|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **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 3](#_Toc40745770)

[Флаги: 3](#_Toc40745771)

[Тасклеты 4](#_Toc40745772)

[Листинг 1 – tasklet.c 6](#_Toc40745773)

[Результат работы программы 7](#_Toc40745774)

[ЗАДАНИЕ 2 9](#_Toc40745775)

[Очереди работ 10](#_Toc40745776)

[Флаги 10](#_Toc40745777)

[Листинг 2 – wq.c 12](#_Toc40745778)

[Результат работы программы 14](#_Toc40745779)

# 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**)**

**{**

**if** **(**irq **==** IRQ\_KB**)**

**{**

tasklet\_schedule**(**tasklet**);**

printk**(**KERN\_INFO "[INTERRUPT %d] handled on ISR"**,** cnt**);**

**return** IRQ\_HANDLED**;**

**}**

**return** IRQ\_NONE**;**

**}**

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

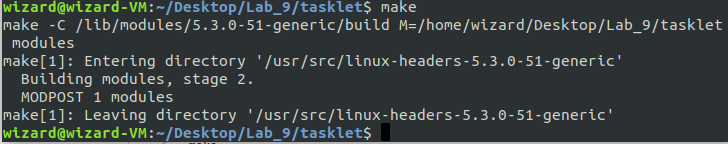


Рис. 1 – сборка модуля ядра

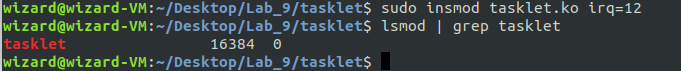


Рис. 2 – загрузка модуля ядра

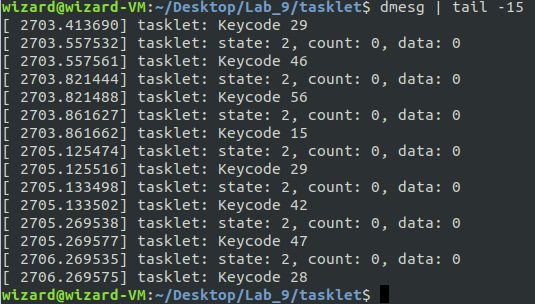


Рис. 3 – последние 15 сообщений, выведенных модулями ядра



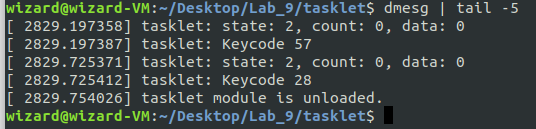
Рис. 4 – выгрузка модуля ядра

Рис. 5 – последние 5 сообщений, выведенных модулями ядра

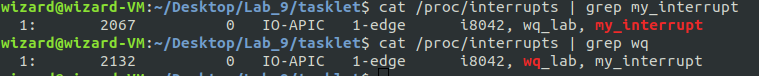


Рис. 6 Разделение IRQ в системе.

* **/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»

# 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 очередяхwork’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 очередяхwork’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"**);**

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

**if** **(**irq **==** IRQ\_KB**)**

**{**

work\_struct **\***work **=** **(**work\_struct **\*)**kmalloc**(sizeof(**work\_struct**),** GFP\_KERNEL**);**

**if** **(**work**)**

**{**

INIT\_WORK**((**struct work\_struct **\*)**work**,** wq\_func**);**

sprintf**(**work**->**x**,** "Work for request %d"**,** cnt**);**

**}**

queue\_work**(**que**,** work**);**

**return** IRQ\_HANDLED**;**

**}**

**return** IRQ\_NONE**;**

**}**

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

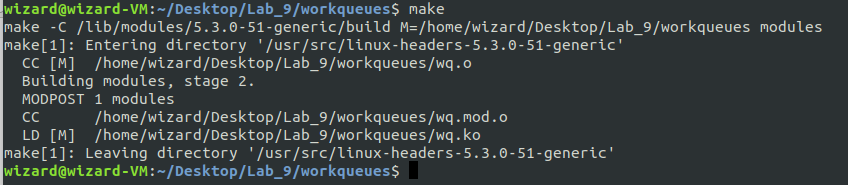


Рис. 7 – сборка модуля ядра



Рис. 8 – загруженный модуль ядра

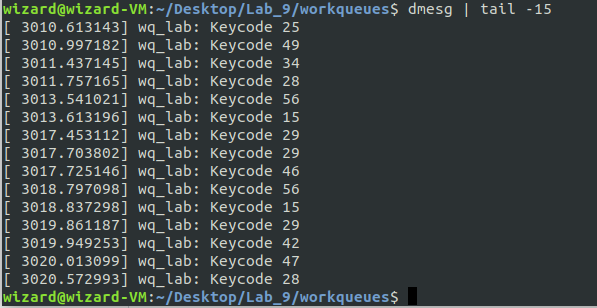


Рис. 9 – последние 15 сообщений, выведенных модулями ядра



Рис. 10 – выгрузка модуля ядра

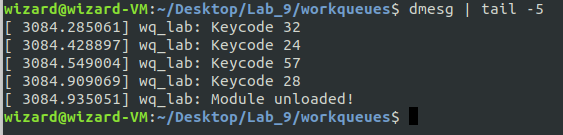


Рис. 11 – последние 5 сообщений, выведенных модулями ядра