Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа №9

По предмету: «Операционные системы»

Тема: Обработчики прерываний

Преподаватель: Рязанова Н.Ю.
Студент: Гасанзаде М.А.,
Группа: ИУ7-66Б

Москва, 2020 г.

ОГЛАВЛЕНИЕ

ЗАДАНИЕ 1

- Написать загружаемый модуль ядра, в котором зарегистрировать обработчик аппаратного прерывания с флагом IRQF_SHARED.
- Инициализировать тасклет.
- В **обработчике прерывания** запланировать тасклет на выполнение.
- Вывести информацию о тасклете используя, или printk(), или seq_file interface - <linux/seq_file.h> (Jonathan Corber: http://lwn.net//Articales//driver-porting/).

Обработчики медленных прерываний делятся на две части: верхнюю (top) и нижнюю (bottom) половины (half).

В настоящее время нижные половины могут быть трех типов:

- Отложенные прерывания (softirq)
- Тасклеты (tasklets)
- Очереди работ (work queue).

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

```
<linux/interrupt.h>
int request_irq(unsigned int irq,
irqreturn_t(*handler)( int, void *,
                       struct    pt_regs *), unsigned
long irqflags, const char *devname,
                        void *dev_id);
```

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

Флаги:

```
#define IRQF_SHARED     0x00000080 /*разрешает
разделение irq несколькими устройствами*/
#define IRQF_PROBE_SHARED     0x00000100
/*устанавливается абонентами, если возможны проблемы
при совместном использовании irq*/
#define IRQF_TIMER    0x00000200 /*флаг,   маскирующий
данное прерывание как прерывание от таймера*/
```

```
#define IRQF_PERCPU     0x00000400  .*прерывание
закрепленное за определенным процессором*/
#define IRQF_NOBALANCING    0x00000800  /*флаг,
запрещающий использование данного прерывания для
балансировки irq*/
#define IRQF_IRQPOLL     0x00001000  /*прерывание
используется для опроса*/
#define IRQF_ONESHOT     0x00002000
#define IRQF_NO_SUSPEND     0x00004000
#define IRQF_FORCE_RESUME     0x00008000
#define IRQF_NO_THREAD    0x00010000
#define IRQF_EARLY_RESUME     0x00020000
#define IRQF_COND_SUSPEND     0x00040000
```

Флаги были изменены радикально после версии ядра 2.6.19.

Тасклеты

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

```
struct tasklet_struct
 {
    struct tasklet_struct *next;  /* указатель на
следующий тасклет в     списке */
    unsigned long state;          /* состояние тасклета
*/
    atomic_t count;               /* счетчик ссылок */
    void (*func) (unsigned long); /* функция-обработчик
тасклета*/
    unsigned long data; /* аргумент функции-обработчика
тасклета */
);
```

Тасклеты могут быть зарегистрированы как статически, так и динамически.

Статически тасклеты создаются с помощью двух макросов:

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

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

Например.

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

Эта строка эквивалентна следующему объявлению:

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

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

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

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

Тасклеты могут быть запланированы на выполнение с помощью Функций:

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

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

Когда tasklet запланирован, ему выставляется состояние TASKLET_STATE_SCHED, и он добавляется в очередь. Пока он находится в этом состоянии, запланировать его еще раз не получится — в этом случае просто ничего не произойдет. Tasklet не может находиться сразу в нескольких местах в очереди на планирование, которая организуется через поле next структуры tasklet_struct.

После того, как тасклет был запланирован, он выполниться один раз.

Листинг 1 – tasklet.c

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

struct tasklet_struct *tasklet;
int dev_id,scancode, irq = 1;

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Hasanzade M.A.");

#define KBD_DATA_REG 0x60
#define kbd_read_input() inb(KBD_DATA_REG)

void tasklet_function(unsigned long data)
{
    scancode = kbd_read_input();
    if (scancode < 103) {
        printk(KERN_INFO "tasklet: state: %ld, count:
%d, data: %ld\n",
            tasklet->state, tasklet->count, tasklet-
>data);
        printk(KERN_INFO "tasklet: Keycode %d\n",
scancode);
    }

  return;
}

static irqreturn_t my_interrupt(int irq, void *dev_id)
{
    tasklet_schedule(tasklet);


    return IRQ_HANDLED;
}

static int __init module_tasklet_init(void)
{


    if (request_irq(irq, my_interrupt, IRQF_SHARED,
"my_interrupt", &dev_id))
```

```
        return -1;


    tasklet = vmalloc(sizeof(struct tasklet_struct));


    tasklet_init(tasklet, tasklet_function, 0);

    printk(KERN_INFO "tasklet module is loaded.\n");
    return 0;
}

static void __exit module_tasklet_exit(void)
{
  tasklet_kill(tasklet);
  vfree(tasklet);
  free_irq(irq, &dev_id);
  printk(KERN_INFO "tasklet module is unloaded.\n");
  return;
}

module_init(module_tasklet_init);
module_exit(module_tasklet_exit);
```

Результат работы программы

```
wizard@wizard-VM:~/Desktop/Lab_9/tasklet$ make
make -C /lib/modules/5.3.0-51-generic/build M=/home/wizard/Desktop/Lab_9/tasklet
 modules
make[1]: Entering directory '/usr/src/linux-headers-5.3.0-51-generic'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory '/usr/src/linux-headers-5.3.0-51-generic'
wizard@wizard-VM:~/Desktop/Lab_9/tasklet$
```

Рис. 1 – сборка модуля ядра

```
wizard@wizard-VM:~/Desktop/Lab_9/tasklet$ sudo insmod tasklet.ko irq=12
wizard@wizard-VM:~/Desktop/Lab_9/tasklet$ lsmod | grep tasklet
tasklet                16384  0
wizard@wizard-VM:~/Desktop/Lab_9/tasklet$
```

Рис. 2 – загрузка модуля ядра

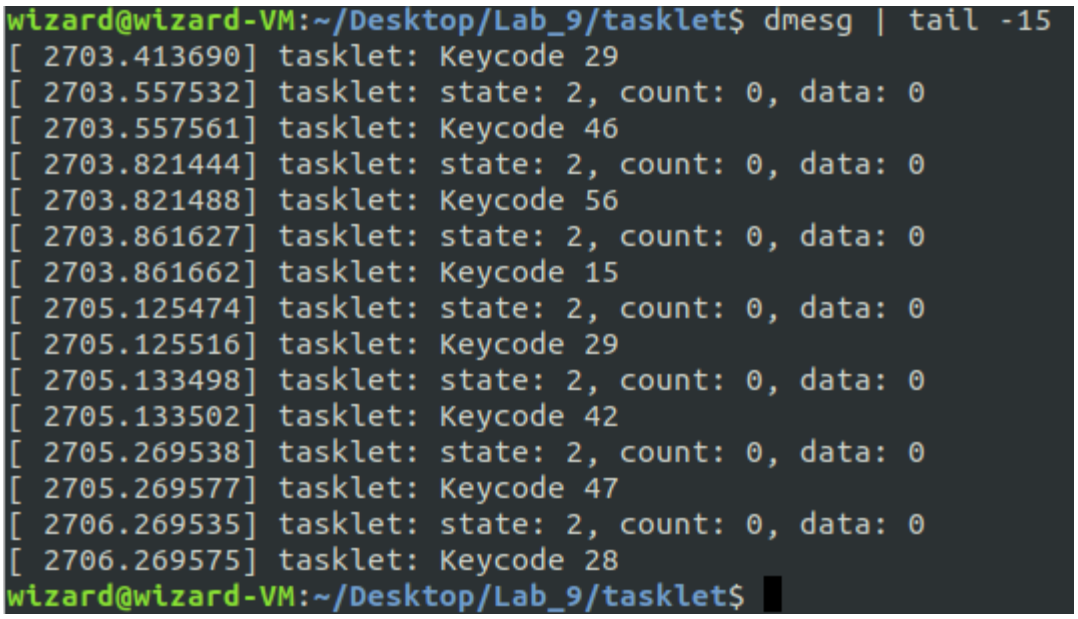

Рис. 3 – последние 15 сообщений, выведенных модулями ядра

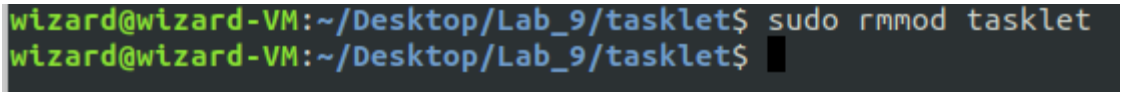

Рис. 4 – выгрузка модуля ядра

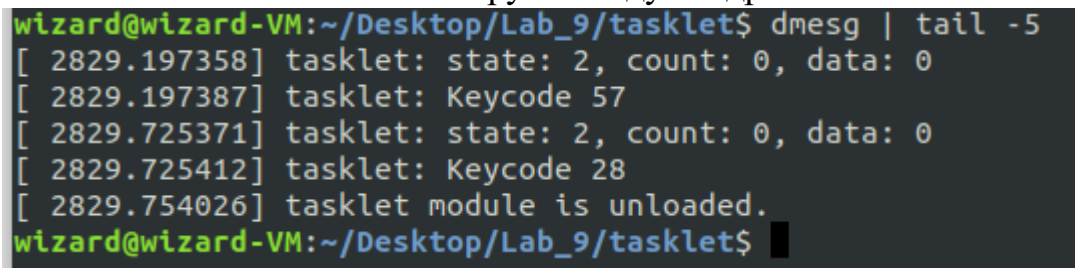

Рис. 5 – последние 5 сообщений, выведенных модулями ядра

ЗАДАНИЕ 2

- Написать загружаемый модуль ядра, в котором зарегистрировать обработчик аппаратного прерывания с флагом IRQF_SHARED.
- Инициализировать очередь работ.
- В обработчике прерывания запланировать очередь работ на выполнение.
- Вывести информацию об очереди работ используя, или printk(), или seq_file interface - <linux/seq_file.h> (Jonathan Corber: http://lwn.net//Articales//driver-porting/).

Очереди работ

Очередь работ создается функцией (см. приложение 1):

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

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

Флаги

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

коде драйвера ATA, который всегда должен иметь возможность запускать свои процедуры завершения ввода-вывода.

- **WQ_HIGHPRI**: задания, представленные в такой workqueue, будут поставлены в начало очереди и будут выполняться (почти) немедленно. В отличие от обычных задач, высокоприоритетные задачи не ждут появления ЦП; они будут запущены сразу. Это означает, что несколько задач, отправляемых в очередь с высоким приоритетом, могут конкурировать друг с другом за процессор.
- **WQ_CPU_INTENSIVE**: имеет смысл только для привязанных очередей. Этот флаг— отказ от участия в дополнительной организации параллельного исполнения. Задачи в такой workqueue могут использовать много процессорного времени. Интенсивно использующие процессорное время worker’ы будут задерживаться.

Листинг 2 – wq.c

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


MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Hasanzade M.A.");

int irq = 1;
int dev_id,scancode;

struct workqueue_struct *que;
struct work_struct *work;

#define KBD_DATA_REG 0x60
#define kbd_read_input() inb(KBD_DATA_REG)

void wq_func(struct work_struct *work) {
    scancode = kbd_read_input();
    if (scancode < 103)
        printk("wq_lab: Keycode %d\n", scancode);
}

static irqreturn_t irq_handler(int irq, void *dev_id) {
    //Планирование на выполнение
    queue_work(que, work);
    return IRQ_HANDLED;
}

static int __init load_module(void) {

    //irq— номер линии запрашиваемого прерывания;
    //handler— указатель на функцию-обработчик типа
irqreturn_t;
    //flags- битовая маска опций, связанная с
управлением прерыванием;
            //IRQF_SHARED— разрешить разделение
(совместное использование) линии IRQ с другими PCI
устройствами;
    //name— символьная строка, используемая в
/proc/interrupts для отображения владельца прерывания;
```

```
    //dev– указатель на уникальный идентификатор
устройства на линии IRQ,
        //для не разделяемых прерываний (например, шины
ISA) может указываться NULL.

    int res = request_irq(irq, irq_handler,
IRQF_SHARED, "wq_lab", &dev_id);
    if (res < 0) {
        printk(KERN_ERR "wq_lab: Couldn't register
interrupt handler!\n");
        return res;
    }


    que = create_workqueue("my_wq");
    if (!que) {
        printk(KERN_ERR "wq_lab: Coulnd't create
queue!\n");
        return -1;
    }


    work =vmalloc(sizeof(struct work_struct));

    if (!work) {
        printk(KERN_ERR "wq_lab: Can't allocate memory
for work!\n");
        return -1;
    }
    //wq_func -функция обработчик
    INIT_WORK(work, wq_func);

    printk(KERN_INFO "wq: Module loaded!\n");
    return 0;
}

static void __exit exit_module(void) {
    free_irq(irq, &dev_id);
    flush_workqueue(que);
    destroy_workqueue(que);
    vfree(work);
    printk(KERN_INFO "wq_lab: Module unloaded!\n");
}
module_init(load_module);
module_exit(exit_module);
```

Результат работы программы

```
wizard@wizard-VM:~/Desktop/Lab_9/workqueues$ make
make -C /lib/modules/5.3.0-51-generic/build M=/home/wizard/Desktop/Lab_9/workqueues modules
make[1]: Entering directory '/usr/src/linux-headers-5.3.0-51-generic'
  CC [M]  /home/wizard/Desktop/Lab_9/workqueues/wq.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/wizard/Desktop/Lab_9/workqueues/wq.mod.o
  LD [M]  /home/wizard/Desktop/Lab_9/workqueues/wq.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.3.0-51-generic'
wizard@wizard-VM:~/Desktop/Lab_9/workqueues$
```

Рис. 6 – сборка модуля ядра

```
wizard@wizard-VM:~/Desktop/Lab_9/workqueues$ lsmod | grep wq
wq                     16384  0
wizard@wizard-VM:~/Desktop/Lab_9/workqueues$
```

Рис. 7 – загруженный модуль ядра

```
wizard@wizard-VM:~/Desktop/Lab_9/workqueues$ dmesg | tail -15
[ 3010.613143] wq_lab: Keycode 25
[ 3010.997182] wq_lab: Keycode 49
[ 3011.437145] wq_lab: Keycode 34
[ 3011.757165] wq_lab: Keycode 28
[ 3013.541021] wq_lab: Keycode 56
[ 3013.613196] wq_lab: Keycode 15
[ 3017.453112] wq_lab: Keycode 29
[ 3017.703802] wq_lab: Keycode 29
[ 3017.725146] wq_lab: Keycode 46
[ 3018.797098] wq_lab: Keycode 56
[ 3018.837298] wq_lab: Keycode 15
[ 3019.861187] wq_lab: Keycode 29
[ 3019.949253] wq_lab: Keycode 42
[ 3020.013099] wq_lab: Keycode 47
[ 3020.572993] wq_lab: Keycode 28
wizard@wizard-VM:~/Desktop/Lab_9/workqueues$
```

Рис. 8 – последние 15 сообщений, выведенных модулями ядра

```
wizard@wizard-VM:~/Desktop/Lab_9/workqueues$ sudo rmmod wq
wizard@wizard-VM:~/Desktop/Lab_9/workqueues$
```

Рис. 9 – выгрузка модуля ядра

```
wizard@wizard-VM:~/Desktop/Lab_9/workqueues$ dmesg | tail -5
[ 3084.285061] wq_lab: Keycode 32
[ 3084.428897] wq_lab: Keycode 24
[ 3084.549004] wq_lab: Keycode 57
[ 3084.909069] wq_lab: Keycode 28
[ 3084.935051] wq_lab: Module unloaded!
wizard@wizard-VM:~/Desktop/Lab_9/workqueues$
```

Рис. 10 – последние 5 сообщений, выведенных модулями ядра

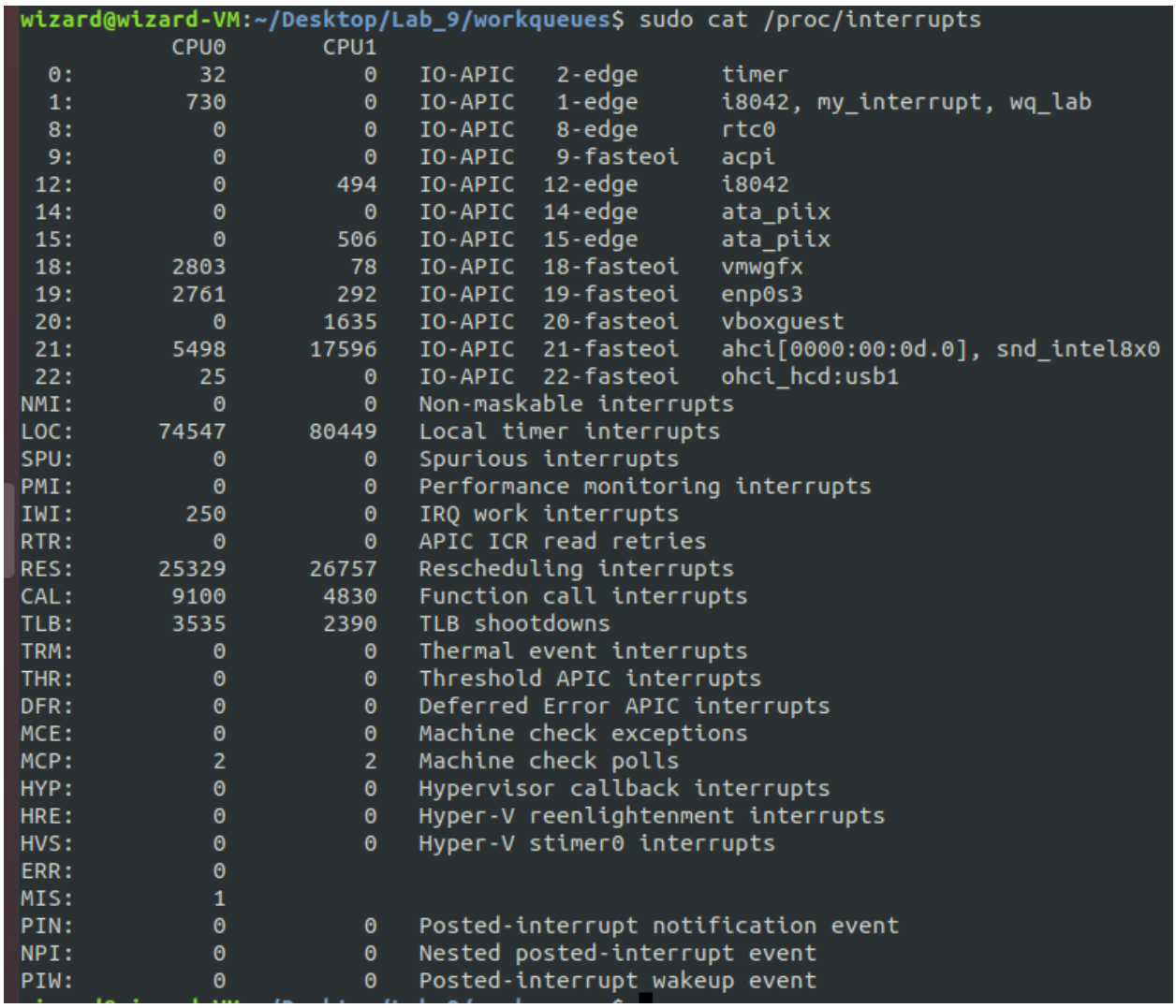

Рис. 11 Разделение IRQ в системе.

- **/proc/interrupts** – содержит данные, которые относятся к системе
- Первый столбец – строка IRQ
- Второй столбец – количество сработавших прерываний
- Третий столбец – связан с PIC
- Шестой (последний) – список имен устройств, которые зарегистрировали обработчики данного прерывания. При данном случае IRQ обрабатывается устройствами «my_interrupt» и «wq_lab»