Как пользоваться файликом

ОЧЕНЬ ВАЖНО:

- Если пишем про прерывания – обязательно написать про адресацию аппаратных прерываний

- Если пишем про одну или несколько нижних половин (например, 1, 2, 3, 11, 18 билет) – обязательно расписать про демона ksoftirqd

- В пункте “нижние половины” есть 2 подпункта-сравнения: softirq vs tasklet и tasklet vs workqueue. Там очень важная инфа, и собрать ее нормально и понятно получается именно в сравнении. Если пишем про все нижние половины, то спокойненько пишем весь пункт (“нижние половины”) в том порядке, в котором там все расписано. Если же только про некоторые – обязательно дописываем соответствующее сравнение, даже если об этом не просят. То есть для softirq – softirq vs tasklet, для tasklet – softirq vs tasklet и tasklet vs workqueue, для workqueue - tasklet vs workqueue

Просто полезно знать:

- Откуда инфа про драйверы и USB: файл ZУправление… лежит в папке. Нашла файлик у Якубы. Сравнила с тем, что Х читала на лекции -- видимо, она читала как раз по нему, а в прошлом году просто скинула им файликом. Поэтому инфу буду брать оттуда

- В файле рассматривается 4 типа устройств:

  b — блок-ориентированное устройство; • c — байт-ориентированное (символьное) устройство; • u — не буферизованное байт-ориентированное устройство; • p — именованный канал.

  В лекции были только первые 2. Поэтому оставшиеся 2 привожу, но выделяю курсивом

- Многие структуры достаточно громоздкие. Подчеркивала те поля, что обсуждаются по тексту или особо важны. Писать ли остальное – хз

- Про Управление внешними устройствами: в билетах о них спрашивают в разных формулировках, в некоторых упускают один/несколько подпунктов, которые их касаются. Я уверена, что если пишешь про эту тему, то нужно написать все (не напишешь – спросят дополнительно, а нам это нужно?) Поэтому объединила все в один блок и даю везде ссылки на него целиком. Подпункты выделены, так что можно писать в принципе написать только то, что спрашивают (но не советую)
- АП = аппаратное прерывание

- По поводу тасклетов: у старшекуров можно увидеть немного другие функции. Это взято из методы (видимо, более старые версии). На лекции же нам давали то же, что я потом в исходном коде нашла. Оставляю на всякий оба, но из методы – курсивом

Освещаемые билеты

- Управление внешними устройствами, Система прерываний, верхние и нижние половины (все типы как раз со сравнением) - Билет 1, Билет 1 Старый, Билет 2, Билет 3 старый. Билет 11

- Управление устройствами: драйверы, 4 типа передачи данных. - Билет 5, Билет 5 старый. ,Билет 18 (не знаю год) Билет 19 (не знаю год),

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. (в билетете нет, но надо, иначе как перейти к тасклетам потом: Верхние и нижние половины обработчиков прерываний.) Обработчики аппаратных прерываний: регистрация в системе — функция и ее параметры, примеры.

Тасклеты — объявление, планирование (пример лаб. раб). *(можно пропустить пункт про softirq, дальше пишем все до очередей работ)*

Билет 1 старый

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. (в билетете нет, но надо, иначе как перейти к тасклетам потом: Верхние и нижние половины обработчиков прерываний.) Обработчики аппаратных прерываний: регистрация в системе, примеры.

Тасклеты — объявление, планирование (пример лаб. раб). *(можно пропустить пункт про softirq, дальше пишем все до очередей работ)*

Билет 2

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

Система прерываний: типы прерываний и их особенности. Быстрые и медленные прерывания. Верхние и нижние половины обработчиков прерываний. Обработчики аппаратных прерываний: регистрация в системе, примеры.*(можно пропустить пункт про прерывания в последовательности ввода-вывода)*

Нижние половины: тасклеты и очереди работ — объявление, создание, постановка работы в очередь, планирование (пример лаб. раб). *(можно пропустить пункт про softirq)*

Билет 3 старый

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 (это есть в описании верхних и нижних половин)* *(можно пропустить пункт про прерывания в последовательности ввода-вывода)*

softirg, тасклеты, очереди работ - особенности, сравнение, примеры (лабораторная работа).

Билет 11

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

Быстрые и медленные прерывания. пример быстрого прерывания, флаги. Нижняя и верхняя половины обработчиков прерываний: регистрация обработчика аппаратного прерывания, функция регистрации и ее параметры. *(пишем до конца пункта (то есть до следующего пункта “нижние половины”)*

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

Билет 5

Управление устройствами: абстракция устройств, типы устройств и идентификация в ядре Unix/Linux.

Управление устройствами: драйверы.

USB-шина: особенности, хост и хабы, конечные точки и каналы, 4 типа передачи данных.

Структура USB-драйвера (struct usb_driver), таблица id_table, основные точки входа драйвера USB. Регистрация usb-драйвера в системе

Билет 5 старый.

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.

Билет 18 (не знаю год)

Специальные файлы устройств, каталог /dev, старший и младший номера устройств.

(в билете нет, но надо бы): драйверы в целом

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

(в билете нет, но надо бы Быстрые и медленные прерывания.) Верхняя и нижняя половины обработчиков прерываний. Обработчики аппаратных прерываний: регистрация. *(можно пропустить пункт про прерывания в последовательности ввода-вывода)*

Примеры тасклета и очереди работ. *(надо бы и их концепцию вкратце рассказать, так что даю ссылку на более верхний подпункт, а там внутри примеры)*

Билет 19 (не знаю год)

Управление устройствами: абстракция и типы устройств и идентификация в Unix/Linux.

Драйверы и

обработчики прерываний в Linux
Не знаю, что именно хотят. Думаю, можно написать про Система прерываний: типы прерываний и их особенности., Быстрые и медленные прерывания., Верхняя и нижняя половины обработчиков прерываний., Регистрация обработчика АП в

системе — функция и ее параметры, примеры., *ну и* *про адресацию аппаратных прерываний*

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.

Управление внешними устройствами

Абстракция устройств

Следуя парадигме UNIX “в UNIX все – файл”, внешние устройства представляются в системе как специальные файлы и имеют inode (индексный дескриптор), который обеспечивает связь имени специального файла с устройством *(про них будет далее)*.

Специальные файлы устройств

Специальные файлы устройств обеспечивают единообразный доступ к внешним устройствам и обеспечивают связь между файловой системой и драйверами устройств. Специальные файлы устройств в действительности являются только указателями на соответствующие драйверы устройств в ядре. Такая интерпретация специальных файлов обеспечивает доступ к внешним устройствам как к обычным файлам. Так же как обычный файл, файл устройства может быть открыт, закрыт, в него можно писать или из него можно читать. Обычно эти файлы можно увидеть в каталоге /dev корневой ФС. Каждому внешнему устройству unix и линукс ставит в соответствие как минимум 1 специальный файл.

(не знаю, зачем, но Х тут упомянула): Подкаталог /dev/fd содержит файлы с именами 0, 1, 2. Но в некоторых ФС имеется /dev/stdin, /dev/stdout и /dev/stderr.

В юникс и линукс связь имени специального файла с блочным устройством обеспечивает inode (индексный дескриптор).

(Предложение выше – с лекции, следующее – из методы. Как-то они разнятся. Я бы вот это не писала, но на всякий приведу): В отличие от обычных файлов, специальные файлы устройств в действительности являются только указателями на соответствующие драйверы устройств в ядре.

По сравнению с обычными файлами файлы устройств имеют три дополнительных атрибута, которые характеризуют устройство, соответствующее данному файлу: класс устройства, старший и младший номера устройства (рассматриваются далее)

Класс устройства

! На лекции Х назвала это типом, хотя тип – это старший адрес

В Unix определены 2 класса:

1. Символьные (character) (байт-ориентированные) устройства, например, принтер и модем
    - передают данные посимвольно, как непрерывный поток байтов.
    - не буферизуемый (non-buffered)

    - при выводе в терминале обозначаются буквой “c”
2. Блочные (блок-ориентированные) устройства, например, жесткий диск,
    - передают данные блоками
    - буферизуемый (то есть взаимодействие с ними – лишь через буферную память)
    - при выводе в терминале обозначаются буквой “b”

Блочные - это внешние запоминающие, а все остальные - символьные.

Кроме этих двух классов устройств имеются еще два

- *небуферизованные байт-ориентированные устройства (что? байт-ориентированные же и так небуферизованные)*
- *именованные каналы (FIFO). (хотя вот это я бы упомянула)*

Адресация внешних устройств и их идентификация в системе

Для идентификации специальных файлов устройств в системе есть система идентификации (цитата))). Используется общий подход: символьные и блочные устройства представляются парой чисел: старший и младший номера устройств: major и minor address

Пример:

Если перейти каталог /dev и выполнить команду ls –l, то можно увидеть эти два числа, которые разделены запятой.

```
ls -l (вызов выполняется в /dev)
crw-rw-rw- 1 root root 1,3 <data> null
crw-------- 1 root root 10, 1 <data> psaux
crw-------- 1 root root 4,1 <data> tty1
crw-rw-rw- 1 root tty 4, 64 <data> ttys0
...
crw-rw-rw- 1 root root 1,6 <data> zero
```

Старший (основной, major) номер устройства

Обозначает тип устройства (например, жесткий диск или звуковая плата). Определяет драйвер, связанный с устройством, т.е. это номер драйвера.

В примере выше: устройства /dev/null и /dev/zero используют драйвер с номером 1. Виртуальные консоли и терминалы (то есть последовательные интерфейсы) (tty0, tty1) – драйвер 4.

Некоторые major номера зарезервированы для определенных устройств. Текущий список старших номеров устройств можно найти в файле /linux/major.h

Старший номер	Тип устройства
1	ОЗУ (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
13	мышь
14	Звуковые карты
22	Второй контроллер для жестких IDE-дисков

Другие major номера динамически присваиваются драйверам устройств, когда Linux загружается.

Чтобы посмотреть, какие старшие номера уже используются в текущей реализации линукс (то есть известны ядру), надо посмотреть /proc/devices.

На всякий случай прогнала, нам не давали вывод

```
$ cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
…

Block devices:
  7 loop
  8 sd
```

```
...
```

С помощью команды `ls -l /dev | grep "^c"` можно посмотреть список файлов символьных устройств системы.

На всякий случай прогнала, нам не давали вывод (^c == начинается с “c”, а c – как раз и означает специальный файл символьного устройства)

```
$ ls -l /dev | grep "^c"
crw-r--r--  1 root root     10,   235 Jun 10 11:42 autofs
crw-rw----  1 root disk     10,   234 Jun 10 11:42 btrfs-control
crw--w----  1 root tty       5,     1 Jun 10 11:42 console
...
crw-rw-rw-  1 root tty       5,     0 Jun 10 11:42 tty
crw--w----  1 root tty       4,     0 Jun 10 11:42 tty0
crw--w----  1 root tty       4,     1 Jun 10 11:42 tty1
```

На лекции звучал 1 вариант, в файлике – 2. как-то они разнятся в формулировке, не знаю, что лучше написать.

а) Современное ядро линукс позволяет множеству драйверов разделять старшие номера, но большинство все еще организованы по принципу: один старший номер= один драйвер.

б) Модуль драйвера устройства может зарегистрировать столько разных основных номеров, сколько он поддерживает, хотя обычно это не делается.

(Потом будет разобрано подробно, но можно упомянуть и тут): ориентируясь на выделенные номера можно выбрать номер для своего драйвера.

Младший (дополнительный, minor) номер устройства

Применяется для нумерации устройств одного типа, т. е. устройств с одинаковыми старшими номерами. Младший номер устройства используется самим драйвером, чтобы различать отдельные физические или логические устройства. Никакие другие части ядра не используют младший номер устройства.

тип dev_t

Для хранения номеров устройств, как старшего так и младшего, в ядре используется тип dev_t, определенный в <linux/types.h>.

```
typedef __kernel_dev_t dev_t;
```

Стандарт POSIX.1 определяет существование этого типа, но не оговаривает формат полей и их содержание. Начиная с версии ядра 2.6.0: dev_t является 32-х разрядным, 12 бит отведены для старшего номера и 20 - для младшего.

Для получения старшей или младшей части dev_t можно использовать макросы, которые позволяют получить старший и младший номера устройств:

```
<linux/kdev_t.h>
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

Если имеются старший и младший номера, то, наоборот, их необходимо преобразовать в dev_t, используя:

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

В ядре Linux определены аналогичные функции:

```
#include <sys/sysmacros.h>
dev_t makedev(unsigned int maj, unsigned int min);
unsigned int major(dev_t dev);
unsigned int minor(dev_t dev);
```

Можно сказать по формату, что с 2.6 ядра, система может поддерживать очень много устройств. Ранее было ограничение по 255 штук для старшего и младшего номеров.

Символьные и блочные устройства и их inode (структуры,описывающие символьные и блочные устройства).

inode

Следуя парадигме UNIX в UNIX все – файл, внешние устройства представляются в системе как специальные файлы и имеют inode, который содержит метаданные о файле. Приложения обращаются к символьным и блочным устройствам через inode. Когда создается inode устройства, он сопоставляется с номерами major и minor.
В struct inode имеются соответствующие поля:

```
struct inode {

      umode_t i_mode;
      ...
      const struct inode_operations *i_op; /* операции,
      определенные на inode*/
      struct super_block *i_sb;
      struct address_space *i_mapping;
      ...
      /* Stat data, not accessed from path walking */
      unsigned long i_ino; /*номер inode*/
      ...
      dev_t i_rdev; /*фактический номер устройства, содержащий
      major, minor (как st_rdev, который описан ниже)*/
      ...
      union {

            const struct file_operations *i_fop; /* former ->i_op-
            >default_file_ops */
            void (*free_inode)(struct inode *);

      };
      ...
      struct list_head i_devices;

      union { // вот как раз объединение – может храниться блочное
      устройство/символьное устройство/(что-то еще, что нас здесь
      не интересует)


            struct pipe_inode_info *i_pipe;
            struct block_device *i_bdev;
```

```
            struct cdev *i_cdev;
            char *i_link;
            unsigned i_dir_seq;

        };
        …

} __randomize_layout;
```

Приложение может извлечь метаданные из inode, используя системный вызов stat(), который возвращает структуру stat.

```
struct stat {

        dev_t st_dev; /* ID of device containing file */
        ino_t st_ino; /* Inode number */
        mode_t st_mode; /* File type and mode */
        nlink_t st_nlink; /* Number of hard links */
        uid_t st_uid; /* User ID of owner */
        gid_t st_gid; /* Group ID of owner */
        dev_t st_rdev; /* Device ID (if special file) */
        off_t st_size; /* Total size, in bytes */
        blksize_t st_blksize; /* Block size for filesystem I/O */
        blkcnt_t st_blocks; /* Number of 512B blocks allocated */
        /* Since Linux 2.6, the kernel supports nanosecond
        precision for the following timestamp fields.
        For the details before Linux 2.6, see NOTES. */
        struct timespec st_atim; /* Time of last access */
        struct timespec st_mtim; /* Time of last modification */
        struct timespec st_ctim; /* Time of last status change */
        #define st_atime st_atim.tv_sec /* Backward compatibility */
        #define st_mtime st_mtim.tv_sec
        #define st_ctime st_ctim.tv_sec

};
```

- *Поле st_dev - описывает устройство, на котором «живет» inode. Это устройство идентифицируется комбинацией его major идентификатором, который идентифицирует общий класс устройства, и minor идентификатором, который идентифицирует конкретный экземпляр в общем классе.*
- *Поле st_mode – определяет тип файла (обычный/директория/символьное устройство/…) и режим*

Мало ли, спросит

```
// вот нужные нам типы
S_IFBLK 0060000 block device
S_IFCHR 0020000 character device
//соответствующие макросы
S_ISCHR(m) character device?
S_ISBLK(m) block device?

// пример
stat(pathname, &sb);
if (S_ISCHR(sb.st_mode)) {
/* обрабатываем символьное устройство */
}
```

- *Поле st_rdev – устройство, представленное данным inode. Если этот файл, имеющий inode, представляет устройство, тогда inode содержит major и minor идентификаторы этого устройства.*

(то есть если inode описывает устройство, то вот разница: st_dev - устройство, где расположен сам inode, st_rdev – устройство, которое он описывает)

Структуры для символьных и блочных устройств

Для описания символьных и блочных устройств существуют дополнительные структуры, детализирующие особенности типов устройств, и на эти структуры ссылается inode.

Символьное устройство – struct cdev

```
struct cdev {

      struct kobject kobj;
      struct module *owner;
      const struct file_operations *ops;
      struct list_head list;
      dev_t dev; // номер устройства (эта структура обсуждалась выше)
      unsigned int count;

} __randomize_layout;
```

При создании символьного устройства в данной структуре нужно заполнить только два поля: file_operation и owner (Обязательное значение - THIS_MODULE)

Существует два способа выделения и инициализации структуры:
1) Runtime Allocation
2) Own allocation.

Если нужно получить автономную структуру cdev во время выполнения, то это можно сделать с помощью следующего кода:

```
struct cdev *my_cdev = cdev_alloc();
my_cdev->ops = &my_fops;
```

Или же можно встроить структуру cdev *(наверное, имелось в виду file_operations)* в собственную структуру устройства, используя следующую функцию:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

Как только структура cdev настроена с file_operations и owner, последний шаг - сообщить ядру об этом с помощью вызова:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

Где:

- dev - структура cdev,
- num - номер первого устройства, на которое отвечает это устройство,
- count - количество номеров устройств, которые должны быть связаны с устройством. Часто счет равен единице, но есть ситуации, когда имеет смысл иметь более одного номера устройства, соответствующего конкретному устройству.
- Если эта функция возвращает отрицательный код ошибки, то устройство не было добавлено в систему.

После вызова cdev_add () устройство сразу же активируется. Все функции, которые были определены (через структуру file_operations), могут быть вызваны.

Чтобы удалить устройство char из системы, нужно вызвать функцию:

```
void cdev_del(struct cdev *dev)
```

Блочное устройство - struct block_device

Драйвер блочного устройства обеспечивает доступ к устройствам, которые передают произвольно доступные данные блоками фиксированного размера, в первую очередь на дисках. Очевидно, что блочные устройства принципиально отличаются от символьных устройств, и эти особенности не могут не учитываться в ядре. Драйверы блочных устройств имеют соответствующий интерфейс, который определяется специфическими задачами и проблемами блочных устройств.

Можно сказать, что драйверы блочных устройств - это канал между памятью ядра и вторичным хранилищем; следовательно, они могут рассматриваться как составляющие подсистемы виртуальной памяти. Большая часть дизайна блочного слоя сосредоточена на производительности.

```
struct block_device {

      dev_t bd_dev; /* not a kdev_t - it's a search key */
      int bd_openers;
      struct inode * bd_inode; /* will die */
      struct super_block * bd_super;
      struct mutex bd_mutex; /* open/close mutex */
      void * bd_claiming;
      void * bd_holder;
      int bd_holders;
      bool bd_write_holder;
      #ifdef CONFIG_SYSFS
      struct list_head bd_holder_disks;
      #endif
      struct block_device * bd_contains;
      unsigned bd_block_size;
      u8 bd_partno;
      struct hd_struct * bd_part;
      /* number of times partitions within this device have been
      opened. */
      unsigned bd_part_count;
      int bd_invalidated;
      struct gendisk * bd_disk;
      struct request_queue * bd_queue;
      struct backing_dev_info *bd_bdi;
      struct list_head bd_list;
      /*
      * Private data. You must have bd_claim'ed the block_device
      * to use this. NOTE: bd_claim allows an owner to claim
      * the same device multiple times, the owner must take special
      * care to not mess up bd_private for that case.
      */
      unsigned long bd_private;
      /* The counter of freeze processes */
      int bd_fsfreeze_count;
      /* Mutex for freeze */
      struct mutex bd_fsfreeze_mutex;

} __randomize_layout;
```

Драйверы блочных устройств, также как и драйверы символьных устройств, должны использовать набор интерфейсов для регистрации, чтобы сделать блочные устройства
доступными для ядра. Концепции похожи, но детали регистрации блочных и символьных устройств различны.

Первым шагом, предпринимаемым большинством блочных драйверов, является регистрация себя в ядре:

```
<linux/fs.h>
int register_blkdev(unsigned int major, const char *name);
```

Аргументами данной функции являются:

- major - основной номер, который будет использовать устройство, Если major передается как 0, ядро выделяет новый главный номер и возвращает его вызывающей стороне.
- name – имя, которое ядро будет отображать в /proc/devices.
- отрицательное возвращаемое значение указывает, что произошла ошибка.

Начиная с ядра 2.6 вызов этой функции совершенно необязателен. Единственными задачами, которые выполняет этот вызовом в настоящий момент, являются: 1) выделение динамического старшего номера, если требуется, и 2) создание записи в /proc/devices.

Символьные устройства для регистрации в системе нужных операций используют структуру file_operations. Блочные устройства используют для этого struct block_device_operations *(вот такая иерархия: в struct block_device есть struct gendisk, в которой есть block_device_operations)*,

include/linux/blkdev.h:

```
struct block_device_operations {

      int (*open) (struct block_device *, fmode_t);
      void (*release) (struct gendisk *, fmode_t);
      int (*rw_page)(struct block_device *, sector_t, struct page
      *, unsigned int);
      int (*ioctl) (struct block_device *, fmode_t, unsigned,
      unsigned long);
      int (*compat_ioctl) (struct block_device *, fmode_t,
      unsigned, unsigned long);
      unsigned int (*check_events) (struct gendisk *disk,
      unsigned int clearing);

      /* ->media_changed() is DEPRECATED, use ->check_events()
      instead */
      int (*media_changed) (struct gendisk *);
      void (*unlock_native_capacity) (struct gendisk *);
      int (*revalidate_disk) (struct gendisk *);
      int (*getgeo)(struct block_device *, struct hd_geometry *);
      /* this callback is with swap_lock and sometimes page table
      lock held */
      void (*swap_slot_free_notify) (struct block_device *,
```

```
        unsigned long);
        int (*report_zones)(struct gendisk *, sector_t sector,

        struct blk_zone *zones, unsigned int *nr_zones, gfp_t
        gfp_mask);

        struct module *owner;
        const struct pr_ops *pr_ops;

};
```

На эту структуру ссылается struct gendisk (объявлена в <linux/genhd.h>), которая представляет в ядре отдельное дисковое устройство.

```
struct gendisk {

        /* major, first_minor and minors are input parameters only,
        * don't use directly. Use disk_devt() and disk_max_parts().
        */
        int major; /* major number of driver */
        int first_minor;
        int minors; /* maximum number of minors, =1 for disks that
        can't be partitioned. */
        char disk_name[DISK_NAME_LEN]; /* name of major driver */
        char *(*devnode)(struct gendisk *gd, umode_t *mode);
        unsigned int events; /* supported events */
        unsigned int async_events; /* async events, subset of all */
        /* Array of pointers to partitions indexed by partno.
        * Protected with matching bdev lock but stat and other
        * non-critical accesses use RCU. Always access through
        * helpers.
        */
        struct disk_part_tbl __rcu *part_tbl;
        struct hd_struct part0;
        const struct block_device_operations *fops;
        struct request_queue *queue;
        void *private_data;
        int flags;
        struct rw_semaphore lookup_sem;
        struct kobject *slave_dir;
        struct timer_rand_state *random;
        atomic_t sync_io; /* RAID */
        struct disk_events *ev;
        #ifdef CONFIG_BLK_DEV_INTEGRITY
        struct kobject integrity_kobj;
        #endif /* CONFIG_BLK_DEV_INTEGRITY */
```

```
        int node_id;
        struct badblocks *bb;
        struct lockdep_map lockdep_map;

};
```

Выделение и освобождение номеров устройств (devfs)

(у нас этого не было, но в файле много текста уделено. Сократила, как могла. Можно сразу перейти к подпунктам этого пункта)

Сама идея отображения устройств как специальных файлов хороша, но обычные Linux системы управляют ими далеко не оптимальным способом. В каталоге /dev "обитают" сотни специальных файлов для "презентации" соответствующих hardware. Большинство таких специальных файлов "don't even map" на реально существующее на вашей машине устройство.

Традиционный (без devfs) kernel-based device driver "прописывает" устройство в остальной части системы через системные вызовы register_blkdev() или register_chrdev(). Major номер передается как параметр этим функциям, и после регистрации устройства ядро знает, что этот конкретный major номер соответствует конкретному драйверу для устройства, которое выполнило вызов register_xxxdev().

А какой major номер разработчик драйвера должен использовать?

Проблемы нет, если developer драйвера не планирует его использования "внешним миром". В этом случае сгодится любой major номер, лишь бы он не конфликтовал с другими major номерами, используемыми в конкретном частном случае. Как альтернатива, разработчик может "динамически" ассигновать major номер перед register_???dev(). Однако, "по большому счету", такое решение приемлемо только в случае, если драйвер не предназначен для широкого использования.

Иначе разработчик драйвера должен войти в контакт с Linux kernel developers и получить для своего специфического устройства "официальный" major номер. После этого, для всех Linux пользователей это устройство будет связано с таким major номером.

- *В таком случае, когда device node (специальный файл) создается, он получит тот же major, как зарегистрирован во внутренних структурах ядра: ядро знает, с каким драйвером нужно связаться.*
- *Иначе, mapping от специального файла на kernel driver сделано по major номером, а не по именам устройств.*

Но проблема не только в этом. Linux подошел к границе, когда все "официальные" major и minor номера будут исчерпаны. Конечно, можно просто увеличить разрядность номеров. Но лучше – перейти на devfs.

После правильного конфигурирования devfs, суперпользователь перезагружает систему. Теперь device drivers для регистрации своих устройств используют улучшенный вызов: devfs_register(), в котором можно не указать major и minor номера. Вместо этого вызов devfs_register() передает path на устройство как параметр, и именно так оно впоследствии появиться под /dev.

Для блочных устройств

(Мы рассматривали только символьное, но привожу из файлика и блочное на всякий. Все аналогично символьным. Из интересного: нет struct file_operations. Почему? Мы уже рассмотрели структуры устройств, и выяснили, что в struct block_device есть struct gendisk, в которой есть block_device_operations)

```
int register_blkdev(unsigned int major, const char * name);
int unregister_blkdev(unsigned int major, const char *name);
```

Для символьных устройств

(этих 2 функций у нас не было)

```
#include <linux/fs.h>
int register_chrdev(unsigned int major, const char*name, struct
file_operations*ops);
int unregister_chrdev(unsigned int major, const char *name);
```

register_chrdev() связывает major номер символьного устройства с набором точек входа драйвера (=регистрирует устройство. Ему выделяется major номер)

- *Структура file_operations содержит указатели на функции, которые драйвер использует для реализации интерфейса ядра с драйвером.*
- *Параметр major - это major номер [1..255], назначаемый драйверу символьного устройства и сопоставляемый с таблицей функций. Если major = 0, то выделяется какой-либо неиспользованный номер major. Модуль драйвера устройства может зарегистрировать столько разных основных номеров, сколько он поддерживает, хотя обычно это не делается.*
- *Параметр name представляет собой краткое имя устройства и отображается в списке /proc/devices. Он также должен точно соответствовать имени, переданному функции unregister_chrdev() при освобождении функций.*

- *В случае успеха, register_chrdev возвращает 0, если major - это число, отличное от 0. В противном случае (если major == 0) Linux выберет старший*

номер и вернет выбранное значение. В случае ошибки возвращается один из следующих кодов:

- *-EINVAL Указанный номер недействителен (> MAX_CHRDEV)*
- *-EBUSY Основной номер занят*

Регистрация имени устройства создаёт соответствующую запись в файле /proc/devices, но не создает самого устройства в /dev

unregister_chrdev() освобождает старший номер и обычно вызывается в функции module_cleanup для удаления драйвера из ядра.

- *вернет 0 в случае успеха или -EINVAL, если основной номер не зарегистрирован с соответствующим именем.*

Одним из первых шагов, который необходимо сделать разрабатываемому драйверу при установке символьного устройства, является получение одного или нескольких номеров устройств для работы с ними.

Статически:

Одна из функций для выполнения этой задачи:

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

- first это - начало диапазона номеров устройств, который вы хотели бы выделить. Младшее число first часто 0, но не существует никаких требований на этот счёт.
- count - запрашиваемое общее число смежных номеров устройств. Заметим, что если число count большое, запрашиваемый диапазон может перекинуться на следующей старший номер, но всё будет работать правильно, если запрашиваемый диапазон чисел доступен.
- name - имя устройства, которое должно быть связано с этим диапазоном чисел; оно будет отображаться в /proc/devices и sysfs.

- Правило возвращаемого значения аналогично другим функциям ядра.

Функция register_chrdev_region хорошо работает, если заранее известно, какие именно номера устройств будут использоваться.

Например, можно посмотреть, какие номера уже используются в системе (как – мы рассмотрели выше), и на основе этого выбрать новый номер. Также можно назначить случайным образом и посмотреть, работает ли.

Проблема:

- часто неизвестно, какие старшие номера устройств будут использоваться.
- такое решение приемлемо только в случае, если драйвер не предназначен для широкого использования (то есть чисто на своей машине)

Динамически

Поэтому сообщество разработчиков ядра Linux прилагает постоянные усилия, чтобы перейти к использованию динамически выделяемых номеров устройств:

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

- dev является только выходным значением, которое при успешном завершении содержит первый номер выделенного диапазона.
- firstminor должен иметь значение первого младшего номера для использования; как правило, 0.
- Параметры count и name аналогичны register_chrdev_region.

Несмотря на то, как выполняется распределение номеров устройств в драйвере, их необходимо освободить, когда они больше не используются с помощью

```
void unregister_chrdev_region(dev_t first, unsigned int count)
```

Обычное место для вызова unregister_chrdev_region будет в функции module_cleanup или exit загружаемого модуля ядра.

Недостаток: – нельзя заранее создать узлы устройства, так как главный номер будет известен лишь позже. Решение: вызов insmod следует заменить скриптом, который после вызова insmod будет читать /proc/devices, чтобы создать специальный файл (файлы). Рекомендуется в скрипте использовать инструмент awk. Эта команда читает документ по строкам и выполняет указанные разработчиком действия, а результат выводит на стандартный вывод. Условий-действий может быть несколько

```
Синтаксис: awk опции "условие {действие}"
Пример: major = $(awk "\\$2 == \"$module\" {print \\$1}"
/proc/devices
```

Таким образом, разработчик должен выбрать: просто подобрать номер, которые кажется неиспользуемым, или выделить старшие номера динамически и использовать скрипт.

Все приведенные выше функции выделяют номера устройств для использования драйвером, но ничего не говорят ядру, как в действительности эти номера будут использоваться. Перед тем, как какая-либо программа из пространства пользователя сможет получить доступ к одному из этих номеров устройств, драйверу необходимо подключить их к своим внутренним функциям, которые осуществляют операции, связанные с устройством.

Пример работы с символьным устройством из файла:

```
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
...
dev_t dev = 0;
static int __init etx_driver_init(void)
{

      /*Allocating Major number*/
      if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
          printk(KERN_INFO "Cannot allocate major number\n");
          return -1;
      }
      printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev),
      MINOR(dev));

      /*Creating cdev structure*/
      cdev_init(&etx_cdev,&fops);
      etx_cdev.owner = THIS_MODULE;
      etx_cdev.ops = &fops;

      /*Adding character device to the system*/
      if((cdev_add(&etx_cdev,dev,1)) < 0){
          unregister_chrdev_region(dev,1);
          return -1;

       }
       ...
}
```

```
void __exit etx_driver_exit(void)
{
...
unregister_chrdev_region(dev, 1);
printk(KERN_INFO "Device Driver Remove...Done!!!\n");
}
```

Подключение нового устройства к системе

Если необходимо подключить к системе какое-то новое устройство, вначале следует проверить, что в каталоге /dev имеется специальный файл (или ссылка на специальный файл) для этого устройства. Специальные файлы устройств создаются с помощью команды mknod:

```
mknod [опции] <имя_устройства> <тип_устройства> <старший_номер> <младший_номер>
```

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

- b — блок-ориентированное устройство;
- c — байт-ориентированное (символьное) устройство;
- *u — не буферизованное байт-ориентированное устройство;*
- *p — именованный канал.*

Для блок-ориентированных и байт-ориентированных устройств (b, c, u) нужны и старший и младший номера, для именованных каналов номера не используются.

Пример: создается специальный файл для терминала, подключенного к порту COM3, который в Linux обозначается как /dev/ttyS2. Устройства-терминалы представляют собой байт-ориентированные устройства, которые имеют старший номер 4 и младшие номера, которые начинаются с 64.

```
mknod -m 660 /dev/ttyS2 c 4 66
```

(-m права: Устанавливает права доступа (тут чтение и запись владельцу и группе, ничего остальным)

Каталог /dev

!!!!Внимание! Этот пункт просят только в 18 билете неизвестного года. Если он – welcome. Иначе – лучше прочитать, но не писать. Ибо мы этого здесь не проходили, а вопросов вытекающих задать могут много)

/dev – это файловая система для устройств

Большинство устройств в Linux’е представлено как файлы в особой файловой системе. Эти файлы хранятся в каталоге /dev, куда к ним обращается система для выполнения задач, связанных с вводом/выводом.

Какие есть интересные поддиректории – мы уже писали в пунктах про старшие и младшие адреса.

Представление устройств: struct device, struct pci_device, struct usb_device

этот пункт на грани: скорее, он относится к следующей теме (ниже), но возможно, стоит и тут упомянуть. Поэтому вот ссылка на него

Драйверы

Драйверы

Драйвер – это часть кода ядра, которая предназначена для управления конкретным устройством. Он должен преобразовывать данные, передаваемые устройству к формату данных, определенному на устройстве, а данные, получаемые от устройства – к формату, понятному приложению.

Код драйвера пишется по жестким правилам системы, на основании структур ядра, определенных в системе. Такие структуры перечисляют точки входа в драйвер (и другие параметры).

Внести собственную функциональность в ядро виндов можно только с помощью написания драйвера. Написать драйвер грамотно может только разработчик устройства, так как только он знает формат всех передаваемых данных.

В Linux драйверы устройств бывают трех типов.

1. Встроенные (являются частью программного кода ядра).

   Соответствующие устройства автоматически обнаруживаются системой и становятся доступны для приложений. Выполнение их инициализируется при запуске системы. Обычно таким образом обеспечивается поддержка тех

устройств, которые необходимы для монтирования корневой файловой системы и запуска компьютера. Примеры: видеоконтроллер VGA, контроллеры IDE-дисков, материнская плата, последовательные и параллельные порты.

2. Драйверы второго типа представлены загружаемыми модулями ядра. Hid human interface drivers – взаимодействие с пользователем (мышь, клава). Они оформлены в виде отдельных файлов и для их подключения необходимо выполнить отдельную команду подключения модуля.

   Если необходимость в использовании устройства отпала, модуль можно выгрузить из памяти (отключить). Поэтому использование модулей обеспечивает большую гибкость, так как каждый такой драйвер может быть переконфигурирован без остановки системы. Если пишем для ввода мышью азбуки морзе, надо выгрузить сначала старый, и только потом установить собственный.

   Модули часто используются для управления такими устройствами как SCSI-адаптеры, звуковые и сетевые карты.

   Файлы модулей ядра располагаются в подкаталогах каталога /lib/modules. Обычно при инсталляции системы задается перечень модулей, которые будут автоматически подключаться на этапе загрузки. Список загружаемых модулей хранится в файле /etc/modules. А в файле /etc/modules.conf находится перечень опций для таких модулей. Редактировать этот файл "вручную" не рекомендуется, для этого существуют специальные скрипты (типа update-modules).

   Для подключения или отключения модулей в работающей системе есть модули(утилиты) lsmod, insmod, rmmod, modprobe (автоматически загружает модули). чтобы отобразить всю инфу modprobe -c

3. Код драйвера поделен между ядром и специальной утилитой, предназначенной для управления данным устройством. Например, для драйвера принтера ядро отвечает за взаимодействие с параллельным портом, а формирование управляющих сигналов для принтера осуществляет демон печати lpd, который для этого использует специальную программу filter. Другим примером этого типа являются драйвера модемов.

Во всех трех случаях непосредственное взаимодействие с устройством осуществляет ядро или какой-то модуль ядра. А пользовательские программы взаимодействуют с драйверами устройств через специальные файлы, расположенные в каталоге /dev и его подкаталогах. Таким образом, взаимодействие прикладных программ с аппаратной частью ядра под управлением ОС юникс/линукс осуществляется по следующей схеме (текст между стрелками – в квадратики)

устройство ‹-› ядро ‹-› драйвер ‹-› системный вызов, файловая система, специальные файлы ‹-› приложение

Такая схема обеспечивает единый подход ко всем устройствам, которые с точки зрения приложений выглядят как обычные файлы.

Представление устройств: struct device, struct pci_device, struct usb_device

На самом низком уровне каждое устройство в Linux представлено экземпляром struct device. Это базовая структура.

Поля заполняются при выключении некоторого устройства. (так сказала X, но я не уверена в истинности)

```
struct device {

	struct kobject kobj;
	…
	struct device *parent; // //устройство, к которому
	подключается новое устройство. Обычно шина или хост-
	контроллер
	const char *init_name; /* исходное название устройства */
	const struct device_type *type;
	struct bus_type *bus; /* важно, к какой шине подключается
	устройство */
	struct device_driver *driver; /* какой драйвер разместил это
	устройство */
	void *platform_data; /* Platform specific data, device
	core doesn't touch it */
	void *driver_data; /* Driver data, set and get with */
	...
	#ifdef CONFIG_GENERIC_MSI_IRQ_DOMAIN
	struct irq_domain *msi_domain;
	#endif
	#ifdef CONFIG_PINCTRL
	struct dev_pin_info *pins;
	#endif
	#ifdef CONFIG_GENERIC_MSI_IRQ
	struct list_head msi_list;
	#endif
	const struct dma_map_ops *dma_ops;
	u64 *dma_mask; /* dma mask (if dma'able device) */
	struct dma_coherent_mem *dma_mem; //direct memory access - это метод
```

```
        освобождения процессора от рутинной перекачки данных от устройства в
        оперативную память. При передаче блоков данных в оперативку, если нет
        dma, то все через регистры, и процессор загружен. Coherent DMA mapping
        API. В этом случае нет необходимости предварительно выделять буфер
        DMA, установка такого поля используется для устойчивого соединения
        многократно используемых драйверов.
        ...
        #ifdef CONFIG_NUMA
        int numa_node; /* NUMA node this device is close to */
        #endif
        dev_t devt; /* для создания в sys/fs девайса */
        u32 id; /* экземпляр устройства */
        …

};
```

Но ядро содержит набор подсистем таких как pci, pci express, usb. Большинство подсистем отслеживают дополнительную информацию об устройствах, поэтому устройства обычно представляются более детальными структурами (например, struct pci_device или struct usb_device), которые в себя включают саму struct device.

(Назвали, и хватит, но вот на всякий сами структуры)

Устройства, подключаемые к шине pci представляются структурой struct pci_dev, которая содержит строку struct device dev.

```
/* The pci_dev structure describes PCI devices */
struct pci_dev {

        struct list_head bus_list; /* Node in per-bus list */
        struct pci_bus *bus; /* Шина, на которой находится устройство
        */
        struct pci_bus *subordinate; /* Bus this device bridges to */
        ...
        unsigned short vendor; /*поставщик*/
        unsigned short device; /*устройство*/
        unsigned short subsystem_vendor;
        unsigned short subsystem_device;
        unsigned int class; /* 3 bytes: (base,sub,prog-if) */
        ...
        u8 pin; /* Interrupt pin this device uses */
        ...
        struct pci_driver *driver; /* Драйвер, связанный с конкретным
        устройством */
        ...
        struct device_dma_parameters dma_parms;
        ...
```

```
        struct device dev; /* универсальный интерфейс устройства */
        ...
        /*
        Вместо непосредственного использования линии прерывания и
        регистров базового адреса используйте значения, хранящиеся
        здесь. Они могут быть разными!
        */
        unsigned int irq;
        struct resource resource[DEVICE_COUNT_RESOURCE]; /* I/O and
        memory regions + expansion ROMs */
        ...
        pci_dev_flags_t dev_flags;
        ...
        unsigned long priv_flags; /* Private flags for the PCI driver
        */

};
```

Аналогично, struct pci_dev структура struct usb_dev содержит struct device dev.

```
/* USB_DT_DEVICE: Device descriptor - вспомогательная структура */
struct usb_device_descriptor {

        __u8 bLength;
        __u8 bDescriptorType;
        __le16 bcdUSB;
        __u8 bDeviceClass;
        __u8 bDeviceSubClass;
        __u8 bDeviceProtocol;
        __u8 bMaxPacketSize0;
        __le16 idVendor;
        __le16 idProduct;
        __le16 bcdDevice;
        __u8 iManufacturer;
        __u8 iProduct;
        __u8 iSerialNumber;
        __u8 bNumConfigurations;

} __attribute__ ((packed));


struct usb_device {

        int devnum;
        char devpath[16];
        u32 route;
```

```
enum usb_device_state state; /*Состояние устройства:
настроено, не
подключено и т. д.*/
enum usb_device_speed speed; /*Скорость устройства: высокая /
полная / низкая (или ошибка)*/
...
struct usb_device *parent;
struct usb_bus *bus; /*шина, частью которого мы являемся*/
struct usb_host_endpoint ep0; /*данные конечной точки 0
(канал управления по умолчанию)*/
struct device dev; /* универсальный интерфейс устройства
*/
struct usb_device_descriptor descriptor;
struct usb_host_bos *bos;
struct usb_host_config *config;
struct usb_host_config *actconfig;
struct usb_host_endpoint *ep_in[16]; /*массив конечных точек
IN*/
struct usb_host_endpoint *ep_out[16]; /*массив конечных точек
OUT*/
...
u8 portnum; /*номер родительского порта (источник 1)*/
u8 level; /*количество предков USB-концентраторов*/
u8 devaddr;
unsigned can_submit:1; /*URB могут быть представлены*/
...
int string_langid; /*идентификатор языка для строк*/
/* static strings from the device */
char *product; /*идентификатор продукта, если есть
(статический)*/

char *manufacturer; /*Строка i-производителя, если имеется
(статическая)*/
char *serial; /* серийный номер*/
struct list_head filelist;
int maxchild; /*количество портов в хабе*/
u32 quirks;
atomic_t urbnum; /*количество URB, представленных для всего
устройства*/
unsigned long active_duration;
#ifdef CONFIG_PM
unsigned long connect_time; /*время, когда устройство было
впервые подключено/*
...
unsigned port_is_suspended:1;
#endif
```

```
	struct wusb_dev *wusb_dev;
	int slot_id;
	enum usb_device_removable removable;
	…

}
```

Структуры драйверов: struct device_driver, struct pci_driver, struct usb_driver

При разработке драйверов устройств необходимо ориентироваться на структуры, определенные в ядре: struct device_driver, struct pci_driver, struct usb_driver, так как в ином случае драйвер не сможет выполнять свои обязанности.

В состав каждой структуры входят поля с функциями, определенными для работы с драйверами: probe, disconnect, suspend, resume и другие.

Драйверы устройств различаются не только в зависимости от типов устройств, но и от того с какой шиной они работают.

Аналогично тому, что struct device представляет универсальный интерфейс устройства struct device_driver представляет универсальную структуру драйвера.

```
struct device_driver {

	const char *name;
	struct bus_type *bus; //шина, если используем, то выше тип по умолчанию (не знаю, что это значит, но так сказали на лекции)
	struct module *owner;
	const char *mod_name; //имя модуля (для встраиваемых модулей)
	bool suppress_bind_attrs; /* disables bind/unbind via sysfs */
	enum probe_type probe_type;
	const struct of_device_id *of_match_table;
	const struct acpi_device_id *acpi_match_table;

	//точки входа- probe, disconnect, suspend, resume, …Нет open, close .., так как они в struct file_operations
	int (*probe) (struct device *dev); /*Вызывается для запроса
```

```
    существования определенного устройства, может ли этот драйвер
    работать с ним, и связать драйвер с конкретным устройством*/
    void (*sync_state)(struct device *dev);
    int (*remove) (struct device *dev); /*Вызывается, когда
    устройство удаляется из системы, чтобы отсоединить устройство
    от этого драйвера.*/
    void (*shutdown) (struct device *dev); /*Вызывается во время
    выключения, чтобы отключить устройство*/
    int (*suspend) (struct device *dev, pm_message_t state);
    /*Вызывается перевести устройство в спящий режим. Обычно в
    состоянии низкого энергопотребления*/

    int (*resume) (struct device *dev); /* Вызывается, чтобы
    вывести устройство из спящего режима.*/
    const struct attribute_group **groups;
    const struct attribute_group **dev_groups;
    const struct dev_pm_ops *pm;
    void (*coredump) (struct device *dev);
    struct driver_private *p;

};
```

Для драйверов устройств, подключенных к шине PCI определена структура pci_driver. Для USB драйверов определена структура struct usb_driver (в ядре Linux объявлена еще одна структура struct usb_device_driver)

(опять же, назвали и привели главную, остальные просто назвали, и хватит. Если захочется, то структура usb_driver рассматривается в контексте разговора о usb, а вот pci_driver вот на всякий)

Для драйверов устройств, подключенных к шине PCI определена структура:

```
struct pci_driver {

    struct list_head node;
    const char *name;
    const struct pci_device_id *id_table; /* Должен быть
    ненулевым, чтобы вызывалась probe */
    int (*probe)(struct pci_dev *dev, const struct pci_device_id
    *id); /* Новое устройство вставлено */
    void (*remove)(struct pci_dev *dev); /* Устройство удалено
    (NULL, если драйвер без поддержки «горячей» замены)*/
    int (*suspend)(struct pci_dev *dev, pm_message_t state); /*
    Устройство приостановлено */
    int (*resume)(struct pci_dev *dev); /* Устройство проснулось
    */
    void (*shutdown)(struct pci_dev *dev);
```

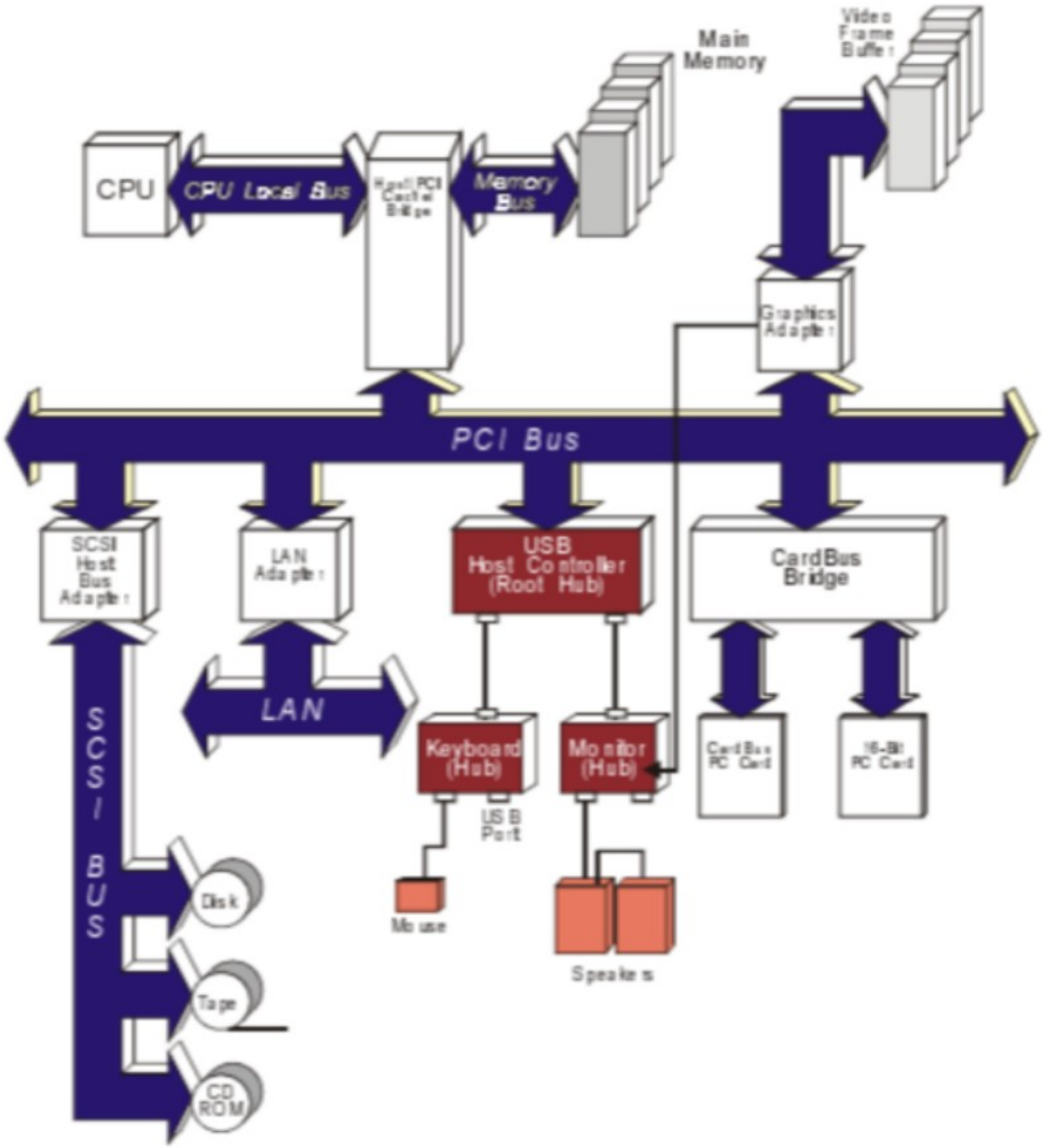
```
        int (*sriov_configure)(struct pci_dev *dev, int num_vfs); /*
        On PF */
        const struct pci_error_handlers *err_handler;
        const struct attribute_group **groups;
        struct device_driverdriver;
        struct pci_dynids dynids;

};
```

USB-шина

Особенности, хост и хабы

USB (Universal Serial Bus — универсальная последовательная шина) является промышленным стандартом расширения архитектуры PC, ориентированным на интеграцию с телефонией и устройствами бытовой электроники.

Данная иллюстрация показывает шину USB, реализованную в системе, построенной на шине PCI *(по сути, важно только красное)*



Устройства (Device) USB могут являться:

- Хабами (Hub). Обеспечивает дополнительные точки подключения устройств к шине.
- Функциями (Function) USB. Предоставляют системе дополнительные возможности, например подключение к ISDN, цифровой джойстик, акустические колонки с цифровым интерфейсом и т. п.
- комбинацией хабов и функций

Устройство USB должно иметь интерфейс, обеспечивающий полную поддержку протокола USB, выполнение стандартных операций (конфигурирование и сброс) и предоставление информации, описывающей устройство.

Работой всей системы USB управляет хост-контроллер (Host Controller), являющийся программно-аппаратной подсистемой хост-компьютера.

Физическое соединение устройств осуществляется по топологии многоярусной звезды. Центром каждой звезды является хаб, каждый кабельный сегмент соединяет две точки - хаб с другим хабом или с функцией. В системе имеется один (и только один) хост-контроллер, расположенный в вершине пирамиды устройств и хабов. Хост-контроллер интегрируется с корневым хабом (Root Hub), обеспечивающим одну или несколько точек подключения - портов. Контроллер USB, входящий в состав чипсетов *(это видимо Keyboard hub и monitor hub с рисунка)*, обычно имеет встроенный двухпортовый хаб. Логически устройство, подключенное к любому хабу USB и сконфигурированное (см. ниже), может рассматриваться как непосредственно подключенное к хост-контроллеру.

Функции представляют собой устройства, способные передавать или принимать данные или управляющую информацию по шине. Функции представляют собой отдельные ПУ с кабелем, подключаемым к порту хаба. Физически в одном корпусе может быть несколько функций со встроенным хабом, обеспечивающим их подключение к одному порту. Перед использованием функция должна быть сконфигурирована хостом - ей должна быть выделена полоса в канале и выбраны опции конфигурации.

Топология шины - это модель соединения между хостом и периферийными устройствами USB. На рисунке 3 (ниже) показаны 7 уровней топологии:

1. USB host with host controller
2. 2-port root hub integrated into the host controller

3. 4-port hub integrated into the keyboard (part of the compound device)
4. USB keyboard (part of the compound device)
5. USB keypad (part of the compound device)

6. 4-port hub (part of the 7-port hub)
7. 4-port hub (part of the 7-port hub)

8. USB mouse

9. USB flash drive
10. 4-port hub

11. 4-port hub
12. USB bluetooth adapter

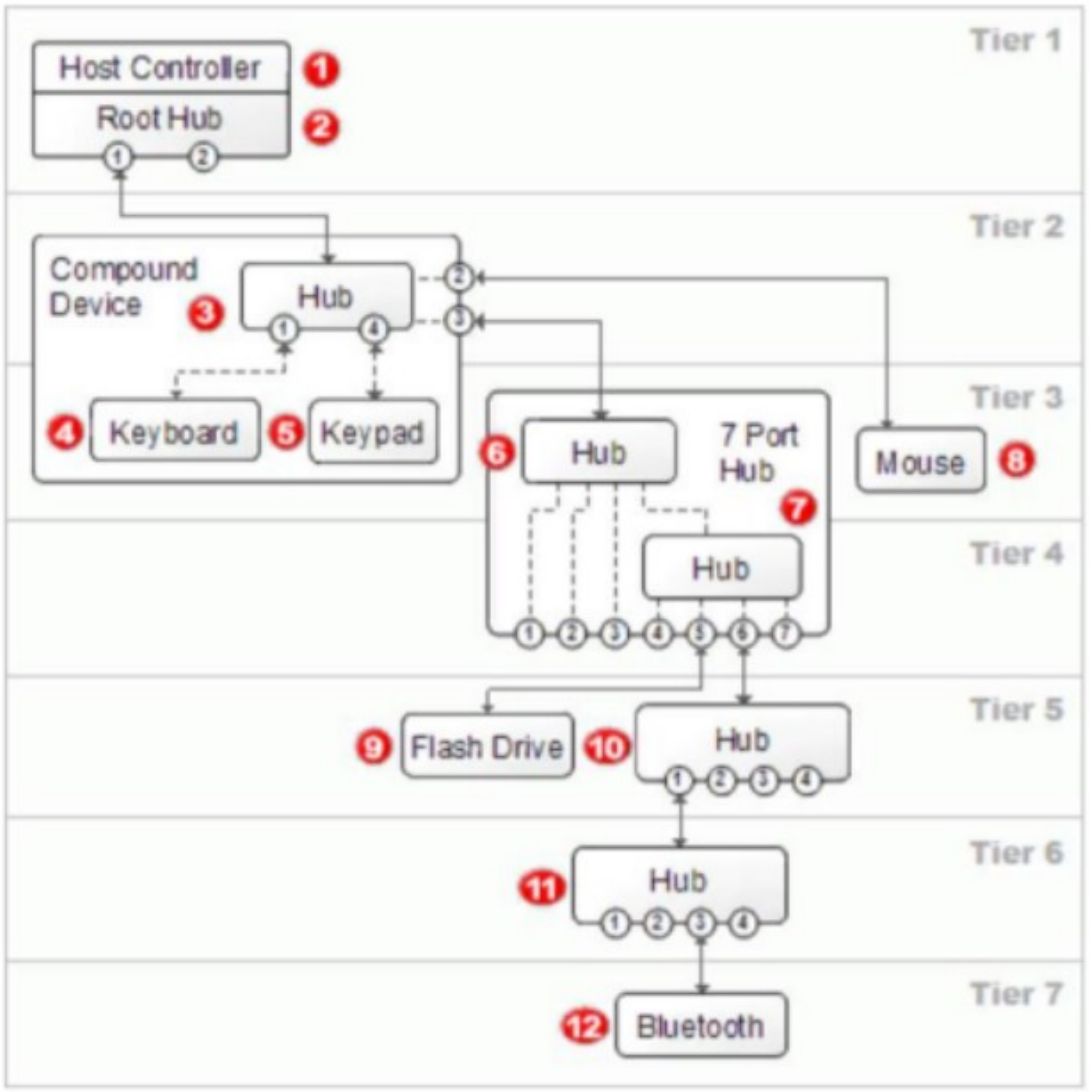


Рис.3 Пример топологии USB с точки зрения хоста

Можно сказать, что на рисунке показано USB-дерево или, так называемая, многоярусная звезда (tiered star). Host контроллер это – корневой узел USB дерева. Hub – концентратор или повторитель имеет одно соединение, которое называется upstream port, к верхнему уровню USB дерева и некоторое количество портов для подключения внешних устройств или других хабов. Хабы являются активными электронными устройствами.

Передача данных (конечные точки и каналы, 4 типа передачи данных)

(далее надо понимать, что в этом контексте функция=устройство USB)

Обсуждалось: в USB-системе может быть только один мастер – host-компьютер. USB - устройства всегда отвечают на запросы host-компьютера, но они никогда не могу посылать информацию самостоятельно.

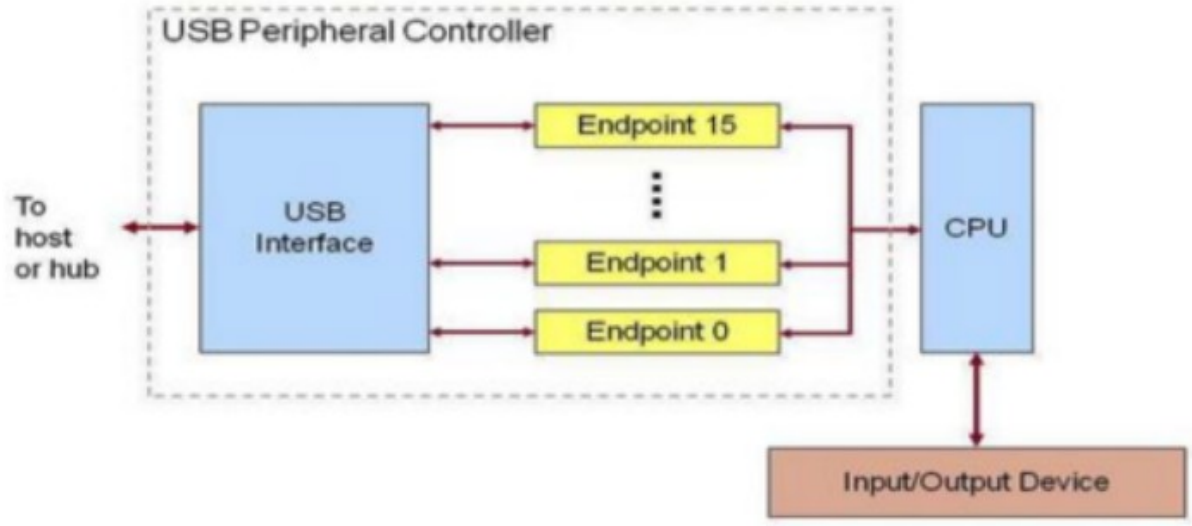
Передача данных выполняется между буфером в памяти хост компьютера и конечной точкой универсальной последовательной шины USB устройства. Перед передачей данные организуются в пакеты. Используемый тип передачи зависит от канала, по которому выполняется передача.

На логическом уровне USB устройство поддерживает транзакции приема и передачи данных. Хост всегда является мастером, а обмен данными должен осуществляться в обоих направлениях:

- OUT - отсылая пакет с флагом OUT, хост посылает данные устройству
- IN - отсылая пакет с флагом IN, хост отправляет запрос на прием данных из устройства.

В устройстве с интерфейсом USB, имеется периферийный контроллер USB. Как показано на рисунке ниже, этот контроллер имеет две основные функции:

1. он взаимодействует с USB-системой (соединяясь с хостом или хабом) *(влево)*
2. содержит в себе буферы, называемые конечными точками *(желтые)*



Конечные точки (endpoints). Это базовый объект связи интерфейса USB. Устройство может иметь от 1 (*как минимум 1!)* до 16 конечных точек (нумерация с 0). Каждая конечная точка может включать в себя два буфера (адреса): входной и выходной (итого от 2 до 32 адресов конечных точек). Это реальная физическая сущность.

Каналы (pipes). USB это протокол, построенный по принципу master/slave. Все общение инициализируется хостом. Хост определяет каналы, которые связаны с конечными точками функции. Это лишь логическая концепция. После установки канала, становится определенным и тип передачи данных, который он поддерживает.

Передачи (transfers). Данные отправляются и принимаются посредством передач или сообщений, состоящих из ряда транзакций, каждая из которых в свою очередь состоит из пакетов. В любой момент времени система USB может поддерживать несколько передач и связанные с ними транзакции. Существует 4 типа передач:

1. control
    - является двунаправленной и предназначена для обмена с устройством короткими пакетами типа «вопрос-ответ». Обеспечивает гарантированную доставку данных. Обычно осуществляется конечной точкой 0, но могут использоваться и другие

    - Используется системным ПО USB для выдачи определенных общих команд на USB устройство и позволяет ПО ОС прочитать информацию об устройстве, такую как коды производителя и модели (PID/VID).

2. isochronous (перевод: изохронный, равновременный)

    - имеет гарантированную пропускную способность (N пакетов за один период шины) и обеспечивает непрерывную передачу данных. Передача осуществляется без подтверждения приема.

    - Используются для устройств с очень большим объемом данных, где синхронизация по времени является более критической, чем точность передаваемых данных. Для приложений реального времени, например для передачи аудио и видео информации

3. interrupt

    - позволяет доставлять короткие пакеты без гарантии доставки и без подтверждений приема, но с гарантией времени доставки – пакет будет доставлен не позже, чем через N миллисекунд.

    - используется в устройствах ввода таких, как клавиатура, мышь или джойстики.

4. bulk (перевод: поточная, сплошная)
    - дает гарантию доставки каждого пакета. Поддерживает автоматическую приостановку передачи данных. Не дает гарантии скорости и задержки доставки. Bulk пакеты передаются в последнюю очередь, т.к. имеет самый низкий приоритет передачи и занимают всю свободную полосу пропускания шины.

    - используется устройствами, отправляющими и принимающими большое количество данных.

USB Драйвер

Структура USB-драйвера (struct usb_driver), таблица id_table, основные точки входа драйвера USB.

Подсистема usb драйверов в Linux связана с драйвером usb_core, который называется usb ядром. Каждое подключенное к машине usb устройство представляется usb-core структурой struct usb_device

При подключении устройства usb-core подбирает для каждого его интерфейса соответствующий драйвер. Каждый интерфейс инкапсулирует одну высокоуровневую функцию, такую как подача аудиопотока на динамик или сообщение об изменении в регуляторе громкости. Устройство может иметь несколько драйверов (принтер может работать как принтер и как сканер). Основная структура, которая заполняется драйвером:

```
struct usb_driver {

	const char *name; //имя драйвера
	//точки входа- probe, disconnect, suspend, resume
	int (*probe) (struct usb_interface *intf, const struct
	usb_device_id *id);
	void (*disconnect) (struct usb_interface *intf);
	int (*unlocked_ioctl) (struct usb_interface *intf, unsigned
	int code, void *buf);
	int (*suspend) (struct usb_interface *intf, pm_message_t
	message);
	int (*resume) (struct usb_interface *intf);
	int (*reset_resume)(struct usb_interface *intf);
	int (*pre_reset)(struct usb_interface *intf);
	int (*post_reset)(struct usb_interface *intf);
	const struct usb_device_id *id_table;
	const struct attribute_group **dev_groups;
	struct usb_dynids dynids;
	struct usbdrv_wrap drvwrap;
	unsigned int no_dynamic_id:1;
	unsigned int supports_autosuspend:1;
	unsigned int disable_hub_initiated_lpm:1;
	unsigned int soft_unbind:1;

};
```

Рассмотрим подчеркнутые в struct usb_driver поля:
(это прям важно расписать. Тут сразу упоминаются функции, которые обычно используются при работе с полями. В каком порядке вызывать их – понятно из пункта ниже, просто расписать логичнее сразу здесь. Про горячее подключение – тоже в следующем пункте)

- name— имя драйвера, которое должно быть уникальным среди всех USB-драйверов в ядре и обычно устанавливается равным имени модуля драйвера (после загрузки драйвера это имя появляется в /sys/bus/pci/drivers/).

- id_table – массив записей usbi_device_id. Используется для “горячего подключения”

  Для использования некоторой группы устройства USB, код модуля определяет массив описания устройств, обслуживаемых этим модулем (то есть это список всех различных видов устройств, которые драйвер может распознать). Каждому новому устройству в этом списке соответствует новый элемент, который заполняется одним из специальных макросов (обычно USB_DEVICE(vendor_id=поставщик, product_id)). Последний элемент массива всегда нулевой, это и есть признак завершения списка устройств.

  Чтобы экспортировать созданную таблицу в пространство пользователя, используется макрос MODULE_DEVICE_TABLE(usb, таблица). Это делается для того, чтобы системы горячего подключения и загрузки модулей (sysfs, udev и т.д.) смогли узнать, с какими устройствами работает данный модуль.

- probe— функция обратного вызова, используемая для инициализации устройства;

  Функция probe вызывается для всех устройств PCI, которые соответствуют таблице идентификаторов и еще не «принадлежат» другим драйверам, когда: во время выполнения функции usb_register_dev() для уже существующих устройств или позже, если новое устройство вставляется. Если таблица не создана, то функция обратного вызова никогда не будет вызвана.

  Функция probe вызывается, когда запись в поле id_table name совпала с именем устройства.

  Эта функция получает struct usb_interface* для каждого устройства, чья запись в таблице идентификаторов соответствует устройству (запись в поле id_table name совпала (match) с именем устройства). Функция probe всегда вызывается из контекста процесса, поэтому она может спать.

  Если драйвер готов работать с устройством (то есть driver выбирает «владение» устройством) probe возвращает 0. Использует usb_set_infdata чтобы связать с интерфейсом. Если драйвер не соответствует устройству или не готов, то возвращается ENODEV.

- remove— функция обратного вызова при удалении устройства;

  Функция remove() вызывается всякий раз, когда устройство, обрабатываемое этим драйвером, удаляется (либо во время отмены регистрации драйвера usb_deregister, либо когда оно вручную извлекается из слота с возможностью

горячей замены). Функция удаления всегда вызывается из контекста процесса, поэтому она может спать.

- suspend— функция менеджера энергосохранения вызывается, когда устройство переходит в пассивное состояние (засыпает).

- resume— функция менеджера энергосохранения, вызываемая при пробуждении устройства.

- shutdown - Предназначен для остановки любых операций DMA на холостом ходу. *Полезно для включения функции пробуждения по локальной сети (NIC) или изменения состояния питания устройства перед перезагрузкой*

- disconnect – вызывается ядром USB при горячем отключении устройства

Как видно из функций, драйвер работает не с struct usb_device, а с интерфейсом struct usb_interface. То есть USB драйверы пишутся для интерфейсов устройств, а не для самого устройства.

Вот структуры на всякий

```
struct usb_device_id {
	/* which fields to match against? */
	__u16		match_flags;

	/* Used for product specific matches; range is inclusive */
	__u16		idVendor; // идентифицирует производителя устройства
	__u16		idProduct;// идентифицирует само устройство
	__u16		bcdDevice_lo;
	__u16		bcdDevice_hi;

	/* Used for device class matches */
	__u8		bDeviceClass;
	__u8		bDeviceSubClass;
	__u8		bDeviceProtocol;

	/* Used for interface class matches */
	__u8		bInterfaceClass;
	__u8		bInterfaceSubClass;
	__u8		bInterfaceProtocol;

	/* Used for vendor-specific interface matches */
	__u8		bInterfaceNumber;

	/* not matched against */
	kernel_ulong_t	driver_info
```

```
        __attribute__((aligned(sizeof(kernel_ulong_t))));
};
```

```
struct usb_interface {

     struct usb_host_interface *altsetting ; //массив
     интерфейсных структур, по одной для каждого альтернативной
     настройки (функции), которая может быть выбрана. Каждая из
     них включает в себя набор
     конфигураций конечных точек.

     struct usb_host_interface *cur_altsetting ; //текущая
     настройка
     ...
     int minor; //младший номер, присвоенный этому интерфейсу,
     если этот интерфейс привязан к драйверу, использующему
     основной номер USB. Если этот интерфейс не использует USB-
     порт, это поле должно быть неиспользованным. Драйвер должен
     установить это значение в функции probe() драйвера, после
     того как ему был назначен второстепенный номер из ядра USB,
     вызвав usb_register_dev().

     ...
     unsigned sysfs_files_created: 1 ; //атрибуты sysfs
     существуют
     ...
     struct device dev ; //представление модели драйвера этого
     устройства
     //является внутренней для usb_interface
     struct device *usb_dev ; //если интерфейс привязан к USB-
     порту, это будет указывать в представление sysfs для этого
     устройства.

     ...
     struct work_struct reset_ws ; //Используется для
     планирования сброса из атомарного контекста.

};
```

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

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

Регистрация usb-драйвера в системе

При разработке драйверов устройств необходимо ориентироваться на структуры, определенные в ядре. Рассмотрим основные действия, которые должен выполнять драйвер устройства, на примере простого драйвера USB.

USB поддерживает «горячее» (plug’n’play) соединение с динамически загружаемым и выгружаемым драйвером. Пользователь не заботится ни о терминировании, ни об IRQ и адресах портов. Загрузка подходящего драйвера осуществляется по комбинации PID/VID (Product ID/Vendor ID). Интерфейсы API для ядра USB выглядят следующим образом (прототип в <linux/usb.h>):

```
int usb_register(struct usb_driver *driver);
void usb_deregister(struct usb_driver *);
```

В структуре usb_driver в соответствующих полях должны быть указаны

- имя устройства - name,
- идентификационная таблица – id_table, используемая для автоматического обнаружения конкретного устройства,
- две функции обратного вызова – probe и disconnect, которые вызываются ядром USB при горячем подключении и отключении устройства, соответственно.

Другие поля драйвера являются необязательными.

Код простейшего драйвера имеет следующий вид:

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

#include <linux/kernel.h>
#include <linux/usb.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("XXX");
MODULE_DESCRIPTION("USB Skeleton Registration Driver");

static int skel_probe(struct usb_interface *interface, const
struct usb_device_id *id)
{
    printk(KERN_INFO "Skel drive plugged\n");
    return 0;
```

```
}
static void skel_disconnect(struct usb_interface *interface)
{
    printk(KERN_INFO "Skel drive removed\n");
}

// массив структур (таблица)
static struct usb_device_id skel_table[] =
{
    { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
    {}
};

//Экспортируем созданную таблицу в пространство пользователя.
MODULE_DEVICE_TABLE(usb, skel_table);

//определение структуры драйвера.
static struct usb_driver pen_driver =
{
    .name = "skel",
    .id_table = skel_table,
    .probe = skel_probe,
    .disconnect = skel_disconnect,
};

static int __init usb_skel_init(void)
{
    int result;
    // регистрация драйвера (передаем указатель на заполненную
структуру)
    result = usb_register(&skel_driver);
    if(result<0)
    {
        err("usb_register failed for the "__FILE__"driver. Error
number %d",result);
        return -1;
    }
    //устройство готово к работе
    …
    return 0;
}

static void __exit usb_skel_exit(void)
{
    //отмена регистрации драйвера
    usb_deregister(&pen_driver);
```

```
}

module_init(usb_skel_init);
module_exit(usb_skel_exit);
```

Далее для привязки драйвера к устройству вызывается

```
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 регистрирующее 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
...
}
```

Прерывания

Система прерываний: типы прерываний и их особенности.

Выделяют:

1. Системные вызовы (программные прерывания) (software interrupts (trups))
    - Вызываются искусственно с помощью соответствующей команды из программы, когда требуется сервис системы (ввод/вывод, обращение к внешнему устройству) – клавиатура, мышь, вторичная память (флешки, диски). Можно вызвать из программы с помощью команды int.
    - Почему обращение к внешним устройствам – системный вызов? Потому что ОС не позволяет программам напрямую обращаться, так как иначе был бы открыт доступ к структуре ядра. СВ-тот интерфейс, который предоставляет пользователю ОС (Application function interface API) - набор функций, определенных в системе, которые может использовать приложение, чтобы получать сервис (обслуживание) системой.
    - ОС старается минимизировать количество СВ особенно ввод/вывод, так как они связаны с обращением к внешним устройствам, *а в UNIX все файл, даже устройства, чтобы со всеми устройствами система работала единообразно (read/write)*
    - СВ – синхронные (по отношению к выполняемой программе) события, которые происходят в процессе работы программы.

2. Исключения (Исключительные ситуации)
    - являются реакцией микропроцессора на нестандартную ситуацию (то есть возникновение исключения), возникшую внутри микропроцессора во время выполнения команды
    - Делятся на
        i. исправимые (пример: страничное прерывание) (приводят к вызову менеджера системы, в результате работы которого может быть продолжена работа процесса) и
        ii. неисправимые (пример: ошибки (/0), ошибки адресации) (процесс будет завершён)
    - Являются синхронными событиями по отношению к коду

3. Аппаратные прерывания (interrupts)

- это прерывания, поступающие от устройств (от таймера, от устройства ввода вывода и т.д.) (поступают от контроллера прерываний)
- это асинхронные события в системе: происходят вне зависимости от какой-либо работы, выполняемой процессором
- Примеры:
    i. Всегда отдельно рассматривается прерывание от системного таймера – особое и единственное периодическое прерывание с важнейшими системными функциями *(В системах разделения времени – декремент кванта)*
    ii. Прерывание от внешнего устройства по завершении операции ввода/вывода – внешние устройства информируют процессор о том, что ввод/вывод завершен и процесс может перейти к обработке. При этом (даже вывод) происходит получение данных об успешности или неуспешности завершения операции.
    iii. Отдельно рассматривается – прерывание от действий оператора (win: ctrl+alt+delete, unix :ctrl+C)
- Бывают (У них разные входы: вход INTR (от INTerrupt Request — запрос на прерывание) и вход NMI (от Not Maskable Interrupt — немаскируемое прерывание):
    i. Маскируемые - прерывания, которые м.б. отложены или запрещены
    ii. Немаскируемые - невозможно запретить или отложить

(не забыть: все про аппаратные прерывания)

Прерывания в последовательности ввода-вывода — обслуживание запроса процесса на ввод-вывод (диаграмма).

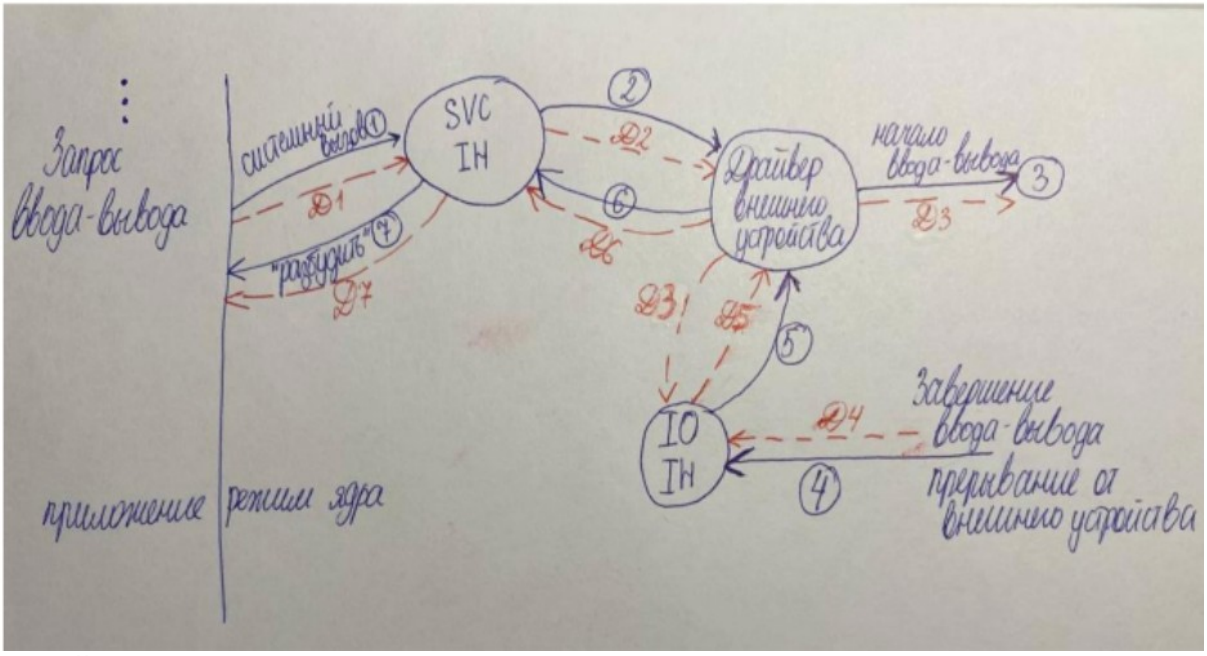
(вроде это называется диаграммой Шоу)

Основную группу прерываний составляют прерывания от устройств ВВ. Без устройств ВВ пользоваться вычислительной системой невозможно, тк именно они предназначены для взаимодействия с пользователем.

SVC — supervisor call

IH — interrupt handler
Супервизор — ОС в стадии выполнения

Последовательность действий в системе при запросе приложения на ввод/вывод:



Любой запрос ввода/вывода блокирует процесс на какое-то время (даже open()!). Здесь происходит вызов read/write и система переходит в режим ядра (ни одна система не дает прямое обращение к внешним устройствам)

Действия (красным на схеме):

(обратить внимание, что у Д3 две стрелочки)
(на лекции все писалось сплошным текстом, но вроде если раскидать по действиям, то получится так)

1. Системный вызов из функции, который переводит систему в режим ядра, туда же соответствующие данные.
2. В результате обработки системного вызова будет вызван драйвер (одна из его точек входа) внешнего устройства (программа ввода/вывода). Драйверу будут переданы данные в его формате
3. Драйвер инициализирует работу внешнего устройства, передает по шине данных в контроллер устройства данные в формате, «понятном» устройству (кавычки обязательны!!!!). На этом управление процессором работой заканчивается, он отключается, потому что работой внешнего устройства управляют контроллеры.
4. По завершении операции ввода/вывода данные от контроллера поступают в регистр данных контроллера прерываний и находятся там пока не сработает цепочка обработки прерывания. Контроллером устройства будет сформировано прерывание, которое в простой схеме поступит на контроллер

прерывания (IO IH) и в результате будет определен адрес точки входа обработчика прерывания.

5. Процессор перейдет на выполнение обработчика, тк аппаратные прерывания имеют наивысший приоритет. **Обработчики прерываний** входят в состав драйвера и **является одной из точек входа драйвера** внешнего устройства. Драйвер всегда содержит 1 обработчик прерывания.
6. Поскольку обработчик – точка входа драйвера, у драйвера есть call back функция – задача вернуть запрашиваемые данные приложению. В результате драйвер через подсистему ввода/вывода должен передать данные приложению.

7. Для этого работа приложения дб возобновлена (разбудить). Процесс, который запросил вв разблокируется.

С монитором мы работаем как с памятью – mov. А input/output – использование команд ввода вывода, и это приводит к блокировкам.

Возникновение прерывания происходит асинхронно. Чтобы получить значение, процесс разблокируется. Поэтому эта схема называется блокирующий синхронный ввод-вывод

Это возможно в архитектуре, где реализовано распараллеливание функций: канальная (используются каналы) и шинная (контроллеры или адаптеры) архитектуры – устройствами управляют устройства.

Даже если при выполнении ВВ не удалось прочитать или записать данные, приложение все-равно получит информацию (об ошибке)

для информирования процессора о завершении операции ввода-вывода контроллер внешнего устройства формирует сигнал, который по линии IRQ№ поступает на контроллер прерывания; контроллер формирует сигнал прерывания, который по линии шины данных поступает на выделенный pin процессора; в конце цикла выполнения каждой команды процессор проверяет наличие сигнала прерывания и при его поступлении переходит на выполнение обработчика данного прерывания; обработчик прерывания должен сохранить данные, пришедшие от контроллера устройства, в буфере ядра, чтобы затем функции обратного вызова драйвера устройства доставили полученные данные в буфер приложения. Для этого приложение разблокируется.

Быстрые и медленные прерывания.

С точки зрения обслуживания прерываний, в ядре ОС Linux различаются 2 вида АП: быстрые и медленные.

Важно отметить, что в версии 2.6.19 все флаги, связанные с прерываниями, были радикально изменены. Раньше была приставка SA: для обозначения быстрых прерываний использовался флаг SA_INTERRUPT.

Эта приставка теперь заменена на приставку IRQF, а единственным быстрым прерыванием осталось прерывание от системного таймера __IRQF_TIMER.

Level_triggered u edge_triggered

В быстрых прерываниях обработчики выполняется полностью – от начала до конца – это их особенность. Медленные прерывания – все остальные прерывания от внешних устройств (а все устройства – внешние, в том числе вторичная память), их обработка требует значительной больше времени.

Верхняя и нижняя половины обработчиков прерываний.

Аппаратные прерывания выполняются в ядре на высочайшем уровне приоритета *(не путать с уровнем привилегий – про другое!)*. Все АП выше DBC dispatch Когда выполняется АП, никакая другая работа в системе выполняться не может. *(Именно так будет выполняться верхняя половина)*

Внимание: это ответ на вопрос “особенности выполнения прерываний в SMP-системах”. На лабораторных никто не мог понять, что на это отвечать, она потом сама сказала, о чем этот вопрос:
Так происходит в SMP (symmetric multiprocessing) архитектуре – равноправные процессоры, которые работают с общей памятью. На том процессоре, который выполняет обработчик возникшего прерывания, запрещены все прерывания, а для остальных процессоров запрещены прерывания по этой линии IRQ. (а есть еще архитектура MSI (Message Signalled Interrupts))

Обычно триггерится линия IRQ (то есть выделенный путь, отдельно от основного пути передачи данных), а MSI заменяет выделенную линию прерывания внутриполосной сигнализацией, обмениваясь специальным сообщением для указания прерываний по основному каналу передачи данных. В частности, MSI позволяет устройству записывать небольшое количество данных, описывающих прерывания, на специальный адрес ввода-вывода, отображаемый в памяти, а затем набор микросхем передает соответствующее прерывание процессору

Обработчики АП должны завершаться как можно быстрее, поэтому выполняют минимально необходимый набор действий. Иначе все сказывается на отзывчивости (быстродействии) системы. Пример: для устройства ввода обработчик АП должен получить данные от устройства и поместить их в буфер ядра. Все. Но ведь процесс, который запрашивал ввод, должен получить данные. То есть не все задачи по обработке прерывания можно выполнить за несколько инструкций и это приводит к

необходимости отложить продолжительную работу и выполнять ее вне контекста IRQ драйвера устройства.

Поэтому, чтобы сократить время выполнения обработчиков прерываний, обработчики (медленных) АП делятся на 2 части – верхнюю и нижнюю половину (top and bottom half) (верхняя половина выполняется как АП – на высочайшем уровне, при запрещенных прерываниях и тд, а нижняя – как отложенное действие).

Такое деление действий, связанных с обслуживанием работы устройств ВВ, связано с уровнем приоритета, на котором должны выполняться АП и уровнем приоритета, на котором должны завершиться операции ВВ (данные об операции приходят и на вводе, и на выводе).

Помимо получения данных от того порта, к которому подключено внешнее устройство и сохранения этих данных в буфере ядра, top half также инициализирует выполнение так называемых отложенных действий (нижней половины) для того, чтобы система могла завершить обработку ВВ. То есть перед завершением обработчик АП (top half) инициализирует применение своей нижней половины (отложенного действия, bottom half). После завершения выполнения обработчика АП (выполнена return) завершается взаимодействие с контроллером прерываний. То есть код АП выполнен, восстанавливаются (разрешаются) локальные прерывания (на том процессоре, на котором выполнялся обработчик), восстанавливается старая маска прерываний (iret).

Пример. Обработчик прерывания от сетевого адаптера просто копирует пришедший пакет в ядро. В ядре пакет ставится в буферную очередь, в которой ожидает обработки соответствующим потоком ядра. Инициализируется выполнение отложенного действия, и уже нижняя половина завершит обработку получения пакета.

В современных UNIX/LINUX 3 типа нижних половин

- Soft IRQS (гибкие прерывания)
- Tasklet
- Workqueue (очереди работ)

Каждый вид обладает особенностями, которые необх учитывать при выборе механизма инициализации отложенного действия.

Регистрация обработчика АП в системе — функция и ее параметры, примеры.

Обработчик прерываний в драйверах устройств отвечают за взаимодействие с внешними устройствами на этапе передачи данных от устройств. Он является одной из точек входа драйвера. Один драйвер имеет один обработчик.

С помощью следующей функции любой драйвер может зарегистрировать собственный обработчик прерывания на определенной линии прерывания.

(для собственного понимания: так мы регистрируем верхнюю половину, а уже она запланирует нижнюю)

```
<linux/interrupt.h>
typedef irqreturn_t (*irq_handler_t)(int, void *);

int request_irq(unsigned int  irq, irq_handler_t handler, unsigned
long flags, const char *name, void *dev);

//В старых версиях эта функция выглядела так:
int request_irq(unsigned int irq, void (*handler) (int, void *,
struct pt_regs *), unsigned long irqflags, const char *devname,
     void *dev_id );
/*
То есть сам обработчик: 1) не был определен отдельно с помощью
typedef. 2) возвращал void 3) имел третий параметр типа struct
pt_regs *, который демонстрировал регистры, сохраняемые в стеке
при возникновении прерывания. Обычно с этими регистрами не стоит
связываться (их можно прочитать, чтобы определить, когда произошло
прерывание  - когда процесс выполнялся в режиме ядра или
пользователя). в современных версиях сохранения регистров
процессора забито в железе, поэтому он стал не нужен
*/
```

Параметры функции request_irq:

1. Номер прерывания
   Для некоторых устройств, например унаследованных (legacy) PC устройств таких, как таймер или клавиатура, это значение обычно жестко определено. Для большинства других устройств эта величина подбирается или назначается динамически и программно.
2. Указатель на обработчик прерывания. Этот обработчик будет вызван при возникновении указанного прерывания в системе *(то есть прерывания с соответствующим значением irq)*

   Возвращаемое значение: irqreturn_t.
   Два варианта, как описать, выбирать самим)

- Из лекции:
  Определены

```
typedef int irqreturn_t;
#define IRQ_NONE (0)
#define IRQ_HANDLED (1)
#define IRQ_RETVAL (x) ((x)!=0)
```

Отсюда следует, что результат работы обработчика может возвращать или IRQ_NONE (если не удалось обработать), или IRQ_HANDLED (если прерывание обработано)

- Из исходного кода

```
/**
 * enum irqreturn
 * @IRQ_NONE           interrupt was not from this device or was
not handled
 * @IRQ_HANDLED        interrupt was handled by this device
 * @IRQ_WAKE_THREAD    handler requests to wake the handler thread
 */
enum irqreturn {
      IRQ_NONE         = (0 << 0),
      IRQ_HANDLED            = (1 << 0),
      IRQ_WAKE_THREAD  = (1 << 1),
};

typedef enum irqreturn irqreturn_t;
#define IRQ_RETVAL(x) ((x) ? IRQ_HANDLED : IRQ_NONE) // по сути
отсюда следует то же самое
```

Первый параметр – номер возникшего прерывания. Мы использовали, чтобы проверить, то ли прерывание произошло:

```
irqreturn_t my_handler(int irq, void *dev)
{
    if (irq == IRQ_NUM) // номер прерывания, который мы хотели
обработать и передавали собственно в request_irq
    {
          // обработка
          return IRQ_HANDLED; // прерывание обработано
    }
    return IRQ_NONE; // прерывание не обработано
}
```

3. Флаги: 0 или битовая маска одного или нескольких следующих флагов: (обязательно написать подчеркнутые. Остальные – по желанию)

```
#define IRQF_SHARED         0x00000080 /*разрешить разделение линии
```

```
IRQ несколькими устройствами.  Устанавливается абонентами (теми, кто вызывает), чтобы разрешить разделение линии IRQ разными устройствами. ДРАЙВЕР управляет устройством – разные драйвера устройства могут быть заинтересованы в использовании одной и той же линии IRQ. Одно устройство может иметь несколько обработчиков прерываний*/
#define IRQF_PROBE_SHARED  0x00000100 /*устанавливается вызывающими, когда они ожидают, что произойдет несовпадение при обмене.  То есть если предполагается возможность наличия проблем при совместном использовании линии IRQ*/
#define __IRQF_TIMER       0x00000200 /* прерывание помечается как прерывание по таймеру.*/
#define IRQF_PERCPU        0x00000400 /*прерывание устанавливается на процессор: только он будет монопольно выполнять конкретный обработчик*/

#define IRQF_NOBALANCING   0x00000800 /*флаг, чтобы исключить это прерывание из балансировки irq*/
#define IRQF_IRQPOLL       0x00001000 /*прерывание используется для опроса (только соображение производительности, которое зарегистрировано первым в общем прерывании, рассматривается)*/
#define IRQF_ONESHOT       0x00002000 /*Прерывание не включается после завершения работы обработчика hardirq. Используется потоковыми прерываниями, которые должны держать линию irq отключенной до тех пор, пока не будет запущен потоковый обработчик*/
#define IRQF_NO_SUSPEND          0x00004000 /*не отключайте этот IRQ во время приостановки. Не гарантирует, что это прерывание выведет систему из приостановленного состояния. См. Documentation / power / suspend-and-interrupts.txt*/
#define IRQF_FORCE_RESUME  0x00008000 /* принудительно включить его при возобновлении, даже если установлен IRQF_NO_SUSPEND*/
#define IRQF_NO_THREAD           0x00010000 /*Прерывание не может быть связано*/
#define IRQF_EARLY_RESUME  0x00020000 /* Возобновить IRQ на ранней стадии во время syscore, а не во время возобновления работы устройства*/
#define IRQF_COND_SUSPEND  0x00040000 /*если IRQ используется совместно с пользователем NO_SUSPEND, запустите этот обработчик прерываний после приостановки прерываний. Для системных устройств пробуждения пользователи должны реализовать обнаружение пробуждения в своих обработчиках прерываний*/
```

4. devname – имя устройства, связанного с прерыванием. Например, для клавиатуры это - "keyboard". Это имя используется в /proc/irq и /proc/interrupt.
5. dev – Используется прежде всего для разделения (shared) линии прерывания. Данные по указателю dev требуются для удаления только конкретного устройства. Указатель void позволяет передавать все, что требуется, например

указатель на handler. В результате free_irq() освободит линию irq от указанного обработчика. Можно установить значение NULL, если линия прерывания не разделена.

Важно отметить, что функция request_irq() может блокироваться и поэтому не может быть вызвана из контекста прерывания.

Когда драйвер выгружается, необходимо отменить регистрацию соответствующего обработчика прерывания. Для этого существует функция:

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

Если указанная линия прерывания не является разделяемой, то функция free_irq() удаляет обработчик и отключает линию. Если линия прерывания разделяется, то обработчик, определённый dev_id удаляется, но линия прерывания будет отключена только при удалении последнего обработчика.

Управление линиями прерываний

(хз, может этот пункт вообще не надо. Но на лекции нам вот что поведали)

Линиями прерывания можно управлять. Макросы управления линиями IRQ определены в <linux/irqflags.h>

Для локального процессора:

```
_local_irq_disable()
_local_irq_enable()
```

Для запрета одной линии прерывания:

```
void disable_irq(unsigned int irq);
void disable_irq_nosync(unsigned int irq);
void enable_irq(unsigned int irq)
```

synchronize_irq предназначена для ожидания завершения обработчика прерывания по линии irq, если он выполняется.

```
void synchronize_irq(unsigned int irq);
```

Нижние половины

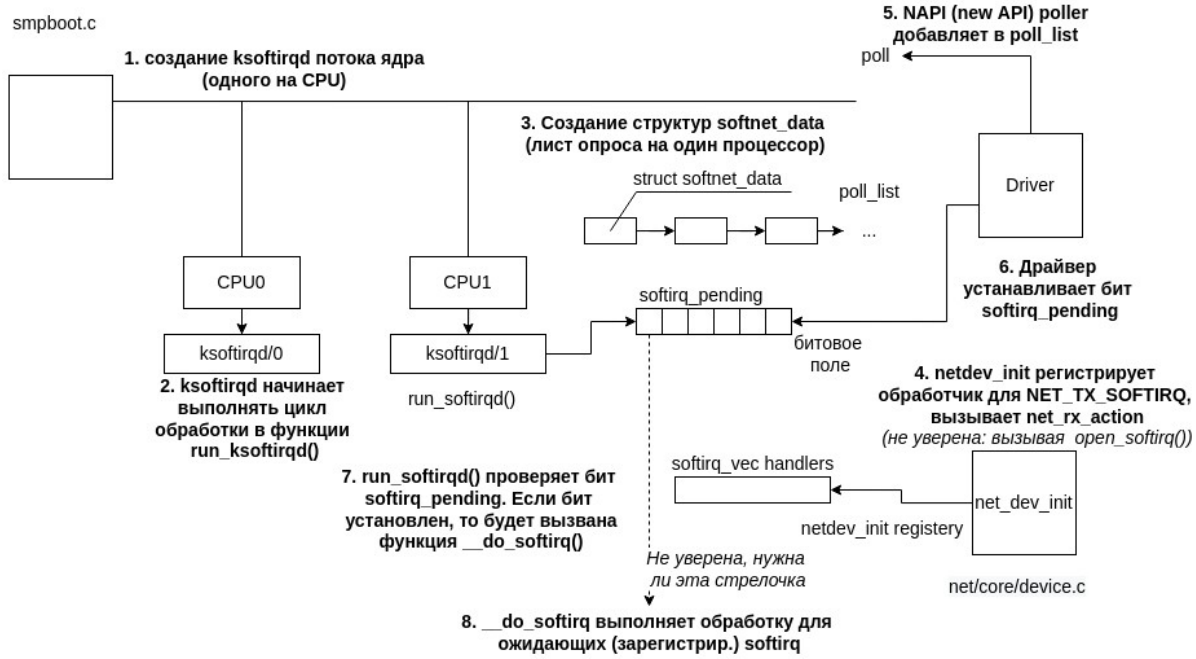
+ Демон ksoftirqd (выполнение softirq)

Обработка отложенных прерываний (softirq) и, соответственно, тасклетов осуществляется с помощью набора потоков пространства ядра (по одному потоку на каждый процессор).

Проверка ожидающих выполнения обработчиков типа softirq выполняется в следующих случаях

1. При возврате из АП
2. В контексте потока ядра ksoftirqd (демон)
3. В любом коде ядра, в котором явно проверяются и запускаются ожидающие выполнения обработчики softirq.

Независимо от способа вызова softirq, его выполнение осуществляется в функции run_softirq(), которая в цикле проверяет наличие отложенных прерываний, то есть ksoftirqd выполняет функцию run_softirq()



(дублирую, что на рисунке)

1. создание ksoftirqd потока ядра (одного на CPU)
2. ksoftirqd начинает выполнять цикл обработки в функции run_ksoftirqd()
   (каждый поток ядра ksoftirqd выполняет функцию run_ksoftirqd())
3. Создание структур softnet_data (лист опроса на один процессор)
4. netdev_init регистрирует обработчик для NET_TX_SOFTIRQ,
5. вызывает net_rx_action *(не уверена: вызывая open_softirq())*
6. NAPI (new API) poller добавляет в poll_list

7. Драйвер устанавливает бит softirq_pending
8. run_softirqd() проверяет бит softirq_pending. Если бит установлен, то будет вызвана функция __do_softirq()

функция __do_softirq *считывает битовую маску softirq __softirq_pending (Пендинг локального процессора) и выполняет отложенные функции соответствующие каждому установленному биту.*

9. __do_softirq выполняет обработку для ожидающих (зарегистрир.) softirq

Демон ksoftirqd – поток ядра каждого процессора, в задачи которого входит планирование и запуск на выполнение softirq. Когда машина нагружена гибкими прерываниями soft interrupts, которые обслуживаются при завершении аппаратного прерывания, который их инициализировал (таких АП может возникать в единицу времени очень много, и каждый инициирует отложенный вызов, организуется очередь soft irq (сюда же входят tasklet)). Например, по размеру очереди или времени работы ksoftirqd можно определить степень давления на систему обращений от сетевого устройства. Чтобы управлять выполнением отложенных действий и предназначены эти демоны.

Следует обратить внимание на то, что демон, то есть kernel_softirq_daemon, запускает соответствующие softirq. То есть в SMP архитектуре каждое отдельное действие в системе выполняется как отдельный поток.

Во время выполнения отложенной функции могут возникнуть новые ожидающие softirq. Основная проблема заключается в том, что выполнение кода пространства пользователя может быть отложено на длительное время пока функция __do_softirq обрабатывает отложенное прерывание. Для исключения такой ситуации устанавливается предел времени выполнения - MAX_SOFTIRQ_TIMER.

1.softirq (гибкие/отложенные irq)

В <linux/interrupt.h> определена struct softirq_action

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

Когда ядро выполняет обработчик отложенного прерывания, то функция action вызывается с указателем на соответствующую структуру softirq_action в качестве аргумента. Например, если переменная my_softirq содержит указатель на элемент массива softirq_vec, то ядро вызовет функцию-обработчик соответствующего отложенного прерывания в следующем виде: my_softirq->action(my_softirq);

В системе существует перечисление определенных в системе гибких прерываний. Они определяются статически при компиляции ядра. В настоящее время определено 10 обработчиков. *(у якубы написано, но я не уверена: имеется возможность создать 32 обработчика softirq)*

Индекс	Приоритет	Назначение
HI_SOFTIRQ	0	Высокоприоритетные
TIMER_SOFTIRQ	1	Таймер
NET_TX_SOFTIRQ	2	Отправка сетевых пакетов
NET_RX_SOFTIRQ	3	Прием сетевых пакетов
BLOCK_SOFTIRQ	4	Блочные устройства (вся вторичная память к ним относится)
BLOCK_IOPOLL_SOFTIRQ	5	опрос
TASKLET_SOFTIRQ	6	тасклеты
SHED_SOFTIRQ	7	планировщик
HRTIMER_SOFTIRQ	8	Не используется
RCU_SOFTIRQ	9	Должен быть последним (учитывать это, если хотим добавлять новые)
NR_ SOFTIRQ		Это не обработчик. Все штуки в первом столбце определены в enum (второй столбец – индекс имени в этом enum), и в его конце NR_ SOFTIRQ. Таким образом, индекс NR_ SOFTIRQ – это количество определенных обработчиков=10

Таким образом мы видим, что индекс определяет имя softirq. softirq с меньшими номерами имеют более высокий приоритет. Из перечисления видно, что tasklet—один из типов softirq.

Все прерывания представлены в массиве:

```
static struct softirq_action softirq_vec[NR_SOFTIRQS]
__cacheline_aligned_in_smp;

const char * const softirq_to_name[NR_SOFTIRQS] = {
     "HI", "TIMER", "NET_TX", "NET_RX", "BLOCK", "IRQ_POLL",
     "TASKLET", "SCHED", "HRTIMER", "RCU"
};
```

Функция open_softirq заполняет массив SOFTIRQ_VEC заданным типом softirq_action. (Рекомендуется этого не делать)

/proc/softirqs

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

То есть эта функция фактически выполняет регистрацию отложенного действия softirq. Чтобы зарегистрированное отложенное действие было поставлено в очередь на выполнение, необходимо вызвать raise_softirq *(должна быть вызвана из обработчика АП)*

```
void raise_softirq(unsigned int nr)
```

Добавить новый уровень softirq можно только путем перекомпиляции ядра, то есть число не может быть изменено динамически. При этом смысл имеет только такой новый softirq, у которого индекс на 1 меньше tasklet: нет смысла переопределять после tasklet, ибо можно tasklet и использовать.

Прервать выполнение (вытеснить) softirq может только аппаратное прерывание.

Сетевая подсистема является важной, softirq - специально определенный набор функций для работы с пакетами

+Softirq VS tasklet

Тасклет – частный случай реализации softirq.
Отличия:

1. различие

Одно и то же softirq может одновременно выполняться на разных процессорах, то есть одновременно в системе может выполняться некоторое количество одних и тех же softirq (это видно на рисунке из предыдущего пункта). Поэтому код softirq должен быть реентерабельным и в нём должно реализовываться взаимоисключение. Благодаря этому же softirq хорошо масштабируется в многопроцессорных системах, это его преимущество

На тасклеты же есть ограничение – обработчик тасклета в каждый момент времени может выполняться только на одном процессоре (то есть один и тот же тасклет не может выполняться параллельно (разные – могут)). В силу этого к тасклетам не предъявляются такие жесткие требования по реентерабельности.

Предпочтение поэтому отдается softirq. Но реализовать tasklet намного проще. Указывается, что тасклет – компромисс между производительностью системы и простой использования.

2. сходство

Блокироваться softirq не может и для реализации взаимоисключения использоваться может только команда test_and_set (обычно используются spinlock, но это макрокоманда в которой используется test_and_set).

Тасклеты также как softirq не могут блокироваться (тоже используется spinlock**)** (но у них и менее жесткие требования по реентерабельности)

Почему не может блокироваться?: Отложенное прерывание выполняется в контексте прерывания, а значит для него недопустимы блокирующие операции.

3. различие

softirq регистрируются при компиляции системы (и их количество определено, 10, и не может быть изменено динамически). Для добавления нового softirq нужно перекомпилировать ядро

тасклеты могут регистрироваться как статически, так и динамически.

2.tasklet (тасклеты)

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

Тасклеты представлены двумя типами отложенных прерываний:

- HI_SOFTIRQ
- TASKLET_SOFTIRQ.

Единственная разница между ними в том, что тасклеты типа HI_SOFTIRQ выполняются всегда раньше тасклетов типа TASKLET_SOFTIRQ.

Структура

Тасклеты, как объекты ядра, описываются соответствующей структурой.

```
struct tasklet_struct
 {
    struct tasklet_struct *next;  /* указатель на следующий
тасклет в     списке (очередь на планирование)*/
    unsigned long state;         /* текущее состояние тасклета,
enum (см ниже)*/
    atomic_t count;              /* счетчик ссылок (см пункт про
активацию и деактивацию) */
    unsigned long data; /* аргумент функции-обработчика тасклета
*/


// два варианта описания одного и того же участка кода (то есть
обработчика)
//на лекции (и в исходном коде) так. Я бы вот этот вариант писала
---------
    bool use_call_back;
    union
     {
          void (*func) (unsigned long data);
          void (*callback)(struct tasklet_struct *t);
}
//-----------------------------------------------------------------
//в методе так (видимо, старые
версии)----------------------------------
    void (*func) (unsigned long); /* функция-обработчик тасклета*/
//-----------------------------------------------------------------
);

//Поле state может принимать одно из следующих значений:
enum
{
     TASKLET_STATE_SCHED, //запланирован на выполнение
     TASKLET_STATE_RUN //выполняется (только в SMP)

//но в лабе, если выводить состояние тасклета из функции-
обработчика, то выводится цифра 2 (при этом 0 и 1 в нужных местах
выводятся, все ок). Что это за 2 – загадка природы
}

/* то есть прототип функции-обработчика тасклета выглядит так*/
void tasklet_handler(unsigned long data)
```

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

Объявление

Тасклет может быть объявлен статически и динамически (в отличие от softirq, см различия).

1. Статически – с помощью 2 макросов. Оба макроса статически создают экземпляр структуры struct tasklet_struct с указанным именем name и обработчиком _callback. Первый макрос создает тасклет, у которого поле count = 0 и следовательно, он разрешен. Второй создает тасклет со счетчиком ссылок count = 1 , то есть он “запрещен”.

```
<linux/interrupts.h>
#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,                    \
}
```

Раньше была еще data

```
<linux/interrupts.h>
DECLARE_TASKLET(name, func, data)
DECLARE_TASKLET_DISABLED(name, func, data);
```

Например

```
DECLARE_TASKLET(my_tasklet, my_tasklet_handler, dev);
// Эта строка эквивалентна следующему объявлению:
struct tasklet_struct my_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.

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

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

Планирование

Тасклеты могут быть запланированы на выполнение функциями:
(с лекции)

```
extern void __tasklet_schedule(struct tasklet_struct *t);
static inline tasklet_shedule(struct  tasklet_struct *t)
{
     if (!(test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
          __tasklet_schedule(t);
}

extern void __tasklet_hi_schedule(struct tasklet_struct *t);
static inline tasklet_hi_shedule(struct  tasklet_struct *t)
{
     If (!(test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
          __tasklet_hi_schedule(t);
}
```

Эти функции очень похожи и отличие состоит в том, что одна функция 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, т.е. в этом случае просто ничего не произойдет. Tasklet 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 выполнится только один раз.

Для оптимизации тасклет всегда (по умолчанию) выполняется на том процессоре, который его запланировал на выполнение.

В методе и исходном коде (здесь более показательны эти номера: TASKLET_SOFTIRQ и HI_SOFTIRQ, и добавление в очередь)

```
static void __tasklet_schedule_common(struct tasklet_struct *t,
```

```
                                struct tasklet_head __percpu *headp,
                                unsigned int softirq_nr)
{
      struct tasklet_head *head;
      unsigned long flags;

      local_irq_save(flags);
      head = this_cpu_ptr(headp);
      t->next = NULL;
      *head->tail = t;
      head->tail = &(t->next);
      raise_softirq_irqoff(softirq_nr);
      local_irq_restore(flags);
}

void __tasklet_schedule(struct tasklet_struct *t)
{
      __tasklet_schedule_common(t, &tasklet_vec,
                                TASKLET_SOFTIRQ);
}

void __tasklet_hi_schedule(struct tasklet_struct *t)
{
      __tasklet_schedule_common(t, &tasklet_hi_vec,
                                HI_SOFTIRQ);
}
```

Активация и деактивация

Tasklet можно активировать и деактивировать функциями:

```
void tasklet_disable_nosync(struct tasklet_struct *t); /*
деактивация. возвращает управление без ожидания завершения
выполнения тасклета */
void tasklet_disable(struct tasklet_struct *t);  /* деактивация с
ожиданием завершения работы  tasklet'a */
void tasklet_enable(struct tasklet_struct *t);  /* активация. Эта
функция должна быть вызвана для того, чтобы можно было
использовать тасклет, созданный с помощью макроса
DECLARE_TASKLET_DISABLED() */
```

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.

Удаление из очереди

tasklet_kill (struct tasklet_struct *t) – ждет завершения тасклета и удаляет тасклет из очереди на выполнение только в контексте процесса. Так как данная функция может переходить в состояние ожидания, то ее нельзя вызывать из контекста прерывания.

Возможность удаления запланированного на выполнение тасклета из очереди очень полезна в случае, когда используются тасклеты, которые сами себя планируют на выполнение. Однако это, конечно, не может предотвратить возможности, что другой код запланирует этот же тасклет на выполнение.

*В методе была еще одна. В исходном коде этой функции нет: tasklet_kill_immediate(struct tasklet_struct *t, unsigned int cpu) – удаляет тасклет в любом случае.*

Свойства планирования

- Если вызывается tasklet_scheduler, то тасклет будет гарантированно выполняться на каком-то процессоре хотя бы 1 раз после этого
- Если уже запланирован, но его выполнение еще не началось, то он будет выполнен один раз (через какое-то время)
- Если тасклет уже выпоняется на другом cpu или планирование тасклета вызвано из самого кода тасклета (он может сам себя запланировать), то его перепланирование будет отложено.
- Тасклет строго сериализован сам по себе, но не по отношению к другим тасклетам. Если клиенту нужна синхронизация между задачами, он делает это с помощью спин-блокировки spin-lock (Тасклеты не могут блокироваться, нельзя использовать семафоры)

  Также в системе есть определенная на тасклетах функция блокировки tasklet_trylock (которая сама использует test_and_set). Она выставляет tasklet'у состояние TASKLET_STATE_RUN и тем самым блокирует tasklet, что предотвращает исполнение одного и того же tasklet'а на разных CPU.

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

Примеры из лабораторной

Вопросы с лабы:

- *показать, что линия прерываний разделяется (IRQ_SHARED).*

См строку if (request_irq(IRQ_NUM, my_handler, IRQF_SHARED, "my_dev_name", &my_handler))

- *когда выполняется обработчик прерывания*

При возникновении прерывания (при нажатии/отпускании клавиши)

- *когда выполняется тасклет*

После обработчика аппаратного прерывания

- *показать в коде последовательность действий, чтобы запланировать work*

my_tasklet = kmalloc(sizeof(struct tasklet_struct), GFP_KERNEL);
tasklet_init(my_tasklet, my_tasklet_function, (unsigned long)my_tasklet_data);
и в обработчике прерывания
tasklet_schedule(my_tasklet)

```c
#include <asm/io.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/vmalloc.h>
#include <linux/slab.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <linux/sched.h>
#include <linux/unistd.h>
#include <linux/time.h>
#include <linux/delay.h>

#define IRQ_NUM 1 // прерывание от клавиатуры

MODULE_LICENSE("GPL");
MODULE_AUTHOR("ALENA");

char *my_tasklet_data = "my_tasklet data";
struct tasklet_struct* my_tasklet;


void my_tasklet_function(unsigned long data)
{
    int code = inb(0x60);
    char * ascii[84] =  {" ", "Esc", "1", и тд};

    if (code < 84)
    {
        printk( ">> my_tasklet: "  "(func): keyboard=%s, state=
%lu\n", ascii[code], my_tasklet->state);
    }
}



irqreturn_t my_handler(int irq, void *dev)
{
    printk(">> my_tasklet: " "(handler) (time=%llu): \n",
```

```
ktime_get());
    if (irq == IRQ_NUM)
    {
        printk(">> my_tasklet: " "info before scheduling: state=
%lu, counter=%u\n", my_tasklet->state, my_tasklet->count.counter);

        tasklet_schedule(my_tasklet); // планирование
тасклета

        printk(">> my_tasklet: " "after:                 state=
%lu, counter=%u\n", my_tasklet->state, my_tasklet->count.counter);
        return IRQ_HANDLED; // прерывание обработано
    }
    printk(">> my_tasklet: " "irq wasn't handled\n");
    return IRQ_NONE; // прерывание не обработано
}


static int __init my_init(void)
{
    //гарантированно получаете непрерывные физические блоки памяти
    my_tasklet = kmalloc(sizeof(struct tasklet_struct),
GFP_KERNEL);
    if (!my_tasklet)
    {
        printk(">> my_tasklet: " "ERROR kmalloc!\n");
        return -1;
    }

    //Динамическое создание (регистрация) тасклета.
    tasklet_init(my_tasklet, my_tasklet_function,
(unsigned long)my_tasklet_data);

    /*
    регистрация обработчика аппаратного прерывания и разрешение
определенной линии irq

    IRQF_SHARED 0x00000080 - устанавливается абонентами  (теми,
кто вызывает), чтобы разрешить разделение линии IRQ разными
устройствами.
    (ДРАЙВЕР управляет устройством - разные драйвера устройств мб
заинтересованы в использовании одной и той же линии IRQ). Одно
устройство может иметь несколько обработчиков прерываний

    4 параметр devname - имя устройства (можно потом посмотреть
```

```
в /proc/interrupts)
    5 параметр dev_id - используется прежде всего для разделения
(shared) линии прерывания. free_irq() освободит линию irq от
указанного обработчика.
    */

    if (request_irq(IRQ_NUM, my_handler, IRQF_SHARED,
"my_dev_name", &my_handler))
    {
        printk(">> my_tasklet: " "ERROR request_irq\n");
        return -1;
    }
    printk(">> my_tasklet: " "module loaded\n");
    return 0;
}

static void __exit my_exit(void)
{
    //ждет завершения тасклета и удаляет тасклет из очереди на
выполнение только в контексте процесса.
    tasklet_kill(my_tasklet);
    kfree(my_tasklet);
    //освободит линию irq от указанного обработчика.
    free_irq(IRQ_NUM, &my_handler);
    printk(">> my_tasklet: " "module unloaded\n");
}

module_init(my_init)
module_exit(my_exit)

/*
sudo insmod my_tasklet.ko
lsmod | grep my_tasklet
sudo dmesg | grep my_tasklet:

Посмотреть инф-ию о обработчике прерывания. Там CPUi - число
прерываний, полученных i-ым процессорным ядром.
cat /proc/interrupts | head -n 1 && cat /proc/interrupts| grep
my_dev_name
*/
```

```
[16696.302126] >> my_tasklet: module loaded
[16697.484795] >> my_tasklet: (handler) (time=16696510482278):
[16697.484798] >> my_tasklet: info before scheduling: state=0, counter=0
[16697.484799] >> my_tasklet: after:                  state=1, counter=0
[16697.484820] >> my_tasklet: (func): keyboard=Enter, state=2
```

Вот еще пример со статическим созданием

```
static void my_tasklet_function(unsigned long data)
{
     printk(KERN_INFO "irq_tasklet_handler call");
}

DECLARE_TASKLET(my_tasklet, my_tasklet_function, 0);

irqreturn_t irq_handler(int irq, void *dev, struct pt_regs *regs)
{
    if (irq == my_irq)
     {
          tasklet_schedule(&my_tasklet);
          return IRQ_HANDLED; // прерывание обработано
     }
     else
          return IRQ_NONE; // прерывание не обработано
}
```

+tasklet VS workqueue

1.

Тасклеты выполняются в контексте прерывания и за короткий промежуток времени после того, как были запланированы. Код тасклета должен быть неделимым.

Workqueue выполняются в контексте процесса (специальных потоков ядра) и имеют большую свободу действий и задержки по времени. Могут блокироваться (в лабе мы вызывали sleep, чтобы показывать это). Код не обязан быть неделимым (это ключевое отличие)

Код ядра требует, чтобы выполнение функций очереди работ откладывались на определенный интервал времени

2.

Тасклеты всегда выполняются на процессоре, на котором выполнялось прерывание, запланировавшее тасклет.

Очереди работ по умолчанию также выполняются на том же процессоре, но могут выполняться и на другом процессоре.

3. А вот когда очень схожи: Если очередь работ привязанная (normal) (то есть не установлен флаг unbound), то второе отличие уходит, и тогда она напоминает тасклет

3.workqueue (очереди работ)

Структуры

На очередях работ определены несколько структур:

1. Работа (work);
2. Очередь работ (workqueue) – коллекция work.
   Workqueue и work относятся как один-ко-многим: в одну очередь работ мб поставлено много работ (работа связывается с конкретной очередью работ)

3. Рабочий (worker). Worker соответствует потоку ядра worker_thread;
4. Пул рабочих потоков (worker_pool) это – набор worker.
   Worker_pool и worker относятся как один-ко-многим: много worker могут принадлежать одному workerpool

5. Pwq (pool_workqueue) – посредник, который отвечает за отношение workqueue и worker_pool:
   a. workqueue и pwq – отношение один-ко-многим,
   b. worker_pool и pwq– отношение один-к-одному.

CMWQ - Concurrency managed workqueue (Concurrency переводится как параллелизм)

(kernel/workqueue.c):

```
// Передает “выданную” (issued) work_struct подходящему
worker_poll через своих pool_workqueues (это мой перевод документации,
исключительно для собственного понимания)
struct workqueue_struct
{

     struct list_head pwqs;      // все посредники (pwqs)
     struct list_head list;  //все очереди работ (workqueue). Но
     на конкретное CPU есть свой список rcu_head (см последнее
     поле), чтобы не перебираться все очереди
     struct mutex mutex;
     ...
     char name[WQ_NAME_LEN];


     /*
     * Destruction of workqueue_struct is RCU protected to allow
     walking the workqueues list without grabbing wq_pool_mutex.
     This is used to dump all workqueues from sysrq.
     */
     struct rcu_head rcu;
```

```
      …

}
```

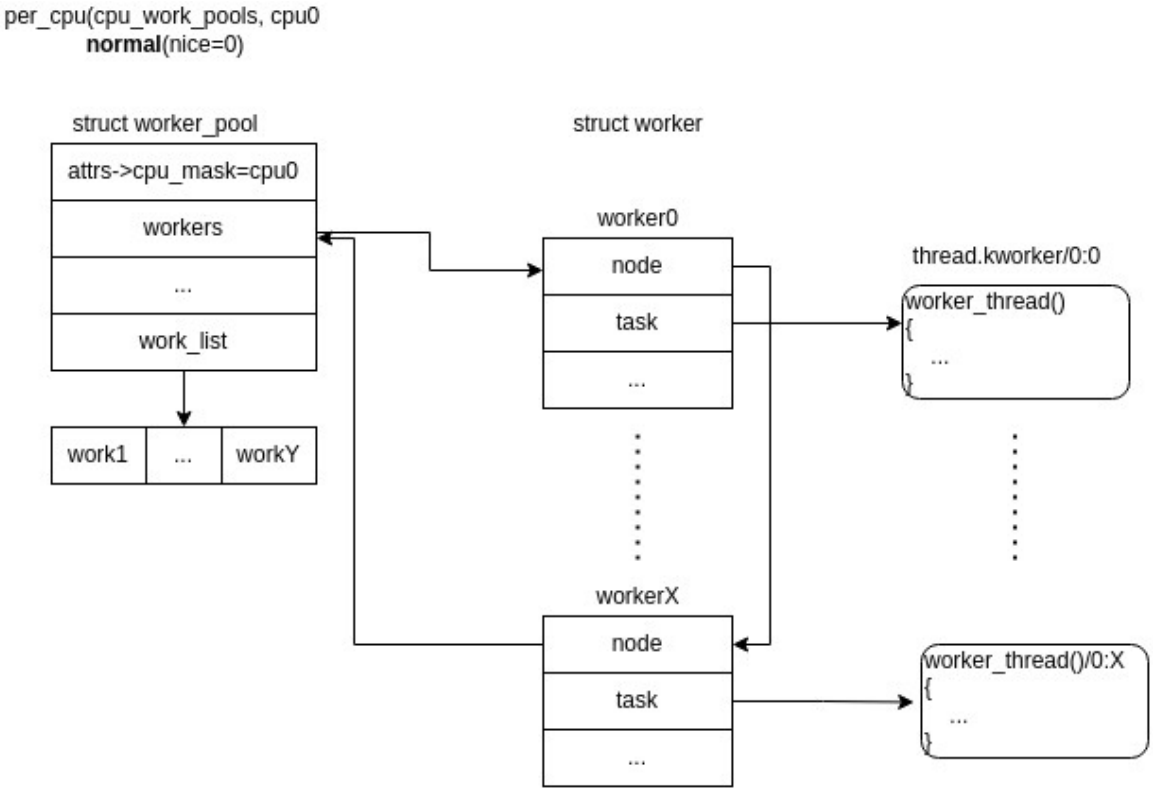
```
struct work_struct
{

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

};
```

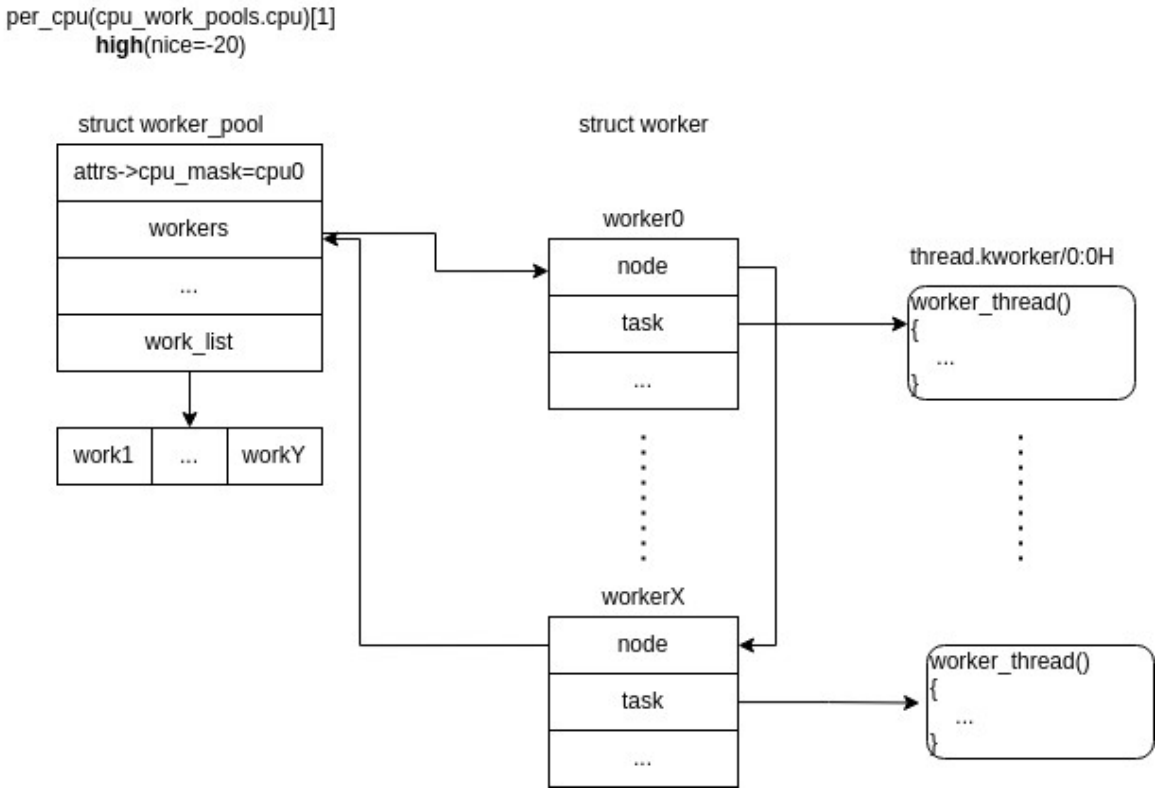
func - функция, которая будет запланирована в рабочей очереди, то есть исполняемая задача, и data - параметр этой функции.

Рисунки

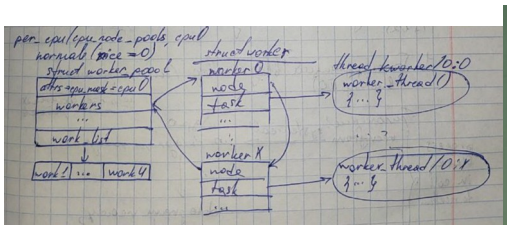
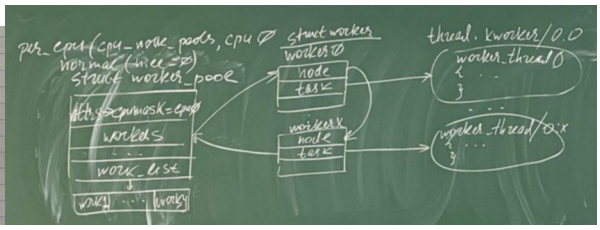
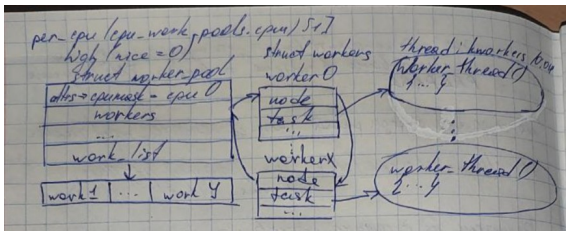
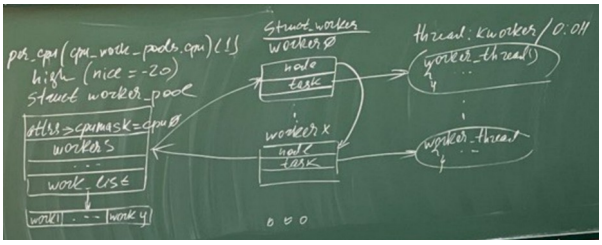
Ядро Linux предоставляет специальные потоки для каждого процессора, которые называются kworker.

На каждое ядро (процессор) есть 2 пула – normal и high.
(Пул рабочих потоков (worker_pool) это – набор worker)





Внизу ставим … и пишем: И так для каждого процессора (n-й cpu)
(источники, вдруг надо подправить):





)

(не уверена в истинности, но вот инфа на всякий)
Заметим, что каждый тип рабочих потоков имеет одну 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.

offtop вброс:
Unix не различают процессы и потоки. В ядре есть функция clone и по современным представлениям (несмотря на многопоточность, юникс писался для параллельного программирования). Все называется task (struct task). Разница – потоки могут не все наследовать – надо указывать соотв флаги. Винды написаны позже. Есть структуры для процесса и потока (у потока полей меньше).

workqueue: флаги

Эти флаги определяют, как очередь работ будет выполняться

```
enum
{
   WQ_UNBOUND              =1<<1,/*not bound to any cpu*/
   WQ_FREEZABLE            =1<<2,/*работа будет заморожена, когда система
будет приостановлена.*/
   WQ_MEM_RECLAIM          =1<<3,/*may be used for memory reclaim*/
   WQ_HIGHPRI              =1<<4,/*high priority*/
   WQ_CPU_INTENSIVE        =1<<5,/*cpu intensive workqueue*/
   WQ_SYSFS                =1<<6,/*visible in sysfs/

   WQ_POWER_EFFICIENT      =1<<7
   WQ_MAX_ACTIVE           =512
```

```
...
}
```

- WQ_UNBOUND: По наличию этого флага очереди (workqueue) делятся на привязанные (normal) и непривязанные (unbound)

    - В привязанных 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. Полезно, когда задачи могут выполняться в течение длительного времени, причем так долго, что лучше разрешить планировщику управлять своим местоположением.

- В системе есть очереди нормального и повышенного приоритета. WQ_HIGHPRI: задания, представленные в такой workqueue, будут поставлены в начало очереди и будут выполняться (почти) немедленно. Несколько задач, отправляемых в очередь с высоким приоритетом, могут конкурировать друг с другом за процессор.

- WQ_CPU_INTENSIVE: имеет смысл только для привязанных очередей. Задачи в такой workqueue могут использовать много процессорного времени.

workqueue: создание и уничтожение

Очередь работ создается функцией

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

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

Также может использоваться вызов create_workqueue:

```
#define create_workqueue(name)
alloc_workqueue("%s", __WQ_LEGACY | WQ_MEM_RECLAIM, 1, (name))
```

Для уничтожения очереди работ используется функция:

```
destroy_workqueue(struct workqueue_struct * wq);
```

work_struct: создание

Структура work_struct представляет задачу (обработчик нижней половины) в workqueue (очереди работ). *Её можно воспринимать как аналог структуры tasklet.* Прежде чем поместить work_struct в очередь работ ее надо сначала заполнить (инициализировать). Сделать это можно двумя способами:

1. во время компиляции (статически) с помощью макросов

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

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

2. Динамически с помощью макроса:

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

(вспоминаем курс Си. В 1. макрос прям просто заменяется на текст, поэтому “статически”, а в 2. в макросе написано do {...} while (0), то есть он подставится, но выполнится уже динамически во время работы программы. Х это наверняка не интересует, но мало ли)

work_struct: добавление в workqueue

После того, как будет инициализирована структура для объекта work, следующим шагом будет помещение этой структуры в очередь работ. Это можно сделать несколькими способами. Все функции возвращают false если work уже в wq, иначе true

1. Добавить работу в конкретную очередь работ

queue_work назначает работу текущему процессору, а с помощью функции queue_work_on можно указать процессор cpu, на котором будет выполняться обработчик.

```
bool queue_work(struct workqueue_struct *wq, struct work_struct *work)
{
      return queue_work_on(WORK_CPU_UNBOUND, wq, work);
}
bool queue_work_on(int cpu, struct workqueue_struct *wq, struct
work_struct *work);
```

И аналогичные функции для отложенной работы (в этих функциях инкапсулирована структура work_struct и таймер, определяющий задержку, то есть планируется выполнение отложенных на определённое время работ).

```
bool queue_delayed_work(struct workqueue_struct *wq, struct delayed_work
*dwork, unsigned long delay)
{
      return queue_delayed_work_on(WORK_CPU_UNBOUND, wq, dwork, delay);
}
bool queue_delayed_work_on(int cpu, struct workqueue_struct *wq, struct
delayed_work *dwork, unsigned long delay)
```

2. 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 не нужно определять структуру очереди работ.

```
bool schedule_work(struct work_struct *work)
{
      return queue_work(system_wq, work);
}
bool schedule_work_on(int cpu, struct work_struct *work)
{
      return queue_work_on(cpu, system_wq, work);
}
// а вот эти 2 есть в методе, но в коде не нашла. В методе вообще
у всех возвращается int, но смысл тот же
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 );
```

Завершения

Есть также целый ряд вспомогательных функций, которые можно использовать, чтобы принудительно завершить (flush) или отменить работу из очереди работ.

```
//принудительно завершить конкретный элемент work и блокировать прочую
обработку прежде, чем работа будет закончена
int flush_work( struct work_struct *work );
//принудительно завершить все работы в данной очереди
int flush_workqueue( struct workqueue_struct *wq );
//принудительно завершить глобальную очередь работ ядра
```

```
void flush_scheduled_work(void);
```

Завершить работу в очереди, либо возникнет блокировка до тех пор, пока не будет завершен обратный вызов (если работа уже выполняется обработчиком):

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

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

```
#define work_pending(work) \
     test_bit(WORK_STRUCT_PENDING_BIT, work_data_bits(work))

#define delayed_work_pending(w) \
     work_pending(&(w)->work)
```

Пример из лабораторной

Первый вариант (классический): normal очередь работ (работы выполняются на том процессоре, который ее запланировал).

```
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/workqueue.h>
#include <linux/delay.h>
#include <asm/io.h>

MODULE_LICENSE("GPL");

#define IRQ 1
static int devID;

struct workqueue_struct *workQueue;
char *simbol;

void queue_functionF(struct work_struct *work)
{
    int code = inb(0x60);
    char * ascii[84] =  {" ", "Esc", "1", "2", и тд};
```

```
    if (code < 84)
    {
        simbol = ascii[code];
        printk(KERN_INFO ">> my_Queue WHAT IS PRESSED: keyboard
%s\n", ascii[code]);
    }
    printk(KERN_INFO "<<<my_Queue: Key was clicked (1 worker) and
we sleep\n");
    msleep(10);
    printk(KERN_INFO "<<<my_Queue: Key was clicked (1 worker)
return\n");
}

void queue_functionS(struct work_struct *work)
{
    // For kernel 5.4
     atomic64_t data64 = work->data;
     long long data = data64.counter;
     // For kernel 5.4

     printk(KERN_INFO "<<<my_Queue: queue_functionS data = %lld\
n", data);
    printk(KERN_INFO "<<<my_Queue: Key was clicked (2 worker)\n");
}

struct work_struct fWork;
struct work_struct sWork;

irqreturn_t handler(int irq, void *dev)
{
    printk(KERN_INFO "<<<my_Queue: move work to queue...\n");
    if (irq == IRQ)
    {
        // Помещаем структуру в очередь работ.
           // queue_work назначает работу текущему процессору.
        queue_work(workQueue, &fWork);
        queue_work(workQueue, &sWork);
        return IRQ_HANDLED;
    }
    return IRQ_NONE;
}


static int __init work_queue_init(void)
{
    int ret = request_irq(IRQ,      /* номер irq */
```

```
                    handler,        /* наш обработчик */
                    IRQF_SHARED,    /* линия может быть разделена,
IRQ
                                                        (разрешено
совместное использование)*/
                    "my_irq2_handler",      /* имя устройства
(можно потом посмотреть в /proc/interrupts)*/
                    &devID);        /* Последний параметр
(идентификатор устройства) irq_handler нужен
                                                        для того,
чтобы можно отключить с помощью free_irq  */

    if (ret)
    {
        printk(KERN_ERR "<<<my_Queue: handler wasn't registered\
n");
        return ret;
    }

    if (!(workQueue = create_workqueue("my_queue"))) //создание
очереди работ
    {
        free_irq(IRQ, &devID);
        printk(KERN_INFO "<<<my_Queue: workqueue wasn't created");
        return -ENOMEM;
    }

    INIT_WORK(&fWork, queue_functionF);
    INIT_WORK(&sWork, queue_functionS);

    printk(KERN_INFO "<<<my_Queue: module loaded\n");
    return 0;
}

static void __exit work_queue_exit(void)
{
    // Принудительно завершаем все работы в очереди.
     // Вызывающий блок блокируется до тех пор, пока операция не
будет завершена.
    flush_workqueue(workQueue);
    destroy_workqueue(workQueue);
    free_irq(IRQ, &devID);
    // remove_proc_entry("workqueue", NULL);
    printk(KERN_INFO "<<<my_Queue: module unloaded\n");
}

module_init(work_queue_init)
```

```
module_exit(work_queue_exit)
```

Второй вариант (ну мало ли). Создается очередь работ с флагом WQ_UNBOUND, что позволяет указывать, на каком процессоре будет выполняться добавляемая в очередь работа. Обработчик прерывания от клавиатуры с помощью функции queue_work_on планирует выполнение 2 работ, у первой функция – анализировать код нажатой клавиши, у второй – засыпать (так показываем, что может блокироваться)

```
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/workqueue.h>
#include <linux/delay.h>
#include <linux/sched.h>
#include <linux/unistd.h>
#include <linux/time.h>
#include <asm/io.h>
#include <cpuid.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("ALENA");

#define IRQ_NUM 1

struct workqueue_struct *my_work_queue;
struct work_struct my_work1;
struct work_struct my_work2;

void my_work_func1(struct work_struct *work)
{
    int code = inb(0x60);
    char * ascii[84] = {" ", "Esc", "1", "2", и тд};

    if (code < 84)
    {
        printk( ">> my_workqueue7: "  "(func1): keyboard=%s\
n", ascii[code]);
    }
}

void my_work_func2(struct work_struct *work)
```

```
{
    printk( ">> my_workqueue7: "  "(func2): go to sleep at
%llu\n", ktime_get());
    mdelay(100);
    printk( ">> my_workqueue7: "  "(func2): return at
%llu\n", ktime_get());
}

irqreturn_t my_handler(int irq, void *dev)
{
    if (irq == IRQ_NUM)
    {
        //добавить работу в очередь работ (назначает работу
текущему процессору)
        printk(">> my_workqueue7: " "add works to queue\n");

        queue_work_on(0, my_work_queue, &my_work1);
        queue_work_on(1, my_work_queue, &my_work2);

        return IRQ_HANDLED;
    }
    return IRQ_NONE;
}

static int __init my_init(void)
{
    int rc = request_irq(IRQ_NUM, my_handler, IRQF_SHARED,
"my_dev_name", &my_handler);
    if (rc)
    {
        printk(">> my_workqueue7: " "ERROR request_irq\n");
        return rc;
    }

    if (!(my_work_queue = alloc_workqueue("my_workqueue",
__WQ_LEGACY | WQ_MEM_RECLAIM | WQ_UNBOUND, 1)))
    {
        free_irq(IRQ_NUM, &my_handler);
        printk(">> my_workqueue7: " "ERROR
create_workqueue");
        return -ENOMEM;
    }
```

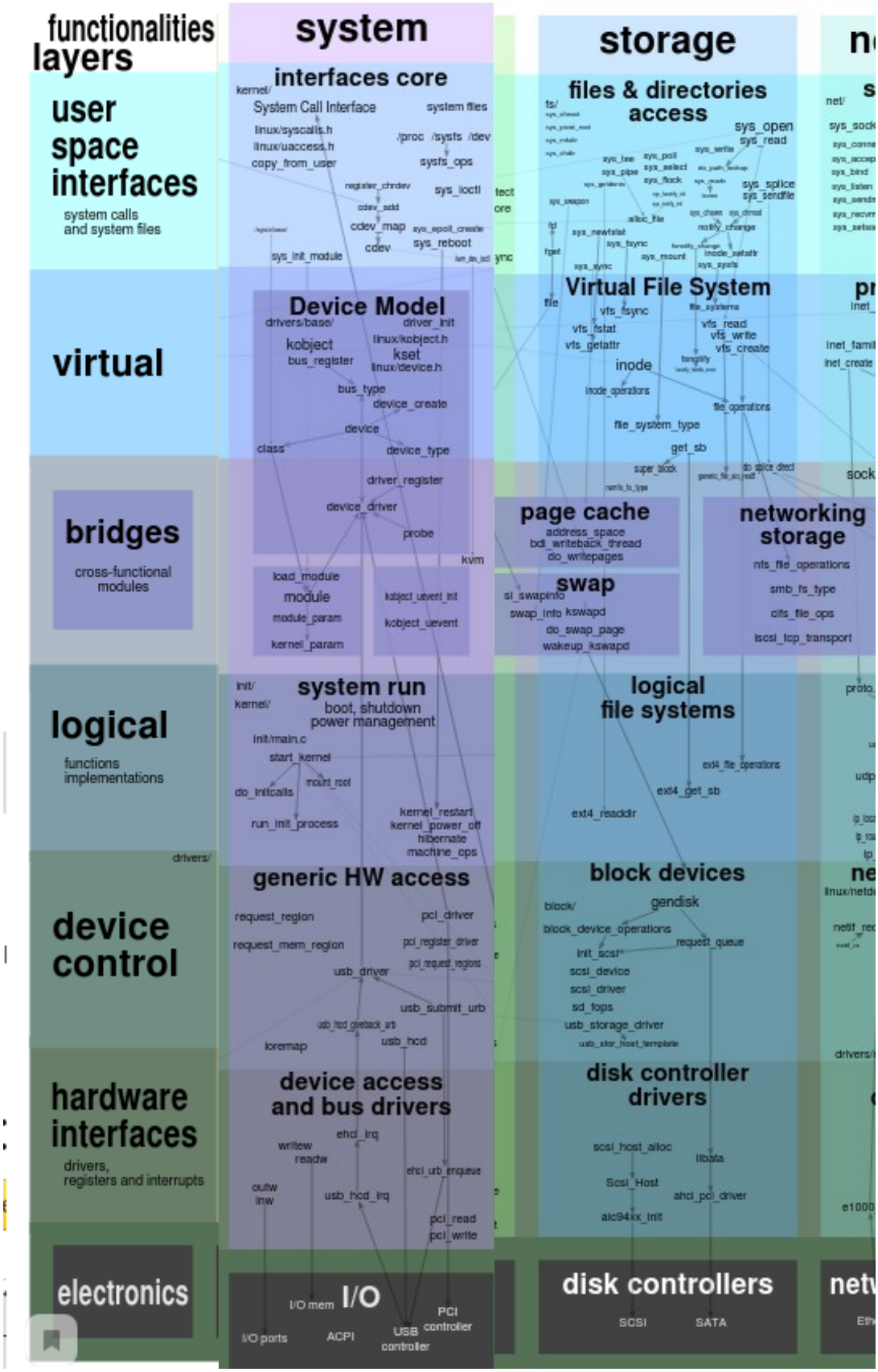
```
    INIT_WORK(&my_work1, my_work_func1);
    INIT_WORK(&my_work2, my_work_func2);


    printk(">> my_workqueue7: " "module loaded\n");
    return 0;
}

static void __exit my_exit(void)
{
    //принудительно завершить все работы в данной очереди
    flush_workqueue(my_work_queue);
    // удалить
    destroy_workqueue(my_work_queue);
    free_irq(IRQ_NUM, &my_handler);
    printk(">> my_workqueue7: " "module unloaded\n");
}

module_init(my_init)
module_exit(my_exit)
```

OFFTOP

Картинка: (ядро linux) уровни управления внешними устройствами

В файле также приводится картинка с описанием (но нам про это ничего не говорили, только одно предложение она упомянула: Блочные устройства связаны с файловой системой)

В этой картинке взяты столбцы layers, system и storage из linux kernel map Не думаю, что можно адекватно перерисовать, но можно переписать основное. Как я поняла этот рисунок: можно рассматривать как таблицу: слева – названия строк (user space interfaces, virtual, …), сверху – названия столбцов (system и storage). Если не поняла, что я имею в виду – напиши мне))



На рисунке показаны уровни управления внешними устройствами. Это многоуровневая система (смотрим сверху-вниз):

- Начинается с Пользовательского интерфейса (User space interfaces), содержащего системные вызовы (Interfaces cores) и системные файлы (files and directories access) (например, виртуальная файловая система /proc, /sysfs, /dev)
- Затем идет виртуальный уровень: «Модель устройств» и «Виртуальная файловая система». В позиции «Модель устройств» можно выделить три позиции:
    - шину (bus),
    - устройство (device)
    - драйвер устройства (device_driver). К позиции device_driver направлена стрелка от probe.
- Между виртуальным уровнем и логическим уровнем находятся, так называемые, мосты – кросс-функциональные модули.
- Логический уровень расшифровывается как реализация функций.
- На уровне «Управление устройствами» (device control) :
    - в позиции Система (второй столбец) определяется общий доступ к оборудованию
    - в позиции Запоминающее устройство (3 столбец) – блочные устройства. Блочные устройства рассматриваются по отношению к файловой системе, так как именно на блочных устройствах хранятся обычные файлы и выделяется область свопинга. Блочные устройства связаны с файловой системой.
- Интерфейс аппаратных средств (строка Hardware interfaces) - драйверы, регистры и прерывания.
    - В позиции «Доступ к устройству и драйвер шины» включены позиции, связанные с EHCI — Enhanced Host Controller Interface (Расширенный интерфейс хост-контроллера), который является улучшенной версией UHCI - Universal Host Controller Interface, который работает как PCI-устройство, но, в отличии от EHCI использует порты, а не MMIO (Memory-Mapped-IO).
    - В позиции Драйверы контроллера диска указан интерфейс SCSI (Small Computer System Interface «скази»), который представляет собой набор стандартов для физического подключения и передачи данных между компьютерами и периферийными запоминающими устройствами.
- На самом нижнем уровне находится физические устройства, которые здесь обозначены как electronics.

Все про (аппаратные) прерывания:

Аппаратные прерывания (interrupts)

- это прерывания, поступающие от устройств (от таймера, от устройства ввода вывода и т.д.) (поступают от контроллера прерываний)
- это асинхронные события в системе: происходят вне зависимости от какой-либо работы, выполняемой процессором
- Примеры:

i. Всегда отдельно рассматривается прерывание от системного таймера – особое и единственное периодическое прерывание с важнейшими системными функциями (а также единственное быстрое) *(В системах разделения времени – декремент кванта)*

ii. Прерывание от внешнего устройства по завершении операции ввода/вывода – внешние устройства информируют процессор о том, что ввод/вывод завершен и процесс может перейти к обработке. При этом (даже вывод) происходит получение данных об успешности или неуспешности завершения операции.

iii. Отдельно рассматривается – прерывание от действий оператора (win: ctrl+alt+delete, unix :ctrl+C)

- Бывают (У них разные входы: вход INTR (от INTerrupt Request — запрос на прерывание) и вход NMI (от Not Maskable Interrupt — немаскируемое прерывание):
    i. Маскируемые - прерывания, которые м.б. отложены или запрещены
    ii. Немаскируемые - невозможно запретить или отложить

Запрос прерывания и линии IRQ

IRQ(Interrupt Request) - запрос на обработку прерывания;

В конце каждой выполняемой команды процессор проверяет наличие прерывания на ножке процессора. Если сигнал был получен, то процесс переходит к исполнению обработчика прерывания - это процесс обработчика аппаратного прерывания.

В трёх-шинной архитектуре внешними устройствами управляют контроллеры или адаптеры.

Контроллером называется устройство которое как правило входит в состав внешнего устройства. Адаптером называется устройство, которое как правило находится на материнской плате.

По завершению операции процессор информируют специальные устройства - **контроллеры.**

Контроллер – программно-управляемое устройство, в нем имеется набор регистров и некоторая логика.

Контроллер получает от процессора команду, выполняя которую, контроллер берет на себя управление операцией ввода/вывода. По завершении операции ввода/вывода контроллер посылает на вход контроллера прерываний сигнал.

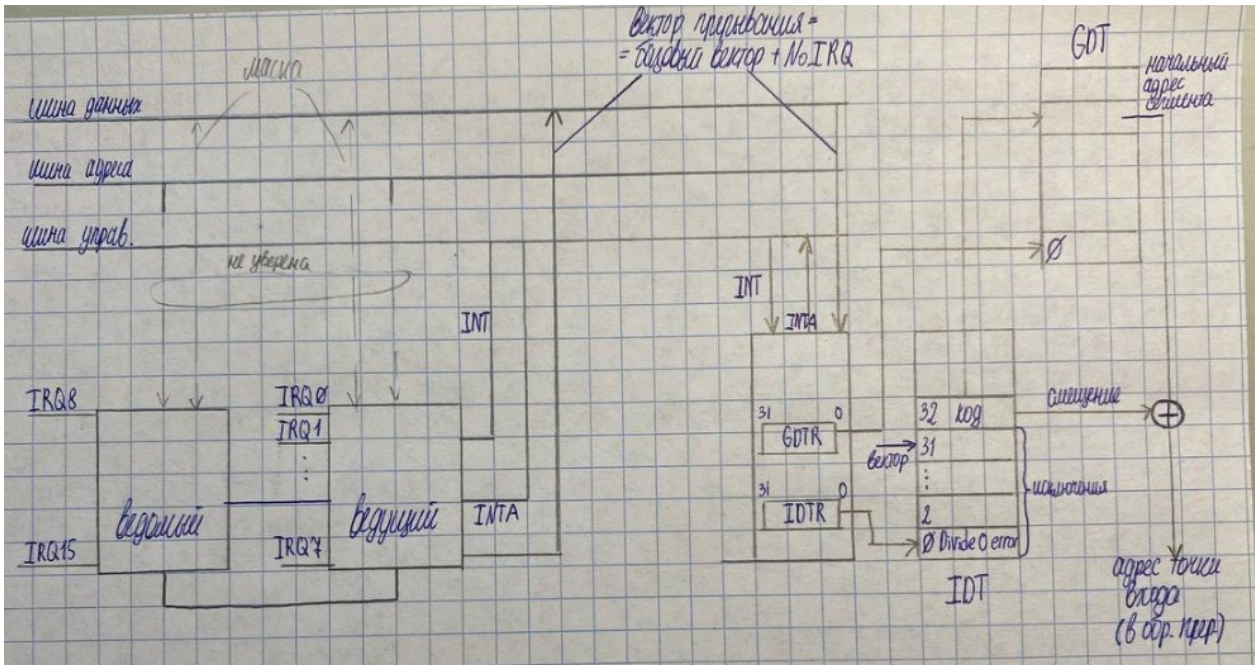
ЧТО ТАКОЕ ЛИНИЯ IRQ

Ответ в схеме!!!!!!
Если взять простейшую схему соединения внешних устройств с контроллером прерывания, **выделенная линия irq – та линия, на которую устройство посылает сигнал прерывания.** В старых системах каждому устройству выделялась конкретная линия – таймеру, клаве, принтеру. В досе это было видно. Эти понятия остались. **Это линия, которая передает сигнал прерывания**. И когда смотри proc/interrupts, 12 - мышь psпополам, x-таймер,y- клава, остальные – назначаются динамически, но суть та же.

Линия IRQ - контакт контроллера

Адресация аппаратных прерываний в з-р (трехшинная архитектура).

(если прерывание не замаскировано)



тут “32 код” = “32 сегмент кода”

(точка входа)

IDTR-32 разрядный регистр, который содержит начальный линейный адрес IDT (все эти регистры есть в каждом ядре).

На вход контроллера приходит сигнал, контроллер формирует сигнал int, который по шине управления переходит на ножку процессора. Процессор посылает intA по шине управления в контроллер. Контроллер выставляет на шину данных вектор прерывания, который содержит селектор к IDT.

Вектор поступает в процессор. Процессор берет значение базового адреса IDT из регистра IDTR и по селектору (из вектора прерывания) находит нужный дескриптор.

Этот дескриптор содержит селектор для сегмента кода, смещение к точке входа и атрибуты. По селектору выбираем дескриптор из GDT, берём оттуда базовый адрес сегмента и прибавляем его к смещению. Получаем линейный адрес точки входа

Отдельно адресуются ведущий и ведомый контроллеры.

Зубков

Существует два контроллера прерываний.

Первый контроллер, обслуживающий запросы на прерывания от IRQO до IRQ7, управляется через порты 20h и 21h, а второй (IRQ8 - IRQ15) - через порты OAOh и OAlh. если несколько прерываний происходят одновременно, обслуживается в первую очередь то, у которого высший приоритет. При инициализации контроллера высший приоритет имеет IRQO (прерывание от системного таймера), а низший - IRQJ7. Все прерывания второго контроллера (IRQ8 - IRQ15) оказываются в этой последовательности между IRQ1 и IRQ3, так как именно IRQ2 используется для каскадирования этих двух контроллеров. В тот момент, когда выполняется обработчик аппаратного прерывания, других прерываний с низшими приоритетами нет, даже если обработчик выполнил команду sti. Чтобы разрешить другие прерывания, каждый обработчик обязательно должен послать команду EOI - конец прерывания - в соответствующий контроллер..

еще инфа

В ЗР для адресации прерывания имеется специальная таблица IDT. Если первые 32 исключения, и мы возьмем 8, то попадем на double fault. Чтобы адресовать прерывания, надо перепрограммировать контроллер на: ведущий-на базовый вектор 32, и этот номер использовать для обращения к таблице. Базового адреса нет, есть смещение и селектор. Отдельно адресуются ведущий и ведомый. От контроллера они могут получить маску??

В РР процессор использует вектор и таблицу векторов прерываний. У ведущего контроллера базовый вектор=8 (8+0=8h) и номер используется для получения смещения к адресу в таблице векторов прерываний. В DOS адрес наз. вектор (4 байт)

В з-р для каждого сегмента программы должен быть определен дескриптор - 8-байтовое поле, в котором в определенном формате записываются базовый адрес сегмента, его длина и некоторые другие характеристики. В р-р сегменты определяются их линейными адресами. (Р-Ф).

Таблица дескрипторов прерываний (IDT)

IDT (interrupt descriptor table) - системная таблица, предназначенная для хранения адресов обработчиков прерываний. (нужна для адресации обработчиков прерываний)

Первые 32 элемента таблицы - под исключения (синхронные события в процессе работы программы) (внутренние прерывания процессора) (в 386 всего 19 исключений (0-19), остальные (20-31) зарезервированы, а *в 486 – и того меньше (реально-18, остальные-зарезервированы)*

32-255 – определяются пользователем (user defined)

Смещение=номер исключения*8

Нам надо адресовать 2 обработчика – таймера и клавиатуры (аппаратные прерывания) через программирование контроллера прерываний. Сейчас прерывания как MSI (message signal interrupts)

- 0-divide error (ошибка деления на 0)
- 8-double fault (если выполнить исключение или маскируемое/немаскрируемое прерывание и возникла ошибка (паника…), завершается работа компьютера)
- 11-segment not present (сегмент отсутствует – надо выполнить определенные действия, чтобы сделать сегмент доступным. Касается управления памятью (нашей программе-не очень))
- 13-general protection (общая защита, должно быть обработано специальным образом. На все исключения-заглушки (double fault-не искл.), а на 13-специальная заглушка (у РФ отражено в структуре таблицы дескрипторов прерываний-без dup))
  (нарушение общей защиты (нарушение, код ошибки-та команда), происходит: за пределами сегмента, запрет чтения, за гр. таблицы дескр., int с отс. Номером)
- 14-page fault (fault переводится как исключение, но по-русски здесь прерывание) (страничное прерывание) – обращение к команде/данным, отсутствующим в программе – система должна загрузить нужную страницу. В CR2 адрес, на котором произошло прерывание)

Вектор	Название исключения	Класс исключения	Код ошибки	Команды, вызывающие исключение
0	Ошибка деления	Нарушение	Нет	div, idiv
1	Исключение отладки	Нарушение /ловушка	Нет	Любая команда
2	Немаскируемое прерывание			
3	int 3	Ловушка	Нет	int 3
4	Переполнение	Ловушка	Нет	into
5	Нарушение границы массива	Нарушение	Нет	bound
6	Недопустимый код команды	Нарушение	Нет	Любая команда
7	Сопроцессор недоступен	Нарушение	Нет	esc, wait
8	Двойное нарушение	Авария	Да	Любая команда
9	Выход сопроцессора из сегмента(80386)	Авария	Нет	Команда сопроцессора с обращением к памяти
10	Недопустимый сегмент состояния задачи TSS	Нарушение	Да	jmp, call, iret, прерывание
11	Отсутствие сегмента	Нарушение	Да	Команда загрузки сегментного регистра
12	Ошибка обращения к стеку	Нарушение	Да	Команда обращения к стеку
13	Общая защита	Нарушение	Да	Команда обращения к памяти
14	Страничное нарушение	Нарушение	Да	Команда обращения к памяти
15	Зарезервировано			
16	Ошибка сопроцессора	Нарушение	Нет	esc, wait
17	Ошибка выравнивания	Нарушение	Да	Команда обращения к памяти
18...31 Зарезервированы				
32...255 Предоставлены пользователю для аппаратных прерываний и команд int				

Типы шлюзов.

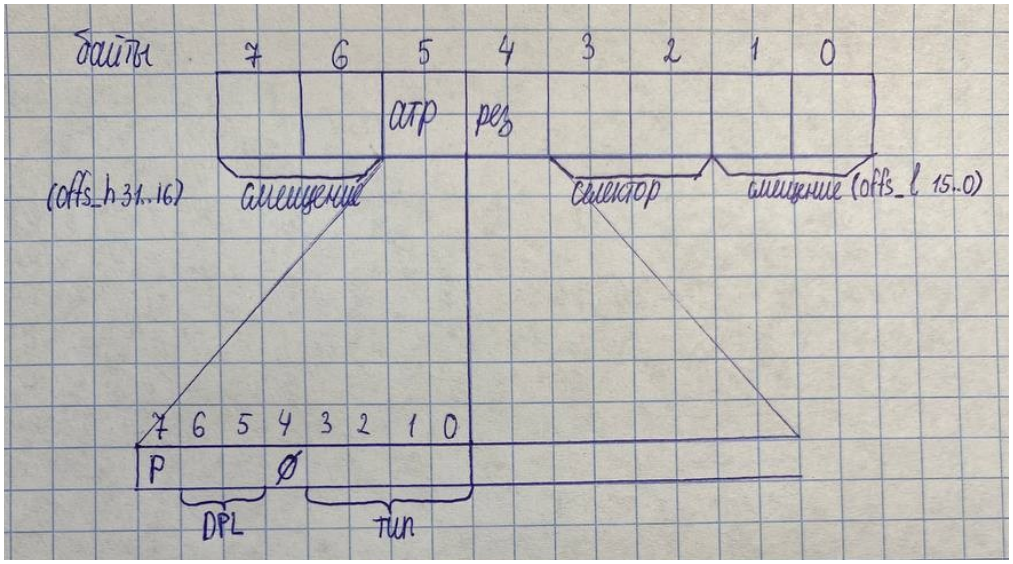
1. Ловушки (trap gate) - обработка исключений и программных прерываний (системных вызовов).

2. Прерывания (interrupt gate) - обработка аппаратных прерываний.

3. Задач (task gate)- *переключение задач в многозадачном режиме.*

Формат дескриптора прерывания

Формат дескриптора (шлюза) для IDT (в скобках – из учебника)



- Байты 0-1 (offs_1), 6-7 (offs_h): 32-битное смещение обработчика
- Байты 2-3 (sel): селектор (сегмента команд) (итого полный 3-хсловный адрес обработчика селектор: смещение)
- Байт 4 зарезервирован
- Байт 5: байт атрибутов - как в дескрипторах памяти за исключением типа:

  Типы: назначение:
  - 0-не определен
  - 1-свободный сегмент состояния задачи TSS 80286
  - 2-LDT
  - 3-занятый сегмент состояния задачи TSS 80286
  - 4-шлюз вызова Call Gate 80286
  - 5-шлюз задачи Task Gate
  - 6-шлюз прерываний Interrupt Gate 80286
  - 7-шлюз ловушки Trap Gate 80286
  - 8-не определен
  - 9- свободный сегмент состояния задачи TSS 80386+
  - Ah-не определен
  - Bh- занятый сегмент состояния задачи TSS 80386+
  - Ch-шлюз вызова Call Gate 80386+
  - Dh- не определен
  - Eh-шлюз прерываний Interrupt Gate 80386+
  - Fh- шлюз ловушки Trap Gate 80386+

Может принимать 16 значений, но в IDT допустимо 5: 5(задачи), 6(прерываний 286), 7(ловушки 286), Eh(прерываний 3/486), Fh(ловушки 3/486) (это по РФ)

  - 4-0 (а вообще это S - system (0 - системный объект, 1 - обычный)
  - 5-6-DPL - уровень привилегий (0 - уровень привилегий ядра, 3 - пользовательский/приложений, *1-2 - не используется в системах общего назначения*)

- 7-1 (а вообще это P - бит присутствия (1 - если сегмент в оперативной памяти, 0 - иначе)

Пример заполнения IDT из лабораторной работы.

```
;Структура idescr для описания дескрипторов (шлюзов) прерываний
idescr struc
    offs_l  dw 0
    sel     dw 0
    cntr    db 0
    attr    db 0
    offs_h  dw 0
idescr ends
```

```
;Таблица дескрипторов прерываний IDT
;Дескриптор: <offs_l, sel, rsrv, attr, offs_h>
;смещение позже?, селектор 32-разрядного сегмента кода
idt label byte

; Первые 32 элемента таблицы - под исключения-внутренние прерывния процессора
;(реально-18, остальные-зарезервированы)
;attr=8Fh: тип=ловушка 386/486(обр. программные пр. и искл., IF не меняется),
;системный объект, УП ядра, P=1
idescr_0_12 idescr 13 dup (<0,code32s,0,8Fh,0>)
; исключение 13 - нарушение общей защиты (нарушение, код ошибки-та команда)
; происходит: за пределами сегмента, запрет чтения, за гр. таблицы дескр., int с отс. номером
idescr_13 idescr <0,code32s,0,8Fh,0>
idescr_14_31 idescr 18 dup (<0,code32s,0,8Fh,0>)


; Затем 16 векторов аппаратных прерываний,
;attr=8Eh: тип=прерывание 386/486(обр. аппаратные пр., IF сбрасывается a iret восстанавливает),
;системный объект, УП ядра, P=1
;Дескриптор прерывания от таймера
int08 idescr <0,code32s,0,8Eh,0>
int09 idescr <0,code32s,0,8Eh,0>

idt_size = $-idt        ;размер
ipdescr df 0            ;псевдодескриптор
ipdescr16 dw 3FFh, 0, 0 ;содержимое регистра IDTR в РР: с адреса 0, 256*4=1кб=2^10
```

```
; Заносим в дескрипторы прерываний (шлюзы) смешение обработчиков прерываний.
lea eax, es:except_13
mov idescr_13.offs_l, ax
shr eax, 16
mov idescr 13.offs h, ax
```