

Структуры хранятся тут -

<https://docs.google.com/document/d/1piusiBJq0JuJ3QCFgoE9xMRoywtAHqHbv7SqQtM8fSY/edit?usp=sharing>

Подготовка того семестра

<https://docs.google.com/document/d/19TjjhZyrbhZEjOoMGhdvsMXJU6UI1GEJOfdBblw7b7k/edit?usp=sharing>

4

Билет 1	2
Билет 2(2019)	9
Билет 3	14
Билет 4	20
Билет 5	27
Билет 6	27
Билет 7	33
Билет 8	41
Билет 9	47
Билет 10	49
Билет 11	51
Билет 12	55
Билет 13	62
Билет 14	66
Билет 15	72
Билет 16	75
Билет 18	79
Билет 19	88
Билет 20	96

Билет 1

Управление внешними устройствами: подключение внешних устройств и их идентификация в системе. Система прерываний: типы прерываний и их особенности, прерывания в последовательности ввода-вывода - обслуживание запроса процесса на ввод-вывод. Быстрые и медленные прерывания. Обработчики аппаратных прерываний: регистрация в системе, примеры. Тасклеты - объявление, планирование (пример лаб.раб).

Одна из задач, которые решает ОС - управление устройствами. Unix/Linux рассматривают внешние устройства как специальные файлы устройств.

Специальные Файлы устройств.

Специальные файлы устройств обеспечивают унифицированный доступ к периферийным (внешним) устройствам. Эти файлы обеспечивают связь между файловой системой и драйверами устройств. Такая интерпретация специальных файлов обеспечивает доступ к внешним устройствам как к файлам. Файл устройства может быть открыт, закрыт, в него можно писать, из него можно читать. Каждому внешнему устройству ОС ставит в соответствие минимум один специальный файл в каталоге /dev корневой ФС. В системе имеется 2 типа специальных файлов устройств:

- c - символьные
- b - блочные

Поскольку специальные файлы устройств - файлы, они имеют **inode** (возможно стоит нарисовать структуру inode, но вероятно это оверхед и Рязанова не оценит)

В Linux/Unix устройства идентифицируются так называемыми **старшим** и **младшим** номером устройства. Старший номер идентифицирует драйвер устройства, младший идентифицирует конкретное устройство

Для идентификации устройства в системе имеется тип **dev_t** (POSIX1), определенный в **<sys/types.h>**. Формат полей и их содержание не оговаривается. Для разных версий системы формат отличается, например для 32-битной 12 бит - старший, 20 бит - младший.

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

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

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

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

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

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

struct device написать надо

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

Система прерываний

Прерывание — сигнал от программного или аппаратного обеспечения, сообщающий процессору о наступлении какого-либо события, требующего немедленного внимания. (ВИКИПЕДИЯ)

Классификация прерываний (в зависимости от источника)

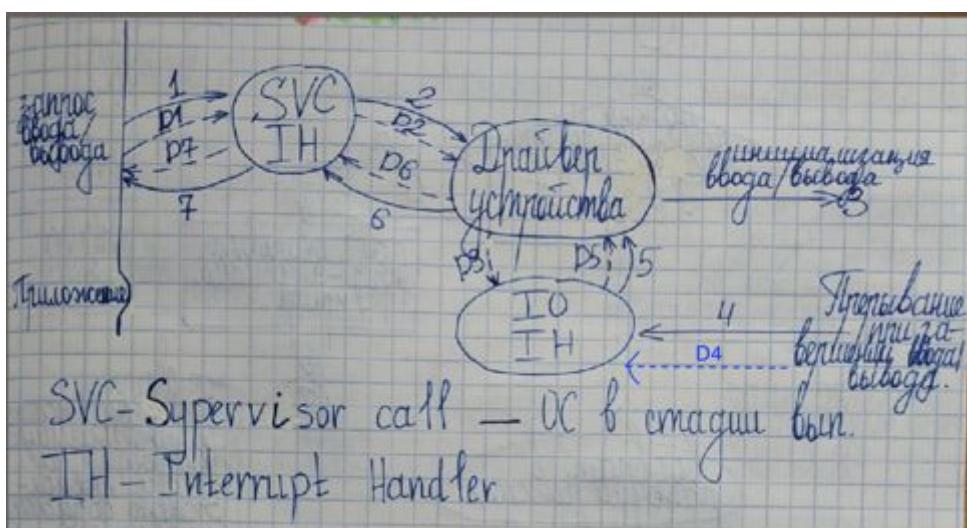
- **Программные** (системные вызовы) – вызываются искусственно с помощью соответствующей команды из программы, предназначены для выполнения некоторых действий ОС (фактически запрос на услуги ОС), является синхронным событием.
- **Аппаратные** – возникают как реакция микропроцессора на физический сигнал от некоторого устройства (клавиатура, системные часы, мышь, жесткий диск и т.д.), по времени возникновения эти прерывания асинхронны, т.е. происходят в случайные моменты времени.
 - От **таймера**
 - От **действий оператора** (Напр. Ctrl + alt + delete)
 - От устройств **ввода/вывода** (посыпается сигнал о завершении процесса вв/выв на контроллер прерываний)
- **Исключения** – являются реакцией микропроцессора на нестандартную ситуацию, возникшую внутри микропроцессора во время выполнения некоторой команды

программы (деление на ноль, прерывание по флагу TF (трассировка)), являются синхронным событием.

- **Исправимые** – приводят к вызову определенного менеджера системы, в результате работы которого может быть продолжена работа процесса (пр. страничная неудача с менеджером памяти)
- **Неисправимые** – в случае сбоя или в случае ошибки программы (пр. ошибка адресации). В этом случае процесс завершается.

Обслуживание запроса процесса на ввод-вывод.

Диаграмма Шоу:



На диаграмме Шоу показана последовательность действий, которые выполняет система, когда приложение выполняет запрос на ввод/вывод.

Любой запрос ввода/вывода блокирует процесс на какое-то время (даже open()!).

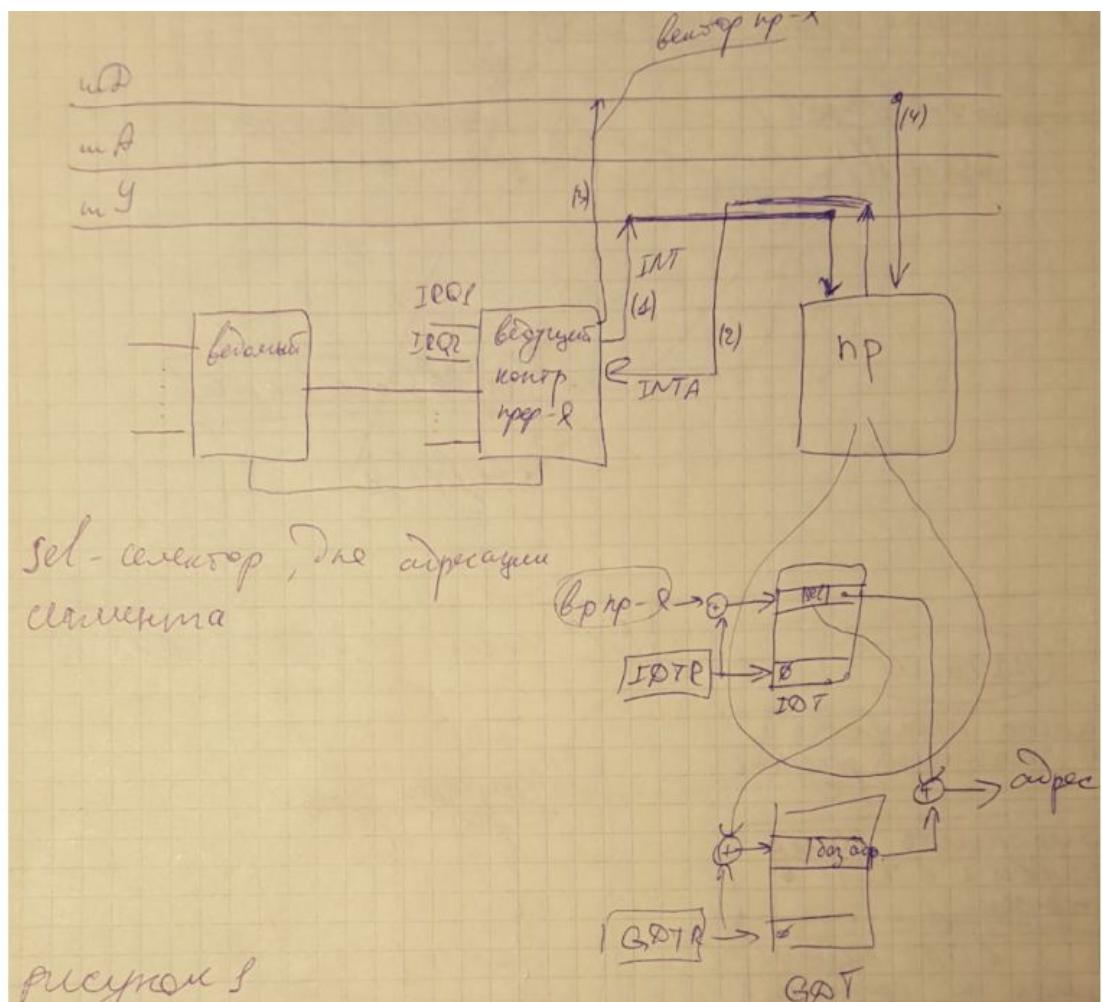
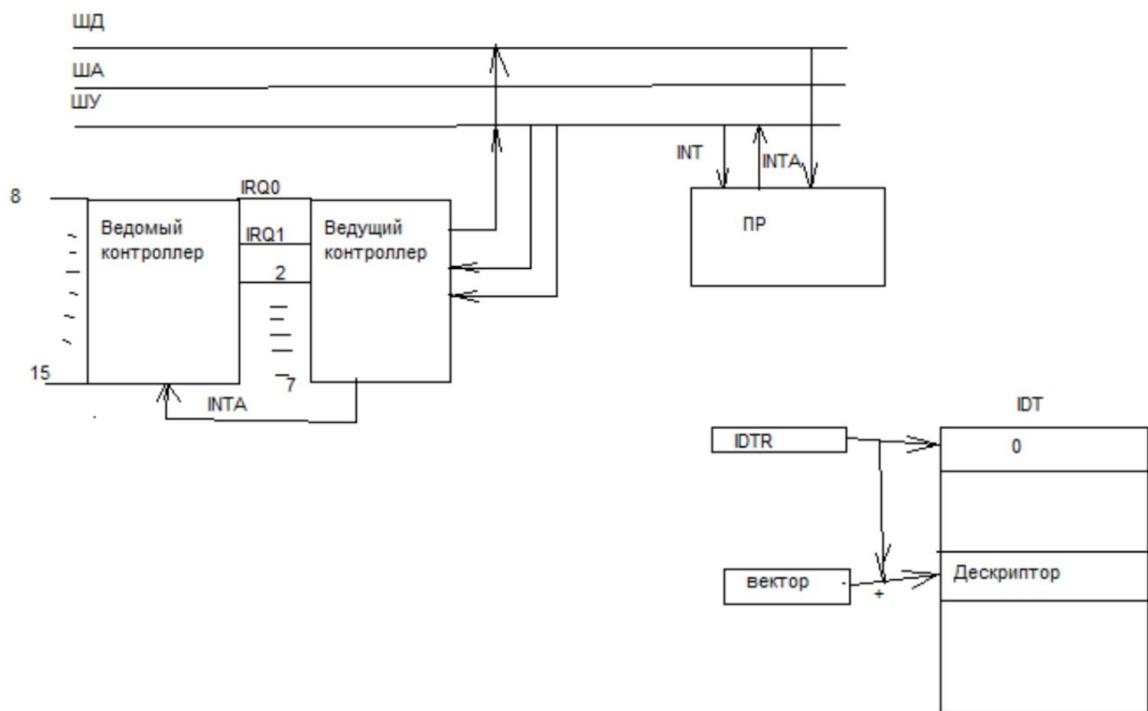
Ни одна система не позволяет процессу напрямую обращаться к внешнему устройству.

// Программа выполняется, когда ей выделен квант времени

При запросе ввода/вывода будут сохранены параметры вызова, произойдет блокировка процесса, а система продолжит обработку запроса.

В результате цепочки вызовов будет вызвана функция драйвера устройства (одна из его точек входа). Драйвер получит параметры вызова и преобразует эти данные в формат «понятный» устройству (кавычки обязательны!!!). Далее драйвер должен сформировать запрос в формате, воспринимаемом устройством, данные идут по шине данных в контроллер устройства.

По завершении операции ввода-вывода контроллер устройства формирует сигнал **прерывания**. Данные от контроллера поступают в регистр данных контроллера прерываний и находятся там пока не сработает цепочка обработки прерывания.



В конце выполнения каждой команды процессор проверяет входной сигнал с шины управления (если прерывания не замаскированы в ОС). Если получен сигнал (INT) – посыпается ответный сигнал контроллеру прерываний (INTA), в ответ контроллер прерываний формирует и посыпает вектор прерывания. Вектор передается по шине данных. Полученный вектор используется для процедуры обработки прерываний.

Затем данные копируются в буфер ядра и вызывается обработчик прерывания (функция callback).

Быстрые и медленные прерывания

Все прерывания от устройств ввода/вывода и системного – таймера аппаратные. Все аппаратные прерывания выполняются на высоком уровне приоритета, и другие операции не могут выполняться пока аппаратное прерывание не будет завершено.

В windows эта же идея реализована в виде DPC – отложенный вызов процедуры.

Аппаратные прерывания принято делить на **быстрые** и **медленные**. Быстрые не делятся на две половины, выполняются как единое целое. Медленные прерывания делятся на две части: **top half** и **bottom half**. В современном Linux быстрым прерыванием является только прерывание от системного таймера. Нижняя половина прерывания выполняется как отложенное действие. Фактически аппаратным прерыванием является верхняя половина. В ее задачи входит получение данных из регистра устройства и помещение их в буфер ядра.

Регистрация обработчика прерывания в системе

Для обработки прерывания драйвер должен разрешить определенную линию прерываний IRQ:

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

irq – номер, прерывания, который будет обрабатывать обработчик.

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

irqflags – битовая маска либо NULL

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

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

dev_id – используется для разделения SHARED линии прерывания.

возвращаемые значение – 0-OK, !=0 – ошибка, E_BUSY – данная линия прерывания уже используется и не указан флаг IRQF_SHARED.

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

irq - численное значение линии прерывания, которую обслуживает обработчик.

dev_id - общий указатель на тот же самый dev_id, что задается в request_irq, когда обработчик был зарегистрирован.

regs - структура, содержащая регистры процессора и состояние перед обслуживанием прерывания. Обычно используется для отладки.

возвращаемое значение: IRQ_NONE - обработчик прерывания обнаруживает прерывание, которое его устройство не инициировало; IRQ_HANDLED - обработчик прерывания был вызван корректно и его устройство действительно вызвало прерывание.

Тасклеты

Главное отличие очередей работ от тасклетов является то, что очереди работ могут блокироваться, а тасклеты нет.

Тасклеты — это механизм обработки нижних половин, построенный на основе механизма отложенных прерываний. Тасклеты представлены двумя типами отложенных прерываний: HI_SOFTIRQ и TASKLET_SOFTIRQ. Единственная разница между ними в том, что тасклеты типа HI_SOFTIRQ выполняются всегда раньше тасклетов типа TASKLET_SOFTIRQ.

struct tasklet_struct написать

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).

Например,

```
DECLARE_TASKLET(my_tasklet, tasklet_handler, dev);
```

Эта строка эквивалентна следующему объявлению:

```
struct tasklet_struct rny_tasklet = { NULL, 0, ATOMIC_INIT(0), tasklet_handler, dev} ;
```

В данном примере создается тасклет с именем my_tasklet , который разрешен для выполнения. Функция tasklet_handler будет обработчиком этого тасклета. Значение параметра dev передается в функцию-обработчик при вызове данной функции.

При динамическом создании тасклета объявляется указатель на структуру

struct tasklet_struct *t а затем для инициализации вызывается функция:

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

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 */
```

Эти функции очень похожи (отличие состоит в том, что одна использует отложенное прерывание с номером TASKLET_SOFTIRQ, а другая — с номером HI_SOFTIRQ).

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 объявления и планирования работы тасклета

```
/* Declare a Tasklet (the Bottom-Half) */
```

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

```
DECLARE_TASKLET(tasklet_example,tasklet_function,tasklet_data);
```

```
...
```

```
/* Schedule the Bottom-Half */
```

```
tasklet_schedule( &tasklet_example );
```

Tasklet можно активировать и деактивировать.

Функции:

```
void tasklet_disable_nosync(struct tasklet_struct *t); /* деактивация */
```

```
tasklet_disable(struct tasklet_struct *t); /* с ожиданием завершения работы tasklet'a */
```

```
tasklet_enable(struct tasklet_struct *t); /* активация */
```

Если tasklet деактивирован, его по-прежнему можно добавить в очередь на планирование, но исполняться на процессоре он не будет до тех пор, пока не будет вновь активирован. Причем, если tasklet был деактивирован несколько раз, то он должен быть ровно столько же раз активирован, поле count в структуре как раз для этого.

tasklet_trylock() выставляет tasklet'у состояние TASKLET_STATE_RUN и тем самым блокирует tasklet, что предотвращает исполнение одного и того же tasklet'a на разных CPU.

tasklet_kill (struct tasklet_struct *t) — ждет завершения тасклета и удаляет тасклет из очереди на выполнение только в контексте процесса.

tasklet_kill_immediate (struct tasklet_struct *t, unsigned int cpu) — удаляет тасклет в любом случае.

Причем, убит он будет только после того, как tasklet исполнится, если он уже запланирован.

Билет 2(2019)

Управление внешними устройствами: подключение внешних устройств и их идентификация в системе. Система прерываний: типы прерываний и их особенности, прерывания в последовательности ввода-вывода -обслуживание процесса на ввод-вывод. Быстрые и медленные прерывания .Обработчики аппаратных прерываний: регистрация в системе, примеры. Очереди работ - объявление, создание, постановка работы в очередь, планирование(пример лаб. раб)

Отмеченное красным написано в билете 1, смотреть там. Тут только про очереди работ!

Очереди работ

Главное отличие очередей работ от тасклетов является то, что очереди работ могут блокироваться, а тасклеты нет.

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.

Флаги

- **WQ_UNBOUND:** По наличию этого флага очереди делятся на привязанные и непривязанные. В привязанных очередях work'и при добавлении привязываются к текущему CPU, то есть в таких очередях work'и исполняются на том ядре, которое его планирует (на котором выполнялся обработчик прерывания). В этом плане привязанные очереди напоминают tasklet'ы. В непривязанных очередях work'и могут исполняться на любом ядре. Рабочие очереди были разработаны для запуска задач на определенном процессоре в расчете на улучшение поведения кэша памяти. Этот флаг отключает это поведение, позволяя отправлять заданные рабочие очереди на любой процессор в системе. Флаг предназначен для ситуаций, когда задачи могут выполняться в течение длительного времени, причем так долго, что лучше разрешить планировщику управлять своим местоположением. В настоящее время единственным пользователем является код обработки объектов в подсистеме FS-Cache.
- **WQ_FREEZEABLE:** работа будет заморожена, когда система будет приостановлена. Очевидно, что рабочие задания, которые могут запускать задачи как часть процесса приостановки / возобновления, не должны устанавливать этот флаг.

- **WQ_RESCUER**: код workqueue отвечает за гарантированное наличие потока для запуска worker'a в очереди. Он используется, например, в коде драйвера ATA, который всегда должен иметь возможность запускать свои процедуры завершения ввода-вывода.
- **WQ_HIGHPRI**: задания, представленные в такой workqueue, будут поставлены в начало очереди и будут выполняться (почти) немедленно. В отличие от обычных задач, высокоприоритетные задачи не ждут появления ЦП; они будут запущены сразу. Это означает, что несколько задач, отправляемых в очередь с высоким приоритетом, могут конкурировать друг с другом за процессор.
- **WQ_CPU_INTENSIVE**: имеет смысл только для привязанных очередей. Этот флаг — отказ от участия в дополнительной организации параллельного исполнения. Задачи в такой workqueue могут использовать много процессорного времени. Интенсивно использующие процессорное время worker'ы будут задерживаться.

Может использоваться вызов create_workqueue()

ПОЛНЫЙ КОД СМОТРЕТЬ В БИЛЕТЕ 18

Например:

```
static simp_synthesizer_dev_t synth =
{
    .keyboard_irq = 1,
    .name = "synthesizer",
    .wq = NULL
};
static int __init synthesizer_init(void)
{
    printk(KERN_INFO "Init synth.");
    // регистрация обработчика прерывания
    int res = request_irq(synth.keyboard_irq, irq_handler, IRQF_SHARED, synth.name, &synth);
    if (res == 0)
    {
        printk(KERN_INFO "Keyboard irq handler was registered successfully.");
        // создание workqueue
        synth.wq = alloc_workqueue("sound_player", WQ_UNBOUND, 0);
        if (synth.wq)
            printk(KERN_INFO "Workqueue was allocated successfully");
    }
    else
    {
        free_irq(synth.keyboard_irq, &synth);
```

```
    printk(KERN_ERR "Workqueue allocation failed");
    return -ENOMEM;
}
}
```

Очередь работ (workqueue) описывается структурой:

```

long insert_sequence; /* следующий элемент для добавления */
struct list_head worklist; /* список действий */
wait_queue_head_t more_work;
wait_queue_head_t work_done;
struct workqueue_struct *wq; /* соответствующая структура
                           workqueue_struct */
task_t *thread; /* соответствующий поток */
int run_depth; /* глубина рекурсии функции run_workqueue() */
};

};


```

Заметим, что каждый *тип* рабочих потоков имеет одну, связанную с этим типом структуру `workqueue_struct`. Внутри этой структуры имеется по одному экземпляру структуры `cpu_workqueue_struct` для каждого рабочего потока и, следовательно, для каждого процессора в системе, так как существует только один рабочий поток каждого типа на каждом процессоре.

work item (или просто `work`) — это структура, описывающая функцию (например, обработчик нижней половины), которую надо запланировать. Её можно воспринимать как аналог структуры `tasklet`.

Для того, чтобы поместить задачу в очередь работ надо заполнить (инициализировать) структуру:

```

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


```

`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. Вы можете сделать это 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.

```
int queue_work( struct workqueue_struct *wq, struct work_struct *work );
```

```
int queue_work_on( int cpu, struct workqueue_struct *wq, struct work_struct *work );  
Две дополнительные функции обеспечивают те же функции для отложенной работы (в которой  
инкапсулирована структура work_struct и таймер, определяющий задержку).
```

```
int queue_delayed_work( struct workqueue_struct *wq,  
                      struct delayed_work *dwork, unsigned long delay );
```

```
int queue_delayed_work_on( int cpu, struct workqueue_struct *wq,  
                      struct delayed_work *dwork, unsigned long delay );
```

Кроме того, можно использовать глобальное ядро - глобальную очередь работ с четырьмя функциями, которые работают с этой очередью работ. Эти функции имитируют предыдущие функции, за исключением лишь того, что вам не нужно определять структуру очереди работ.

```
int schedule_work( struct work_struct *work );  
int schedule_work_on( int cpu, struct work_struct *work );
```

```
int scheduled_delayed_work( struct delayed_work *dwork, unsigned long delay );  
int scheduled_delayed_work_on(  
    int cpu, struct delayed_work *dwork, unsigned long delay );
```

Есть также целый ряд вспомогательных функций, которые можно использовать, чтобы принудительно завершить (flush) или отменить работу из очереди работ. Для того, чтобы принудительно завершить конкретный элемент work и блокировать прочую обработку прежде, чем работа будет закончена, вы можете использовать функцию flush_work. Все работы в данной очереди работ могут быть принудительно завершены с помощью функции flush_workqueue. В обоих случаях вызывающий блок блокируется до тех пор, пока операция не будет завершена. Для того, чтобы принудительно завершить глобальную очередь работ ядра, вызовите функцию flush_scheduled_work.

```
int flush_work( struct work_struct *work );  
int flush_workqueue( struct workqueue_struct *wq );  
void flush_scheduled_work( void );
```

Можно отменить работу, если она еще не выполнена обработчиком. Обращение к функции cancel_work_sync завершит работу в очереди, либо возникнет блокировка до тех пор, пока не будет завершен обратный вызов (если работа уже выполняется обработчиком). Если работа отложена, вы можете использовать вызов функции cancel_delayed_work_sync.

```
int cancel_work_sync( struct work_struct *work );  
int cancel_delayed_work_sync( struct delayed_work *dwork );
```

Наконец, можно выяснить приостановлен ли элемент work (еще не обработан обработчиком) с помощью обращения к функции work_pending или delayed_work_pending.

```
work_pending( work );  
delayed_work_pending( work );
```

Билет 3

Система прерываний: типы прерываний и их особенности. Быстрые и медленные прерывания. Обработчики прерываний: деление на верхнюю и нижнюю половины; обработчики аппаратных прерываний-регистрация в системе, разделение линии IRQ и отложенные действия: softirq, тасклеты, очереди работ - особенности, сравнение, примеры (лаб.раб.)

Классификация прерываний, типы и их особенности написаны в билете 1

Драйвер может иметь один обработчик прерывания. Если устройство использует прерывания, то драйвер устройства регистрирует один обработчик прерывания. Для обработки прерывания драйвер может разрешить определенную линию прерывания IRQ.

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

- Первый параметр `irq` — номер прерывания, который будет обрабатывать обработчик. Для некоторых устройств это величина, которая обычно устанавливается аппаратно. Для большинства других устройств определяется программно и динамически.
- Второй параметр `handler` — указатель на функцию обработчика прерывания (актуального обработчика прерывания, кот. обслуживает прерывания).

Обработчик прерывания получает три параметра. Типичное объявление обработчика:

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

- `irq` - численное значение линии прерывания, которую обслуживает обработчик.
 - `dev_id` - общий указатель на тот же самый `dev_id`, что задается в `request_irq`, когда обработчик был зарегистрирован. `dev_id` может так же указывать на структуру, которая может быть так же использоваться обработчиком прерывания. Т.к. структура `struct device` уникальна для каждого устройства и может потенциально использоваться обработчиком, то она обычно передаётся как `dev_id`.
 - `struct pt_regs *regs` — содержит указатель на структуру, содержащие регистры процессора и состояния перед обслуживанием прерывания. Этот параметр используется редко — обычно для отладки. Возвращаемое значение: тип `irq_t`. Функция может вернуть определенных значения `IRQ_NONE` или `IRQ_HANDLED`. Первое значение возвращается при обнаружении прерывания. Второе значение возвращается, когда обработчик прерывания был вызван корректно и его устройство действительно сформировало прерывание.
 - `irqflags` — могут быть либо нулём, либо битовой маской одного или более флагов `#define IRQF_SHARED` — очень важный флаг, который указывает что линия прерывания разделяется на несколько обработчиков прерывания

Быстрые прерывания

Они очень важны в системе, потому что быстрые прерывания выполняются на высоком уровне приоритета и прервать выполнение такого прерывания нельзя. В SMP архитектурах на всех процессорах запрещается данное прерывание. Таких

прерываний в системе не может быть много. **Самое быстрое** прерывание — прерывание **от системного таймера**.

- **IRQF_SHARED** — данный флаг разрешает разделение или совместное использование линии IRQ. С помощью него мы можем на этот флаг повесить собственное прерывание и увидеть наш обработчик прерывания в системе.
- **IRQF_PROBE_SHARED** — устанавливается, если предполагается возможность возникновения проблем при совместном использовании линии IRQ.
- **IRQF_PERCPU** — указывает, что прерывание закреплено за определенным процессором.
- **IRQF_NOBALANCING** — запрещает использовать данное прерывание для балансировки IRQ.
- **DEV_NAME** — ASCII текст, представляющий устройство, которое связано с этим прерыванием. Текст имени или строка имени используется в /proc/irq и в /proc/interrupts, что удобно для пользователя
- **DEV_ID** — используется прежде всего для разделения линии прерывания.

Когда обработчик прерывания освобождается, **DEV_ID** обеспечивает уникальные **cookie** (прим: **думаю это слово писать не стоит**) файлы, чтобы выполнить удаление только данного обработчика прерывания с линии прерывания. **DEV_ID** — указатель типа void, т.е. может указывать на что угодно. Но обычно используется **указатель на структуру специфичную для устройства**. В случае успеха, функция `request_irq` возвращает 0. Ненулевая величина — ошибка. Обычно ошибка **E_BUSY**, которая обозначает, что данная линия прерывания уже используется и либо текущий пользователь, либо вы не указали **IRQF_SHARED**.

Медленные прерывания. Верхние и нижние половины.

Обработчик прерывания делится на две части:

- одна часть (верхняя) остаётся обработчиком прерывания,
- другая (нижняя) выполнение на более низком приоритете в более простых условиях

Задачи верхней половины:

1. т.к. обработчик верхней половины выполняется при запрещенных прерываниях, то обработчик возвращает управление обычным `return`.
2. перед своим завершением она должна обеспечить последующее выполнение нижней половины, которая завершит работу начатую верхней половиной. Тем или иным способом верхняя половина **должна поставить в очередь на выполнение нижнюю половину**.

Существует три типа нижних половин:

1. Отложенные прерывания (SOFT IRQ)

В файле `<linux/interrupt.h>` определена структура `struct softirq_action`. В настоящее время в этой структуре только одна строка, определяющая функцию, которая должна

выполняться. Есть ещё одно поле — поле данных. В файле `<kernel/softirq.c>` определен массив из 32 экземпляров `softirq_action`.

`static struct softirq_action softirq_vec [NP_SOFTIRQS]`, где `NP_SOFTIRQS` задействованное число номеров, т.е. можно создать 32 обработчика `SOFTIRQS`. В настоящее время определено 10.

Посмотреть работающий `softirq` в системе можно следующим образом: `cat/proc/softirqs`.

Особенности

- a. Определяются статически при компиляции ядра
- b. Добавить новый уровень обработчика `softirq` можно перекомпилировав ядро.
- c. Максимальное количество не может быть изменено динамически.

Прим: мне кажется про создание не нужно, но оставлю на всякий случай

Для **создания** нового уровня `softirq` нужно:

- 1) определить новый индекс отложенного прерывания, вписав его константу в перечисление;
- 2) во время инициализации модуля должен быть зарегистрирован обработчик прерывания с помощью вызова `open_softirq`. индекс `softirq`, функцию обработчик и значение поле `data`.
Должна соотв. правильному прототипу;
- 3) зарегистрированная `softirq` должна быть поставлена в очередь на выполнение — как говорят должна быть отмечена — используется слово `raise`. Для этого должна быть вызвана функция `raise_softirq` — генерация отложенного прерывания. Обычно обработчик верхней половины перед возвратом должен инициализировать выполнение нижней половины;
- 4) как говорится, в некоторый подходящий для системы момент времени отложенное прерывание выполняется при разрешенных аппаратных прерываниях на данном процессоре, но `softirq` запрещены. Если обработчики `softirq` могут выполняться параллельно, то функция должна быть реентерабельной и, если обработчик обращается к разделяемым данным, значениям, переменным, то необходимо обеспечить монопольный доступ обработчика к разделяемым ресурсам. Это одна из основных проблем, связанная с отложенными прерываниями. `ksoftirqd` — демон `softirq`. В любом коде ядра, в котором явно проверяются и запускаются ожидающие обработчики отложенных прерываний. Функция `do_softirq`, которая в цикле проверяет наличие отложенных прерываний. **softirq никогда не вытесняет другой softirq. Единственное событие, которое может вытеснить softirq — это аппаратное прерывание.** Демон `softirq` — это поток ядра каждого процессора `PER_CPU`, который выполняется, когда в машине запущены отложенные прерывания. Компьютер обменивается данными с устройствами, когда возникает прерывание от устройства, ОС прерывает выполнение текущей задачи и т.д..

В рез-те ОС не может закончить обслуживание одного прерывания до прихода другого прерывания. Такое может быть, например, когда сетевая карта с высокой скоростью получает пакеты в течение короткого промежутка времени. Поскольку ОС не может справиться, создаётся очередь, которой должен управлять демон `ksoftirqd`. Если демон `softirq` занимает большую часть процессорного времени, то это означает, что машина находится под большой нагрузкой.

2. Тасклеты

Тасклеты базируются на softirq. Но тасклеты — это отложенные прерывания, для которых обработчик не может выполняться на нескольких процессорах. Тасклеты надо понимать как простые в использовании отложенные прерывания. Разные тасклеты могут выполняться параллельно. **Тасклеты одного типа не могут выполняться параллельно.** Тасклеты — хороший компромисс между производительностью и простотой. *Прим: тут надо нарисовать структуру тасклета, смотрите файлик.* Тасклет **может быть создан как статический, так и динамический.** Статически тасклеты задаются через макросы. Объявление тасклета с указанием имени и функции и аргументом данных `DECLARE_TASKLET(name, func, data)` или `DECLARE_TASKLET_DISABLED(name, func, data)`. Первый макрос создаёт тасклет с `count == 0`, т.е. этот тасклет разрешен. А второй наоборот.

`TASKLET_STATE_SCHED` // тасклет запланирован
`TASKLET_STATE_RUN` // тасклет выполняется **Подробнее про тасклеты в 1-ом билете**
или **можно** **почитать** **тут**
https://www.ibm.com/developerworks/ru/library/l-linux_kernel_60/

3. Очереди работ

Прим: тут должны быть структуры `work_struct`, `workqueue_struct`, `cpu_workqueue_struct`

Очереди работ должны быть явно созданы до использования для создания очередей работ используется функция

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

- `char *name` - имя очереди `unsigned`
- `int flags` - определяют особенности выполнения
- `int max_active` - количество задач которые могут одновременно выполняться на сри

Подробнее про очереди работ во 2-ом билете или можно почитать тут
https://www.ibm.com/developerworks/ru/library/l-linux_kernel_60/

Отличие от тасклетов:

1. тасклеты выполняются в контексте прерывания и в результате код тасклета должен быть атомарным (неделимым); А код очередей работ выполняется в контексте специального потока ядра в результате **очереди работ могут блокироваться.**

Прим: находил немного другую трактовку - (по ссылке выше): "Очереди отложенных действий позволяют откладывать некоторые операции для последующего выполнения отдельным потоком пространства ядра. Такие отложенные действия всегда выполняются в контексте процесса "

2. Тасклеты всегда выполняются на процессорах, на которых они были запланированы очереди работ по умолчанию выполняются точно также
3. Код ядра может запросить, чтобы выполнение функции очереди работы было отложено на неопределенное время, в тасклете так нельзя.
4. Тасклеты выполняются всегда после выполнения аппаратного прерывания т.к. очереди работ выполняются в контексте специального рабочего ядра => **очереди работ не должны быть атомарными**

Примеры

- **Тасклет**

```
#include <linux/module.h> // module_init, ...
#include <linux/kernel.h> // printk
#include <linux/init.h> // __init, __exit
#include <linux/interrupt.h>
#include <linux/timex.h>

MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Pandoral");

#define SHARED_IRQ 17
static int irq = SHARED_IRQ, my_dev_id, irq_counter = 0;
module_param(irq, int, S_IRUGO);

char tasklet_data[] = "tasklet_function was called";

void tasklet_function( unsigned long data );

DECLARE_TASKLET( my_tasklet, tasklet_function,
                 (unsigned long)&tasklet_data );

/* Bottom Half Function */
void tasklet_function( unsigned long data ) {
    printk(KERN_INFO "[TASKLET] state: %ld, count: %d, data: %s",
           my_tasklet.state, my_tasklet.count, my_tasklet.data);
    return;
}
```

```
static irqreturn_t my_interrupt( int irq, void *dev_id ) {
    irq_counter++;
    printk( KERN_INFO "[INTERRUPT] In the ISR: counter = %d\n", irq_counter );
    tasklet_schedule( &my_tasklet );
    return IRQ_NONE; /* we return IRQ_NONE because we are just observing */
}

static int __init my_tasklet_init(void) {
    if( request_irq( irq, my_interrupt, IRQF_SHARED, "my_interrupt", &my_dev_id ) )
        return -1;
    printk( KERN_INFO "[INTERRUPT] Successfully loading ISR handler on IRQ %d\n", irq );
    printk(KERN_INFO "[MODULE] Module is now loaded.\n");
    return 0;
}

static void __exit my_tasklet_exit(void) {
    /* Stop the tasklet before we exit */
    tasklet_kill( &my_tasklet );
    synchronize_irq( irq );
    free_irq( irq, &my_dev_id );
    printk( KERN_INFO "[INTERRUPT] Successfully unloading, irq_counter = %d\n", irq_counter );
    printk(KERN_INFO "[MODULE] Module is now unloaded.\n");
    return;
}

module_init(my_tasklet_init);
module_exit(my_tasklet_exit);
```

- **Очереди работ**

```

#include <linux/module.h> // module_init, ...
#include <linux/kernel.h> // printk
#include <linux/init.h> // __init, __exit
#include <linux/interrupt.h>
#include <linux/workqueue.h>

MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("MY");

#define SHARED_IRQ 17
static int irq = SHARED_IRQ, my_dev_id, irq_counter = 0;
module_param(irq, int, S_IRUGO);

struct workqueue_struct *wq;
void hardwork_function(struct work_struct *work);

DECLARE_WORK(hardwork, hardwork_function);

/* Bottom Half Function */
void hardwork_function(struct work_struct *work) {
    printk(KERN_INFO "[WQ] data: %d", work->data);
    return;
}

static irqreturn_t my_interrupt( int irq, void *dev_id ) {
    irq_counter++;
    printk( KERN_INFO "[INTERRUPT] In the ISR: counter = %d\n", irq_counter );
    queue_work( wq, &hardwork );
    return IRQ_NONE; /* we return IRQ_NONE because we are just observing */
}

static int __init my_wokqueue_init(void) {
    if( request_irq( irq, my_interrupt, IRQF_SHARED, "my_interrupt", &my_dev_id ) )
        return -1;
    printk( KERN_INFO "[INTERRUPT] Successfully loading ISR handler on IRQ %d\n", irq );
    wq = create_workqueue( "my_queue" );
    if ( wq ) {
        printk(KERN_INFO "[MODULE] Workqueue created.\n");
    }

    printk(KERN_INFO "[MODULE] Module is now loaded.\n");
    return 0;
}

static void __exit my_wokqueue_exit(void) {
    flush_workqueue( wq );
    destroy_workqueue( wq );
    synchronize_irq( irq );
    free_irq( irq, &my_dev_id );
    printk( KERN_INFO "[INTERRUPT] Successfully unloading, irq_counter = %d\n", irq_counter );
    printk( KERN_INFO "[MODULE] Module is now unloaded.\n");
    return;
}

module_init(my_wokqueue_init);
module_exit(my_wokqueue_exit);

```

Билет 4

Файловая подсистема /proc назначение, особенности, файлы, поддиректории, ссылка self, interrupt. 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. Пример передачи данных из пространства пользователя в пространство ядра и обратно, вывод информации о выполняемых в системе процессах (лаб.раб.).

Файловая подсистема /proc:

Ф.С. proc не является монтированной. Каталог proc имеет inode равный 1. ВФС создается "на лету" - в процессе обращения к системе. При обращении к информации в файловой системе (ФС), ФС предоставляет соответствующую информацию в режиме пользователя. ФС /proc создавалась для предоставления информации о файлах процессам и пользователям.

Поскольку это ФС работа в ней выполняется с файлами, следовательно, используется стандартный интерфейс ФС и системных вызовов, следовательно, работа с информацией ФС proc осуществляется с помощью прав доступа, обычных для файлов - read, write, execute. По умолчанию r/w разрешены для владельцев. В proc для каждого процесса существует поддиректория, которая имеет имя - pid процесса: /proc/. Как процесс может получить свой id: getpid(), getppid() - id предка. Информация в ФС proc по id процесса это:

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

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

* образ процесса - это код программы. ** - до запуска программа это файл, и он принадлежит ФС, следовательно, процесс существует в этой ФС.

В системе есть просто файлы, а есть открытые обычные файлы - регулярные (regular). Обычные файлы хранятся во внешней памяти (энергонезависимой памяти - для долговременного хранения)

Кроме /proc/ к информации процесса можно обратиться через [proc/self](#).

Файлы в proc являются **виртуальными**, не существуют ни на каком носителе, генерируются при каждом обращении. **Обращение - чтение информации**. Любой файл в proc - текстовый. proc предоставляет информацию не только о процессах, но и о ресурсах системы. Например, мы можем получить информацию о текущем CPU (proc/cpufreq), об установленных в системе драйверах устройств и номерах устройств, об источниках прерываний и о частоте возникновения этих прерываний ([proc/interrupts](#)), об устройствах, подключенных к шине pci (proc/pci), о состоянии батареи ноутбука (proc/arp).

Linux и многие разработчики системного ядра негативно относятся к системному вызову `ioctl()`, не без оснований. Это неконтролируемый способ добавления нестандартных интерфейсов в ядро. Это можно заменить созданием ВФС. Структура `proc_dir_entry`. Функции и структуры ядра не регламентированы, их можно переписывать

```
{  
...  
typedef int (read_proc_t) (char *page,  
                           char **start,  
                           off_t off,  
                           int count,  
                           int *coef,  
                           void *data);  
typedef int (write_proc_t) (struct file *file,  
                           const char __user *buffer,  
                           unsigned long count,  
                           void *data);  
}
```

Используется структура `proc_dir_entry` (прим: тут нужен рисунок структуры)

Комментарии к структуре:

- **low_ino** - номер индекса `inode` (структуре, описывающей физический файл). Физические файлы (регулярные) находятся на (диске) во вторичной памяти. `inode` - там же (дисковый). У них есть номера - это метаданные. К `inode` обращаются по их номерам.
- **namelen** - длина имени файла.
- **name** - имя файла
- **mode** - (см функцию `stat`) права доступа, информация о типе (7 типов файлов). Как правило, временные файлы - буфера. Но с ними работают как с файлами, например, именнованный программный канал (буфер в системной области памяти).
- **nlink** - число жестких ссылок (еще одно имя файла). Нельзя удалить файл, на который еще есть жесткие ссылки.
- **uid, gid** - обертка типа - `unsigned long`.
- **proc_iops, proc_fops** - операции, определенные на `inode`, это функции, которые объявлены, описаны и в их задачу входит выполняются действия на `inode`. Стандартные функции (аналогично с файлами) `fops` - `file_operations`. `read`, `write`, `release`. `r/w` - в системе производится работа с любыми файлами.
- **read_proc, write_proc** - они то включены в `proc_dir_entry`, то выключаются из нее. Есть уже `fops` в ней определены функции чтения и записи файлов, следовательно `write_proc/read_proc` для работы с файлами не нужны.

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

Определены также функции:

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

Ядреный буфер не доступен никому кроме функций. Можно использовать тетсру. 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.

Чтобы передать информацию из режима пользователя в режим ядра и обратно используются функции: **copy_from_user()** и **copy_to_user()**. Библиотека linux/uaccess.h.

Необходимо переписать стандартные функции read, write. Система предоставляет такую возможность.

```
static struct file_operations fops =  
{  
    .open = my_proc_open,  
    .read = my_proc_read,  
    .write = my_proc_write,  
    .release = my_proc_release  
};
```

Функции my_proc_open, my_proc_read, my_proc_write, my_proc_release описываются до структуры.

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/string.h>
#include <linux/vmalloc.h>
#include <linux/uaccess.h>
#include <linux/sched.h>
#include <linux/init_task.h>
#define FORTUNE_BUF_SIZE PAGE_SIZE
MODULE_LICENSE("GPL");

ssize_t fortune_read(struct file* file, char* buf, size_t count, loff_t* f_pos);
ssize_t fortune_write(struct file* file, const char* buf, size_t count,
                      loff_t* f_pos);
int fortune_init(void);
void fortune_exit(void);
struct file_operations fops;
char *fortune_buf, *process_buffer;
struct proc_dir_entry* proc_entry;

unsigned int read_index;
unsigned int write_index;

// Чтение предсказаний cat fortune
ssize_t fortune_read(struct file* file, char* buf, size_t count, loff_t* f_pos)
{
    int len;
    if (write_index == 0 || *f_pos > 0) {
        17 return 0;
    }
    if (read_index >= write_index) {
        read_index = 0;
    }
    copy_to_user(buf, &fortune_buf[read_index], count);
    len = strlen(&fortune_buf[read_index]) + 1;
    read_index += len;
    *f_pos += len;
    return len;
}
```

```
// Запись предсказания echo "message" >> fortune
ssize_t fortune_write(struct file* file, const char* buf, size_t count, loff_t* f_pos)
{
    int free_space = (FORTUNE_BUF_SIZE - write_index) + 1;
    if (count > free_space) {
        printk(KERN_INFO "fortune pot full.\n");
        return -ENOSPC;
    }
    if (copy_from_user(&fortune_buf[write_index], buf, count)) {
        return -EFAULT;
    }
    write_index += count;
    fortune_buf[write_index - 1] = 0;
    return count;
}
```

```

int fortune_init(void)
{
    fortune_buf = vmalloc(FORTUNE_BUF_SIZE); // выделение памяти
    if (!fortune_buf) {
        printk(KERN_INFO "Not enough memory for the fortune pot.\n");
        return -ENOMEM;
    }

    // Переопределение операций чтения и записи
    fops.owner = THIS_MODULE;
    fops.read = fortune_read;
    fops.write = fortune_write;
    process_buffer = vmalloc(2 * FORTUNE_BUF_SIZE); // здесь пометка внизу
    if (!process_buffer) {
        vfree(buffer);
        printk(KERN_INFO "Can't vmalloc process_buffer\n");
        return -ENOMEM;
    }
    memset(fortune_buf, 0, 2 * FORTUNE_BUF_SIZE);
    proc_entry = proc_create_data("fortune", 0666, NULL, &fops, NULL);
    if (!proc_entry) {
        vfree(fortune_buf);
        printk(KERN_INFO "Cannot create fortune file.\n");
        return -ENOMEM;
    }
    read_index = 0;
    write_index = 0;
    proc_mkdir("my_dir_fortune", NULL);
    proc_symlink("symbolic_link_fortune", NULL, "/proc/fortune");
    printk(KERN_INFO "Fortune module loaded.\n");
    return 0;
}

void fortune_exit(void)
{
    remove_proc_entry("fortune", NULL);
    if (fortune_buf) {
        vfree(fortune_buf);
    }
    printk(KERN_INFO "Fortune module unloaded.\n");
}
module_init(fortune_init);
module_exit(fortune_exit);

```

Пометка: дополнительным заданием было вывести все процессы с помощью [task_struct](#).

```

struct task_struct *task = &init_task; // указатель на структуру процесса init
do {
    sprintf(process_buffer, "%s process:name=%s, pid=%d; parent:name=%s, pid=%d\n",
            process_buffer, task->comm, task->pid, task->real_parent->comm,
            task->real_parent->pid);
    task = next_task(task); // следующая структура
} while (task != &init_task);
printk(KERN_INFO "Fortune module loaded.\n");

```

Если переопределять операцию `read` для того, чтобы вместо чтения выполнялась запись в буффер информации о процессах, будет переполнение. Поэтому вместо двух страниц, нужно выделять память на 4 страницы. Структура [task_struct](#) описывает запущенный в системе процесс. Создается динамически, кроме процесса `init`, для него создается структура [init_task](#) - статическая структура.

С помощью поля `*next_task, *prev_task` создается **круговой список** (не путать с кольцевым буфером!), замкнутый на `init_task`. Мы можем посмотреть эту структуру и вывести любую информацию о любом процессе. `task_struct` доступна в режиме ядра, мы можем получить из нее информацию

Билет 5

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

Отмеченное красным написано в билете 19, смотреть там

Билет 6

Файловая система, задачи файловой системы и иерархическая организация ФС. Файловая подсистема LINUX: поддержка большого числа файловых систем. VFS: четыре основные структуры файловой системы и связь между ними. Раздел жесткого диска и суперблок, точка монтирования-корневой каталог и inode. Пример (лаб.раб.)

Назначение.

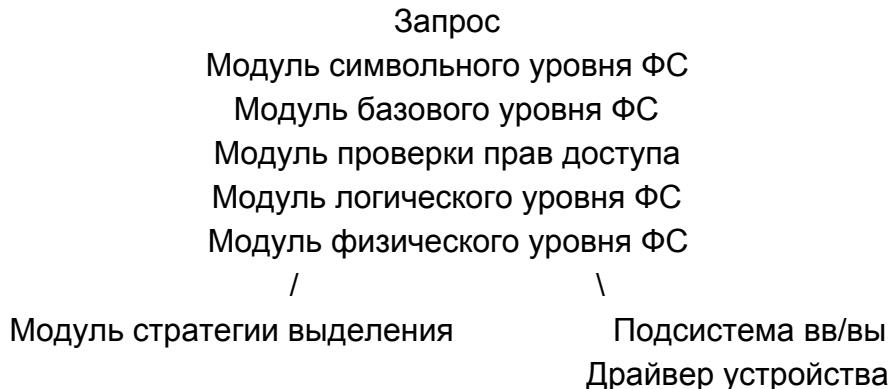
Управление файлами осуществляется частью ОС – файловой подсистемой (file system), она отвечает за возможность длительного хранения информации и доступа к ней.

- Регулярные Файлы - для долговременного хранения, которое обеспечивается внешними устройствами.
- Временные файлы - создаваемые для обслуживания действий в системе в текущий момент времени

Задачи, которые решает ФС:

- 1) обеспечение удобного доступа к файлам как часть задачи именования файлов
- 2) собственно именование файлов – присвоение файлам уникальных идентификаторов, с помощью которых обеспечивается доступ к файлам
- 3) обеспечение программного интерфейса для работы с файлами пользователей и приложений
- 4) отображение логической модели (представления) файлов на физическую организацию хранения данных на соответствующих носителях.
- 5) обеспечение надежного хранения файлов, доступа к ним, обеспечения защиты от несанкционированного доступа.
- 6) обеспечение совместного использования файлов

Иерархическая структура ФС



Символьный уровень

предоставляет возможность систематизации документов.

уровень именования файлов и системат. управление информацией, доступной пользователю.

Базовый уровень

уровень формирования дескриптора файлов (описан в системе).

должны быть соответствующие структуры, позволяющие хранить необходимую для файла информацию

Логический уровень

логическое а.п. файла аналогично логич.а.п. процесса

позволяет обеспечить доступ к данным в файле в формате, отличном от формата физ.хранения

обычно система не накладывает ограничения на внутр. структуру данных в файле

Физический уровень

обеспечение непосредственного доступа к информации на внешнем носителе

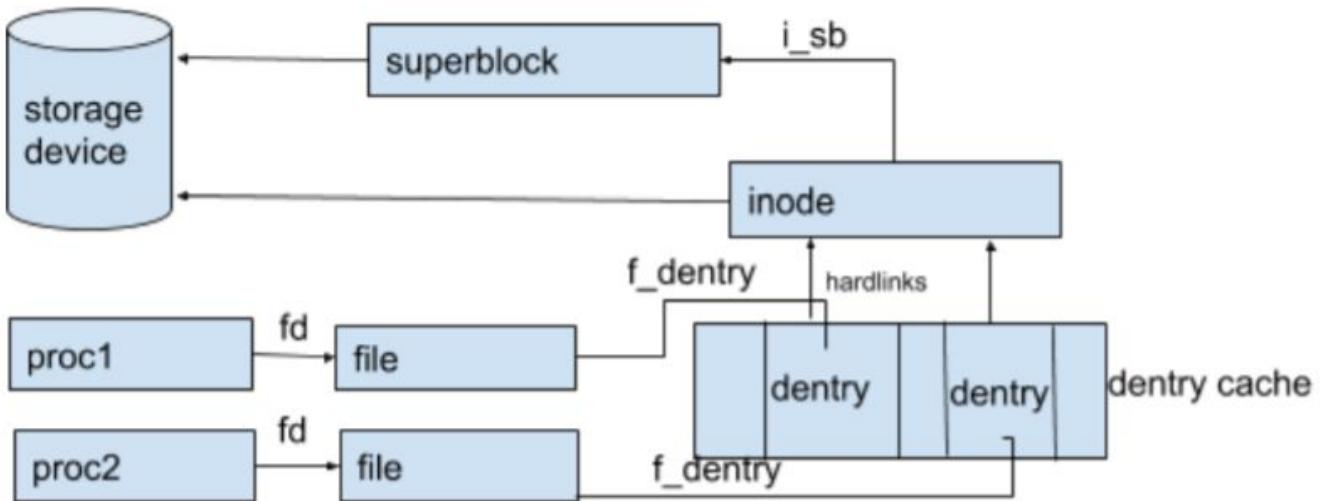
Внешние устройства хранения

UNIX поддерживает большое количество ФС. Одна ФС описывается структурой `file_system_type`. Файловых систем может быть подмонтировано сколько угодно.

VFS (virtual file system) – юниксовый интерфейс, предоставляет общую файловую модель, способную отображать общие возможности и поведение любой мыслимой ФС.

Код ФС скрывает детали конкретной реализации, однако все ФС поддерживают такие понятия как файлы, каталоги, поддерживают такие действия как создание файла удаление файла, переименование файла, открытие, чтение, запись, закрытие и т д. большинство ФС запрограммировано так, что API, которые предоставляют ФС рассматриваются как абстрактный интерфейс, который ожидаем и понятен VFS.

Связь 4 главных структур VFS.



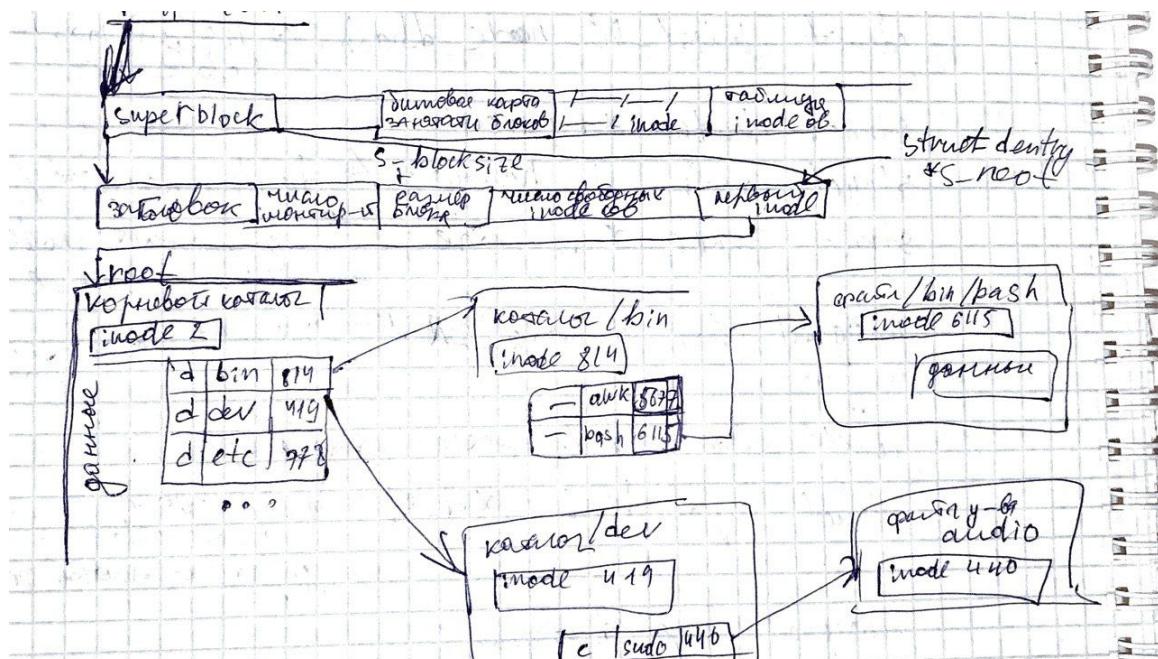
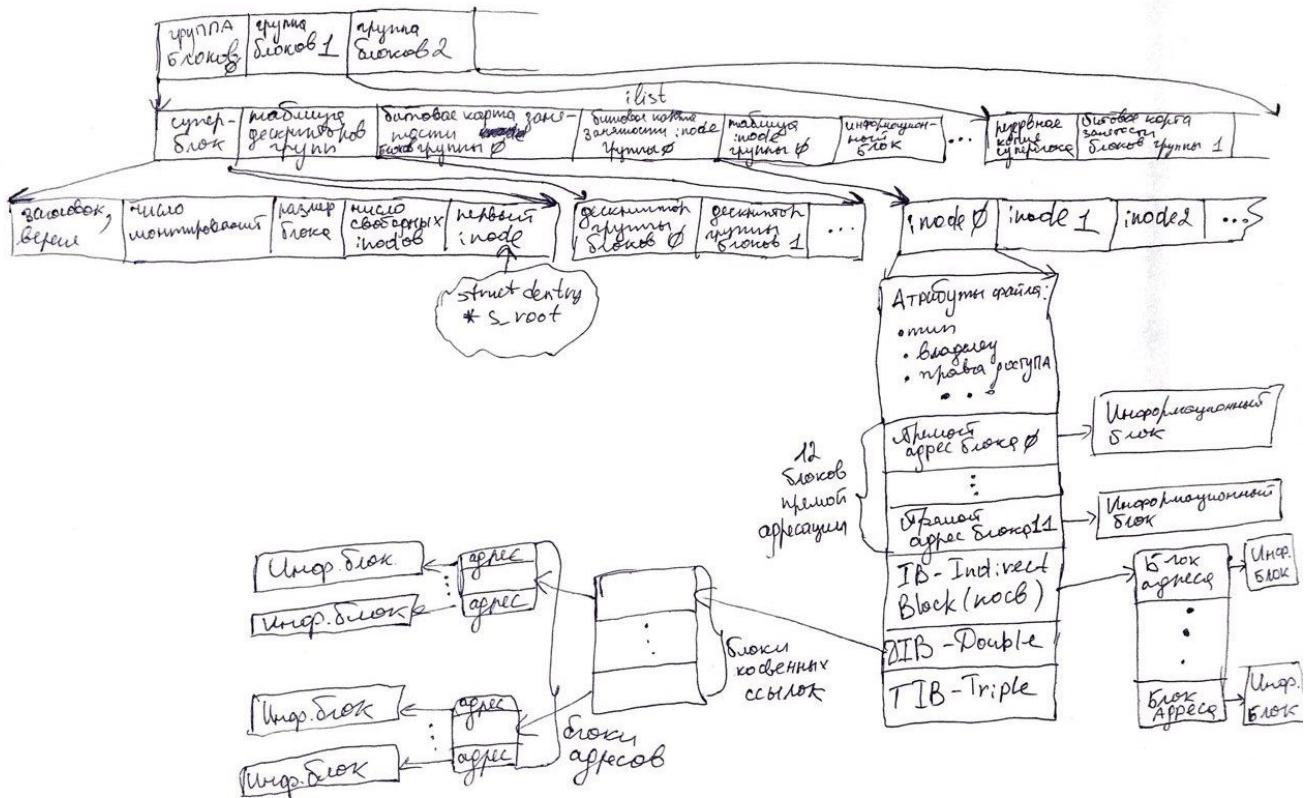
Индексный узел inode - описывает физический файл

inode- **дескрипторы** физических файлов. файлы могут **иметь имена** в виде символьных строк, **имя** файла в их родных файловых подсистемах **не является идентификатором, им является номер inode**.

В **Inode bitmap** хранится инф о том какие inode **заняты**, а какие **свободны**. В **суперблоке** хранится инф о том, **сколько в данной ФС имеется свободных айнодов**.

Система, чтобы получить **доступ к файлу**, ищет **номер inode** в **таблице «таблица inode'ов»**.

Разделы жесткого диска (partition)



Далее нужно написать 4 основные структуры и прототипы функций из лабораторной работы.

В `file_system_type` есть `mount`, которая возвращает указатель на точку монтирования. Основные функции:

```

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

struct dentry *mount_nodev(struct file_system_type *fs_type,
                          int flags, void *data,
                          int (*fill_super)(struct super_block *, void *, int))

```

По просьбе закидываю лабу, где создаем свою ФС.

```

static const unsigned long MYFS_MAGIC_NUMBER = 0x31337;

static void myfs_put_super(struct super_block *sb)
{
    printk(KERN_INFO "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 ,
};

static struct inode *myfs_make_inode(struct super_block *sb, int mode)
{
    struct inode *ret = new_inode(sb);
    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);
    }
    return ret;
}

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 = MYFS_MAGIC_NUMBER;
sb->s_op = &myfs_super_ops;

root = myfs_make_inode(sb, S_IFDIR | 0755);
if (!root)
{
    printk(KERN_ERR "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 "root creation failed!\n");
    return -ENOMEM;
}
return 0;
}
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_INFO "myfs mounted\n");
    return entry;
}

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

```

```

};

static int __init myfs_init(void)
{
    int ret = register_filesystem(&myfs_type);
    if (ret != 0)
    {
        printk(KERN_ERR "MYFS_MODULE cannot register filesystem!\n");
        return ret;
    }
    printk(KERN_INFO "MYFS_MODULE loaded\n");
    return 0;
}

static void __exit myfs_exit(void)
{
    int ret = unregister_filesystem(&myfs_type);

    if (ret != 0)
        printk(KERN_ERR "MYFS_MODULE cannot unregister filesystem!\n");

    printk(KERN_INFO "MYFS_MODULE unloaded\n");
}
module_init(myfs_init);
module_exit(myfs_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("MantiCore");

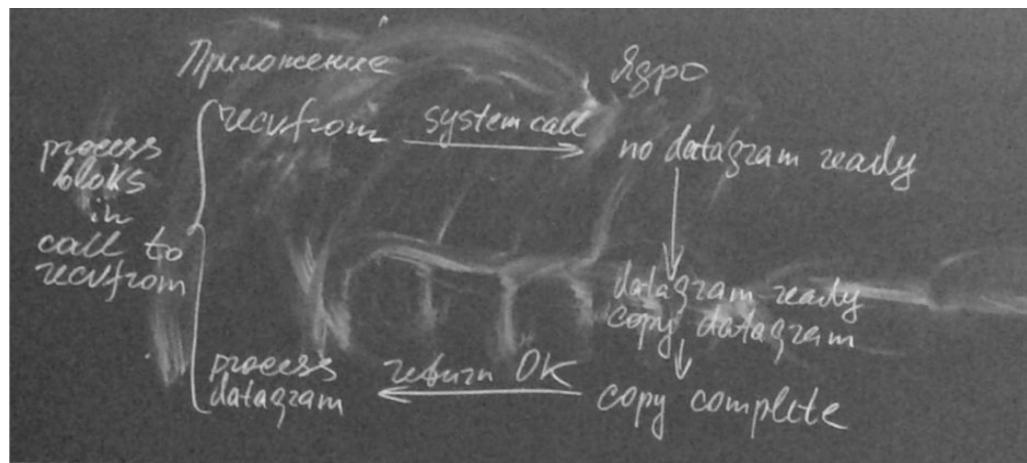
```

Билет 7

Классификация типов ввода-вывода с точки зрения программиста: описание и диаграммы последовательности действий для каждого типа ввода, вывода. Пример мультиплексирования для сокетов AF_INET, SOCK_STREAM. Пример (лаб. раб.)

Рассмотрим модели ввода/вывода с точки зрения программиста:

1. **Блокирующий ввод/вывод - Blocking I/O (подчеркивается, что он асинхронный)**



матрица базовых моделей ввода/вывода

команды(функции read/write) ...

базовый способ работы с обычными файлами

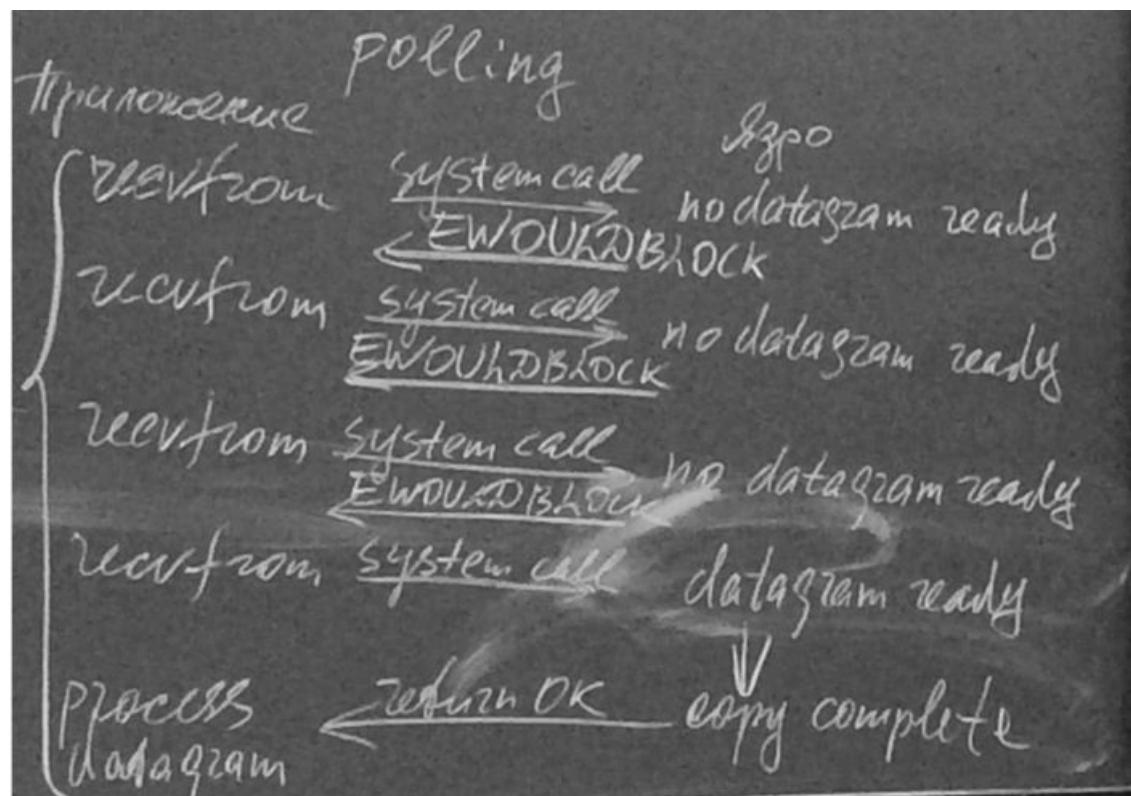
процесс запросивший ввод/вывод блокируется

все время пока данные не будут готовы для того чтобы быть перемещенными в буфер приложения

все это время процесс блокирован

2. Не блокирующий ввод/вывод Polling (опрос) (синхронный) речь идет о готовности данных

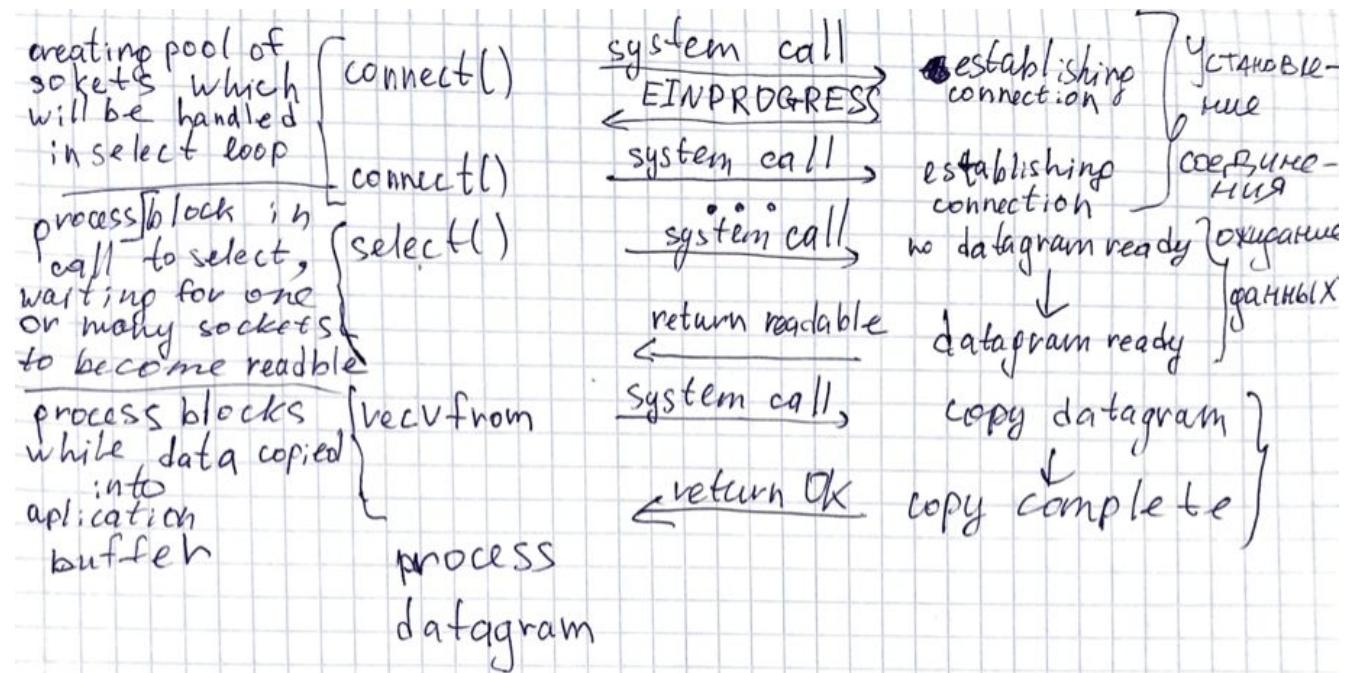
Мы видим из диаграммы, что процесс повторяет вызов, ожидая получения сообщения, что данные получены. Процессор постоянно занят тем, что опрашивает готовность устройства.



При работе с сокетами, может быть, установлен неблокирующий режим. Т.е. `receive_from` будет вызываться в цикле до тех пор, пока ядро не вернет сообщение о том, что данные готовы. Если данные не готовы, то ядро вернет `EWOULDBLOCK`.

3. Мультиплексирование ввода/вывода (ассинхронный, блокирующий)

не смотря на то что процесс блокирован на мультиплекс - это асинхронный ввод/вывод; обработка по мере возникновения соединения



Мультиплексор — устройство, которое объединяет информацию, которая поступает по нескольким каналам входа и выдаёт её по одному выходному каналу

Мультиплексирование — процесс совмещения нескольких сообщений, передаваемы одновременно в одной физической или логической среде. Существует два основных вида мультиплексирования: а) временное; б) частотное.

Временное (Time Division Multiplexing)

Устройству отводятся интервалы времени, в которые оно может использовать передающую среду.

Относится к асинхронному блокирующему.

Multiplexing IO

Для реализации мультиплексирования используется один из мультиплексоров: `select`, `pull`, `pselect`, `epoll`.

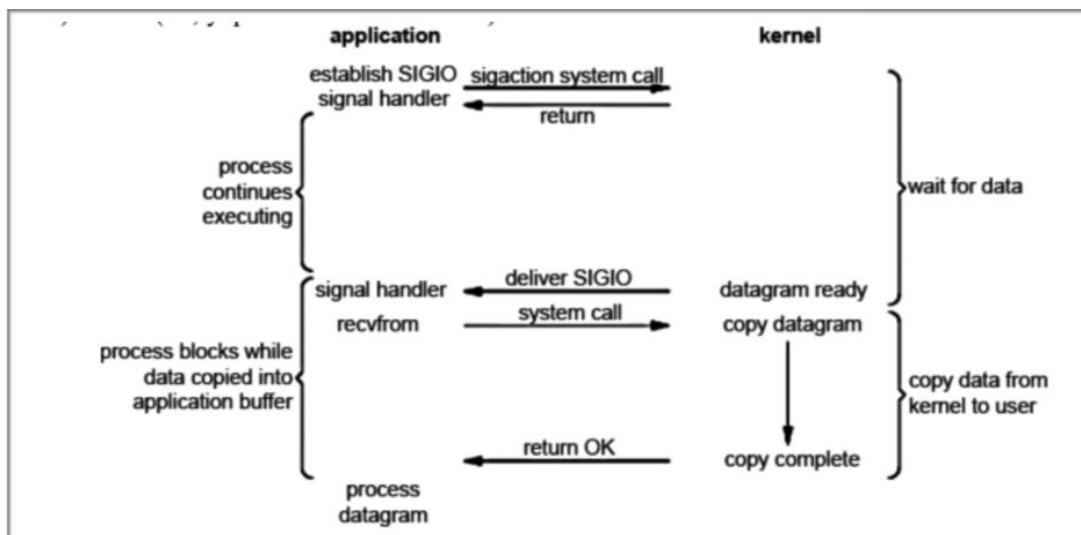
В результате вызова `select` процесс блокируется, ожидая одного из многих возможных сокетов. «Готовность» подразумевает, что на какой-то сокет поступили данные. Мультиплексирование предполагает, что опрашивается много сокетов. Системный

вызов как раз выполняет эти действия. Затем, после того, как поступила информация о готовности сокета для чтения, выполняется `receive_from`, процесс блокируется на время копирования данных из ядра в адресное про-во процесса.

Преимущество мультиплексирования перед блокируемым вводом/ выводом: обрабатывается не один, а сразу много дескрипторов. `select` блокирует процесс, но время блокировки будет в этом случае меньше, т.к. вероятность того, что данные поступят на любой из наборов дескрипторов выше чем вероятность того, что данные поступят на конкретных дескриптор или сокет.

Системный вызов `connect` создает пул сокетов, который будет обрабатываться в цикле `select`. Системный вызов `select` блокирует процесс в ожидании готовности, хотя бы одного сокета. После того, как возвращено сообщение, что данные готовы для чтения, выполняет `recieve_from`, процесс блокируется до того момента, как данные будут скопированы в буфер приложения. При мультиплексировании в цикле проверяются все сокеты и берётся первый готовый. Пока обрабатывается первый сокет, могут подоспеть остальные. Таким образом сокращается время блокировки. Недостатки этого способа: дорогие (требуют большое кол-во накладных расходов) потоки в Linux. Если взять Python, то в нем существует GIL, т.е. в каждом процессе может выполняться только один поток.

4. Ввод/вывод, управляемый сигналами (Signal Driving IO) (асинхронный, неблокирующий)



Как видно из диаграммы, необходимо установить обработчик сигнала. При этом используется SIGIO. Обработчик устанавливается системным вызовом `sig_action`, который входит в POSIX 1. Результат `sig_action` возвращается сразу, приложение не блокируется. Оно может продолжать выполняться. Всё работу на себя берет ядро. Ядро отслеживает, когда данные будут готовы. После чего, посылает сигнал SIGIO.

Который вызывает установленный на него обработчик (call back- функция). Вызов `recvfrom` можно выполнить либо в обработчике сигнала, который сообщит основному циклу, что данные готовы, либо в основном потоке программы.

Ожидание может выполняться в цикле, который выполняется в основном потоке. Сигнал SIGIO может быть для каждого процесса только один. В результате за один раз можно работать с одним файловым дескриптором. Можно отметить, что за время выполнения обработчика сигнала. Данный сигнал блокируется. Если в период блокировки сигнала доставляются несколько раз, то эти сигналы теряются. Если маска сигнала равна Null, то во время выполнения обработчика сигнала, другие сигналы не блокируются.

5. **Асинхронный не блокирующий ввод/вывод** (асинхронный, неблокирующий) – приложение запросив данные продолжает что-то делать имеется ввиду единственный поток

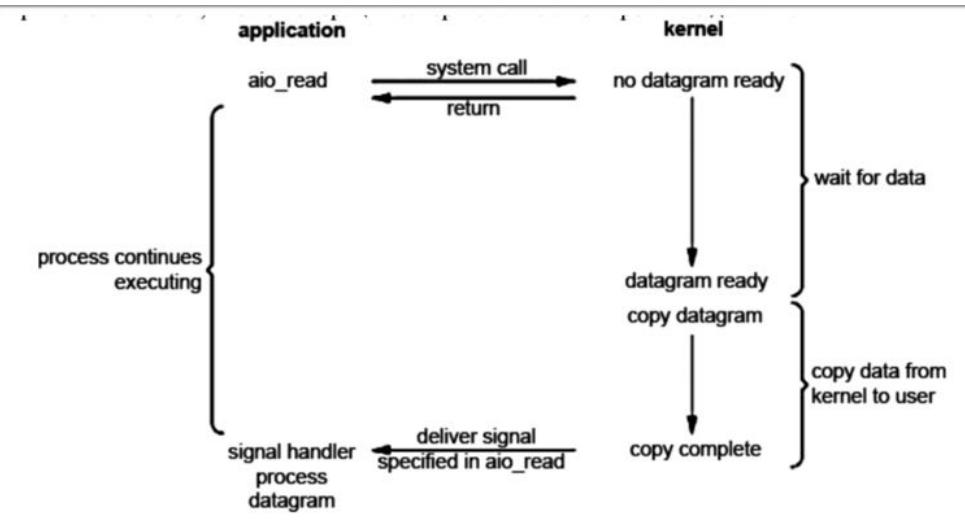
приложение может выполнять другую обработку

в то время как фоновая операция чтения завершается

когда операция чтения возвращает ответ в виде сигнала или в виде call back функции (/ которая может быть реализована в виде потока)

по модели генерируется сообщение что операция выполнена

процесс не блокируется

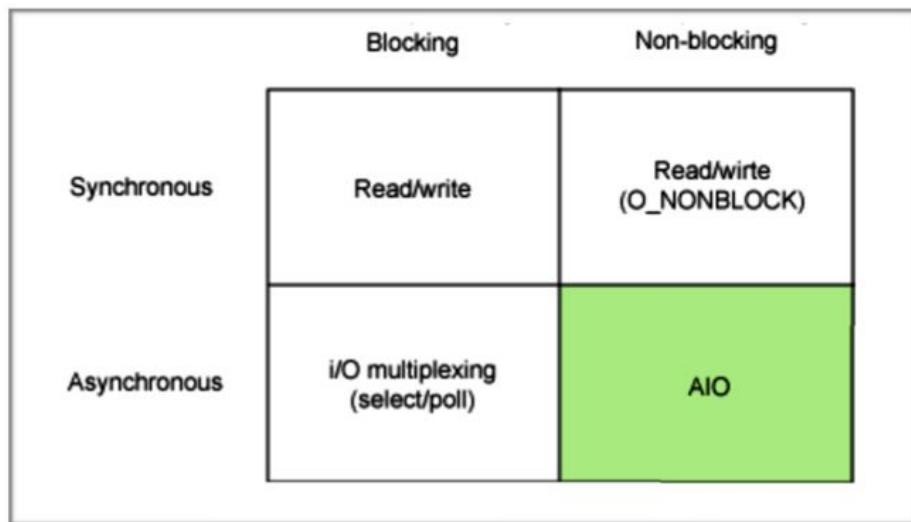


Осуществляется с помощью специальных системных вызовов. Идея: дать ядру команду начать операции ввода/вывода, а затем сообщить процессу или с помощью сигналов (или ещё каким-то способом), что операция ввода/вывода завершена.

Асинхронный ввод/вывод предполагает, что процесс может выполняться. Проблема состоит в том, что необходимо получить асинхронные события асинхронно. POSIX имеют приставку AIO или LIO. Системный вызов `ioread` должен иметь параметры: дескриптор, адрес буфера, размер буфера. Параметры такие же как в системном вызове `read`. Стандарт POSIX (спецификация POSIX) согласовала различия в функциях

реального времени, которые появились в разных стандартах, объединив их. Основное отличие этой модели от модели управляемой сигналом заключается в том, что в модели управляемой сигналом ядро сообщает приложению, когда операция ввода/вывода может быть инициирована. В асинхронном вводе/выводе ядро сообщает приложению, когда операция ввода/вывода завершена.

Из этого сравнения видно, что первая фаза выполняется по разному. Вторая фаза выполняется одинаково. За исключением асинхронного ввода/вывода. Но результат должен быть один — приложение должно получить данные.



Сервер сетевой сокет (Пример мультиплексирования для сокетов AF_INET, SOCK_STREAM)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <arpa/inet.h>
#include <netdb.h>

#include "info.h"

#define MAX_CLIENTS 10
int clients[MAX_CLIENTS] = { 0 };

void manageConnection(unsigned int fd)
```

```

{
    struct sockaddr_in client_addr;
    int addrSize = sizeof(client_addr);

    int incom = accept(fd, (struct sockaddr*) &client_addr, (socklen_t*) &addrSize);
    if (incom < 0)
    {
        perror("Error in accept(): ");
        exit(-1);
    }

    printf("\nNew connection: \nfd = %d \nip = %s:%d\n", incom,
           inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));

    for (int i = 0; i < MAX_CLIENTS; i++)
    {
        if (clients[i] == 0)
        {
            clients[i] = incom;
            printf("Managed as client # %d\n", i);
            break;
        }
    }
}

void manageClient(unsigned int fd, unsigned int client_id)
{
    char msg[MSG_LEN];
    memset(msg, 0, MSG_LEN);

    struct sockaddr_in client_addr;
    int addrSize = sizeof(client_addr);

    int recvSize = recv(fd, msg, MSG_LEN, 0);
    if (recvSize == 0)
    {
        getpeername(fd, (struct sockaddr*) &client_addr, (socklen_t*) &addrSize);
        printf("User %d disconnected %s:%d\n", client_id,
               inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
        close(fd);
        clients[client_id] = 0;
    }
    else
    {
        msg[recvSize] = '\0';
        printf("Message from %d client: %s\n", client_id, msg);
    }
}

```

```
}
```

```
int main(void)
{
    int my_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (my_socket < 0)
    {
        perror("Error in sock()");
        return my_socket;
    }

    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SOCK_PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY; //any address for binding

    if (bind(my_socket, (struct sockaddr*) &server_addr, sizeof(server_addr)) < 0)
    {
        perror("Error in bind()");
        return -1;
    }
    printf("Server is listening on the %d port...\n", SOCK_PORT);

    if (listen(my_socket, 3) < 0)
    {
        perror("Error in listen()");
        return -1;
    }
    printf("Waiting for the connections...\n");

    for (;;)
    {
        fd_set readfds; //set of file descriptors that will monitor select()
        int max_fd; //The largest file descriptor in the set
        int active_clients_count; //number of sockets for which select() has fixed
        activity

        FD_ZERO(&readfds);
        FD_SET(my_socket, &readfds);
        max_fd = my_socket;

        //add to the set of all already connected clients
        for (int i = 0; i < MAX_CLIENTS; i++)
        {
            int fd = clients[i];

            if (fd > 0)
```

```

    {
        FD_SET(fd, &readfds);
    }

    max_fd = (fd > max_fd) ? (fd) : (max_fd);
}

active_clients_count = pselect(max_fd + 1, &readfds, NULL, NULL, NULL,
NULL);

if (active_clients_count < 0 && (errno != EINTR))
{
    perror("Error in select():");
    return active_clients_count;
}

if (FD_ISSET(my_socket, &readfds))
{
    manageConnection(my_socket);
}

// check if there was activity on one of the clients
for (int i = 0; i < MAX_CLIENTS; i++)
{
    int fd = clients[i];
    if ((fd > 0) && FD_ISSET(fd, &readfds))
    {
        manageClient(fd, i);
    }
}

return 0;
}

```

Билет 8

Средства взаимодействия процессов-сокеты Беркли. Типы сокетов. Адресация. Сокеты AF_UNIX. Сетевые сокеты - сетевой стек, аппаратный и сетевой порядок байтов. Примеры реализации взаимодействия процессов по модели клиент-сервер с использованием сокетов (лаб.раб.).

Сокеты были созданы для организации взаимодействия параллельных процессов, причем безразлично, где они работают: на одной машине или на разных. Абстракция сокетов была введена в BSD Unix, поэтому сокеты часто называют сокетами BSD(Berkley Software Distribution). Сокет – это абстракция конечной точки, конечная точка связи или коммуникации. Сокет – файл, обозначается `s`.

На транспортном уровне сокет описывается тремя параметрами:

1. Family;
2. Type;
3. Протокол.

Эти три параметра передаются системному вызову `socket`.

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

Family определяет природу взаимодействия, включая формат адреса. Это так называемый параметр Address Family (AF). Пространство имен.

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

Type:

1. `SOCK_STREAM` - Обеспечивает создание двусторонних, надёжных потоков байтов на основе установления соединения. Может также поддерживаться механизм внепоточных данных.
2. `SOCK_DGRAM` - Поддерживает дейтаграммы (ненадежные сообщения с ограниченной длиной без установки соединения).
3. `SOCK_RAW`(сырой сокет) - Обеспечивает прямой доступ к сетевому протоколу.

Protocol:

- для `SOCK_STREAM` всегда выбирают TCP
- для `SOCK_DGRAM` – UDP
- 0 - протокол будет выбран по умолчанию

Адресация сокетов

Функция `socket()` создает "безымянный" сокет, т.е. не связанный ни с локальным адресом, ни с номером порта. Прежде чем передавать данные через сокет, его необходимо связать с адресом в выбранном домене (этую процедуру называют именованием сокета).

Для явного связывания сокета с некоторым адресом используется функция `bind`. Её прототип имеет вид:

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

В качестве первого параметра передаётся дескриптор сокета, который привязывается к заданному адресу. Второй параметр – **addr**- содержит указатель на структуру `struct sockaddr`, а третий – длину этой структуры.

```
struct sockaddr
{
    unsigned short sa_family; // Семейство адресов, AF_...
    char sa_data[14]; // 14 байтов для хранения адреса
};
```

Структура определяет адрес в самом общем виде. Для сетевого взаимодействия определена другая структура **sockaddr_in**:

```
struct sockaddr_in
{
    short int sin_family; // Семейство адресов
    unsigned short int sin_port; // Номер порта
    struct in_addr sin_addr; // IP-адрес
    unsigned char sin_zero[8]; // Дополнение до размера структуры sockaddr
};
```

Здесь поле **sin_family** соответствует полю **sa_family** в `sockaddr`, в **sin_port** записывается номер порта, а в **sin_addr** - IP-адрес хоста. Поле **sin_addr** само является структурой, которая имеет вид:

```
struct in_addr {
    unsigned long s_addr;
};
```

Семейство сокетов **AF_UNIX** (также известное, как **AF_LOCAL**) используется для эффективного взаимодействия между процессами на одной машине. Доменные сокеты UNIX могут быть как безымянными, так и иметь имя файла в файловой системе (типовизированный сокет). В Linux также поддерживается абстрактное пространство имён, которое не зависит от файловой системы.

Сетевые сокеты являются универсальным типом сокетов. Система приложений, предназначенных для работы на одном компьютере, может использовать сетевые сокеты для обмена данными. Использование сетевых сокетов позволит сделать процесс

масштабирования проекта беспроблемным. 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.

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

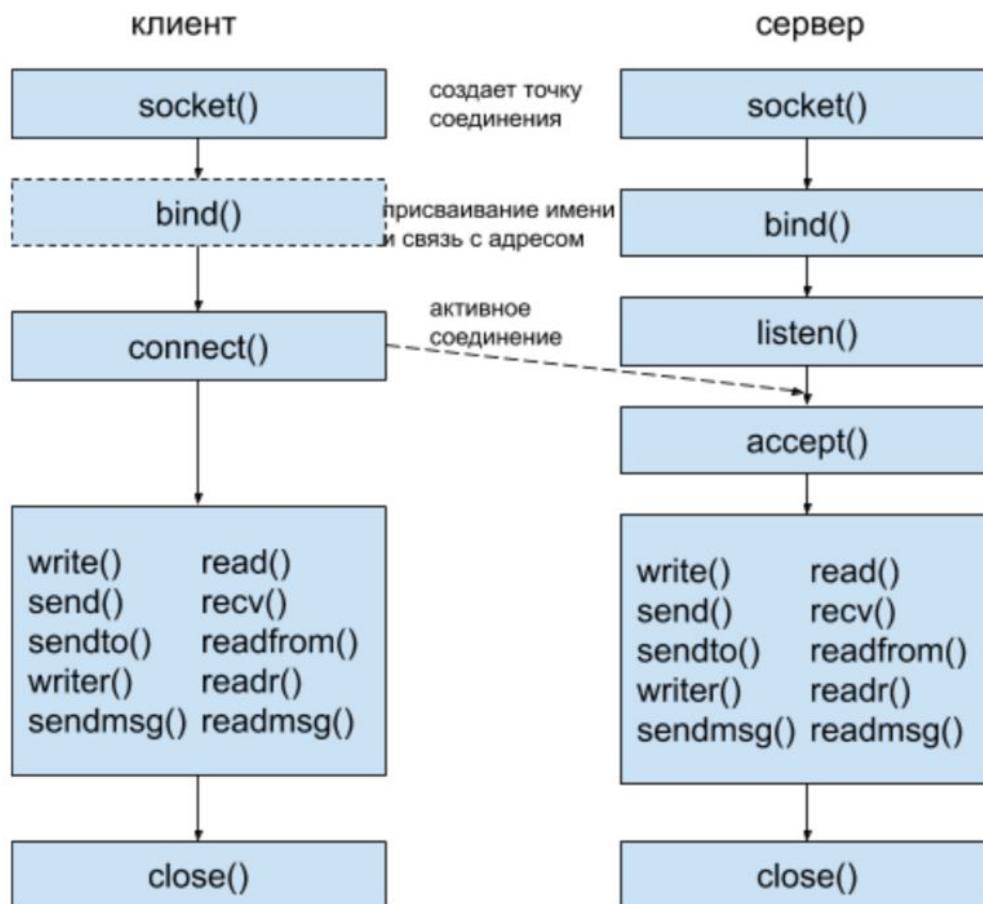
BIG ENDIAN - обратный порядок ($n \rightarrow n+1 \rightarrow n+2$) - network byte order (TCP/IP) LITTLE ENDIAN - прямой порядок ($n+2 < n+1 < n$) - аппаратный

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

Преобразования выполняются с помощью функций:

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

При работе с сокетами различают две роли - клиент и сервер



Примеры реализации взаимодействия процессов по модели клиент-сервер с использованием сокетов (Unix клиент-сервер):

client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>

#include "info.h"

int main(void)
{
    int sockfd = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (sockfd < 0)
    {
        printf("Error in socket()\n");
        return sockfd;
    }

    struct sockaddr server_addr;
    server_addr.sa_family = AF_UNIX;
    strcpy(server_addr.sa_data, SOCKET_NAME);

    char msg[MSG_LEN];
    sprintf(msg, "Hi, from %d\n", getpid());
    sendto(sockfd, msg, strlen(msg), 0, &server_addr, sizeof(server_addr));

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

server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <sys/socket.h> //socket, bind

#include "info.h"

int sockfd;
```

```
int main(void)
{
    char msg[MSG_LEN];
    struct sockaddr client_addr;

    sockfd = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (sockfd < 0)
    {
        perror("Error in socket(): ");
        return sockfd;
    }

    client_addr.sa_family = AF_UNIX;
    strcpy(client_addr.sa_data, SOCKET_NAME);

    if (bind(sockfd, &client_addr, sizeof(client_addr)) < 0)
    {
        printf("Closing socket....\n");
        close(sockfd);
        unlink(SOCKET_NAME);
        perror("Error in bind(): ");
        return -1;
    }

    printf("\nServer is waiting for the message...\n");

    for(;;)
    {

        int receivedSize = recv(sockfd, msg, sizeof(msg), 0);
        if (receivedSize < 0)
        {
            close(sockfd);
            unlink(SOCKET_NAME);
            perror("Error in recv(): ");
            return receivedSize;
        }

        msg[receivedSize] = 0;
        printf("Client send this message: %s\n", msg);
    }

    printf("Closing socket....\n");
    close(sockfd);
    unlink(SOCKET_NAME);
    return 0;
}
```

Билет 9

Файловая система: файлы и открытые файлы, процесс и файлы открытые процессом. Структуры описывающие процесс и открытые им файлы в системе, основные поля структур. Пример: файл два раза открывается для записи и в него последовательно записывается строка “aaaaaaa” и строка “bbbb”, затем файл закрывается два раза. Показать, что будет записано в файл и пояснить результат, если используется библиотека буферизированного ввода/вывода

Управление памятью осуществляется частью системы, которую принято называть **файловой системой**(подсистемой, *file system*) - отвечает за возможность надежного длительного хранения информации и доступа к ней.

ФС определяет способ организации хранения, именования и доступа к данным на вторичных носителях.

Если файл предназначен для хранения данных, то ФС управляет процессом хранения и обеспечивает последующий доступ к этой информации. Файл - каждая поименованная единица данных, хранящаяся во вторичной памяти

Обычные файлы расположены во вторичной памяти, на устройстве долговременного хранения, такие файлы также называются **регулярными**. Такой файл описывается структурой [inode](#). ВФС построена на 4 основных структурах:

- [inode](#),
- [dentry](#),
- [super_block](#),
- [file](#).

[inode](#) - описывает физический файл.

[inode](#) - дисковый inode в памяти (dentry - directory entry). [file](#) - описывает открытые файлы, т.е. в системе существует 2 структуры - описывающая файл, который лежит на диске, и структуры, описывающая открытый файл.

В системе существует 1 таблица всех открытых файлов - [struct file](#) .

тут надо описать структуру [file](#)

Открыть файл = обратиться к файлу на диске. ([struct file](#) это СТРУКТУРА, которая позволяет организовать работу с открытым файлом, а сам файл описывается inode-ом, а он обеспечивает доступ к файлам, которые находятся на физическом носителе).

Процесс – единица декомпозиции системы, именно ему выделяются ресурсы системы.

Процесс – программа в стадии выполнения.

Структуры данных, связанные с процессами, кроме `struct file` (формируется системная таблица открытых файлов, каждый открытый файл данной таблицы имеет дескриптор (строку)), но каждый процесс связан со следующими структурами:

- `task_struct` описывает запущенный в системе процесс. Создается динамически, кроме процесса `init`, для него создается структура `init_task` - статическая структура.
тут надо описать структуру task_struct
- `struct files_struct *files` - эта структура формирует таблицу открытых файлов процесса. Каждый процесс имеет собственную таблицу открытых файлов.
тут надо описать структуру files_struct
- `struct fs_struct *fs` - описывает файловую систему, к которой относится процесс. До запуска программы - это был исполняемый (обычный, регулярный) файл, и он тихо лежал на диске. Любой файл принадлежит какой-то ФС. `fs_struct` - описывает соответствующую ФС.
тут надо описать структуру fs_struct

Комментарии к структуре:

- `atomic_t` - атомарное, неделимое, действие по увеличению счетчика исполняется структурой как неделимое.
- `file_block` - спинблокировка, примитивная блокировка с активным ожиданием.

Процесс может открыть 256 файловых дескрипторов. Если массив полностью заполняется, то создастся еще 1 массив.

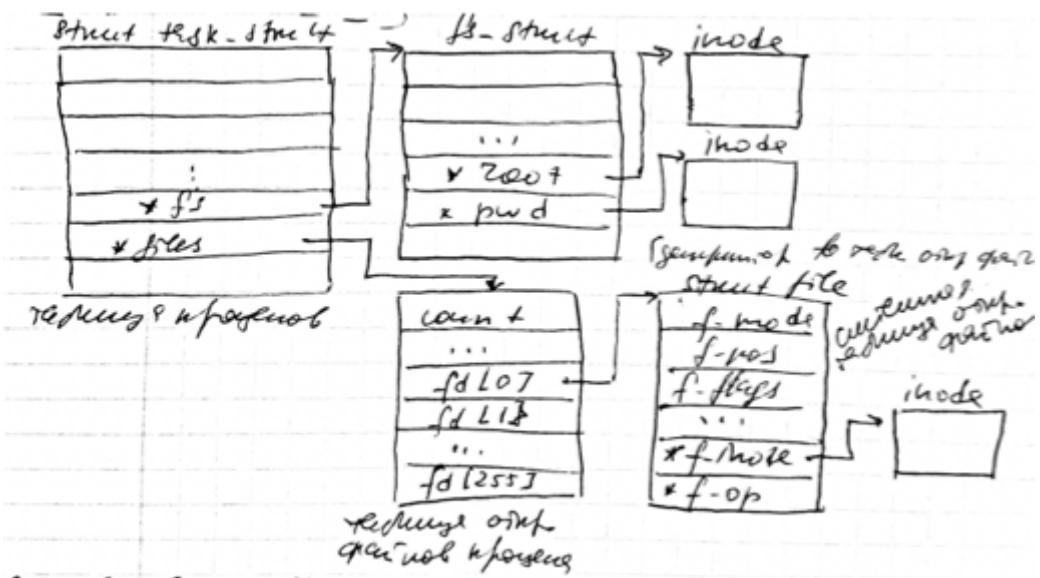
Информацией `fs_struct` может пользоваться процесс. `lock` - read/write lock, Блокировка чтения/записи - защита структуры.

`struct vfsmount` - структура, описывающая подмонтированные ФС.

тут надо описать структуру vfsmount

`rootmnt` - объект монтирования к корневой директории. Объект, например, файл (т.е. открытый файл), объект монтирования - то что подмонтировано.

Эти структуры связаны следующим образом:



(таблица открытых файлов процесса = системная таблица открытых файлов = files_struct)

Результатом программы будет “bbbbbaaaa”

Т.к. при вызове функции fopen() создаются 2 файловых дескриптора и 2 записи в таблице открытых файлов. Вспомним, что при вызове fprintf(...), запись производится в буфер. И только тогда, когда буфер будет заполнен полностью или если, будут вызваны функции fclose(...) данные будут записаны в файл. По этой причине, когда в программе вызывалась функция fclose(fs1), в файл записались “aaaaaaaa”, которые были в буфере файлового дескриптора fs1, тогда как после вызова функции fclose(fs2), содержимое файла затерлось, и записалась информация из буфера fs2.

Билет 10

Создание собственной файловой системы. Регистрация и deregistration файловых систем. Монтирование файловой системы. Пример файловой системы, ее регистрации и монтирования (лаб. раб.)

Функция `myfs_mount` должна примонтировать устройство и вернуть структуру описывающую корневой каталог файловой системы. По факту, 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. Для магического числа можно использовать значение 0x13131313.

Проинициализировав суперблок, `myfs_fill_sb` берется за построение корневого каталога нашей ФС. Первым делом для него создается `inode` вызовом `myfs_make_inode`, реализация которого будет показана ниже. Он нуждается в указателе на суперблок и аргументе `mode`, который задает разрешения на создаваемый файл и его тип (маска `S_IFDIR` говорит функции, что мы создаем каталог). Файловые и `inode`-операции, которые мы назначаем новому `inode`, взяты из `libfs`, т.е. предоставляются ядром. 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`.

Сборка и загрузка драйвера файловой системы ничем не отличается от сборки и загрузки обычного модуля, т.е. используются уже знакомые команды `insmod` и `rmmod`.

Вместо реального диска для экспериментов будем использовать `loop` устройство. Это такой драйвер "диска", который пишет данные не на физическое устройство, а в файл (образ диска). Создадим образ диска, пока он не хранит никаких данных, поэтому все просто: `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`. И опять проверяем системный лог.

код 6 лабы: https://github.com/zakolm/Operating-Systems/blob/master/Semestr%232/lab_06/pfs.c

Пример создания Ф.С. Структура

`file_system_type` написать ее

Заполняем структуру `struct file_system_type`:

```
static struct file_system_type encfs_fs_type = {
    .owner = THIS_MODULE,
    .name = "encfs",
    .mount = encfs_mount,
    .kill_sb = encfs_kill_block_super,
    .fs_flags = FS_REQUIRES_DEV,
};
```

В функции загрузки модуля вызывается register_filesystem с заполненной структурой:

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

В функции выгрузки модуля вызывается unregister_filesystem:

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

Передаем эти функции макросами:

```
module_init(encfs_init);
module_exit(encfs_cleanup);
```

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

```
static struct dentry *encfs_mount(struct file_system_type *fs_type,
                                  int flags, const char* dev_name, void *data) {
    struct dentry *ret;
    ret = mount_bdev(fs_type, flags_dev_name, data, encfs_fill_super);
    if (unlikely(IS_ERR(ret)))
        ...
    return ret;
}
```

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

Билет 11

Аппаратные прерывания в Linux: нижняя и верхняя половины обработчиков прерываний; softirq, tasklet, очереди работ - особенности реализации и выполнения в SMP-системах. Средства взаимоисключения в обработчиках прерываний. Примеры, связанные с планированием отложенных действий (лаб. раб.)

IRQ(Interrupt Request) - запрос на обработку прерывания;

Линия IRQ - контакт контроллера.

В конце каждой выполняемой команды процессор проверяет наличие прерывания ножке. Если сигнал поступил, то процесс переходит на выполнения обработчика прерывания - это процесс обработчика аппаратного прерывания.

Все прерывания от устройств ввода/вывода и системного таймера аппаратные. Все аппаратные прерывания выполняются на высоком уровне приоритета, и другие операции не могут выполняться пока аппаратное прерывание не будет завершено. Аппаратные прерывания делятся на две части:

- top half
- bottom half

Аппаратные прерывания принято делить на быстрые и медленные. Быстрые не делят на две части и выполняют как единое целое(в современном Linux быстрым является только прерывание от системного таймера). Нижняя половина прерывания выполняется как отложенное действие. Фактически аппаратным прерыванием является верхняя половина. В ее задачи входит получение данных из регистра устройства и помещение их в буфер ядра.

Завершается аппаратное прерывание инициализацией отложенного действия, например путем постановки нижней половины в очередь на выполнение. Процесс должен получить информацию об успешном завершении операции. С помощью соответствующих функций драйвер должен инициировать передачу данных из буфера ядра в буфер приложения, для этого процесс должен быть разблокирован.

Для постановки нижней половины в очередь на выполнение вызывается scheduler.

В настоящее время обработчики нижней половины бывают 3-х видов:

- softirq - определяется статически во время компиляции ядра;
- tasklet
- workqueue

В файле linux/interrupt.h определена структура softirq_action.

В kernel/softirq.c определен массив из 32-х экземпляров этой структуры.

Можно создать 32 обработчика softirq, в настоящее время определено всего 10.

Когда ядро выполняет обработчик отложенного прерывания, вызывается функция `action()` с указателем на соответствующую структуру `softirq_action` в качестве аргумента.

// отложенные прерывания описываются с помощью структуры (cat/proc/softirq)

```

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

// каждое зарегистрированное отложенное прерывание соответствует одному элементу
// массива
static struct softirq_action softirq_vec[NR_SOFTIRQS];

// обработчик отложенного прерывания
void softirq_handler(struct softirq_action *);

// вызов функции-обработчика
my_softirq->action(my_softirq);

// выполнение осуществляется, в цикле проверяется наличие отложенных прерываний
do_softirq()

// регистрация обработчика
open_softirq(NET_TX_SOFTIRQ, net_tx_action);

// генерация отложенного прерывания(маркировка), чтобы поставить в очередь на выполнение
raise_softirq(NET_TX_SOFTIRQ);

// инициализация отложенного действия (softirq)
static irqreturn_t xxx_inrerrupt(int irq, void *dev_id)
{
    raise_softirq(XXX_SOFT_IRQ);
    return IRQ_HANDLER;
}

```

Для создания нового уровня softirq нужно описание того, что выше:

1. Определить новый индекс отложенного прерывания вписав его константу в виде `xxx_soft_irq` в перечисление. Оно должно иметь уровень хоть на 1 ниже `TASKLET_SOFTIRQ`, иначе зачем определять новый уровень, если можно использовать тасклет;
2. Во время инициализации модуля должен быть зарегистрирован новый обработчик отложенного прерывания с помощью вызова `open_softirq()`;
3. Зарегистрированное `softirq` должно быть поставлено в очередь на выполнение, для этого оно должно быть возбуждено с помощью функции `raise_softirq()`.

Проверка ожидающих выполнения отложенный обработчиков прерываний и их запуск осуществляется в случаях:

1. при возврате из аппаратного прерывания
2. в контексте `ksoftirq`(демон)
3. в любом коде ядра, в котором явно проверяются и запускаются ожидающие обработчики отложенных прерываний, как это делается в сетевой подсистеме

Независимо от метода вызова softirq его выполнение происходит функцией `do_softirq`, которая в цикле проверяет наличие отложенных прерываний. Не смотря на то что у softirq есть приоритеты, **softirq никогда не вытесняет другой softirq**. Единственное событие, которое может вытеснить softirq - аппаратное (системное) прерывание. При этом на другом процессоре может выполняться другой обработчик этого же softirq. В результате для softirq **остро стоит проблема взаимоисключения**. Это означает, что **функция должна быть реентабельной, а если есть критический участок, то он должен быть защищен**. Демон `ksoftirq` это поток ядра каждого процессора, то есть `reg_CPU`. Когда машина нагружена мягкими прерываниями, которые как правило обслуживаются по возвращении из аппаратных прерываний, то возможна ситуация, когда такое softirq переключается значительно быстрее, чем может быть обслужено. Это связано с тем, что возможна ситуация, в которых IRQ приходит очень быстро и в результате ОС не может закончить обслуживание одного до прихода другого. Это может произойти тогда, когда сетевая карта с высокой скоростью получает пакеты в течение короткого промежутка времени. В результате операционная система не может с этим справится и создает очередь для последовательной обработки softirq с помощью специального процесса `ksoftirqd`. Если функция занимает больше чем какой-то небольшой процент процессорного времени это говорит о том, что машина находится под большим давлением. (говорит о большой нагрузке на машину).

* **Тасклеты** - это отложенные прерывания для которых обработчик(тасклет одного типа) **не может одновременно выполняться на нескольких процессорах**. Для них не требуется использовать средства взаимоисключения. Проще всего понимать тасклеты как простые в использовании `softirq` (прим: вот вообще ни разу проще не стало) . Разные тасклеты могут выполняться параллельно на разных процессорах. Компромисс между простотой и производительностью. Тасклеты **могут быть зарегистрированы в системе как статически так и динамически**.

Для того чтобы запланировать тасклет на выполнение должна быть вызвана функция `tasklet_schedule()`.

Отличия очередей работ от тасклетов.

1. Тасклеты выполняются в контексте прерывания, следовательно код тасклета должен быть атомарным. Функции очередей работ (код) выполняется в контексте специального потока ядра, следовательно, очереди могут блокироваться.
2. Тасклеты всегда выполняются на процессоре, где произошло аппаратное прерывание. Очереди по умолчанию выполняются точно также.

3. Код ядра может запросить чтобы выполнение функции очереди отложилось на определенный интервал времени, в тасклете так нельзя.
4. Очереди работ выполняются в контексте специального потока, следовательно, не должны быть атомарными.

Следующая структура определена в , в этой очереди находится задание очереди к процессору.

структуры картинки

Очереди работ должны быть автоматически созданы до использования

Существует две функции для отправки работы в очередь работ:

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

Функция будет выполняться в контексте потока worker и может спать если надо, но разработчик должен понимать как этот сон может повлиять на остальные задачи в этой же очереди. Не смотря на то, что в качестве параметров указана очередь на самом деле работы кладутся не в сами очереди работ а в список-очередь структуры `worker_pool` и эта структура по сути является главной в организации очереди работ

Рабочая очередь существует в `worker_pool` через `rwq`. Для каждого сри статически выделяется `worker_pool`: один для высоко приоритетных, один для обычных. Когда создается рабочая очередь, для каждого сри выделяется служебный `pool` - `rwq`. Через них рабочая очередь взаимодействует с пулом. Worker-ы выполняют все work-и подряд не разбирая из какой они изначально очереди. Для непривязанных очередей `working_pool` выделяется динамически. Для освобождения работы используются следующие функции: (Для принципиального завершения) `bool flush_work(struct work_struct *work);` `bool flush_delay_work(struct delayed_work *dwork);` Для отмены работы: `bool cancel_work_sync(struct work_struct *work);` `int cancel_delayed_work(struct work_struct *work);` вернут не ноль, если была отменена до вызова функции, ноль - вход уже начат и может еще выполниться, тогда `cancel_delay_work();`

В SMP архитектурах на всех процессорах запрещаются быстрые прерывания. Таких прерываний в системе не может быть много. Самое быстрое прерывание — прерывание от системного таймера.

P.S. Примеры смотри в 3 билете ;)

Билет 12

Аппаратные принципы взаимодействия с внешними устройствами: опрос, прерывания, прямой доступ к памяти. Аппаратные прерывания: обработчики аппаратных прерываний, регистрация обработчика аппаратных прерываний, регистрация обработчика прерывания в системе, задачи обработчика аппаратного прерывания, отложенные действия - задачи. Примеры(лаб. Раб.)

Способы организации ИО и виды ИО В современных ОС существуют 3 метода:

1. программируемый ИО
2. ИО с использованием прерываний
3. прямой доступ к памяти

1) Программируемый. Когда процессору встречается команда вв/выв, выполняя эту команду, процессор вызывает библиотечную функцию, кот. переводит процессор в режим ядра, осуществляет передачу команды вв/выв. Контроллер выполняет необходимые действия, затем устанавливает соответствующие биты в регистрах состояний вв/выв. Процессор д. периодически проверять состояние этих битов, т.е. опрашивать соответствующие регистры контроллера. На процессор возложено непосредственное управление вв/выв: опрос битов, пересылка команд чтения/записи, передача данных. Т.о. необходимо иметь следующий набор команд:

- команды управления (используется для инициализации внешних устройств, указывают им, что делать);
- команды анализа состояния (проверка состояний флагов вв/выв(прим:на другом ресурсе видел по другому - "проверки битов состояния контроллера"));
- команды передачи (из регистров ЦП в регистры устройства и наоборот)

Недостаток: процессор постоянно опрашивает флаги – остается мало времени для остального.

2) С использованием прерываний. Прерывание от устройства вв/выв поступает, когда оно завершило процесс вв/выв. Процессор освобождается от проверки флагов и м. переключиться на др. работу. Метод требует включения в состав ОС контроллера прерываний. Контроллер прерываний посыпает по шине управления сигнал. В конце выполнения каждой команды процессор проверяет входной сигнал с шины управления (если прерывания не замаскированы в ОС). Если получен сигнал – посыпается ответный сигнал контроллеру прерываний, в ответ контроллер прерываний формирует и посыпает вектор прерывания. Вектор передается по шине данных. Полученный вектор используется для процедуры обработки прерывания.

Процесс с точки зрения контроллера: Контроллер получает от ЦП команду (пр., read). Он переходит к считыванию данных со своего устройства. Как только эти

данные окажутся в регистрах контроллера, посыпается сигнал прерывания. После получения вектора ЦП - ом, контроллер посыпает по шине данных данные из своих регистров.

Процесс с точки зрения процессора: Процессор генерирует команду read. ЦП выполняет вызов библиотечной функции, которая переводит процессор в режим ядра (выполняется смена контекста: сохраняется содержимое программного счётика команд, другие регистров), выполняющийся процесс блокируется, процессор переходит на выполнение другой работы. В конце каждого цикла команд процессор проверяет наличие прерываний. При поступлении прерывания ЦП сохраняет контекст выполняемой задачи и переходит к выполнению п/п обработки прерывания. При этом процессор читает слова из контроллера вв/выв и пересыпает их в память (через свои регистры). По завершению восстанавливает свой контекст команды, то кот. поступила команда вв/выв (при условии соответствия приоритета – система динамического изменения приоритетов).

Недостатки: Кажд. прерыв. вызывает остановку выполнения текущих процессов и сохранение аппаратного контекста, любое слово, передающ. между памятью и устройством, должно пройти через регистры процессора. Но все равно намного эффективнее программируемого, т.к. искл. активное ожидание на процессоре.

3) Прямой доступ к памяти (ПДП, DMA – Direct Memory Access). Для передачи больших объёмов данных используется именно этот способ. Для реализации прямого доступа к памяти (далее ПДП) в состав компьютера включается контроллер ПДП, но иногда функции ПДП возлагаются на контроллер IO.. Устройству вв/выв или каналу временно передается управление памятью (обычно на время, равное 1 циклу памяти), благодаря чему слово или группа слов м.б. записана в память. Процессор своей памяти не имеет, поэтому все осущ. через ОП. Передача – через регистры процессора.

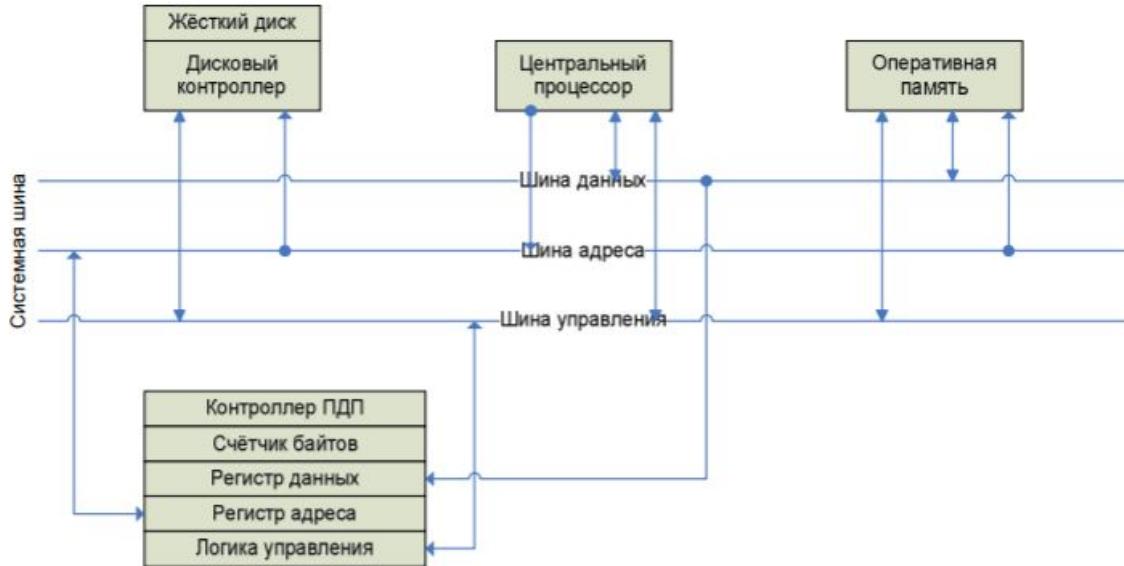
ЦП передает ПДП:

- тип команды (read или write);
- адрес устройства;
- начальный адрес блока в ОП, используемого для чтения или записи;
- количество байтов или слов - размер передаваемого блока

Преимущество: Сокращение числа прерываний

Недостатки: Контроллер ПДП отстает по быстродействию от ЦП. Поэтому, если контроллер ПДП не может поддерживать текущую скорость IO, то ПДП не используется, а используются 2 предыдущих способа.

Контроллер ПДП получает доступ к системной шине независимо от ЦП.



(Ввод-вывод с прямым доступом к памяти (Direct Memory Access – DMA) - более эффективная схема организации ввода-вывода, основанная на использовании фрагмента основной памяти в качестве буфера устройства для выполнения ввода-вывода. Схема используется с целью избежать программируемого ввода-вывода для больших пересылок данных. Схема требует специальной аппаратуры – DMA-контроллера – в настоящее время такие контроллеры приобретают все более широкое распространение. DMA позволяет избежать участия процессора в пересылках больших объемов данных непосредственно между устройством ввода-вывода и памятью. <https://www.intuit.ru/studies/courses/641/497/lecture/11306?page=2>)

Все **прерывания** от устройств ввода/вывода и системного таймера **аппаратные**. Все аппаратные прерывания выполняются на высоком уровне приоритета, и другие операции не могут выполняться пока аппаратное прерывание не будет завершено. Аппаратные прерывания делятся на две части:

- top half
- bottom half

В windows эта же идея реализована в виде DPC - отложенный вызов процедуры. Аппаратные прерывания принято делить на быстрые и медленные.

Быстрые не делятся на две половины, выполняются как единое целое. В современном Linux быстрым прерыванием является только прерывание от системного таймера.

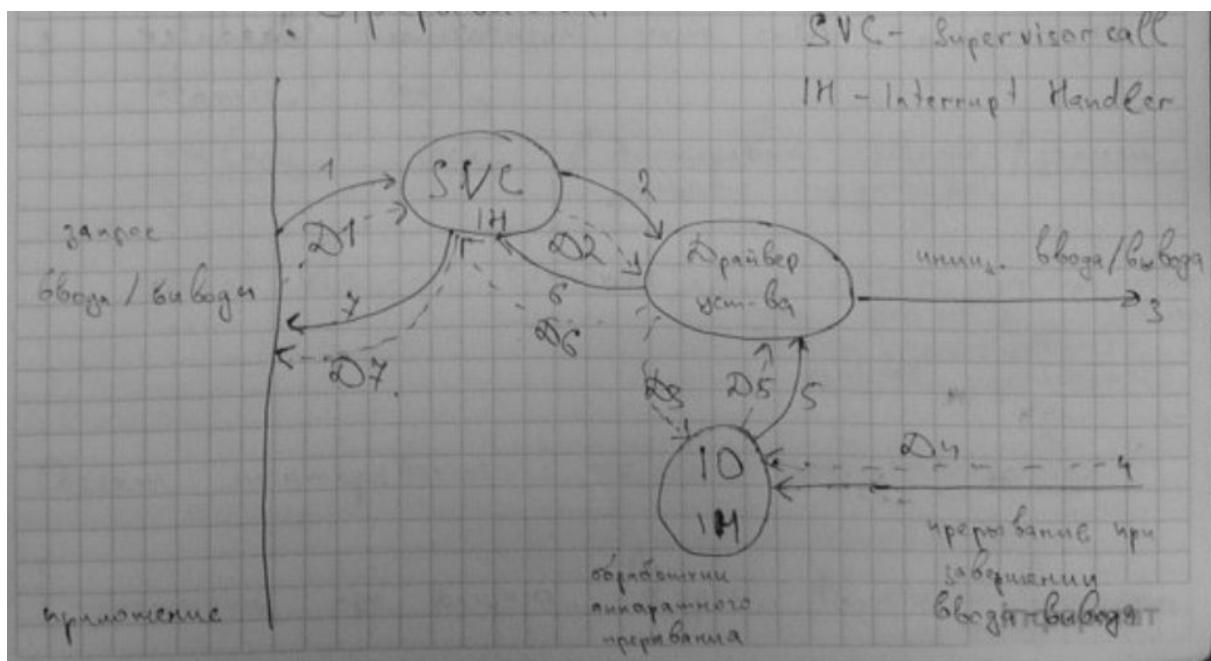
Нижняя половина прерывания выполняется как отложенное действие. Фактически аппаратным прерыванием является верхняя половина. В ее задачи входит получение данных из регистра устройства и помещение их в буфер ядра. Завершается аппаратное прерывание инициализацией отложенного действия, например путем

постановки нижней половины в очередь на выполнение. Процесс должен получить информацию об успешном завершении операции. С помощью соответствующих функций драйвер должен инициировать передачу данных из буфера ядра в буфер приложения, для этого процесс должен быть разблокирован.

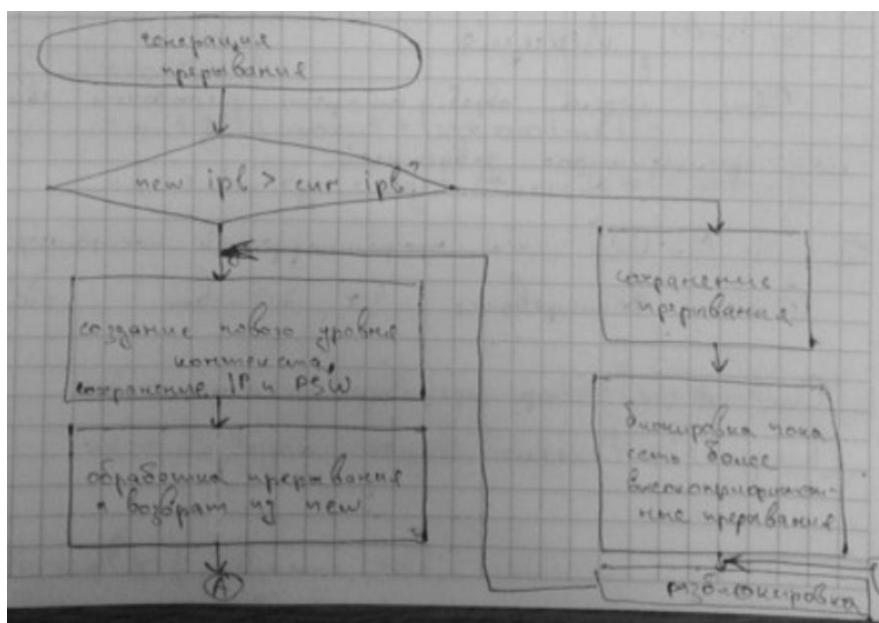
Хз нужно ли вот два названия

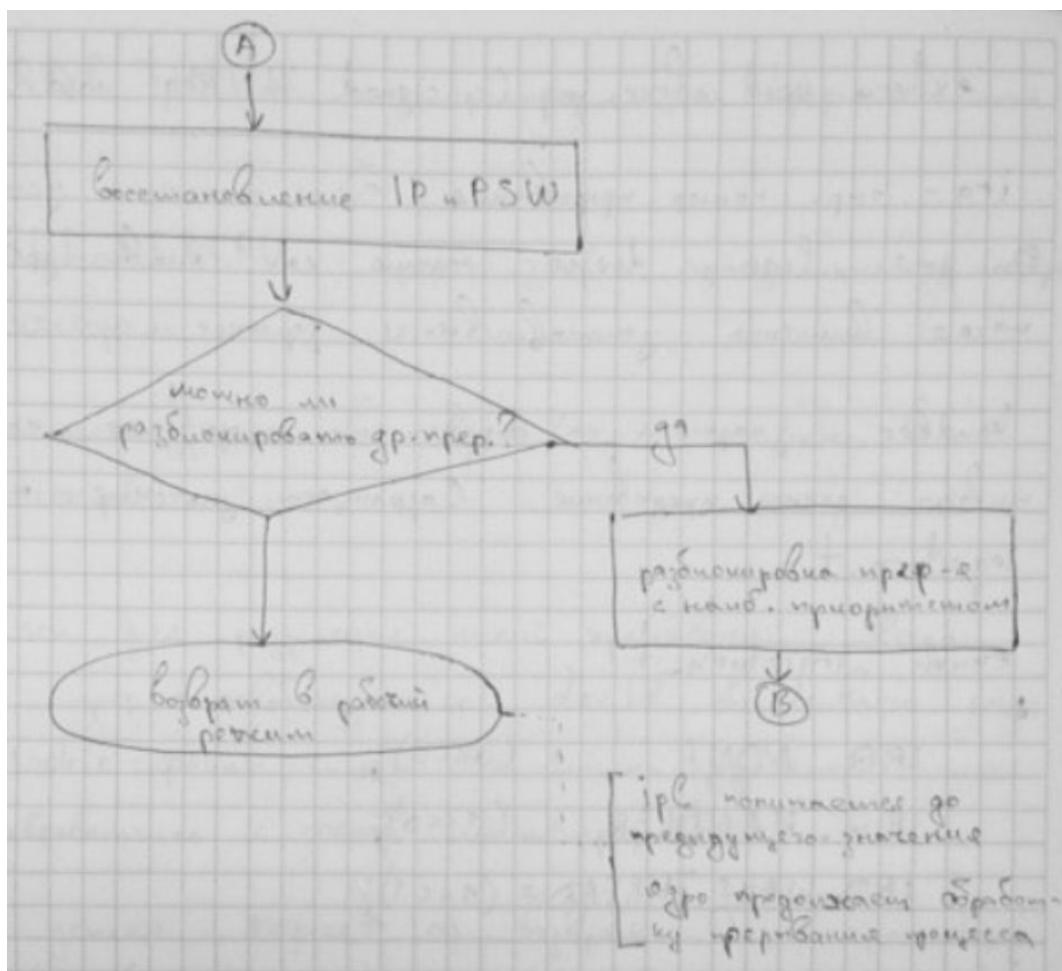
Действия системы на ввод/вывод программы.

Диаграмма состояний процесса при запросе ввода вывода



Общий алгоритм обработки прерываний Unix.





Драйвер может иметь один обработчик прерывания. Если устройство использует прерывания, то драйвер устройства регистрирует один обработчик прерывания. Для обработки прерывания драйвер может разрешить определенную линию прерываний IRQ:

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

- **irq** - номер, прерывания, который будет обрабатывать обработчик.
- **handler** - указатель на функцию обработчика прерывания, который обслуживает прерывание
- **irqflags** - битовая маска:
 - NULL
 - IRQF_SHARED - на одну линию прерывания можно зарегистрировать несколько обработчиков
 - IRQF_PROBE_SHARED - устанавливается вызывающим, когда предполагается возможное возникновение проблем при использовании линии IRQ

- IRQF_TIMER - маскирует прерывание как прерывание от таймера
- IRQF_NOBALANCING - запрещает использовать это прерывание для балансировки IRQ
- **dev_name** - текст, представляющий устройство, связанное с этим прерыванием.(Используется в /proc/irq и /proc/interrupts) **dev_id** - используется для разделения SHARED линии прерывания. возвращаемое значение -
- 0 - OK
- не 0 - ошибка
- E_BUSY - данная линия прерывания уже используется и не указан флаг IRQF_SHARED.

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

- irq - численное значение линии прерывания, которую обслуживает обработчик.
- dev_id - общий указатель на тот же самый dev_id, что задается в request_irq, когда обработчик был зарегистрирован.
- regs - структура, содержащая регистры процессора и состояние перед обслуживанием прерывания. Обычно используется для отладки

Возвращаемое значение:

- IRQ_NONE - обработчик прерывания обнаруживает прерывание, которое его устройство не инициировало;
- IRQ_HANDLER - обработчик прерывания был вызван корректно и его устройство действительно вызвало прерывание.

Медленные прерывания top half <--> bottom half

Обработчик прерывания делится на 2 части:

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

Завершение обработки прерывания выполняется нижней половиной

Задачи верхней половины:

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

В настоящее время есть 3 типа нижних половин:

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

Пример тасклета в 3-ем билете

Билет 13

Файловая подсистема: особенности файловой подсистемы Unix/Linux.: иерархическая структура файловой подсистемы. Монтирование файловых систем. Виртуальная файловая система - интерфейс vfs, суперблок, индексные узлы inode и dentry. Адресация файлов большого размера и пример, показывающий доступ к /usr/ast/mbox.

Мы рассматривали машину Медника Донаавана. На самом верхнем уровне был уровень управления данными.

Управление файлами осуществляется файловой системой (file system), она отвечает за возможность длительного хранения информации и доступа к ней.

Файловая система управляет процессом хранения и обеспечивает последовательный доступ к информации. **Файл – единица хранения информации**

Обычные файлы хранятся во вторичной памяти – для длительного хранения. Временные файлы - в оперативной памяти для обслуживания действий системы в текущий момент времени

Файл – любая поименованная совокупность информации, хранящаяся во вторичной памяти.

7 типов файлов:

1. - : **обычный** файл
2. d : **директория**
3. c : файл **символьного** устройства
4. b : файл **блочного** устройства
5. s : **сокет**
6. p : именованный **канал**
7. l : **символическая** ссылка

В юникс 2 типа файлов:

- **байт-ориентированные** (инфо хранится по байтам, передача инфы байтами)
- **блок-ориентированные** (инфо хранится блоками фиксированного размера), могут работать те программы, которые знают структуру блока.

Файловая подсистема определяет

- формат, способ физического хранения
- связывает формат физического хранения и API доступа к файлам. (Фс не интерпретирует инфу, хранящуюся во вторичной памяти, этим занимаются соответствующие программы.)

2 неравные части жесткого диска.

1 – область хранения файлов, управляемая файловой системой

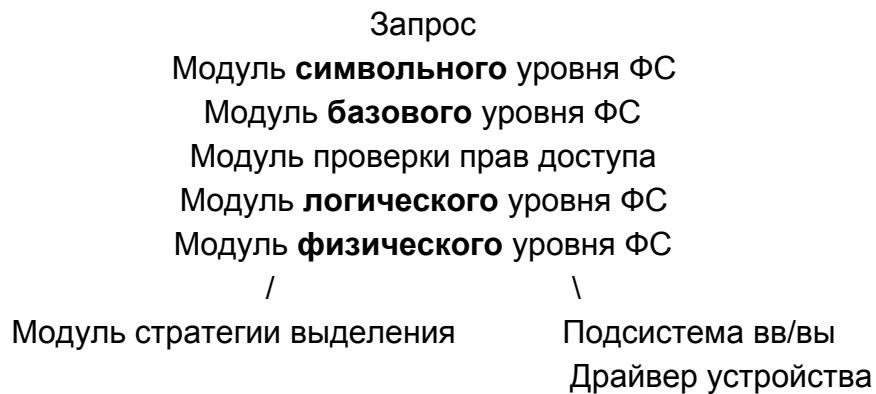
2 – область пейджинга или своппинга для управления виртуальной памятью – подкачка. когда оперативной памяти не хватает, некоторые страницы загружаются в область подкачки.

Задачи ФС:

- 1) обеспечение удобного доступа к файлам как часть задачи именования файлов

- 2) собственно именование файлов – присвоение файлам уникальных идентификаторов, с помощью которых обеспечивается доступ к файлам
- 3) обеспечение программного интерфейса для работы с файлами пользователей и приложений
- 4) отображение логической модели (представления) файлов на физическую организацию хранения данных на соответствующих носителях.
- 5) обеспечение надежного хранения файлов, доступа к ним, обеспечения защиты от несанкционированного доступа.
- 6) обеспечение совместного использования файлов

Иерархическая организация ФС.



Символьный уровень — уровень именования файлов (тот интерфейс который предоставляется системой в распоряжение пользователя) Разные ограничения. Они касаются как длины имени, регистра, расширения

Базовый уровень — это уровень формирования дескриптора файла . Для этого система должна иметь соответственные структуры, которые позволяют хранить о файле всю необходимую информацию.

Логический уровень - лог адресное пространство файла аналогично лог адр пр-ву процесса. (начинается с нулевого адреса и представляет собой непрерывную последовательность адресов)

Логический уровень позволяет обеспечить доступ к данным, хранящимся в файле в формате, отличном от формата физического хранения . Обычно система не накладывает никаких ограничений на внутреннюю структуру данных, записанных в файл, и никак ее не интерпретирует .

Физический уровень — задача обеспечить непосредственный доступ к информации, которая хранится на внешнем носителе.

Разные ФС должны обеспечивать возможность существования в системе нескольких файлов с одинаковыми именами, возможность доступа к 1 и тому же файлу по разным именам. Поэтому в системах должна существовать соответствующая инфа – **справочник**.

Такой справочник состоит из 2 частей:

Символьный уровень именования файлов - удобный пользователям.

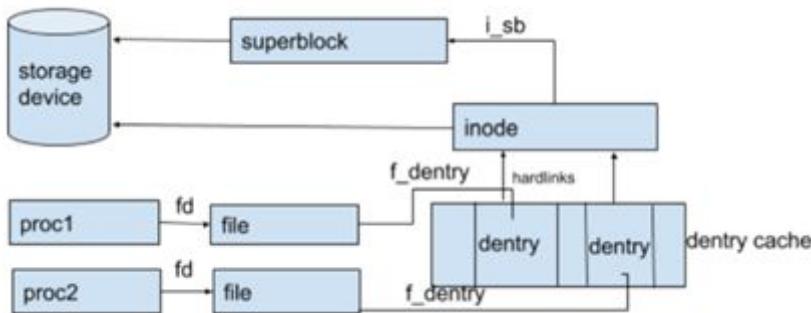
Базовая часть – уровень идентификации.

UNIX поддерживает большое количество ФС. Одна ФС описывается структурой `file_system_type`. Файловых систем может быть подмонтировано сколько угодно.

VFS предоставляет общую файловую модель, способную отображать общие возможности и поведение любой мыслимой ФС. Уровень абстракции вфс работает на основе базовых концептуальных интерфейсов и структур данных. Каждая ФС определяет особенности того, как файл открывается, как выполняется обращение к файлу как считывается инфа из файла и т.д. Код ФС скрывает детали конкретной реализации, однако все ФС поддерживают такие понятия как файлы, каталоги, такие действия как создание файла, удаление, переименование, открытие, чтение, запись, закрытие и т.д. большинство ФС запрограммировано так, что API, которые предоставляют ФС рассматриваются как абстрактный интерфейс, который ожидаем и понятен VFS.

`Write (f, fbuf, lim)` – сначала обрабатывается общим системным вызовом VFS (`sys.write`). Затем осуществляется вызов соответствующего вызова конкретной файловой системы, в данном случае для записи данных в файл.

Внутренняя организация VFS. Базируется на 4 структурах: superblock, dentry, Inode, file



1) Суперблок (контейнер) содержит высокоуровневые **метаданные** о **ФС**. Содержит информацию, необходимую для **монтирования** и **управления** ФС. Каждая **ФС** имеет **один суперблок**, но на **диске** в **нескольких** экземплярах для **надежности** хранения, ибо суперблок – **ключевая** структура.

Структура определяет параметры для управления ФС (например, суммарное число блоков). Суперблок на диске предоставляет ядру системы информацию о структуре ФС, информацию для управления смонтированной ФС.

Суперблоков столько, сколько ФС. Они хранятся в списке `struct list_head`.

`Struct super_operations` определяет операции над суперблоком. Когда ФС нуждается в выполнении операции над суперблоком, то она следует за указателем на желаемый метод объекта суперблока.

ФС описывается структурой `file_system_type`.

Тип файловой системы – один, но подмонтировано их может быть сколько угодно.
get_sb вызывается во время монтирования ФС, создает суперблок – сейчас mount.
Чтобы создать ФС, надо выделить раздел – выделить объем вторичной памяти для хранения данных конкретной ФС. Суперблок описывает данный раздел.
Суперблок располагается на **диске**, но для **активных** ФС эта структура копируется в **оперативную память**, чтобы **ускорить** доступ к информации.

struct super_block написать

2) Индексный узел inode - описывает физический файл

inode- **дескрипторы** физических файлов. файлы могут **иметь имена** в виде символьных строк, **имя** файла в их родных файловых подсистемах **не является идентификатором, им является номер inode**.

В **Inode bitmap** хранится инф о том какие inode **заняты**, а какие **свободны**. В **суперблоке** хранится инф о том, **сколько в данной ФС имеется свободных айнодов**.

Система, чтобы получить **доступ к файлу**, ищет **номер inode** в **таблице** «таблица inode'ов».

Такие **устройства как CD, DVD, флэшки** системы рассматриваются как **файлы**. их мы можем увидеть в директории dev - это **устройства, но для ОС это файлы**.

inode кеш состоит из:

глобальный хеш массив – где каждый айнод хешируется по значению указателя на суперблок, если айнод не связан с суперблоком он помещается в список анонимных айнодов – пример сокет.

глобальный список Inode-in-use. в этом списке содержится допустимые inode с i_count > 0 и i_nlink > 0. в этот же список записываются вновь созданные объекты inode

глобальный список inode_unused. содержит допустимые inode с i_count = 0

Список измененных айнодов. список для каждого суперблока (sb->s_dirty), что содержит inode с i_count > 0 и i_nlink > 0

SLAB cache inode_cache. объекты inostruct inode написатьde могут вставляться, освобождаться и изыматься из SLAB кеша

struct inode написать

3) Объект dentry - это определенный компонент пути.

Причем все объекты dentry – это компоненты пути включая обычные файлы. Элементы пути могут включать в себя **точки монтирования**. Объект dentry не **соответствует** какой-либо **структуре** данных на **диске** во вторичной памяти. Подсистема **VFS** создает эти объекты **быть** – **налету в оперативной памяти** по строковому представлению имени пути. Поскольку объекты элементов каталога не хранятся на физическом диске, структура dentry **не имеет флагов**, которые **указывают**, что объект был **изменен** и был **записан на диск**.

1 айноду может соотв несколько дентри.

Актуальный объект dentry может находиться в одном из **3х состояний**:

- **used** - связан с айнод, счетчик ссылок больше нуля
- **unused** – связ с йнод, но счетчик ссылок = 0

- **negative** – не связь с айнод

Дентри кешируется и состоит из:

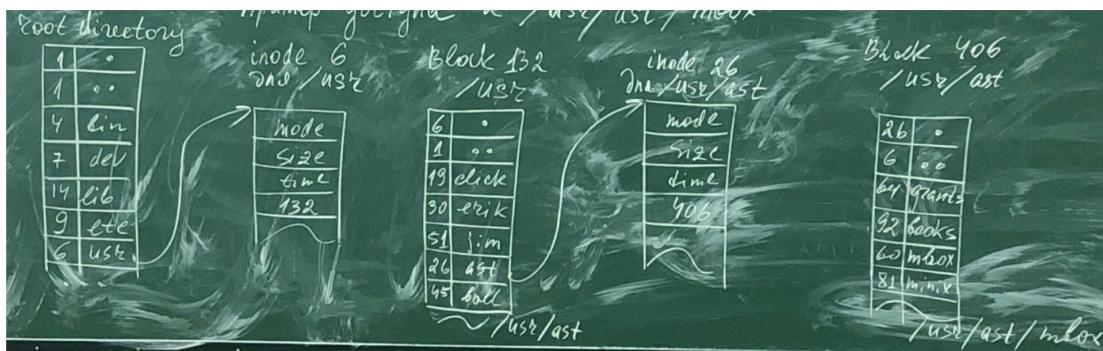
список used дентри

список неисп и негативных дентири (по алг лру - в начале списка – самый новый дентри.

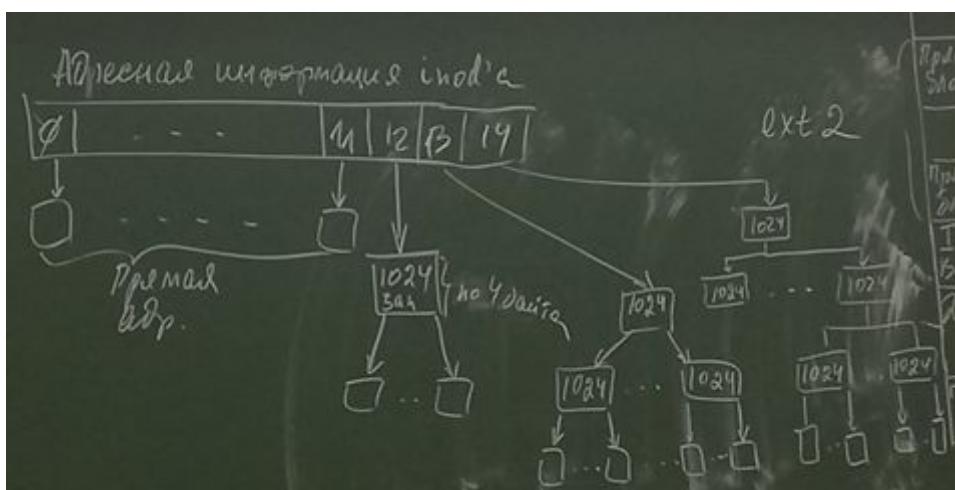
Удаление последнего в списке дентри)

хеш таблица, хеш функция кот преобразуют путь в связанный объект дентри.

struct dentry написать



Пример доступа /usr/ast/mbox.



В настоящее время используется **слаб-кеширование (слаб - брусков)**. Слаб-кеш – фактически выделение памяти одного и того же размера для данных одного и того же типа. Поэтому доступ к этим данным ускоряется. Делается это для того, чтобы не инициализировать некоторые объекты заново.

Билет 14

Файловая система Linux - открытые файлы: структуры, связанные с процессом и struct file, struct file_operations. Примеры. Регистрация и дерегистрация файловых систем. Примеры (лаб. раб.)

Управление файлами осуществляется файловой системой (file system), она отвечает за возможность длительного хранения информации и доступа к ней.

Файловая система управляет процессом хранения и обеспечивает последовательный доступ к информации.

Обычные регулярные файлы предназначены для долговременного хранения информации на энергонезависимых носителях.

80% файлов занимают не больше 1 блока (4 Кб).

Когда мы работаем с файлами, вся информация буферизуется. Доступ к этой информации ускоряется. Это делается автоматически.

struct file описывает открытые файлы - файлы, с которыми работает процесс. Существует системный вызов `open()`. Он может создать новый файл. Структура предназначена для работы с открытым файлом.

struct file

`f_op` - указатель на таблицу файловых операций, в ней перечислены все действия, определенные на файлах; например, для разработчиков драйверов можно даже переопределять функции доступа к файлам; любой драйвер может переопределить файловые операции; каждый тип ФС имеет свой набор файловых операций.

struct file_operations

Пример: Необходимо переписать стандартные функции `read`, `write`. Система предоставляет такую возможность.

```
static struct file_operations fops =  
{  
    .open = my_proc_open,  
    .read = my_proc_read,  
    .write = my_proc_write,  
    .release = my_proc_release  
};
```

Когда открываем файл, создается объект файл.

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

Обе функции выполняют задачу перемещения данных. Оно связано с перемещением из режима `user` в режим ядра или наоборот. Следовательно действия не могут выполняться обычным образом (обращение к памяти) через `memcpy()`. Адреса пространства пользователя не могут использоваться напрямую в режиме ядра по ряду причин. Основная причина - память режима пользователя может своппироваться (`swapped out`).

Когда ядро обращается к какому-то указателю режима пользователя, то связанная с ним страница может не находиться в памяти. В результате генерируется страничное исключение, которое переводит процесс в режим сна или блокировки до тех пор, пока нужная страница не будет загружена в память. Для безопасного перемещения данных существует функции, которые мы использовали в 4 лабе: `copy_to_user`, `copy_from_user`.

- `unsigned long copy_to_user(void *to, const void *from, unsigned long count);`
- `unsigned long copy_from_user(void *to, const void *from, unsigned long count)`

Таблица открытых файлов – это системная таблица, пользователю не доступна. Процесс имеет указатели на следующие файловые структуры:

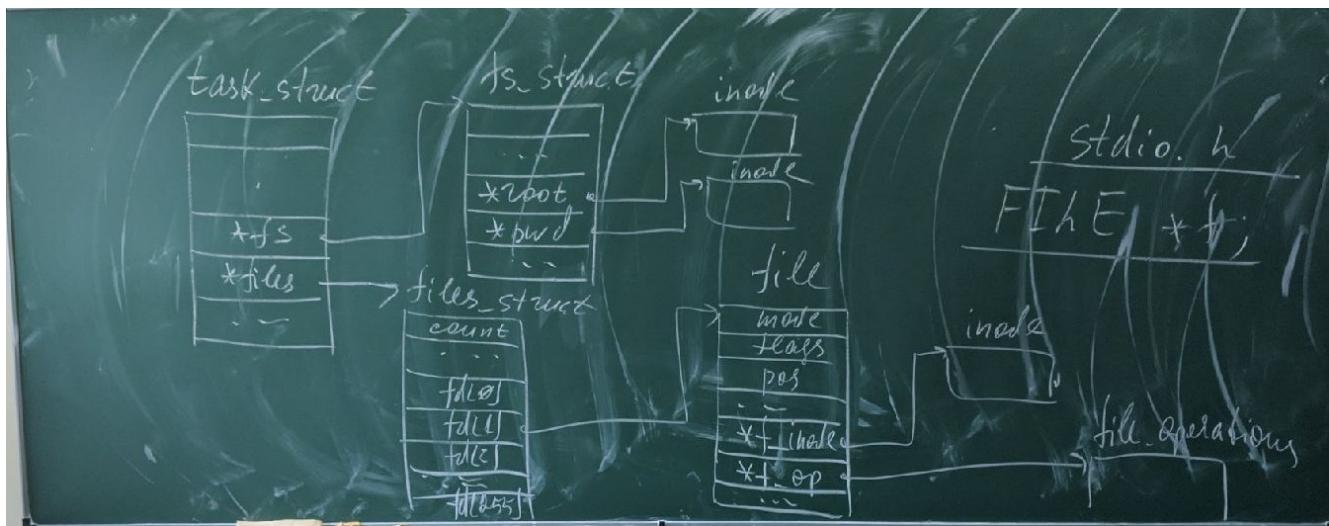
Структуры данных, связанные с процессами, кроме `struct file` (формируется системная таблица открытых файлов, каждый открытый файл данной таблицы имеет дескриптор (строку)), но каждый процесс связан со следующими структурами:

- `task_struct` описывает запущенный в системе процесс. Создается динамически, кроме процесса `init`, для него создается структура `init_task` - статическая структура.
тут надо описать структуру task_struct
- `struct files_struct *files` - эта структура формирует таблицу открытых файлов процесса. Каждый процесс имеет собственную таблицу открытых файлов.
тут надо описать структуру files_struct
- `struct fs_struct *fs` - описывает файловую систему, к которой относится процесс. До запуска программы - это был исполняемый (обычный, регулярный) файл, и он тихо лежал на диске. Любой файл принадлежит какой-то ФС. `fs_struct` - описывает соответствующую ФС.
тут надо описать структуру fs_struct

Комментарии к структуре:

- `atomic_t` - атомарное, неделимое, действие по увеличению счетчика исполняется структурой как неделимое.
- `file_block` - спинблокировка, примитивная блокировка с активным ожиданием.

связь структур:



Файловая система должна иметь **корневой каталог**, чтобы ее можно было смонтировать.

Каталог, *прямо или косвенно включающий в себя все прочие каталоги и файлы файловой системы, называется **корневым**.*

Короткое имя - имя файла

Полное имя - путь

Любая файловая система должна иметь тип. Тип определяется `struct file_system_type`.

Смонтированных файловых систем **одного типа** может быть **много**. при монтировании файловой системы создается суперблок (потому что он описывает файловую систему).

Нельзя выгрузить файловую систему пока счетчик ссылок на нее не будет равен 0.

Необходимо зарегистрировать файловую систему. Когда нужда в ней отпадает – нужно отреестрировать.

```
static const unsigned long MYFS_MAGIC_NUMBER = 0x31777;

static void myfs_put_super(struct super_block *sb)
{
    printk(KERN_INFO "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,
};

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

```
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 = MYFS_MAGIC_NUMBER;
    sb->s_op = &myfs_super_ops;

    root = myfs_make_inode(sb, S_IFDIR | 0755);
    if (!root)
    {
        printk(KERN_ERR "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 "root creation failed!\n");
        return -ENOMEM;
    }

    return 0;
}
```

```

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_INFO "myfs mounted\n");
    return entry;
}

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

static int __init myfs_init(void)
{
    int ret = register_filesystem(&myfs_type);
    if (ret != 0)
    {
        printk(KERN_ERR "MYFS_MODULE cannot register filesystem!\n");
        return ret;
    }
    printk(KERN_INFO "MYFS_MODULE loaded\n");
    return 0;
}

static void __exit myfs_exit(void)
{
    int ret = unregister_filesystem(&myfs_type);

    if (ret != 0)
        printk(KERN_ERR "MYFS_MODULE cannot unregister filesystem!\n");

    printk(KERN_INFO "MYFS_MODULE unloaded\n");
}

module_init(myfs_init);
module_exit(myfs_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Ananastyta");

```

Билет 15

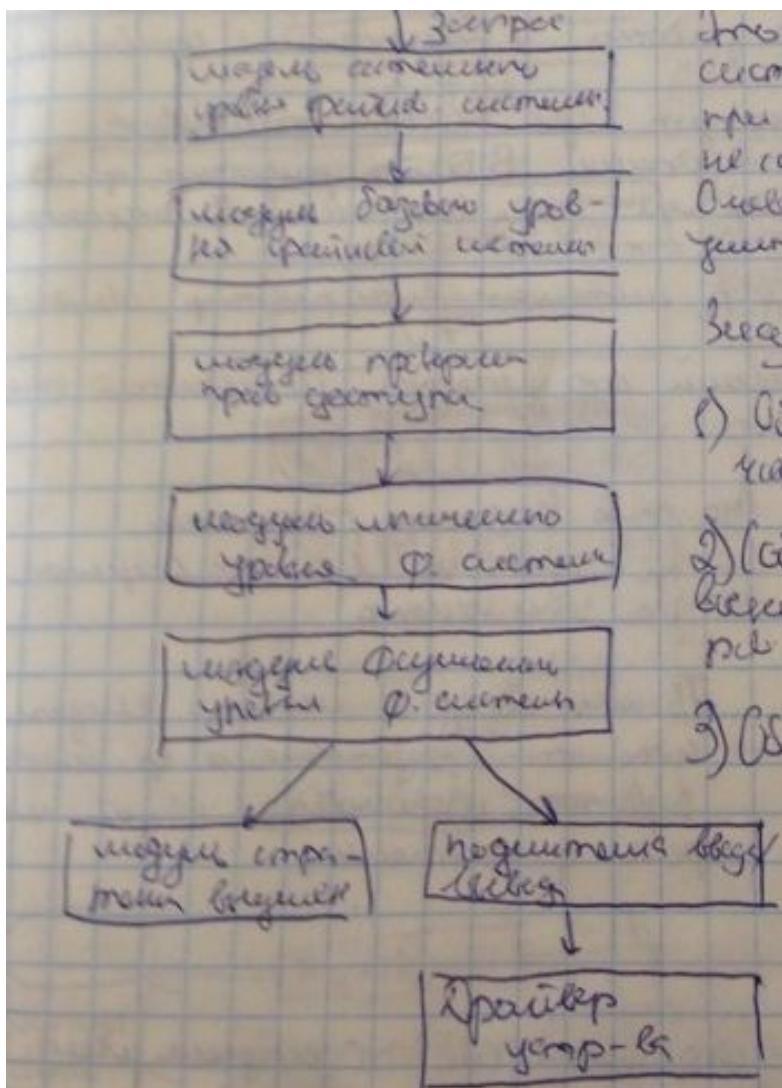
Файловые системы. Иерархическая структура файловой системы. Задачи уровней файловой системы. Доступ к файлам - работа с директориями: struct dentry, структура inode каталога. Пример, демонстрирующий доступ файлу.

Файл - любая поименованная совокупность данных. Их содержание системе не интересует, только возможность доступа к ним. Файл - информация, хранимая во вторичной памяти или во вспомогательном ЗУ с целью ее сохранения после завершения конкретного задания и в преодолении ограничений, связанных с объемом рабочих ЗУ. **Файловая система** - порядок, определяющий способ организации хранения, именования и доступа к данным на вторичных носителях.

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

Задачи Ф.С.

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



1. символный уровень - предоставляет возможность систематизации документов. уровень именования файлов и системат. управление информацией, доступной пользователю.
2. базовый уровень - уровень идентификации файла, файл должен иметь уникальный идентификатор и файл должен быть описан в системе, чтобы можно было с ним работать.
3. логическое а.п. файла аналогично логич.а.п. процесса позволяет обеспечить доступ к данным в файле в формате, отличном от формата физ.хранения обычно система не накладывает ограничения на внутр. структуру данных в файле
4. физический уровень - обеспечение непосредственного доступа к информации на внешнем носителе.

struct dentry - directory entry

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

Для ускорения доступа информация, полученная с помощью dentry кешируется.

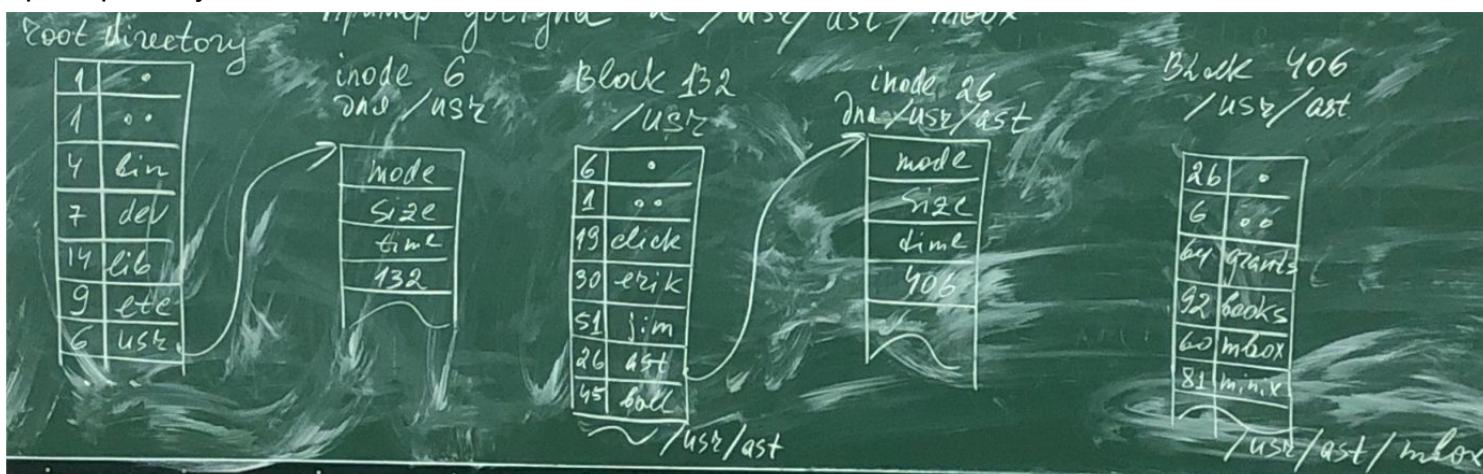
dcache состоит из 3х частей:

1. список используемых объектов dentry, которые связаны с inode'ми полем d_inode т.к. отдельно взятый inode может иметь много link, то может быть много объектов dentry связанных с этим inode
2. двухсвязный список LRU - это список неиспользуемых и отрицательных объектов dentry этот список отсортирован по времени соответственно в начале этого списка будет самый новый объект dentry
3. hash_table - хеширует функции для быстрого определения пути. Эта таблица реализована в виде dentry_hash_table массива каждый элемент является указателем на список dentrys хешированному по тому же самому имени.

struct inode

В Ф.С. каждый файл имеет свой inode. (/proc - 1, / - 2) При создании Ф.С. создается таблица inode-ов с информацией о том, какие inode заняты, а какие свободны. В суперблоке хранится информация о том, сколько в системе свободных inode-ов. ОС хранит информацию о файле в inode - метаданные о данных. Когда пользователь ли программа нуждается в доступе к файлу система ищет точный и уникальный inode в таблице inode-ов. Чтобы получить доступ к файлу по его имени, нужно найти inode данного файла. Для доступа к номеру inode имя файла не требуется. Структура inode:

Пример доступа к /usr/ast/mbox



Билет 16

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.
Реализация системного вызова open() в системе: связь функций open(), sys_open(), filp_open(). f_op->open(). Назначение inode кешей и их виды, slab cache- особенности.

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

```

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

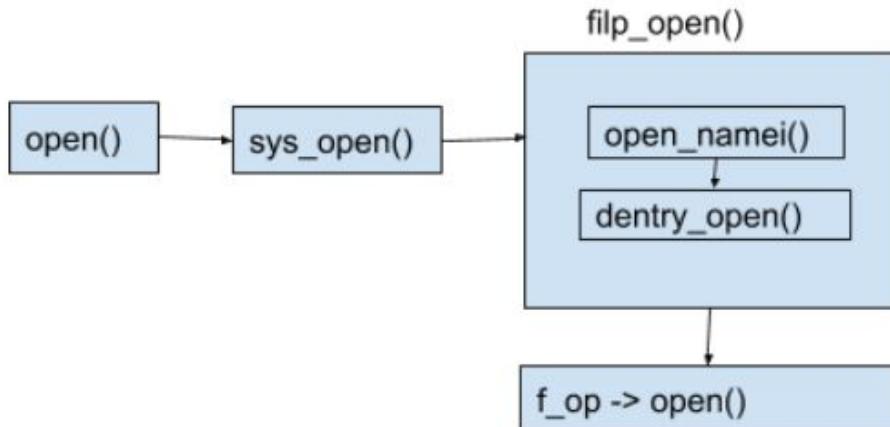
```

Открывает файл, определенный pathname. Если файл не существует, то если установлен флаг O_CREATE, то он м.б. создан. Флаг O_EXCL приводит к ошибке, если файл уже существует. Комбинация O_CREATE и O_EXCL проводит проверку существования файла и его создание и делается это автоматом. Возвращают файловый дескриптор открытого файла. Смещение файла - в начале файла. Создается новый дескриптор открытого файла, запись в системной таблице открытых файлов. Получить файловый дескриптор можно с помощью функции

int fileno(FILE *fp);
mode - битовая маска доступа

(user, group, other).

S_IRWXU - 00700 (user read/write/execute);
S_IRWXG - 00070 (group read/write/execute);
S_IRWXO - 00007 (other read/write/execute);



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

```

int sys_open(const char *name, int flags, int mode)
{
    char *tmp = getname(name);
    int fd = get_unused_fd();
    struct file *f = filp_open(tmp, flags, mode);
    19fd_install(fd, f);
    putname(tmp);
    return fd;
}

```

getname() - функция, копирующая имя файла из user в kernel space. Используются специальные функции типа copy_to_user. Она возвращает char * , но это будет имя в пространстве ядра.

get_unused_fd() - ищет первый неиспользованный дескриптор (пустой слот в таблице дескрипторов открытых файлов процесса). Если процесс открыл слишком много файлов, то вернется ошибка, в случае успеха будет вызвана функция filp_open() . Данная функция должна будет создать структуру file (заполнить ее) (возвращает указатель на struct file , будет создан дескриптор открытого файла).

get_unused_fd ищет пустую строку в таблице дескрипторов файлов. В случае успеха - вызов filp_open:

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

```
struct nameidata
{
    struct dentry *dentry;
    struct vfsmount *mnt;
    struct qstr last;
}
```

path_walk вызывается в open_namei.

path_walk - для полей устанавливаются значения и в общем-то происходит обращение за dentry к

кешу. Выполняется вызов кеш lookup. В системе существует dentry и inode cache, эта информация изначально ищется там, и только если ее там не найдется, то обращение происходит к информации на диске. Открытие уже существующего файла отличается от создания нового - при открытии для

чтения или дозаписи, создается структура, связанная с файлом, а если создается новый - создается inode, который должен быть кеширован.

filp_open после заполнения полей структуры nameidata выполняет dentry_open . После выполнения path_walk (путь по которому мы идем) аргумент nd содержит требуемое dentry, который связан с информацией, содержащейся в inode о файле. dentry_open инициализирует file_struct . Ищет пустой filp и заполняет поле fdentry , f_pos обнуляется и полю f_op присваивается значение f->f_op = dentry->d_inode->i_fop . Выполняется спуск по дереву, для каждой компоненты ищется информация в кеше, если ее там нет, то обращение происходит к ФС, т.е. к структурам, которые описывают inode, к дисковым inode (super_block , inode). Каждый тип ФС определяет структуры super_operations - операции на суперблоке, file_operations -

операции над открытыми файлами в ФС конкретного типа, inode_operations - операции на inode.

inode хеш представляет из себя:

- глобальный хеш массив `inode_hashtable`, в котором каждый inode хешируется по значению, указанному на суперблоке и по 32 разрядному номеру inode.
- глобальный список `inode_in_use`
- глобальный список `inode_unused` - неиспользуемые inode, содержащие допустимые inode с `icount=0`
- список для каждого суперблока

`sb->s_dirty` который содержит inode-ы с `icount > 0` и `inlink > 0` (когда inode помечен как грязный, он добавляется к списку `sb->s_dirty`, при условии что он был хеширован. Поддержка такого списка уменьшает расходы на взаимоисключение).

SLAB-аллокатор (мы рассматриваем SLAB с точки зрения выделения памяти под inode) - это кеш ссылок, который позволяет выделять блоки памяти одного размера. SLAB распределение стало использоваться для упрощения управления памятью и выделения памяти. В ядре значительные объемы памяти выделяются на ограниченный объем объектов, таких как дескриптор файлов, inode и другие. Идея Бонвика базируется на том, что количество времени необходимого для инициализации обычного объекта объекта в ядре превышает количество времени, которое необходимо для его выделения и освобождения. Его идея заключается в том, чтобы вместо того, чтобы возвращать освободившуюся память системе, можно оставлять ее в проинициализированном состоянии, рассчитывая на то, что в ближайшем будущем она будет использоваться для этих же целей.

В случае распределения SLAB участки памяти подходящие для размещения объектов данных определенного типа и размера определяются заранее. Распределение (аллокатор) SLAB хранит информацию о размещении этих участков, которые известны так же как кэши. За счет того, что

выделяются блоки памяти одного размера, ускоряется выделение памяти. Аллокатор запрашивает у ОС большой кусок памяти и далее из него выделяет небольшие куски. Обращение к менеджеру

памяти происходит все реже, а запросы пользователя удовлетворяются быстрее. Но куда важнее не скорость выделения, а сокращение времени на инициализацию памяти. Это связано с тем, что аллокатор использует для выделения объектов одного и того же типа, что позволяет пропускать

инициализацию некоторых полей и структур при повторном выделении такого же участка памяти, например: мьютексы, спинлоки, inode-ы и тому подобное при освобождении объекта скорее всего имеют правильное значение и скорее всего при повторном выделении не нуждаются в повторной инициализации полей.

Билет 18

Специальные файлы устройств, каталог `/dev`, старший и младший номера устройств. Структура `usb_driver`: функции `probe()` и `disconnect()`, параметры и возвращаемое значение. Обработчики аппаратных прерываний: регистрация. Верхняя и нижняя половины обработчиков прерываний. Примеры тасклета и очереди работ.

Специальные файлы устройств

Устройства идентифицируются старшим и младшим номером устройства. Устройства

- символьные - `c`
- блочные - `b`
- сетевые

Отличие специальных файлов от обычных

По сравнению с обычными файлами устройства имеют три дополнительных атрибута, которые характеризуют устройство, соответствующее данному файлу:

1. **Класс(тип) устройства.** В ОС Linux различают следующие типы
 - а. **Блочные** устройства, например, *жесткий диск*, передают данные блоками. Обозначаются '`b`'
 - б. **Байт-ориентированные** (или **символьные**) устройства, например, *принтер* и *модем*, передают данные посимвольно, как непрерывный поток байтов. Взаимодействие с блочными устройствами может осуществляться лишь через буферную память, а для символьных устройств буфер не требуется. Обозначаются '`c`'
 - с. **Небуферизованные байт-ориентированные устройства.** Обозначаются как '`u`'
 - д. **Именованные каналы (FIFO).** Для блок-ориентированных и байт-ориентированных устройств (`b`, `c`, `u`) нужны и старший и младший номера, для именованных каналов номера не используются. Но про эти номера дальше. Обозначаются как '`p`'
2. **Старший номер устройства** идентифицирует **драйвер устройства**, например, *жесткий диск* или *звуковая плата*. Текущий список старших номеров устройств можно найти в файле `/usr/include/linux/major.h`. Вот небольшая выдержка из этого списка

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

3. **Младший номер устройства** идентифицирует конкретное устройство. Иными словами применяется для нумерации устройств одного типа, т. е. устройств с одинаковыми старшими номерами.

младший номер отражает конкретное устройство например жесткий диск (устройство и у него 1 старший номер) но разделы диска будут в системе иметь младшие номера в результате каждый раздел диска будет иметь 2 номера старший и младший.

Взято из https://www.linuxcenter.ru/lib/books/kostromin/gl_09_02.phtml

/dev - каталог в системах типа UNIX, содержащий специальные файлы (прим: фраза взята из вики, еще там есть дописка, что 'специальные файлы - интерфейсы работы с драйверами ядра' - писать на свой страх и риск).

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

Старшие номера известных ядру устройств можно увидеть, выполнив команду

```
[user]$ cat /proc/devices
```

Для идентификации устройства в системе имеется тип dev_t (POSIX1), определенный в <sys/types.h>. Формат полей и их содержание не оговаривается. Для разных версий системы формат отличается. для 32-битной: 12 бит - старший, 20 бит - младший. Старший и младший номера устройств можно получить с помощью макросов, что избавляет от необходимости задумываться как хранятся эти два номера:

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

Прим: хз, зачем структура stat, но она была в pdfке

```
struct stat {
    dev_t st_dev;
    ino_t st_ino;
    mode_t st_mode;
    ...
    dev_t st_rdev;
}
```

st_dev - описывает устройство, на котором находится файл - идентификатор устройства

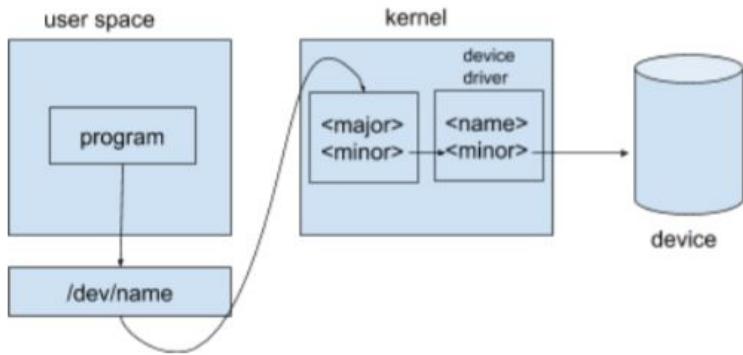
st_mode - S_ISREG, S_ISDIR, S_ISCHR, S_ISBLK, S_ISFIFO, S_ISLINK, S_ISSOCK

st_rdev - устройство, которое представляет этот файл (inode)

Прим: про makedev тоже не вижу смысла писать, но окей

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

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



Структура `usb_driver`: функции `probe()` и `disconnect()`, параметры и возвращаемое значение.

Примечание: по подфке `usb_driver` дается в связке с `usb_device`, поэтому здесь я приведу обе структуры и пояснения в том числе и к `usb_device`. Если что взято со страницы 40

Каждое подключенное к машине usb устройство представляется usb-core структурой `usb_device`.

Комментарии:

- `int devnum` - номер устройства на usbшине
- `char devpath[16]` - путь к файлу
- `enum usb_device_state state` - состояние устройства
- `struct usb_host_endpoint ep0` - данные первой конечной точки
- `char * product` - версия устройства
- `char * manufacturer` - производитель
- `char * serial` - серийный номер

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)

Прим: не нашел в подфке определение `USB CORE`, поэтому оставляю здесь выдержку из вики:

USB Core — это подсистема ядра `Linux`, созданная для поддержки `USB`-устройств и контроллеров шины `USB`. Цель её создания — абстрагирование от аппаратной реализации стандарта `USB` (либо аппаратно-зависимых функций) путём определения набора структур данных, макросов и функций.

Основная структура `usb_driver`, которая заполняется драйвером:

Прим: часть полей взята с https://www.opennet.ru/base/dev/write_linux_driver.txt.html, а часть с http://dmilvdv.narod.ru/Translate/LDD3/ldd_writing_usb_driver.html#probe_disconnect_detail

Комментарии:

- `name` - имя драйвера, уникальное. Такое же как имя модуля.
- `id_table` - это массив структур `usb_device_id`. Этот список предназначен для определения соответствия подключаемого устройства определенным параметрам.

Только те устройства, которые соответствуют перечисленным параметрам, могут быть подключены к драйверу. Если массив пуст, система будет пытаться подключить каждое устройство к драйверу. Определение структуры `usb_device_id` можно видеть в `include/linux/mod_devcitable.h`

- `driver` говорит о том, что `usb_driver` унаследован от `device_driver`.
- `struct file_operations *fops` - Указатель на `struct file_operations`, которую этот драйвер определил, чтобы использовать для регистрации в качестве символьного устройства. см [Главу 3](#) для получения дополнительной информации об этой структуре.
- `mode_t mode` - Режим для файла `devfs`, который будет создан для этого драйвера; иначе неиспользуемый. Типичный установкой для этой переменной будет значение `S_IRUSR` в сочетании со значением `S_IWUSR`, которыми владелец файла устройства предоставит доступ только для чтения и записи.
- `int minor_base` - Это начало установленного младшего диапазона для этого драйвера. Все устройства, связанные с этим драйвером, создаются с уникальными, увеличивающимися младшими номерами, начиная с этого значения.

Точки входа:

- `probe`
- `disconnect`
- `suspend`
- `resume`

[Подробнее про probe и disconnect](#)

`probe` и `disconnect` - это callback-функции, вызываемые системой при подключении и отключении USB-устройства. Вызов `probe` фактически означает регистрацию устройства в драйвере, а вызов `disconnect` - удаление устройства.

Вызывается для запроса существования конкретного устройства, если драйвер может с ним работать.

Если драйвер готов работать с устройством `probe` возвращает 0. Использует `usb_set_intfdata` чтобы связать с интерфейсом. Эта функция принимает указатель на любой тип данных и сохраняет его в структуре `struct usb_interface` для последующего доступа. Для получения данных должна быть вызвана функция `usb_get_intfdata`. `usb_get_intfdata` обычно вызывается в функции `open` USB драйвера и снова в функции `disconnect`. Если драйвер не соответствует устройству или не готов, то возвращается ENODEV.

Входные параметры

- у `probe`
- указатель на структуру `usb_interface`

- указатель на структуру `usb_device_id`. Определение структуры `usb_device_id` можно видеть в `include/linux/mod_devcitable.h` -

https://github.com/torvalds/linux/blob/master/include/linux/mod_devcitable.h

В самом простом случае содержит пару идентификаторов:

- идентификатор производителя (Vendor ID)
- идентификатор устройства (Device ID).

```
static int my_driver_probe(struct usb_interface* interface, const struct usb_device_id* id)
{
    printk("My moudle probe\n");
}
```

- у disconnect указатель на структуру `usb_interface`

```
static void my_driver_disconnect(struct usb_interface* interface)
{
    printk("My driver disconnect\n");
}
```

Аппаратные прерывания

Все **прерывания** от устройств ввода/вывода и системного таймера **аппаратные**. Все аппаратные прерывания выполняются на высоком уровне приоритета, и другие операции не могут выполняться пока аппаратное прерывание не будет завершено. Аппаратные прерывания делятся на две части:

- top half
- bottom half

В windows эта же идея реализована в виде DPC - отложенный вызов процедуры.

Аппаратные прерывания принято делить на быстрые и медленные.

Быстрые не делятся на две половины, выполняются как единое целое. В современном Linux быстрым прерыванием является только прерывание от системного таймера.

Нижняя половина прерывания выполняется как отложенное действие. Фактически аппаратным прерыванием является верхняя половина. В ее задачи входит получение данных из регистра устройства и помещение их в буфер ядра. Завершается аппаратное прерывание инициализацией отложенного действия, например путем постановки нижней половины в очередь на выполнение. Процесс должен получить информацию об успешном завершении операции. С помощью соответствующих функций драйвер должен инициировать передачу данных из буфера ядра в буфер приложения, для этого процесс должен быть разблокирован.

Медленные прерывания top half <--> bottom half

Обработчик прерывания делится на 2 части:

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

Завершение обработки прерывания выполняется нижней половиной

Задачи верхней половины:

1. Так как обработчик верхней половины выполняется при запрещенных прерываниях, то обработчик возвращает управление обычным return

2. 2. Должна обеспечить последующее выполнение нижней половины (поставить ее в очередь на выполнение)

Драйвер может иметь один обработчик прерывания. Если устройство использует прерывания, то драйвер устройства регистрирует один обработчик прерывания. Для обработки прерывания драйвер может разрешить определенную линию прерываний IRQ:

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

- **irq** - номер, прерывания, который будет обрабатывать обработчик.
- **handler** - указатель на функцию обработчика прерывания, который обслуживает прерывание
- **irqflags** - битовая маска:
 - NULL
 - IRQF_SHARED - на одну линию прерывания можно зарегистрировать несколько обработчиков
 - IRQF_PROBE_SHARED - устанавливается вызывающим, когда предполагается возможное возникновение проблем при использовании линии IRQ
 - IRQF_TIMER - маскирует прерывание как прерывание от таймера
 - IRQF_NOBALANCING - запрещает использовать это прерывание для балансировки IRQ
- **dev_name** - текст, представляющий устройство, связанное с этим прерыванием.(Используется в /proc/irq и /proc/interrupts) **dev_id** - используется для разделения SHARED линии прерывания. возвращаемое значение - • 0 - OK • не 0 - ошибка 28 29 • E_BUSY - данная линия прерывания уже используется и не указан флаг IRQF_SHARED.

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

- **irq** - численное значение линии прерывания, которую обслуживает обработчик.
- **dev_id** - общий указатель на тот же самый dev_id, что задается в request_irq, когда обработчик был зарегистрирован.
- **regs** - структура, содержащая регистры процессора и состояние перед обслуживанием прерывания. Обычно используется для отладки

Возвращаемое значение:

- **IRQ_NONE** - обработчик прерывания обнаруживает прерывание, которое его устройство не инициировало;
- **IRQ_HANDLER** - обработчик прерывания был вызван корректно и его устройство действительно вызвало прерывание.

пример тасклета

```
#include <linux/module.h> // module_init, ...
#include <linux/kernel.h> // printk
#include <linux/init.h> // __init, __exit
#include <linux/interrupt.h>
#include <linux/timex.h>

MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Pandoral");

#define SHARED_IRQ 17
static int irq = SHARED_IRQ, my_dev_id, irq_counter = 0;
module_param(irq, int, S_IRUGO);

char tasklet_data[] = "tasklet_function was called";

void tasklet_function( unsigned long data );

DECLARE_TASKLET( my_tasklet, tasklet_function,
                (unsigned long)&tasklet_data );

/* Bottom Half Function */
void tasklet_function( unsigned long data ) {
    printk(KERN_INFO "[TASKLET] state: %ld, count: %d, data: %s",
           my_tasklet.state, my_tasklet.count, my_tasklet.data);
    return;
}
```

```
static irqreturn_t my_interrupt( int irq, void *dev_id ) {
    irq_counter++;
    printk( KERN_INFO "[INTERRUPT] In the ISR: counter = %d\n", irq_counter );
    tasklet_schedule( &my_tasklet );
    return IRQ_NONE; /* we return IRQ_NONE because we are just observing */
}

static int __init my_tasklet_init(void) {
    if( request_irq( irq, my_interrupt, IRQF_SHARED, "my_interrupt", &my_dev_id ) )
        return -1;
    printk( KERN_INFO "[INTERRUPT] Successfully loading ISR handler on IRQ %d\n", irq );
    printk(KERN_INFO "[MODULE] Module is now loaded.\n");
    return 0;
}

static void __exit my_tasklet_exit(void) {
    /* Stop the tasklet before we exit */
    tasklet_kill( &my_tasklet );
    synchronize_irq( irq );
    free_irq( irq, &my_dev_id );
    printk( KERN_INFO "[INTERRUPT] Successfully unloading, irq_counter = %d\n", irq_counter );
    printk(KERN_INFO "[MODULE] Module is now unloaded.\n");
    return;
}

module_init(my_tasklet_init);
module_exit(my_tasklet_exit);
```

пример очереди работ:

```
#include <linux/module.h> // module_init, ...
#include <linux/kernel.h> // printk
#include <linux/init.h> // __init, __exit
#include <linux/interrupt.h>
#include <linux/workqueue.h>

MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("MY");

#define SHARED_IRQ 17
static int irq = SHARED_IRQ, my_dev_id, irq_counter = 0;
module_param(irq, int, S_IRUGO);

struct workqueue_struct *wq;
void hardwork_function(struct work_struct *work);

DECLARE_WORK(hardwork, hardwork_function);

/* Bottom Half Function */
void hardwork_function(struct work_struct *work) {
    printk(KERN_INFO "[WQ] data: %d", work->data);
    return;
}

static irqreturn_t my_interrupt( int irq, void *dev_id ) {
    irq_counter++;
    printk( KERN_INFO "[INTERRUPT] In the ISR: counter = %d\n", irq_counter )
    queue_work( wq, &hardwork );
    return IRQ_NONE; /* we return IRQ_NONE because we are just observing */
}
```

```

static int __init my_wokqueue_init(void) {
    if( request_irq( irq, my_interrupt, IRQF_SHARED, "my_interrupt", &my_dev_id ) )
        return -1;
    printk( KERN_INFO "[INTERRUPT] Successfully loading ISR handler on IRQ %d\n", irq );
    wq = create_workqueue( "my_queue" );
    if ( wq ) {
        printk(KERN_INFO "[MODULE] Workqueue created.\n");
    }

    printk(KERN_INFO "[MODULE] Module is now loaded.\n");
    return 0;
}

static void __exit my_wokqueue_exit(void) {
    flush_workqueue( wq );
    destroy_workqueue( wq );
    synchronize_irq( irq );
    free_irq( irq, &my_dev_id );
    printk( KERN_INFO "[INTERRUPT] Successfully unloading, irq_counter = %d\n", irq_counter );
    printk(KERN_INFO "[MODULE] Module is now unloaded.\n");
    return;
}

module_init(my_wokqueue_init);
module_exit(my_wokqueue_exit);

```

Билет 19

Управление устройствами: абстракция и типы устройств и идентификация в Unix/Linux. Драйверы и обработчики прерываний в Linux. USB-шина: особенности, usb-core, хост и конечные точки, 4 типа передачи данных. Структура USB-драйвер (struct usb_driver), таблица id_table, основные точки входа драйвера USB, передаваемые им параметры. Регистрация usb-драйвера в системе. Пример.

Типы устройств:

- символные
- блочные
- (сетевые)

Блочные и символьные рассматриваются системой как специальные файлы, в ОС такие файлы обозначаются С и В, отдельно выделяют сетевые. Это было в лекции про буферы, в случае если она захочет конкретики: При рассмотрении различных методов буферизации необходимо учитывать существование двух типов устройств, а именно блок ориентированных (в которых инфа хранится блоками) и байт ориентированных (в которых инфа хранится байтами, а передача информации выполняется не структурированным потоком байтов). Здесь слово неструктурированный подчеркивает что нет смыслового объединения байтов (байт за байтом идет). К блок ориентированным устройствам относятся диски. К байт ориентированным

относятся все остальные (принтеры , мышь коммуникационные порты , терминалы, джойстики, а также большинство устройств которые не являются внешними запоминающими устройствами).

Идентификация в Unix/Linux:

Если в каталоге /dev выполнить команду ls -l, то получим перечень адресов и два номера: первый - старший номер, идентифицирует драйвер устройства; второй - младший идентифицирует конкретное устройство. Для идентификации устройства в системе имеется тип **dev_t**. Формат полей и их содержание не оговаривается. Для разных версий системы формат отличается. для 32-битной: 12 бит - старший, 20 бит - младший. Старший и младший номера устройств можно получить с помощью макросов, что избавляет от необходимости задумываться как хранятся эти два номера.

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

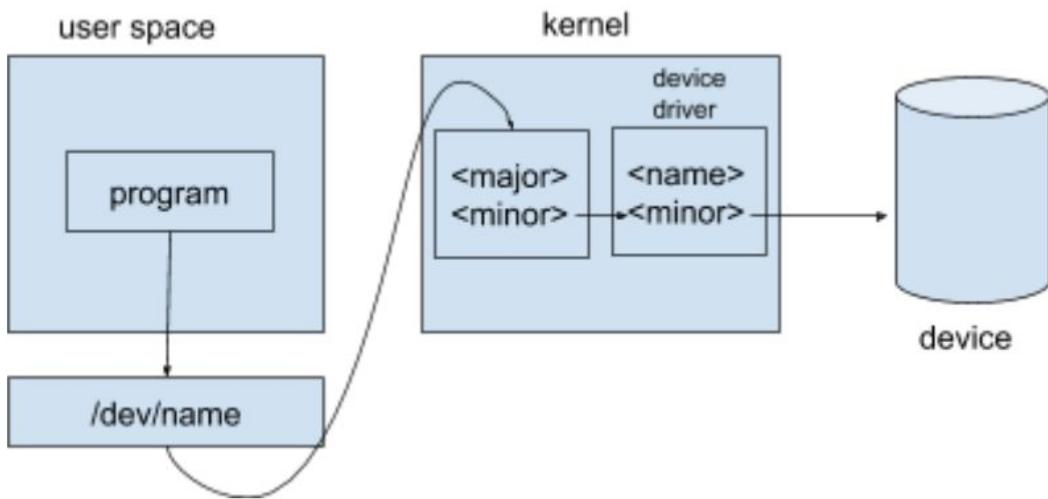
```
struct stat {
    dev_t st_dev;      //описывает устройство, на
котором находится файл - идентификатор
устройства
    ino_t st_ino;
    mode_t st_mode;   //S_ISREG, S_ISDIR, S_ISCHR, S_ISBLK, S_ISFIFO,
S_ISLINK, S_ISSOCK
    ...
    dev_t st_rdev;    //устройство, которое
представляет этот файл (inode)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

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

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

В билете про stat писать не нужно как мне кажется, на случай если спросит, то stat содержит информацию о файле: устройство и т.д.

Картинка лишняя (не давала на лекции)



Драйверы и обработчики прерываний в Linux.

Драйвер - программа или часть кода ядра, предназначенная для управления конкретным устройством. Обычно, драйверы устройств содержат последовательность команд, специфических для данного устройства. Имеют разные точки входа в зависимости от действий, которые выполняются на устройстве.

Драйверы Linux:

1. Драйверы, встроенные в ядро. Автоматически распознаются системой и становятся доступными приложению:

- материнская плата
- последовательные и параллельные порты
- контроллеры IDE

2. Драйверы, реализованные как загружаемые модули ядра используются для управления устройствами

- SCSI адаптеры

- звуковые и сетевые карты Файлы модулей ядра располагаются в /lib/modules. При установке системы задается перечень модулей, которые будут подключены автоматически на этапе загрузки. Список хранится в /etc/modules. В .conf находится перечень опций для таких модулей

3. Драйверы, код которых поделен между ядром и специальной утилитой.

- у драйвера принтера ядро осуществляет взаимодействие с параллельным портом, а демон печати lpd осуществляет формирование управляющих сигналов для принтера, используя для этого специальную программу.

- драйвера модемов

Любое устройство, подключенное к системе регистрируется ядром и ему присваивается / выделяется специальный дескриптор в виде структуры (**не давала структуру на лекции**)

struct device

struct device_driver (на лекциях была структура usb_driver)

Прерывания в Linux:

Драйвер может иметь один обработчик прерывания. Если устройство использует прерывания, то драйвер устройства регистрирует один обработчик прерывания. Для обработки прерывания драйвер может разрешить определенную линию прерываний IRQ:

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

irq - номер, прерывания, который будет обрабатывать обработчик.

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

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

- ❖ **irq** - численное значение линии прерывания, которую обслуживает обработчик.
- ❖ **dev_id** - общий указатель на тот же самый dev_id, что задается в request_irq, когда обработчик был зарегистрирован.
- ❖ **regs** - структура, содержащая регистры процессора и состояние перед обслуживанием прерывания. Обычно используется для отладки.
- ❖ **возвращаемое значение:**
 - **IRQ_NONE** - обработчик прерывания обнаруживает прерывание, которое его устройство не инициировало;
 - **IRQ_HANDLER** - обработчик прерывания был вызван корректно и его устройство действительно вызвало прерывание. irqflags - битовая маска либо NULL
 - **IRQF_SHARED** - на одну линию прерывания можно зарегистрировать несколько обработчиков
 - **IRQF_PROBE_SHARED** - устанавливается вызывающим, когда предполагается возможное возникновение проблем при использовании линии IRQ
 - **IRQF_TIMER** - маскирует прерывание как прерывание от таймера
 - **IRQF_NOBALANCING** - запрещает использовать это прерывание для балансировки
 - **IRQ dev_name** - текст, представляющий устройство, связанное с этим прерыванием.(Используется в /proc/irq и /proc/interrupts) dev_id - используется для разделения SHARED линии прерывания. возвращаемое значение -
 - **0** - OK
 - **не 0** - ошибка
 - **E_BUSY** - данная линия прерывания уже используется и не указан флаг IRQF_SHARED.

Про прерывания еще много чего написано, текста и так очень много, я не уверен, что целесообразно сюда кидать это, инфа на страницах 29-35 из красивой методы на 50 страниц. Перед выходом на ринг стоит удостоверится, что все понятно из этих страниц.

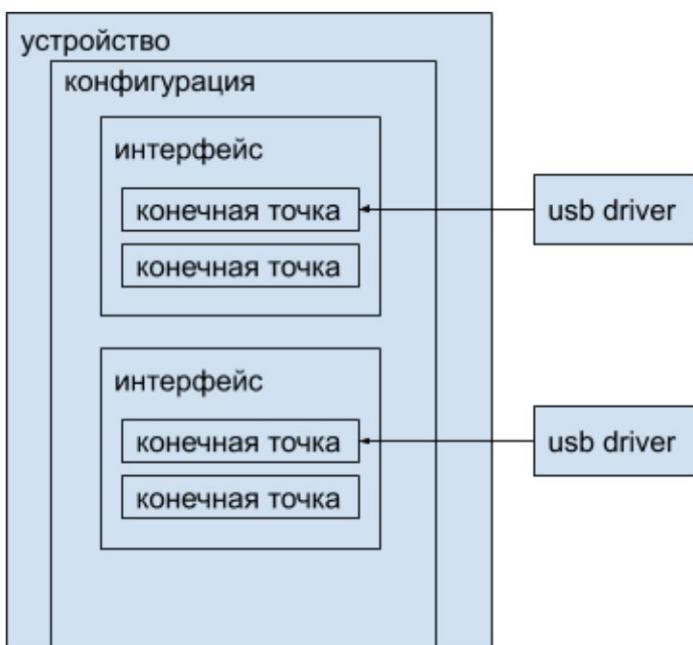
USB-шина: особенности, **usb-core**, хост и конечные точки, 4 типа передачи данных:

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

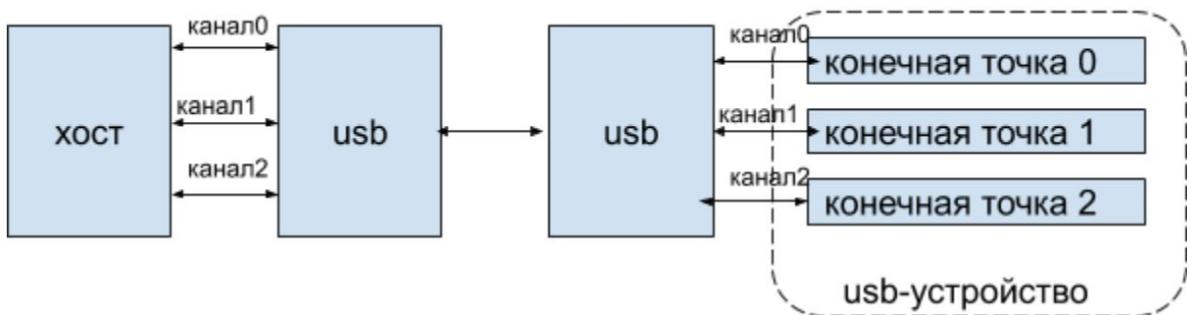
usb шина является шиной, в которой главной является **host** (что такое host может спросить она, это контроллер. Как говорит интернет, usb (Universal Serial Bus) является интерфейсом (не программным), который обеспечивает связь между внешними устройствами и контроллером host).

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 id 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 (данные, передаваемые в data packet)
- handshake (финальный статус транзакции)
- SOF - start of frame

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

1. Control (управляющий)
2. Interrupt (прерывания)
3. Bulk (поточний?)
4. Isochronous (изоХронный)

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

- ❖ 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.

Структура USB-драйвер (struct usb-driver), таблица id_table, основные точки входа драйвера USB:

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

name - имя драйвера, уникальное. Такое же как имя модуля точки входа - probe, disconnect, suspend, resume. Если драйвер готов работать с устройством probe возвращает 0. Использует usb_set_infdata чтобы связать с интерфейсом. Если драйвер не соответствует устройству или не готов, то возвращается ENODEV.

id_table - поле struct usb_device_id. struct usb_driver формирует интерфейс. Драйвер использует **id_table** чтобы поддерживать “горячее подключение” (HotPlug is adding components without stopping or shutting down the system. With the appropriate software installed on the computer, a user can plug such components without rebooting.).

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.

Регистрация usb-драйвера в системе. Пример:

Вызывается функция usb_register, в которую передается указатель на заполненную структуру

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

id_table:

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

После заполнения таблицы надо вызвать макрос

MODULE_DEVICE_TABLE(usb, pen_table)

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

Для привязки драйвера к устройству вызывается функция `usb_register_dev`

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

Для отключения связи драйвера с устройством используется

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

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

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

Пример:

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

Вызов `usb_register_dev` выполняется в функции `probe`
device имеет тип `struct usb_device *`

```

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

```

В disconnect вызывается функция usb_deregister_dev

```

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

```

Билет 20

Файловая подсистема /proc: тип и предоставляемая информация. Загружаемые модули ядра: правила программирования и сборки. Взаимодействие модулей ядра. Пример. Перемешивание данных между ядром и пользовательским пространством: функции. Функция printk() и строковый код приоритета(макросы). Пример (лаб.раб.).

Файловая подсистема /proc:

Обозначение /proc. ВФС создается "на лету" - в процессе обращения к системе. При обращении к информации в файловой системе (ФС), ФС предоставляет соответствующую информацию в режиме пользователя. ФС /proc создавалась для предоставления информации о файлах процессам и пользователям. Поскольку это ФС работа в ней выполняется с файлами, следовательно, используется стандартный интерфейс ФС и системных вызовов, следовательно, работа с информацией ФС proc осуществляется с помощью прав доступа, обычных для файлов - read, write, execute. По умолчанию r/w разрешены для владельцев. В proc для каждого процесса существует поддиректория, которая имеет имя - pid процесса: /proc/<pid>. Так процесс может получить свой id: getpid(), getppid() - id предка.

Загружаемые модули ядра: правила программирования и сборки:

Unix - система с монолитным ядром. Чтобы добавить новые функции такое ядро нужно перекомпилировать, что не безопасно. Поэтому есть средства для внесения изменений без

перекомпиляции. В UNIX/Linux - загружаемые модули ядра (ЗМЯ). Рассмотрим структуру простейшего ЗМЯ на примере Hello world.

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

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

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

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

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

Надеюсь никто не забудет в MODULE_AUTHOR написать свое имя))

MODULE_AUTHOR("Name"), MODULE_DESCRIPTION("LAB"), MODULE_LICENSE("GPL") макросы установки информации о модуле. Обязательный модуль лицензии - он сообщает ядру под какой лицензией находится данный код, и это влияет на то, какие функции и какие символы ядра этот код может использовать. GPL - данный модуль может иметь доступ ко всем символам ядра, без всяких уточнений - GNU Public License (v 2 и выше).

Структура ЗМЯ. Обязательны модули **init** и **exit**. Мы передаем им свои функции инициализации модулей. **printk** - те же действия, что и printf, но это функция ядра. Эта функция пишет сообщение в системный журнал (изначально буфер ядра, а из него данные могут быть прочитаны демоном протоколирования). В printk на первое место выносится константа, определяющая уровень протоколирования сообщения, или же приоритет. Существует 8 приоритетов.

Сборка:

```
obj-m := hello_m.o

KDIR = /lib/modules/$(shell uname -r)/build

default:
    make -C $(KDIR) M=$(shell pwd) modules
clean:
    make -C $(KDIR) M=$(shell pwd) clean
```

Взаимодействие модулей ядра:

После компиляции модуля, его можно загрузить командой insmod(выгрузить rmmod). ЗМЯ могут взаимодействовать друг с другом по данным, по функциям. Каждый отдельный модуль это отдельный файл.

Пример:

md1.c

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

MODULE_LICENSE( "GPL" );

char *md1_data = "md1 message";

extern char* md1_proc(void)
{
    return md1_data;
}

static char* md1_local(void)
{
    return md1_data;
}

extern char* md1_noexport(void)
{
    return md1_data;
}

EXPORT_SYMBOL(md1_data);
EXPORT_SYMBOL(md1_proc);

static int __init md_init( void )
{
    printk("+ module md1 start\n");
    return 0;
}

static void __exit md_exit( void )
{
    printk("- module md1 unload\n");
}
module_init( md_init );
module_exit( md_exit );

extern : (многомодульные программы) - означает "внешнее". export -
экспортируется во вне. proc - функция, которая будет использоваться
другими модулями.
```

md2.c

```
MODULE_LICENSE( "GPL" );
```

```
static int __init md_init( void )
{
    printk("+ module md2 start\n");
    printk("+ message exported: %s\n", md1_data);
    printk("+ message returned: %s\n", md1_proc());
    printk("+ message returned noexport: %s\n", md1_noexport());
    printk("+ message returned local: %s\n", md1_local());
    return 0;
}

static void __exit md_exit( void )
{
    printk("- module md2 unload\n");
}
module_init( md_init );
module_exit( md_exit );
```

md.h

```
extern char* md1_data;
extern char* md1_proc(void);
static char* md1_local(void);
extern char* md1_noexport(void);
```

Функции **md1_local** и **md1_noexport** не экспортируются, так как **md1_local** - не помечена, как внешняя функция и не экспортирована, а **md1_noexport** - не экспортирована. Вызов этих функций в модуле md2 вызовет ошибку и модуль не будет загружен.

md3.c

```

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

MODULE_LICENSE( "GPL" );

static int __init md_init( void )
{
    printk("+ module md3 start\n");
    printk("+ message exported: %s\n", md1_data);
    printk("+ message returned: %s\n", md1_proc());
    return -1;
}

static void __exit md_exit( void )
{
    printk("- module md3 unload\n");

}

module_init( md_init );
module_exit( md_exit );

```

Функция init модуля md3 возвращает -1, то есть ошибку, поэтому модуль не будет загружен.

obj-m := md1.o md2.o md3.o

KDIR = /lib/modules/\$(shell uname -r)/build

default:

make -C \$(KDIR) M=\$(shell pwd) modules

clean:

make -C \$(KDIR) M=\$(shell pwd) clean

Makefile:

obj-m := md1.o md2.o md3.o

KDIR = /lib/modules/\$(shell uname -r)/build

default:

make -C \$(KDIR) M=\$(shell pwd) modules

clean:

make -C \$(KDIR) M=\$(shell pwd) clean

Перемешивание данных между ядром и пользовательским пространством: функции:

Чтобы передать информацию из режима пользователя в режим ядра и обратно используются функции: `copy_from_user()` и `copy_to_user()`. Библиотека `linux/uaccess.h`.

Функция `printk()` и строковый код приоритета(макросы):

`printk` - выводит информацию в файл журнала: `/var/log/messages` . Содержимое файла журнала можно просматривать командой `dmesg`.

уровень	константа	назначение
7	KERN_DEBUG	отладочное сообщение
6	KERN_INFO	информационное сообщение
5	KERN_NOTICE	уже не информационное, но еще не предупреждение
4	KERN_WARNING	предупреждение
3	KERN_ERR	ошибка
2	KERN_KRIT	критическая ошибка
1	KERN_ALERT	предупреждение о скором крахе системы
0	KERN_EMERG	система упала

Пример:

`fortune.c`

```
MODULE_LICENSE("GPL");

ssize_t fortune_read(struct file *file, char *buf, size_t count, loff_t *f_pos);
ssize_t fortune_write(struct file *file, const char *buf, size_t count,
                     loff_t *f_pos);
int fortune_init(void);
void fortune_exit(void);

struct file_operations fops;

char *fortune_buf, *process_buffer;
struct proc_dir_entry *proc_entry;
unsigned int read_index;
unsigned int write_index;

// Чтение предсказаний cat fortune
ssize_t fortune_read(struct file *file, char *buf, size_t count, loff_t *f_pos)
{
    int len;

    if (write_index == 0 || *f_pos > 0)
    {
```

```
    return 0;
}

if (read_index >= write_index)
{
    read_index = 0;
}

copy_to_user(buf, &fortune_buf[read_index], count);
len = strlen(&fortune_buf[read_index]) + 1;
read_index += len;
*f_pos += len;

return len;
}

// Запись предсказания echo "message" >> fortune
ssize_t fortune_write(struct file *file, const char *buf, size_t count, loff_t *f_pos)
{
    int free_space = (FORTUNE_BUF_SIZE - write_index) + 1;

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

    if (copy_from_user(&fortune_buf[write_index], buf, count))
    {
        return -EFAULT;
    }

    write_index += count;
    fortune_buf[write_index - 1] = 0;

    return count;
}

// Инициализация
int fortune_init(void)
{
    fortune_buf = vmalloc(FORTUNE_BUF_SIZE); // выделение памяти

    if (!fortune_buf)
    {
        printk(KERN_INFO "Not enough memory for the fortune pot.\n");
        return -ENOMEM;
    }
}
```

```
// Переопределение операций чтения и записи
fops.owner = THIS_MODULE;
fops.read = fortune_read;
fops.write = fortune_write;

process_buffer = vmalloc(2 * FORTUNE_BUF_SIZE); // здесь пометка внизу
if (!process_buffer) {
    vfree(buffer);
    printk(KERN_INFO "Can't vmalloc process_buffer\n");
    return -ENOMEM;
}

memset(fortune_buf, 0, 2*FORTUNE_BUF_SIZE);
proc_entry = proc_create_data("fortune", 0666, NULL, &fops, NULL);

if (!proc_entry)
{
    vfree(fortune_buf);
    printk(KERN_INFO "Cannot create fortune file.\n");
    return -ENOMEM;
}

read_index = 0;
write_index = 0;

proc_mkdir("my_dir_fortune", NULL);
proc_symlink("symbolic_link_fortune", NULL, "/proc/fortune");

printk(KERN_INFO "Fortune module loaded.\n");

return 0;
}

void fortune_exit(void)
{
    remove_proc_entry("fortune", NULL);

    if (fortune_buf)
    {
        vfree(fortune_buf);
    }

    printk(KERN_INFO "Fortune module unloaded.\n");
}

module_init(fortune_init);
module_exit(fortune_exit);
```

