

1. 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. Обработчики аппаратных прерываний: регистрация в системе, примеры. Верхние и нижние половины обработчиков прерываний. Нижние половины: тасклеты и очереди работ — объявление, создание, постановка работы в очередь, планирование (пример лаб. раб).

1.1. Специальные файлы устройств

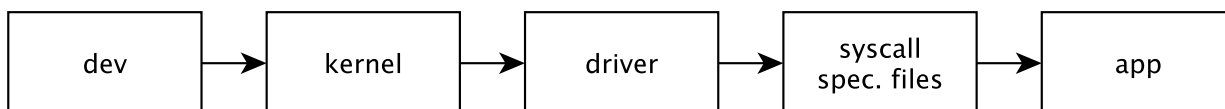
Специальные файлы устройств обеспечивают унифицированный доступ к периферийным (внешним) устройствам. Устройства, как и всё в UNIX, представлены файлами, что обеспечивает доступ к устройствам, как к обычным файлам — можно открывать/закрывать/читать/писать. В отличие от обычных файлов, специальные файлы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

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

Файлы устройств связывают файловую систему с драйверами устройств. Каждому устройству соответствует хотя бы 1 специальный файл. Обычно они лежат в `/dev` кор-

невой ФС. Подкаталоги `/dev/fd` содержат файлы с именами 0, 1, 2 (в некоторых системах `stdio`, `stdout`, `stderr...`). Со всеми устройствами система работает одинаково.

Использование СФУ для работы с аппаратной частью:



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

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

Имеется 2 типа файлов устройств:

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

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

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

На структуры, описывающие устройства, ссылается `inode`:

```
1  struct inode {  
2      ...  
3      union {  
4          ...
```

```

5      struct block_device      *i_bdev;
6      struct cdev              *i_cdev;
7      ...
8  };
9      ...
10 } __randomize_layout;

```

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

```

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

```

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

```

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

```

Чтобы сделать блочные устройства доступными для ядра, драйверы блочных устройств регистрируются в ядре с помощью функции `register_blkdev`, но с версии 2.6.0 этот вызов необязателен и нужен только для динамического выделения

старшего номера и создания записи в `/proc/devices`.

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

Устройство — специальный файл, не можем идентифицировать его как обычный файл. В ядре используется тип `dev_t` (`<linux/types.h>`) для того, чтобы определять (содержать) номера устройств. С версии 2.6.0 `dev_t` 32-разрядный. Представляет число, в котором 12 бит — для старшего номера, 20 — для младшего. POSIX.1 определяет существование типа, но не его формат.

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

Код пользователя не должен делать предположений о внутренней организации номера устройства, а должен просто использовать набор макросов

`<linux/kdev_t.h>`, используемых для получения старшего и младшего номеров устройства.

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

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

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

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

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

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

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

1.3. Выделение/ освобождение номеров устройств

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

Для выделения символического устройства нужно вызвать функцию

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

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

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

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

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

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

Есть тенденция по переходу на использование динамически разделяемых номеров устройств:

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

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

Для освобождения используется:

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

1.4. Статическое выделение номеров устройств

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

Старш. №	Тип устройства
1	Оперативная память
2	Дисковод флоппу
3	1-ый контроллер жёстких IDE-дисков
4	Терминалы
5	Терминалы
6	Принтеры (параллельный код)
8	Жёсткие SCSI-диски
13	Мышь
14	Звуковые карты
22	2-ой контроллер жёстких IDE-дисков

1.5. Система прерываний: типы прерываний и их особенности

Система прерываний включает в себя:

1. Системные вызовы (синхронные — возникают в процессе выполнения программы, вызываются соответствующей командой).
2. Исключения (синхронные — возникают в процессе выполнения программы, переполнение стека/деление на 0...).
3. Аппаратные прерывания (асинхронные — не зависят ни от каких действий в системе, выполняются на высоких уровнях приоритетов, их выполнение нельзя прервать, задача — информирование процессов о событиях в системе, от системного таймера/клавиатуры...).

Про таблицы прерываний:

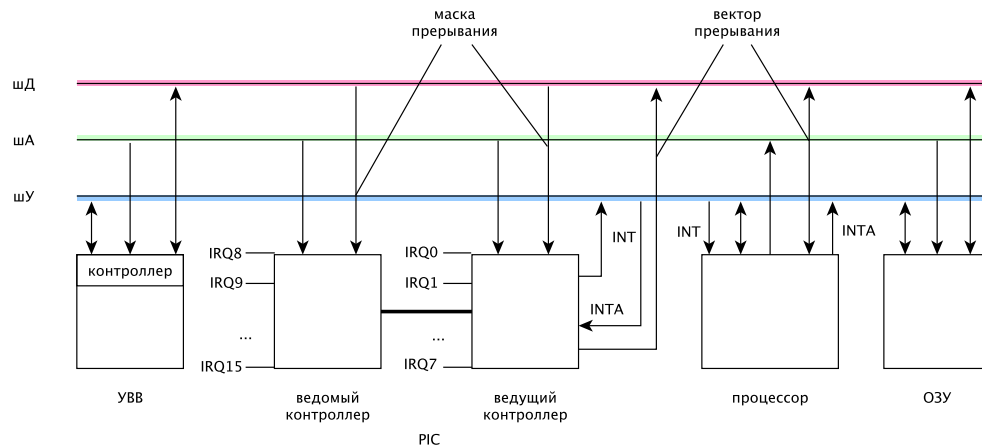
В 16-разрядной ОС существует таблица векторов прерываний — таблица, содержащая векторы прерывания (far-адреса обработчиков прерывания). Она начинается с нулевого адреса, занимает 1 Мб. Вектор прерывания - смещение в этой таблице.

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

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

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

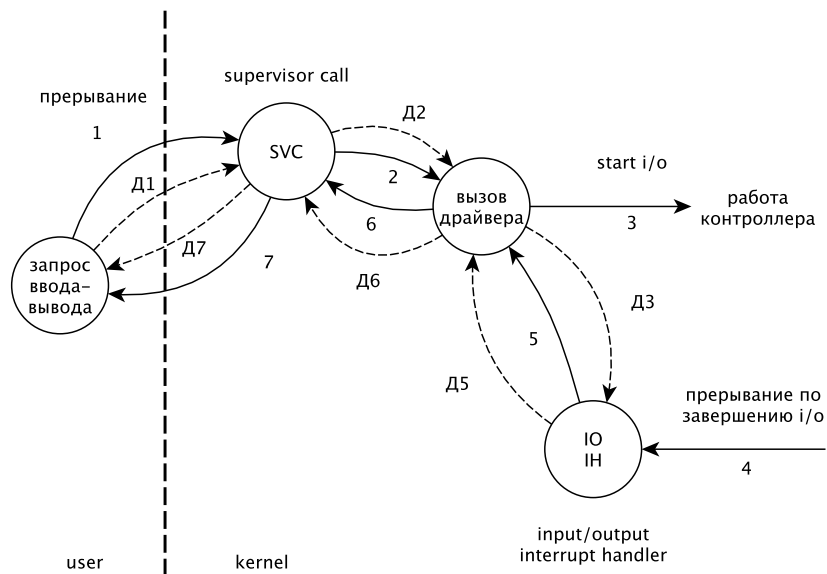
1.6. Прерывания в последовательности ввода-вывода — обслуживание запроса процесса на ввод-вывод



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

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

8. Процессор по шине данных отправляет в контроллер маску прерываний.



Прерывания, в зависимости от возможности запрета, делятся на:

Маскируемые — прерывания, которые можно запрещать установкой соответствующего флага ('IF' - Interruption Flag). Если он сброшен, то обработка прерываний запрещена. Если он установлен, то прерывания будут обрабатываться. Здесь речь идёт об аппаратных прерываниях, потому что программные запрещать нет смысла.

Немаскируемые (англ. Non-maskable interrupt, NMI) — обрабатываются всегда, независимо от запретов на другие прерывания

1.7. Быстрые и медленные прерывания

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

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

Аппаратные прерывания — важнейшие в системе, поэтому, чтобы исключить в многопроцессорных системах ошибки обработки данных, поступающих от внешних устройств,

на том ядре, на котором обрабатывается прерывание, запрещаются все остальные прерывания, а в системе запрещаются все прерывания по данной линии IRQ. Никакая другая работа на данном процессоре выполняться не может, что негативно сказывается на производительности и отзывчивости системы, поэтому АП не могут выполняться длительное время. Поэтому работа, которую выполняют обработчики прерываний, делится на 2 части.

Эти 2 части — `top_half` и

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

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

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

Системная функция для регистрации обработчика прерывания:

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

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

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

Пример:

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

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

1.9. Тасклеты — объявление, планирование

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

Тасклет не может выполняться параллельно, выполняется атомарно на том же процессоре, на котором выполняется обработчик вызвавшего его прерывания. Инициализируется как статически, так и динамически. Выполняется в контексте прерывания — нельзя использовать средства взаимного исключения, кроме `spinlocks`. Выполняется `ksoftirqd`. Является упрощённым интерфейсом `softirq`.

Определена структура:

```
1 struct tasklet_struct
2 {
3     struct tasklet_struct *next;
4     unsigned long state;
5     atomic_t count;
6     bool use_callback;
7     union {
8         void (*func)(unsigned long data);
9         void (*callback)(struct tasklet_struct *t);
10    };
11    unsigned long data;
12 };
```

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

```
1 #define DECLARE_TASKLET(name, _callback) \
2 struct tasklet_struct name = { \
3     .count = ATOMIC_INIT(0), \
4     .callback = _callback, \
5     .use_callback = true, \
6 }
```

```

7 #define DECLARE_TASKLET_DISABLED(name, _callback) \
8 struct tasklet_struct name = {          \
9     .count = ATOMIC_INIT(1),          \
10    .callback = _callback,              \
11    .use_callback = true,               \
12 }

```

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

```

1 DECLARE_TASKLET(my_tasklet, tasklet_handler);

```

Эта строка эквивалентна следующему объявлению:

```

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

```

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

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

```

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

```

Пример:

```

1 tasklet_init(t, tasklet_handler, data);

```

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

```

1 tasklet_schedule(struct tasklet_struct *t);
2 tasklet_hi_scheduler(struct tasklet_struct *t);

```

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

Свойства:

1. Если вызывается функция `tasklet_schedule()`, то после этого тасклет гарантированно будет выполнен на каком-либо процессоре хотя бы один раз.
2. Если тасклет уже запланирован, но его выполнение все еще не запущено, он будет выполнен только один раз.
3. Если этот тасклет уже запущен на другом процессоре (или `schedule` вызывается из самого тасклета), выполнение переносится на более поздний срок.
4. Тасклет строго сериализован по отношению к самому себе, но не по отношению к другим тасклетам. Если разработчик считает, что в данном тасклете нужно выполнить действия, которые могут выполняться в других таскетах, он реализует взаимное исключение с помощью `spinlocks`.

Сравнение (лучше в таблицу)

softirq

- Определение — 10 шт. определено в ядре статически
- Взаимоискл. — Да
- Возможность паралл. вып. — Да
- Выполнение — В контексте спец. потоков ядра (`ksoftirqd`)
- Блокируются — Наверное нет

tasklet

- Определение — статически и динамически
- Взаимоискл. — Нет
- Возможность паралл. вып. — Нет
- Выполнение — В контексте запланировавшего обраб. прерыв.
- Блокируются — Нет

workqueue

- Определение — статически и динамически
- Взаимоискл. — Да
- Возможность паралл. вып. — Да
- Выполнение — В контексте спец. потоков ядра (kworker)
- Блокируются — Да

1.10. spin_lock

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

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

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

```

1  void spin_lock(spin_lock_t* c)
2  {
3      while (test_and_set(*c) != 0)
4      {
5          /* ресурсзанят */
6
7          /*
8           while (*c != 0)
9           {
10              ...
11              }
12          */
13      }
14  }
15

```

```

16  void spin_unlock(spin_lock_t* c)
17  {
18      *c = 0;
19  }

```

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

Пример из лабораторной:

```

1  #include <linux/kernel.h>
2  #include <linux/module.h>
3  #include <linux/interrupt.h>
4  #include <linux/slab.h>
5  #include <asm/io.h>
6
7  #include "ascii.h"
8
9  #define IRQ_NUMBER 1
10
11 MODULE_LICENSE("GPL");
12 MODULE_AUTHOR("Karpova_Ekaterina");
13
14 struct tasklet_struct *tasklet;
15 char tasklet_data[] = "key_pressed";
16
17 void tasklet_func(unsigned long data)
18 {
19     printk(KERN_INFO "+:_____");
20     printk(KERN_INFO "+:tasklet_began");
21     printk(KERN_INFO "+:tasklet_count=%u", tasklet->count.counter);
22     printk(KERN_INFO "+:tasklet_state=%lu", tasklet->state);
23
24     printk(KERN_INFO "+:key_code=%d", data);
25     if (data < ASCII_LEN)
26         printk(KERN_INFO "+:key_press=%s", ascii[data]);
27     if (data > 128 && data < 128 + ASCII_LEN)

```

```

28         printk(KERN_INFO "+:_key_release_-%s", ascii[data - 128]);
29
30     printk(KERN_INFO "+:_tasklet_ended");
31     printk(KERN_INFO "+:_-----");
32 }
33
34 static irqreturn_t my_irq_handler(int irq, void *dev_id)
35 {
36     int code;
37     printk(KERN_INFO "+:_my_irq_handler_called\n");
38
39     if (irq != IRQ_NUMBER)
40     {
41         printk(KERN_INFO "+:_irq_not_handled");
42         return IRQ_NONE;
43     }
44
45     printk(KERN_INFO "+:_tasklet_state_(before_schedule)_=%lu",
46           tasklet->state);
47     code = inb(0x60);
48     tasklet->data = code;
49     tasklet_schedule(tasklet);
50     printk(KERN_INFO "+:_tasklet_scheduled");
51     printk(KERN_INFO "+:_tasklet_state_(after_schedule)_=%lu",
52           tasklet->state);
53
54     return IRQ_HANDLED;
55 }
56
57 static int __init my_init(void)
58 {
59     if (request_irq(IRQ_NUMBER, my_irq_handler, IRQF_SHARED, "
60         tasklet_irq_handler", (void *) my_irq_handler))
61     {
62         printk(KERN_ERR "+:_cannot_register_irq_handler\n");
63         return -1;
64     }

```



```

65 tasklet = kmalloc(sizeof(struct tasklet_struct), GFP_KERNEL);
66
67 if (tasklet == NULL)
68 {
69     printk(KERN_ERR "+:_kmalloc_error");
70     return -1;
71 }
72
73 tasklet_init(tasklet, tasklet_func, (unsigned long)tasklet_data);
74
75 printk(KERN_INFO "+:_module_loaded\n");
76 return 0;
77 }
78
79 static void __exit my_exit(void)
80 {
81     tasklet_kill(tasklet);
82     free_irq(IRQ_NUMBER, my_irq_handler);
83
84     printk(KERN_INFO "+:_module_unloaded\n");
85 }
86
87 module_init(my_init);
88 module_exit(my_exit);

```

1.11. Workqueue

Очереди работ — еще один тип bottom_half (отложенного действия). Кроме очередей, которые мы создаём, есть общесистемные очереди и соответствующие возможности для работы с этими очередями. Определена структура в <linux/workqueue.h>.

```

1  struct workqueue_struct {
2  struct list_head pwqs;    /* WR: all pwqs of this wq */
3  struct list_head list;    /* PR: list of all workqueues */
4
5  struct mutex mutex;    /* protects this wq */

```

```

6  int          work_color;  /* WQ: current work color */
7  int          flush_color; /* WQ: current flush color */
8  atomic_t     nr_pwqs_to_flush; /* flush in progress */
9  struct wq_flusher *first_flusher; /* WQ: first flusher */
10 struct list_head flusher_queue; /* WQ: flush waiters */
11 struct list_head flusher_overflow; /* WQ: flush overflow list */
12
13 struct list_head maydays; /* MD: pwqs requesting rescue */
14 struct worker    *rescuer; /* MD: rescue worker */
15
16 int          nr_drainers; /* WQ: drain in progress */
17 int          saved_max_active; /* WQ: saved pwq max_active */
18
19 struct workqueue_attrs *unbound_attrs; /* PW: only for unbound wqs */
20 struct pool_workqueue *dfl_pwq; /* PW: only for unbound wqs */
21
22 #ifdef CONFIG_SYSFS
23     struct wq_device *wq_dev; /* I: for sysfs interface */
24 #endif
25 #ifdef CONFIG_LOCKDEP
26     char          *lock_name;
27     struct lock_class_key key;
28     struct lockdep_map lockdep_map;
29 #endif
30     char          name[WQ_NAME_LEN]; /* I: workqueue name */
31
32     /*
33      * Destruction of workqueue_struct is RCU protected to allow walking
34      * the workqueues list without grabbing wq_pool_mutex.
35      * This is used to dump all workqueues from sysrq.
36      */
37     struct rcu_head rcu;
38
39     /* hot fields used during command issue, aligned to cacheline */
40     unsigned int    flags ____cacheline_aligned; /* WQ: WQ* flags */
41     struct pool_workqueue __percpu *cpu_pwqs; /* I: per-cpu pwqs */
42     struct pool_workqueue __rcu *numa_pwq_tbl[]; /* PWR: unbound pwqs
        indexed by node */

```

На процессор имеется 2 kworker: normal и high. Это соответствует созданию очередей с нормальным и высоким уровнем приоритета, вторые будут выполняться раньше, но это разные воркеры.

worker — рабочий поток ядра
(worker_thread).

Все воркеры выполняются как обычные потоки ядра, при этом они выполняют функцию worker_thread().

1. После инициализации worker_thread() она входит в бесконечный цикл и засыпает (прерываемый сон).
2. Она просыпается, когда в очередь ставится какая-то работа и начинает выполнять ее.
3. Закончив выполнение работы, поток опять засыпает.

В системе также есть структура

```

1  /*
2  * Очередьотложенныхдействий , связаннаяспроцессором :
3  */
4  struct cpu_workqueue_struct
5  {
6      spinlock_t lock; /* Очередьдлязащитыданнойструктуры */
7      long remove_sequence; /*последнийдобавленныйэлементследующий
8
9      ( длязапуска
10     ) */
11     long insert_sequence; /* следующийэлементдлядобавления
12     */
13     struct list_head worklist; /* списокдействийнакаждое
14     cpu */
15     wait_queue_head_t more_work;
16     wait_queue_head_t work_done;
17     struct workqueue_struct *wq; /* соответствующаяструктура
18     workqueue_struct */
19     task_t *thread; /* соответствующийпотокфункция
20     () */
21     int run_depth; /* глубинарекурсиифункции

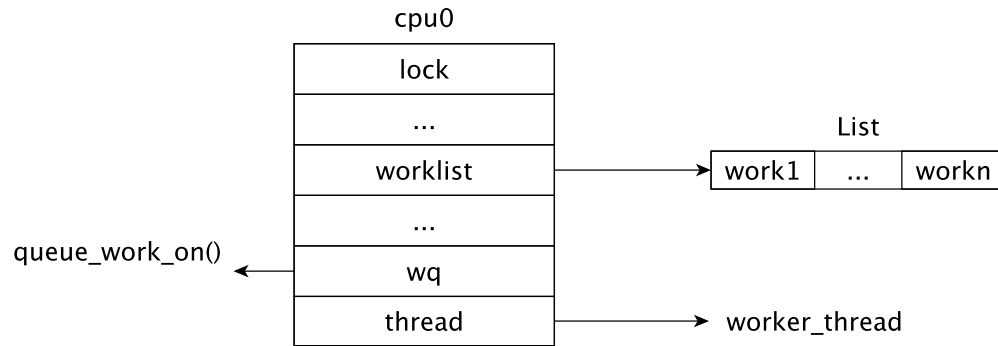
```

```

22     run_workqueue() */
23 };

```

Иллюстрация для `cpu0` (для всех `cpu` одинаково):



`wq` — указатель на инициализированную структуру очереди работ.

Работа описывается структурой (1 экземпляр на каждое отложенное действие):

```

1 typedef void (*work_func_t) (struct work_struct *work);
2 struct work_struct {
3     atomic_long_t data;
4     struct list_head entry;
5     work_func_t func; // обработчик работы
6 #ifdef CONFIG_LOCKDEP
7     struct lockdep_map lockdep_map;
8 #endif
9 };

```

В современных системах очередь работ создается функцией `alloc_workqueue()`:

```

1 struct workqueue_struct *alloc_workqueue(const char *fmt,
2     unsigned int flags,
3     int max_active, ...);

```

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

Кроме этой, есть еще функция:

```

1 #define alloc_ordered_workqueue(fmt, flags, args...) \
2     alloc_workqueue(fmt, WQ_UNBOUND | __WQ_ORDERED | \

```

`create_workqueue()` — устаревшая.

Несколько объектов, связанных с очередью работ (`workqueue`), представлены в ядре соответствующими структурами:

1. Работа (`work`);
2. Очередь работ (`workqueue`) — коллекция `work`. `Workqueue` и `work` относятся как один-ко-многим;
3. Рабочий (`worker`). `Worker` соответствует потоку ядра `worker_thread`;
4. Пул рабочих потоков (`worker_pool`) это — набор рабочих (`worker`).
`Worker_pool` и `worker` относятся как «один ко многим»;
5. `Pwd` (`pool_workqueue`) это — посредник, который отвечает за отношение `workqueue` и `worker_pool`: `workqueue` и `pwd` является отношением один-ко-многим, а `pwd` и `worker_pool` — отношение один-к-одному.

Для начала пулы для привязанных очередей (на картинке). Для каждого CPU статически выделяются два `worker pool`: один для высокоприоритетных `work`'ов, другой — для `work`'ов с нормальным приоритетом. То есть, если ядра у нас четыре, то привязанных пулов будет всего восемь, не смотря на то, что `workqueue` может быть сколько угодно.

Когда мы создаем `workqueue`, у него для каждого CPU выделяется служебный `pool_workqueue` (`pwq`). Каждый такой `pool_workqueue` ассоциирован с `worker pool`, который выделен на том же CPU и соответствует по приоритету типу очереди. Через них `workqueue` взаимодействует с `worker pool`.

`Worker`'ы исполняют `work`'и из `worker pool` без разбора, не различая, к какому `workqueue` они принадлежали изначально.

Для непривязанных очередей `worker pool`'ы выделяются динамически.

1.12. Флаги

1. `WQ_UNBOUND` — очередь работ не связана ни с каким сри.
2. `WQ_FREEZABLE` — очередь работ может блокироваться.

3. WQ_HIGHPRI — задания, представленные в такую очередь, будут поставлены в начало очереди и будут выполняться (почти) немедленно.
4. WQ_CPU_INTENSIVE — очередь работ может интенсивно использовать процессор.
5. WQ_POWER_EFFICIENT — очереди работ, связанные с конкретным процессором, являются более предпочтительными, так как имеют более высокую производительность благодаря локальному кешу. Но такая привязанность к конкретному процессору имеет отрицательный побочный эффект — увеличение энергопотребления.

Продолжение про
WQ_POWER_EFFICIENT

Это связано с тем, что если энергия не тратится, процессор простаивает.

Такие соображения учитываются в любой системе.

Таким образом, если нет `workqueue`, привязанных к конкретному процессору, можно «не трогать» этот неработающий процессор и, если позволяют соответствующие условия, выполнять эту работу на том процессоре, который работает.

Фактически мы можем ограничиться одним процессором...

1.13. Создание работы

Мы создаем очередь работ для того, чтобы поместить в нее нужные нам действия (работы).

Существует статический способ определения работы с помощью макроса

Листинг 1..1: Статический способ определения

```
1 DECLARE_WORK(name, void (*func)(void*)) ;
2 struct work_struct *name; // имяэкземпляра () структуры work_struct
```

`func` — имя функции, которая вызывается из `workqueue` (код для выполнения отложенных действий, нижняя половина).

Работу можно определить динамически т.е. в процессе выполнения программы:

Листинг 1..2: Динамический способ определения

```
1 #define INIT_WORK(_work, func) __INIT_WORK((_work, (func), 0)
```

Листинг 1..3: Определение `__INIT_WORK()`

```

1  __INIT_WORK(( _work, _func, _onstack)
2  do {
3      ...
4      INIT_LIST_HEAD(&(work)->entry);
5      (_work)->func=(-func);
6  } while (0);

```

Также может встретиться функция `PREPARE_WORK()`.

`INIT_WORK()` производит инициализацию работ более тщательно (советую Вам использовать её), а `PREPARE_WORK()` используется, если работа была инициализирована, для сокращения времени повторной инициализации.

1.14. Постановка работы в очередь

Работа поставлена в очередь — она попадает в список и через какое-то время (не мгновенно) будет выполнена.

Инициализировав работу, требуется поставить ее в очередь. Это делает функция:

```

1 bool queue_work(struct workqueue_struct *queue, struct work_struct *work);

```

`queue` — очередь, в которую мы хотим поставить работу; `work` — инициализированная структура `work_struct`.

Также есть

```

1 bool queue_delayed_work(..., unsigned long delay); // ... — те же два параметра

```

`delay` — время, через которое данная работа может быть поставлена в очередь. Определяет количество `jiffies` («мгновений»).

`queue_work()` определена через `queue_work_on()`.

1.15. Завершение работы/очереди работ

Принудительное завершение очереди работ:

```

1 extern void _flush_workqueue(struct workqueue_struct *wq);

```

Принудительно завершить работу и заблокировать прочую обработку прежде, чем работа будет закончена:

```
1 extern bool flush_work(struct work_struct *work);
```

Чтобы абсолютно точно отменить выполнение (отменить работу, если она еще не выполнена обработчиком; завершит работу в очереди, либо возникнет блокировка до тех пор, пока не будет завершен обратный вызов (если работа уже выполняется обработчиком)), советуют использовать:

```
1 extern bool cancel_work (...);
```

Если работа отложена, вы можете использовать функции `flush_delayed_work()` и `cancel_delayed_work()`.

Пример:

```
1 #include <linux/module.h>
2 #include <linux/kernel.h>
3 #include <linux/interrupt.h>
4 #include <linux/slab.h>
5 #include <asm/io.h>
6 #include <linux/stddef.h>
7 #include <linux/workqueue.h>
8 #include <linux/delay.h>
9
10 #include "ascii.h"
11
12 #define IRQ_NUMBER 1
13
14 MODULE_LICENSE("GPL");
15 MODULE_AUTHOR("Karpova_Ekaterina");
16
17 typedef struct
18 {
19     struct work_struct work;
20     int code;
21 } my_work_struct_t;
22
23 static struct workqueue_struct *my_wq;
24
25 static my_work_struct_t *work1;
26 static struct work_struct *work2;
27
28 void work1_func(struct work_struct *work)
```



```

29 {
30     my_work_struct_t *my_work = (my_work_struct_t *)work;
31     int code = my_work->code;
32
33     printk(KERN_INFO "+:_____");
34     printk(KERN_INFO "+:_work1_began");
35
36     printk(KERN_INFO "+:_key_code_-%d", code);
37     if (code < ASCII_LEN)
38         printk(KERN_INFO "+:_key_press_-%s", ascii[code]);
39     if (code > 128 && code < 128 + ASCII_LEN)
40         printk(KERN_INFO "+:_key_release_-%s", ascii[code - 128]);
41
42     printk(KERN_INFO "+:_work1_ended");
43     printk(KERN_INFO "+:_____");
44 }
45
46 void work2_func(struct work_struct *work)
47 {
48     printk(KERN_INFO "+:_____");
49     printk(KERN_INFO "+:_work2_began");
50     msleep(10);
51     printk(KERN_INFO "+:_work2_ended");
52     printk(KERN_INFO "+:_____");
53 }
54
55 irqreturn_t my_irq_handler(int irq, void *dev)
56 {
57     int code;
58     printk(KERN_INFO "+:_my_irq_handler_called\n");
59
60     if (irq != IRQ_NUMBER)
61     {
62         printk(KERN_INFO "+:_irq_not_handled");
63         return IRQ_NONE;
64     }
65
66     code = inb(0x60);

```

```

67     work1->code = code;
68
69     queue_work(my_wq, (struct work_struct *)work1);
70     queue_work(my_wq, work2);
71
72     return IRQ_HANDLED;
73 }
74
75 static int __init my_init(void)
76 {
77     int rc = request_irq(IRQ_NUMBER, my_irq_handler, IRQF_SHARED, "
        work_irq_handler", (void *) my_irq_handler);
78     if (rc) {
79         printk(KERN_ERR "+:_cannot_register_irq_handler");
80         return rc;
81     }
82
83     my_wq = alloc_workqueue("%s", __WQ_LEGACY | WQ_MEM_RECLAIM, 1, "my_wq"
        );
84     if (my_wq == NULL)
85     {
86         printk(KERN_ERR "+:_cannot_alloc_workqueue");
87         return -1;
88     }
89
90     work1 = kmalloc(sizeof(my_work_struct_t), GFP_KERNEL);
91     if (work1 == NULL)
92     {
93         printk(KERN_ERR "+:_cannot_alloc_work1");
94         destroy_workqueue(my_wq);
95         return -1;
96     }
97
98     work2 = kmalloc(sizeof(struct work_struct), GFP_KERNEL);
99     if (work2 == NULL)
100    {
101        printk(KERN_ERR "+:_cannot_alloc_work2");
102        destroy_workqueue(my_wq);

```

```

103         kfree (work1);
104         return -1;
105     }
106
107     INIT_WORK((struct work_struct *)work1, work1_func);
108     INIT_WORK(work2, work2_func);
109
110     printk(KERN_INFO "+:~module_loaded");
111
112     return 0;
113 }
114
115 static void __exit my_exit(void)
116 {
117     synchronize_irq (IRQ_NUMBER);
118     free_irq (IRQ_NUMBER, my_irq_handler);
119
120     flush_workqueue (my_wq);
121     destroy_workqueue (my_wq);
122     kfree (work1);
123     kfree (work2);
124
125     printk(KERN_INFO "+:~module_unloaded");
126 }
127
128 module_init (my_init);
129 module_exit (my_exit);

```

Обобщенные загадки:

Линия ирк — линия на контроллерах прерываний, по этим линиям передается сигнал о прерывании от контроллера прерывания ВУ

К /proc/interrupts инфа о прерываниях и их обработчиках

Колонки: номер линии ирк, счетчик прерываний на каждом ядре процессора, вид прерывания, названия обработчиков прерывания (i8042 номер контроллера пс/2 клавиатуры).

Что сделать чтобы установить обработчик прерывания? — вызываем request_irq и описать параметры:

irq - номер линии прерывания
 handler - указатель на
 функцию-обработчик прерывания
 flags - флаги обработки (IRQF_SHARED — указывает, что данная линия ирк может
 быть разделена)
 name - имя устройства
 dev_id - указатель на void (у нас - на обработчик прерывания), используется, чтобы
 освободить линию ирк от обработчика прерывания
 Что сделать чтобы заработала очередь работ? — зарегистрировать обработчик преры-
 вания, создать очередь, создать и инициализировать работы и в обработчике поставить
 работу в очередь.
 Как инициализируем работы? — у нас - динамически, что значит? во время выполнения
 программы
 Когда вызывается обработчик прерывания ps/2? — когда (отпускаем или) нажимаем
 клавишу — ЯВЛЯЕТСЯ ТОЧКОЙ ВХОДА
 В обработчике прерывания для работ: получили аргумент (код нажатой клавиши),
 поставили работы в очередь (1 - печатает символ и код, 2 - спит)
 В обработчике прерывания для тасклет: получили аргумент (код нажатой клавиши),
 запланировали тасклет на выполнение, вызвав функ планировщика (1 - печатает символ
 и код, 2 - спит)
 Что делает тасклет? — преобразует код клавиши в символ
 Кто запускает кворкеров? — стандартный планировщик, это видим в лабе на кольцевой
 список процессов, см. поле полиси и прио в таск_стракт.