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: Мирзоян С.А.,
Группа: ИУ7-65Б

Москва, 2020 г.

Задание

Часть 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>

Часть 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>

Прерывания

Аппаратные прерываний возникают от внешних устройств, являются в системе асинхронными событиями, которые возникают независимо от какой-либо выполняемой в системе работы, и их принято делить на следующие группы:

- Прерывание от системного таймера, которое возникает в системе периодически.
- Прерывания от устройств ввода-вывода. Возникают по инициативе устройства, когда устройству нужно сообщить процессору о завершении операции ввода-вывода.
- Прерывания от действий оператора, например, в ОС Windows при нажатии клавишей ctrl_alt_del для вызова task manager.

Аппаратные прерывания освобождают процессор от необходимости опрашивать внешние устройства с целью определения их готовности передать запрошенные процессом данные. Но требуют от системы выполнения последовательности действий по их обслуживанию. Пока данные не готовы процессор может выполнять какую-то другую работу. Но, когда поступает сигнал прерывания, процессор должен переключиться на его обслуживание.

Первой была микросхема Intel 8259 PIC, которая имела 8 входных линий (IRQ0-7), и одну выходную линию INTR (или просто INT). Сигналы прерывания от устройств ввода-вывода поступают на входы IRQ (Interrupt Request), а контроллер прерывания формирует сигнал прерывания, который по шине управления (линии INTR) поступает на соответствующую ножку (pin) процессора. Сигнал прерывания будет передан процессору, если он не замаскирован, т.е. его обработка разрешена. Для увеличения числа обрабатываемых прерываний контроллеры стали подключать в виде каскада: ведущий и ведомый контроллеры (всего 15 линий IRQ, одна линия используется для каскадного соединения). Этого было достаточно для систем с шиной ISA.

Раскладка прерываний под шину ISA.

IRQ 0 — system timer
IRQ 1 — keyboard controller
IRQ 2 — cascade (прерывание от slave контроллера)
IRQ 3 — serial port COM2
IRQ 4 — serial port COM1
IRQ 5 — parallel port 2 and 3 or sound card
IRQ 6 — floppy controller
IRQ 7 — parallel port 1
IRQ 8 — RTC timer
IRQ 9 — ACPI
IRQ 10 — open/SCSI/NIC
IRQ 11 — open/SCSI/NIC
IRQ 12 — mouse controller
IRQ 13 — math co-processor
IRQ 14 — ATA channel 1
IRQ 15 — ATA channel 2

На смену шине ISA пришла шина PCI. И количество устройств, требующих подключения, стало больше. Кроме того, в отличие от статической шины ISA шина PCI позволяла добавляться устройства в систему динамически. В данной шине прерывания могут быть разделяемыми

Для обеспечения обратной совместимости со старыми системами первый 16 линий прерывания отводятся под старые прерывания ISA.

Медленные и быстрые прерывания

В ОС Linux принято различать быстрые и медленные прерывания. В ядрах до версии 2.6.19 для обозначения быстрых прерываний использовался флаг SA_INTERRUPT. В современных версиях такого флага нет и флаги обозначаются как **IRQF_*** и единственным быстрым прерывание осталось прерывание от таймера **__IRQF_TIMER** 0x00000200.

Быстрые прерывания это такие, которые можно обрабатывать очень быстро. Быстрые прерывания выполняются при запрете всех прерываний на текущем процессоре. На других процессорах прерывания могут обрабатываться, но при запрете прерываний по линии IRQ, относящейся к выполняемому быстрому прерыванию. Таким образом, выполнение быстрого прерывания не может быть прервано.

Медленный обработчик прерываний не запрещает другие прерывания пока они обрабатываются. Это гарантирует малое время входа в прерывание, потому что другой с более высоким приоритетом прервёт текущий и не будет блокирован. Поведение "медленного" обработчика реакции на особую ситуацию - это то, что разработчик хочет для приложений реального времени.

Чтобы сократить время выполнения обработчиков прерываний обработчики медленных аппаратных прерываний делятся на две части, которые традиционно называются верхняя (top) и нижняя (bottom) половины (half). Верхними половинами остаются обработчики, устанавливаемы функцией request_irq() на определенных IRQ. Выполнение нижних половин инициируется верхними половинами, т.е. обработчиками прерываний.

В современных ОС Linux имеется три типа нижних половин (bottom half):

- softirq – отложенные прерывания;
- tascklet – тасклеты;
- workqueue – очереди работ.

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.

Тасклеты описываются структурой:

```
1.struct tasklet_struct
2.{
3.    struct tasklet_struct *next; /* указатель на
  следующий тасклет*/
4.    unsigned long state; /* текущее состояние*/
5.    atomic_t count; /* счетчик ссылок*/
6.    void (*func)(unsigned long); /* обработчик*/
7.    unsigned long data; /* данные*/
8.}
```

Тасклеты могут быть зарегистрированы как статически, так и динамически.

Статически тасклеты создаются с помощью двух макросов, определенных в файле linux/interrupt.h:

```
1.DECLARE_TASKLET(name, func, data);
2.DECLARE_TASKLET_DISABLED(name, func, data);
```

Оба макроса статически создают экземпляр структуры struct_tasklet с именем name и регистрируют обработчик нижней половины – тасклет.

Первый макрос создает тасклет, у которого поле count = 0 и, следовательно, он разрешен.

Второй создает тасклет со счетчиком ссылок count = 1 и, следовательно, он запрещен.

Динамически тасклет инициализируется функцией init_tasklet():

```
1.void tasklet_init(struct tasklet_struct *t,
2.          void (*func)(unsigned long), unsigned
  long data)
3.{
4.    t->next = NULL;
5.    t->state = 0;
6.    atomic_set(&t->count, 0);
7.    t->func = func;
8.    t->data = data;
```

```
9. }
```

Тасклеты планируются на выполнение. Для планирования тасклетов на выполнение используются две функции:

- tasklet_schedule()
- tasklet_hi_schedule().

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

Когда тасклет 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 запланирован на выполнение, он выполняется один раз в некоторый момент времени в ближайшем будущем. Если тасклет, который запланирован на выполнение, будет запланирован еще раз до того, как он выполнится, то он также выполнится всего один раз. Если тасклет уже выполняется, скажем, на другом процессоре, то будет запланирован снова и снова выполнится. Для оптимизации тасклет всегда выполняется на том процессоре, который его запланировал на выполнение, что дает надежду на лучшее использование кэша процессора.

Листинг

```
1.#include <linux/module.h>
2.#include <linux/kernel.h>
3.#include <linux/init.h>
4.#include <linux/interrupt.h>
5.#include <linux/sched.h>
6.
7.struct tasklet_struct *tasklet;
8.int dev_id,scancode, irq = 1;
9.
10.     MODULE_LICENSE("GPL");
11.     MODULE_AUTHOR("Sergey Mirzoyan");
12.
13.     #define KBD_DATA_REG 0x60
14.     #define kbd_read_input() inb(KBD_DATA_REG)
```

```
15.
16.     void tasklet_function(unsigned long data)
17.     {
18.         scancode = kbd_read_input();
19.         if (scancode < 103) {
20.             printk(KERN_INFO "Тасклет:\n\tтекущее состояние: %ld, \tсчетчик ссылок: %d, \tданные: %ld\n", tasklet->state, tasklet->count, tasklet->data);
21.             printk(KERN_INFO "Тасклет: \n\tKeycode %d\n", scancode);
22.         }
23.         return;
24.     }
25.
26.     static int __init module_tasklet_init(void)
27.     {
28.         if (request_irq(irq, my_interrupt, IRQF_SHARED, "my_tasklet", &dev_id))
29.             return -1;
30.         tasklet = vmalloc(sizeof(struct tasklet_struct));//vmalloc - выделяет страницы памяти, которые только виртуально смежные и необязательно смежные физически.
31.                                                          //гарантирует только, что страницы будут смежными в виртуальном адресном пространстве ядра.
32.
33.         tasklet_init(tasklet, tasklet_function, 0);
34.         printk(KERN_INFO "Модуль тасклета загружен.\n");
35.         return 0;
36.     }
37.
38.     static irqreturn_t my_interrupt(int irq, void *dev_id)
39.     {
40.         if (! irq == IRQ)// проверка того, что обслуживаемое устройство запросило прерывание
41.             return IRQ_NONE;
42.         tasklet_schedule(tasklet);
43.         return IRQ_HANDLED;
44.     }
45.
46.     static void __exit module_tasklet_exit(void)
```

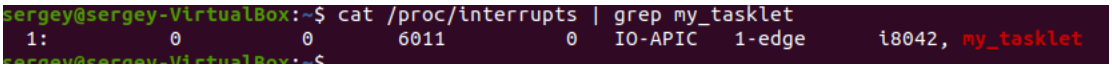
```
47.     {
48.         tasklet_kill(tasklet);
49.         vfree(tasklet);
50.         free_irq(irq, &dev_id);
51.         printk(KERN_INFO "Модуль тасклета выгружен.\n");
52.         return;
53.     }
54.
55.     module_init(module_tasklet_init);
56.     module_exit(module_tasklet_exit);
```

Результат работы программы

Сообщения модулей ядра (dmesg | tail -250)

```
[17352.970810] Модуль тасклета выгружен.
[17360.940707] Модуль тасклета загружен.
[17361.570621] Тасклет:
                текущее состояние: 2,
                счетчик ссылок: 0,
                данные: 0
[17361.570658] Тасклет:
                Keycode 72
[17361.984413] Тасклет:
                текущее состояние: 2,
                счетчик ссылок: 0,
                данные: 0
[17361.984450] Тасклет:
                Keycode 72
[17363.039215] Тасклет:
                текущее состояние: 2,
                счетчик ссылок: 0,
                данные: 0
[17363.039253] Тасклет:
                Keycode 72
[17365.056326] Тасклет:
                текущее состояние: 2,
                счетчик ссылок: 0,
                данные: 0
[17365.056362] Тасклет:
                Keycode 72
[17366.250094] Тасклет:
                текущее состояние: 2,
                счетчик ссылок: 0,
                данные: 0
[17366.250131] Тасклет:
                Keycode 72
[17367.617363] Тасклет:
                текущее состояние: 2,
                счетчик ссылок: 0,
                данные: 0
[17367.617399] Тасклет:
                Keycode 28
```

Разделение IRQ в системе.



Файл /proc/interrupts предоставляет таблицу о количестве прерываний на каждом из процессоров в следующем виде:

- Первая колонка: номер прерывания
- Колонки CPUx: счётчики прерываний на каждом из процессоров
- Следующая колонка: вид прерывания:
  - IO-APIC-edge — прерывание по фронту на контроллер I/O APIC
  - IO-APIC-fasteoi — прерывание по уровню на контроллер I/O APIC
  - PCI-MSI-edge — MSI прерывание
  - XT-PIC-XT-PIC — прерывание на PIC контроллер

Последняя колонка: устройство, ассоциированное с данным прерыванием

Очереди работ

Очередь заданий является еще одной концепцией для обработки отложенных функций. Это похоже на тасклет с некоторыми отличиями. Функции рабочих очередей выполняются в контексте процесса ядра, но функции тасклетов выполняются в контексте программных прерываний. Это означает, что функции очереди задач не должны быть атомарными, как функции тасклета. Тасклеты всегда выполняются на процессоре, с которого они были отправлены. Рабочие очереди работают таким же образом, но только по умолчанию.

Подсистема рабочей очереди представляет собой интерфейс для создания потоков ядра для обработки работы (work), которая ставится в очередь. Такие потоки ядра называются рабочими потоками. Рабочая очередь поддерживается типом struct work_struct, который определён в include/linux/workqueue.h:

```
1.struct work_struct {
2.    atomic_long_t data;
3.    struct list_head entry;
4.    work_func_t func;
5.      #ifdef CONFIG_LOCKDEP
6.    struct lockdep_map lockdep_map;
```

```
7.      #endif
8.};
```

Обратим внимание на два поля: func - функция, которая будет запланирована в рабочей очереди, и data - параметр этой функции.

Очередь работ создается функцией:

```
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

```
1.#include <linux/module.h>
2.#include <linux/kernel.h>
3.#include <linux/init.h>
4.#include <linux/interrupt.h>
5.#include <linux/workqueue.h>
6.
7.
8.MODULE_LICENSE("GPL");
9.MODULE_AUTHOR("Sergey Mirzoyan");
10.
11.     int irq = 1;
12.     int dev_id;//,scancode;
13.
14.     struct workqueue_struct *que;
15.     struct work_struct *work;
16.
17.     #define KBD_DATA_REG 0x60
18.     #define kbd read input() inb(KBD DATA REG)
19.
20.     void wq_func ( struct work_struct * work )
21.     {
22.         printk ( KERN_INFO "Рабочая очередь: \nДанные --> %u)\n",
   atomic_read (( atomic_t *) & work->data ) ) ;
23.     }
24.
25.     static irqreturn_t irq_handler(int irq, void *dev_id)
26.     {
27.         queue_work(que, work);
28.         return IRQ_HANDLED;
29.     }
30.
```

```
31.
32.     static int __init load_module(void)
33.     {
34.         int res = request_irq(irq, irq_handler, IRQF_SHARED,
    "my_woking_queue", &dev_id);
35.         if (res < 0)
36.         {
37.             printk(KERN_ERR "Рабочая очередь: Не удалось
    зарегистрировать обработчик прерываний!\n");
38.             return res;
39.         }
40.
41.
42.         que = create_workqueue("my_woking_queue"); // возвращает
    ссылку на workqueue_struct
43.         if (!que)
44.         {
45.             printk(KERN_ERR "Рабочая очередь: Невозожно создать
    очередь!\n");
46.             return -1;
47.         }
48.
49.
50.         work =vmalloc(sizeof(struct work_struct));
51.
52.         if (!work)
53.         {
54.             printk(KERN ERR "wq lab: Can't allocate memory for
    work!\n");
55.             return -1;
56.         }
57.
58.         INIT WORK(work, wq func);//wq func -функция обработчик
59.
60.         printk(KERN_INFO "Рабочая очередь: Модуль загружен!\n");
61.         return 0;
62.     }
63.
64.     static void __exit exit_module(void)
65.     {
```

```
66.         free_irq(irq, &dev_id);
67.         flush_workqueue(que);
68.         destroy_workqueue(que);
69.         vfree(work);
70.         printk(KERN_INFO "Рабочая очередь: Модуль выгружен!\n");
71.     }
72.
73.     module_init(load_module);
74.     module_exit(exit_module);
```

Результат работы программы

Сообщения модулей ядра (dmesg | tail -40)

```
[30280.795611] Рабочая очередь: Модуль загружен!
[30280.886288] Рабочая очередь:
               Данные --> 128)
[30281.353431] Рабочая очередь:
               Данные --> 128)
[30281.385970] Рабочая очередь:
               Данные --> 128)
[30281.405197] Рабочая очередь:
               Данные --> 128)
[30281.406396] Рабочая очередь:
               Данные --> 128)
[30281.487195] Рабочая очередь:
               Данные --> 128)
[30281.490159] Рабочая очередь:
               Данные --> 128)
[30281.606422] Рабочая очередь:
               Данные --> 128)
[30281.608353] Рабочая очередь:
               Данные --> 128)
[30281.715302] Рабочая очередь:
               Данные --> 128)
[30281.717427] Рабочая очередь:
               Данные --> 128)
[30282.076313] Рабочая очередь:
               Данные --> 128)
[30282.086117] Рабочая очередь: Модуль выгружен!
```