1. Билет №1

Управление внешними устройствами: специальные файлы устройств, адресация внешних устройств и их идентификация в системе, тип dev_t. Система прерываний: типы прерываний и их особенности. Прерывания в последовательности ввода-вывода — обслуживание запроса процесса на ввод-вывод (диаграмма). Быстрые и медленные прерывания. Обработчики аппаратных прерываний: регистрация в системе — функция и ее параметры, примеры. Тасклеты — объявление, планирование (пример лаб. раб).

1.1. Специальные файлы устройств

Специальные файлы устройств обеспечивают унифицированный доступ к периферийным (внешним) устройствам. Устройства, как и всё в UNIX, представлены файлами, что обеспечивает доступ к устройствам, как к обычным файлам — можно открывать/закрывать/читать/писать. В отличие от обычных файлов, специальные файлы устройств в действительности являются только указателями на соответствующие драйверы устройств в ядре.

> *Драйвер устройства — программа, управляющая работой внешнего устройства со стороны системы.*

Файлы устройств связывают файловую систему с драйверами устройств. Каждому устройству соответствует хотя бы 1 специальный файл. Обычно они лежат в /dev корневой ФС. Подкаталоги /dev/fd содержат файлы с именами 0, 1, 2 (в некоторых системах stdio, stdout, stderr...). Со всеми устройствами система работает одинаково.

Использование СФУ для работы с аппаратной частью:

Если процесс пользователя открывает для чтения обычный файл, то системные вызовы open и read обрабатываются встроенными в ядро функциями open и read соответственно. Если файл является специальным, то будут вызваны подпрограммы open и read, определенные в соответствующем драйвере устройства.

После того, как по inode ФС распознает, что данный файл является специальным, ядро использует старший номер специального файла как индекс в конфигурационной таблице драйверов устройств. Элементом таблицы является структура, элементы которой содержат указатели на функции соответствующего драйвера. Младший номер передается драйверу, поддерживающему несколько устройств, как дополнительный параметр, указывающий конкретное устройство.

Имеется 2 типа файлов устройств:

1. Символьные — небуферизуемые (non-buffered).
2. Блочные — буферизуемые (устройства вторичной памяти).

Связь имени специального файла с конкретным внешним устройством обеспечивает inode: есть поле dev_t i_rdev.

Тип специального файла задает поле mode. Для специального блок-ориентированного файла в этом поле устанавливается маска 06000, для байт-ориентированного 020000.

На структуры, описывающие устройства, ссылается inode:

```
1  struct inode {
2      ...
3      union {
4          ...
5          struct block_device     *i_bdev;
6          struct cdev         *i_cdev;
7          ...
8      };
9      ...
10 } __randomize_layout;
```

Таким образом, структуры block_device (для блочных устройств) и cdev (для символьных устройств) определены в файловой системе Linux.

```
1  struct cdev {
2      ...
3      struct module *owner;
4      const struct file_operations *ops;
5      struct list_head list;
6      dev_t dev;
7      ...
8  } __randomize_layout;
```

Символьные устройства для регистрации в системе (нужных операций) используют структуру file_operations. При создании символьного устройства в данной структуре нужно заполнить только два поля: ops и owner (обязательное значение - THIS_MODULE).

```
1  struct block_device {
2      dev_t              bd_dev;
3      ...
4      struct inode *         bd_inode;
5      struct super_block *     bd_super;
6      ...
7      struct gendisk *     bd_disk; // определяет(
           устройствоспециализированный ( интерфейс), ссылаетсяна  struct
           block_device_operations)
8      ...
9  } __randomize_layout;
```

Чтобы сделать блочные устройства доступными для ядра, драйверы блочных устройств регистрируются в ядре с помощью функции register_blkdev, но с версии 2.6.0 этот вызов необязателен и нужен только для динамического выделения старшего номера и создания записи в /proc/devices.

1.2. Адресация внешних устройств и их идентификация в системе, тип dev_t

Устройство — специальный файл, не можем идентифицировать его как обычный файл. В ядре используется тип dev_t (<linux/types.h>) для того, чтобы определять (содержать)

номера устройств. С версии 2.6.0 dev_t 32-разрядный. Представляет число, в котором 12 бит — для старшего номера, 20 — для младшего. POSIX.1 определяет существование типа, но не его формат.

Пример: жёсткий диск — устройство. Дисковое адресное пространство поделено на разделы (используются для монтирования ФС). Диск будет иметь старший номер (major id) — идентификатор класса устройства, а его разделы — младший номер (minor id).

Код пользователя не должен делать предположений о внутренней организации номера устройства, а должен просто использовать набор макросов <linux/kdev_t.h>, используемых для получения старшего и младшего номеров устройства.

```
1  MAJOR(dev_t dev);
2  MINOR(dev_t dev);
```

Для преобразования в dev_t по старшему и младшему номерам используется макрос:

```
1  MKDEV(int major, int minor);
```

До версии 2.6.0 количество номеров было ограничено: до 255 младших и до 255 старших. Позже ограничения были сняты.

Файлы устройств одного типа имеют одинаковые номера и различаются по номеру, который добавляется в конец имени.

Пример: все файлы сетевых плат имеют номера Ethernet: eth0, eth1...

Если внутри /dev вызвать ls -l, увидим два числа через запятую — старший и младший номера устройств. Если выполнить cat /proc/devices — увидим старшие номера устройств, известные ядру.

1.3. Выделение/освобождение номеров устройств

Одно из первых действий, которое должен сделать драйвер при установке, например, символьного устройства, — получение 1 или более номеров устройств для работы с ними.

Для выделения символьного устройства нужно вызвать функцию

```
1  int register_chrdev_region(dev_t first, unsigned int count, char* name);
```

first — начальный номер диапазона устройств, которые мы хотим выделить (minor). К нему нет требований.

count — количество запрашиваемых номеров устройств. Если count > запрашиваемого диапазона, то может перейти на следующий старший номер. Все будет работать до тех пор, пока запрашиваемый диапазон доступен.

name — имя номера устройства, связанного с диапазоном. Можно увидеть в /proc/devices и sysfs.

Возвращает 0, если успех, отрицательное число, если ошибка.

Функция эффективна, если заранее известно конкретное устройство, которому нужен номер. Но часто неизвестно, какой старший номер использует конкретное устройство.

Есть тенденция по переходу на использование динамически разделяемых номеров устройств:

```
1  int alloc_chrdev_region(dev_t* dev, unsigned int first_minor, unsigned
       int count, char* name);
```

dev — только выходной параметр, содержит первый номер в выделенном диапазоне при успехе. firstminor — первый младший номер (нужен для идентификации устройства).

Для освобождения используется:

```
1  int unregister_chrdev_region(dev_t first, unsigned int count);
```

1.4. Статическое выделение номеров устройств

Некоторые старшие номера назначаются большинству обычных устройств статически (см. Documentation/devices.txt).

Старший номер	Тип устройства
1	Оперативная память
2	Дисковод floppy
3	1-ый контроллер жёстких IDE-дисков
4	Терминалы
5	Терминалы
6	Принтеры (параллельный код)
8	Жёсткие SCSI-диски
13	Мышь
14	Звуковые карты
22	2-ой контроллер жёстких IDE-дисков

1.5. Система прерываний: типы прерываний и их особенности

Система прерываний включает в себя:

1. Системные вызовы (синхронные — возникают в процессе выполнения программы, вызываются соответствующей командой).

2. Исключения (синхронные — возникают в процессе выполнения программы, переполнение стека/деление на 0...).

3. Аппаратные прерывания (асинхронные — не зависят ни от каких действий в системе, выполняются на высоких уровнях приоритетов, их выполнение нельзя прервать, задача — информирование процессов о событиях в системе, от системного таймера/клавиатуры...).

Про таблицы прерываний:

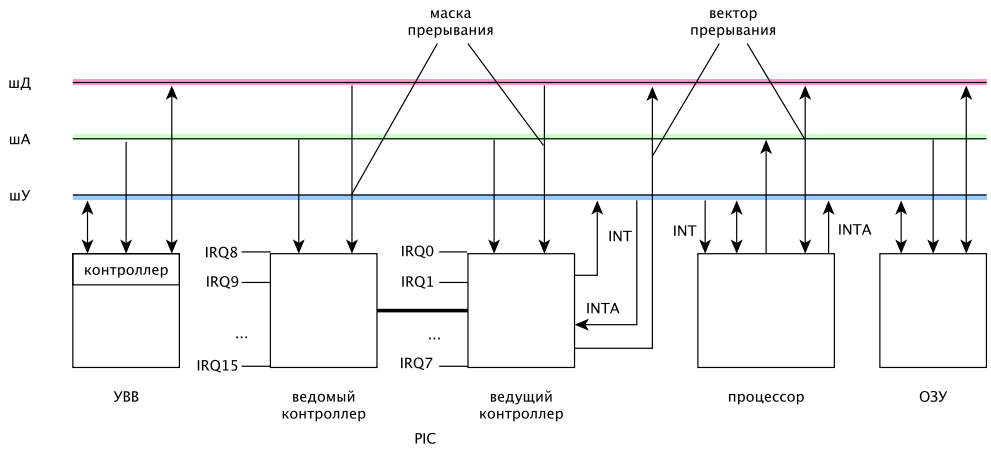
В 16-разрядной ОС существует таблица векторов прерываний — таблица, содержащая векторы прерывания (far-адреса обработчиков прерывания). Она начинается с нулевого адреса, занимает 1 Мб. Вектор прерывания - смещение в этой таблице.

В 32-разрядной ОС существует IDT (Interrupt Descriptor Table), нужная для получения адресов обработчиков прерываний, содержащая 8-байтовые дескрипторы прерываний, хранящие информацию о смещении к обработчику прерывания в соответствующем сегменте.

В 64-разрядной ОС существует IDT и список прерываний — запутанная система.

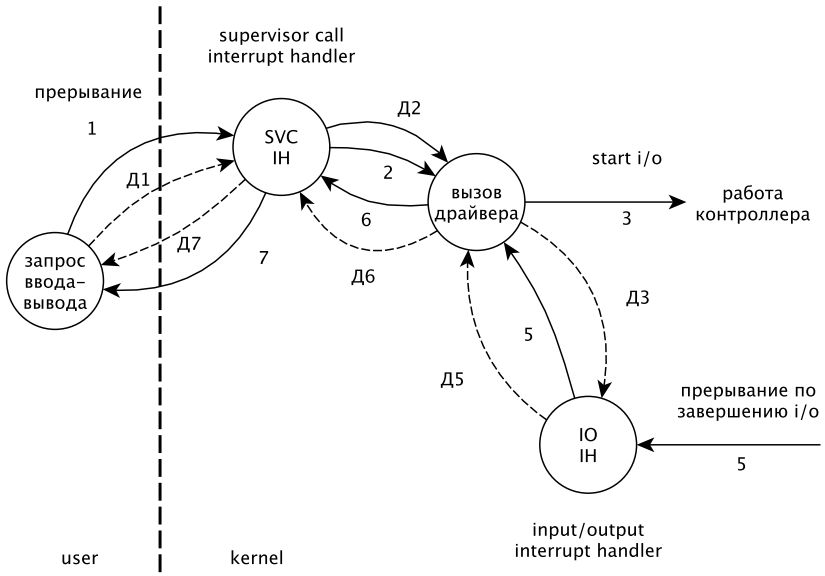
В современных системах часть перываний относится к APIC, а часть к подсистеме ОС MSI (Message Signal Interrupts).

1.6. Прерывания в последовательности ввода-вывода — обслуживание запроса процесса на ввод-вывод



Взаимодействие компьютера с внешними устройствами выполняется с помощью аппаратных прерываний — идея распараллеливания действий, когда управление внешним устройством берёт на себя контроллер устройства (замена опроса готовности ВУ).

1. Запрос приложения на ввод/вывод переводит систему в режим ядра.
2. Подсистема ввода/вывода вызывает драйвер устройства (в драйвере есть 1 обработчик прерывания от данного устройства).
3. По окончании операции ввода/вывода контроллер устройства формирует сигнал прерывания, приходящий на соответствующую линию прерывания контроллера прерываний.
4. Контроллер формирует сигнал INT, который по ШУ идёт на выделенную ножку процессора.
5. В конце цикла выполнения каждой команды (выборка-дешифрирование-выполнение) процессор проверяет наличие сигнала на ножке. Если есть сигнал, процессор выставляет на ШУ INTA.
6. Контроллер отправляет по ШД вектор прерывания в регистры процессора.
7. Процессор смещается относительно нулевого адреса и находит в таблице адрес обработчика.



1.7. Быстрые и медленные прерывания

Выделяются быстрые и медленные аппаратные прерывания. Быстрые — выполняются атомарно, не делятся на части, в современных системах это только прерывание от системного таймера. Медленные — все остальные, делятся на 2 части.

Любые АП выполняются на самом высоком уровне приоритета (самый высокий — системный таймер, выше него только migration для перебалансировки нагрузки между процессорами и power при падении напряжения — система отключается от питания и не может работать).

Аппаратные прерывания — важнейшие в системе, поэтому, чтобы исключить в многопроцессорных системах ошибки обработки данных, поступающих от внешних устройств, на том ядре, на котором обрабатывается прерывание, запрещаются все остальные прерывания, а в системе запрещаются все прерывания по данной линии IRQ. Никакая другая работа на данном процессоре выполняться не может, что негативно сказывается на производительности и отзывчивости системы, поэтому АП не могут выполняться длительное время. Поэтому работа, которую выполняют обработчики прерываний, делится на 2 части.

Эти 2 части — top_half и bottom_half. top_half (interrupt handler) копирует данные устройства в специальный буфер, инициализирует отложенное действие bottom_half (softirq, tasklet или workqueue) и завершается. top_half должен выполниться быстро, так как при этом игнорируются остальные прерывания. bottom_half выполняется при разрешённых остальных прерываниях и делает всю остальную обработку прерывания.

1.8. Обработчики аппаратных прерываний: регистрация в системе — функция и ее параметры, примеры

Основная задача обработчика прерывания — передача данных по шине данных устройства, если была запрошена операция чтения. Даже если была запрошена операция записи, устройство все равно оправляет по шине данных информацию об успешности операции. Чтобы обработать ее, обработчик должен сохранять информацию, поступающую от устройства в буфер ядра. Затем эта информация поступит приложению, запросившему ввод/вывод.

Системная функция для регистрации обработчика прерывания:

```
1  typedef irqreturn_t (*irq_handler_t)(int, void *);
2
3  int request_irq(unsigned int irq, irq_handler_t handler, unsigned long
       flags, const char* name, void *dev);
```

irq — номер линии прерывания. handler — обработчик прерывания. flags — флаги. name — имя устройства/обработчика. dev - указатель на некоторый объект, используется для освобождения линии прерывания от указанного обработчика:

```
1  const void *free_irq(unsigned int irq, void *dev_id);
```

Пример:

```
1  rc = request_irq(IRQ_NUMBER, interrupt_handler, IRQF_SHARED, "
       tasklet_interrupt_handler", interrupt_handler);
```

Флаг IRQF_SHARED указывает, что данная линия прерывания может быть разделена несколькими обработчиками. IRQF_TIMER — флаг, маскирующий данное прерывание как прерывание от таймера. IRQF_PROBE_SHARED — устанавливается абонентами, если возможны проблемы при совместном использовании линии.

1.9. Тасклеты — объявление, планирование

Тасклет — специальный тип softirq. Тасклеты представлены двумя типами отложенных прерываний: HI_SOFTIRQ и TASKLET_SOFTIRQ.

Тасклет не может выполняться параллельно, выполняется атомарно на том же процессоре, на котором выполняется обработчик вызвавшего его прерывания. Инициализируется как статически, так и динамически. Выполняется в кон-

тексте прерывания — нельзя использовать средства взаимоисключения, кроме spinlocks. Выполняется ksofirqd. Является упрощённым интерфейсом softirq.

Определена структура:

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

Тасклеты в отличие от softirq могут быть зарегистрированы как статически, так и динамически. Статически тасклеты создаются с помощью двух макросов (man 6.2.1):

```
#define DECLARE_TASKLET(name, _callback)      \
struct tasklet_struct name = {          \
  .count = ATOMIC_INIT(0),        \
  .callback = _callback,          \
  .use_callback = true,         \
}
#define DECLARE_TASKLET_DISABLED(name, _callback) \
struct tasklet_struct name = {          \
  .count = ATOMIC_INIT(1),        \
  .callback = _callback,          \
  .use_callback = true,         \
}
```

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

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

Эта строка эквивалентна следующему объявлению:

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

В данном примере создается тасклет с именем my_tasklet, который разрешен для выполнения. Функция tasklet_handler будет обработчиком этого тасклета. Поле dev отсутствует в текущих ядрах.

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

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

Пример:

```
1 tasklet_init(t, tasklet_handler, data);
```

Тасклеты должны быть зарегестрированы для выполнения. Тасклеты могут быть запланированы на выполнение функциями:

```
1 tasklet_schedule(struct tasklet_struct *t);
2 tasklet_hi_sheduler(struct tasklet_struct *t);
```

Когда тасклет запланирован, ему выставляется состояние TASKLET_STATE_SCHED, и тон добавляется в очередь. Пока он находится в этом состоянии, запланировать его еще раз не получится, т.е. в этом случае просто ничего не произойдет. Тасклет не может находиться сразу в нескольких местах очереди на планирование, которая организуется через поле next структуры tasklet_struct. После того, как тасклет был запланирован, он выполниться только один раз.

Свойства:

1. Если вызывается функция tasklet_schedule(), то после этого тасклет гарантированно будет выполнен на каком-либо процессоре хотя бы один раз.

2. Если тасклет уже запланирован, но его выполнение все еще не запущено, он будет выполнен только один раз.

3. Если этот тасклет уже запущен на другом процессоре (или schedule вызывается из самого тасклета), выполнение переносится на более поздний срок.

4. Тасклет строго сериализован по отношению к самому себе, но не по отношению к другим тасклетам. Если разработчик считает, что в данном тасклете нужно выполнить действия, которые могут выполняться в других таскоетах, он реализует взаимоисключение с помощью spinlocks.

Критерии	softirq	tasklet	workqueue
Определение	10 шт. определено в ядре статически	С + Д	С + Д
Сериализация	Да	Нет	Да
Возможность паралл. вып.	Да	Нет	Да
Выполнение	В контексте спец. потоков ядра (ksoftirqd)	В контексте запланировавшего обраб. прерыв.	В контексте спец. потоков ядра (kworker)
Блокируются	???	Нет	Да

1.10. spin_lock

Race condition — условия гонок: процессы выполняются с разной скоростью и пытаются получить доступ к разделяемым переменным.

Аппаратная реализация взаимоисключения — test_and_set. Атомарная функция, реализующая как неделимое действие проверку и установку значения в памяти. По сути, читает значение переменной b, копирует его в a, устанавливает b значение true. Считается, что это не приведет к бесконечному откладыванию.

Использование test_and_set в цикле проверки значения переменной называется циклической блокировкой. Spin-блокировки — активное ожидание на процессоре (время непроизводительно расходуется на проверку флага другого процессора). Реализация test_and_set связана с блокировкой локальной шины памяти, в результате один поток может занять шину на длительное время, что понижает отзывчивость (решение — 2 вложенных цикла, если переменная занята — выполняется обычный цикл без блокировки шины).

```
1  void spin_lock(spin_lock_t* c)
2  {
3    while (test_and_set(*c) != 0)
4    {
5      /* ресурсзанят  */
6
7      /*
8      while (*c != 0)
9      {
10     ...
11     }
```

```
12          */
13      }
14    }
15
16    void spin_unlock(spin_lock_t* c)
17    {
18      *c = 0;
19    }
```

Циклическую блокировку может удерживать только 1 поток. Захваченная — в состоянии contended. Если другой поток пытается захватить уже захваченную, то блокировка находится в состоянии конфликта, а поток выполняет цикл проверки busy_loop. Блокировка должна быть связана с тем, что она блокирует. Запрещаются данные (разделяемый ресурс), а не код.

Пример из лабораторной:

```
1  #include <linux/kernel.h>
2  #include <linux/module.h>
3  #include <linux/interrupt.h>
4  #include <linux/slab.h>
5  #include <asm/io.h>
6
7  #include "ascii.h"
8
9  #define IRQ_NUMBER 1
10
11 MODULE_LICENSE("GPL");
12 MODULE_AUTHOR("Karpova Ekaterina");
13
14 struct tasklet_struct *tasklet;
15 char tasklet_data[] = "key pressed";
16
17 void tasklet_func(unsigned long data)
18 {
19      printk(KERN_INFO "+: ----------------------");
20      printk(KERN_INFO "+: tasklet began");
21      printk(KERN_INFO "+: tasklet count = %u", tasklet->count.counter);
22      printk(KERN_INFO "+: tasklet state = %lu", tasklet->state);
23
```

```
24      printk(KERN_INFO "+: key code - %d", data);
25      if (data < ASCII_LEN)
26          printk(KERN_INFO "+: key press - %s", ascii[data]);
27      if (data > 128 && data < 128 + ASCII_LEN)
28          printk(KERN_INFO "+: key release - %s", ascii[data - 128]);
29
30      printk(KERN_INFO "+: tasklet ended");
31      printk(KERN_INFO "+: ------------------------");
32  }
33
34  static irqreturn_t my_irq_handler(int irq, void *dev_id)
35  {
36    int code;
37    printk(KERN_INFO "+: my_irq_handler called\n");
38
39    if (irq != IRQ_NUMBER)
40    {
41      printk(KERN_INFO "+: irq not handled");
42      return IRQ_NONE;
43    }
44
45    printk(KERN_INFO "+: tasklet state (before schedule) = %lu",
46                    tasklet->state);
47    code = inb(0x60);
48    tasklet->data = code;
49    tasklet_schedule(tasklet);
50    printk(KERN_INFO "+: tasklet scheduled");
51    printk(KERN_INFO "+: tasklet state (after schedule) = %lu",
52                     tasklet->state);
53
54    return IRQ_HANDLED;
55  }
56
57  static int __init my_init(void)
58  {
59    if (request_irq(IRQ_NUMBER, my_irq_handler, IRQF_SHARED, "
        tasklet_irq_handler", (void *) my_irq_handler))
60    {
```

```
61      printk(KERN_ERR "+: cannot register irq handler\n");
62      return -1;
63    }
64
65    tasklet = kmalloc(sizeof(struct tasklet_struct), GFP_KERNEL);
66
67    if (tasklet == NULL)
68    {
69          printk(KERN_ERR "+: kmalloc error");
70          return -1;
71    }
72
73    tasklet_init(tasklet, tasklet_func, (unsigned long)tasklet_data);
74
75    printk(KERN_INFO "+: module loaded\n");
76    return 0;
77  }
78
79  static void __exit my_exit(void)
80  {
81    tasklet_kill(tasklet);
82    free_irq(IRQ_NUMBER, my_irq_handler);
83
84    printk(KERN_INFO "+: module unloaded\n");
85  }
86
87  module_init(my_init);
88  module_exit(my_exit);
```