Билет №1

1. Управление внешними устройствами: специальные файлы устройств, адресация внешних устройств и их идентификация в системе, тип dev_t.

unix/linux рассматривает внешние устройства как специальные файлы.

Специальные файлы устройств обеспечивают единообразный доступ к внешним устройствам. Эти файлы обеспечивают связь между файловой системой и драйверами устройств. В отличие от обычных файлов, специальные файлы устройств в действительности являются только указателями на соответствующие драйверы устройств в ядре. Такая интерпретация специальных файлов обеспечивает доступ к внешним устройствам как к обычным файлам. Так же как обычный файл, файл устройства может быть открыт, закрыт, в него можно писать или из него можно читать. По сравнению с обычными файлами файлы устройств имеют три дополнительных атрибута, которые характеризуют устройство, соответствующее данному файлу:

*** Класс устройства**. В ОС Linux различают устройства блок-ориентированные и байт-ориентированные. Блок-ориентированные (или блочные) устройства, например, жесткий диск, передают данные блоками. Байт-ориентированные (или символьные) устройства, например, принтер и модем, передают данные посимвольно, как непрерывный поток байтов.

*** Старший номер устройства**, обозначающий тип устройства, например, жесткий диск или звуковая плата.

*** Младший номер устройства** применяется для нумерации устройств одного типа, т. е. устройств с одинаковыми старшими номерами.

% каждому внешнему устройству ОС ставит в соответсвие минимум 1 специальный файл эти файлы находятся в каталоге /dev корневой файловой системы подкаталог /dev/fd содержит файлы с именами 0, 1, 2 %

в ОС имеются 2 типа специальных фалов устройств:

1. символьный

2. блочный

система должна идентифицировать внешнее устройство для этого используется старший и младший номера устройств (major/minor). (символьные и блочные устройства представляются парой чисел: <major>:<minor>.) (идентифицируется так называемым старшим и младшим номером идентифицируется типом dev_t)

Старший номер устройства определяет драйвер, связанный с устройством, т.е. это номер драйвера. Ядро использует старший номер устройства для диспетчеризации запроса на нужный драйвер.

Младший номер устройства используется самим драйвером, чтобы различать отдельные физические или логические устройства.

т. к. специальные файлы устройств это файлы => они имеют inode, т.е. описываются в системе соответствующим индексным дескриптором. (Следуя парадигме UNIX в UNIX все – файл, внешние устройства представляются в системе как специальные файлы и имеют inode. inode файла содержит метаданные о файле. Приложения обращаются к символьным и блочным устройствам через inode. Когда создается inode устройства, он сопоставляется с номерами major и minor.)

в struct inode

```
struct inode
{
        …
        dev_t i_rdev; /*фактический номер устройства, содержащий major, minor*/
        strut t list_head i_devices;
        union
        {
                struct pipe_inode_into *i_pipe;
                struct block_device *i_bdev;
                struct cdv *i_cdev;
                char *_link;
                unsigned i_dir_seq;
        }; // отражает перечисление специальных файлов
        ...
};
```

Для хранения номеров устройств, как старшего так и младшего, в ядре используется тип **dev_t**, определённый в ***<linux/types.h>***. Начиная с версии ядра 2.6.0, **dev_t** является 32-х разрядным, 12 бит отведены для старшего номера и 20 - для младшего.

2. Система прерываний: типы прерываний и их особенности. Прерывания в последовательности ввода-вывода – обслуживание запроса процесса на ввод-вывод (диаграмма). Быстрые и медленные прерывания. Обработчики аппаратных прерываний: регистрация в системе – функция и ее параметры, примеры. Тасклеты – объявление, планирование (пример лаб.раб).

Классификация прерываний:
* Синхронные
        -системные вызовы
        -исключения (исправимые/неисправимые)
* Ассинхронные (не зависят от других работ выполняющихся в системе)

-аппаратные прерывания (прерывание от системного таймера/прерывание от системного таймера/от устройств ввода — вывода) (они выполняются на высоких уровнях привилегий и их выполнение нельзя прервать)

Диаграмма
SVC - supervisor call
IH - interrupt handler

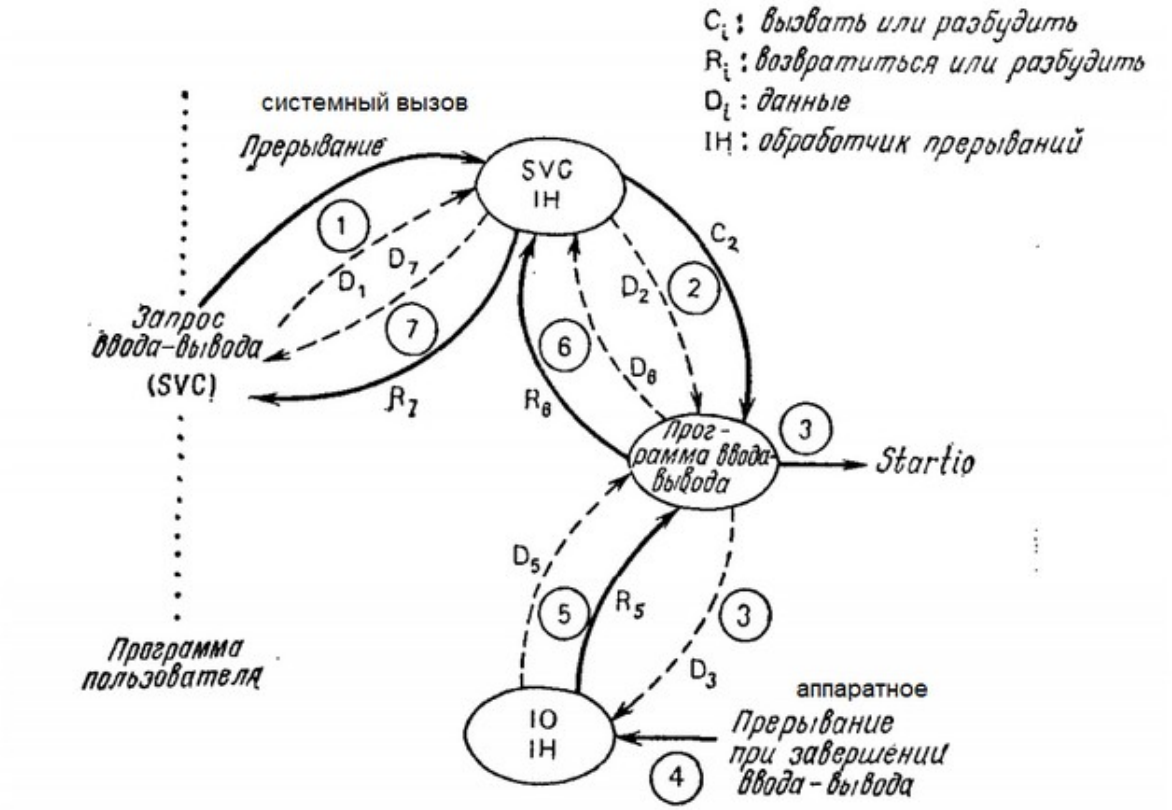
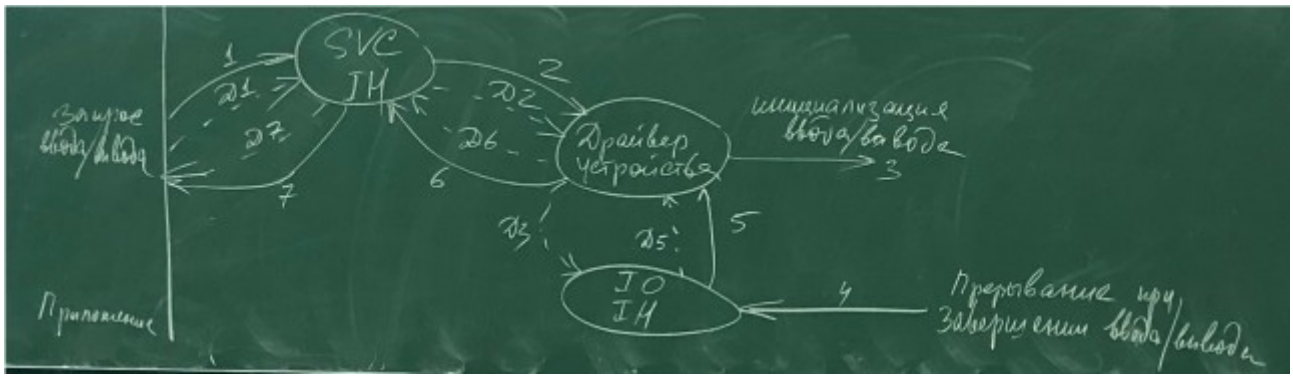


Рис. 4.5. Управленне прерываннями в последовательности ввода-вывода.



Сигналы прерывания от устройств ввода-вывода поступают на входы IRQ (Interrupt Request), а контроллер прерывания формирует сигнал прерывания, который по шине управления (линии INTR) поступает на соответствующую ножку (pin) процессора. Сигнал прерывания будет передан процессору, если он не замаскирован, т.е. его обработка разрешена.

% в конце цикла выполнения каждой команды контроллер проверяет наличие сигнала
прерывания на своем входе
если сигнал поступил, то процессор переходит на выполнение обработчика прерывания и
это обработчик аппаратного прерывания
все прерывания ввода/вывода - аппаратные + прерывания от системного таймера %

Быстрые прерывания
В ОС Linux принято различать быстрые и медленные прерывания. Быстрые прерывания выполняются при запрете всех прерываний на текущем процессоре. На других процессорах прерывания могут обрабатываться, но при запрете прерываний по линии IRQ, относящейся к выполняемому быстрому прерыванию. Таким образом, выполнение быстрого прерывания не может быть прервано. В современной ОС Linux быстрым прерыванием является только прерывание от системного таймера.

Медленные прерывания
Медленные прерывания могут потребовать значительных затрат процессорного времени. Чтобы сократить время выполнения обработчиков прерываний обработчики медленных аппаратных прерываний делятся на две части, которые традиционно называются верхняя (top) и нижняя (bottom) половины (half).

Верхними половинами остаются обработчики, устанавливаемы функцией request_irq() на определенных IRQ. Выполнение нижних половин инициируется верхними половинами, т.е. обработчиками прерываний.

С современных ОС Linux имеется три типа нижних половин (bottom half):

- softirq – отложенные прерывания;
- tascklet – тасклеты;
- workqueue – очереди работ.

Драйверы регистрируют обработчик аппаратного прерывания и разрешают определенную линию irq посредством функции:

<linux/interrupt.h>

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

где: irq – номер прерывания, *handler –указатель на обработчик прерывания, irqflags – флаги, devname – ASCII текст, представляющий устройство, связанное с прерыванием, dev_id – используется прежде всего для разделения (shared) линии прерывания.

Тасклеты

Тасклеты — это механизм обработки нижних половин, построенный на основе механизма отложенных прерываний.

Тасклеты описываются структурой:

```
struct tasklet_struct
{
        struct tasklet_struct *next; /* указатель на следующий
тасклет*/
        unsigned long state; /* текущее состояние*/
        atomic_t count; /* счетчик ссылок*/
        void (*func)(unsigned long); /* обработчик*/
        unsigned long data; /* данные*/
};
```

Поле state может иметь одно из следующих значений:

```
enum
{
    TASCLET_STATE_SCHED, /* запланирован на выполнение*/
     TASKLET_STATE_RUN /* выполняется – только для SMP*/
}
```

Если поле count = 0, то тасклет разрешен и может выполняться, если он помечен как запланированный, иначе тасклет запрещен и не может выполняться.

Объявление:

Тасклеты могут быть зарегистрированы как статически, так и динамически.

Статически тасклеты создаются с помощью двух макросов:

```
DECLARE_TASKLET(name, func, data)
DECLARE_TASKLET_DISABLED(name, func, data);
```

Оба макроса статически создают экземпляр структуры struct tasklet_struct с указанным именем (name).

При динамическом создании тасклета объявляется указатель на структуру struct tasklet_struct *t а затем для инициализации вызывается функция:

```
tasklet_init(t , tasklet_handler , dev) ;
```

Планирование:

Тасклеты могут быть запланированы на выполнение с помощью Функций:

```
tasklet_schedule(struct tasklet_struct *t);
tasklet_hi_sheduler(struct tasklet_struct *t);
void tasklet_hi_schedule_first(struct tasklet_struct *t); /* вне очереди */
```

Когда tasklet запланирован, ему выставляется состояние **TASKLET_STATE_SCHED**, и

он добавляется в очередь. Пока он находится в этом состоянии, запланировать его еще раз не получится — в этом случае просто ничего не произойдет. Tasklet не может находиться сразу в нескольких местах в очереди на планирование, которая организуется через поле next структуры tasklet_struct.
После того, как тасклет был запланирован, он выполниться один раз.

Пример из ЛР:

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

MODULE_LICENSE( "GPL" );
MODULE_AUTHOR( "Anastasia Neklepaeva" );

static int irq = 1;
static int dev_id;
void tasklet_func( unsigned long data );
char tasklet_data[] = "tasklet_func was called";

// keyboard controller
#define DFN_IRQ 1

// статическое создание тасклета
DECLARE_TASKLET( tasklet, tasklet_func, (unsigned long) &tasklet_data );

// обработчик прерывания
static irqreturn_t irq_handler( int irq, void *dev_id )
{
   if (irq == DFN_IRQ)
   {
      // запланирование тасклета на выполнение
      tasklet_schedule( &tasklet );
        return IRQ_HANDLED; // прерывание обработано
   }
   else return IRQ_NONE; // прерывание не обработано
}

void tasklet_func(unsigned long data)
{
   // получение кода нажатой клавиши клавиатуры
   int code = inb(0x60);
   char * ascii[84] =
   {" ", "Esc", "1", "2", "3", "4", "5", "6", "7", "8", "9", "0", "-", "+", "Backspace",
```

```
    "Tab", "Q", "W", "E", "R", "T", "Y", "U", "I", "O", "P", "[", "]", "Enter", "Ctrl",
    "A", "S", "D", "F", "G", "H", "J", "K", "L", ";", "\"", "`", "Shift (left)", "|",
    "Z", "X", "C", "V", "B", "N", "M", "<", ">", "?", "Shift (right)",
    "*", "Alt", "Space", "CapsLock",
    "F1", "F2", "F3", "F4", "F5", "F6", "F7", "F8", "F9", "F10",
    "NumLock", "ScrollLock", "Home", "Up", "Page-Up", "-", "Left",
    " ", "Right", "+", "End", "Down", "Page-Down", "Insert", "Delete"};
   if (code < 84)
   {
      printk("+ tasklet: keyboard %s\n", ascii[code]);
   }
}

static int __init md_init( void )
{
   // регистрация обработчика аппаратного прерывания
        int rc = request_irq(irq, irq_handler, IRQF_SHARED, "my_irq_handler",
&dev_id);
   if (rc)
   {
      printk("+ tasklet: register interrupt handler error!\n" );
      return rc;
   }
   printk("+ tasklet: module loaded!\n" );
   return 0;
}

static void __exit md_exit( void )
{
   tasklet_kill( &tasklet );
   free_irq( irq, &dev_id );
   printk("+ tasklet: module unloaded!\n" );
}

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

Билет №2

1. Управление внешними устройствами: специальные файлы устройств, идентификация внешних устройств в системе (тип dev_t), символьные и блочные устройства и их inode (структуры, описывающие символьные и блочные устройства).

См. билет №1
+

struct cdev является одним из элементов структуры inode, которая используется ядром для представления файлов. Структура cdev - это внутренняя структура ядра, которая представляет символьные устройства.
Структура struct block_device является одним из элементов структуры inode, которая используется ядром для представления файлов. Структура block_device - это внутренняя структура ядра, которая представляет блочные устройства. Это поле содержит указатель на эту структуру, когда inode ссылается на файл устройства типа block_device.

2. Система прерываний: типы прерываний и их особенности. Быстрые и медленные прерывания. Обработчики аппаратных прерываний: регистрация в системе, примеры. Верхние и нижние половины обработчиков прерываний. Нижние половины: тасклеты и очереди работ – объявление, создание, постановка работы в очередь, планирование (пример лаб. Раб).

см. билет №1
+

Очередь заданий является еще одной концепцией для обработки отложенных функций. Это похоже на тасклет с некоторыми отличиями. Функции рабочих очередей выполняются в контексте процесса ядра, но функции тасклетов выполняются в контексте программных прерываний.

Рабочая очередь поддерживается типом struct work_struct, который определён в include/linux/workqueue.h:

```
struct work_struct {
        atomic_long_t data;
        struct list_head entry;
        work_func_t func;
    #ifdef CONFIG_LOCKDEP
        struct lockdep_map lockdep_map;
    #endif
};
```

Обратим внимание на два поля: func - функция, которая будет запланирована в рабочей очереди, и data - параметр этой функции. Ядро Linux предоставляет специальные потоки для каждого процессора, которые называются kworker

Очередь работ создается функцией (см. приложение 1):

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

- name - имя очереди, но в отличие от старых реализаций потоков с этим именем не создается
- flags - флаги определяют как очередь работ будет выполняться
- max_active - ограничивает число задач из данной очереди, которые могут одновременно выполняться на одном CPU.

Может использоваться вызов create_workqueue();
work_struct представляет задачу (обработчик нижней половины) в очереди работ.

Поместить задачу в очередь работ можно во время компиляции (статически):

DECLARE_WORK(name, void (*func)(void *));
где: name – имя структуры work_struct, func – функция, которая вызывается из workqueue – обработчик нижней половины.

Если требуется задать структуру work_struct динамически, то необходимо использовать следующие два макроса:
INIT_WORK(sruct work_struct *work, void (*func)(void),void *data);
PREPARE_WORK(sruct work_struct *work, void (*func)(void),void *data);

После того, как будет инициализирована структура для объекта work, следующим шагом будет помещение этой структуры в очередь работ. Это можно сделать несколькими способами. Во-первых, просто добавить работу (объект work) в очередь работ с помощью функции queue_work (которая назначает работу текущему процессору). Можно с помощью функции queue_work_on указать процессор, на котором будет выполняться обработчик.

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

MODULE_LICENSE( "GPL" );
MODULE_AUTHOR( "Anastasia Neklepaeva" );

static int irq = 1;
static int dev_id;
struct workqueue_struct *workqueue;
void work_func(struct work_struct *w);

// keyboard controller
#define DFN_IRQ 1

// поместить задачу в очередь работ
DECLARE_WORK(work, work_func);

// обработчик прерывания
static irqreturn_t irq_handler( int irq, void *dev_id )
{
   if (irq == DFN_IRQ)
   {
      // добавление задачи в очередь работ
      queue_work(workqueue, &work);
         return IRQ_HANDLED; // прерывание обработано
   }
   else return IRQ_NONE; // прерывание не обработано
}
```

```
void work_func(struct work_struct *w)
{
   // получение кода нажатой клавиши клавиатуры
   int code = inb(0x60);
   char * ascii[84] =
   {" ", "Esc", "1", "2", "3", "4", "5", "6", "7", "8", "9", "0", "-", "+", "Backspace",
    "Tab", "Q", "W", "E", "R", "T", "Y", "U", "I", "O", "P", "[", "]", "Enter", "Ctrl",
    "A", "S", "D", "F", "G", "H", "J", "K", "L", ";", "\"", "`", "Shift (left)", "|",
    "Z", "X", "C", "V", "B", "N", "M", "<", ">", "?", "Shift (right)",
    "*", "Alt", "Space", "CapsLock",
    "F1", "F2", "F3", "F4", "F5", "F6", "F7", "F8", "F9", "F10",
    "NumLock", "ScrollLock", "Home", "Up", "Page-Up", "-", "Left",
    " ", "Right", "+", "End", "Down", "Page-Down", "Insert", "Delete"};
   if (code < 84)
   {
      printk("+ workqueue: keyboard %s\n", ascii[code]);
   }
}

static int __init md_init( void )
{
   // регистрация обработчика аппаратного прерывания
   int rc = request_irq(irq, irq_handler, IRQF_SHARED, "my_irq_handler", &dev_id);
   if (rc)
   {
      printk("+ workqueue: register interrupt handler error!\n" );
      return rc;
   }

   //создание очереди работ
   workqueue = create_workqueue( "workqueue" );

   if (!workqueue)
   {
      printk("+ workqueue: create_workqueue error!\n" );
      return -1;
   }

   printk("+ workqueue: module loaded!\n" );
   return 0;
}

static void __exit md_exit( void )
{
   flush_workqueue(workqueue);
   destroy_workqueue(workqueue);
   free_irq( irq, &dev_id );
   printk("+ workqueue: module unloaded!\n" );
}

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

Билет №3

1. Модели ввода-вывода: представление с помощью диаграмм, описание и особенности. Классификация моделей ввода-вывода.

В Unix/Linux поддерживается 5 моделей ввода-вывода. Речь идет о моделях ввода-вывода с точки зрения программиста.

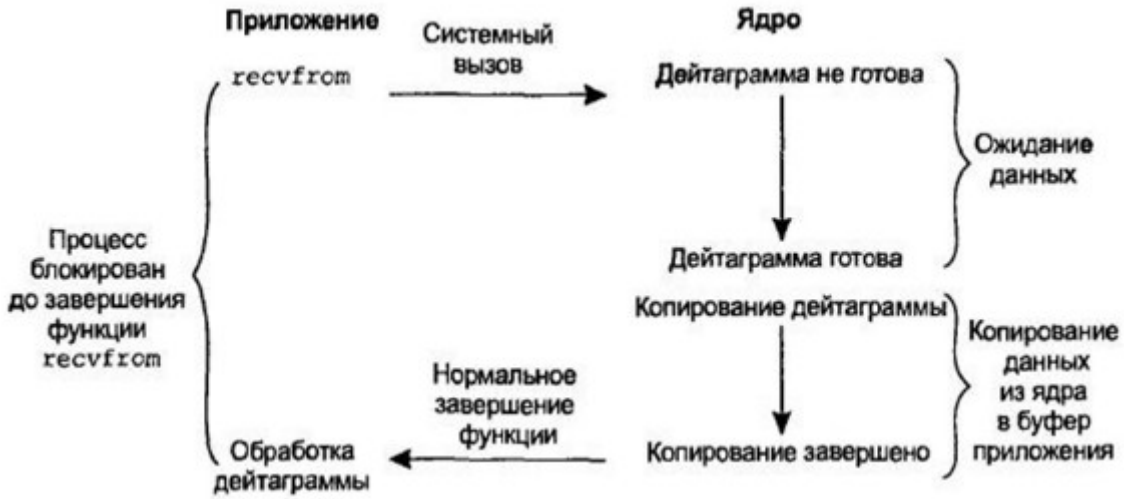
Пять моделей ввода-вывода:

- блокирующий ввод-вывод (blocking I/O)
- неблокирующий ввод-вывод (nonblocking I/O)
- ввод-вывод с мультиплексированием (I/O multiplexing: select and poll)
- ввод-вывод, управляемый сигналом (signal driven I/O: SIGIO)
- асинхронный ввод-вывод (asynchronous I/O: the POSIX aio functions)

Обычно есть две отдельные фазы для операции ввода:
1.Ожидание готовности данных
2.Копирование данных из ядра в буфер процесса

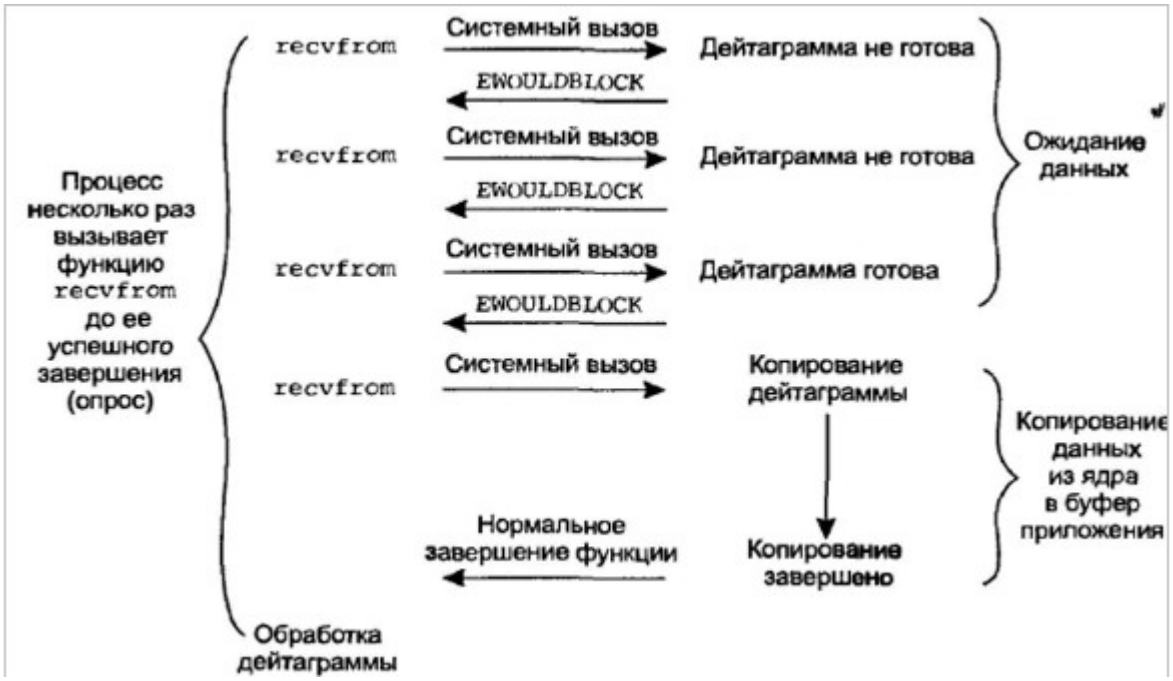
Модель блокирующего ввода-вывода

Самая распространенная модель ввода-вывода.При блокирующем вводе-выводе процесс блокирован все время от момента, когда был выполнен вызов recvfrom, до момента, когда дейтаграмма поступает и копируется в буфер приложения. Системный вызов может передать в буфер приложения данные или ошибку. При успешном завершении вызова приложение обрабатывает полученные данные.



Модель неблокирующего ввода-вывода

Неблокирующий ввод-вывод не ожидает наличия данных (или возможности вывода), а получает результат выполнения операции или невозможность её выполнения в данный момент, что определяется по коду возврата.



Первые три раза системный вызов на чтение данные не возвращает, а возвращает ошибку EWOULDBLOCK. Следующий системный вызов выполняется успешно, так как данные готовы для чтения. Ядро скопирует данные в буфер приложения и будут обработаны.

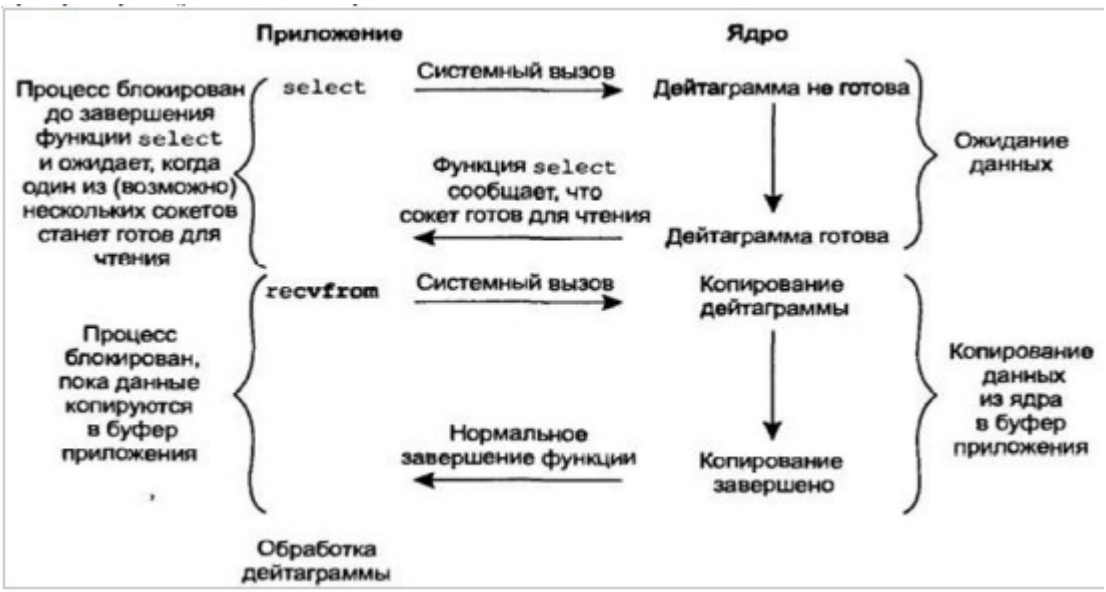
Такой режим работы называется опросом (polling). Для организации опроса может быть создан цикл, вызыв соотв сист вызов. Такой подход приводит к большим накладным расходам — пустой трате проц времени.

Модель с мультиплексирование ввода-вывода

При мультиплексировании ввода-вывода вызываются, так называемые мультиплексоры, select или poll и вместо блокировки в непосредственном системном вызове ввода-вывода процесс блокируется на одном из этих системных вызовов.

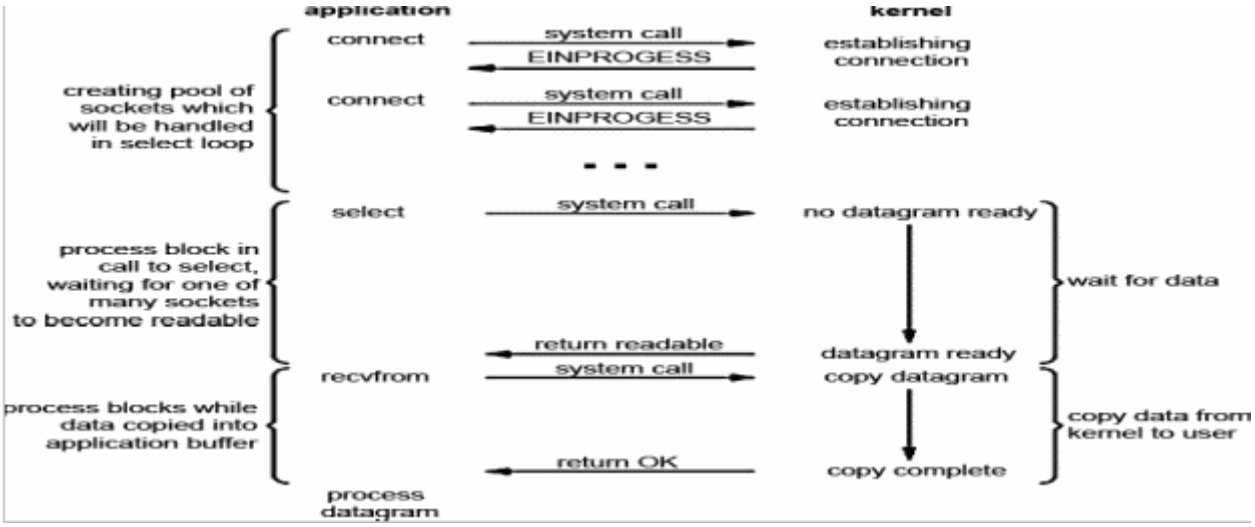
мультиплексор; коммутатор – устройство, которое объединяет информацию, поступающую по нескольким каналам ввода, и выдает ее по одному выходному каналу; *мультиплексирование; уплотнение* – совмещение нескольких сообщений, передаваемых одновременно в одной физической или логической среде; существует два основных типа мультиплексирования: временное мультиплексирование и частотное мультиплексирование.

Схематически изображена последовательность действий на стороне сервера при мультиплексировании.



Приложение блокируется при вызове select, ожидая когда сокет станет доступным для чтения. Затем ядро возвращает приложению статус *readable*, сообщая, что можно получать данные помощью *recvfrom*. Снова блокировка, да еще вместо одного два системных вызова (select и recvfrom), что увеличивает накладные расходы.

Преимущества: возможность ожидания данных не от одного, а сразу от нескольких файловых дескрипторов. Время блокировки снижается, что видно из поясняющего рис. ниже.



Клиенты пытаются создать соединение с сервером, вызывая системный вызов connect. В результате создается пул (pool) дескрипторов сокетов. Получение процессом-клиентом EINPROGRESS означает, что соединение устанавливается. На работу сервера получение клиентом EINPROGRESS никак не влияет, так как мультиплексор обработает первое соединение, которое произойдет: в цикле проверяются все сокеты и берется первый, который готов. Пока соединение принимается (accept), поступают другие соединения. Таким образом снижается время простоя, так как первый раз ожидание может затянуться, но последующие соединения требуют значительно меньших задержек.

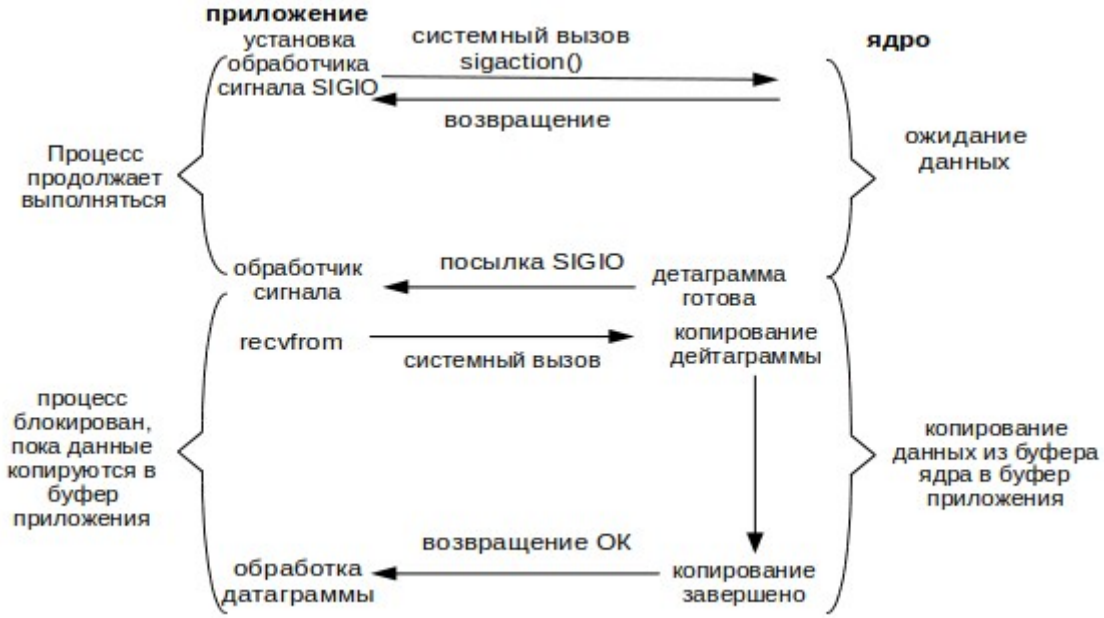
Несколько потоков или процессов для обработки запросов к серверу

Создается несколько потоков или процессов, в каждом из которых выполняется блокирующий ввод-вывод. Недостатки — дорогие потоки в линукс(большое количество накладных расходов)

Модель ввода-вывода, управляемого сигналом

В этой модели используются сигналы. Когда данные готовы к считыванию ядро должно уведомить приложение и послать сигнал SIGIO.

При использовании данного способа на сетевом сокете необходимо установить режим ввода-вывода, управляемый сигналом и обработчик сигнала, используя системный вызов **`sigaction()`**.



Особенностью модели ввода-вывода, управляемого сигналом, является то, что приложение не блокируется, а продолжает работать несмотря на то, что данные не готовы.

Результат выполнения системного вывода sigaction() возвращается сразу и приложение не блокируется. Всю работу берет на себя ядро:

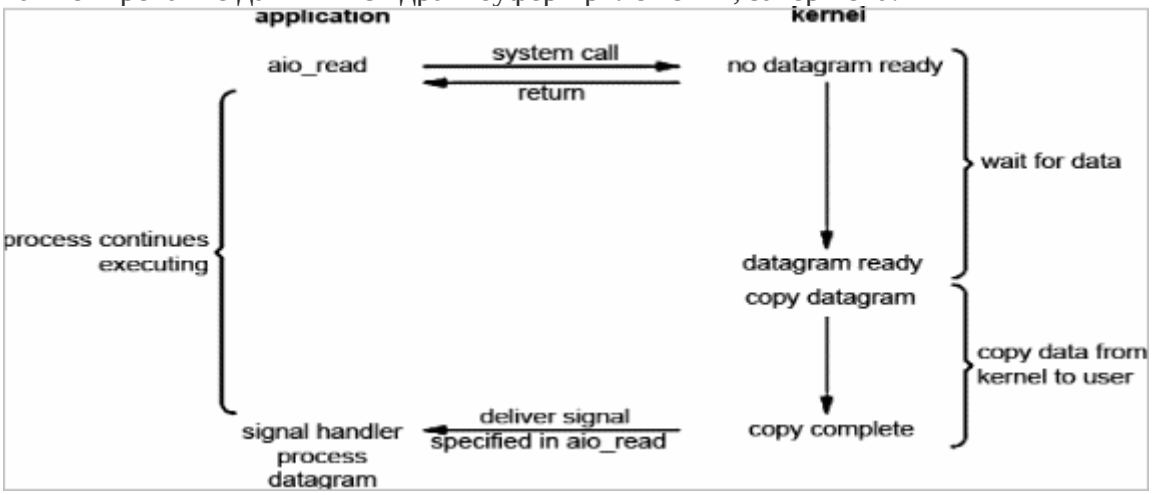
- отслеживает готовность данных
- затем посылает сигнал SIGIO, который вызывает установленный на него обработчик (функция обратного вызова – callback).

Сам вызов recvfrom() может быть выполнен либо в обработчике сигнала, который отправляет основному циклу информацию, что данные готовы, либо в основном потоке программы. Ожидание может выполняться в цикле, который выполняется в основном потоке.

Сигнал SIGIO для каждого процесса может быть только один => за один раз можно обработать только один файловый дескриптор. Во время выполнения обработчика сигнала сигнал блокируется. Если за время блокировки сигнал доставляется несколько раз, то они теряются. Если маска сигнала sa_mask = NULL, то во время выполнения обработчика другие сигналы не блокируются.

Модель асинхронного ввода-вывода

Осуществляется с помощью специальных системных вызовов. Эти функции работают так, что сообщают ядру о начале операции и уведомляют приложение, когда вся операция, включая копирование данных из ядра в буфер приложения, завершена.



В вызове **`aio_read()`** даётся указание ядру начать операцию ввода-вывода, и указывается, каким сигналом уведомить процесс о завершении операции (включая копирование данных в пользовательский буфер). При этом вызывающий процесс не блокируется, а результат операции может быть обработан в обработчике сигнала. Разница с предыдущей моделью ввода-вывода, который управляется сигналом, состоит в том, что в предыдущей модели сигнал уведомляет о возможности начала операции (вызове операции чтения), а в асинхронной модели сигнал уведомляет уже о завершении операции копирования данных в буфер пользователя.

Параметры функций: файловый дескриптор, адрес буфера, номер сигнала **`sigev_signo`** и функция, используемая для уведомления потока.

В основном в асинхронном вводе-выводе внимание сосредотачивается на двух моментах:

- на возможности определить, что ввод-вывод можно выполнить быстро;
- на завершении операции ввода-вывода в любом случае: при невозможности выполнения ввода-вывода сразу возвращается ошибка.

Проблема асинхронного ввода-вывода состоит в том, что необходимо получать результаты асинхронных действий синхронно, т.е. асинхронные операции в итоге должны доставлять процессам данные, а процессы должны иметь возможность получать эти данные по необходимости.

Классификация моделей ввода-вывода

Операции ввода-вывода могут быть блокирующими и неблокирующими, синхронными и асинхронными

	Блокирующие	**Неблокирующие**
Синхронные	read/write	**read/write (O_NONBLOCK)**
Асинхронные	I/O multiplexing (select/poll)	AIO

В соответствии с данной классификацией:

- блокирующий ввод-вывод – блокирующий синхронный;
- неблокирующий ввод-вывод или опрос (polling) – синхронный неблокирующий;
- мультиплексированный ввод-вывод – блокирующий асинхронный;
- асинхронный ввод-вывод – неблокирующий асинхронный

Ввод-вывод, управляемый сигналами, относится к неблокирующему асинхронному, но при получении сигнала о готовности данных вызывается блокирующий системный вызов.

Блокирующий синхронный ввод-вывод является наиболее обычным типом ввода-вывода: при обращении процесса к внешнему устройству с помощью системного вызова ввода-вывода процесс блокируется в ожидании завершения операции ввода-вывода; ….

Неблокирующем синхронном вводе-выводе приложение, выдавшее запрос ввода-вывода не блокируется, а делает многочисленные запросы, проверяя готовность данных.

Асинхронный ввод-вывод (AIO) - это метод выполнения операций ввода-вывода таким образом, что процесс, выдавший запрос ввода-вывода, не блокируется до завершения операции.

2. Мультиплексирование при взаимодействии процессов в распределенных системах по модели клиент-сервер. Сетевой стек. Примеры мультиплексоров и пример из лабораторной работы.

Мультиплексирование применяется:
* когда клиенты обрабатывают множество дескрипторов
* если сервер TCP обрабатывает и прослушиваемый сокет и присоед сокет
* если сервер обраб/раб и с TCP и UDP(??)
* если сервер обраб несколько служб

В модели клиент-сервер роли определены: сервер предоставляет ресурсы и службы одному или нескольким клиентам, которые обращаются к серверу за обслуживанием.

Сокеты являются универсальным средством межпроцессного взаимодействия в том смысле, что они могут использоваться как для взаимодействия процессов на отдельно стоящей машине, так и для взаимодействия процессов в сети. Для поверки состояния неблокирующих сокетов можно использовать мультиплексирование и воспользоваться функциями **select**(2) или poll() и их современные версии pselect и epoll.

```
struct socket {
        socket_state            state;

        kmemcheck_bitfield_begin(type);
        short                   type;
        kmemcheck_bitfield_end(type);

        unsigned long           flags;

        struct socket_wq __rcu  *wq;

        struct file             *file;//указатель на дескриптор открытого файла
        struct sock             *sk;[1]
```

---

1 Сетевой стек ядра Linux имеет две структуры:

- struct socket, как правило, хранится в переменной sock

```
        const struct proto_ops        *ops;
    };
```

Код клиента:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <time.h>
#include <string.h>

#define PORT 3425
#define BUF_SIZE 256
#define COUNT 5
#define SOCK_ADDR "localhost"

int main(int argc, char ** argv)
{
   int sock;
   struct sockaddr_in serv_addr;
   struct hostent *host;
   char buf[BUF_SIZE];
   char message[BUF_SIZE];

   sock = socket(AF_INET, SOCK_STREAM, 0);
   if (sock < 0)
   {
    printf("socket() failed: %d", errno);
    return EXIT_FAILURE;
   }

   host = gethostbyname(SOCK_ADDR);
   if (!host)
   {
      perror("gethostbyname() failed: ");
      return EXIT_FAILURE;
   }

   serv_addr.sin_family = AF_INET;
```

- `struct sock`, как правило, хранится в переменной `sk`

  struct socket по-видимому, это интерфейс более высокого уровня, который используется для системных вызовов (именно поэтому он также имеет указатель struct file, который представляет здесь файловый дескриптор).

  struct sock это имплементация в ядре для AF_INET сокетов (есть также struct unix_sock для AF_UNIX сокетов, которые являются производными от этого), которые

```
    serv_addr.sin_port = htons(PORT);
    serv_addr.sin_addr = *((struct in_addr*) host->h_addr_list[0]);

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {
      printf("connect() failed: %d", errno);
      return EXIT_FAILURE;
    }

    srand(time(NULL));

    for (int i = 0; i < COUNT; i++)
    {
        memset(message, 0, BUF_SIZE);
        sprintf(message, "message[%d]\n", i);

        if (send(sock, message, sizeof(message), 0) < 0)
        {
            perror("send() failed:");
            return EXIT_FAILURE;
        }

        recv(sock, buf, sizeof(message), 0);

        printf("Server got %s\n", buf);

        sleep(1 + rand() % 5);
    }

    close(sock);

    return EXIT_SUCCESS;
}
```

Код сервера:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define BUF_SIZE 256
#define PORT 3425
#define NUMBER_OF_CLIENTS 5

void receive(int *clients, int n, fd_set *set)
```

```
{
    char buf[BUF_SIZE];
    int bytes;

    for (int i = 0; i < n; i++)
    {
        if (FD_ISSET(clients[i], set))
        {
            // Поступили данные от клиента, читаем их
            bytes = recv(clients[i], buf, BUF_SIZE, 0);

            if (bytes <= 0)
            {
                // Соединение разорвано, удаляем сокет из множества
                printf("Client[%d] disconnected\n", i);
                close(clients[i]);
                clients[i] = 0;
            }
            else
            {
                // Отправляем данные обратно клиенту
                buf[bytes] = 0;
                printf("Client[%d] sent %s\n", i, buf);
                send(clients[i], buf, bytes, 0);
            }
        }
    }
}

int main(int argc, char ** argv)
{
    int sock;
    int new_sock;
    struct sockaddr_in serv_addr;
    fd_set set;
    int clients[NUMBER_OF_CLIENTS] = {0};
    int mx;
    int flag = 1;

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (socket < 0)
    {
      printf("socket() failed: %d\n", errno);
      return EXIT_FAILURE;
    }

    fcntl(sock, F_SETFL, O_NONBLOCK);

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(PORT);
```

```
if (bind(sock, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
{
  printf("bind() failed: %d\n", errno);
  return EXIT_FAILURE;
}

if (listen(sock, 6) < 0)
{
    printf("listen() failed: %d\n", errno);
    return EXIT_FAILURE;
}

printf("waiting...\n");

while(1)
{
    // Заполняем множество сокетов
    FD_ZERO(&set);
    FD_SET(sock, &set);
    mx = sock;

    for (int i = 0; i < NUMBER_OF_CLIENTS; i++)
    {
        if (clients[i])
        {
            FD_SET(clients[i], &set);
        }
        mx = (mx > clients[i]) ? mx : clients[i];
    }

    // Ждём события в одном из сокетов
    if (select(mx + 1, &set, NULL, NULL, NULL) <= 0)
    {
        perror("select");
        exit(1);
    }

    // Определяем тип события и выполняем соответствующие действия
    if (FD_ISSET(sock, &set))
    {
        // Поступил новый запрос на соединение, используем accept
        new_sock = accept(sock, NULL, NULL);

        if (new_sock < 0)
        {
            perror("accept");
            exit(1);
        }

        fcntl(new_sock, F_SETFL, O_NONBLOCK);

        flag = 1;
```

```
            for (int i = 0; i < NUMBER_OF_CLIENTS && flag; i++)
            {
                if (!clients[i])
                {
                    clients[i] = new_sock;
                    printf("Added as client №%d\n", i);
                    flag = 0;
                }
            }
        }

        receive(clients, NUMBER_OF_CLIENTS, &set);
    }
    return EXIT_SUCCESS;
}
```

Билет №4

1. Файловая подсистема /proc – назначение, особенности, файлы, поддиректории, ссылка self, информация об окружении, состоянии процесса, прерываниях. Структура proc_dir_entry: функции для работы с элементами /proc. Использование структуры file_operations для регистрации функций работы с файлами. Передача данных из адресного пространства пользователя в адресное пространство ядра и обратно (лаб. раб.). Обоснование необходимости использования специальных функций для передачи данных из пространства пользователя в ядро и из ядра в пространство пользователя.

Назначение и особенности

Файловая система /proc - виртуальная файловая система, предоставляющая интерфейс для доступа к структурам ядра. Папки (каталоги) и файлы виртуальной файловой системы /proc не хранятся на диске. Они создаются динамически при обращении к ним.

Большинство файлов в ней доступны только для чтения, однако некоторые из них доступны для записи, что позволяет изменять переменные ядра.

Файловая система /proc фактически представляет собой интерфейс ядра, который позволяет получать информацию о процессах и ресурсах, которые они используют.

Файлы и поддиректории

Данные о каждом процессе хранятся в поддиректории с именем, которым является идентификатор процесса: /proc/<PID>. В поддиректории процесса находятся файлы и поддиректории, содержащие данные о процессе .

Таблица – файлы и поддиректории /proc/<PID>

Элемент	Тип	Содержание
cmdline	файл	Указывает на директорию процесса
cwd	символическая ссылка	Указывает на директорию процесса
environ	файл	Список окружения процесса
exe	символическая ссылка	Указывает на образ процесса (на его файл)
fd	директория	Ссылки на файлы, которые «открыл» процесс
root	символическая ссылка	Указывает на корень файловой системы процесса
stat	файл	Информация о процессе

Ссылка self

Процесс может получить свой идентификатор с помощью функции getpid().

Другой способ – использовать ссылку self: /proc/self.

Информация об окружении:

/proc/[pid]/environ

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

Данный файл содержит исходное окружение, которое было установлено при запуске текущего процесса (вызове execve()). Переменные окружения разделены символами конца строки ('\0'). Если после вызова execve() окружение процесса будет модифицировано (например, вызовом функции putenv() или модификацией переменной окружения напрямую), этот файл не отразит внесенных изменений.

Окружение (environment) или среда — это набор пар ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ, доступный каждому пользовательскому процессу. Иными словами, окружение — это набор переменных окружения.

Информация о состоянии процесса:

Код вывода на экран содержимого файла stat.

```
  FILE *f = fopen("/proc/self/stat", "r");
  fread(buf, 1, BUF_SIZE, f);
  char *pch = strtok(buf, " ");

  printf("stat: \n");
  while(pch != NULL)
  {
        printf("%s\n", pch);
        pch = strtok(NULL, " ");
  }
fclose(f);
```

разделены пробелами.

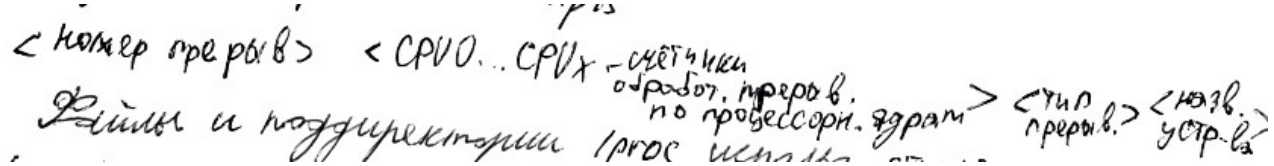
Содержимое файла /proc/[pid]/stat:

3. pid - уникальный идентификатор процесса.
4. comm - имя исполняемого файла в круглых скобках.
5. state - состояние процесса.
6. ppid - уникальный идентификатор процесса-предка.
7. pgrp - уникальный идентификатор группы.
8. session - уникальный идентификатор сессии.
9. tty_nr – управляющий терминал.
10. tpgid – уникальный идентификатор группы управляющего терминала.
11. flags – флаги.
12. minflt - Количество незначительных сбоев, которые возникли при выполнении процесса, и которые не требуют загрузки страницы памяти с диска.
13. cminflt - количество незначительных сбоев, которые возникли при ожидании окончания работы процессов-потомков.
14. majflt - количество значительных сбоев, которые возникли при работе процесса, и которые потребовали загрузки страницы памяти с диска.
15. cmajflt - количество значительных сбоев, которые возникли при ожидании окончания работы процессов-потомков.
16. utime - количество тиков, которые данный процесс провел в режиме пользователя.
17. stime - количество тиков, которые данный процесс провел в режиме ядра.
18. cutime - количество тиков, которые процесс, ожидающий завершения процессов-потомков, провёл в режиме пользователя.
19. cstime - количество тиков, которые процесс, ожидающий завершения процессов-потомков, провёл в режиме ядра.
20. priority – для процессов реального времени это отрицательный приоритет планирования минус один, то есть число в диапазоне от -2 до -100, соответствующее приоритетам в реальном времени от 1 до 99. Для остальных процессов это необработанное значение nice, представленное в ядре. Ядро хранит значения nice в виде чисел в диапазоне от 0 (высокий) до 39 (низкий), соответствующих видимому пользователю диапазону от -20 до 19.
21. nice - значение для nice в диапазоне от 19 (наиболее низкий приоритет) до -20 (наивысший приоритет).
22. num_threads – число потоков в данном процессе.
23. itrealvalue – количество мигов до того, как следующий SIGALARM будет послан процессу интервальным таймером. С ядра версии 2.6.17 больше не поддерживается и установлено в 0.
24. starttime - время в тиках запуска процесса после начальной загрузки системы.
25. vsize - размер виртуальной памяти в байтах.
26. rss - резидентный размер: количество страниц, которые занимает процесс в памяти. Это те страницы, которые заняты кодом, данными и пространством стека. Сюда не

включаются страницы, которые не были загружены по требованию или которые находятся в своппинге.

27. rsslim - текущий лимит в байтах на резидентный размер процесса.
28. startcode - адрес, выше которого может выполняться код программы.
29. endcode - адрес, ниже которого может выполняться код программ.
30. startstack - адрес начала стека.
31. kstkesp - текущее значение ESP (указателя стека).
32. kstkeip - текущее значение EIP (указатель команд).
33. signal - битовая карта ожидающих сигналов. Устарела, потому что не предоставляет информацию о сигналах реального времени, необходимо использовать /proc/[pid]/status.
34. blocked - битовая карта блокируемых сигналов. Устарела, потому что не предоставляет информацию о сигналах реального времени, необходимо использовать /proc/[pid]/status.
35. sigignore - битовая карта игнорируемых сигналов. Устарела, потому что не предоставляет информацию о сигналах реального времени, необходимо использовать /proc/[pid]/status.
36. sigcatch - битовая карта перехватываемых сигналов. Устарела, потому что не предоставляет информацию о сигналах реального времени, необходимо использовать /proc/[pid]/status.
37. wchan - "канал", в котором ожидает процесс.
38. nswap - количество страниц на своппинге (не обслуживается).
39. cnswap - суммарное nswap для процессов-потомков (не обслуживается).
40. exit_signal - сигнал, который будет послан предку, когда процесс завершится.
41. processor - номер процессора, на котором последний раз выполнялся процесс.
42. rt_priority - приоритет планирования реального времени, число в диапазоне от 1 до 99 для процессов реального времени, 0 для остальных.
43. policy - политика планирования.
44. delayacct_blkio_ticks - суммарные задержки ввода/вывода в тиках.
45. guest_time – гостевое время процесса (время, потраченное на выполнение виртуального процессора на гостевой операционной системе) в тиках.
46. cguest_time - гостевое время для потомков процесса в тиках.
47. start_data - адрес, выше которого размещаются инициализированные и неинициализированные (BSS) данные программы.
48. end_data - адрес, ниже которого размещаются инициализированные и неинициализированные (BSS) данные программы.
49. start_brk - адрес, выше которого куча программы может быть расширена с использованием brk().
50. arg_start - адрес, выше которого размещаются аргументы командной строки (argv).
51. arg_end - адрес, ниже которого размещаются аргументы командной строки (argv).
52. env_start - адрес, выше которого размещается окружение программы.
53. env_end - адрес, ниже которого размещается окружение программы.
54. exit_code – статус завершения потока в форме, возвращаемой waitpid().

Информация о прерываниях: cat /proc /interrupts



Структура proc_dir_entry:
функции для работы с элементами /proc

Файлы и поддиректории файловой системы /proc используют структуру proc_dir_entry:

```
struct proc_dir_entry {

    const char *name;           // имя виртуального файла

    mode_t mode;                // режим доступа

    uid_t uid;              // уникальный номер
пользователя -
                                                //
владельца файла

    uid_t uid;              // уникальный номер группы,
которой
                                                //
принадлежит файл

    struct inode_operations *proc_iops; // функции-
обработчики операций с inode

    struct file_operations *proc_fops; // функции-
обработчики операций с файлом

    struct proc_dir_entry  *next, *parent, *subdir;

    ...

    read_proc_t *read_proc;         // функция чтения
из /proc

    write_proc_t *write_proc;       // функция записи в /
proc

    void *data;             // Указатель на локальные
данные

    atomic_t count;             // счетчик ссылок на файл

    ...

};
```

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

Можно также создавать каталоги в файловой системе /proc, используя proc_mkdir(), а также символические ссылки с proc_symlink(). Для простых /proc-записей, для которых требуется только функция чтения, используется create_proc_read_entry() (proc_create()), которая создает запись /proc и инициализирует функцию read_proc в одном вызове.

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()

Фактически само ядро и каждый из процессов располагаются в своих собственных изолированных адресных пространствах.

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

Система предоставляет средства взаимодействия с приложениями:
- copy_to_user(Копирует данные из ядра в пространство пользователя. Вызывающий абонент должен проверить указанный блок с помощью access_ok до вызова этой функции.
Функция возвращает количество байт, которые не могут быть скопированы. В случае успешного выполнения будет возвращен 0.)
- copy_from_user (Копирует данные из пространства пользователя в пространство ядра. Вызывающий абонент должен проверить указанный заданный блок с помощью access_ok до вызова этой функции.
Функция возвращает количество байт, которые не могут быть скопированы. В случае успешного выполнения будет возвращен 0.)

Структуры file_operations для регистрации функций работы с файлами:

Структура file_operations используется для определения обратных вызовов (call back) чтения и записи.

Пример из ЛР:

```
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/vmalloc.h>
#include <linux/uaccess.h>

#define BUFSIZE 100
#define MAX_COOKIE_LENGTH PAGE_SIZE

MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR( "Neklepaeva A. N.");

static struct proc_dir_entry *proc_entry;
static struct proc_dir_entry *proc_link;
static struct proc_dir_entry *proc_dir;
static char *cookie_pot = NULL;
int cookie_index;
int next_fortune;

static ssize_t fortune_write(struct file *file, const char __user *ubuf, size_t count, loff_t *ppos)
{
    int free_space = (MAX_COOKIE_LENGTH - next_fortune) + 1;

    if (count > free_space)
    {
        printk(KERN_INFO "Cookie pot full.\n");
        return -ENOSPC;
    }

    if (copy_from_user(&cookie_pot[cookie_index], ubuf, count))
    {
        return -EFAULT;
    }

    cookie_index += count;
```

```
    cookie_pot[cookie_index - 1] = 0;

    return count;
}

static ssize_t fortune_read(struct file *file, char __user *ubuf,
size_t count, loff_t *ppos)
{
    int len;
    char buf[256];

    if (cookie_index == 0 || *ppos > 0)
    {
        return 0;
    }

    if (next_fortune >= cookie_index)
    {
        next_fortune = 0;
    }

    len = sprintf(buf, "%s\n", &cookie_pot[next_fortune]);
    copy_to_user(ubuf, buf, len);
    next_fortune += len;
    *ppos += len;

    return len;
}

static struct file_operations myops =
{
    .owner =  THIS_MODULE,
    .read = fortune_read,
    .write = fortune_write
};

static int __init fortune_init(void)
{
    cookie_pot = (char *)vmalloc(MAX_COOKIE_LENGTH);
    if (!cookie_pot)
    {
```

```
            printk(KERN_INFO "Not enough memory for the cookie
pot.\n");
            return -ENOMEM;
        }

        memset( cookie_pot, 0, MAX_COOKIE_LENGTH );

        proc_entry = proc_create("fortune", 0666, NULL, &myops);
        proc_dir = proc_mkdir("mydir", NULL);

        if (proc_entry == NULL)
        {
            vfree(cookie_pot);
            printk(KERN_INFO "fortune: Couldn't create proc entry\
n");
            return -ENOMEM;
        }

        proc_link = proc_symlink("link", NULL, "fortune");

        cookie_index = 0;
        next_fortune = 0;

        printk(KERN_INFO "fortune: Module inited.\n");

        return 0;
    }

    static void __exit fortune_exit(void)
    {
        remove_proc_entry("fortune", NULL);
        remove_proc_entry("mydir", NULL);
        remove_proc_entry("link", NULL);

        if (cookie_pot)
            vfree(cookie_pot);

        printk(KERN_INFO "fortune: Module exited.\n");
    }

    module_init(fortune_init);
```

```
module_exit(fortune_exit);
```

Обоснование необходимости использования специальных функций для передачи данных из пространства пользователя в ядро и из ядра в пространство пользователя:

Для передачи данных из ядра и в ядро из режима пользователя нужны специальные функции copy_from_user() и copy_to_user(), потому что в Linux память сегментирована, то есть указатель не ссылается на уникальную позицию в памяти, а ссылается на позицию в сегменте (адресация "сегмент-смещение"). Процессу доступен только собственный сегмент памяти. Он может обращаться только к собственным сегментам. Если выполняется обычная программа, то адресация происходит автоматически в соответствии с принятой в системе адресацией. Если выполняется код ядра и необходимо получить доступ к сегменту кода ядра, то всегда нужен буфер, но когда нужно передать информацию между текущим процессом и кодом ядра, то соответствующие функции ядра получают указатель на буфер в сегменте процесса. (тоже самое только другими словами:
1)у процесса свое адресное пространство защищенное
2)для процесса создается виртуальное адресное пространство
3)у него есть последовательность адресов
и отсчитывается смещение от базового адреса сегмента и получаем виртуальный адрес)

Билет №5

1. Управление устройствами: абстракция устройств, типы устройств и идентификация в ядре Unix/Linux. Управление устройствами: драйверы. USB-шина: особенности, хост и хабы, конечные точки и каналы, 4 типа передачи данных. Структура USB-драйвера (struct usb_driver), таблица id_table, основные точки входа драйвера USB. Регистрация usb-драйвера в системе.

Специальные файлы устройств обеспечивают единообразный доступ к внешним устройствам.

(см билет 1)

+

Старший номер	**Тип устройства**
1	Оперативная память
2	Дисковод гибких дисков
3	Первый контроллер для жестких IDE-дисков
4	Терминалы
5	Терминалы
6	Принтер (параллельный разъем)

8	Жесткие SCSI-диски
14	Звуковые карты
22	Второй контроллер для жестких IDE-дисков

Файлы устройств одного типа имеют одинаковые имена и различаются по номеру, прибавляемому к имени. Например, все файлы сетевых плат Ethernet имеют имена, начинающиеся на **eth:** eth0, eth1 и т. д.

где параметр – «тип_устройства» может принимать одно из четырех значений:

1. b — блок-ориентированное устройство;
2. c — байт-ориентированное (символьное) устройство;
3. u — не буферизованное байт-ориентированное устройство;
4. p — именованный канал.

Драйверы

Драйвер – программа или часть кода ядра, которая предназначена для управления конкретным внешним 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 инсталяции 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)

3 типа драйверов:

Драйверы первого типа являются частью программного кода ядра (встроены в ядро). Соответствующие устройства автоматически обнаруживаются системой и становятся доступны для приложений.

Драйверы второго типа представлены загружаемыми модулями ядра. Они оформлены в виде отдельных файлов и для их подключения (на этапе загрузки или впоследствии) необходимо выполнить отдельную команду подключения модуля.

Третий тип драйверов. В этих драйверах устройств программный код драйвера поделен между ядром и специальной утилитой, предназначенной для управления данным устройством.

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
...
}
```

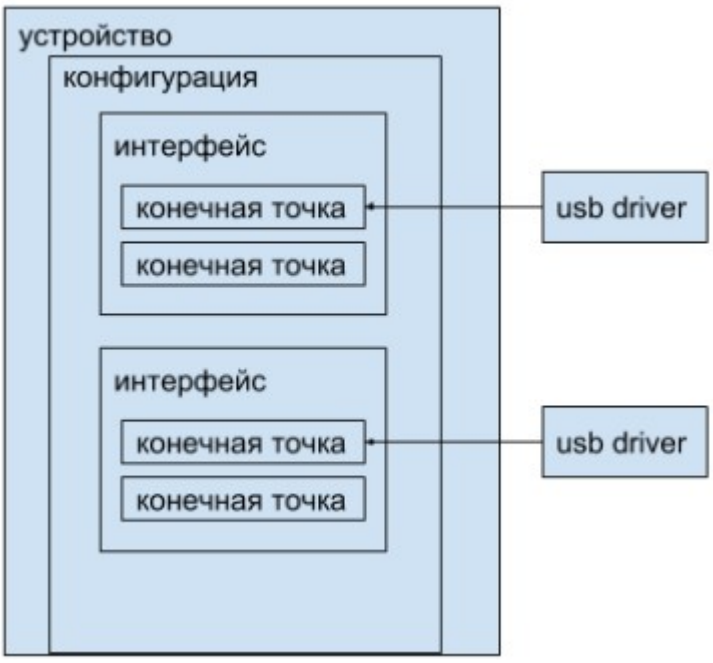
Структура struct device_driver

```
struct device_driver {
const char *name;
```

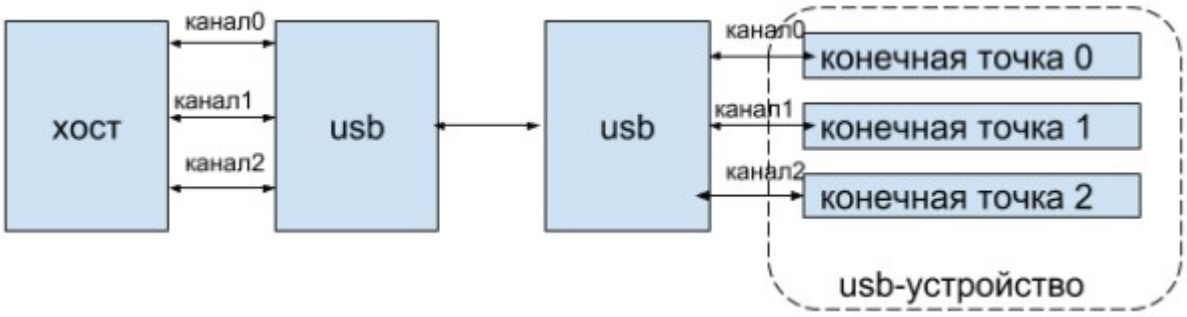
```
struct bus_type * bus ;
struct module * owner ;
const char mod_name;
//имя модуля (для встраиваемых модулей)
...
//точки входа в драйвер
//вызывается для запроса существования конкретного устройства, если
драйвер может с ним работать
int (*probe)(struct device *dev);
//удаляет драйвер
int (*remove)(struct 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-а.
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.

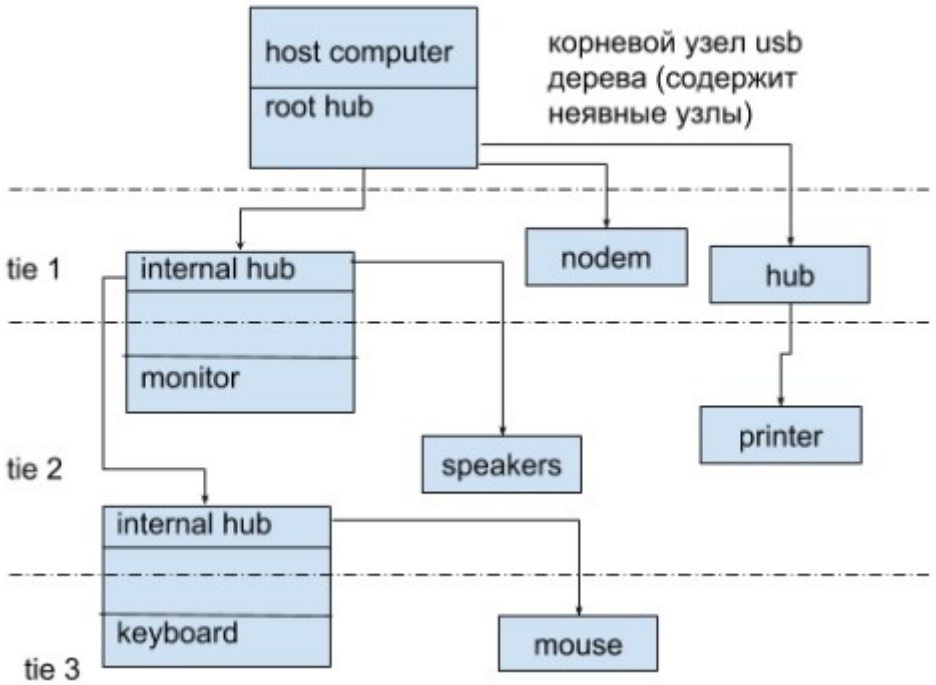


Совокупность конечной точки и структур данных ядра - канал (pipe).



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.
- Изохронные 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 ;

};
nsme - 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 которуб 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
...
}
```

Билет №6

1. Файловая система, задачи файловой системы и иерархическая организация ФС. Файловая подсистема LINUX: поддержка большого числа файловых систем и структура, описывающая файловую систему. VFS: четыре основные структуры файловой системы и связь между ними.
Раздел жесткого диска и суперблок. Структура struct super_operations. Монтирование файловой системы, команда mount и функции монтирования и их параметры, точка монтирования – корневой каталог и inode. Пример (лаб.раб.)

управление файлами осуществляется частью ОС - файловой системой (файловой подсистемой) File System
файл - каждая поименованная совокупность данных хранящаяся во вторичной памяти

файловая система - это часть ОС, которая отвечает за возможность хранения информации и доступа к ней

файловая система
а) определяет формат сохраненных данных и способ их физического хранения
б) связывает формат физического хранения и api для доступа к файлам
в) Должна обеспечивать создание, чтение, запись, удаление, переименование файлов

Задачи:
1.Именование файлов с точки зрения пользователя
2.Обеспечение программного интерфейса для работы с файлами пользователя и приложения
3.Отображение логической модели файлов на физ. реализацию хранения данных во вторич. памяти.
4.Обеспечение надежного хранения файлов, доступа к ним и защиты от несанкционированного доступа
5.Обеспечение совместного использования файлов.

Иерархическая организация ФС:

Символьный уровень - уровень именования файлов удобный для пользователей (уровень каталогов и имен файлов которыми оперирует приложение)
Базовый уровень - уровень идентификации файла файловая подсистема - программа, которая работает по определенным принципам => любая переменная должна быть идентифицирована
Логический уровень - логический уровень позволяет обеспечить доступ к данным в файле в формате отличном от формата их физического хранения
Физический уровень - обеспечение непосредственного доступа к информации на внешнем носителе
Физическое уст-во

Файловая подсистема LINUX: поддержка большого числа файловых систем:

Чтобы иметь возможность поддерживать большое кол-во файловых систем поддерживается спец. интерфейс
в unix VFS/vnode
в linux VFS

Виртуальная файловая система (Virtual File System, известная также как Virtual Fileystem Switch) - это подсистема в ядре Linux, который обеспечивает интерфейс файловой системы для программ пользовательского пространства. Он также предоставляет абстракцию в ядре, которая позволяет сосуществовать различным реализациям файловой системы.

VFS определяет интерфейс, который должны поддерживать конкретные файловые системы, чтобы работать в Linux.

struct vfs

```
{
    struct vfs *vfs_next;
    struct vfsops *vfs_op;
    ...
}
```

VFS определяет четыре базовые абстракции:

1. struct superblock

определяет конкретную файловую систему

суперблок - это структура, которая содержит информацию необходимую для монтирования и управления файловой системой

каждая файловая система имеет один суперблок

2. struct inode

структура описывает конкретный файл (2 типа inode: * дисковый inode (содержит адреса всех блоков диска, в которых наход данные конкрет файла) * inode памяти)

3. 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

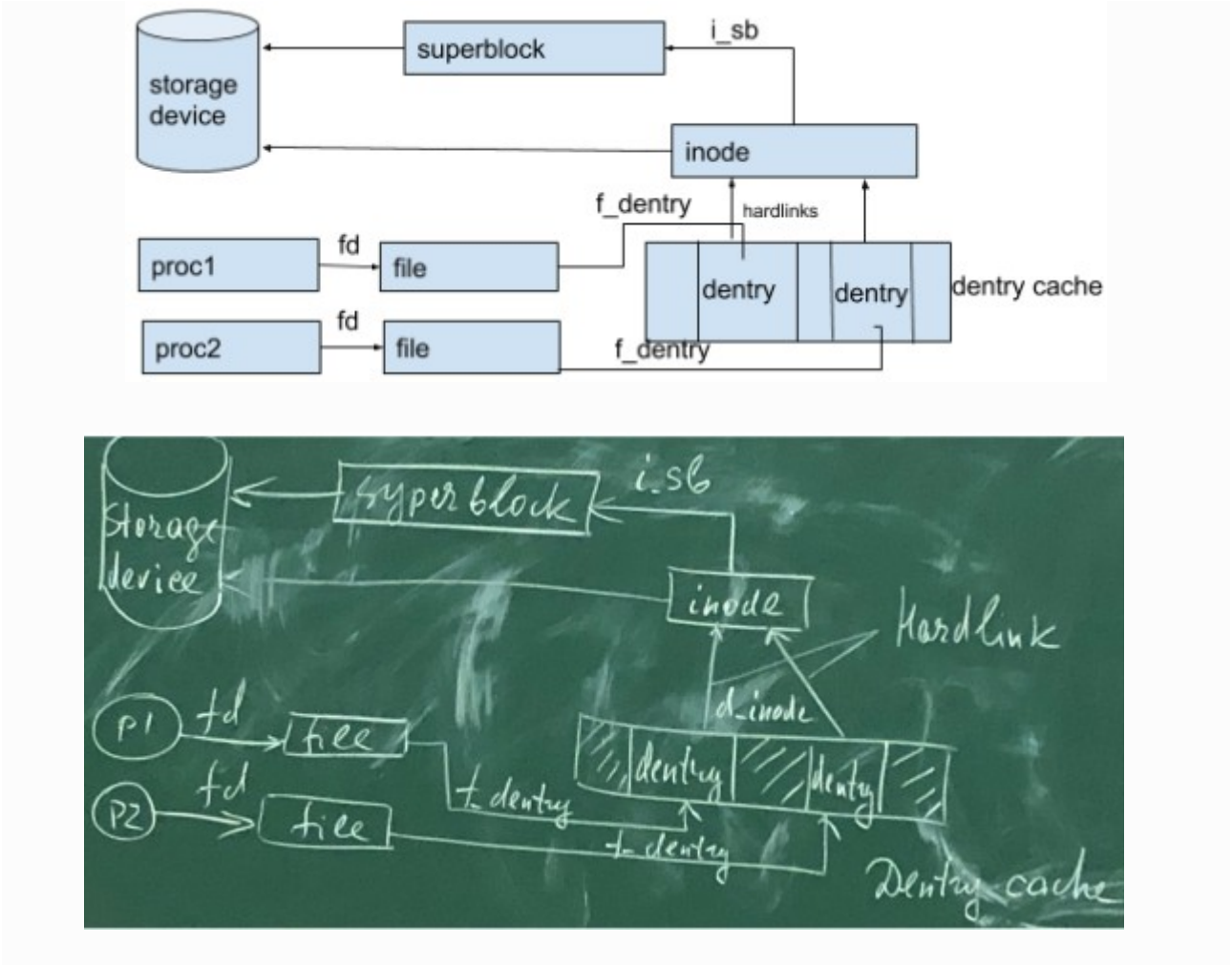
4. struct file

- одна из четырех структур, определяемых как основные для работы с файлами.

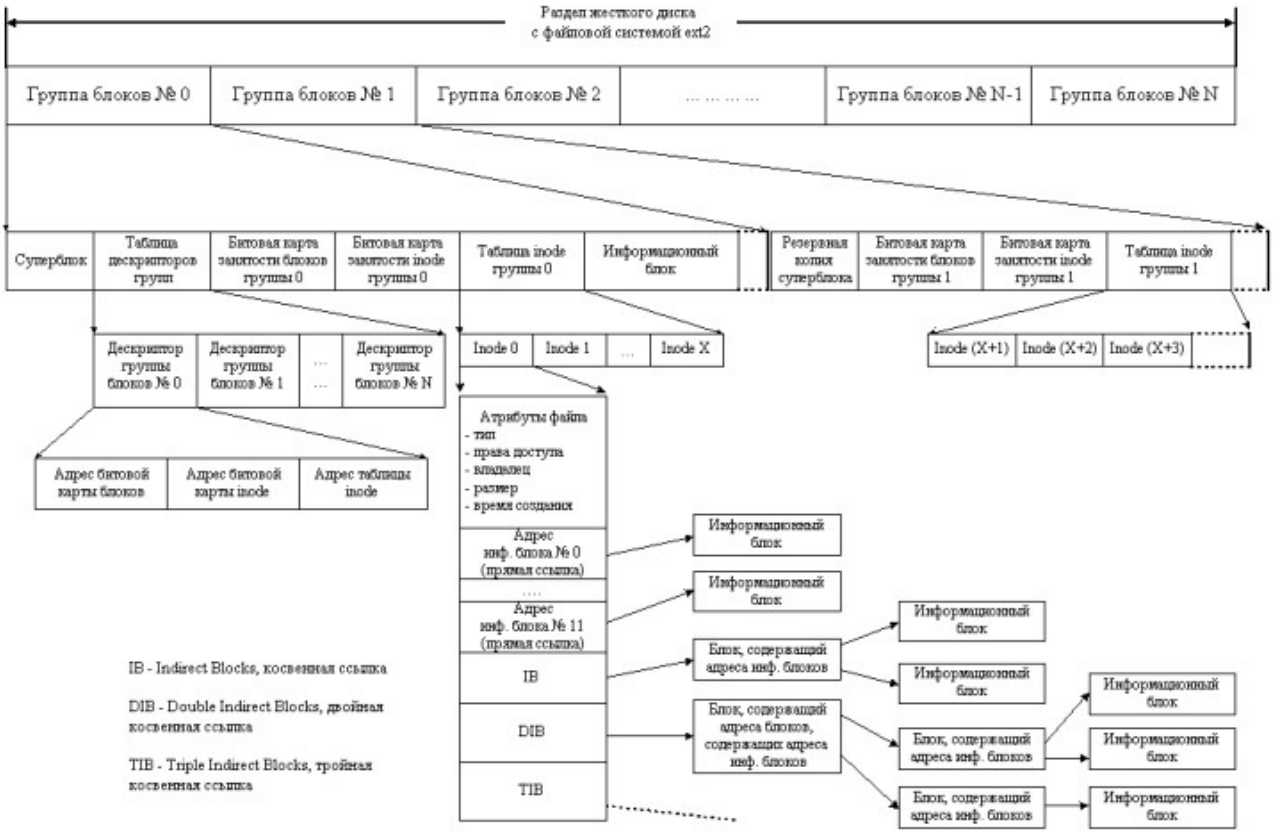
FILE - структура, представляющая открытые файлы.

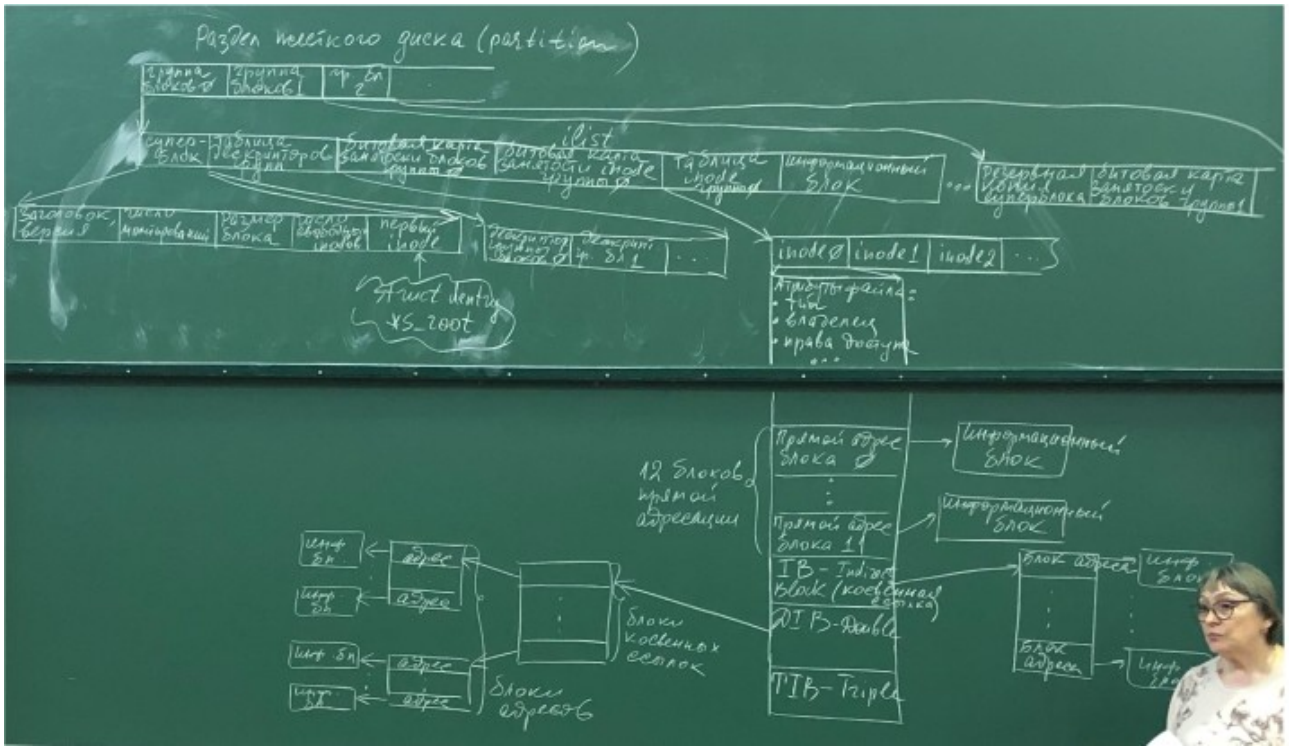
(выписать данные структуры)

Связь:



Раздел жесткого диска и суперблок:





Суперблок считывается в память ядра при монтировании Ф.С. и остается там до отмонтирования. Поэтому есть флаги dirty и пр.
суперблок - это структура, которая содержит информацию необходимую для монтирования и управления файловой системой
суперблок - это структура, которая находится на диске
каждая файловая система имеет один суперблок
но на диске суперблок находится в нескольких экземплярах (для надежности)
такая структура должна определять параметры для управления файловой системы (суммарное число блоков, корневой inode, ...)
суперблок существует не только на диске
суперблок на диске предоставляет ядру системы информацию о структуре файловой системы (но это не точно)
суперблок в памяти предоставляет информацию необходимую для управления смонтированной файловой системой
<linux/fs.h>
struct super_block
{
struct list_head s_list; // список суперблоков
k_dev_t s_dev; // указывает устройство на котором находится файловая система
unsigned long long s_blocksize; // размер блока в байтах
unsigned char s_dirt; // флаг показывающий, что суперблок был изменен
unsigned long s_max_byte; //
struct file_system_type *s_type; // тип файловой системы
struct super_operations *s_op; // перечислены действия определенные на суперблоках
...
unsigned long s_magic; // магическое число файловой системы
struct dentry *s_root; // точка монтирования файловой системы // каталог монтирования файловой системы
...
int s_count; // счетчик ссылок на суперблок
...

struct list_head s_inodes; // все inode
const struct dentry_operations *s_d_op; // определяет dentry_operations для директорий
struct list_head s_dirty; // список измененных индексов
...
struct block_device *s_bdev; // драйвер
...
char s_id[32]; // строка имени
...
}
тк линукс поддерживает большое количество одновременно смонтированных файловых подсистем
то будет большое количество блоков
и они хранятся в struct list_heard - список суперблоков
суперблок описывает смонтированные файловые системы
но каждая файловая система описывается структурой struct file_system_type

файловая система занимает раздел жесткого диска
поделена на группы блоков
такая файловая система описывается структурой суперблок
очевидно
суперблок описывает файловую систему
задача файловой системы хранить информацию - файлы
чтобы иметь доступ к ним
они должны быть нумерованные
нумеруются inodами
описывается в struct inode
эффективнее всего хранить это в виде битовой карты
в соответствии со структурой struct superblock
там видим dentry *s_root (указатель на корневой каталог)
в системе все является файлом
все файлы в системе имеют inode
entry - directory entry
struct dentry создается налету
она нигде не хранится
и она создается на основе информации из inode

дисковый inode хранит информацию о физическом файле и позволяет получить доступ к информации записанной в файле

ОС linux поддерживает файлы очень больших размеров
чтобы получить доступ к блоку необходимо хранить его адрес
существует 12 блоков прямой адресации
блок indirect - этот блок ссылается на блок, в котором хранятся адреса блоков по тому же принципу

видим DIB - двойная косвенная адресация
TIB - тройная косвенная адресация

Структура struct super_operations.

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 дейтсвие с супеблоком (низкоуровневые действия с Ф.С.)
Когда система нуждается в выполнении операции над суперблоком:
sb -> s_op -> write_super(sb);

Монтирование файловой системы, команда mount и функции монтирования и их параметры, точка монтирования – корневой каталог и inode. Пример (лаб.раб.)

Жизнь любой файловой системы начинается с регистрации. Зарегистрировать файловую систему можно с помощью системного вызова register_filesystem(). Регистрация файловой системы выполняется в функции инициализации модуля. Для дерегистрации файловой системы используется функция unregister_filesystem(), которая вызывается в функции выхода загружаемого модуля.

Обе функции принимают как параметр указатель на структуру file_system_type, которая "описывает" создаваемую файловую систему.

```
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,
};

  static struct file_system_type myfs_type  =  {
          .owner  =  THIS_MODULE,
          .name  =  "myfs",
          .mount  =  myfs_mount,
          .kill_sb  =  kill_block_super,
  };
```

owner отвечает за счетчик ссылок на модуль
name хранит название файловой системы

mount и kill_sb два поля хранящие указатели на функции. Первая функция будет вызвана при монтировании файловой системы, а вторая при размонтировании.

Теперь рассмотрим функцию myfs_mount. Она должна примонтировать устройство и вернуть структуру, описывающую корневой каталог файловой системы:

```
  static struct dentry* myfs _ mount (struct file_system_type *type, int flags, char const *dev,
       void *data)
  {
          struct dentry* const entry = mount_bdev(type,  flags,  dev,  data, myfs_fill_sb) ;
          if (IS_ERR(entry))
                  printk(KERN_ERR  "MYFS mounting failed !\n") ;
          else
                 printk(KERN_DEBUG "MYFS mounted!\n") ;
          return entry;
  }
```

большая часть работы происходит внутри функции mount_bdev(), но нас интересует лишь ее параметр myfs_fill_sb. Это указатель на функцию, которая будет вызвана из mount_bdev, чтобы проинициализировать суперблок. Сама функция myfs_mount должна вернуть структуру dentry (переменная entry), представляющую корневой каталог нашей файловой системы, а создаст его функция myfs_fill_sb.

В первую очередь заполняется структура super_block: магическое число, операции для суперблока, его размер.

Проинициализировав суперблок, функция myfs_fill_sb() выполняет построение корневого каталога нашей ФС. Первым делом для него создается inode вызовом myfs_make_inode. Он нуждается в указателе на суперблок и аргументе mode. Далее для корневого каталога создается структура dentry, через которую он помещается в directory-кэш.

Пример из ЛР:

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

```
MODULE_LICENSE( "GPL" );
MODULE_AUTHOR( "Anastasia Neklepaeva" );

#define MAGIC_NUMBER 0x13131313

#define SLABNAME "my_cache"
struct kmem_cache *cache = NULL;
static int number = 7;
module_param(number, int, 0);

struct myfs_inode
{
    int i_mode;
    unsigned long i_ino;
};

static int size = sizeof(struct myfs_inode);

// деструктор суперблока
static void myfs_put_super(struct super_block * sb)
{
    printk(KERN_DEBUG "MYFS super block destroyed!\n" ) ;
}

// put_super хранит деструктор нашего суперблока
static struct super_operations const myfs_super_ops =
{
    .put_super = myfs_put_super,
    .statfs = simple_statfs,
    .drop_inode = generic_delete_inode,

};

// размещение новой структуры inode
// и заполнение ее значениями: размером и временами
static struct inode *myfs_make_inode(struct super_block * sb, int mode)
{
    struct inode *ret = new_inode(sb);
    struct myfs_inode *my_inode;

    if (ret)
    {
        inode_init_owner(ret, NULL, mode);
        ret->i_size = PAGE_SIZE;
        ret->i_atime = ret->i_mtime = ret->i_ctime = current_time(ret);

        my_inode = kmem_cache_alloc(cache, GFP_KERNEL);
```

```
        my_inode->i_mode = ret->i_mode;
        my_inode->i_ino = ret->i_ino;

        ret->i_private = my_inode;
    }
    return ret;
}

// инициализация суперблока, построение корневого каталога
static int myfs_fill_sb(struct super_block* sb, void* data, int silent)
{
    struct inode* root = NULL;
    sb->s_blocksize = PAGE_SIZE;
    sb->s_blocksize_bits = PAGE_SHIFT;
    sb->s_magic = MAGIC_NUMBER;
    sb->s_op = &myfs_super_ops;

    // создание inode для корневого каталога
    root =  myfs_make_inode(sb, S_IFDIR | 0755);
    if (!root)
    {
        printk (KERN_ERR "MYFS inode allocation failed !\n") ;
        return -ENOMEM;
    }

    root->i_op  =  &simple_dir_inode_operations;
    root->i_fop = &simple_dir_operations;
    sb->s_root  = d_make_root(root) ;

    if (!sb->s_root)
    {
        printk(KERN_ERR " MYFS root creation failed !\n") ;
        iput(root);
        return -ENOMEM;
    }

    return 0;
}

// функция вернёт структуру, описывающую корневой каталог файловой системы
static struct dentry* myfs_mount (struct file_system_type *type, int flags,
                                  char const *dev, void *data)
{
    struct dentry* const entry = mount_nodev(type,  flags,  data, myfs_fill_sb) ;
    if (IS_ERR(entry))
        printk(KERN_ERR  "MYFS mounting failed !\n") ;
    else
        printk(KERN_DEBUG "MYFS mounted!\n") ;
    return entry;
```

```
}

// структура, "описывающая" создаваемую файловую систему
// owner отвечает за счетчик ссылок на модуль
// name хранит название файловой системы
static struct file_system_type myfs_type  =
{
    .owner  =  THIS_MODULE,
    .name  =  "myfs",
    .mount  =  myfs_mount,
    .kill_sb  =  kill_anon_super,
};

static int __init myfs_init(void)
{
    // регистрация файловой системы
    int ret = register_filesystem(& myfs_type);
    if(ret != 0)
    {
        printk(KERN_ERR "MYFS_MODULE cannot register filesystem!\n");
        return ret;

    }

    cache = kmem_cache_create(SLABNAME, size, 0, SLAB_HWCACHE_ALIGN, NULL);

    if (!cache)
    {
        printk( KERN_ERR "kmem_cache_create error\n");
        kmem_cache_destroy(cache);
         return -ENOMEM;
    }

    printk(KERN_DEBUG "MYFS_MODULE loaded !\n");
    return 0;
}


static void __exit myfs_exit(void)
{
    int ret;

    // дерегистрация файловой системы
    ret = unregister_filesystem(&myfs_type);
    if (ret != 0)
        printk(KERN_ERR "MYFS_MODULE cannot unregister filesystem !\n");
    printk(KERN_DEBUG "MYFS_MODULE unloaded !\n");
```

```
    kmem_cache_destroy(cache);
}

module_init( myfs_init );
module_exit( myfs_exit );
```

Билет №7

1. Классификация типов ввода-вывода с точки зрения программиста: диаграммы последовательности действий для каждого типа ввода-вывода и описание. Классификация моделей ввода-вывода. Особенности и назначение асинхронного ввода-вывода.
Мультиплексирование. Пример мультиплексирования для сокетов AF_INET, SOCK_STREAM. Сетевой стек. Пример (лаб.раб.)

(см. Билет №3)

сокет семейства AF INET(что указы-
вает, что сокет должен быть сетевым)
типа SOCK STREAM(потоковый сокет)

Билет №8

1. Средства взаимодействия процессов – сокеты Беркли. Создание сокета – семейство, тип, протокол. Системный вызов sys_socket() и struct socket. Состояния сокета. Адресация сокетов и ее особенности для разных типов сокетов. Модель клиент-сервер.
Сетевые сокеты – сетевой стек, аппаратный и сетевой порядок байтов. Примеры реализации взаимодействия процессов по модели клиент-сервер с использованием сокетов и мультиплексированием (лаб. Раб.).

Сокет – это абстракция конечной точки взаимодействия. Сокеты являются универсальным средством межпроцессного взаимодействия в том смысле, что они могут использоваться как для взаимодействия процессов на отдельно стоящей машине, так и для взаимодействия процессов в сети.

Дизайн сокетов Беркли следует парадигме UNIX: в идеале **отобразить все объекты, к которым осуществляется доступ для чтения или записи, на файлы,** чтобы с ними можно было работать с использованием обычных функций записи и чтения в/из файла.

Протоколы для взаимодействия с использованием сокетов выбираются на основе трех параметров:

- семейство или домен (family);
- тип сокета (type);
- протокол (protocol)
  системного вызова socket():

```
#include <sys/types.h> /* See NOTES */
 #include <sys/socket.h>
```

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

socket () создает конечную точку для связи и возвращает файловый дескриптор, который относится к этой конечной точке. Дескриптор файла при успешном вызове возвращается файловый дескриптор с наименьшим номером дескриптора файла, который в данный момент еще не открыт для процесса.

Домен определяет семейство протоколов, которое будет использоваться для связи.
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

Параметр тип (type) определяет семантику соединения:
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.

Параметр протокол определяет конкретный протокол, который будет использоваться с сокетом.
protocol - протокол для определенного вида сокетов.
● для SOCK_STREAM - TCP
● для SOCK_DGRAM - UDP
● 0 - протокол будет выбран по умолчанию

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;
}
```

Как видно из описания функции, она инициализирует структуру socket. Значение retval и есть тот самый дескриптор, который возвращает функция socket().

Структура сокета:

```
struct socket {
        socket_state                state;

        kmemcheck_bitfield_begin(type);
        short                       type;
```

```
        kmemcheck_bitfield_end(type);

        unsigned long           flags;

        struct socket_wq __rcu      *wq;

        struct file             *file;//указатель на дескриптор открытого файла
        struct sock                 *sk;[2]
        const struct proto_ops      *ops;
};
```

Поле структуры state определяет одно из пяти состояний сокета:

- SS_FREE – не занят
- SS_UNCONNECTED – не соединен
- SS_CONNECTING – соединяется в данный момент
- SS_CONNECTED – соединен
- SS_DISCONNECTING – разъединяется в данный момент

Поле flags используется для синхронизации доступа.

Адресация

Сокеты поддерживают множество протоколов, поэтому была определена общая структура адреса sockaddr, так как при создании коммуникационных отношений нужно указывать адрес конечной точки коммуникационного партнера.

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

Когда сокет создается с помощью socket (2), он существует в пространстве имен (семейство адресов), но ему не назначен адрес. Специальная функция bind () назначает адрес, указанный addr, сокету, указанному дескриптором файла sockfd. Параметр addrlen определяет размер в байтах структуры адреса, на которую указывает addr. Традиционно эта операция называется «присвоение имени сокету».

```
#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>

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

Как видно, структура объявлена в самом общем виде. Поле sa_family определяет семейство адресов. Но точный формат адреса не определен.

Поэтому для адресов интернета используется другая структура sockaddr_in. Структура sockaddr_in описывает сокет для работы с протоколами IP.

---

2 Сетевой стек ядра Linux имеет две структуры:

- `struct socket`, как правило, хранится в переменной `sock`
- `struct sock`, как правило, хранится в переменной `sk`

struct socket по-видимому, это интерфейс более высокого уровня, который используется для системных вызовов (именно поэтому он также имеет указатель struct file, который представляет здесь файловый дескриптор).

struct sock это имплементация в ядре для AF_INET сокетов (есть также struct unix_sock для AF_UNIX сокетов, которые являются производными от этого), которые могут использоваться как ядром, так и пространством пользователей (via struct sock).

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

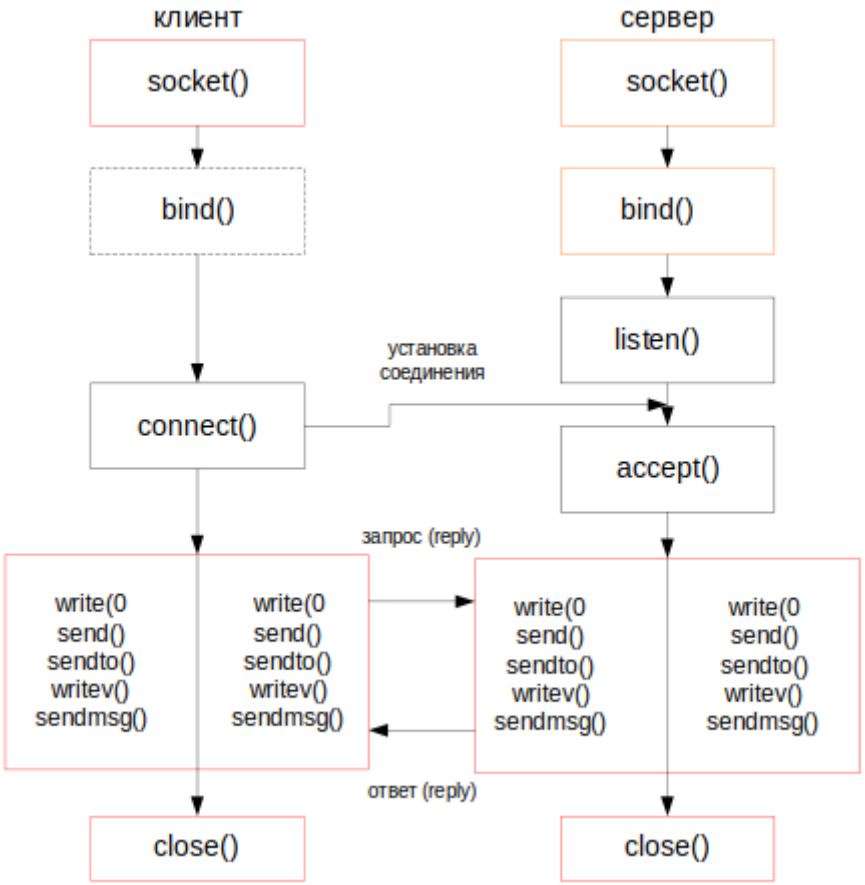
```
struct sockaddr_in {
sa_family_t sin_family;
//семейство адресов
unsigned short int sin_post; //номер порта
struct in_addr sin_addr ;
//IP-адрес
unsigned char sin_zero[ 8 ];
}

struct sockaddr_un {
sa_family_t sun_family;
char
sun_path[ 108 ];
}
```

Модель клиент-сервер

В модели клиент-сервер роли определены: сервер предоставляет ресурсы и службы одному или нескольким клиентам, которые обращаются к серверу за обслуживанием.

Когда клиент запрашивает соединение с сервером, сервер может либо принять, либо отклонить это соединение. Если соединение принято, сервер устанавливает и поддерживает соединение с клиентом по определенному протоколу . Клиент не предоставляет общий доступ ни к одному из своих ресурсов, но запрашивает данные или службу у сервера.



Сетевые сокеты:

Сетевые сокеты являются универсальным типом сокетов. Система приложений, предназначенных для работы на одном компьютере, может использовать сетевые сокеты для обмена данными. Однако даже если сокеты используются для обмена данными на одной и той же машине, передаваемые данные должны пройти все уровни сетевого стека, что отрицательно сказывается на быстродействии и нагрузке на систему.

Аппаратный и сетевой порядок байтов.

При присвоении значений номеру порта и адресу следует учитывать, что порядок следования байтов на разных архитектурах различен.

Существуют два порядка следования байтов:

При передаче данных по сети общепринятым является представление чисел в формате big-endian

BIG ENDIAN - обратный порядок (n -> n+1 -> n+2) (самый старший байт целого числа имеет наименьший адрес, а самом младшем байте находится наибольший значащий байт адреса.) - network byte order (TCP/IP)

LITTLE ENDIAN - прямой порядок (n+2 <- n+1 <- n) (наименьший адрес имеет самый младший байт, а наибольший адрес имеет самый старший байт.*)*- аппаратный

Для преобразования числа из той схемы, которая используется на компьютере к той, которая используется в сети, и наоборот, применяются функции:

```
uint32_t htonl(uint32_t hostlong);//host to network long
uint16_t htons(uint16_t hostshort);// host to network short
uint32_t ntohl(uint32_t netlong);//network to host long
```

uint16_t ntohs(uint16_t netshort); //network to host short

* код из лабы *

Билет №9

1. Файловая система: процесс и файловые структуры связанные с процессом. Файлы и открытые файлы, связь структур, представляющих открытые файлы на разных уровнях. Системный вызов open() и библиотечная функция fopen(): параметры и флаги, определенные на функции open(). Реализация системного вызова open() в ядре Linux. Пример: файл открывается два раза системным вызовом open() для записи и в него последовательно записывается строка «ааааааааааа» по первому дескриптору и затем строка «вввв» по второму дескриптору, затем файл закрывается два раза. Показать, что будет записано в файл и пояснить результат

Процесс - программа в стадии выполнения. Является единицей декомпозиции системы. Является потребителем системных ресурсов.

файловая система - это часть ОС, которая отвечает за возможность хранения информации и доступа к ней

файл - каждая поименованная совокупность данных хранящаяся во вторичной памяти

Структуры данных, связанные с процессом

С каждым процессом в системе связаны список открытых им файлов, корневая файловая система, текущий рабочий каталог, точки монтирования и т.д.

1) task_struct описывает запущенный в системе процесс, создается динамически.

struct task_struct{

```
…..
int                             prio;
int                             static_prio;
…
struct list_head                tasks;
…
struct mm_struct                *mm;
struct mm_struct                *active_mm;
…
pid_t                           pid;
pid_t                           tgid;
…
struct list_head                children;
struct list_head                sibling;
…
/*  Filesystem information   */
struct fs_struct                *fs;
/*  Open file information   */
struct files_struct             *files;
/*   Namespaces   */
struct nsproxy                  *nsproxy;
```

2) Структура struct fs_struct содержит информацию о файловой системе, к которой принадлежит процесс. Структура определена в файле <linux/fs_srtuct.h>.

```
#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 * pwdmnt ; //объект монтирования текущей директории
struct vfsmount * altrootmnt ; //объект монтир. альтерн. корневой дир-рии
}
```

3) Структура struct files_struct определяет дескрипторы файлов, открытых процессом.
Формирует таблицу открытых файлов процесса. Каждый процесс имеет собственную таблицу открытых файлов.

```
#include <linux/file.h>
struct files_struct {
atomic_t count;
//количество пользователей
spinlock_t file_lock;
```

```
//средство взаимоисключения
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
}
```

4) Структура struct file определяет дескриптор открытого файла в системе. Любой открытый файл будет иметь в ядре объект файл.

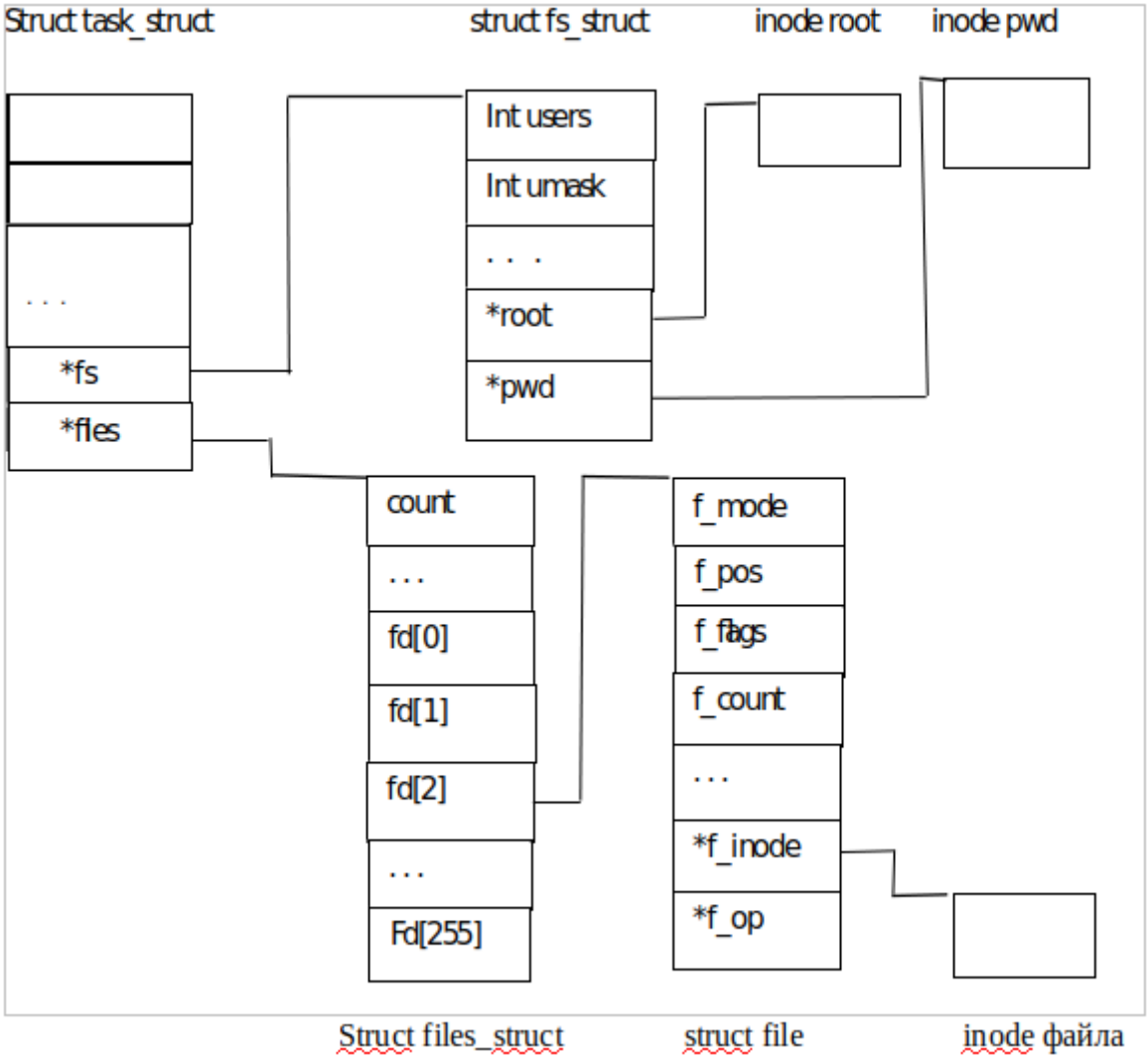
```
struct file
{
   …
   struct path f_path;
   struct inode *f_inode;
   const struct file_operations *f_op; // указат на табл файл операций
   spinlock_t f_lock;
   …
   atomic_long_t f_count;
   unsigned int f_flags;
  fmode_t f_mode; // режим доступа к файлу
  long offset type;
  struct mutex f_pos_lock;
  loff_t f_pos; // смещение в файле
  struct fown_struct f_owner;
}
```

5) Структура struct inode описывает созданный файл.

```
struct inode
{
umode_t i_mode;
unsigned short i_opflags;
kuid_t i_uid;
kgid_t i_gid;
unsigned int i_flags;
...
const struct inode_operations *i_op; // указатель на структуру inode_operations
struct super_block *i_sb;
struct address_space *i_mapping;
...
// stat data, not accessed from path walking
unsigned long i_ino;
```

```
...
loff_t i_size;
struct timespec i_atime; // обращение
struct time_spec i_mtime; // изменение
struct timespec i_ctime; // изменение параметров управления
...
struct heist_node i_hash;
...
const struct file_operations *i_op;
}
```

Обычные файлы расположены во вторичной памяти, на устройстве долговременного хранения, такие файлы называются регулярными. Две большие разницы — открытые файл и просто файлы. Для того, чтобы работать с файлом, файл должен быть открыт.



Системный вызов open() и библиотечная функция fopen():
параметры и флаги, определенные на функции open().

Чтобы получить возможность прочитать что-то из файла или записать что-то в файл, его нужно открыть. Это делает системный вызов open().
open – открывает и возможно создает файл.

SYNOPSIS top

```
#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);
```

Системный вызов open() открывает файл, указанный в pathname. Если указанный файл не существует, он может (необязательно) (если указан флаг O_CREAT) быть создан open().

Возвращаемое значение open() - это дескриптор файла.

Вызов open () создает новый файловый дескриптор для открытого файла, запись в общесистемной таблице открытых файлов. Эта запись регистрирует **смещение в файле** и флаги состояния файла (модифицируемые с помощью the fcntl (2) операции F_SETFL). Дескриптор файла является ссылкой на одну из этих записей; эта ссылка не влияет, если путь впоследствии удален или изменен ссылаться на другой файл. Новый дескриптор открытого файла изначально не разделяется с любым другим процессом, но разделение может возникнуть через ***fork***((2).

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

Параметр *flags* - это флаги **O_RDONLY**, **O_WRONLY** или **O_RDWR**, открывающие файлы "только для чтения", "только для записи" и для чтения и записи соответственно.

Аргумент *mode* задает права доступа, которые используются в случае создания нового файла.

mode всегда должен быть указан при использовании **O_CREAT**; во всех остальных случаях этот параметр игнорируется. **creat** эквивалентен **open** с *flags*, которые равны **O_CREAT** | **O_WRONLY** | **O_TRUNC**.

Для режима (mode) предусмотрены следующие символические константы:

- Пользователь (владелец файла) имеет права доступа:

S_IRWXU 00700 – на чтение, на запись, на исполнение;
S_IRUSR 00400 - на чтение;
S_IWUSR 00200 - на запись;
S_IXUSR 00100 - на выполнение;

- Группа пользователя имеет права доступа:

S_IRWXG 00070 - на чтение, запись и выполнение;
S_IRGRP 00040 - на чтение;
S_IWGRP 00020 - на запись;
S_IXGRP 00010 - на выполнение;

- Остальные имеют права доступа:

S_IRWXO 00007 - на чтение, запись и выполнение;
S_IROTH 00004 - на чтение;
S_IWOTH 00002 - на запись;
S_IXOTH 00001 - на выполнение,

Если установить флаги следующим образом:
S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH или **0664** это означает для пользователя - чтение/запись, для группы - чтение/запись и для остального мира -чтение.

…

O_PATH (since Linux 2.6.39)
Получите файловый дескриптор, который можно использовать для двух целей: указать местоположение в дереве каталогов файловой системы и выполнить операции, которые действуют исключительно на уровне файлового дескриптора. Сам файл не открывается, и другие файловые операции (например, read (2), write (2), fchmod (2), fchown (2), fgetxattr (2), ioctl (2), mmap (2)) завершаются неудачно с ошибкой EBADF.

…

O_TMPFILE (since Linux 3.11)
Создать неназванный временный обычный файл. Аргумент pathname указывает каталог; безымянный индекс будет создан в файловой системе этого каталога. Все, что записано в результирующий файл, будет потеряно при закрытии последнего дескриптора файла, если файлу не присвоено имя.

Флаг O_TMPFILE должен быть указан с одним из O_RDWR или O_WRONLY и, необязательно, O_EXCL. Если O_EXCL не указан, то linkat (2) может использоваться для связывания временного файла с файловой системой.

и т.д.

Некоторые флаги создания файлов и флаги состояния файлов:
O_CREAT - если файл не существует, то он будет создан
O_APPEND - Файл открывается в режиме добавления

* Флаги создания файла: O_CLOEXEC, O_CREAT, O_DIRECTORY, O_EXCL, O_NOCTTY, O_NOFOLLOW, O_TMPFILE и O_TRUNC. Флаги состояния файла это – достаточно большой набор флагов, которые учитываются при открытии или создании файлов в функциях ядра, предназначенных для этой работы. Различие между этими двумя группами флагов состоит в том, что флаги создания файла влияют на семантику самой операции открытия, в то время как флаги состояния файла влияют на семантику последующих операций ввода-вывода. Флаги состояния файла могут быть извлечены и (в некоторых случаях) изменены системным вызовом fcntl(2). *

#include <stdio.h>

FILE *fopen(const char **filename*, const char **mode*);

Функция fopen() открывает файл, имя которого указано аргументом fname, и возвращает связанный с ним указатель. Тип операций, разрешенных над файлом, определяется аргументом mode. fopen() выполняет ввод-вывод с буферизацией.

Реализация системного вызова open() в ядре Linux.

Системный вызов **open()** является оберткой функции ядра **ksys_open()**, которая в свою очередь вызывает функцию **do_sys_open().** В теле функции do_sys_open() вызывается

функция build_open_how() и функция do_sys_openat2(). Функция do_sys_openat2 вызывает функции: build_open_flags(), getname(filename), get_unused_fd_flags() и, если функция get_unused_fd_flags() возвращает fd > 0 , то вызывается функция do_filp_open(). Функция **do_sys_open**at2() выполняет все задачи системного вызова open. Сначала выполняется проверка правильности флагов и их преобразование во внутреннее представление функцией **build_open_flags**(). В случае неуспешного преобразования и проверки будет возвращена ошибка преобразования и проверки. Затем вызывается функция **getname** (const char__**user** ***filename**), которая выполняет копирование имени файла из пространства пользователя в пространство ядра. Вызываемая функция get_unused_fd_flags() является оберткой функции **__alloc_fd().** Функция __alloc_fd() находит для процесса свободный файловый дескриптор открытого файла и помечает его как занятый. Функция do_filp_open() возвращает указатель на struct **file**, т.е. создается дескриптор открытого файла в системной таблице открытых файлов. Функция set_nameidata() инициализирует структуру nameidata. В последней строке структура записывается в контекст текущего процесса. Затем вызывается функция path_openat(), которая возвращает указатель на struct file, т.е. на дескриптор открытого файла в системной таблице открытых файлов. Функция path_openat выполняет поиск пути.

Пример: файл открывается два раза системным вызовом open() для записи и в него последовательно записывается строка «ааааааааааа» по первому дескриптору и затем строка «вввв» по второму дескриптору, затем файл закрывается два раза. Показать, что будет записано в файл и пояснить результат

```
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main()
{
   int fd1 = open("exam_9.txt",O_RDWR);
   int fd2 = open("exam_9.txt",O_RDWR);

   char *data1 = "aaaaaaaaaaa";
   char *data2 = "bbbb";

   write(fd1, data1, strlen(data1));
   write(fd2, data2, strlen(data2));

   close(fd1);
   close(fd2);

   return 0;
}
```

Результат: bbbbaaaaaaaa

В данной программе с помощью системного вызова open() создаются два файловых дескриптора одного и того же открытого файла, то есть создаются две разные записи в системной таблице открытых файлов. Каждая запись будет иметь свою текущую позицию в файле, то есть положения указателей в файле независимы друг от друга. Поэтому каждая строка будет записана с начальной позиции в файле.

Билет №10

1. Создание собственной файловой системы. Структура, описывающая файловую систему и пример ее заполнения. Регистрация и дерегистрация файловых систем. Монтирование файловой системы. Структура struct super_operations. Структура inode_operations. Функции simple и generic. Точка монтирования. Функции монтирования. Функция printk(). Пример создания файловой системы, ее регистрация и монтирование (лаб.раб.).

Структура, описывающая файловую систему и пример ее заполнения. Функции simple и generic.

Структура file_system_type "описывает" создаваемую файловую систему. Среди всех полей этой структуры нас интересуют лишь некоторые из них, поэтому при определении структуры инициализируем только следующие поля:

```
static struct file_system_type myfs_type  =  {
        .owner  =  THIS_MODULE,
        .name  =  "myfs",
        .mount  =  myfs_mount,
        .kill_sb  =  kill_block_super,
};
```

owner отвечает за счетчик ссылок на модуль
name хранит название файловой системы

mount и kill_sb два поля хранящие указатели на функции. Первая функция будет вызвана при монтировании файловой системы, а вторая при размонтировании.

kill_block_super - фнукция, которую предоставляет ядро. Можно использовать соответствующую generic-функцию
kill_little_super() - это generic-функция, предоставляемая VFS, она освобождает все внутренние структуры при размонтировании ФС.

Для каждого типа файловой системы существует только одна структура file_system_file, независимо от того, сколько таких файловых систем смонтировано и смонтирован ли хотя бы один экземпляр файловой системы.

Регистрация и дерегистрация файловых систем.

Жизнь любой файловой системы начинается с регистрации. Зарегистрировать файловую систему можно с помощью системного вызова register_filesystem(). Регистрация файловой системы выполняется в функции инициализации модуля. Для дерегистрации файловой системы используется функция unregister_filesystem(), которая вызывается в функции выхода загружаемого модуля.

Монтирование файловой системы.

Теперь рассмотрим функцию myfs_mount. Она должна примонтировать устройство и вернуть структуру, описывающую корневой каталог файловой системы:

```
static struct dentry* myfs _ mount (struct file_system_type *type, int flags, char const *dev,
    void *data)
{
```

```
        struct dentry* const entry = mount_bdev(type,  flags,  dev,  data, myfs_fill_sb) ;
        if (IS_ERR(entry))
              printk(KERN_ERR  "MYFS mounting failed !\n") ;
        else
             printk(KERN_DEBUG "MYFS mounted!\n") ;
        return entry;
    }
```

Когда делается запрос на монтирование файловой системы в каталог в определенном пространстве имен, VFS вызывает соответствующий метод mount() для конкретной файловой системы.

Когда файловая система монтируется, создается структура struct vfsmount, которая представляет конкретный экземпляр файловой системы, или, другими словами, точка монтирования.

Определены следующие флаги монтирования:

```
#define MNT_NOSUID 0x01  /*запрещает использование флагов setuid и setgid*/
#define MNT_NODEV  0x02  /*запрещает доступ к файлам устройств*/
#define MNT_NOEXEC          0x04 /*запрещает выполнение программ*/
#define MNT_NOATIME         0x08
#define MNT_NODIRATIME      0x10
#define MNT_RELATIME        0x20
#define MNT_READONLY        0x40        /* does the user want this to be r/o? */

struct vfsmount {
        struct dentry *mnt_root;      /* root of the mounted tree */
        struct super_block *mnt_sb;           /* pointer to superblock */
        int mnt_flags; /*флаги монтирования*/
    } __randomize_layout;
```

большая часть работы происходит внутри функции mount_bdev(), но нас интересует лишь ее параметр myfs_fill_sb. Это указатель на функцию, которая будет вызвана из mount_bdev, чтобы проинициализировать суперблок. Сама функция myfs_mount должна вернуть структуру dentry (переменная entry), представляющую корневой каталог нашей файловой системы, а создаст его функция myfs_fill_sb.

В первую очередь заполняется структура super_block: магическое число, по которому драйвер файловой системы может проверить, что на диске хранится именно та самая файловая система, а не что-то еще или прочие данные , операции для суперблока, его размер.

```
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 дейтсвие с супеблоком (низкоуровневые действия с Ф.С.)
Когда система нуждается в выполнении операции над суперблоком:
sb -> s_op -> write_super(sb);

Заполнение
В put_super мы сохраним деструктор нашего суперблока.

```
static void myfs_put_super(struct super_block * sb)
{
        printk(KERN_DEBUG "MYFS super block destroyed!\n" ) ;
}

static struct super_operations const myfs_super_ops = {
        .put_super = myfs_put_super,
        .statfs = simple_statfs,
        .drop_inode = generic_delete_inode,
};
```

Проинициализировав суперблок, функция myfs_fill_sb() выполняет построение корневого каталога нашей ФС. Первым делом для него создается inode вызовом myfs_make_inode. Он нуждается в указателе на суперблок и аргументе mode.

Отдельный dentry обычно имеет указатель на inode. inode - это объекты файловой системы

На inode определены следующие операции:

```
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*);
...
}
```

Далее для корневого каталога создается структура dentry, через которую он помещается в directory-кэш. Заметим, что суперблок имеет специальное поле, хранящее указатель на dentry корневого каталога, которое также устанавливается myfs_fill_sb.

Функции монтирования

Монтирование это – система действий, в результате которой файловая система устройства становиться доступной.

Монтирование корневой файловой системы (/) является частью процесса инициализации ОС. Все остальные файловые системы невозможно использовать до тех пор, пока они не будут смонтированы в определенных точках монтирования. Файловые системы, перечисленные в /etc/fstab монтируются в процессе загрузки системы. Файл / etc / fstab содержит список записей в следующей форме:

```
[File System] [Mount Point] [File System Type] [Options] [Dump] [Pass]
```

Для монтирования нужно знать имя устройства, связанного с конкретным устройством хранения, и каталог, к которому файловая система монтируется.
Точкой монтирования является обычная директория дерева каталогов.
Создадим образ диска:

touch image

Кроме того, нужно создать каталог, который будет точкой монтирования (корнем) файловой системы:

mkdir dir

Теперь, используя этот образ, примонтируем файловую систему:

sudo mount -o loop -t myfs ./image ./dir

Если операция завершилась удачно, то в системном логе можно увидеть сообщения от модуля (dmesg). Чтобы размонтировать файловую систему делаем так:

sudo umount ./dir

Функция printk().
Функция printk позволяет отправлять сообщения в системный журнал.
* Printk() - функция ядра, которая выводит информацию в журнал var/log/message. * Часто используется для диагностического вывода из загружаемого модуля ядра. Первому параметру **может** предшествовать (а может и не предшествовать) константа квалификатор, определяющая уровень сообщений.
Есть восемь возможных строк уровня логирования, определённых в заголовке <linux/kernel.h>; мы перечисляем их в порядке убывания серьёзности:
KERN_EMERG

Используется для аварийных сообщений, как правило, тем, которые предшествуют катастрофе.

KERN_ALERT

Ситуации, требующей немедленных действий.

KERN_CRIT

Критические условия, часто связанные с серьёзными аппаратными или программными сбоями.

KERN_ERR

Используется, чтобы сообщить об ошибочных условиях; драйверы устройств часто используют KERN_ERR для сообщения об аппаратных проблемах.

KERN_WARNING

Предупреждения о проблемных ситуациях, которые сами по себе не создают серьёзных проблем с системой.

KERN_NOTICE

Ситуации, которые являются нормальными, но всё же достойны внимания. Различные обстоятельства, относящиеся к безопасности, сообщаются с этим уровнем.

KERN_INFO

Информационные сообщения. Многие драйверы печатают информацию об оборудовании, которую они находят во время запуска с этим уровнем.

KERN_DEBUG

Используется для отладочных сообщений.

Каждая строка (в макроподстановках) представляет собой целое число в угловых скобках. Целые числа в диапазоне от 0 до 7, чем меньше величина, тем больше приоритет.

Пример создания файловой системы, ее регистрация и монтирование (лаб.раб.).

* код из 8 лр *

Билет №11

1. Аппаратные прерывания в Linux: запрос прерывания и линии IRQ. Простейшая схема аппаратной поддержки прерываний (концептуальная трех шинная архитектура системы).

Быстрые и медленные прерывания, пример быстрого прерывания, флаги. Нижняя и верхняя половины обработчиков прерываний: регистрация обработчика аппаратного прерывания, функция регистрации и ее параметры.

Нижние половины: softirq, tasklet, work queue – особенности реализации и выполнения в SMP-системах. Примеры, связанные с планированием отложенных действий (лаб. Раб.)

Аппаратные прерываний возникают от внешних устройств, являются в системе асинхронными событиями, которые возникают независимо от какой-либо выполняемой в системе работы, и их принято делить на следующие группы:

1)Прерывание от системного таймера, которое возникает в системе периодически.

2)Прерывания от устройств ввода-вывода. Возникают по инициативе устройства, когда устройству нужно сообщить процессору о завершении операции ввода-вывода.

3)Прерывания от действий оператора, например, в ОС Windows при нажатии клавишей ctrl_alt_del для вызова task manager.

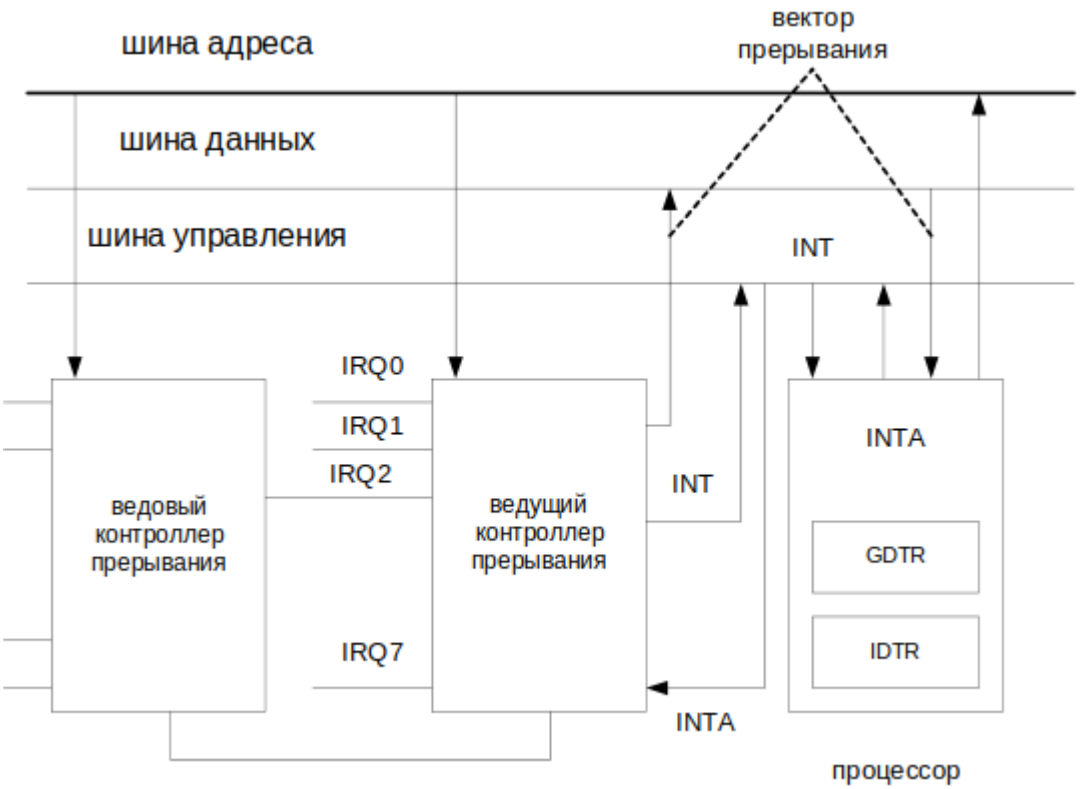
Аппаратные прерывания освобождают процессор от необходимости опрашивать внешние устройства с целью определения их готовности передать запрошенные процессом данные.

Схема обработки аппаратных прерываний — это принципиально архитектурно зависимое действие, связанное с непосредственным взаимодействием с контроллером прерываний.

Первой была микросхема Intel 8259 PIC, которая имела 8 входных линий (IRQ0-7), и одну выходную линию INTR (или просто INT). Сигналы прерывания от устройств ввода-вывода поступают на входы IRQ (Interrupt Request), а контроллер прерывания формирует сигнал прерывания, который по шине управления (линии INTR) поступает на соответствующую ножку (pin) процессора. Сигнал прерывания будет передан процессору, если он не замаскирован, т.е. его обработка разрешена.

Операции, выполняемые на единственном процессоре, рассматриваются только как тривиальный, вырожденный случай SMP (SMP - **S**ymmetric **M**ulti**P**rocessing —симметричное мультипроцессирование).

Схема, представленная на рис.3, может рассматриваться как абстракция, позволяющая лаконично продемонстрировать адресацию обработчиков прерываний в защищенном режиме.



Быстрые и медленные прерывания, пример быстрого прерывания, флаги.

Быстрые прерывания

В ОС Linux принято различать быстрые и медленные прерывания. Быстрые прерывания выполняются при запрете всех прерываний на текущем процессоре. На других процессорах прерывания могут обрабатываться, но при запрете прерываний по линии IRQ, относящейся к выполняемому быстрому прерыванию. Таким образом, выполнение быстрого прерывания не может быть прервано. В современной ОС Linux быстрым прерыванием является только прерывание от системного таймера **__IRQF_TIMER** 0x00000200.

Медленные прерывания

Медленные прерывания могут потребовать значительных затрат процессорного времени. После инициирования прерывания его обработчик должен выполняться так быстро как только возможно, чтобы не прерывать текущую активность на длительное время. Обработчики аппаратных прерываний выполняются на очень высоком уровне приоритета и блокируют возникновение других запросов прерываний. Буквальный перевод soft irq – мягкие или гибкие irq, более соответствует их назначению, чем принятое название – «отложенные прерывания», так как все нижние половины выполняются как отложенные действия.

Выполнение отложенных действий возложено на потоки ядра, которые реализованы как демоны softirq (ksoftirqd – сокращение от kernel softirq deamon). Чтобы сократить время выполнения обработчиков прерываний обработчики медленных аппаратных прерываний делятся на две части, которые традиционно называются верхняя (top) и нижняя (bottom) половины (half).

Нижняя и верхняя половины обработчиков прерываний: регистрация обработчика аппаратного прерывания, функция регистрации и ее параметры.

Верхними половинами остаются обработчики, устанавливаемы функцией request_irq() на определенных IRQ. Выполнение нижних половин инициируется верхними половинами, т.е. обработчиками прерываний.

С современных ОС Linux имеется три типа нижних половин (bottom half):

- softirq – отложенные прерывания;
- tascklet – тасклеты;
- workqueue – очереди работ.

Драйверы регистрируют обработчик аппаратного прерывания и разрешают определенную линию irq посредством функции:

<linux/interrupt.h>

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

где: irq – номер прерывания, *handler –указатель на обработчик прерывания, irqflags – флаги, devname – ASCII текст, представляющий устройство, связанное с прерыванием, dev_id – используется прежде всего для разделения (shared) линии прерывания.

Флаги:

#define IRQF_SHARED 0x00000080 /*разрешает разделение irq несколькими устройствами*/
#define IRQF_PROBE_SHARED 0x00000100 /*устанавливается абонентами, если возможны проблемы при совместном использовании irq*/
#define IRQF_TIMER 0x00000200 /*флаг, маскирующий данное прерывание как прерывание от таймера*/
#define IRQF_PERCPU 0x00000400 .*прерывание закрепленное за определенным процессором*/
#define IRQF_NOBALANCING 0x00000800 /*флаг, запрещающий использование данного прерывания для балансировки irq*/
#define IRQF_IRQPOLL 0x00001000 /*прерывание используется для опроса*.
#define IRQF_ONESHOT 0x00002000
#define IRQF_NO_SUSPEND 0x00004000
#define IRQF_FORCE_RESUME 0x00008000
#define IRQF_NO_THREAD 0x00010000
#define IRQF_EARLY_RESUME 0x00020000
#define IRQF_COND_SUSPEND 0x00040000

Нижние половины: softirq, tasklet, work queue – особенности реализации и выполнения в SMP-системах:

SMP - **S**ymmetric **M**ulti**P**rocessing —симметричное мультипроцессирование

Отложенные прерывания (softirq)

Буквальный перевод soft irq – мягкие или гибкие irq, более соответствует их назначению, чем принятое название – «отложенные прерывания», так как все нижние половины выполняются как отложенные действия.

Выполнение отложенных действий возложено на потоки ядра, которые реализованы как демоны softirq (ksoftirqd – сокращение от kernel softirq deamon). Каждый процессор имеет собсбвенный поток, который называется ksoftirqd/n, где n – номер процессора.

Функция open_softirq() инициализирует гибкое прерывание, присваивая заданному элементу массива softirq_vec значение action.

```
void open_softirq(int nr, void (*action)(struct softirq_action *))
{
    softirq_vec[nr].action = action;
}
```

Данная функция получает два параметра: 1) индекс массива softirq_vec и 2) указатель на функцию softirq, которая будет выполняться.

Структура softirq_action состоит из одного поля: указатель на функцию action, которой передается указатель на отложенное прерывание.

Функция raise_softirq() ставит его в очередь на выполнение.

Могут выполняться параллельно на любом процессоре.

Отложенное прерывание выполняется в контексте прерывания, а значит для него недопустимы блокирующие операции.

Единственное событие, которое может вытеснить softirq это – аппаратное прерывание.

Тасклеты (tasklets)

Тасклеты, в отличие от гибких прерываний, могут создаваться динамически, а также их легче использовать в связи с более простыми требованиями к блокировкам. Это связано с тем, что одинаковые тасклеты не могут выполняться в системе одновременно. Но разные тасклеты могут выполняться параллельно.

Тасклеты — это механизм обработки нижних половин, построенный на основе механизма отложенных прерываний.

Тасклеты описываются структурой:

```
struct tasklet_struct
{
        struct tasklet_struct *next; /* указатель на следующий
тасклет*/
        unsigned long state; /* текущее состояние*/
        atomic_t count; /* счетчик ссылок*/
        void (*func)(unsigned long); /* обработчик*/
        unsigned long data; /* данные*/
};
```

Поле state может иметь одно из следующих значений:

```
enum
{
    TASCLET_STATE_SCHED, /* запланирован на выполнение*/
     TASKLET_STATE_RUN /* выполняется – только для SMP*/
}
```

Если поле count = 0, то тасклет разрешен и может выполняться, если он помечен как запланированный, иначе тасклет запрещен и не может выполняться.

Объявление:

Тасклеты могут быть зарегистрированы как статически, так и динамически.

Статически тасклеты создаются с помощью двух макросов:

```
DECLARE_TASKLET(name, func, data)
DECLARE_TASKLET_DISABLED(name, func, data);
```

Оба макроса статически создают экземпляр структуры struct tasklet_struct с указанным именем (name).
При динамическом создании тасклета объявляется указатель на структуру struct tasklet_struct *t а затем для инициализации вызывается функция:

tasklet_init(t , tasklet_handler , dev) ;

Планирование:

Тасклеты могут быть запланированы на выполнение с помощью Функций:
tasklet_schedule(struct tasklet_struct *t);
tasklet_hi_sheduler(struct tasklet_struct *t);
void tasklet_hi_schedule_first(struct tasklet_struct *t); /* вне очереди */
Когда tasklet запланирован, ему выставляется состояние **TASKLET_STATE_SCHED**, и
он добавляется в очередь. Пока он находится в этом состоянии, запланировать его еще раз не получится — в этом случае просто ничего не произойдет. Tasklet не может находиться сразу в нескольких местах в очереди на планирование, которая организуется через поле next структуры tasklet_struct.
После того, как тасклет был запланирован, он выполниться один раз.

Очереди работ (workqueues)

Очередь заданий является еще одной концепцией для обработки отложенных функций. Это похоже на тасклет с некоторыми отличиями. Функции рабочих очередей выполняются в контексте процесса ядра, но функции тасклетов выполняются в контексте программных прерываний.

Рабочая очередь поддерживается типом struct work_struct, который определён в include/linux/workqueue.h:

```
struct work_struct {
        atomic_long_t data;
        struct list_head entry;
        work_func_t func;
    #ifdef CONFIG_LOCKDEP
        struct lockdep_map lockdep_map;
    #endif
};
```

Обратим внимание на два поля: func - функция, которая будет запланирована в рабочей очереди, и data - параметр этой функции. Ядро Linux предоставляет специальные потоки для каждого процессора, которые называются kworker

Очередь работ создается функцией (см. приложение 1):
int alloc_workqueue(char *name, unsigned int flags, int max_active);

- name - имя очереди, но в отличие от старых реализаций потоков с этим именем не создается
- flags - флаги определяют как очередь работ будет выполняться
- max_active - ограничивает число задач из данной очереди, которые могут одновременно выполняться на одном CPU.

Может использоваться вызов create_workqueue();

work_struct представляет задачу (обработчик нижней половины) в очереди работ.

Поместить задачу в очередь работ можно во время компиляции (статически):

DECLARE_WORK(name, void (*func)(void *));

где: name – имя структуры work_struct, func – функция, которая вызывается из workqueue – обработчик нижней половины.

Если требуется задать структуру work_struct динамически, то необходимо использовать следующие два макроса:

INIT_WORK(sruct work_struct *work, void (*func)(void),void *data);

PREPARE_WORK(sruct work_struct *work, void (*func)(void),void *data);

После того, как будет инициализирована структура для объекта work, следующим шагом будет помещение этой структуры в очередь работ. Это можно сделать несколькими способами. Во-первых, просто добавить работу (объект work) в очередь работ с помощью функции queue_work (которая назначает работу текущему процессору). Можно с помощью функции queue_work_on указать процессор, на котором будет выполняться обработчик.

* код из ЛР9 *

Билет №12

1. Создание виртуальных файловых систем. Структура, описывающая файловую систему.

Регистрация и дерегистрация файловой системе. Монтирование файловой системы. Точка монтирования.

Кэширование в системе. Кэши SLAB, функции для работы с кэшем SLAB.

Примеры из лабораторной работы.

Функции, определенные на файлах (struct file_operations), функции, определенные на файлах, и их регистрация. Пример из лабораторной работы по файловой системе /proc.

Создание виртуальных файловых систем. Структура, описывающая файловую систему. Регистрация и дерегистрация файловой системе. Монтирование файловой системы. Точка монтирования.

(см билет 10)

Кэширование в системе. Кэши SLAB, функции для работы с кэшем SLAB. Примеры из лабораторной работы.

Кэш slab

Распределитель памяти slab, используемый в Linux, базируется на алгоритме, впервые введенном Джефом Бонвиком (Jeff Bonwick) для операционной системы SunOS. Распределитель Джефа строится вокруг объекта кэширования.Джеф основывался на том, что количество времени, необходимое для инициализации регулярного объекта в ядре, превышает количество времени, необходимое для его выделения и освобождения. Его идея состояла в том, что вместо того, чтобы возвращать освободившуюся память в общий фонд, оставлять эту память в проинициализированном состоянии для использования в тех же целях. Например, если память выделена для mutex, функцию инициализации mutex (mutex_init) необходимо выполнить только один раз, когда память впервые выделяется для mutex. Последующие распределения памяти не требуют выполнения инициализации, поскольку она уже имеет нужный статус от предыдущего освобождения и обращения к деконструктору.

Структура kmem_cache содержит данные, относящиеся к конкретным CPU-модулям, набор настроек (доступных через файловую систему proc), статистических данных и элементов, необходимых для управления кэшем slab.

Функция ядра kmem_cache_create() используется для создания нового кэша. Обычно это происходит во время инициализации ядра или при первой загрузке модуля ядра. Его прототип определен как:

```
struct kmem_cache *kmem_cache_create( const char *name, size_t size, size_t align,
                unsigned long flags, void (*ctor)(void*, struct kmem_cache *, unsigned long),
                void (*dtor)(void*, struct kmem_cache *, unsigned long));
```

name — строка имени кэша;
size — размер элементов кэша (единый и общий для всех элементов);
offset — смещение первого элемента от начала кэша (для обеспечения соответствующего выравнивания по границам страниц, достаточно указать 0, что означает выравнивание по умолчанию);
flags — опциональные параметры (может быть 0);
ctor, dtor — **конструктор** и **деструктор**

Как для любой операции выделения, ей сопутствует обратная операция по уничтожению слаба:

```
int kmem_cache_destroy( kmem_cache_t *cache );
```

Операция уничтожения может быть успешна (здесь достаточно редкий случай, когда функция уничтожения возвращает значение результата), только если уже все объекты, полученные из кэша, были возвращены в него.

После того, как кэш объектов создан, вы можете выделять объекты из него, вызывая функцию kmem_cache_alloc().

```
void kmem_cache_free( kmem_cache_t *cache, const void *obj );
```

* код из ЛР8 *

Функции, определенные на файлах (struct file_operations), функции, определенные на файлах, и их регистрация. Пример из лабораторной работы по файловой системе /proc.

используют структуру, которая хорошо знакома из разработки драйверов, struct file_operations, чтобы назначить методы доступа: open(), read(), write().

Структура file_operations используется для определения обратных вызовов (call back) чтения и записи.

```
static struct file_operations fops = {
.open = pen_open,
.release = pen_close,
.read = pen_read,
.write = pen_write,
};
```

* код из ЛР4 *

Билет №13

1. Файловая подсистема: особенности файловой подсистемы Unix/Linux.: иерархическая структура файловой подсистемы.

Виртуальная файловая система VFS в Linux. Четыре структуры VFS – super_block, inode, dentry, file их назначение. Адресация файлов большого размера в файловой системе extX и пример, показывающий доступ к файлу /usr/ast/mbox.

Монтирование файловых систем. Команда mount и функции монтирования, пример из лаб. Раб.

управление файлами осуществляется частью ОС - файловой системой (файловой подсистемой) File System

файл - каждая поименованная совокупность данных хранящаяся во вторичной памяти

файловая система - это часть ОС, которая отвечает за возможность хранения информации и доступа к ней

файловая система

а) определяет формат сохраненных данных и способ их физического хранения

б) связывает формат физического хранения и api для доступа к файлам

в) Должна обеспечивать создание, чтение, запись, удаление, переименование файлов

Задачи:

1.Именование файлов с точки зрения пользователя

2.Обеспечение программного интерфейса для работы с файлами пользователя и приложения

3.Отображение логической модели файлов на физ. реализацию хранения данных во вторич. памяти.

4.Обеспечение надежного хранения файлов, доступа к ним и защиты от несанкционированного доступа

5.Обеспечение совместного использования файлов.

Иерархическая организация ФС:

Символьный уровень - уровень именования файлов удобный для пользователей (уровень каталогов и имен файлов которыми оперирует приложение)
Базовый уровень - уровень идентификации файла файловая подсистема - программа, которая работает по определенным принципам =>

любая переменная должна быть идентифицирована
Логический уровень - логический уровень позволяет обеспечить доступ к данным в файле в формате отличном от формата их физического хранения
Физический уровень - обеспечение непосредственного доступа к информации на внешнем носителе
Физическое уст-во

Файловая подсистема LINUX: поддержка большого числа файловых систем:

Чтобы иметь возможность поддерживать большое кол-во файловых систем поддерживается спец. интерфейс
в unix VFS/vnode
в linux VFS

Виртуальная файловая система (Virtual File System, известная также как Virtual Fileystem Switch) - это подсистема в ядре Linux, который обеспечивает интерфейс файловой системы для программ пользовательского пространства. Он также предоставляет абстракцию в ядре, которая позволяет сосуществовать различным реализациям файловой системы.

VFS определяет интерфейс, который должны поддерживать конкретные файловые системы, чтобы работать в Linux.

struct vfs

```
{
    struct vfs *vfs_next;
    struct vfsops *vfs_op;
    ...
}
```

VFS определяет четыре базовые абстракции:

1. struct superblock

определяет конкретную файловую систему

суперблок - это структура, которая содержит информацию необходимую для

монтирования и управления файловой системой

каждая файловая система имеет один суперблок

2. struct inode

структура описывает конкретный файл (2 типа inode: * дисковый inode (содержит адреса всех блоков диска, в которых наход данные конкрет файла) * inode памяти)

3. 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

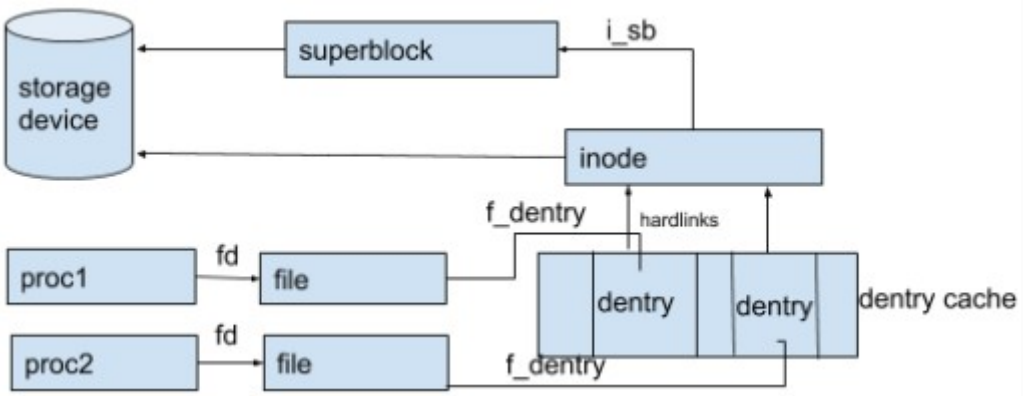
4. struct file

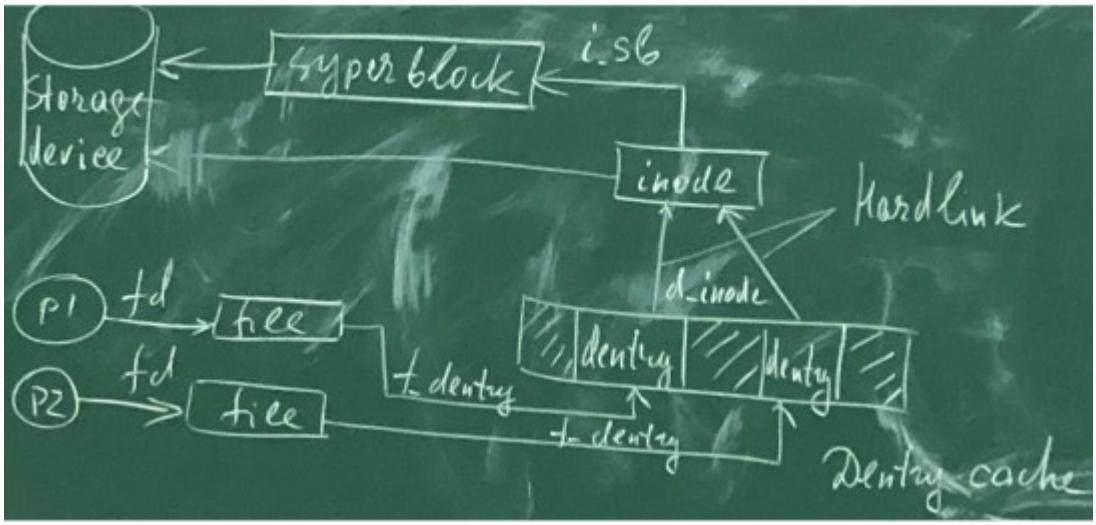
- одна из четырех структур, определяемых как основные для работы с файлами.

FILE - структура, представляющая открытые файлы.

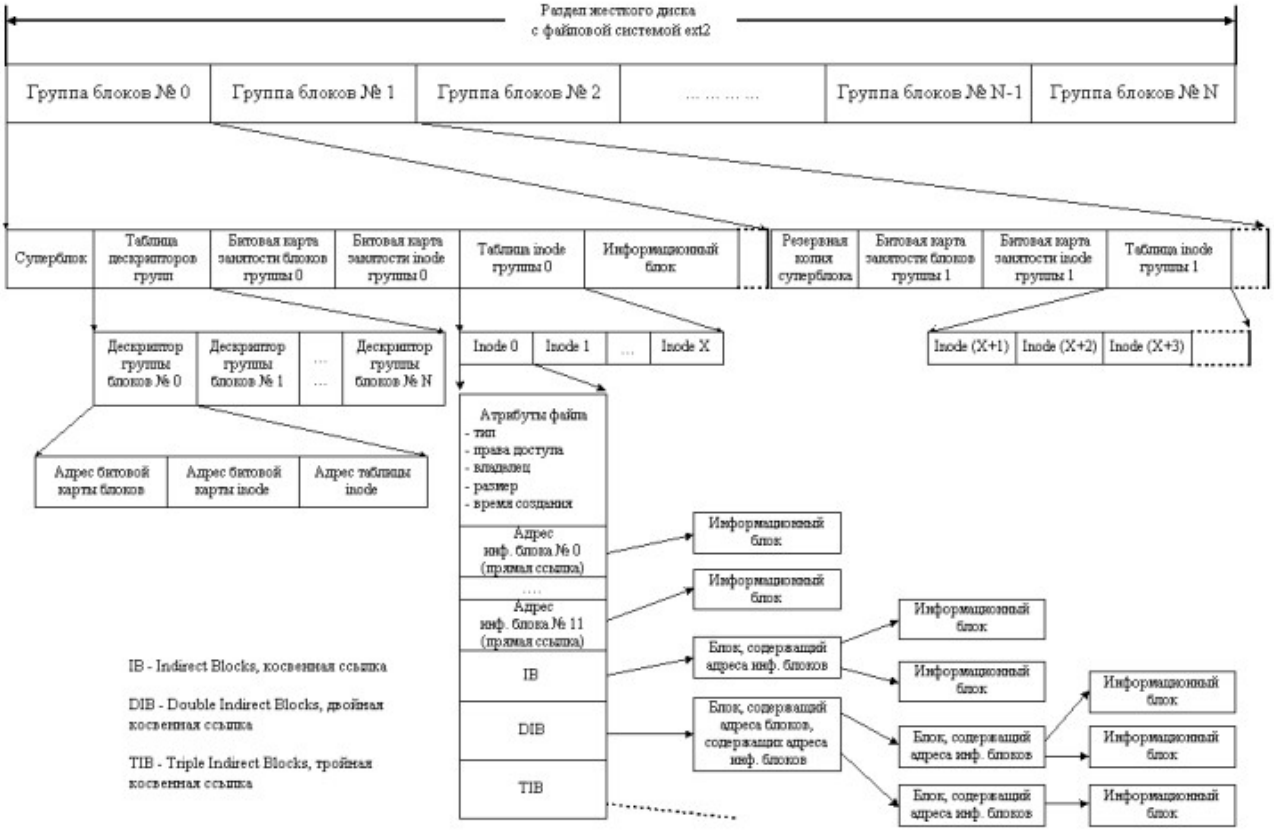
(выписать данные структуры)

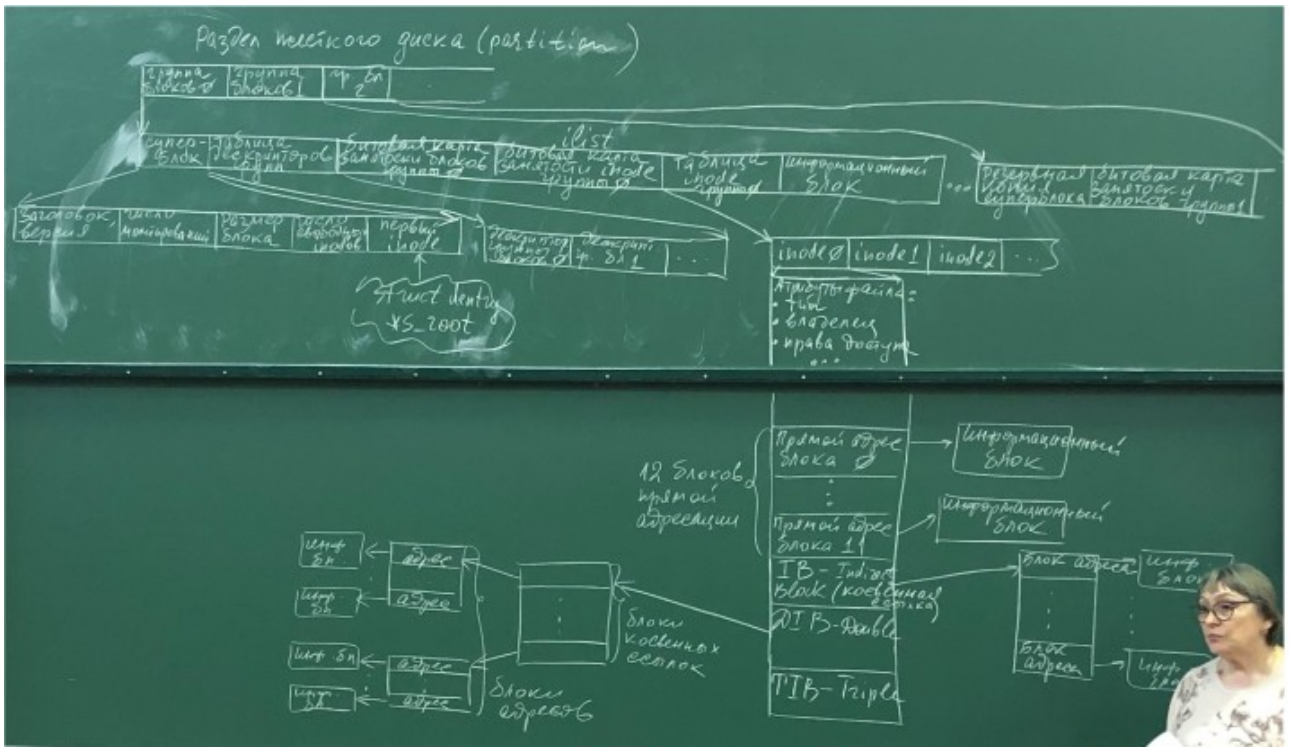
Связь:





Раздел жесткого диска и суперблок:





Суперблок считывается в память ядра при монтировании Ф.С. и остается там до отмонтирования. Поэтому есть флаги dirty и пр.
суперблок - это структура, которая содержит информацию необходимую для монтирования и управления файловой системой
суперблок - это структура, которая находится на диске
каждая файловая система имеет один суперблок
но на диске суперблок находится в нескольких экземплярах (для надежности)

такая структура должна определять параметры для управления файловой системы (суммарное число блоков, корневой inode, ...)
суперблок существует не только на диске
суперблок на диске предоставляет ядру системы информацию о структуре файловой системы (но это не точно)
суперблок в памяти предоставляет информацию необходимую для управления смонтированной файловой системой
<linux/fs.h>
struct super_block
{
struct list_head s_list; // список суперблоков
k_dev_t s_dev; // указывает устройство на котором находится файловая система
unsigned long long s_blocksize; // размер блока в байтах
unsigned char s_dirt; // флаг показывающий, что суперблок был изменен
unsigned long s_max_byte; //
struct file_system_type *s_type; // тип файловой системы
struct super_operations *s_op; // перечислены действия определенные на суперблоках
...
unsigned long s_magic; // магическое число файловой системы
struct dentry *s_root; // точка монтирования файловой системы // каталог монтирования файловой системы
...
int s_count; // счетчик ссылок на суперблок
...
struct list_head s_inodes; // все inode
const struct dentry_operations *s_d_op; // определяет dentry_operations для директорий
struct list_head s_dirty; // список измененных индексов
...
struct block_device *s_bdev; // драйвер
...
char s_id[32]; // строка имени
...
}
тк линукс поддерживает большое количество одновременно смонтированных файловых подсистем
то будет большое количество блоков
и они хранятся в struct list_heard - список суперблоков
суперблок описывает смонтированные файловые системы
но каждая файловая система описывается структурой struct file_system_type

файловая система занимает раздел жесткого диска
поделена на группы блоков
такая файловая система описывается структурой суперблок
очевидно
суперблок описывает файловую систему
задача файловой системы хранить информацию - файлы
чтобы иметь доступ к ним
они должны быть нумерованные
нумеруются inodами
описывается в struct inode
эффективнее всего хранить это в виде битовой карты

в соответствии со структурой struct superblock
там видим dentry *s_root (указатель на корневой каталог)
в системе все является файлом
все файлы в системе имеют inode
entry - directory entry
struct dentry создается налету
она нигде не хранится
и она создается на основе информации из inode
дисковый inode хранит информацию о физическом файле и позволяет получить доступ к информации записанной в файле
ОС linux поддерживает файлы очень больших размеров
чтобы получить доступ к блоку необходимо хранить его адрес
существует 12 блоков прямой адресации
блок indirect - этот блок ссылается на блок, в котором хранятся адреса блоков по тому же принципу

видим DIB - двойная косвенная адресация
TIB - тройная косвенная адресация

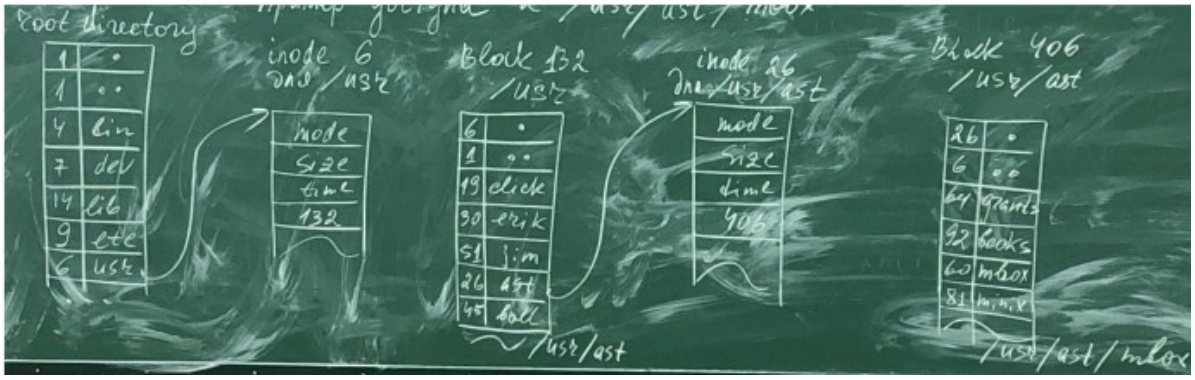
Структура struct super_operations.

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 дейтсвие с супеблоком (низкоуровневые действия с Ф.С.)
Когда система нуждается в выполнении операции над суперблоком:
sb -> s_op -> write_super(sb);

пример, показывающий доступ к файлу /usr/ast/mbox.



Монтирование файловых систем. Команда mount и функции монтирования, пример из лаб. Раб

Монтирование файловой системы.

Теперь рассмотрим функцию myfs_mount. Она должна примонтировать устройство и вернуть структуру, описывающую корневой каталог файловой системы:

```
static struct dentry* myfs _ mount (struct file_system_type *type, int flags, char const *dev,
    void *data)
{
    struct dentry* const entry = mount_bdev(type,  flags,  dev,  data, myfs_fill_sb) ;
    if (IS_ERR(entry))
        printk(KERN_ERR  "MYFS mounting failed !\n") ;
    else
        printk(KERN_DEBUG "MYFS mounted!\n") ;
    return entry;
}
```

Когда делается запрос на монтирование файловой системы в каталог в определенном пространстве имен, VFS вызывает соответствующий метод mount() для конкретной файловой системы.

Когда файловая система монтируется, создается структура struct vfsmount, которая представляет конкретный экземпляр файловой системы, или, другими словами, точка монтирования.

Определены следующие флаги монтирования:

```
#define MNT_NOSUID 0x01  /*запрещает использование флагов setuid и setgid*/
#define MNT_NODEV  0x02  /*запрещает доступ к файлам устройств*/
#define MNT_NOEXEC          0x04 /*запрещает выполнение программ*/
#define MNT_NOATIME         0x08
#define MNT_NODIRATIME      0x10
#define MNT_RELATIME        0x20
#define MNT_READONLY        0x40        /* does the user want this to be r/o? */

struct vfsmount {
        struct dentry *mnt_root;     /* root of the mounted tree */
        struct super_block *mnt_sb;          /* pointer to superblock */
        int mnt_flags; /*флаги монтирования*/
} __randomize_layout;
```

большая часть работы происходит внутри функции mount_bdev(), но нас интересует лишь ее параметр myfs_fill_sb. Это указатель на функцию, которая будет вызвана из mount_bdev, чтобы проинициализировать суперблок. Сама функция myfs_mount должна вернуть структуру dentry (переменная entry), представляющую корневой каталог нашей файловой системы, а создаст его функция myfs_fill_sb.

В первую очередь заполняется структура super_block: магическое число, по которому драйвер файловой системы может проверить, что на диске хранится именно та самая файловая система, а не что-то еще или прочие данные , операции для суперблока, его размер.

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 дейтсвие с супеблоком (низкоуровневые действия с Ф.С.)
Когда система нуждается в выполнении операции над суперблоком:
sb -> s_op -> write_super(sb);

Заполнение
В put_super мы сохраним деструктор нашего суперблока.

```
static void myfs_put_super(struct super_block * sb)
{
        printk(KERN_DEBUG "MYFS super block destroyed!\n" ) ;
}

static struct super_operations const myfs_super_ops = {
        .put_super = myfs_put_super,
        .statfs = simple_statfs,
        .drop_inode = generic_delete_inode,
};
```

Проинициализировав суперблок, функция myfs_fill_sb() выполняет построение корневого каталога нашей ФС. Первым делом для него создается inode вызовом myfs_make_inode. Он нуждается в указателе на суперблок и аргументе mode.

Отдельный dentry обычно имеет указатель на inode. inode - это объекты файловой системы
На inode определены следующие операции:

```
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*);
...
}
```

Далее для корневого каталога создается структура dentry, через которую он помещается в directory-кэш. Заметим, что суперблок имеет специальное поле, хранящее указатель на dentry корневого каталога, которое также устанавливается myfs_fill_sb.

Функции монтирования

Монтирование это – система действий, в результате которой файловая система устройства становиться доступной.
Монтирование корневой файловой системы (/) является частью процесса инициализации ОС. Все остальные файловые системы невозможно использовать до тех пор, пока они не будут смонтированы в определенных точках монтирования. Файловые системы, перечисленные в /etc/fstab монтируются в процессе загрузки системы. Файл / etc / fstab содержит список записей в следующей форме:
`[File System] [Mount Point] [File System Type] [Options] [Dump] [Pass]`
Для монтирования нужно знать имя устройства, связанного с конкретным устройством хранения, и каталог, к которому файловая система монтируется.
Точкой монтирования является обычная директория дерева каталогов.
Создадим образ диска:

touch image

Кроме того, нужно создать каталог, который будет точкой монтирования (корнем) файловой системы:

mkdir dir

Теперь, используя этот образ, примонтируем файловую систему:

sudo mount -o loop -t myfs ./image ./dir

Если операция завершилась удачно, то в системном логе можно увидеть сообщения от модуля (dmesg). Чтобы размонтировать файловую систему делаем так:

sudo umount ./dir

* код из ЛР8 *

Билет №14

1. Файловая подсистема /proc – назначение, особенности, файлы, поддиректории, ссылка self, информация об окружении, состоянии процесса, прерываниях. Структура **proc_dir_entry**: функции для работы с элементами /proc. Структура, перечисляющая функции, определенные на файлах. Использование структуры file_operations для регистрации собственных функций работы с файлами. Передача данных их пространства пользователя в пространство ядра и из ядра в пространство пользователя. Обоснование необходимости этих функций. Функция printk() – назначение и особенности. Пример программы «Фортунки» из лаб. Работы.

(см билет 4)

Структура, перечисляющая функции, определенные на файлах.

используют структуру, которая хорошо знакома из разработки драйверов, struct file_operations, чтобы назначить методы доступа: open(), read(), write().

Структура file_operations используется для определения обратных вызовов (call back) чтения и записи.

```
static struct file_operations fops = {
.open = pen_open,
.release = pen_close,
.read = pen_read,
.write = pen_write,
};
```

Функция printk() – назначение и особенности.

Функция printk позволяет отправлять сообщения в системный журнал.

* Printk() - функция ядра, которая выводит информацию в журнал var/log/message. * Часто используется для диагностического вывода из загружаемого модуля ядра. Первому параметру **может** предшествовать (а может и не предшествовать) константа квалификатор, определяющая уровень сообщений.

Есть восемь возможных строк уровня логирования, определённых в заголовке <linux/kernel.h>; мы перечисляем их в порядке убывания серьёзности:

KERN_EMERG

Используется для аварийных сообщений, как правило, тем, которые предшествуют катастрофе.

KERN_ALERT

Ситуации, требующей немедленных действий.

KERN_CRIT

Критические условия, часто связанные с серьёзными аппаратными или программными сбоями.

KERN_ERR

Используется, чтобы сообщить об ошибочных условиях; драйверы устройств часто используют KERN_ERR для сообщения об аппаратных проблемах.

KERN_WARNING

Предупреждения о проблемных ситуациях, которые сами по себе не создают серьёзных проблем с системой.

KERN_NOTICE

Ситуации, которые являются нормальными, но всё же достойны внимания. Различные обстоятельства, относящиеся к безопасности, сообщаются с этим уровнем.

KERN_INFO

Информационные сообщения. Многие драйверы печатают информацию об оборудовании, которую они находят во время запуска с этим уровнем.

KERN_DEBUG

Используется для отладочных сообщений.

Каждая строка (в макроподстановках) представляет собой целое число в угловых скобках. Целые числа в диапазоне от 0 до 7, чем меньше величина, тем больше приоритет.

* код из ЛР4 *

Билет №15

1. Загружаемые модули ядра. Структура загружаемых модулей. Информация о процессах, доступная в ядре. Пример вывода информации о запущенных процессах, символ current (лаб. раб.).

Взаимодействие загружаемых модулей в ядре. Экспорт данных. Пример взаимодействия модулей (лаб. раб.). Функция printk() – назначение и особенности.

Регистрация функций работы с файлами. Пример заполненной структуры. Передача данных из пространства ядра в пространство пользователя и из пространства пользователя в пространство ядра. Примеры из лабораторный работ.

Загружаемые модули ядра. Структура загружаемых модулей. Информация о процессах, доступная в ядре. Пример вывода информации о запущенных процессах, символ current (лаб. Раб.).

Unix/Linux имеет монолитное ядро. Монолитное ядро состоит из модулей.

Загружаемые модули ядра должны содержать два макроса

module_init и module_exit . Это точки входа. Драйвер имеет много точек входа. Это значит, что эти функции будут вызываться по определенному событию в системе.

Макрос module_init

Служит для регистрации функции инициализации модуля. Макрос принимает имя

функции в качестве фактического параметра. В результате эта функция будет вызываться, при загрузке модуля в ядро. Передаваемая функция должна соответствовать следующему прототипу:

```
int func_init(void);
```

Функция возвращает значение типа int. Если функция инициализации завершилась успешно, то возвращается значение ноль. В случае ошибки возвращается ненулевое значение.

Как правило, функция инициализации предназначена для запроса ресурсов и выделения памяти под структуры данных и т.п.. Так как функция инициализации редко вызывается за пределами модуля, ее обычно не нужно экспортировать и можно объявить с ключевым словом static.

Макрос module_exit

Макрос служит для регистрации функции, которая вызывается при удалении модуля из ядра. Обычно эта функция выполняет задачу освобождения ресурсов. После завершения функции модуль выгружается.

Функция завершения должна соответствовать прототипу:

```
void func_exit(void);
```

Функцию завершения, как и инициализации, можно объявить как static.

Процесс - программа в стадии выполнения. Является единицей декомпозиции системы. Является потребителем системных ресурсов.

task_struct описывает запущенный в системе процесс, создается динамически.

```
struct task_struct{
        …..
        int                     prio;
        int                     static_prio;
        …
        struct list_head        tasks;
        …
        struct mm_struct        *mm;
        struct mm_struct        *active_mm;
        …
        pid_t                   pid;
        pid_t                   tgid;
        …
        struct list_head        children;
        struct list_head        sibling;
```

```
…
/*  Filesystem information   */
struct fs_struct                *fs;
/*  Open file information   */
struct files_struct             *files;
/*   Namespaces   */
 struct nsproxy                 *nsproxy;
```

Ядро Linux использует циклически замкнутый двухсвязный список записей `struct task_struct` для хранения дескрипторов процессов. Эта структура объявлена в файле *linux/sched.h*.

Связанные списки объявлены в *linux/list.h* и их структура очень проста:

```
struct list_head {
        struct list_head *next, *prev;
};
```

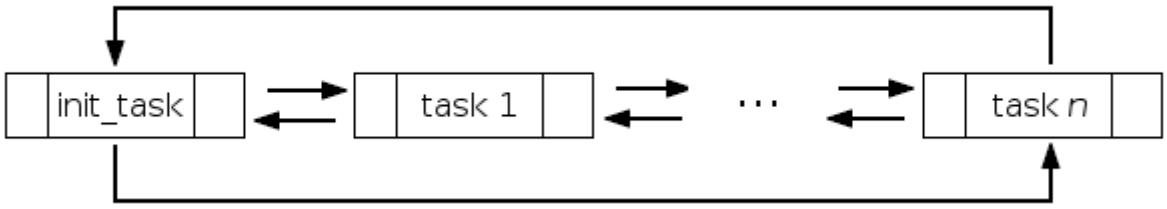
Поиск `struct list head` внутри определения `struct task struct` дает нам следующую строку:

```
struct list_head tasks;
```

Эта строка показывает, что ядро использует циклический связанный список для хранения задач. Это означает, что мы можем использовать стандартные макросы и функции для работы со связанными списками с целью просмотра полного списка задач.

Как известно, "отцом всех процессов" в системе Linux является процесс *init*. Так что он должен стоять в начале списка, хотя, строго говоря, начала не существует раз речь идет о циклическом списке. Дескриптор процесса *init* задается статично (is statically allocated):

```
extern struct task_struct init_task;
```



`next_task()` - макрос, определенный в linux/sched.h, возвращает следующую задачу из списка.

Макрос `current` - это ссылка на дескриптор (указатель на `task_struct`) текущего исполняющегося процесса.

insmod

Загружает модуль в ядро из конкретного файла, если модуль зависит от других модулей,

которые не загружены в ядро, то выдает ошибку и не загружает модуль. Только

суперпользователь может загрузить модуль в ядро.

lsmod

Выводит список модулей, загруженных в ядро.

rmmod

Команда используется для выгрузки модуля из ядра, в качестве параметра передается

имя файла модуля. Только суперпользователь может выгрузить модуль из ядра.

* код из ЛР3 *

Взаимодействие загружаемых модулей в ядре. Экспорт данных. Пример взаимодействия модулей (лаб. Раб.).

Экспортируемые символы

Экспортируемыми символами называются данные и функции, которыми могут пользоваться другие модули ядра. При загрузке модули динамически компонуются с
ядром, в коде модулей могут вызываться только те функции ядра, которые явно экспортируются для использования.

Макрос EXPORTOL

В ядре экспортирование осуществляется с помощью специального макроса EXPORT_SYMBOL(). Функции, которые экспортируются, доступны для использования модулям, остальные функции не могут быть вызваны из модулей. Для экспортируемых
данных правило аналогично.

Пример использования:

```
int md1_data = 42; // экспортируемые данные
extern int md1_func(int n) // экспортируемая функция{
return n * 2;
}
EXPORT_SYMBOL(md1_data);
EXPORT_SYMBOL(md1_func);
```

Для того, чтобы другие модули могли использовать эспортированные символы, они должны “знать” их определение. Для этого применяются заголовочные файлы.

Пример заголовочного файла:

```
extern int md1_data;
extern int md1_func(int n);
```

Макрос EXPORT_SYMBOL_GPL

Иногда необходимо, чтобы символы были доступны только для модулей, имеющих соответствующую лицензию GPL. Для этого используют макрос EXPORT_SYMBOL_GPL().

1. Модуль **md1экспортирует** для использования другими модулями имя процедуры `md1_proc()` и, что далеко не так очевидно, имя структуры данных `md1_data`. Любой другой

модуль (**md2**) может использовать в своём коде любые **экспортируемые** имена. Это могут быть имена, экспортируемые ранее загруженными модулями, но гораздо чаще это имена, **экспортируемые ядром**. Это множество экспортируемых имён ядра далее будет называться **API ядра**. Примером одного из вызовов из набора API ядра в показанных фрагментах кода является вызов `printk()`.
2. Модуль **md2**, использующий экспортируемое имя, связывается с этим именем по прямому **абсолютному адресу**. Как следствие этого, любые изменения (новая сборка), вносимые в ядро или экспортирующие модули, делают собранный модуль непригодным для использования. Именно поэтому бессмысленно предоставлять модуль в собранном виде — он должен собираться только на месте использования.
3.Модуль сможет использовать только те имена, которые явно экспортированы. В модуле **md1** специально показаны два других имени: `md1_local()` является локальным именем (модификатор `static`), непригодным для связывания, а имя `md1_noexport()` не объявлено как экспортируемое имя и также не может быть использовано вне модуля.

* код из ЛР3 *

Функция printk() – назначение и особенности.

Функция printk позволяет отправлять сообщения в системный журнал.
Сама функция не производит запись в системный журнал, а записывает сообщение в специальный буфер ядра. Из буфера ядра записанные сообщения могут быть прочитаны демоном протоколирования.
* Printk() - функция ядра, которая выводит информацию в журнал var/log/message. * Часто используется для диагностического вывода из загружаемого модуля ядра. Первому параметру **может** предшествовать (а может и не предшествовать) константа квалификатор, определяющая уровень сообщений.
Есть восемь возможных строк уровня логирования, определённых в заголовке <linux/kernel.h>; мы перечисляем их в порядке убывания серьёзности:
KERN_EMERG

Используется для аварийных сообщений, как правило, тем, которые предшествуют катастрофе.

KERN_ALERT

Ситуации, требующей немедленных действий.

KERN_CRIT

Критические условия, часто связанные с серьёзными аппаратными или программными сбоями.

KERN_ERR

Используется, чтобы сообщить об ошибочных условиях; драйверы устройств часто используют KERN_ERR для сообщения об аппаратных проблемах.

KERN_WARNING

Предупреждения о проблемных ситуациях, которые сами по себе не создают серьёзных проблем с системой.

KERN_NOTICE

Ситуации, которые являются нормальными, но всё же достойны внимания. Различные обстоятельства, относящиеся к безопасности, сообщаются с этим уровнем.

KERN_INFO

Информационные сообщения. Многие драйверы печатают информацию об оборудовании, которую они находят во время запуска с этим уровнем.

KERN_DEBUG

Используется для отладочных сообщений.

Каждая строка (в макроподстановках) представляет собой целое число в угловых скобках. Целые числа в диапазоне от 0 до 7, чем меньше величина, тем больше приоритет.

Регистрация функций работы с файлами. Пример заполненной структуры. Передача данных из пространства ядра в пространство пользователя и из пространства пользователя в пространство ядра. Примеры из лабораторный работ.

Фактически само ядро и каждый из процессов располагаются в своих собственных изолированных адресных пространствах.

Система предоставляет средства взаимодействия с приложениями:

- copy_to_user(Копирует данные из ядра в пространство пользователя. Вызывающий абонент должен проверить указанный блок с помощью access_ok до вызова этой функции.
Функция возвращает количество байт, которые не могут быть скопированы. В случае успешного выполнения будет возвращен 0.)
- copy_from_user (Копирует данные из пространства пользователя в пространство ядра. Вызывающий абонент должен проверить указанный заданный блок с помощью access_ok до вызова этой функции.
Функция возвращает количество байт, которые не могут быть скопированы. В случае успешного выполнения будет возвращен 0.)

Структуры file_operations для регистрации функций работы с файлами:

Структура file operations используется для определения обратных вызовов (call back) чтения и записи.
используют структуру, которая хорошо знакома из разработки драйверов, struct file_operations, чтобы назначить методы доступа: open(), read(), write().

Структура file_operations используется для определения обратных вызовов (call back) чтения и записи.

```
static struct file_operations fops = {
.open = pen_open,
```

```
    .release = pen_close,
    .read = pen_read,
    .write = pen_write,
    };
```

Обоснование необходимости использования специальных функций для передачи данных из пространства пользователя в ядро и из ядра в пространство пользователя:

Для передачи данных из ядра и в ядро из режима пользователя нужны специальные функции copy_from_user() и copy_to_user(), потому что в Linux память сегментирована, то есть указатель не ссылается на уникальную позицию в памяти, а ссылается на позицию в сегменте (адресация "сегмент-смещение"). Процессу доступен только собственный сегмент памяти. Он может обращаться только к собственным сегментам. Если выполняется обычная программа, то адресация происходит автоматически в соответствии с принятой в системе адресацией. Если выполняется код ядра и необходимо получить доступ к сегменту кода ядра, то всегда нужен буфер, но когда нужно передать информацию между текущим процессом и кодом ядра, то соответствующие функции ядра получают указатель на буфер в сегменте процесса. (тоже самое только другими словами:
1)у процесса свое адресное пространство защищенное
2)для процесса создается виртуальное адресное пространство
3)у него есть последовательность адресов
и отсчитывается смещение от базового адреса сегмента и получаем виртуальный адрес)

* код из ЛР4 *

Билет №16

1. Открытые файлы: системный вызов open():
int open(const char* pathname, int flags);
int open(const char* pathname, int flags, mode_t mode);
пояснить смысл параметров. Основные флаги. Флаг CREATE.
Реализация системного вызова open() в системе – действия в ядре:
SYSCALL_DEFINE3(open,…) -> ksys_open(filename, flags, mode)->do_sys_open()->do_sys_openat2()… найти наименьший файловый дескриптор...
Действия, если флаг O_CREATE установлен?

Открытые файлы: системный вызов open():
int open(const char pathname, int flags);*
int open(const char pathname, int flags, mode_t mode);*
пояснить смысл параметров. Основные флаги. Флаг CREATE.

Процесс - программа в стадии выполнения. Является единицей декомпозиции системы. Является потребителем системных ресурсов.

файловая система - это часть ОС, которая отвечает за возможность хранения информации и доступа к ней

файл - каждая поименованная совокупность данных хранящаяся во вторичной памяти

Структуры данных, связанные с процессом

С каждым процессом в системе связаны список открытых им файлов, корневая файловая система, текущий рабочий каталог, точки монтирования и т.д.

1) task_struct описывает запущенный в системе процесс, создается динамически.

2) Структура struct fs_struct содержит информацию о файловой системе, к которой принадлежит процесс. Структура определена в файле <linux/fs_srtuct.h>.

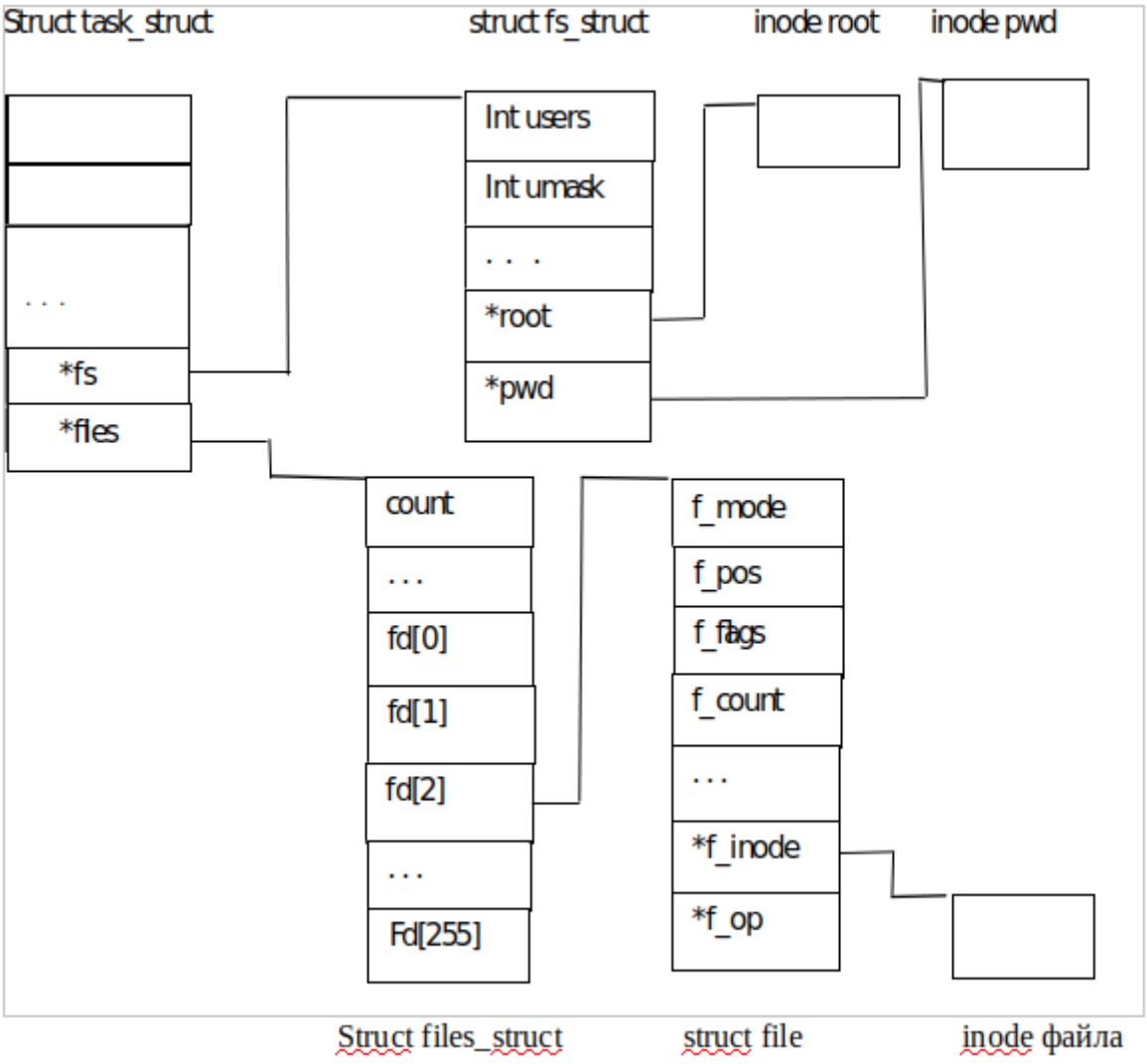
3) Структура struct files_struct определяет дескрипторы файлов, открытых процессом.

Формирует таблицу открытых файлов процесса. Каждый процесс имеет собственную таблицу открытых файлов.

4) Структура struct file определяет дескриптор открытого файла в системе. Любой открытый файл будет иметь в ядре объект файл.

5) Структура struct inode описывает созданный файл.

Обычные файлы расположены во вторичной памяти, на устройстве долговременного хранения, такие файлы называются регулярными. Две большие разницы — открытые файл и просто файлы. Для того, чтобы работать с файлом, файл должен быть открыт.



Чтобы получить возможность прочитать что-то из файла или записать что-то в файл, его нужно открыть. Это делает системный вызов open().

open – открывает и возможно создает файл.

SYNOPSIS top

```
#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);
```

Системный вызов open() открывает файл, указанный в pathname. Если указанный файл не существует, он может (необязательно) (если указан флаг O_CREAT) быть создан open().

Возвращаемое значение open() - это дескриптор файла.

Вызов open () создает новый файловый дескриптор для открытого файла, запись в общесистемной таблице открытых файлов. Эта запись регистрирует **смещение в файле** и флаги состояния файла (модифицируемые с помощью the fcntl (2) операции F_SETFL). Дескриптор файла является ссылкой на одну из этих записей; эта ссылка не влияет, если путь впоследствии удален или изменен ссылаться на другой файл. Новый дескриптор открытого файла изначально не разделяется с любым другим процессом, но разделение может возникнуть через ***fork***((2).

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

Параметр *flags* - это флаги **O_RDONLY**, **O_WRONLY** или **O_RDWR**, открывающие файлы "только для чтения", "только для записи" и для чтения и записи соответственно.

Аргумент *mode* задает права доступа, которые используются в случае создания нового файла.

mode всегда должен быть указан при использовании **O_CREAT**; во всех остальных случаях этот параметр игнорируется. **creat** эквивалентен **open** с *flags*, которые равны **O_CREAT** | **O_WRONLY** | **O_TRUNC**.

Для режима (mode) предусмотрены следующие символические константы:

- Пользователь (владелец файла) имеет права доступа:

S_IRWXU 00700 – на чтение, на запись, на исполнение;
S_IRUSR 00400 - на чтение;
S_IWUSR 00200 - на запись;
S_IXUSR 00100 - на выполнение;

- Группа пользователя имеет права доступа:

S_IRWXG 00070 - на чтение, запись и выполнение;
S_IRGRP 00040 - на чтение;

S_IWGRP 00020 - на запись;
S_IXGRP 00010 - на выполнение;

- Остальные имеют права доступа:

S_IRWXO 00007 - на чтение, запись и выполнение;
S_IROTH 00004 - на чтение;
S_IWOTH 00002 - на запись;
S_IXOTH 00001 - на выполнение,

Если установить флаги следующим образом:
S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH или **0664** это означает для пользователя - чтение/запись, для группы - чтение/запись и для остального мира -чтение.

…

O_PATH (since Linux 2.6.39)
Получите файловый дескриптор, который можно использовать для двух целей: указать местоположение в дереве каталогов файловой системы и выполнить операции, которые действуют исключительно на уровне файлового дескриптора. Сам файл не открывается, и другие файловые операции (например, read (2), write (2), fchmod (2), fchown (2), fgetxattr (2), ioctl (2), mmap (2)) завершаются неудачно с ошибкой EBADF.

…

O_TMPFILE (since Linux 3.11)
Создать неназванный временный обычный файл. Аргумент pathname указывает каталог; безымянный индекс будет создан в файловой системе этого каталога. Все, что записано в результирующий файл, будет потеряно при закрытии последнего дескриптора файла, если файлу не присвоено имя.

Флаг O_TMPFILE должен быть указан с одним из O_RDWR или O_WRONLY и, необязательно, O_EXCL. Если O_EXCL не указан, то linkat (2) может использоваться для связывания временного файла с файловой системой.

и т.д.

Некоторые флаги создания файлов и флаги состояния файлов:
O_CREAT - если файл не существует, то он будет создан
O_APPEND - Файл открывается в режиме добавления

Флаги создания файла: O_CLOEXEC, O_CREAT, O_DIRECTORY, O_EXCL, O_NOCTTY, O_NOFOLLOW, O_TMPFILE и O_TRUNC. Флаги состояния файла это – достаточно большой набор флагов, которые учитываются при открытии или создании файлов в функциях ядра, предназначенных для этой работы. Различие между этими двумя группами флагов состоит в том, что флаги создания файла влияют на семантику самой операции открытия, в то время как флаги состояния файла влияют на семантику последующих операций ввода-вывода. Флаги состояния файла могут быть извлечены и (в некоторых случаях) изменены системным вызовом fcntl(2).

O_APPEND
Перед каждой операцией записи устанавливать указатель текущей позиции на конец файла.

O_SYNC
При открытии обычного файла этот флаг воздействует на последующие операции записи. Если флаг установлен, то каждый вызов write(2) ожидает физического обновления как данных, так и статуса файла.

O_CREAT
Если файл существует, то флаг игнорируется. В противном случае идентификаторы владельца и группы создаваемого файла устанавливаются равными, соответственно, действующим идентификаторам пользователя и группы процесса, а младшие 12 бит

значения режима доступа к файлу устанавливаются равными значению аргумента mode, модифицированному следующим образом [см. creat(2)]:

1. Биты, соответствующие единичным битам маски режима создания файлов текущего процесса [см. umask(2)], устанавливаются равными 0.
2. Бит навязчивости [см. chmod(2)] устанавливается равным 0.

O_TRUNC

Если файл существует, то он опустошается (размер становится равным 0), а режим доступа и владелец не изменяются.

O_EXCL

Если установлены оба флага O_EXCL и O_CREAT, то системный вызов open завершается неудачей, если файл уже существует.

Реализация системного вызова open() в системе – действия в ядре: SYSCALL_DEFINE3(open,…) -> ksys_open(filename, flags, mode)->do_sys_open()->do_sys_openat2()… найти наименьший файловый дескриптор...

Действия, если флаг O_CREATE установлен?

Системный вызов **open()** является оберткой функции ядра **ksys_open()**, которая в свою очередь вызывает функцию **do_sys_open().**

```
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
{
	return ksys_open(filename, flags, mode);

}
```

В теле функции do_sys_open() вызывается функция build_open_how() и функция do_sys_openat2(). Функция do_sys_openat2 вызывает функции: build_open_flags(), getname(filename), get_unused_fd_flags() и, если функция get_unused_fd_flags() возвращает fd > 0 , то вызывается функция do_filp_open(). Функция **do_sys_open**at2() выполняет все задачи системного вызова open. Сначала выполняется проверка правильности флагов и их преобразование во внутреннее представление функцией **build_open_flags**(). В случае неуспешного преобразования и проверки будет возвращена ошибка преобразования и проверки. Затем вызывается функция **getname** (const char**__user** ***filename**), которая выполняет копирование имени файла из пространства пользователя в пространство ядра. Вызываемая функция get_unused_fd_flags() является оберткой функции **__alloc_fd().** Функция __alloc_fd() находит для процесса свободный (наименьший) файловый дескриптор открытого файла и помечает его как занятый. Функция do_filp_open() возвращает указатель на struct **file**, т.е. создается дескриптор открытого файла в системной таблице открытых файлов. Функция set_nameidata() инициализирует структуру nameidata. В последней строке структура записывается в контекст текущего процесса. Затем вызывается функция path_openat(), которая возвращает указатель на struct file, т.е. на дескриптор открытого файла в системной таблице открытых файлов. Функция path_openat выполняет поиск пути. После того, как была выделена новая структура для открытого файла struct file, вызываются функции **do_tmpfile** или **do_o_path** в случае если были установлены флаги **O_TMPFILE|O_CREATE** или **O_PATH** при вызове системного вызова **open().** Иначе будет вызвана функция **path_init()**. Эта функция выполняет некоторые подготовительные работы перед фактическим

поиском пути. После выполнения функции **path_init()** выполняется цикл, в котором вызываются функции **link_path_walk()** и **do_last()**. В do_last() вызывается функция lookup_open(). Функция **lookup_open**() создает inode открываемого файла, заполняя struct file в соответствии со следующим алгоритмом:

1) если не установлен флаг O_CREAT, завершить работу функции;

2) если inode открываемого файла существует, сбросить флаг O_CREAT и завершить работу функции;

3) создать inode открываемого файла с помощью функции **may_o_create**, проверяя при этом права доступа.