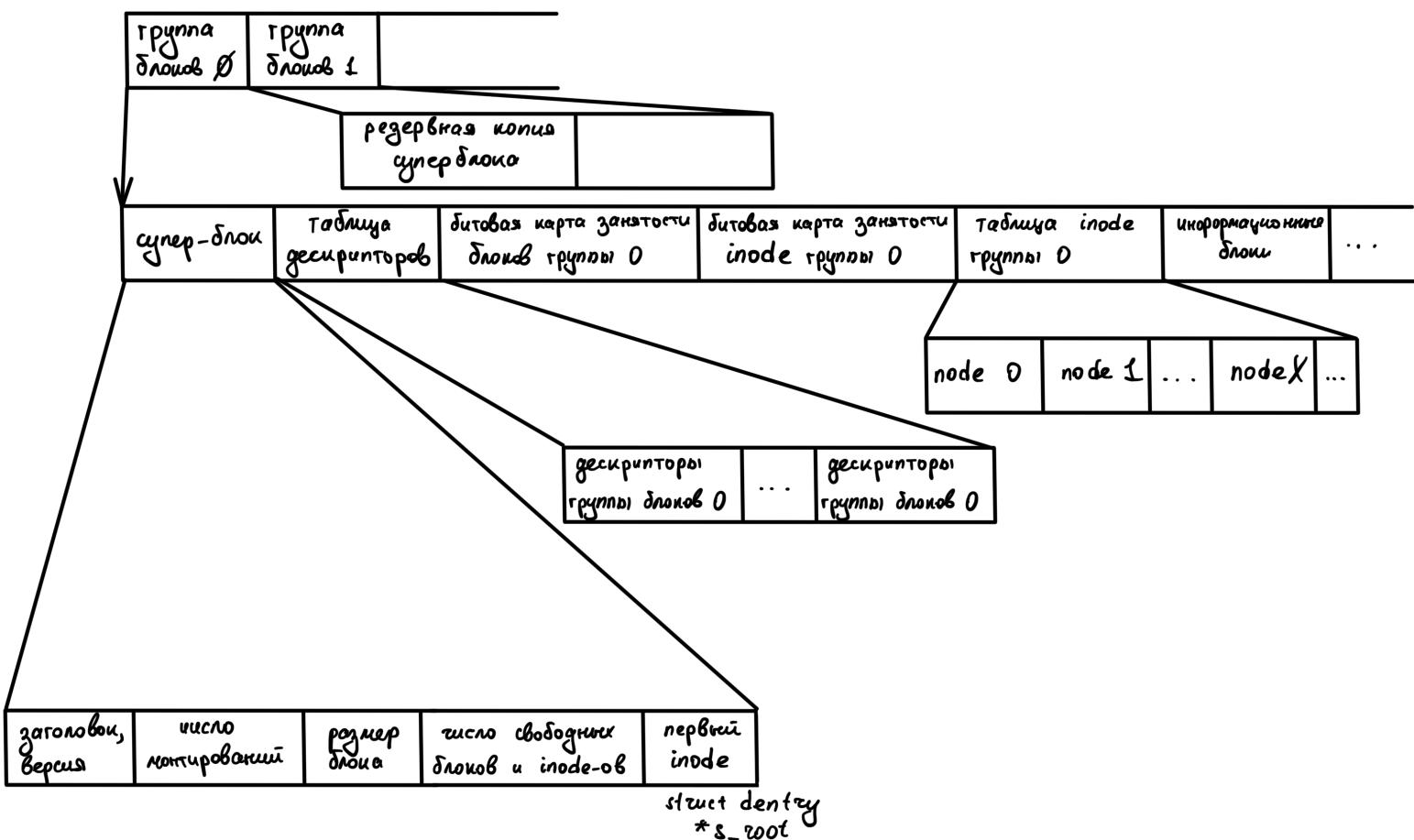
Если речь об обычных (регулярных) файлах, inode должен содержать информацию о блоках на диске.

На изданет можно привести распределенное
буферами с структурами

PC доступна = даёт доступ и каталоги и подсистемы
⇒ нечестно подмонтируется

Обычно все картинки для EXT (EXT2)

Родитель жёсткого диска



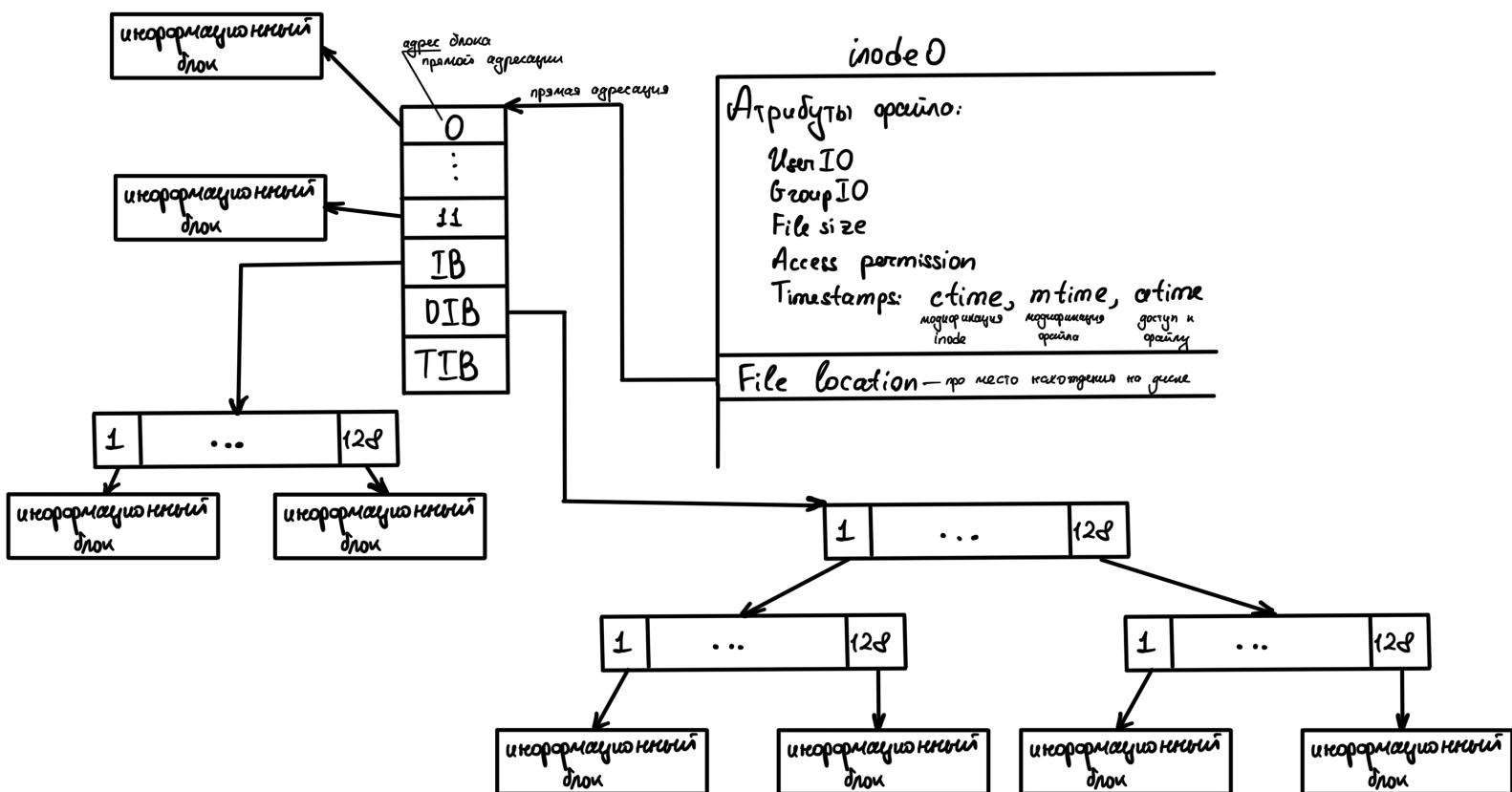
Рекорд размера не имеет!

root	
bin	814
dev	419
etc	778
:	:

Суперблок содержит информацию, необходимую для монтирования и управления РС. В кэшовой РС один суперблок и он располагается в начале раздела.

Суперблок считывается в память ядра при монтировании РС и находится там до её демонтирования или завершения работы системы.

Таблица inode:



Блок bitmap — структура, которую нет понадобится занести блок (1) или ободрить

Битовая карта файл. дескрипторов — аналогично массиву inode (где используется дескрипторов)

struct super_block:

```
struct super_block
```

```
{
```

- struct list_head s_list
- kdev_t s_dev; — описывает устройство
- unsigned long s_blocksize;
- unsigned char s_dirty; — означает
- struct file_system_type *s_type; — вспомогательная
- struct super_operations *s_op; — в ней определены операции на суперблоке
- struct block_device *s_bdev; — описывает устройство, на котором PC. По сути, драйвер
- ...
unsigned long s_magic; — магическое число?
- struct dentry *s_root; — точка монтирования каталога основной системы
- ...
int s_count; — число ссылок
- struct list_head s_dirty; — список изменившихся inode-ов
- ...
char s_id[32]; — строка имени

```
}
```

<linux/fs.h>

```
struct super_operations
```

```
{
```

- struct inode *(*alloc_inode)(struct super_block *sb); — функция выделения inode-а на
указателю на суперблок, т.к.
inode предполагает конкретной
- void (*destroy_inode)(struct inode *);
- void (*dirty_inode)(struct inode*, int flags); — выделяется, когда в inode вносятся изменения
PC ⇒ конкретный суперблочку
используется для обновления структур
inode для изменения → него синхронизируется
- int (*write_inode)(struct inode*, struct writeback_control *wbc);

TODO

16 марта 2023 — Семинар 3

`/[pid]/maps`

A file containing the currently mapped memory regions and their access permissions. See [man pmap](#) for some further information about memory mappings.

Permission to access this file is governed by a ptrace access mode PTRACE_MODE_READ_FSCREDS check; see [man ptrace](#).

The format of the file is:

```
address      perms  offset  dev  inode      pathname
00400000-00452000 r-xp 00000000 08:02 173521 /usr/bin/dbus-daemon
00651000-00652000 r--p 00051000 08:02 173521 /usr/bin/dbus-daemon
00652000-00655000 rw-p 00052000 08:02 173521 /usr/bin/dbus-daemon
00e03000-00e24000 rw-p 00000000 00:00 0      [heap]
00e24000-011f7000 rw-p 00000000 00:00 0      [heap]

...
35b1800000-35b1820000 r-xp 00000000 08:02 135522 /usr/lib64/ld-2.15.so
35b1a1f000-35b1a20000 r--p 0001f000 08:02 135522 /usr/lib64/ld-2.15.so
35b1a20000-35b1a21000 rw-p 00020000 08:02 135522 /usr/lib64/ld-2.15.so
35b1a21000-35b1a22000 rw-p 00000000 00:00 0
35b1c00000-35b1dac000 r-xp 00000000 08:02 135870 /usr/lib64/libc-2.15.so
35b1dac000-35b1fac000 ---p 001ac000 08:02 135870 /usr/lib64/libc-2.15.so
35b1fac000-35b1fb0000 r--p 001ac000 08:02 135870 /usr/lib64/libc-2.15.so
35b1fb0000-35b1fb2000 rw-p 001b0000 08:02 135870 /usr/lib64/libc-2.15.so
...
f2c6ff8c000-7f2c7078c000 rw-p 00000000 00:00 0      [stack:986]
...
7ffffb2c0d000-7ffffb2c2e000 rw-p 00000000 00:00 0      [stack]
7ffffb2d48000-7ffffb2d49000 r-xp 00000000 00:00 0      [vdso]
```

The *address* field is the address space in the process that the mapping occupies. The *perms* field is a set of permissions:

См. мануал `proc - pagemap`

Базовая таблица

Элемент	Тип	Назначение
<code>cmdline</code>	строка	путь к файлу, в который записаны аргументы командной строки
<code>cwd</code>	строка	текущий рабочий каталог
<code>environ</code>	строка	список переменных окружения
<code>exe</code> — путь к исполняемому файлу	строка	имя и путь к исполняемому файлу
<code>-d</code>	строка	запись состояния на дискету, открытую процессором
		Посмотреть

самостоятельно

6 модуля

maps

строк

pages

строк

root
корневой каталог
ФС, который
приведен
(строк) в процесс

символическая
ссылка

???

stat
информация
о процессе

task
директория
которой поддиректории /<tid>
потоков
от: 5 потоков

директория

Энумерации — текстовый строк

Всё — текстовый строк, т.е. строк можно открыть
в любом текстовом редакторе

Пример:

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

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

Статья на pagemap, с примером:

/proc/[pid]/pagemaps and /proc/[pid]/maps | linux

I'm trying to get my head around the two files mentioned in the title. I've looked up what the bits are; however, I'm failing to understand how to extract useful info from them (or I'm simply approaching it the wrong way).

Let me explain: The pagemaps is a rather newer "feature" pseudo file that contains the physical frame information of virtual pages assigned to a current [pid]. That is, given a virtual page that starts at address x, say 'vas' for virtual address start, I can index the pagemap file using vas to get the 64-bits of the mapped physical page frame. These bits contain info about that virtual page. However, when I extract the bits and a bit of shifting I'm getting lost with what I'm seeing.

The bits are represented as follows: 0-54 is the page frame number, 55-60 is the page shift. 63rd bit is the present bit, there are other bits of little interest to me. After I do a bit of mapping using vas addresses from /proc/[pid]/maps, it seems that just about every process' page is swapped. I.e. the 63rd bit is always a zero. {-}

I guess the question would be, how should I go about effectively using pagemaps to get the equivalent physical address of the address given by /proc/[pid]/maps

To be fair, I've posted a similar question but the approach was a bit different a few days earlier.

If anyone can shed some light on this matter I would be greatly appreciative.

==EDIT==

To address the comment below: I'm reading a line from /proc/[pid]/maps and the lines look like:

```
00400000-00401000 r-xp 00000000 08:01 8915461  
/home/janust/my_programs/shared_mem 7ffff1b000-7ffff3c000 rw-p 00000000 00:00 0  
[stack]
```

Then I'm extracting the number of virtual pages it touches and indexing a binary file /proc/[pid]/pagemaps, and for each virtual page I can extract the physical page it is assigned to.

The output looks like:

```
00400000-00401000 08:01 8915461  
/home/janust/my_programs/shared_mem num_pages: 1 : 86000000001464C6
```

One physical address for each virtual page in the virtual range.

The code for reading the line and extracting the physical address is:

```
74 /* process /proc/pid/maps, by line*/  
75 while(gets(line, 256, in_map) != NULL){  
76     unsigned long vas;  
77     unsigned long vae;  
78     int num_pages;  
79  
80     //print line  
81     printf("%s", line);  
82  
83     /*scan for the virtual addresses*/
```

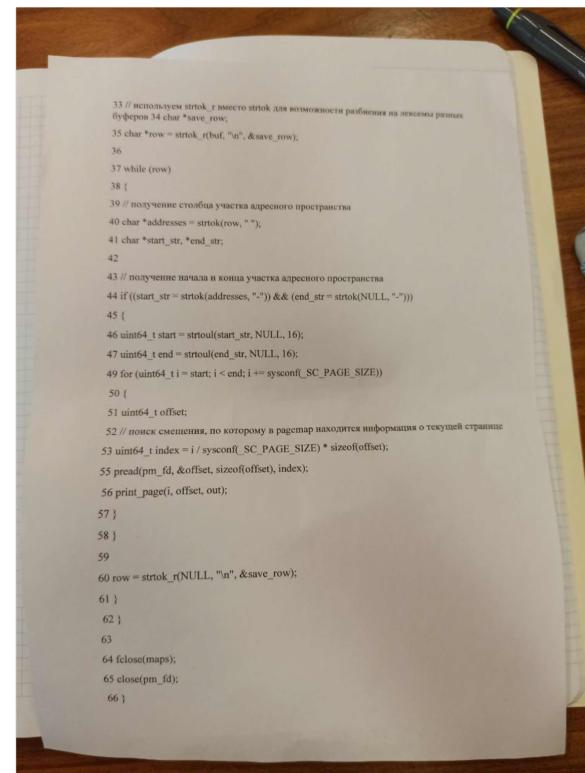
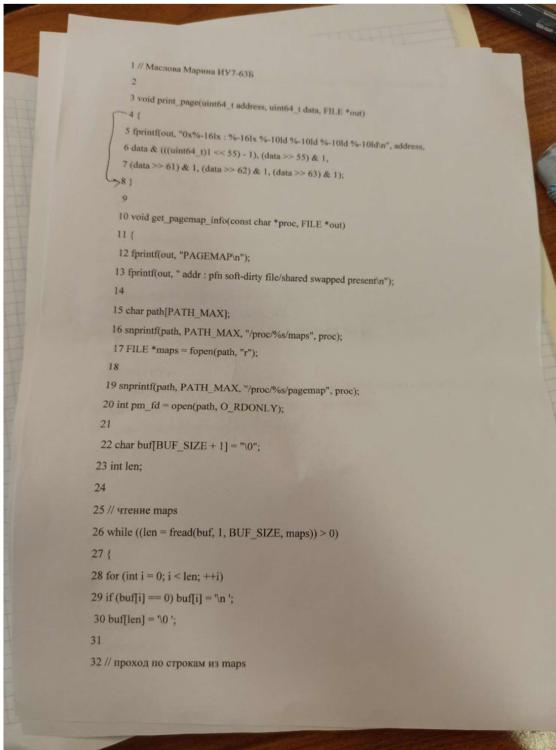
```
84     n = sscanf(line, "%lx-%lx", &vas, &vae); // настройки опоры  
85     if(n != 2){  
86         printf("Invalid line read from %s\n", maps);  
87         continue;  
88     }  
89     num_pages = (vae - vas) / PAGE_SIZE;  
90     printf("%d\n", num_pages);  
91  
92     if(num_pages > 0){  
93         long index = (vas / PAGE_SIZE) * sizeof(unsigned long long);  
94         off64_t o;  
95         size_t t;  
96  
97         /* seek to index in pagemaps */  
98         o = lseek64(pm, index, SEEK_SET);  
99         if(o != index){  
100             if(o < index){  
101                 printf("Error seeking to offset, index:%ld, index:%ld\n", o, index);  
102             }  
103             /* map the virtual to physical page */  
104             while(num_pages > 0){  
105                 unsigned long long pa;  
106  
107                 /* Read a 64-bit word from each pagemap file... */  
108                 t = read(pm, &pa, sizeof(unsigned long long));  
109                 if(t < 0){  
110                     if(t < 0){  
111                         printf("Error reading file '\%s'\n", page_map);  
112                         goto next_line;  
113                     }  
114                     printf("\$0\n");  
115                 }  
116             }  
117         }  
118     }  
119     /* however, I think I'm getting the right output, the index seems to be either a type  
120     mismatch or something else is going on. The output say for instance for the [shared mem] line  
121     in maps gives a wrong index; yet I'm still able to scan through the binary file and get the  
122     physical page address.
```

The example of that output is below:

```
969 7ffff08450000-7ffff08d59000 rw-s 00000000 00:04 0 /SYSV000003039 (deleted)  
970 num_pages: 1  
971 Error seeking to: -1081840960, index:273796065984.  
972 :86000000001464C6  
Ok, now, lastly I should say that this is under a 64bit OS, and this problem doesn't persist in a  
32bit OS.
```

33 // используем strtok, т.к. вместо строк для возможности работы на лексемы разных
буферов 34 char *save_row;

```
35 char *row = strtok_r(buf, "\n", &save_row);  
36  
37 while (row)  
38 {  
39 // получение стобца участка адресного пространства  
40 char *addresses = strtok(row, " ");  
41 char *start_str, *end_str;  
42  
43 // получение начала и конца участка адресного пространства  
44 if((start_str = strtok(addresses, " ")) && (end_str = strtok(NULL, " ")))  
45 {  
46     uint64_t start = strtoul(start_str, NULL, 16);  
47     uint64_t end = strtoul(end_str, NULL, 16);  
48    for (uint64_t i = start; i < end; i += sysconf(_SC_PAGE_SIZE))  
49 {  
50     uint64_t offset;  
51  
52 // поиск смещения, по которому в падсмар находятся информации о текущей странице  
53     uint64_t index = i / sysconf(_SC_PAGE_SIZE) * sizeof(offset);  
54     pread(pm_fd, &offset, sizeof(offset), index);  
55     print_page(i, offset, out);  
56  
57 }  
58 }  
59  
60 row = strtok_r(NULL, "\n", &save_row);  
61 }  
62 }  
63  
64 fclose(maps);  
65 close(pm_fd);  
66 }
```



Загружаемые модули ядра

Напоминание: Linux — ОС с многопоточным ядром. Чтобы не перекомпилировать всё ядро, в современных системах UNIX / Linux существует загружаемые модули ядра.

Пример: на простейшую структуру заг.модуля

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

// например: одна обработка, ..., многочтися

MODULE_LICENSE("GPL");
загружаемые модули ядра
использование
static int __init my_module_init()
{
    printk(KERN_INFO "Hello, world!\n");
    return 0;
}

static void __exit my_module_exit()
{
    printk(KERN_INFO "Stop\n");
}

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

(в функциональном ядре)

В случае можно использовать только функции ядра
кода отдельно стандартного функционала

API知名 printf → printk

Пример (простейший makefile)

obj-m += module.o

ончай скрипта каталога

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

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

Существуют insmode, lsmod, rmmod
modinfo

Для них требуется sudo

dmesg — где запись в лог

Уровни протоколирования

0 KERN_EMERG — система развалилась !!

1 KERN_ALERT — предупреждение

2 KERN_CRIT — критическая ошибка

... TODO — warnings

6 KERN_INFO

7 KERN_DEBUG

Компилирование: инкапсуляция о процессе — task_struct

В системе имеется кольцевой список процессов, для взаимодействия с теми есть отдельные ортодоксы

Алан Чирюлин — ibm.com, статьи и книга

Про Лаб.2: 1. Вывод структур кольцевого списка

2. По Чирюлину: взаимодействие модулей
sgpa

Пример (однородной, только init)

```
int init__module(void)
{
    struct task_struct *task = &init_task; // голова
    do
    {
        prink(KERN_INFO "____ %s, pid-%d, parent:-%s, ppid-%d\n",
              task->comm, task->pid, task->parent->comm, task->parent->pid);
    } while ((task = next_task(task)) != &init_task);
    return 0;
}
```

Поля: state

flags ← их хорошо бы видеть, но они лежат в

prio ← это единственний показатель приоритета

policy

struct mm_struct *mm

struct mm_active *active_mm

← это виртуозно, но в курсовой это поле и не только

Tuan D — еще автор

struct fs_struct *fs; — указатель на РС проекта

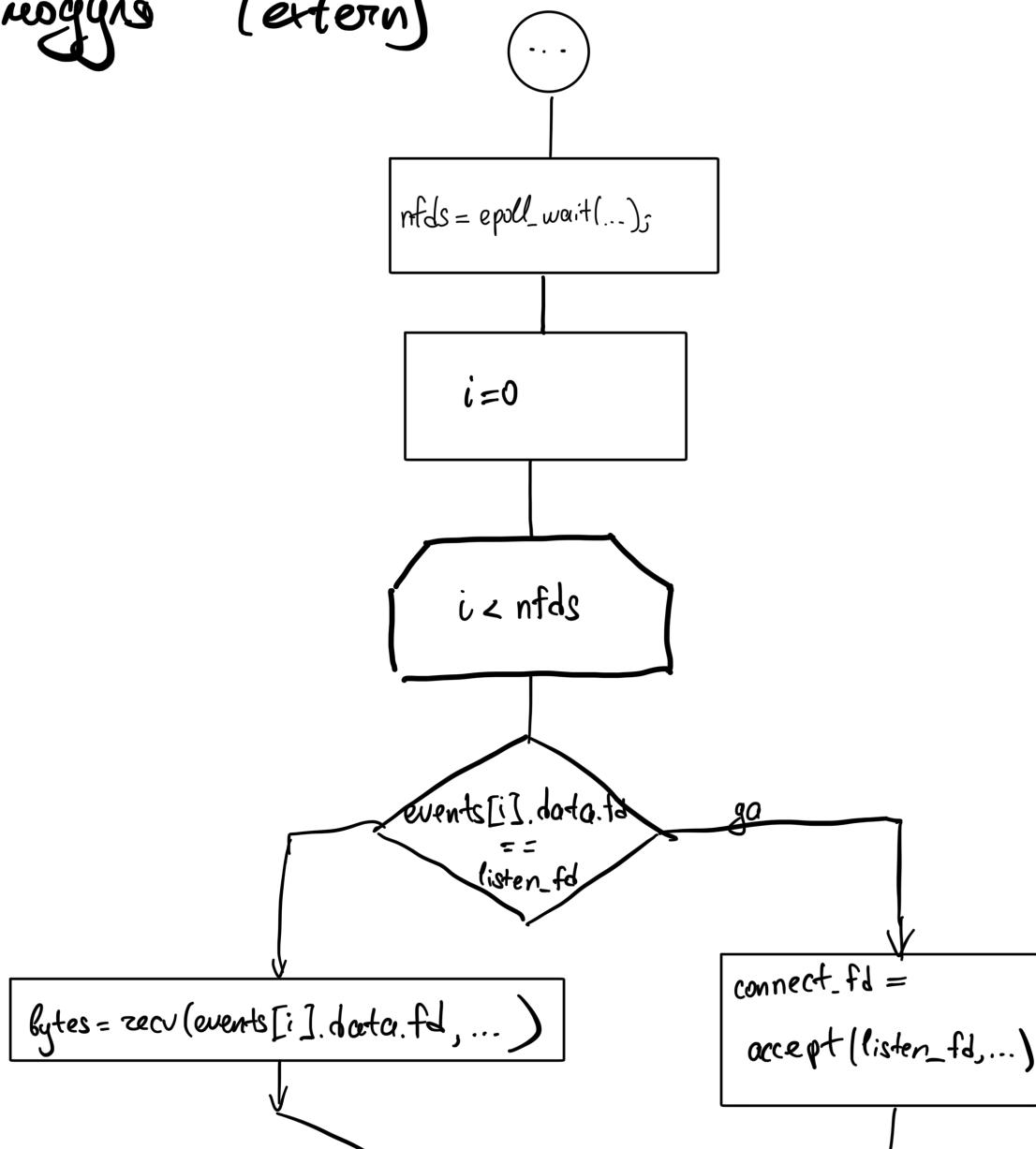
или бордюм $fs \rightarrow \underline{\text{root}}$

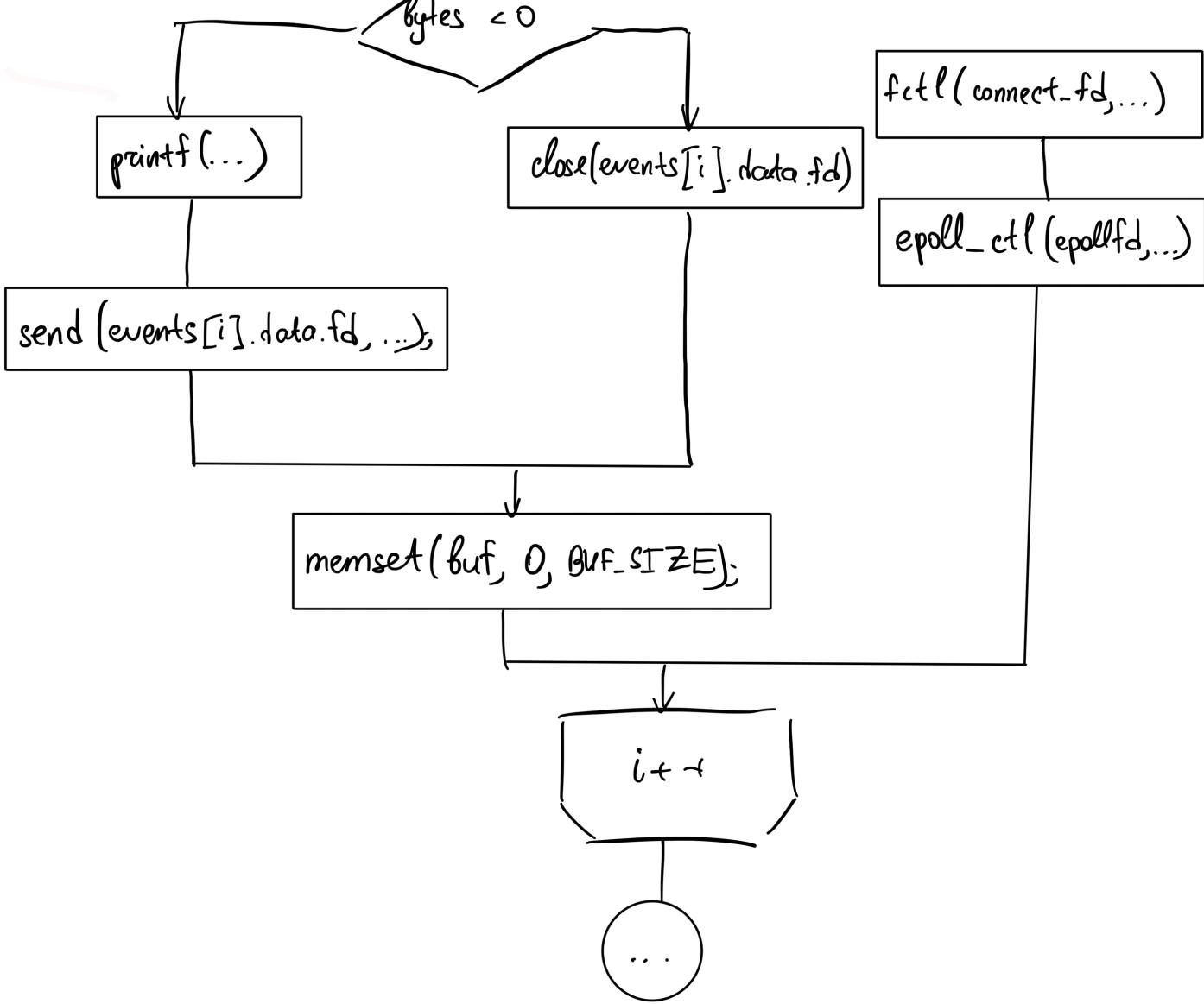
Еще одна тема, которые рассмотрены в курсовой
В курсовой так же есть о проекте.

Взаимодействие между собой ядра

Всё по Алану Уоррену

"Решка" между модулями в Assembly — переходы из
другого модуля (extern)





24 марта 2023 — Лекция 3

Когда система воспоминает, ОС во вторичной памяти
в ПЗУ команды для записи.

Без операционной системы — чудеса

Один объект суперблока — одна FS, которая может
быть подмонтирована

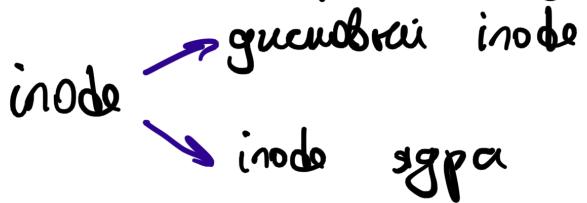
`alloc_super` — создаёт новый суперблок

где определены PC и ядро
он передается через union, потому что ядро
использует только PC

```
static struct super_block *alloc_super (struct file_system_type *type,  
int flags, struct user_namespace *user_ns)  
{  
    struct super_block *s = kzalloc(sizeof(struct super_block), GFP_USER);  
    static const struct super_operations default_op;  
    if (!s) return NULL;  
    INIT_LIST_HEAD(&s->s_mounts); // Оно уточняет, что PC имеет дескт  
    ...  
    INIT_LIST_HEAD(&s->s_inodes);  
    ...  
}
```

struct inode

Описывает операции с файлом



```
struct inode  
{  
    struct list_head i_hash;  
    struct list_head i_list;  
    struct list_head i_dentry;  
    ...  
    unsigned long i_ino;  
    atomic_t i_count;  
    kdev_t i_rdev;  
    umode_t i_node;  
    ...  
    loff_t i_size
```

// none, связанные со временем создания/изменения

// none, связанные с информацией о блоках — только генератор ядра — 6 noneй

unsigned int i_blkbits; // Битовая карта блоков

```

unsigned long i_blksize; // размер блока
unsigned long i_blocks; // количество блоков
...
struct inode_operations *i_op;
struct file_operations i_fop; // открытие/запись/прочитка файлов
struct super_block *i_sb;
struct list_head i_devices;
struct pipe_inode_info i_pipe;
struct block_device i_bdev;
struct char_device i_chdev;
...
unsigned long i_state;
unsigned int i_flags;
...
union // неизвестно трансфер PC,
// proc_inode, socket
{
    struct minix_inode_info minix_i;
    struct ext2_inode_info ext2_i;
    ...
    struct ntfs_inode_info ntfs_i;
    struct msdos_inode_info msdos_i;
    ...
    struct nfs_inode_info nfs_i;
    struct ufs_inode_info ufs_i;
    ...
    struct proc_inode_info proc_i;
    struct socket_info socket_i;
    ...
}
}
}

```

inode_operations

```

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 (*rename)(struct inode*, struct dentry*,
              struct inode*, struct dentry*);
```

..
}

file_operations

```
struct file_operations  
{  
    struct module *owner;  
    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*);  
    ..  
}
```

Структура inode каталога

inode number	3470036
.	3470036
..	3470017
folder1	3470031
file1	3470043
file2	3470023
folder2	3470024
file3	3470065

Структура dentry

реализует механизм (merge и отображение)

type	field	descriptor
atomic_t	d_count	количество использования объекта dentry
unsigned int	d_flags	флаги, определяющие для конкретного объекта dentry
struct inode*	d_inode	указывает на inode, связанный с файлом

struct entry* -parent	наз. эн	
struct list_head d_hash	содержит указатели на соседние элементы элементы в списке с одинаковыми именами	
struct list_head d_lru	список dentry в состоянии unused	
struct list_head d_child	список дочерних элементов	
int d_mounted	описывает, установлен ли dentry в таблице монтирования FS	
struct qstr d_name	имя имени	
struct dentry_operations* d_op	функции, системные вызовы для работы с dentry	
struct super_block* d_sb	указатель на суперблок	
unsigned long d_ufs_flags	ординарные dentry кеша	
struct list_head d_alias	список ассоциирующихся inode-ов	

30 марта 2023 — Семинар 4

Анализ

Узлы фрагментов: Сами напишите ядро Linux

md.h

```
extern char *mds_data;
extern char *mds_proc;
```

1. В node по MDAPI по сегментам делали так, что db; сегменты обединялись

2. Чтобы сегменты написать было

3. В одном сегменте переменные с **public**

а в другом с **extern** — тогда её можно использовать

Речь идет о программах ядерного уровня, которые загружаются и исполняются в ядре

md1.c

```
#include <linux/init.h>
#include <linux/module.h>
#include "md.h"
```

```

MODULE_LICENSE("GPL");
char * md1_data = "aaa";
extern char * md1_proc(void);
{
    return 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)
{
    prink("+ module md1 start\n");
    return 0;
}

static void __exit md_exit(void)
{
    prink("- md1 stop\n");
}

module_init(md_init);
module_exit(md_exit);

```

md2.c

```

#include <linux/init.h>
#include <linux/module.h>
#include "md.h"
MODULE_LICENSE("GPL");
static int __init md2_init(void)
{
    prink("+ module md2 start\n");
    prink("+ string exported from md1: %s\n", md1_data);
    prink("+ exported proc: %s\n", md1_proc());
    return 0; // md3 return -1; — как работает логика
              — от компиляторов, то
              — не запускает см. sys-log.
              — Если логика возвращает ошибку,
              — то ее не будет запущено.
}

static void __exit md2_exit(void)
{
    prink("- md2 stop\n");
}

```

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

Последний printk пишет в sys/log, в чём будет история ошибки — её показать и проанализировать

Про вторую часть, с ресурсами:

вспоминание о линиие выделении
выделение в кирсе — по страницам, поэтому
стартовала — выделение дескриптора, а страницы —
при записи.

у процесса только виртуальное адресное пространство,
т.н. кашевое, не оригинальное
вспоминание о PAE

чтобы загрузить точку входа — нужно создать
таблицы, но острой необходимости (см. ранее в PAE), они
исходятся в ядре.

Следующая надработка — это передача данных
из режима пользователь в режим ядра и
обратно.

Чтобы так передавать, нужно создать один

proc - модули управл. памяти

5.16.8 <linux/proc_fs.h>

struct proc_dir_entry

```
struct proc_dir_entry
{
    atomic_t in_use;
    refcount_t refcnt;
    struct list_head pde_openers;
    spin_lock_t pde_unload_lock; // использует объект, когда используется
                                // соответствующие областя буферизации
    const struct inode_operations *proc_iops;
    union
    {
        const struct proc_ops *proc_ops; // для регистраций файлов
                                         // описывает (read/write)
        const struct file_operations *proc_dir_ops;
    };
    const struct dentry_operations *proc_dops;
    union
    {
        const struct seq_operations *seq_ops;
        int (*single_show)(struct seq_file *, void *);
    };
    proc_write_t write;
    void *data;
    ...
    unsigned int low_ino; // inode номер
    nlink_t nlink;
    loff_t size;
    struct proc_dir_entry *parent; // для каскадов
    ...
    char *name;
    h8 flags;
    ...
}
```

2 способа регистрации информации
ядро → подготавливает - seq - функции

I copy-to-user()

copy_from_user()

II sequence-operations

Проверяя непривилегии?

Проверяя что не стандартные POSIX (в отличие от организованных API)

struct proc_ops

```
struct proc_ops
{
    unsigned int proc_flags;
    int (*proc_open)(struct inode*, struct file*);
    ssize_t (*proc_read)(struct file*, char __user*, size_t, loff_t*);
    ...
    ssize_t (*proc_write)(struct file*, char __user*, size_t, loff_t*);  

    loff_t (*proc_seek)(struct file*, loff_t, int); // gns обращение к конкретному // every в строке
    int (*proc_release)(struct inode*, struct file*); // означает уничтожение файла // или функции close!
    int (*proc_mmap)(struct file*, struct vm_area_struct*);  

}
```

proc_create_data - метод создания операции в BPC proc

```
extern struct proc_dir_entry *proc_create_data(const char *,
                                              umode_t,
                                              struct proc_dir_entry*,
                                              const struct file_operations*  

                                              void*);
```

↑
proc-ops "nogmc" 5.16

7 апреля 2023 - Лекция 4

dentru храниется или inode — как правило — и создается на лету

Общий dentru всегда прилагается конкретной файловой системе

Общий dentru может находиться в одном из 4 состояний:

- 1) free — dentru не содержит достоверной информации и не используется ВРС. Память контролируется slab-демоном (о slab-памяти позже)
- 2) unused — фрагментарное время ядром не используется, d_count = 0, но d_inode указывает на соответствующий дескриптор. Содержит достоверную информацию, но может быть удалён, а память освобождена
- 3) in use — используется, d_count > 0, d_inode — на соответствующий дескриптор. Удалён быть не может
- 4) negative — для такого dentru не существует соответствующего inode
 - inode был удалён с диска
 - dentru был создан как элемент пути к недействующему файлу

d_inode = null, то объект всё ещё находится в кеше
dentry

dentry - кеш

Обращение к диску — физическая операция. Кампай dentry хранится по диску, и оно к драйверу пути может быть другого.

Кампай блокчай — отдельный файл. При обращении к файлу происходит обход всех элементов пути (см. exec — когда проверяли существование файла, делали обход).

Решение (оптимизация) в хранении dentry в кеше

Пример: отредактировали файл — распаковали — удалили — пересчитали часть икоррекции в отдельный файл

Хранение в кеше существенно уменьшает время, которое требуется для обращения к файлу.

В Linux кеш dentry состоит из двух типов структур

1) setof — набор dentry objects в состояниях in use, unused или negative

2) хеш-таблицы (hash table) — для быстрого получения

объекта изображенного с зеркальным отражением и
каталогом.

Всегда, если объект то выходит в кэш, оружие
хэн?
кешированием возникает тупое зеркало

Управление (контроль) для кеш-памяти inode.

Кеш inode — список, slab — его часть

Кеш хранился в оперативной памяти.

Все использование данных выходит в
двусвязный список, обновляемый по алгоритму LRU
(least recently used).

Адреса первого и последнего элемента LRU списка —
— в next и prev у элемента dentry-used.

Каждый dentry в состоянии in use — в
двусвязном списке i-dentry соответствующего объекта
inode. ($inode \rightarrow i_dentry$)

Поле d-alias хранит адреса соседних элементов
в списке (next и prev)

Двусвязные списки, т.к. доступ быстрее
за свой алгоритмов (например бинарный поиск)

Объекты in use могут стать negative, если

указывается последний отецкая ссылка на файл. Сам dentry перемещается в LRU список dentry_unused.

LRU — вспомогатель из прошлого семестра

Hash табл ресурсов как массив dentry-hash-table
spin-блокировка (dcache-lock) защищает структуры
dentry-кода от одновременного доступа в многопоточ-
сопрв системах

dentry-operations

```
struct dentry-operations
{
    int (*d_revalidate)(struct dentry *, unsigned int); // ???
    ...
    int (*d_hash)(const struct dentry *, unsigned int); // добавление
                                                       // dentry в хеш-таблицу. Первый элемент там —
                                                       // родительский каталог
    int (*d_compare)(const struct dentry *, unsigned int, const char *, const struct qstr *); // ищет с кем сравнивать
    int (*d_delete)(const struct dentry *); // если удаляется подкаталог
    int (*d_init)(const struct dentry *); // когда dentry создается
    int (*d_release)(struct dentry *); // когда dentry освобождается
    void (*d_input)(struct dentry *, struct inode *); // вызывается, чтобы сработать
                                                       // дополнительные правила синхронизации
    char (*d_name)(struct dentry *, char *, int); // когда dentry
                                                       // переходит в inode
    ...
}
```

когда необходимо сконструировать путь к элементу dentry

(со временем)

вызывается, чтобы сработать дополнительные правила синхронизации dentry. Первый dentry — родитель того, который обрабатывается

Пример (где генотипически проблема) — показывает генетику
функционирования в системе gnt доступа к файлу mbox.

/usr/ast/mbox — первое обращение

root directory

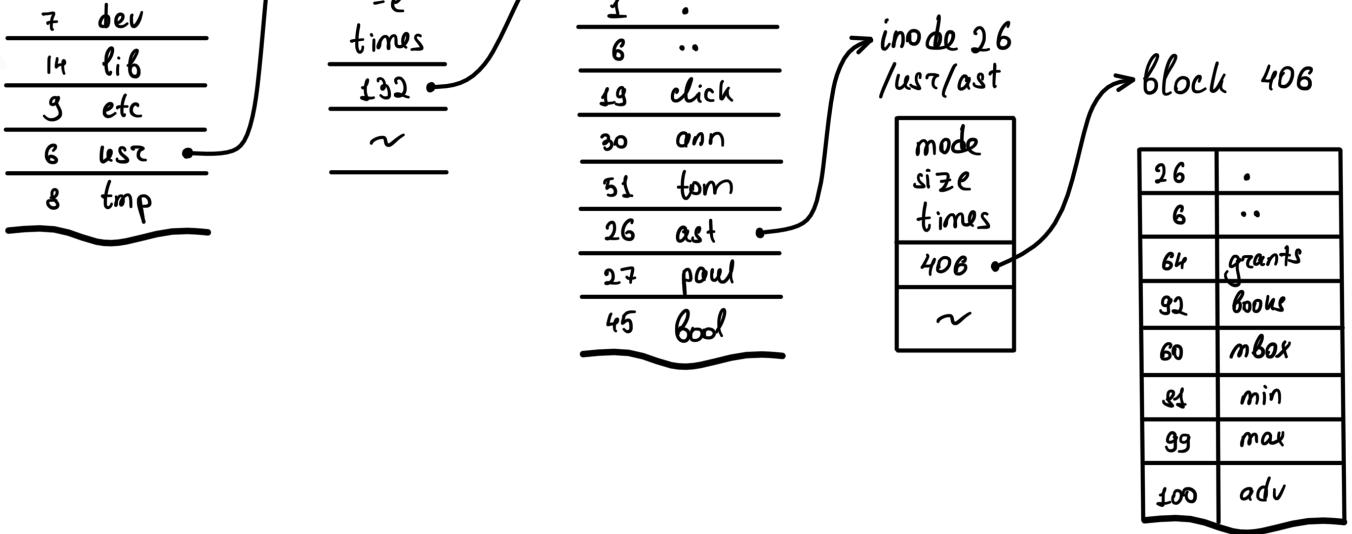
1	.
1	..
4	bin

→ inode 6

/usr

mode
r--

> block 132



struct file — структура с открытыми файлами, которые управляет процессу — дескриптор открытого файла

В системе существует одна таблица открытых файлов. Открытый файл может только процесс (используя `fork`, который наследует дескрипторы открытых файлов) — именно он является владельцем ресурсов.

```
struct file
{
    struct path f_path;
    struct inode *f_inode; /* cached value */ // указатель на файл в queue
    const struct file_operations *f_ops; //

    ...
    atomic long_t f_count; // количество текущих ссылок
    unsigned int f_flags; // опции
    fmode_t f_mode; // режим?

    struct mutex f_pos_lock;
    loff_t f_pos; // указатель файла, при записи/чтении он меняется
    ...
    struct address_space *f_mapping; // !? отображение — каким образом происходит
                                    // отображение по адресам?
};
```

struct file_operations

```
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*); аннотация на open
    int (*release)(struct inode*, struct file*); аннотация на release
    ...
};

}

```

Сравнить с struct proc-ops — от memory, то параметры такие же (полностью)

На подумать: почему в старых системах file-operations не может быть использована для регистрации функций (open, release)? Вместо этого proc-ops, то есть там те же.

При написании драйвера всё сводится к необходимому набору действий (read/write) — и где регистрация издавна использовалась struct file-operations.

Драйверы икого + там go aux nop (менято то получится, драйвера перестанут работать)

A вот с proc-ops проще — это VFS, всё то есть

13 апреля 2023 — Семинар 5

proc-ops vs file-operations — сравнение обновления в организациях. Порядок тот же, организмы ссылаются на

параметром.
(с. эгр 5.16)
Задачи так сделано: чтобы не переопределять уже
изначальные функции работы с файлами (read/write).
Получаются hook-и — перенесённые функции для дальнейшего
функциональности.

Ниже — сделано для возможности управляемого
контролем версий — они и те же функции можно
зарегистрировать как в proc-ops, так и в
file-operations.

Read/write — точки входа в фрагменты.

Изменяются бывшие hook-и на proc-dir-entry (на
read-write), потом в file-operations
и proc-ops

Красорнада — не только для передачи из эгр в
usermode, но и обратно (например, чтобы сделать
бэкдор).

proc-create — создает объект в proc

```
static inline struct proc_dir_entry *proc_create(const char *name,  
        umode_t mode,  
        struct proc_dir_entry *parent,  
        const struct proc_ops *proc_fops);  
  
{ return proc_create_data(name, mode, parent, proc_fops, NULL); }  
                                ↑ 5.16 file-operations  
                                |  
                                | void
```

В node - создает директорию - см.

proc_mkdir

```
extern struct proc_dir_entry * proc_mkdir(const char*, struct proc_dir_entry*);
```

и симбониксую ссылку - см.

proc_symlink

```
extern struct proc_dir_entry * proc_symlink(const char*, struct proc_dir_entry*);
```

В init - бывают различные опции

```
struct proc_dir_entry *my_proc_file;  
#ifndef HAVE_PROC_OPS  
static struct proc_ops file_proc_ops =  
{  
    • proc_read = procfs_read,  
    • proc_write = procfs_write,  
    • proc_open = procfs_open,  
    • proc_release = procfs_close,  
}  
#else  
static const struct  
{  
    • read = procfs_read,  
    • write = procfs_write,  
    • open = procfs_open,  
    • release = procfs_close,  
}  
#endif
```

```
static init __init proc_init(void)  
{
```

нагреватель, то release - API

close - функция

утилита

один файл
создан в кирюхом
которое proc

?
регистрирует
создан опять?

проверить

на
-1

```
my_proc_file = proc_create("my_file", 0644, NULL, file_ops);
... // mkdirs, symlink
return 0;
}
```

0644 - read for all, write & read for owner
0660 - write & read for owner and group

Пример (procfs-write)

```
char *kBuff=NULL;
static ssize_t procfs_write(struct file *file, const char __user *buffer,
                           size_t len, loff_t *off);
{
    ...
    ? - подобраться
    copy_from_user(kBuff[index], buffer, len);
```

<linux/uaccess.h>

```
unsigned long __copy_from_user(void *to, const void __user *from,
                                unsigned long n);  
unsigned long __copy_to_user(void __user *to, const void *from,
                                unsigned long n);
```

Это оригинальные ядра

связано с тем, что пространство оружия — чтобы модуль ядра мог одновременно и переносить, от получать новые дополнения (т.е. оружия) адрес

А у процессов пространство — отдельное под оружиями, реализуется при помощи тайм-страниц, страницы загружаются по прерыванию, могут быть вытеснены (LRU, FIFO, ...)

}

Также см. sprintf(...)

Про exit:

```
static void __exit proc_exit(void)
{
    remove_proc_entry("my_proc_file", NULL);
```

Во второй программе будем использовать ~~внешние~~ ~~внешние~~
sequence операции

```
struct seq_operations;
struct seq_file
{
    char *buf;
    size_t size;
    size_t from;
    size_t count;
    ...
    loff_t index;
    loff_t read_pos;
    struct mutex lock;
    const struct seq_operations *ops;
    int poll_event;
    const struct file *file;
    void *private;
};
```

struct seq_operations

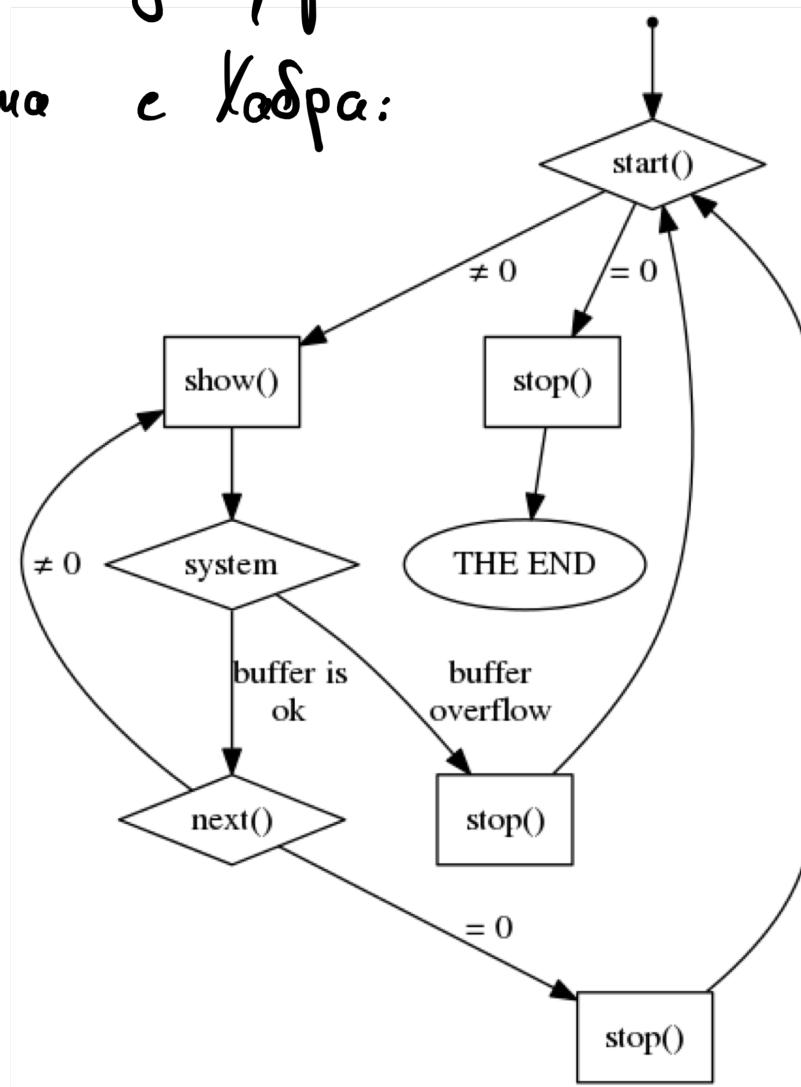
```
struct seq_operations
{
    void *(*start)(struct seq_file *m, loff_t *pos);
    void *(*stop)(struct seq_file *m, void *v);
    void *(*next)(struct seq_file *m, void *v, loff_t *pos);
    int *(*show)(struct seq_file *m, void *v);
};
```

"Использование описов последовательностей sigpro"

Навр

5.4. Manage /proc file with seq file

Диаграмма с кодом:



Результаты испытаний работать автоматически с состоянием в VFS proc описом

stop в 3-х ситуациях

- I После start — запущено работа по буферу данных
- II После show — при выполнении последней seq-printf произошло переполнение буфера
- III После next — тегмы или завершить работу, или

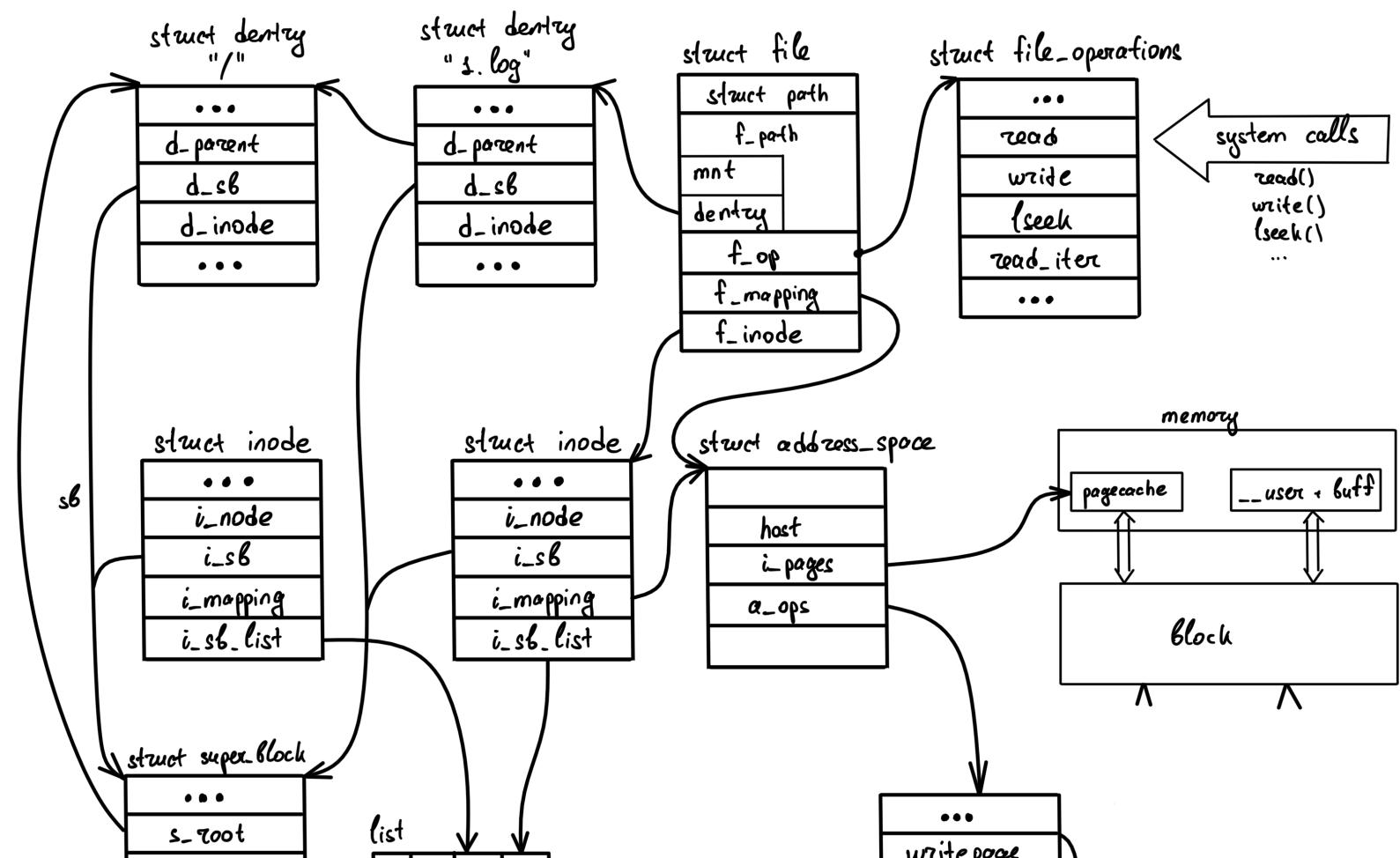
перейти к передаче новых данных — итератор!

Пример на III: возвод информации структур на блоки, о путях по символьным устройствам. После передачи информации по блочным устройствам вызывается stop и start для передачи информации о символьных устройствах.

21 апреля 2023 — Лекция 5

Демонстрация связей структур VFS на основе inode

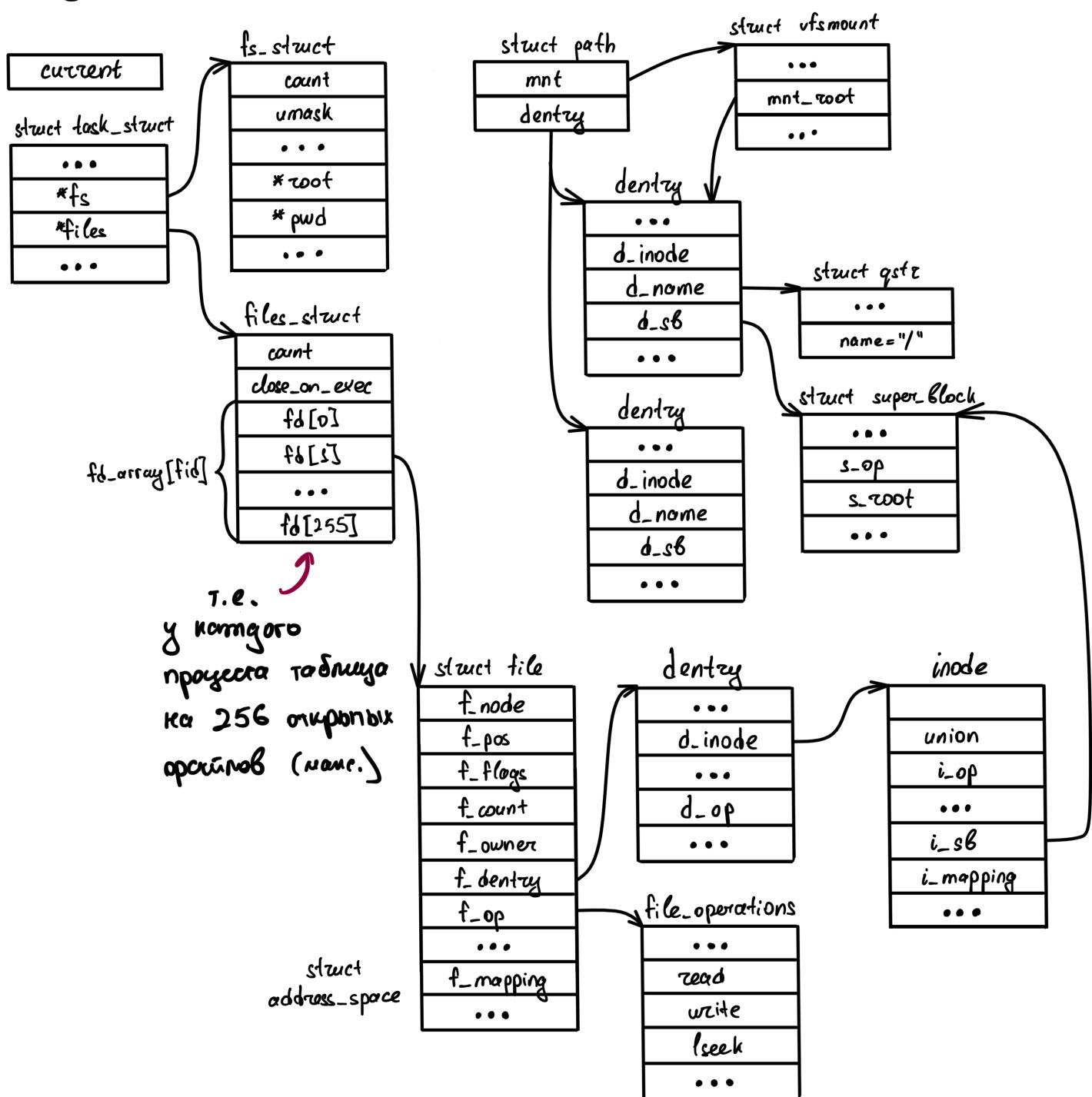
Одна открытая точка — acc. descriptor, связанные с операциями (read, write, fseek) — работают с открытыми файлами (task_struct — дескриптор открытого файла)





Замечание: поскольку работа ведется только с открытыми файлами, тут нет метода open

Другой вариант базаек, с task_struct



См. Тум Домин — статью по проблемам систем

Мораль: Все структуры в常务, все заслуживаются

Сравнение операций:

super_operations	inode_operations	dentry_operations
alloc_inode	lookup	...
destroy_inode	get_link	d_hash
dirty_inode	...	d_compare
write_inode	link	...
drop_inode	symlink	d_init
...	mkdir	d_release
	rmdir	...
	mknod	d_dname
	rename	
	...	

После описания ФС система предоставляет возможность зарегистрировать или удалить ФС.

file_system_type, см. выше)

Одна и та же структура, описывающая тип файловой системы, может быть зарегистрирована несколько раз

Вам то: один с одним и тем же inode-ом может быть открыт несколько раз и иметь несколько дескрипторов. Это может привести к **тормозам** → потеря данных

struct file_system_type

```
struct file_system_type
{
    const char* name;
    int fs_flags;
#define FS_REQUIRE_DEV 1
    ...
#define FS_USERNS_MOUNT 8
    struct dentry *(*mount)(struct file_system_type*, int, const char *, void *);
    void (*kill_sb)(struct super_block *);  
    mount потребуется из сразу же выше define get_sb (из 5.0)
    struct file_system_type * next;
    struct hlist_head fs_supers; //хеш-лист
    struct lock_class_key s_lock_key;
    struct lock_class_key s_umount;
```

s-uc loc-class-key s-uts-toname-keys

}

Регистрация mount_ (bdev, nodev, ns, single)

```
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_nodev(struct file-system-type *fs-type, int flags,
 ↑
выбрасывает root, это
позволяет подмонтировать
PC, когда в inode для
root-a

```
void *data,  
int (*fill-super)(struct super-block *, void *, int));
```

↑
Берётся оставшиеся данные из
искусственного superBlock-а

...

Создание своей PC

1) Инициализировать none struct file-system-type

```
static struct file-system-type fs-type = {  
    .owner = THIS_MODULE,  
    .name = "myfs",  
    .mount = myfs_mount,  
    .kill_sb = kill_litter_super,  
}
```

27 апреля 2023 — Семинар 6

proc

open — скрипты алгоритма системного вызова open (go
организаций ASN) — ответ

VFS - свою функцию

Открытые файлы — отчёт — аналог 3-х программ, готовых на
файлах (как с распределением памяти).^{помимо} + ^{свои} структур.

Файл — единственное средство долговременного хранения
данных

Преобразование?

Пример (extra-simple, без интератора, с show. Ограничение
объёмом данных, который может передаваться — 64 Кб)

Это называется single file (singlefile subsystem)

```
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#define PROC_FILE_NAME "myfile"

static struct proc_dir_entry *proc_file; //создаётся объект в VFS proc
static char *str;
static int proc_show(struct seq_file *m, void *v)
{
    int error = 0;
    error = seq_printf(m, "%s\n", str);
    return error;
}
static int proc_seq_open(struct inode *inode, struct file *file)
{
    return single_open(file, proc_show, NULL);
}
```

static struct proc_ops proc_ops =

```

{
    .open = proc_seq_open, // no canon gene нужна только open
    .release = single_release, // onechar parsee
    .read = seq_read,
}

```

```
static int __init proc_my_init(void)
```

```
{
    str = "aabbb";
    proc_file = proc_create(proc_file_name, S_IRUGO, &proc_ops, NULL);
    if (!proc_file)
        return -ENOMEM;
    return 0;
}
```

```
static void __exit proc_my_exit(void)
{
    if (proc_file)
        remove_proc_entry(proc_file_name);
}
```

Б sequence-операции инициализации от struct file_operations

sequence-операции \rightarrow gpo \rightarrow Результаты

Для чего:

\rightarrow запрос — запрос текущих итераторов, сопровождаемых структурой gpa.

тозер gpo

Каждый загружаемый модуль содержит:

- 1) управляющие функции
- 2) функции модуля
- 3) структуры gns (последователь)

single-open

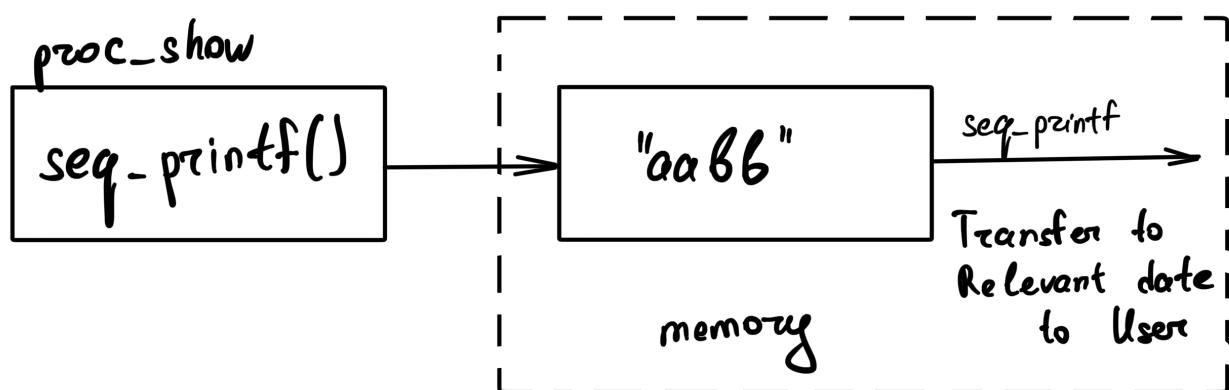
```
int single_open(struct file *file, int (*show)(struct seq_file *, void *),
                 void *data);  
{  
    struct seq_operations *op =  
        kmalloc(sizeof(*op), GFP_KERNEL);  
    int res = -ENOMEM;  
    if (op)  
    {  
        op->start = single_start;  
        op->next = single_next;  
        op->stop = single_stop;  
        op->show = show;  
        res = seq_open(file, op);  
        if (!res) ((struct seq_file *)file->private_data)->private = data;  
        else kfree(op);  
    }  
    return res;  
}
```

Руководство show дает буфер на 1 page.

Если использовать single-open, то гонки неизбежны
u single_release

Разберёмся как построим проце́ссорный
слой (откодусто́йчивость), который ^{объём} ~~занесёт~~ ?
ваша первая функция printf

Базовая идея: записываем данные в отдельную
большую область памяти



В результате — оно...

proc_write : User → Kernel — copy_from_user

```
static ssize_t proc_my_write(struct file *file, const char __user *buffer, size_t max_to_write, loff_t *offset)
```

```
{  
    ssize_t to_copy, not_copied;  
    to_copy = min(max_to_write, sizeof(kernel_buff));  
    not_copied = copy_from_user(kernel_buff, buffer, to_copy);  
    if (not_copied == 0)  
    {  
        printk("kernel buffer: \"%s\"\n", kernel_buff);  
        if (strcmp("deutsch", kernel_buff, 7) == 0)  
        {  
            /* ... */  
        }  
    }  
}
```

```

        output_string = TEXT_GERMAN; // str?
    }
    if (strcmp ("english", kernel_buff, 7) == 0)
    {
        output_string = TEXT_ENGLISH; // str?
    }
    return to_copy - not_copied;
}

```

Если номера генерат в struct proc-ops, то все еще поддается проверке.

Итератор — реализация в struct proc-ops через open, read, write, release, npu для в struct seq_operations через onechar onepage

Пример:

```

static void * tcp_seg_start (struct seq_file *s, loff_t *pos)
{
    loff_t *spos = kmalloc (sizeof (loff_t), GFP_KERNEL);
    if (!pos) return NULL;
    *spos = *pos;
    return spos;
}

static void * tcp_seg_next (struct seq_file *s, void *v, loff_t *pos)
{
    loff_t *spos = v;
    *pos = ++spos;
    return spos;
}

```

```
... sop, so  
const struct seq_operations tcp_seg_op = {  
    .show = tcp_seg_show,  
    .start = tcp_seg_start,  
    .next = tcp_seg_next,  
    .stop = tcp_seg_stop  
};
```

Кто же может `single-open`, кто не может — — оператор. Это пишется в курсивную.

open — открыть существующий
или
создать новый — олаг "create"
создать

возвращает адрес. десир. типа int
получает или адреса, ораги и режим
`O_CREAT | O_EXCL`
проверяя, существует ли

Будет лигт возвр. sys.open, нет, ... — си. макуан,
кто и покемакио — ораги (но если возвртс яго).

В случае создания — создам inode (если `O_CREAT`).

Инче создать струк file, беркую int и зделокио
работу.

Построит в хеди.

и 1. если есть в родителе открытых посейх

процесса.

См. почему, членами связей структур ядра. open — отдельно, т.к. оно может создать новую структуру.

5 маc 2023 — Лекция 6

Список файловых систем: df -hi

1. Список содержит inodes
2. Список inodes используется
3. Точка монтирования ФС

Попытка найти оптимальный способ управления памятью.

Нему dentru — сущ. разрр (in use,...)

К нему dentru есть нем inode-ob, его загораживает
околотка — ускорение поиска и доступа. В Linux это:

1) Глобальный hash-массив inode_hashtable — там конкретный
inode хешируется по значению указателя на суперблок и
32-разрядному номеру inode. Если суперблок отсутствует, то
inode добавляется в физический список anon_hash_chain
— там **активное** inode-ob. Пример **активного** inode-ob —
— сокет, созданные API ядра при **sock_calloc()** — в
котором выделяется **get_empty_inode()**

2) Глобальный список inode_in_use — **пустые** inode-ob,
у которых i_count > 0 и i_nlink > 0.

данное через `ge-emp-g-ino` `ge-new-ino`

добавляется туда же.

3) Глобальный список `inode_unused` — список `inode`,

где `i_count = 0`

4) Для каждого суперблока, у которого

`i_count > 0` и `i_nlink > 0` и `i_state_dirty`

создается список `sb → s_dirty`

inode считается dirty, если от дин изменен. Он обозначается `s_dirty`, то гонко если для хешровки (дружи) — некорректен

5) SLAB cache, называется `inode-cache`

Некоторые истории оптимизаций памяти:

1. Стратегия, базирующаяся на `heap` — однотипной памяти для динамического выделения памяти — там выделение памяти производится динамически (во время работы программы). Алгоритмы `first-fit` и `best-fit` для выделения памяти, который возвращается пользователю. См. сегменты памяти в прошлом семинаре — 3 метода.

Основная проблема — орагнизація — блоки возвращаются в разное время \Rightarrow ошибки — условия и времена, чтобы

и устраниТЬ \Rightarrow компактное расположение
(пристроено)

2. Buddy memory allocation — память устроена по
размерам, кратные 2, используется алгоритм **Best-fit**
Когда блок освобождается, проверяется блок **Buddy**, чтобы
проверить есть ли блоки симметрических свободных блоков (если есть,
объединяются).

Делается для уменьшения орагнизации

3. SLAB cache — считается, что устраивает организа-
цию \Rightarrow управление памятью эффективно

Смысл в том, что количество типов объектов, ^{как видимых, а не реальных}
которыми оперирует разработчик, **очень мало**

Структуры, структуры для mutex, структуры для inode
директорий...

Загрузка объектов, выделение и освобождение памяти \Rightarrow
 \Rightarrow организацию. Привели наблюдение, видоз.

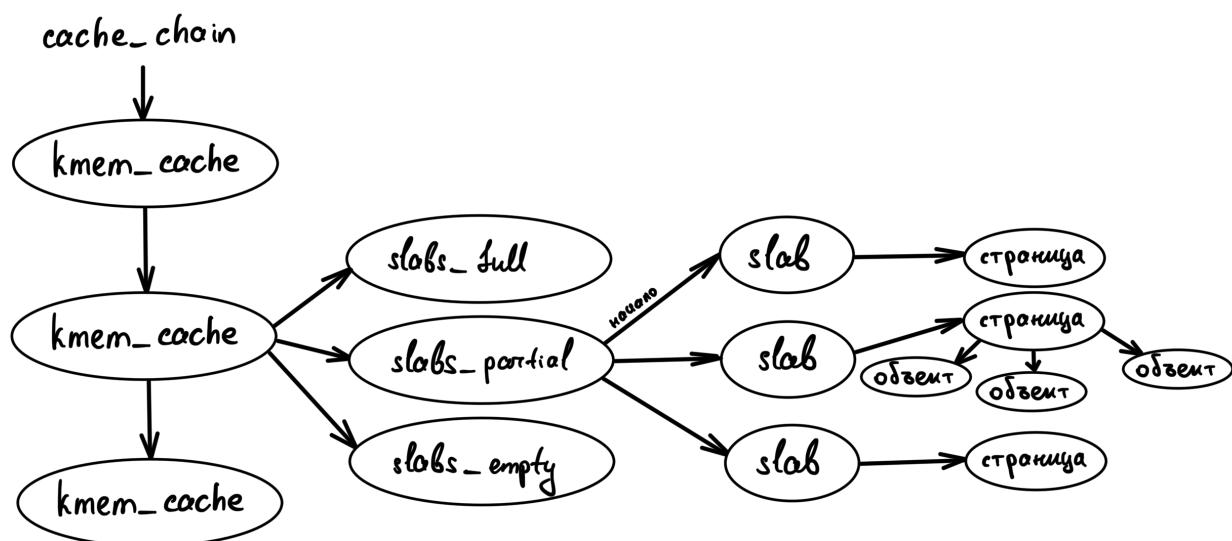
Нет смысла освобождать участок памяти определенного
размера скоро можно воспользоваться. Память же освобождается
и группируется в **slab** кеш.

ри очередном обращении на создание такого же
объекта выделение будет из кеша.

slab — непрерывный участок памяти (одинично —

— несколько смежных страниц) — может состоять из одного
или более slab-ов

Компьютерный кеш содержит список slab-ов.



Виды slab-ов:

`full` — заполненный

`partial` — частично заполненные

`empty` — пустой

основные элементы

Объекты — выделяются из кеша и в него же
возвращаются.

`slabs_empty` — конфигурация для **повторное** использование.

В случае распределения SLAB объекты о динамических
определяемого типа (а размера) определенное заранее.

Аллокатор SFS хранит информацию о этих участках (они избыточны или неисп.).

В результате если поступает запрос на выделение общей определённого размера, он удаляет бордюры при помощи slab'ов неиз.

В под.режиме no VFS — содержание slab

Остальное — см. методичку по VFS

Прерывания

Система прерываний — в ^{контроллер} ядре ОС

Для ОС однотипные прерывания —
1) системные вызовы
2) исключения
3) аппаратные прерывания

16 прерываний (PIC) — мало, 32 (APIC) — тоже мало

16-разр. компьютер

Таблица векторов прерываний

32-разр. компьютер

IDT — Interrupt descriptor table

8-байтовые дескрипторы
Концепция аналогична таблице векторов
— записи и параметры обработчиков, но для
того адрес обработчика, а отступ

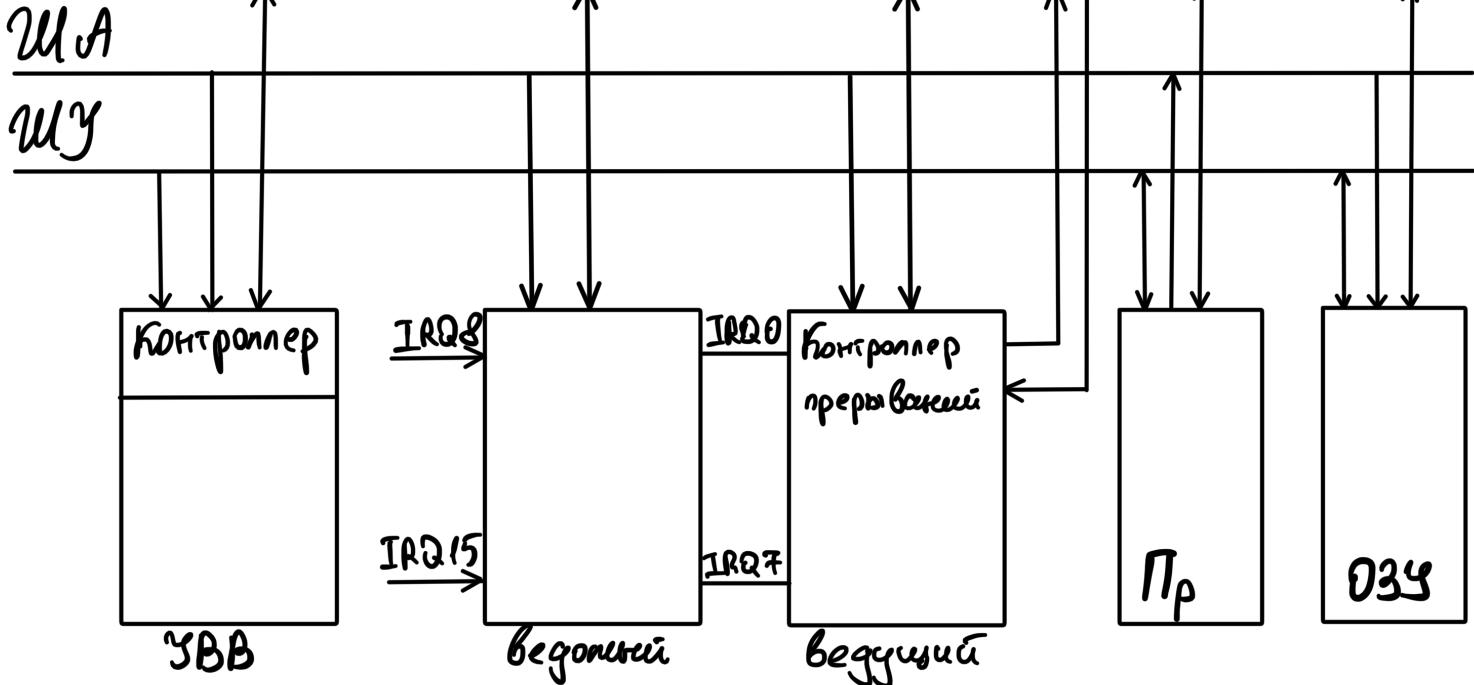
64-разр. компьютер

IDT + список определённых прерываний —

список прерываний от PIC,
а потом MSI —
— **Взять материал**

Простейшая концептуальная схема трёхслойной арх-ры:

UD



Вспомнишь, какой IRQ это
к эмулятору:

PS/2 - клави

маш?

таймер?

ввод-вывод?

окончание
байтова?

MSI (Message Signal Interrupt) – прерывания через сообщения.

Основная задача обработчика – посыпать на чипы нужную информацию (для этого обработчик сохраняет информацию в буфер ядра).

Аппаратные прерывания – на высоком приоритете
(вспомнишь отчёт из прошлого семестра)

Чтобы исключить возможные ошибки,
на ядре запрещаются прерывания
по линии IRQ запрещаются прерывания

— аппаратные прерывания не могут выполняться до конца
(это связывается на **отзывчивости** системы, т.к. ядро
блокируется).

В современных системах UNIX разложение
обработчиков на 2 части:

top half — аппаратное прерывание — interrupt handler

bottom half — отложенное действие

└ softirq

└ tasklets — в цикле броузинга

└ work queue

I Softirq? есть ли у него историческое название — **установка**

Регистрация обработчика прерывания
request_irq, **free_irq**

```
typedef irq_return_t (*irq_handler (int, void *));  
int request_irq(unsigned int irq, void *irq_handler_t handler,  
                unsigned long flags, const char *name, void *dev);  
extern void free_irq(unsigned int irq, void *dev);
```

Рассмотрим некоторые функции SA... — рабочие

IRQF... — сервисные

IRQF_SHARED — позволяет разделять один и тот же IRQ между несколькими устройствами

IRQF_PROBE_SHARED — устанавливается, если это возможно

IRQF_TIMER — определяет единственный быстрое прерывание.
Прерывание → быстрое быстрое всего этого (от таймера) это
недопустимо. Выполняется отдельно для конца и
не делится на 2 части (top и bottom)

IRQF_PERCPU — запрещение (исключением) для CPU

Использование отложенного действия — bottom half, эта же система управления прерываниями называется no-preemption.

softirq — выполняем ksoftirq_daemon по основу на

контрольный процессор (ksoftirq_daemon/0, ...)

Рутинная spawn_ksoftirq порождает демон ksoftirq.

В системе определён набор "глобальных" прерываний — см **\$0**, они определены статически:

индекс	приоритет	описание
HI_SOFTIRQ	0	высокоприоритетное
TIMER_SOFTIRQ	1	таймеры
NET_TX_SOFTIRQ	2	отправка
NET_RX_SOFTIRQ	3	приём сетевых пакетов !
BLOCK_SOFTIRQ	4	блокировка я-реестра
BLOCK_IOPOLL_SOFTIRQ	5	онлос
TACKLET_SOFTIRQ	6	такелет (определяемый пользователем)
SHED_SOFTIRQ	7	планировщик
HRTIMER_SOFTIRQ	8	не используется
RCU_SOFTIRQ	9	

— — —
NR_SOFTIRQS ??? = 10

Real Compute Unit — последний, т.е.
можно выложить в массив для
softirq, но это сделать до
RCU_SOFTIRQ (переключение синхрониз.)
После TASKLET reset скажет RCU
генерировать

softirq_action

```
struct softirq_action
{
    void (*action)(struct softirq_action *);
```

11 мю 2023 — Семинар 7

API seq-функции начинает работать, когда пользователь
вызывает **read** для чтения страницы из проца
(т.е. **read** — точка входа)

start — если возвращается не NULL, то идётся
через **next**.

Командный **read**, когда **next** — это **show**,
которая заменяет данные в буфер пользователю

next — это **nop**, новая она не возвращает NULL —
— если NULL **stop**

После **stop** возвращается **start** (используется след.
последовательность) — **next** — она заменяется, когда

~~start~~ Берёт NULL

seq-опции — на них определяется базовое функции

seq-read

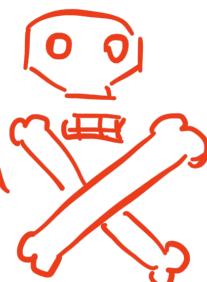
seq_lseek

seq_release

Результат seq-start / seq-read — это абсолютные точки

блога (точки блога — функции, определяющие на основе proc)

Комментарии и черепы с костями



При то, как определяет функции,

```
static int procfs_open(struct inode *inode, struct file *file)
{
    return seq_open(file, &ct_seq_ops);
}

struct proc_ops procfs_ops =
{
    .open = procfs_open,
    .read =
    .lseek = seq_lseek,
    .release = seq_release
}
```

В seq-point'е generic — stop — это всё!

Open

Предстоит: пройти через все виды ординарных зерен чтобы увидеть, как они выглядят. Но примере open, т.к. это значение распространено. Такие ординары могут изменяться (т.к. не стандартизованы POSIX) и перемещаться (могут в структуры, см. `dir_entry`).

Нужно: пройти через все виды.

API `open` выговаривается из приложения, в результате возвращается описанный дескриптор типа `int` — номер^{в массиве} в структуре `struct file_struct` — none в `struct task_struct`, список открытых описателей (открытых процессом), о `mem` и в `reg.node`

У каждого процеса хранится дескрипторов открытых описателей

Существует 2 версии API `open` (см. лекцию)

```
int open (const char *pathname, int flags); // из матрицы право доступа
int open (const char *pathname, int flags, mode_t mode); // один из них может быть использован для изменения и для создания файла
```

Для создания описания — флаг `O_CREATE`

`O_CLOEXEC` – существующий файл, запрещен

открытие (его часть в режиме ядра) должен создать дескриптор открытого файла `struct file` в системной таблице

Если битом идет с `O_CREAT`, обозначено

надо добавить `O_EXCL` – если файл существует ^{то} _{из написано} не будет создан новый (см. методичку голубой пример)

Кстати, есть разрешение файлов

Существует интерактивное `switch -u`, выполняющее на моменте пользования — функции `fstat`, `fstat` —

см. старые лекции по логике дерева файлов

Важно: голубая карточка из методички:



uno API gne oppgaven switch

↓

SYSCALL_DEFINE3 (open, const char __user *filename,
int flags, umode_t mode);

{
if (force_0_largetfile())
 flags |= O_LARGEFILE; //если времена 64-х разрядов
return do_sys_open(AT_FDCWD, filename, flags, mode);
}
} //безопасно
//если подпись

long do_sys_open (int fd, const char __user *filename, int flags,
umode_t mode)

{
struct open_flags op;
int fd = build_open_flags(flags, mode, &op);
struct filename *tmp; //audit - подпись

if (fd)
 return fd; //если подпись
tmp = getname(filename);
if (IS_ERR(tmp))
 return ...;

fd = get_unused_fd_flags(flags);

if (fd >= 0)

показать в отчете

{
struct file *f = do_filp_open(fd, tmp, &op);

if (IS_ERR(f))

{
 put_unused_fd(fd);
 ...

}

}

else

{
 f->snotify_open(f); //убедиться что бретыгает PC

//... / / / //

и т.д. в дальнейшем оставлять


```

struct nameidata nd;
int flags = op->lookup_flags;
struct file * filp;
set_nameidata(&nd, dfd, pathname); // инициализация полей структуры nameidata
filp = path_openat(&nd, op, flags | LOOKUP_RCU); // основная задача - инициализация
// основных полей структуры. Второе - кеширование
// через alloc_empty_file(); - в том
// kmem_cache_zalloc?
if (unlikely (filp == ERR_PTR(-ECHILD)))
    filp = path_openat(&nd, op, flags);
// ... Если -3 рано - вернуть ошибку, почему то
if (unlikely (filp == ERR_PTR(-ESTALE)))
    filp = path_openat(&nd, op, flags | LOOKUP_REVAL);
restore_nameidata();
return filp;
}

```

19 мэр 2023 - Лекция 7

Сетевые подсистемы — получать информацию из сетей, отправлять информацию.

Обработка прерываний — самая главная.

Обработка прерываний от катастроф до нормы (обрабатчики прерываний имеют высочайший приоритет) Влияет на **отзывчивость**

Местное реальное время — не пусто с отзывчивостью.

Время обработки процесса может быть больше

POSIX определение реального времени — **TODO**

Отзывчивость — про такие ответы на действия пользователя,

которые не приводят к отладке.

2 способов обработки: top half & bottom half

Одномаршрутное действие (3 вида, см. выше - IRQ, tasklet...)

tasklet еще не выполнено

softirq определены статически (см. выше)

10 видов обработчиков в системе представлены в массиве.

```
const char * softirq_to_name[NR_SOFTIRQS] =  
= { "HI", "TIMER", ... , "HR-TIMER", "RCU" };
```

softirq-action

```
struct softirq_action  
{  
    void (*action)(struct softirq_action *);  
}
```

Когда ядро выполняет обработка отложенного прерывания, оно вызывает action функцию, передавая ей указатель.

open_softirq — инициализирует softirq

```
extern void open_softirq( int nr, void (*action)(struct softirq_action *));
```

указатель массива softirq_vec - определен аналогично to_name у которого на странице setting, который

указывает на соответствующий softirq, который

Одномаршрутное действие softirq, запускаемое в open_softirq — создается в очередь на выполнение и

антициклическая функция `raise_softirq`

```
extern void raise_softirq(unsigned int nr)
{
    unsigned long flags;
    local_irq_save(flags); // нарушает сохраняет флаг IF (Interrupt Flag)
    raise_softirq(nr); // может выполняться только с подавлением прерываний
    local_irq_restore(flags); // нарушает
}
```

Для каждого ядра есть `ksoftirq_daemon` — он работает с отложенными действиями

При запуске системы гнз каждого ядра начинет работать демон `softirq`

Иллюстрация из лабы (пример сетевого пакета)

Пакет принят → заполнение `poll_list` → `ksoftirq_daemon` вызывает `do_softirq` (пред этим инициализирует⁽²⁾)

Драйвер устанавливает (заполняет) 6-bit field, к которому обращается `ksoftirq_daemon` (7), после чего выполняется если отработал `-- do_softirq`, в котором происходит выполнение зарегистрированного обработчика (8)

NEW API (2.4 ядро — это не окно new) — ответ на вопрос как можно действовать с сетевой подсистемой — дав возможность различных действий с сетевой подсистемой, можно

Генерированию тысячи прерываний в секунду

Каждый softirq проходит через следующие этапы:

- 1) регистрируется в open-softirq
- 2) обработчик softirq отмечается как отложеный при помощи queue-softirq
- 3) отложенные softirq запускаются на выполнение в кадре следующем выше выполнения отложенных функций
- 4) завершение выполнения

Избегать softirq — только если требуется реальная высокая частота (см. макрос)

TASKLET — специальный тип softirq

Все bottom halves должны быть реализованы как tasklets (см. макрос) — они являются многопоточным аналогом bottom halves.

В 5.2: такой подход устарел — потоки IRQ запросы блокируют tasklets — речь об очередях (queue) задач!

В отличие от softirq, tasklet — только на одном процессоре (IRQ могут выполняться параллельно — см. кардиодиод — это для каждого ядра) — отсюда и увеличение скорости таких запросов.

На IRQ — местные требования к функциональности

На системный уровень нет.

Тасклет берётся от уровня go идя (не может блокироваться)

Дал 6.2.2.

tasklet_struct

```
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    bool use_callback;
    union
    {
        void (*func)(unsigned long data);
        void (*callback)(struct tasklet_struct *t);
    };
    unsigned long data;
};
```

Тасклеты могут перекрещиваться статически идинамически
Статически — 2 макроса:

DECLARE_TASKLET (name, -callback); use_callback → true
заполнение полей

DECLARE_TASKLET_DISABLED (name, -callback);

Есть ещё стирные макросы:

DECLARE_TASKLET_OLD (name, -func);

DECLARE_TASKLET_DISABLED_OLD (name, -func);

Динамически — tasklet_init

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

Планирование на выполнение — это это организует
вызов в обработчик прерывания

```
static inline void tasklet_schedule(struct tasklet_struct *t)  
{  
    if (!test_and_set_bit(TASKLET_STATE_SHED, &t->state))  
        --tasklet_schedule(); // таскет гарантированно будет выполнен  
    } // для осн. пот.  
static inline void tasklet_hi_schedule(—); // для высокоприор.
```

Если запланирован, то в roundtrips, то выполнится
и пот.

Если запланирован в roundtrips или вызов из одного
таскета — выполнение отложено до погаснет срока

Если в таскете выполнится генерик, выполнится
в другом таскете — регион использует средство
безопасности (spin-lock)

Сигнализация — не регион средство безопасности и
отложен в конец кода.

На таскетах организует
`tasklet_disable();`

— struct tasklet_struct
`tasklet_enable();`

tasklet_kill struct -os le-s-wc- #;

tasklet-setup()

В node- IRQ-SHARED — регистратор на канал IRQ,
связанный с именем ^{IRQ8} именем. Вызов tasklet из обработчика прерывания
/proc/interrupts

tasklet — например вывод временных (записи, времена);
Как правило, tasklet выполняется на том же процессоре,
на котором выполняется обработчик, его запланировавший —
— на этом процессоре заблокировано прерывание, и
заблокировано имена IRQ

19 мая 2023 — Семинар по открытым проектам

Про open:

6) условных избегать глаголов — например

6) do-filp-open 3 раза path_open() — временный if

1) с ораком ... /LOOKUP_RCU — быстрый проход, go if

головешение не окончено в этом же блоке выполняются все проверки

2) с ораком — одиничный проход или в быстром ожидании

3) с ораком ... /LOOKUP_REVOCAL

где используется дескрипторы — struct file_struct —
не копируется увеличение элементов

Рутинус expand_files() - б alloc_fd -
- отразить spin-lock-и - искоррековать переходы по массивам

files->next_fd = fd + 1 - отразить
после этого

set_open_fd - установить начальный дескриптор
generic

if CLOEXEC
 set_close_on_exec
else
 clear_close_on_exec
 - комментарии к коду

Используется, имена структура инициализированы

build_open_flags - выбором следим



O_APPEND все работает в NFS

Нет опера O_EXECL

Только O_EXCL

Кеширование

Отдельно выделено str_copy_from_user

Семафоры lock[↑] shared

Про наду на будораживший/не будораживший флаг-bitflag

stdio (cigos) — библиотека буферизованного ввода-вывода.

FILE — структура (тип, макрос)

IO_FILE — её отчёт, она унаследована

struct IO_FILE

{

...
char *_IO_buf_base;

char *_IO_buf_end;

int fileno; — одинаковый для всех файлов, получается в результате функции open

...

}

1-я программа

```
int fd = open("alphabet.txt", O_RDONLY);
FILE *fs1 = fopen(fd, "r");
char buff1[20];
setvbuf(fs1, buff1, _IOFBF, 20);

FILE *fs2 = fopen(fd, "r");
char buff2[20];
setvbuf(fs2, buff2, _IOFBF, 20);

while (...)

{
    char c;

    fscanf(fs1, "%c", &c); — считывает буфер первыми 20-ю символами
    printf(...)

    fscanf(fs2, "%c", &c); — оставшиеся 6
    printf(...)

}
...
```

Посмотреть программу между 1 главной позиции и 2-ой.

1 + 2
no rec
чтобы

2-я программа

```
int fd1 = open ("alphabet.txt", O_RDONLY);  
int fd2 = open ("alphabet.txt", O_RDONLY);  
  
while (...) {  
    read (fd1, ...)  
    write (...)  
    read (fd2, ...)  
    write (...)  
}
```

В struct file есть открытие $loff_ + f_pos$ — текущий
позиция в началом операции

struct stat

```
{  
    ...  
    inode_t st_ino;  
    ...  
    off_t st_size; // size ошибок, pos нет  
    ...  
}
```

3-я программа:

Задачи:
букреп заносы
бонбан flush
запрос

stat()

(stat())

{stat()}

```
struct stat statbuf;  
stat ("...", &statbuf);
```

struct file_struct - fcip
/
struct inode, nog rei Borgensee manus

Bogspor re error, o.. .

2 чвн 2023 — Nexus 8

struct workqueue_struct - <linux/workqueue.h>

```
struct workqueue_struct
{
    struct list_head pwqs; // pwqs pool      нын оверфлод
    struct list_head list; // list pool
    struct mutex mutex;
    ...
    struct pool_workqueue *def_pwqs; // only for unbound wqs
    char name[WQ_NAME_LEN];
    unsigned int flags -- cacheline_aligned; // WQ flags
    struct pool_workqueue *percpu *cpu_pwqs;
    struct pool_workqueue *rcu *numa_pwg_tbl[]; // unbound pwqs indexed by node
}
```

Такимеби функциональна от реална го користа в контексте предпобарв. Още гамма доста етоаке предел

Overfull parrot — б контексте очигуанској потоава спра

Отложенные действия — поисковому однодотчики
прерываний не могут выполняться долгое время
На том процессоре, где когдани выполняется, другие
прерывания запрещены, а на других процессорах некий
IRQ заблокирован

Вспоминает трёхуровневую архитектуру.

Компьютер — процессор + память (всё что?)

Всё остальное — внешние устройства, взаимодействие
с ними — через прерывание.

Минимальные действия: получение данных от ВУ,
помещение их в память ядра. Остальное — в отложенных
действиях.

Для подачи: послать worker-у —
normal — ке команды
high — ке остановки
процессор — где очередь работает.

worker — рабочий поток ядра (worker thread), или
выполнится *worklet-thread()*, отт тут идет в верхний уровень
и зачиняет (^(поток) (группировка) прерываний сю). Просыпается, когда в
очередь ставится работа. Завершив работу, зачиняет

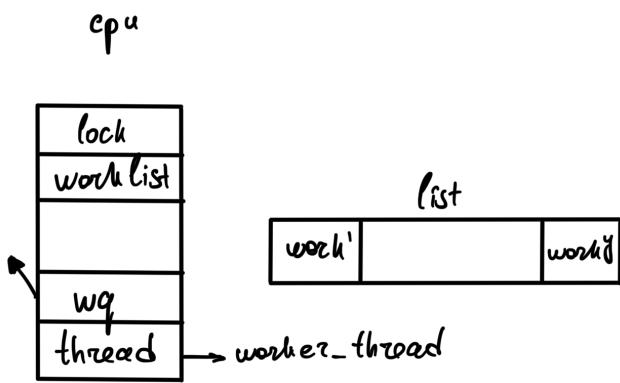
epi-work-queue-struct

```

struct cpu_work_queue_struct
{
    spinlock_t lock;
    long remove_sequence; // next to add
    long insert_sequence; // next to end?
    struct list_head worklist;
    ...
    struct workqueue_struct *wq;
    task_t *threads; // to cancel operations
    int run_depth;
}

```

Умные очереди



```

struct work_struct
{
    atomic_long data;
    struct list_head entry;
    work_func_t func;
    ...
}

typedef void (*work_func_t)
    (struct work_struct *work);

```

Очевидно, падает синхронизация операций на **alloc-workqueue**

Для v6.3.5:

```

struct workqueue_struct *alloc_workqueue(const char *fmt, unsigned int flags,
                                         int max_active, ...)

```

определяет кучу задач, которые могут быть выполнены
одновременно в одном
workqueue на всех
процессорах

alloc_ordered_workqueue

```
#define alloc_ordered_workqueue(lmt, flags, cargs, ...)
    alloc_workqueue((lmt), WQ_UNBOUND)-WQ_ORDERED)
    -WQ_ORDERED_EXPLICIT) (flags), l, #cargs)

    отработку не связана
    с каким CPU
    Это WQ_FREEZABLE -
    - может блокироваться
```

create_workqueue — устарело

Флаги:

WQ_UNBOUND — очередь не связана ни с каким CPU

WQ_FREEZABLE — может блокироваться

WQ_HIGHPRI

WQ_CPU_INTENSIVE — рабочая очередь может интенсивно использовать процессор

...

WQ_POWER_EFFICIENT — имеет более высокую производительность благодаря локальному кешу (т.к. связана с процессором). Подобный подход — расход энергии (т.к. процессор не простаивает)

Стандартные определения (работы):

Статический — макро **DECLARE_WORK**

DECLARE_WORK (name, void (*func)(void *))
 или экземпляр структуры work_struct (name)
 под, который необходимо выполнить
 (bottom_half)

Динамический (в процессе выполнения) **INIT_WORK**

```

#define INIT_WORK(_work, _func)
    __INIT_WORK(_work, _func, 0)
        ^ это шаг есть в manyone
--INIT_WORK(_work, _func, _onstack)
do {
    ...
    INIT_LIST_HEAD(&(_work->entry));
    (_work)->func = (_func);
} while (0);           если some инициализировано

```

Также есть **PREPARE_WORK()**, но нужно использовать

INIT_WORK() — это инициализирует donee текущего

После инициализации — можно писать в очередь — гля

этото **queue-work()**

```

bool
int queue_work(struct work_queue_struct *queue, struct work_struct *work);
                ^ для инициализирования
int queue_delay_work(..., ..., unsigned long delay);
                ^ время, через которое работа поставлена в
                    очередь (т.е. поставлена в очередь)
                    количество jiffies (микровекции)

```

Также есть **queue-work-on(...)** — через реи онработка

queue-work()

drain-work-queue() — смыть?

либо **flush:**

```

extern void _flush_workqueue(struct workqueue_struct *wq);    прерывается приостановка
                                                                завершение очереди
extern bool flush_work(struct work_struct *work);    завершение
                                                                работы
extern bool cancel_work(...);    чтобы ун то что завершить
                                (события)

```

flush_delay_work
cancel_delay_work

Существуют асинхронные очереди, то про них то говорят

В node — 2 реда, и в 1 очередь одна должна блокироваться (это проработано блокировкой, состоящее из прер. стк)

Можно сделать и не в таблице (scan code)

Про spin блокировки

Её может удерживать только 1 поток.

Если поток покидает зону, а её удерживает другой, то блокировка находится в состоянии **contended**, то говорит, что в состоянии конкурирует (уме засвистеть), и поток начинает усил проверки (**busy loop**), проверяя, не освободилась ли блокировка

Особенности

- но есть, быстрая блокировка, её передача удерживается некоторое время, т.к. в spin-блокировке выполняется **test-and-set**, блокируется memory barrier \Rightarrow 1 поток забывает оценку памяти \Rightarrow покинутое

правоохранительности и следств. Поэтому (см. Гендер)

вводится доп. учрн, внутри него учрн отдача по простой переменной.

- активное отдаческое, переключение контекста тои, но нечто вовремя время блокировки, сравнив с переключением контекста 2 раза \Rightarrow

издражение spin-блоки. на время, \leq 2-м переключением контекста

- на 1-процессорной машине spin-блокировки не комбинируются, их просто не существует. Они играют роль мониторов, запрещающих се разбр. блокировок пода в режиме ядра (примитивы) Ядро воссозданное, примитивное. Если примитивы отмогаются, то блок. не комбинируются

- время блокировки \leq время работы сист. мониторов

- в Linux блокировки не ресурссивно (чтобы не возникло временной блокировки)

- могут использоваться в обработчиках прерываний

6 отмече о синхрониз., т.к. они о н.

Если так, то придется или засыпать в другой
место, необходимо запретить все возможное
прерывание. Или же:

- 1) одн.прер. прерывает код ядра, успе удерживавшего
искусственное блокировку
- 2) пытаются заблокировать блокировку, но не может — deadlock

Ядро предоставляет интерфейс для засыпки
блокировки (с ожиданием завершения прерываний)

```
spin_lock_irqsave(...);  
// кр. сенсор  
spin_lock_irqrestore(...);
```

Есть ещё функции, но они не в библиотеке

Рекомендации:

- никогда блокировка должна быть связана с тем,
что блокирован
- блокировкам оставлять, а не коф

Обратите внимание на то что в struct `inode` есть поле `struct rw_semaphore i_rwsem;`

Специальные файлы
устройства

Обеспечивают усовершенствованный доступ и блокировку
(т.е. передачей семафоров) устройствам

связь между файловой системой и
устройствами устройств

Номер дескр. открыт /закрыт, можно читать и писать.

У каждого устройства 1 специальный — открывается в /dev

Подкаталог /dev/fd содержит 0, 1, 2
(stdin, stdout, stderr)

2 типа файлов устройств:

символичные — не физически存在的
блочное — физически存在的

Связь между спец. файлами и устройствами — через inode.

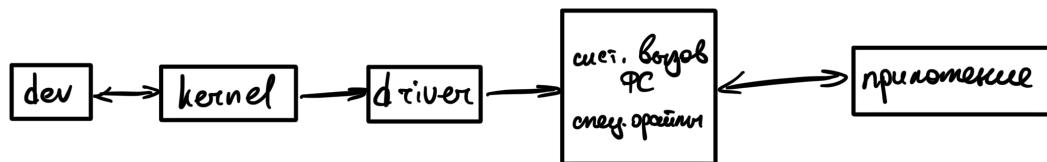
В нём есть none dev-t i_{real} device

Так спец. файла загаёт none inode → mode

Для блок-ориентированного устройства в этом none
маска 060000

Блок — описан правильного 020000

Взаимодействие с файлами по специальной
системе



Целой путь от пользователя к устройству проходит по
всем устройствам

fprintf — отображение и блокировка устройства
fscanf — и ...

Более подробное представление номеров устройств

<linux/types.h>

dev_t где номера устройств

Номер м.д. старшим
младшим

С версии 2.6.0 dev_t - 32-х разрядное значение

16 бит старшего, 16 бит младшего

Стандарт POSIX.1 определяет структуру блочного тайпа, то не обновляет его формат

Дисковое агр. нап. имеет раздельно, или скомбинировано где монтируется PC.

Device - устройство, имеет старший номер (major ID)

По сути, это идентификатор массы устройства.

Команды работы диска будут иметь младший номер (minor ID)

Но это потребует не самого делать предположений о внутренней реализации номеров, а использовать

указанные в <linux/hddev-t.h> - они

То есть итоги получать старшую и младшую часть типа dev-t

MAJOR, MINOR

MAJOR (dev_t dev);

MINOR (dev_t dev);

Если плюснуть → старш. + младш. надо б. dev-t, то

MKDEV MKDEV(int major, int minor);

Do версии 2.6 количество кемеров было ограничено.

255 старших + 255 младших

Выделение и освобождение кемеров устройств

Одним из первых действий драйвера (например, при установке сист. устр). — получить 2 или более кемеров устройств, чтобы работать с ними

Символьные — так просто проще.

Блоки устройства — ус-ба второй линии.

Для этого ведут организацию register-ch dev-region

Обычно выдавалось в --exit

Старые номера

Некоторые изображаются устройствам статически.

Перечень таких устройств - в

Documentation/devices.txt , например

Старый номер	Тип устройства
1	опер. память
2	дисковод
3	первый контроллер твёрдых IDE-дисков
4	теракартка
5	.
6	принтер (нарв. порт) ?
7	
8	SCSI-диски

1

Модем

14

звук. карты

22

второй контроллер жёсткого
IDE диска

Основное отличие конфигурации в
настройках. Для сетевых карт

{eth0, eth1 и т.д.

В настройке /dev/no ls -l, то вместо
размера определяется 2 числа через запятую

cat /proc/devices - список устройств,
включая скрытые.