**Обслуживание прерываний**

1. **Аппаратные прерывания**

Схема обработки аппаратных прерываний — это принципиально архитектурно зависимое действие, связанное с непосредственным взаимодействием с контроллером прерываний. Но схема в основных чертах остаётся неизменной, независимо от архитектуры. Это связано со спецификой решаемой прерываниями задачи информирования процессора о асинхронном событии, возникающем в системе.

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.

В персональных компьютерах на базе процессоров Intel обработка аппаратных прерываний поддерживается аппаратно. Первой была микросхема [Intel 8259 PIC](https://en.wikipedia.org/wiki/Intel_8259), 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.

Обычно раскладка прерываний под устройства была более менее стандартная, например:  
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 (то есть к одной линии IRQ можно подсоединить несколько устройств). В итоге чтобы решить проблему нехватки линий IRQ, прерывания от PCI устройств стали группировать в линии PIRQ (Programmable Interrupt Request).

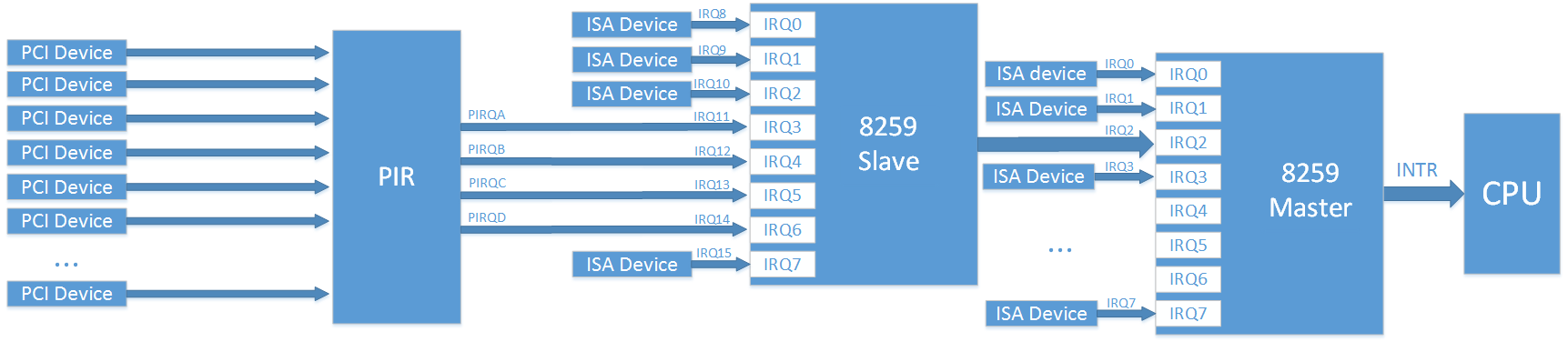


Рис.1

С появлением многопроцессорных систем PIC был заменен APIC (Advanced PIC), чтобы сбалансировать нагрузку по обработке прерываний на процессоры. Для каждого процессора добавляется специальный контроллер LAPIC (Local APIC) и для маршрутизации прерываний от устройств добавляется контроллер [I/O APIC](https://wiki.osdev.org/IOAPIC). Все эти контроллеры объединяются в общую шину с названием APIC (новые системы сейчас уже соединяются по стандартной системной шине).

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

Шина PCI была заменена PCI express. Линии прерываний решили убрать. Чтобы сохранить совместимость, сигналы о возникновении прерываний (INTx#) эмулируются отдельными видами сообщений. В этой схеме логическое сложение линий прерываний, которое раньше производилось физическим соединением проводов, легло на плечи PCI мостов. Однако поддержка legacy INTx прерываний — это лишь поддержка обратной совместимости с шиной PCI. В PCI express был предложен новый метод доставки сообщений о прерываниях — MSI (Message Signaled Interrupts). В этом методе для сигнализации о прерывании устройство просто производит запись в MMIO область отведённую под LAPIC процессора.

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

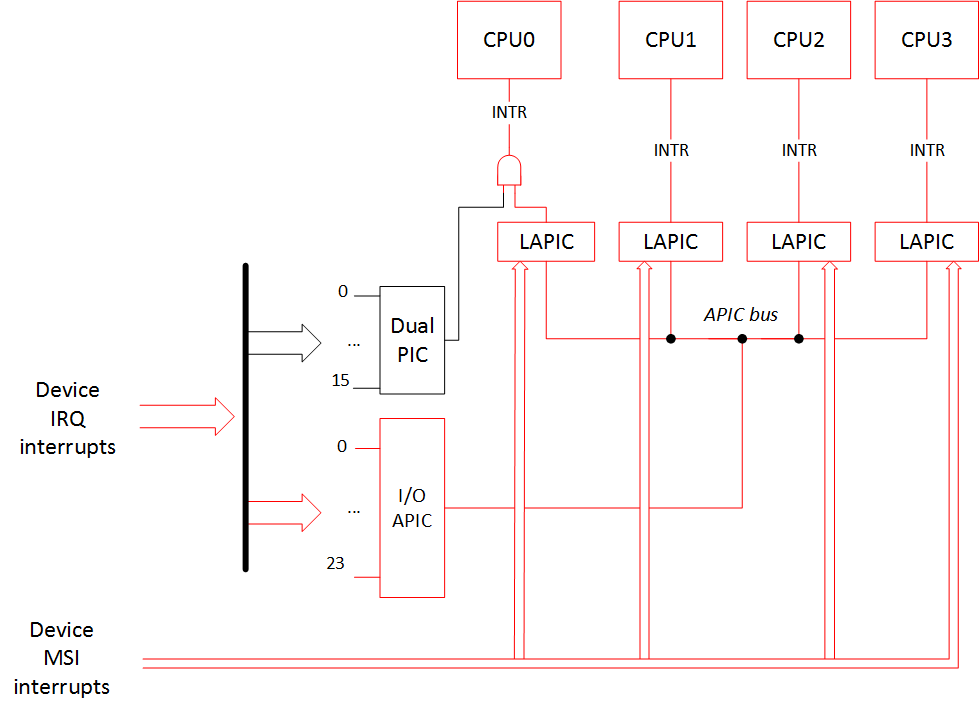
* Первая колонка: номер прерывания
* Колонки CPUx: счётчики прерываний на каждом из процессоров
* Следующая колонка: вид прерывания:
  + IO-APIC-edge — прерывание по фронту на контроллер I/O APIC
  + IO-APIC-fasteoi — прерывание по уровню на контроллер I/O APIC
  + PCI-MSI-edge — MSI прерывание
  + XT-PIC-XT-PIC — прерывание на PIC контроллер
* Последняя колонка: устройство, ассоциированное с данным прерыванием  
  Так вот, как и положено в современной системе, для устройств и драйверов, поддерживающих прерывания MSI/MSI-X, используются именно они. Остальные прерывания «роутятся» ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *routing -* маршрутизация)  через I/O APIC.  
  Упрощённая схема роутинга прерываний показана на рис.2.  
    
  

Рис.2

В многопроцессорных и многоядерных системах устройстваполучают возможностьсамостоятельно выбирать процессор/ядро для обработки конкретного прерывания, причем это делается полностью на уровне аппаратуры без программной поддержки. Это позволяет оптимизировать работу системы путем размещения большей части структур драйвера устройства и связанного с ним ПО, например, сетевых протоколов, в кэше конкретного процессора или же в его «ближней» NUMA-памяти[[1]](#footnote-1)

**2. Адресация обработчиков прерываний в защищенном режиме**

Схема, представленная на рис.3, может рассматриваться как абстракция, позволяющая лаконично продемонстрировать адресацию обработчиков прерываний в защищенном режиме. Операции, выполняемые на единственном процессоре, рассматриваются только как тривиальный, вырожденный случай SMP (SMP - **S**ymmetric **M**ulti**P**rocessing — симметричное мультипроцессирование).  


Рис.3

Регистр GDTR содержит начальный адрес таблицы GDT – Global Descriptor Table – таблицы глобальных дескрипторов, которая в защищенном режиме содержит дескрипторы сегментов физической памяти.

Регистр IDTR содержит начальный адрес таблицы дескрипторов прерываний, которая содержит дескрипторы шлюзов прерываний, ловушек и задач.

Дескриптор глобальной таблицы дескрипторов имеет размер 8 байт и имеет следующие поля (рис.4):

1. Размер – limit – 2 байта
2. Начальный адрес сегмента – младшая часть – base\_low – 2 байта
3. Начальный адрес сегмента – средняя часть – base\_midle – 1 байт
4. Байт атрибутов – младший байт атрибутов – attr1
5. Байт атрибутов – старший байт атрибутов – attr2
6. Начальный адрес сегмента –старшая часть – base\_hight – 1 байт

Таким образом, начальный адрес сегмента составляет 32 разряда в защищенном режиме.

Биты атрибутов 1 имеют следующее назначение:

Бит 0 - A – бит доступа к сегменту; устанавливается аппаратно при обращении к сегменту

Следующие три бита определяют тип сегмента:

Бит 1 – для сегмента кода, если бит сброшен, то чтение из сегмента запрещено, но это не относиться к выборке команд, если установлен, то чтение разрешено;

для сегмента данных, если бит сброшен, то модификация данных запрещена, если установлен, то модификация данных разрешена.

Бит 2 – для сегмента кода это – бит подчинения: если 0, то сегмент подчиненный, если 1, то сегмент кода обычный;

для сегментов данных и стека: если 0, то это сегмент данных, если 1, то это -сегмент стека;

Бит 3 – бит предназначения: если он равен 0, то это – сегмент данных или стека, если равен 1, то это – сегмент данных.

--------------------------------------------------------------------------------------------------------------------

Бит 4 - S - указывает, является ли дескриптор системным (0 - дескриптор системный).

Биты 5-6 - DPL – descriptor privilege level - привилегии дескриптора – определяет уровень привилегий (0 - ядро системы, 3 - пользовательский). В архитектуре x86 поддерживается 4 кольца защиты: от 0 до 3.

Бит 7 - P - present - признак присутствия сегмента в памяти (сегмент может быть выгружен менеджером виртуальной памяти).

Биты байта атрибутов 2 имеют следующие значения:

Биты 0-3 - limit – размер и с учетом двух первых байтов дескриптора – 0-го и 1-го – размер сегмента ограничен 20 битами (220=1024 Кб или 1Мб)

Бит 4 - AVL –

Бит 5 - 0 – зарезервирован, а в 64-разрядных системах это - флаг L, который теперь служит признаком 64-разрядности сегмента. Если он установлен, флаг D/B должен быть сброшен.

Бит 6 - D/B - флаг, указывающий разрядность сегмента: 0 – сегмент 16-разрядный, 1 – сегмент 32 разрядный (этот флаг ещё называют BIG).

Бит 7 - G - бит гранулярности (Granularity) - указывает, в чём измеряется лимит (0 - в байтах, 1 - в страницах по 4 килобайта).



Рис.4 Дескриптор таблицы GDT

Дескриптор таблицы дескрипторов прерываний имеет следующий формат: размер дескриптора 8 байт, которые распределяются следующим образом (рис.5):

* Первые два байта и последние два байта – смещение – 32 бита.
* Байты 2 и 3 – селектор – 16 бит – используется для доступа к дескриптору сегмента, содержащего обработчик прерывания, в таблице GDT (смещение). Получив из дескриптора сегмента его начальный адрес и добавив смещение из дескриптора прерывания, получается адрес точки входа в обработчик прерывания.



Рис.5 Дескриптор таблицы IDT

* Байт 4 – резерв
* Байт 5 – байт атрибутов:

Биты 0, 1, 2, 3 определяют тип дескриптора

|  |  |  |
| --- | --- | --- |
| № | Значение типа | Назначение |
| 1 | 0 | Не определено |
| 2 | 1 | Свободный сегмент задачи (TSS) 80286 |
| 3 | 2 | LDT |
| 4 | 3 | Занятый сегмент состояния задачи (TSS) 80286 |
| 5 | 4 | Шлюзы вызова 80286 |
| 6 | 5 | Шлюз задачи (Task Gate) |
| 7 | 6 | Шлюз прерываний 80286 (Interrupt Gate) |
| 8 | 7 | Шлюз ловушки 80286 (Trap Gate) |
| 9 | 8 | Не определено |
| 10 | 9 | Свободный сегмент состояния задачи (TSS) 80386, 80486 и старше |
| 11 | Ah | Не определено |
| 12 | Bh | Занятый сегмент состояния задачи (TSS) 80386, 80486 и старше |
| 13 | Ch | Шлюз вызова 80386, 80486 и старше |
| 14 | Dh | Не определено |
| 15 | Eh-1110 | Шлюз прерываний (Interrupt Gate) 80386, 80486 и старше |
| 16 | Fh-1111 | Шлюз ловушки (Trap Gate) 80386, 80486 и старше |

4 бит – 0 – не используется

Биты 5-6 – DPL - Уровень привилегий сегмента (**DPL** англ. - **Descriptor** Privilege Level) соответствует значению поля **DPL** в дескрипторе сегмента. Текущий уровень привилегий (CPL англ. ... **DPL** вызываемого сегмента не должен быть численно больше CPL;

**DPL** шлюза не должен быть численно меньше **DPL,** указанного в нём сегмента;

В шлюзе должен быть указан только сегмент кода или TSS (если это шлюз задачи).

Бит 7 – P – present – бит присутствия, если 0 – неиспользуемый вектор.

Для доступа к дескриптору прерывания в качестве смещения в таблице IDT используется вектор прерывания (рис. 3).

**3. Обработчики прерываний**

Обратимся к man(9).

**#include <[asm/irq.h](file:///\\\\usr\\include\\asm\\irq.h)>**  
**#include <[linux/signal.h](file:///\\\\usr\\include\\linux\\signal.h)>**  
**#include <[linux/sched.h](file:///\\\\usr\\include\\linux\\sched.h)>**  
**#include <[linux/interrupt.h](file:///\\\\usr\\include\\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**);

Функция request\_irq() предназначена для регистрации обработчика прерывания на определенной линии прерывания.

Именно обработчики прерываний в драйверах устройств ответственны за взаимодействие в внешними устройствами на этапе передачи данных от устройств, полученных, например, в результате операции ввода-вывода. Драйвер устройства регистрирует в системе один обработчик прерывания. Делается это с помощью функции request\_irq(), которой в качестве параметра передается указатель на обработчик, обслуживающий конкретное прерывание: **irqreturn\_t (\****handler***)(int, void \*).** Эта функция вызывается, когда операционная система получает прерывание. Прототип обработчика принимает три параметра и возвращает значение типа **irqreturn\_t**. Тип **irqreturn\_t**определяется следующим образом:

[**include/linux/irqreturn.h**, line 17 (as a typedef)](https://elixir.bootlin.com/linux/latest/source/include/linux/irqreturn.h#L17)

*/\* SPDX-License-Identifier: GPL-2.0 \*/*

#ifndef [**\_LINUX\_IRQRETURN\_H**](https://elixir.bootlin.com/linux/latest/ident/_LINUX_IRQRETURN_H)

#define [**\_LINUX\_IRQRETURN\_H**](https://elixir.bootlin.com/linux/latest/ident/_LINUX_IRQRETURN_H)

*/\*\**

*\* 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**](https://elixir.bootlin.com/linux/latest/ident/irqreturn) {

[**IRQ\_NONE**](https://elixir.bootlin.com/linux/latest/ident/IRQ_NONE) = (0 << 0),

[**IRQ\_HANDLED**](https://elixir.bootlin.com/linux/latest/ident/IRQ_HANDLED) = (1 << 0),

[**IRQ\_WAKE\_THREAD**](https://elixir.bootlin.com/linux/latest/ident/IRQ_WAKE_THREAD) = (1 << 1),

};

typedef enum [**irqreturn**](https://elixir.bootlin.com/linux/latest/ident/irqreturn) [**irqreturn\_t**](https://elixir.bootlin.com/linux/latest/ident/irqreturn_t);

#define [**IRQ\_RETVAL**](https://elixir.bootlin.com/linux/latest/ident/IRQ_RETVAL)(x) ((x) ? [**IRQ\_HANDLED**](https://elixir.bootlin.com/linux/latest/ident/IRQ_HANDLED) : [**IRQ\_NONE**](https://elixir.bootlin.com/linux/latest/ident/IRQ_NONE))

#endif

Например,

static irqreturn\_t intr\_handler (int irq, void \*dev) {

     if(! /\* проверка того, что обслуживаемое устройство запросило прерывание\*/)

        return IRQ\_NONE;

     /\* код обслуживания устройства \*/

     return IRQ\_HANDLED;

}

Первым параметром функции **request\_irq()** является **unsigned int *irq****, который определяет 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. Функция вызывается всякий раз, когда возникает прерывание с соответствующим значением irq.

Третий параметр – irqflags – может быть или нулем, или битовой маской одного или нескольких следующих флагов, которые используются только ядром как часть IRQ-обработчиков:

#define [**IRQF\_SHARED**](https://elixir.bootlin.com/linux/v4.4/ident/IRQF_SHARED) 0x00000080 /\*разрешить разделение IRQ несколькими устройствами\*/

#define [**IRQF\_PROBE\_SHARED**](https://elixir.bootlin.com/linux/v4.4/ident/IRQF_PROBE_SHARED) 0x00000100 /\*устанавливается вызывающими, когда они ожидают, что произойдет несовпадение при обмене\*/

#define [**\_\_IRQF\_TIMER**](https://elixir.bootlin.com/linux/v4.4/ident/__IRQF_TIMER) 0x00000200 /\* прерывание помечается как прерывание по таймеру.\*/

#define [**IRQF\_PERCPU**](https://elixir.bootlin.com/linux/v4.4/ident/IRQF_PERCPU) 0x00000400 /\*прерывание устанавливается на процессор\*/

#define [**IRQF\_NOBALANCING**](https://elixir.bootlin.com/linux/v4.4/ident/IRQF_NOBALANCING) 0x00000800/\*флаг, чтобы исключить это прерывание из балансировки irq\*/

#define [**IRQF\_IRQPOLL**](https://elixir.bootlin.com/linux/v4.4/ident/IRQF_IRQPOLL) 0x00001000 /\*прерывание используется для опроса (только соображение производительности, которое зарегистрировано первым в общем прерывании, рассматривается)\*/

#define [**IRQF\_ONESHOT**](https://elixir.bootlin.com/linux/v4.4/ident/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**](https://elixir.bootlin.com/linux/v4.4/ident/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\*/

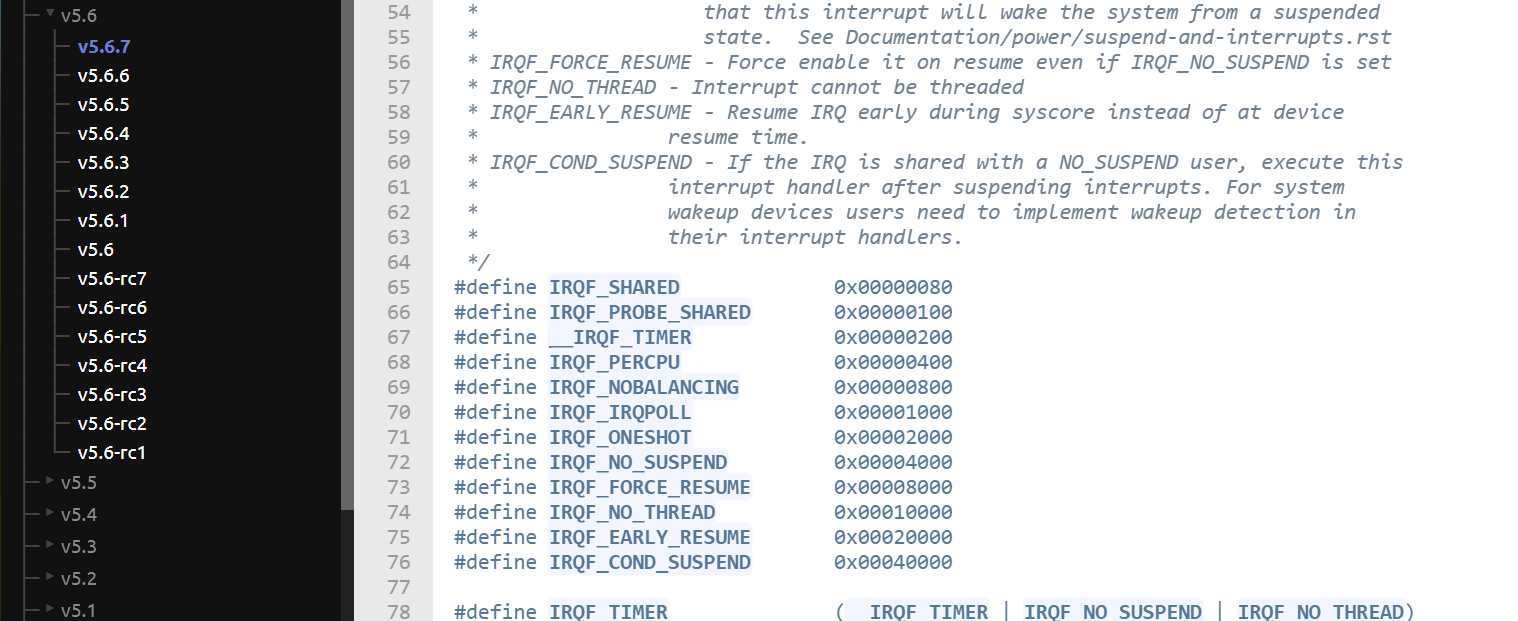
#define [**IRQF\_FORCE\_RESUME**](https://elixir.bootlin.com/linux/v4.4/ident/IRQF_FORCE_RESUME) 0x00008000 /\* принудительно включить его при возобновлении, даже если установлен IRQF\_NO\_SUSPEND\*/

#define [**IRQF\_NO\_THREAD**](https://elixir.bootlin.com/linux/v4.4/ident/IRQF_NO_THREAD) 0x00010000 /\*Прерывание не может быть связано\*/

#define [**IRQF\_EARLY\_RESUME**](https://elixir.bootlin.com/linux/v4.4/ident/IRQF_EARLY_RESUME) 0x00020000 /\* Возобновить IRQ на ранней стадии во время syscore, а не во время возобновления работы устройства\*/

#define [**IRQF\_COND\_SUSPEND**](https://elixir.bootlin.com/linux/v4.4/ident/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\*/

#define [**IRQF\_TIMER**](https://elixir.bootlin.com/linux/v4.4/ident/IRQF_TIMER) ([**\_\_IRQF\_TIMER**](https://elixir.bootlin.com/linux/v4.4/ident/__IRQF_TIMER) | [**IRQF\_NO\_SUSPEND**](https://elixir.bootlin.com/linux/v4.4/ident/IRQF_NO_SUSPEND) | [**IRQF\_NO\_THREAD**](https://elixir.bootlin.com/linux/v4.4/ident/IRQF_NO_THREAD))



*Следует отметить, что прототипы irq\_handler\_t и флаги установки обработчика существенно меняются от версии к версии ядра, например, радикально поменялись после 2.6.19 все флаги. Старое именование SA\_\* изменено на IRQF\_\*.*

*SA\_INTERRUPT был заменен на \_IRQF\_TIMER.*

Четвертый параметр – devname – ASCII текстовое представление устройства, связанного с прерыванием. Например, для клавиатуры это - "keyboard". Это имя используется в /proc/irq и /proc/interrupt.

Пятый параметр – dev\_id – используется прежде всего для разделения линий прерывания. Когда обработчик прерывания освобождается dev\_id предоставляет уникальный файл cookie, позволяющий удалить с линии irq только соответствующий обработчик прерывания. Ядро без этого параметра не может узнать какой обработчик удалять с данной линии прерывания. Можно установить значение NULL, если линия прерывания не разделяется.

Следует отметить, что функция request\_irq() может блокироваться и поэтому не может быть вызвана из контекста прерывания или в ситуации, когда блокировка небезопасна.

Когда драйвер выгружается, необходимо отменить регистрацию соответствующего обработчика прерывания и потенциально отключить линию прерывания. Это делается функцией:

**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 прерывания. Если линия прерывания разделяется, то обработчик, идентифицированный через dev\_id удаляется, но сама линия прерывания будет отключена только при удалении последнего обработчика.

**4. Быстрые и медленные 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 было показано ранее, **IRQF\_\*** и единственным быстрым прерывание осталось прерывание от таймера [**\_\_IRQF\_TIMER**](https://elixir.bootlin.com/linux/v4.4/ident/__IRQF_TIMER) 0x00000200.

Если говорить об аппаратных прерываниях, то они выполняются на высоких уровнях привилегий и их выполнение нельзя прервать. Быстрые прерывания это такие, которые можно обрабатывать очень быстро, тогда как обработка медленных прерываний занимает значительно больше времени. 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.

Медленные прерывания могут потребовать значительных затрат процессорного времени. Очевидно, что запрет прерываний на длительное время может привести к целому ряду негативных явлений, связанных с отзывчивостью системы. Например, задачи, требующие быстрого внимания, могут не выполняться слишком долго.

После инициирования прерывания его обработчик должен выполняться так быстро как только возможно, чтобы не прерывать текущую активность на длительное время. Обработчики аппаратных прерываний выполняются на очень высоком уровне приоритета и блокируют возникновение других запросов прерываний. 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. Именно поэтому в ОС существует система «softIRQ». Например, обработка поступившего в сетевой адаптер пакета требует нескольких тысяч тактов прежде, чем пакет сможет быть передан в соответствующее адресное пространство режима пользователя. Очевидно, что такая обработка не может быть выполнена обработчиком прерывания. 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.

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

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

**5. Нижние половины (bottom half)**

***Отложенные прерывания (softirq)***

Буквальный перевод soft irq – мягкие или гибкие irq, более соответствует их назначению, чем принятое название – «отложенные прерывания», так как все нижние половины выполняются как отложенные действия.

Выполнение отложенных действий возложено на потоки ядра, которые реализованы как демоны softirq (ksoftirqd – сокращение от kernel softirq deamon). Каждый процессор имеет собсбвенный поток, который называется ksoftirqd/n, где n – номер процессора.

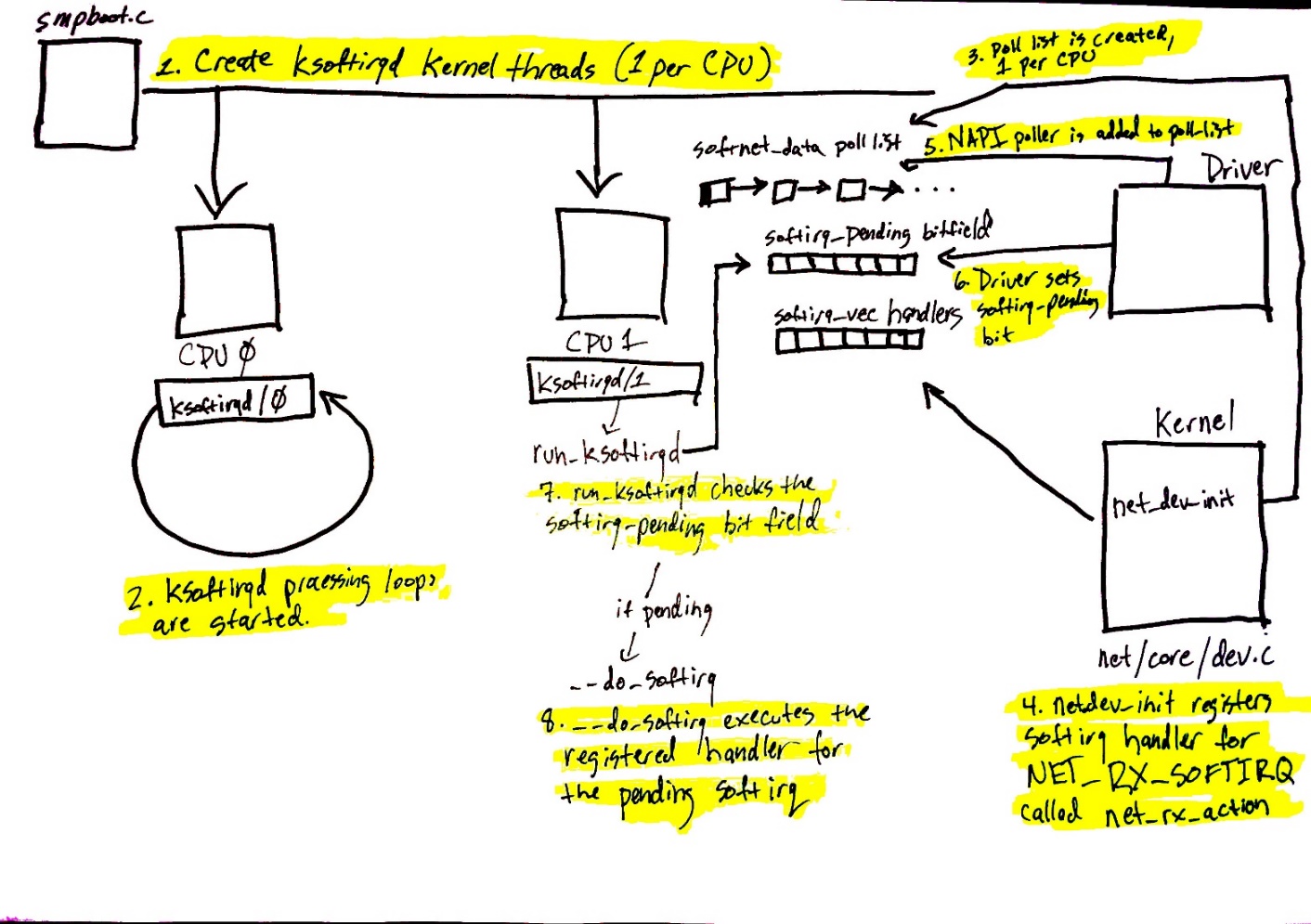


Рис. 6

На рис.6 схематично показана работа демона ksoftirqd на примере при обслуживании сетевых устройств. Softirq отвечает за обработку приходящих пакетов. Потоки ksoftirqd начинают выполнять свои циклы обработки в функции run\_ksoftirqd (2).

Гибкие прерывания определяются статически во время компиляции ядра. Определенная в файле kernel/softirq.c функция open\_softirq выполняет инициализацию softirq:

void open\_softirq(int nr, void (\*action)(struct softirq\_action \*))

{

softirq\_vec[nr].action = action;

}

Данная функция получает два параметра: 1) индекс массива softirq\_vec и 2) указатель на функцию softirq, которая будет выполняться.

Рассмотрим массив softirq\_vec:

static struct softirq\_action softirq\_vec[NR\_SOFTIRQS] \_\_cacheline\_aligned\_in\_smp;

Размер вектора определяется значением NR\_SOFTIRQS и состоит из значений типа struct softirq\_action, определенного в файле linux/interrupt.h. В современных версиях ядра определено 10 видов softirq:

enum

{

[**HI\_SOFTIRQ**](https://elixir.bootlin.com/linux/latest/ident/HI_SOFTIRQ)=0, /\* высокоприоритетные softirq\*/

[**TIMER\_SOFTIRQ**](https://elixir.bootlin.com/linux/latest/ident/TIMER_SOFTIRQ), /\* таймер\*/

[**NET\_TX\_SOFTIRQ**](https://elixir.bootlin.com/linux/latest/ident/NET_TX_SOFTIRQ), /\* отправка сетевых пакетов\*/

[**NET\_RX\_SOFTIRQ**](https://elixir.bootlin.com/linux/latest/ident/NET_RX_SOFTIRQ), /\* прием сетевых пакетов\*/

[**BLOCK\_SOFTIRQ**](https://elixir.bootlin.com/linux/latest/ident/BLOCK_SOFTIRQ), /\* блочное устройство\*/

[**IRQ\_POLL\_SOFTIRQ**](https://elixir.bootlin.com/linux/latest/ident/IRQ_POLL_SOFTIRQ),

[**TASKLET\_SOFTIRQ**](https://elixir.bootlin.com/linux/latest/ident/TASKLET_SOFTIRQ), /\* тасклет\*/

[**SCHED\_SOFTIRQ**](https://elixir.bootlin.com/linux/latest/ident/SCHED_SOFTIRQ), /\* планировщик\*/

[**HRTIMER\_SOFTIRQ**](https://elixir.bootlin.com/linux/latest/ident/HRTIMER_SOFTIRQ),

[**RCU\_SOFTIRQ**](https://elixir.bootlin.com/linux/latest/ident/RCU_SOFTIRQ), */\* Preferable RCU should always be the last softirq – не используется, но сохраняется как последний тип для сохранения нумерации\*/*

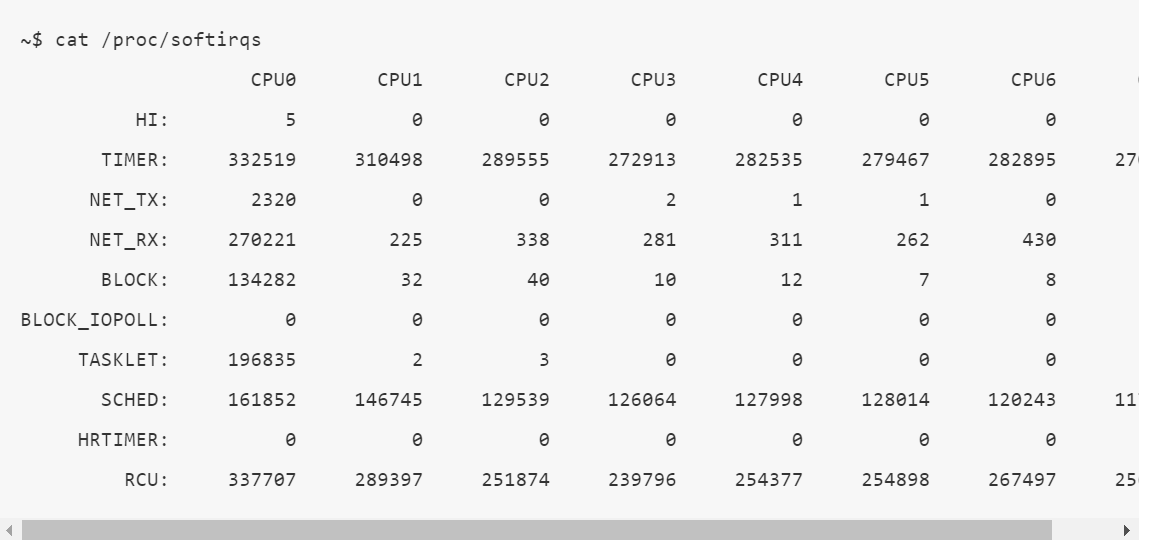
[**NR\_SOFTIRQS**](https://elixir.bootlin.com/linux/latest/ident/NR_SOFTIRQS) /\* количество softirq \*/

};

Все softirq в перечислении представлены в следующем массиве:

|  |
| --- |
| static struct softirq\_action softirq\_vec[NR\_SOFTIRQS] \_\_cacheline\_aligned\_in\_smp; |
|  |  |
|  | DEFINE\_PER\_CPU(struct task\_struct \*, ksoftirqd); |
|  |  |
|  | const char \* const softirq\_to\_name[NR\_SOFTIRQS] = { |
|  | "HI", "TIMER", "NET\_TX", "NET\_RX", "BLOCK", "IRQ\_POLL", |
|  | "TASKLET", "SCHED", "HRTIMER", "RCU" |
|  | }; |

Их можно увидеть это в выводе / proc / softirqs:



Тип struct softirq\_action является основной структурой, связанной с механизмом гибких прерываний (softirqs):

enum [**action**](https://elixir.bootlin.com/linux/latest/ident/action) {[**Fail**](https://elixir.bootlin.com/linux/latest/ident/Fail) = 0, [**Ok**](https://elixir.bootlin.com/linux/latest/ident/Ok) = 1, [**Hold**](https://elixir.bootlin.com/linux/latest/ident/Hold), [**Wait**](https://elixir.bootlin.com/linux/latest/ident/Wait)};

*/\* softirq mask and active fields moved to irq\_cpustat\_t in*

*\* asm/hardirq.h to get better cache usage. KAO*

*\*/*

struct [**softirq\_action**](https://elixir.bootlin.com/linux/latest/ident/softirq_action)

{

void (\*[**action**](https://elixir.bootlin.com/linux/latest/ident/action))(struct [**softirq\_action**](https://elixir.bootlin.com/linux/latest/ident/softirq_action) \*);

};

Структура softirq\_action состоит из одного поля: указатель на функцию action, которой передается указатель на отложенное прерывание. В старых версиях ядра в структуре было поле данных. В современных ядрах его нет.

Функция open\_softirq() инициализирует гибкое прерывание, присваивая заданному элементу массива softirq\_vec значение action. Функция raise\_softirq() ставит его в очередь на выполнение:

void raise\_softirq(unsigned int nr)

{

unsigned long flags;

local\_irq\_save(flags); /\* сохраняет флаг if и запрещает прерывания\*/

raise\_soft\_irqoff(nr); /\*помечает softirq как отложенное\*/

local\_irq\_restore(flags); /\*восстанавливает значение флага if и разрешает прерывания\*/

}

Гибкие прерывания (softirqs) являются повторно входимыми - они могут выполняться параллельно на любом процессоре. Причем параллельно могут выполняться одни и те же softirq. Именно это обстоятельство и требует повторной входимости и использования вреств взаимоисключения. Гибкие прерывания выполняются в соответствии с их номерами, которые определяют их приоритет. Гибкие прерывания с меньшими номерами выполняются раньше. Отложенное прерывание выполняется в контексте прерывания, а значит для него недопустимы блокирующие операции.

Единственное событие, которое может вытеснить softirq это – аппаратное прерывание.

Отложенные прерывания зарезервированы для наиболее важных и критичных ко времени выполнения обработчиков нижних половин в системе. Сейчас только две подсистемы — подсистема SCSI и сетевая подсистема — напрямую используют механизм softirq. В дополнение к этому, таймеры ядра и тасклеты построены на базе отложенных (гибких) прерываний.  Не рекомендуется добавлять новое отложенное прерывание, так как это можно сделать только перекомпилировав ядро. При необходимости включения новой функциональности можно использовать тасклеты, которые, как было отмечено, построены на базе гибких прерываний и это видно из приведенного перечисления.

Главная причина использования гибких прерываний является их масштабируемость на многие процессоры. Отложенные прерывания следует использовать только, если необходима высокая частота выполнения.

***Тасклеты (tasklets)***

Тасклеты, в отличие от гибких прерываний, могут создаваться динамически, а также их легче использовать в связи с более простыми требованиями к блокировкам.  Это связано с тем, что одинаковые тасклеты не могут выполняться в системе одновременно. Но разные тасклеты могут выполняться параллельно. Таким образом, тасклеты являются хорошим компромиссом между производительностью и простотой реализации.

В [7] перечислены свойства тасклетов:

    \* Если вызывается tasklet\_schedule(), то тасклет гарантированно будет выполняться на некотором процессоре хотя бы один раз после этого.

    \* Если тасклет уже запланирован, но его выполнение все еще не запущено, он будет выполнен только один раз.

    \* Если этот тасклет уже запущен на другом процессоре (или расписание вызывается из самого тасклета), он переносится на более поздний срок.

    \* 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.

Тасклеты описываются структурой:

|  |
| --- |
| struct tasklet\_struct |
| { |
| struct tasklet\_struct \*next; /\* указатель на следующий тасклет\*/ |
| unsigned long state; /\* текущее состояние\*/ |
| atomic\_t count; /\* счетчик ссылок\*/ |
| void (\*func)(unsigned long); /\* обработчик\*/ |
| unsigned long data; /\* данные\*/ |
| };  Поле state может иметь одно из следующих значений:  enum  {  TASCLET\_STATE\_SCHED, /\* запланирован на выполнение\*/  TASKLET\_STATE\_RUN /\* выполняется – только для SMP\*/  }  Если поле count = 0, то тасклет разрешен и может выполняться, если он помечен как запланированный, иначе тасклет запрещен и не может выполняться. |
| Тасклеты могут быть зарегистрированы как статически, так и динамически.  Статически тасклеты создаются с помощью двух макросов, определенных в файле linux/interrupt.h:  DECLARE\_TASKLET(name, func, data);  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 = 1и, следовательно, он запрещен.

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.

Динамически тасклет инициализируется функцией init\_tasklet():

void [**tasklet\_init**](https://elixir.bootlin.com/linux/latest/ident/tasklet_init)(struct [**tasklet\_struct**](https://elixir.bootlin.com/linux/latest/ident/tasklet_struct) \*t,

void (\***[func](https://elixir.bootlin.com/linux/latest/ident/func)**)(unsigned long), unsigned long data)

{

t->next = NULL;

t->state = 0;

[**atomic\_set**](https://elixir.bootlin.com/linux/latest/ident/atomic_set)(&t->count, 0);

t->**[func](https://elixir.bootlin.com/linux/latest/ident/func)** = [**func**](https://elixir.bootlin.com/linux/latest/ident/func);

t->data = data;

}

Для инициализации тасклета, на который указывает заданный указатель struct tasklet\_struct\* t — косвенная ссылка на динамически созданную ранее структуру, необходимо использовать следующий вызов. tasklet\_init(t, tasklet\_handler, dev); /\* динамически, а не статически \*/

Функция-обработчик тасклета должна соответствовать правильному прототипу.

void tasklet\_handler(unsigned long data);

Так же как и в случае отложенных прерываний, тасклет не может переходить в состояние ожидания (блокироваться). Это означает, что в тасклетах нельзя использовать семафоры или другие функции, которые могут блокироваться. Тасклеты также выполняются при всех разрешенных прерываниях, поэтому необходимо принять все меры предосторожности (например, может понадобиться запретить прерывания и захватить блокировку), если тасклет имеет совместно используемые данные с обработчиком прерывания. В отличие от отложенных прерываний, ни один тасклет не выполняется параллельно самому себе, хотя два разных тасклета могут выполняться на разных процессорах параллельно. Если тасклет совместно использует данные с обработчиком прерывания или другим тасклетом, то необходимо использовать соответствующие блокировки

Тасклеты планируются на выполнение. Для планирования тасклетов на выполнение используются две функции: tasklet\_schedule() и tasklet\_hi\_schedule().

static void [**\_\_tasklet\_schedule\_common**](https://elixir.bootlin.com/linux/latest/ident/__tasklet_schedule_common)(struct [**tasklet\_struct**](https://elixir.bootlin.com/linux/latest/ident/tasklet_struct) \*t,

struct [**tasklet\_head**](https://elixir.bootlin.com/linux/latest/ident/tasklet_head) [**\_\_percpu**](https://elixir.bootlin.com/linux/latest/ident/__percpu) \*headp,

unsigned int softirq\_nr)

{

struct [**tasklet\_head**](https://elixir.bootlin.com/linux/latest/ident/tasklet_head) \*[**head**](https://elixir.bootlin.com/linux/latest/ident/head);

unsigned long flags;

[**local\_irq\_save**](https://elixir.bootlin.com/linux/latest/ident/local_irq_save)(flags);

[**head**](https://elixir.bootlin.com/linux/latest/ident/head) = [**this\_cpu\_ptr**](https://elixir.bootlin.com/linux/latest/ident/this_cpu_ptr)(headp);

t->next = NULL;

\*[**head**](https://elixir.bootlin.com/linux/latest/ident/head)->[**tail**](https://elixir.bootlin.com/linux/latest/ident/tail) = t;

[**head**](https://elixir.bootlin.com/linux/latest/ident/head)->[**tail**](https://elixir.bootlin.com/linux/latest/ident/tail) = &(t->next);

[**raise\_softirq\_irqoff**](https://elixir.bootlin.com/linux/latest/ident/raise_softirq_irqoff)(softirq\_nr);

[**local\_irq\_restore**](https://elixir.bootlin.com/linux/latest/ident/local_irq_restore)(flags);

}

void [**\_\_tasklet\_schedule**](https://elixir.bootlin.com/linux/latest/ident/__tasklet_schedule)(struct [**tasklet\_struct**](https://elixir.bootlin.com/linux/latest/ident/tasklet_struct) \*t)

{

[**\_\_tasklet\_schedule\_common**](https://elixir.bootlin.com/linux/latest/ident/__tasklet_schedule_common)(t, &tasklet\_vec, [**TASKLET\_SOFTIRQ**](https://elixir.bootlin.com/linux/latest/ident/TASKLET_SOFTIRQ));

}

[**EXPORT\_SYMBOL**](https://elixir.bootlin.com/linux/latest/ident/EXPORT_SYMBOL)([**\_\_tasklet\_schedule**](https://elixir.bootlin.com/linux/latest/ident/__tasklet_schedule));

void [**\_\_tasklet\_hi\_schedule**](https://elixir.bootlin.com/linux/latest/ident/__tasklet_hi_schedule)(struct [**tasklet\_struct**](https://elixir.bootlin.com/linux/latest/ident/tasklet_struct) \*t)

{

[**\_\_tasklet\_schedule\_common**](https://elixir.bootlin.com/linux/latest/ident/__tasklet_schedule_common)(t, &tasklet\_hi\_vec, [**HI\_SOFTIRQ**](https://elixir.bootlin.com/linux/latest/ident/HI_SOFTIRQ));

}

[**EXPORT\_SYMBOL**](https://elixir.bootlin.com/linux/latest/ident/EXPORT_SYMBOL)([**\_\_tasklet\_hi\_schedule**](https://elixir.bootlin.com/linux/latest/ident/__tasklet_hi_schedule));

Функции tasklet\_schedule() в качестве аргумента передается указатель на соответствующий экземпляр структуры tasklet\_struct:

tasklet\_schedule(&my\_tasklet); /\* отметить, что тасклет my\_tasklet ожидает на выполнение \*/

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.

Тасклет может быть запланирован на выполнение в обработчике прерывания.

Например:

static void irqh\_tasklet\_handler(unsigned long data){

printk(KERN\_INFO “(%ld) irqh tasklet is working.\n”, jiffies);

}

DECLARE\_TASKLET(irqh\_tasclet, irqh\_tasklet\_handler, 0);

static irqreturn\_t irqh\_handler(int irq, void \*dev\_id)

{

…

tasklet\_schedule(&irqh\_tasklet);

…

}

Указанный тасклет может быть запрещен с помощью вызова функции tasklet\_disable(). Если тасклет в данный момент времени выполняется, то эта функция не возвратит управление, пока тасклет не закончит выполняться. Как альтернативу можно использовать функцию tasklet\_disable\_nosync(), которая запрещает указанный тасклет, но возвращается сразу и не ждет, пока тасклет завершит выполнение. Это обычно небезопасно, так как в данном случае нельзя гарантировать, что тасклет не закончил выполнение. Вызов функции tasklet\_enable() разрешает тасклет. Эта функция также должна быть вызвана для того, чтобы можно было использовать тасклет, созданный с помощью макроса DECLARE\_TASKLET\_DISABLED(), как показано в следующем примере.

tasklet\_disable(&my\_tasklet); /\* тасклет теперь запрещен \*/

/\* Мы можем делать все, что угодно, зная, что тасклет не может выполняться. \*/

tasklet\_enable(&my\_tasklet); /\* теперь тасклет разрешен \*/

Из очереди тасклетов, ожидающих выполнения, тасклет может быть удален с помощью функции tasklet\_kill(). Эта функция получает указатель на соответствующую структуру tasklet\_struct в качестве единственного аргумента. Возможность удаления запланированного на выполнение тасклета из очереди очень полезна в случае, когда используются тасклеты, которые сами себя планируют на выполнение. Эта функция сначала ожидает, пока тасклет закончит выполнение, а потом удаляет его из очереди. Однако это, конечно, не может предотвратить возможности, что другой код запланирует этот же тасклет на выполнение. Так как данная функция может переходить в состояние ожидания, то ее нельзя вызывать из контекста прерывания.

***Очереди работ (workqueues)***

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. Концепция рабочей очереди представлена:

/ \*

 \* Поля структуры следуют одному из следующих правил исключения.

 \*

 \* I: Изменяемый путём инициализации / уничтожения и доступен только для чтения всем остальным.

 \*

 \* P: Предохранение защищено. Отключение вытеснения достаточно и должно быть изменено и доступно только из локального процессора.

 \*

 \* L: *pool->lock* защищен. Доступ с *pool->lock* удерживается.

 \*

 \* X: во время нормальной работы модификация требует pool-> lock и должна выполняться только из локального процессора. Либо отключение вытеснения на локальном процессоре, либо захват пула-> блокировки достаточно для доступа на чтение. Если

 \* POOL\_DISASSOCIATED установлен, он идентичен L.

 \*

 \* A: wq\_pool\_attach\_mutex защищен.

 \*

 \* PL: wq\_pool\_mutex защищен.

 \*

 \* PR: wq\_pool\_mutex защищен для записи. RCU защищен от чтения.

 \*

 \* PW: wq\_pool\_mutex и wq-> mutex защищены от записи. Либо для чтения.

 \*

 \* PWR: wq\_pool\_mutex и wq-> mutex защищены от записи. Либо или RCU для чтения.

 \*

 \* WQ: wq-> мьютекс защищен.

 \*

 \* WR: wq-> мьютекс, защищенный для записи. RCU защищен от чтения.

 \*

 \* MD: wq\_mayday\_lock protected.

*\*/*

*/\* struct worker is defined in workqueue\_internal.h \*/*

struct [**worker\_pool**](https://elixir.bootlin.com/linux/latest/ident/worker_pool) {

[**spinlock\_t**](https://elixir.bootlin.com/linux/latest/ident/spinlock_t) lock; */\* the pool lock \*/*

int cpu; */\* I: the associated cpu \*/*

int [**node**](https://elixir.bootlin.com/linux/latest/ident/node); */\* I: the associated node ID \*/*

int id; */\* I: pool ID \*/*

unsigned int flags; */\* X: flags \*/*

unsigned long watchdog\_ts; */\* L: watchdog timestamp \*/*

struct [**list\_head**](https://elixir.bootlin.com/linux/latest/ident/list_head) worklist; */\* L: list of pending works \*/*

int nr\_workers; */\* L: total number of workers \*/*

int nr\_idle; */\* L: currently idle workers \*/*

struct [**list\_head**](https://elixir.bootlin.com/linux/latest/ident/list_head) idle\_list; */\* X: list of idle workers \*/*

struct [**timer\_list**](https://elixir.bootlin.com/linux/latest/ident/timer_list) [**idle\_timer**](https://elixir.bootlin.com/linux/latest/ident/idle_timer); */\* L: worker idle timeout \*/*

struct [**timer\_list**](https://elixir.bootlin.com/linux/latest/ident/timer_list) mayday\_timer; */\* L: SOS timer for workers \*/*

*/\* a workers is either on busy\_hash or idle\_list, or the manager \*/*

[**DECLARE\_HASHTABLE**](https://elixir.bootlin.com/linux/latest/ident/DECLARE_HASHTABLE)(busy\_hash, [**BUSY\_WORKER\_HASH\_ORDER**](https://elixir.bootlin.com/linux/latest/ident/BUSY_WORKER_HASH_ORDER));

*/\* L: hash of busy workers \*/*

struct [**worker**](https://elixir.bootlin.com/linux/latest/ident/worker) \*[**manager**](https://elixir.bootlin.com/linux/latest/ident/manager); */\* L: purely informational \*/*

struct [**list\_head**](https://elixir.bootlin.com/linux/latest/ident/list_head) workers; */\* A: attached workers \*/*

struct [**completion**](https://elixir.bootlin.com/linux/latest/ident/completion) \*detach\_completion; */\* all workers detached \*/*

struct [**ida**](https://elixir.bootlin.com/linux/latest/ident/ida) worker\_ida; */\* worker IDs for task name \*/*

struct [**workqueue\_attrs**](https://elixir.bootlin.com/linux/latest/ident/workqueue_attrs) \***[attrs](https://elixir.bootlin.com/linux/latest/ident/attrs)**; */\* I: worker attributes \*/*

struct [**hlist\_node**](https://elixir.bootlin.com/linux/latest/ident/hlist_node) [**hash\_node**](https://elixir.bootlin.com/linux/latest/ident/hash_node); */\* PL: unbound\_pool\_hash node \*/*

int [**refcnt**](https://elixir.bootlin.com/linux/latest/ident/refcnt); */\* PL: refcnt for unbound pools \*/*

*/\**

*\* The current concurrency level. As it's likely to be accessed*

*\* from other CPUs during try\_to\_wake\_up(), put it in a separate cacheline.*

*\*/*

[**atomic\_t**](https://elixir.bootlin.com/linux/latest/ident/atomic_t) [**nr\_running**](https://elixir.bootlin.com/linux/latest/ident/nr_running) [**\_\_\_\_cacheline\_aligned\_in\_smp**](https://elixir.bootlin.com/linux/latest/ident/____cacheline_aligned_in_smp);

*/\**

*\* Destruction of pool is RCU protected to allow dereferences* *from get\_work\_pool().*

*\*/*

struct [**rcu\_head**](https://elixir.bootlin.com/linux/latest/ident/rcu_head) rcu;

} [**\_\_\_\_cacheline\_aligned\_in\_smp**](https://elixir.bootlin.com/linux/latest/ident/____cacheline_aligned_in_smp);

Данная структура определена kernel / workqueue.c в ядре Linux.

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:

struct [**work\_struct**](https://elixir.bootlin.com/linux/latest/ident/work_struct) {

[**atomic\_long\_t**](https://elixir.bootlin.com/linux/latest/ident/atomic_long_t) data;

struct [**list\_head**](https://elixir.bootlin.com/linux/latest/ident/list_head) entry;

[**work\_func\_t**](https://elixir.bootlin.com/linux/latest/ident/work_func_t) [**func**](https://elixir.bootlin.com/linux/latest/ident/func);

#[**ifdef**](https://elixir.bootlin.com/linux/latest/ident/ifdef) CONFIG\_LOCKDEP

struct [**lockdep\_map**](https://elixir.bootlin.com/linux/latest/ident/lockdep_map) [**lockdep\_map**](https://elixir.bootlin.com/linux/latest/ident/lockdep_map);

#endif

};

Обратим внимание на два поля: func - функция, которая будет запланирована в рабочей очереди, и data - параметр этой функции. Ядро Linux предоставляет специальные потоки для каждого процессора, которые называются kworker:

systemd-cgls -k | grep kworker

├─ 5 [kworker/0:0H]

├─ 15 [kworker/1:0H]

├─ 20 [kworker/2:0H]

├─ 25 [kworker/3:0H]

├─ 30 [kworker/4:0H]

...

Этот процесс можно использовать для планирования отложенных функций рабочих очередей (например, ksoftirqd для softirqs). Помимо этого, можно создать отдельный рабочий поток для рабочей очереди. Ядро Linux предоставляет следующие макросы для создания очереди задач:

#define DECLARE\_WORK(n,f) struct work\_struct n = \_\_WORK\_INITIALIZER(n,f)

#define DECLARE\_DELAYED\_WORK(n,f) struct delayed\_work n = \_\_DELAYED\_WORK\_INITIALIZER(n,f,0)

для статического создания. Он принимает два параметра: имя рабочей очереди и функцию рабочей очереди.

#define [**\_\_WORK\_INITIALIZER**](https://elixir.bootlin.com/linux/latest/ident/__WORK_INITIALIZER)(n, f) { \

.data = [**WORK\_DATA\_STATIC\_INIT**](https://elixir.bootlin.com/linux/latest/ident/WORK_DATA_STATIC_INIT)(), \

.entry = { &(n).entry, &(n).entry },

.**[func](https://elixir.bootlin.com/linux/latest/ident/func)** = (f), \

[**\_\_WORK\_INIT\_LOCKDEP\_MAP**](https://elixir.bootlin.com/linux/latest/ident/__WORK_INIT_LOCKDEP_MAP)(#n, &(n)) \

}

#define [**\_\_DELAYED\_WORK\_INITIALIZER**](https://elixir.bootlin.com/linux/latest/ident/__DELAYED_WORK_INITIALIZER)(n, f, [**tflags**](https://elixir.bootlin.com/linux/latest/ident/tflags)) {

.[**work**](https://elixir.bootlin.com/linux/latest/ident/work) = [**\_\_WORK\_INITIALIZER**](https://elixir.bootlin.com/linux/latest/ident/__WORK_INITIALIZER)((n).[**work**](https://elixir.bootlin.com/linux/latest/ident/work), (f)),

.[**timer**](https://elixir.bootlin.com/linux/latest/ident/timer) = [**\_\_TIMER\_INITIALIZER**](https://elixir.bootlin.com/linux/latest/ident/__TIMER_INITIALIZER)(**[delayed\_work\_timer\_fn](https://elixir.bootlin.com/linux/latest/ident/delayed_work_timer_fn)**,

(**[tflags](https://elixir.bootlin.com/linux/latest/ident/tflags)**) | [**TIMER\_IRQSAFE**](https://elixir.bootlin.com/linux/latest/ident/TIMER_IRQSAFE)),

}

Для создания очереди задач во время выполнения используется:

#define \_\_INIT\_WORK(work, func, onstack)

#define INIT\_WORK(work, func)

#define \_\_INIT\_WORK(\_work, \_func, \_onstack)

do {

\_\_init\_work((\_work), \_onstack);

(\_work)->data = (atomic\_long\_t) WORK\_DATA\_INIT();

INIT\_LIST\_HEAD(&(\_work)->entry);

(\_work)->func = (\_func);

} while (0)

макрос, который принимает структуру work\_struct, которая должна быть создана, и функцию, которая должна быть запланирована в этой рабочей очереди.

Пример:

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

if(work)

{

INIT\_WORK(work, workqueue\_function); /\* динамически создается структура work\_struct и планируется выполнение функции workqueue\_function\*/

….

}

После того, как работа была создана одним из этих макросов, нужно поместить ее в очередь задач. Это можно сделать с помощью функций queue\_work или queue\_delayed\_work:

*/\*\**

*\* queue\_work - queue work on a workqueue*

*\* @wq: workqueue to use*

*\* @work: work to queue*

*\**

*\* Returns %false if @work was already on a queue, %true otherwise.*

*\**

*\* We queue the work to the CPU on which it was submitted, but if the CPU dies*

*\* it can be processed by another CPU.*

*\**

*\* Memory-ordering properties: If it returns %true, guarantees that all stores*

*\* preceding the call to queue\_work() in the program order will be visible from*

*\* the CPU which will execute @work by the time such work executes, e.g.,*

*\**

*\* { x is initially 0 }*

*\**

*\* CPU0 CPU1*

*\**

*\* WRITE\_ONCE(x, 1); [ @work is being executed ]*

*\* r0 = queue\_work(wq, work); r1 = READ\_ONCE(x);*

*\**

*\* Forbids: r0 == true && r1 == 0*

*\*/*

static inline [**bool**](https://elixir.bootlin.com/linux/latest/ident/bool) [**queue\_work**](https://elixir.bootlin.com/linux/latest/ident/queue_work)(struct [**workqueue\_struct**](https://elixir.bootlin.com/linux/latest/ident/workqueue_struct) \***[wq](https://elixir.bootlin.com/linux/latest/ident/wq)**,

struct [**work\_struct**](https://elixir.bootlin.com/linux/latest/ident/work_struct) \*[**work**](https://elixir.bootlin.com/linux/latest/ident/work))

{

[**return**](https://elixir.bootlin.com/linux/latest/ident/return) [**queue\_work\_on**](https://elixir.bootlin.com/linux/latest/ident/queue_work_on)([**WORK\_CPU\_UNBOUND**](https://elixir.bootlin.com/linux/latest/ident/WORK_CPU_UNBOUND), [**wq**](https://elixir.bootlin.com/linux/latest/ident/wq), [**work**](https://elixir.bootlin.com/linux/latest/ident/work));

}

Функция queue\_work вызывает функцию queue\_work\_on, чтобы указать очередь конкретного процессора. Обратите внимание, что передаётся WORK\_CPU\_UNBOUND в функцию queue\_work\_on. Это часть enum, которая определена в include / linux / workqueue.h и представляет рабочую очередь, которая не связана с каким-либо конкретным процессором.

enum {

[**WORK\_STRUCT\_PENDING\_BIT**](https://elixir.bootlin.com/linux/v3.13.1/ident/WORK_STRUCT_PENDING_BIT) = 0, */\* work item is pending execution \*/*

[**WORK\_STRUCT\_DELAYED\_BIT**](https://elixir.bootlin.com/linux/v3.13.1/ident/WORK_STRUCT_DELAYED_BIT) = 1, */\* work item is delayed \*/*

…

*/\* special cpu IDs \*/*

[**WORK\_CPU\_UNBOUND**](https://elixir.bootlin.com/linux/v3.13.1/ident/WORK_CPU_UNBOUND) = [**NR\_CPUS**](https://elixir.bootlin.com/linux/v3.13.1/ident/NR_CPUS),

[**WORK\_CPU\_END**](https://elixir.bootlin.com/linux/v3.13.1/ident/WORK_CPU_END) = [**NR\_CPUS**](https://elixir.bootlin.com/linux/v3.13.1/ident/NR_CPUS) + 1,

…

}

Функция queue\_work\_on проверяет и устанавливает бит WORK\_STRUCT\_PENDING\_BIT для данной работы и выполняет функцию \_\_queue\_work с рабочей очередью для данного процессора и данной работы:

*/\*\**

*\* queue\_work\_on - queue work on specific cpu*

*\* @cpu: CPU number to execute work on*

*\* @wq: workqueue to use*

*\* @work: work to queue*

*\**

*\* We queue the work to a specific CPU, the caller must ensure it*

*\* can't go away.*

*\**

*\* Return: %false if @work was already on a queue, %true otherwise.*

*\*/*

*/\*\**

\* queue\_work\_on - очередь работы на конкретном процессоре

  \* @cpu: номер процессора для выполнения работы

  \* @wq: рабочая очередь для использования

  \* @work: работа в очереди

  \*

  \* Мы ставим работу в очередь на конкретный процессор, вызывающий должен убедиться в этом

  \* не может уйти.

  \*

  \* Return:% false, если @work уже был в очереди, в противном случае - % true.

\*/

[**bool**](https://elixir.bootlin.com/linux/latest/ident/bool) [**queue\_work\_on**](https://elixir.bootlin.com/linux/latest/ident/queue_work_on)(int cpu, struct [**workqueue\_struct**](https://elixir.bootlin.com/linux/latest/ident/workqueue_struct) \***[wq](https://elixir.bootlin.com/linux/latest/ident/wq)**, struct [**work\_struct**](https://elixir.bootlin.com/linux/latest/ident/work_struct) \*[**work**](https://elixir.bootlin.com/linux/latest/ident/work))

{

[**bool**](https://elixir.bootlin.com/linux/latest/ident/bool) ret = [**false**](https://elixir.bootlin.com/linux/latest/ident/false);

unsigned long flags;

[**local\_irq\_save**](https://elixir.bootlin.com/linux/latest/ident/local_irq_save)(flags);

if (!**[test\_and\_set\_bit](https://elixir.bootlin.com/linux/latest/ident/test_and_set_bit)**([**WORK\_STRUCT\_PENDING\_BIT**](https://elixir.bootlin.com/linux/latest/ident/WORK_STRUCT_PENDING_BIT), [**work\_data\_bits**](https://elixir.bootlin.com/linux/latest/ident/work_data_bits)([**work**](https://elixir.bootlin.com/linux/latest/ident/work)))) {

[**\_\_queue\_work**](https://elixir.bootlin.com/linux/latest/ident/__queue_work)(cpu, [**wq**](https://elixir.bootlin.com/linux/latest/ident/wq), [**work**](https://elixir.bootlin.com/linux/latest/ident/work));

ret = [**true**](https://elixir.bootlin.com/linux/latest/ident/true);

}

[**local\_irq\_restore**](https://elixir.bootlin.com/linux/latest/ident/local_irq_restore)(flags);

[**return**](https://elixir.bootlin.com/linux/latest/ident/return) ret;

}

[**EXPORT\_SYMBOL**](https://elixir.bootlin.com/linux/latest/ident/EXPORT_SYMBOL)(**[queue\_work\_on](https://elixir.bootlin.com/linux/latest/ident/queue_work_on)**);

Функция \_\_queue\_work получает рабочий пул. Да, рабочий пул не рабочий. На самом деле, все работы помещаются не в рабочую очередь, а в рабочий пул, который представлен структурой worker\_pool в ядре Linux. Как вы можете видеть выше, структура workqueue\_struct имеет поле pwqs, которое является списком worker\_pools. Когда мы создаем рабочую очередь, она выделяется для каждого процессора pool\_workqueue. Каждый pool\_workqueue связан с worker\_pool, который размещен на том же процессоре и соответствует типу очереди приоритетов. Через них workqueue взаимодействует с worker\_pool. Таким образом, в функции \_\_queue\_work мы устанавливаем процессор на текущий процессор с raw\_smp\_processor\_id (вы можете найти информацию об этом макросе в четвертой части главы процесса инициализации ядра Linux), получая pool\_workqueue для заданной workqueue\_struct и вставляя данную работу в заданную workqueue.

Рабочие очереди создаются с помощью макроса с именем create\_workqueue, который возвращает ссылку на workqueue\_struct.

*/\**

*\* The externally visible workqueue. It relays the issued work items to*

*\* the appropriate worker\_pool through its pool\_workqueues.*

\*Внешне видимая рабочая очередь. Он передает выпущенные рабочие элементы в соответствующий worker\_pool через \*его pool\_workqueues.

*\*/*

struct [**workqueue\_struct**](https://elixir.bootlin.com/linux/latest/ident/workqueue_struct) {

struct [**list\_head**](https://elixir.bootlin.com/linux/latest/ident/list_head) pwqs; */\* WR: all pwqs of this wq \*/*

struct [**list\_head**](https://elixir.bootlin.com/linux/latest/ident/list_head) list; */\* PR: list of all workqueues \*/*

struct [**mutex**](https://elixir.bootlin.com/linux/latest/ident/mutex) [**mutex**](https://elixir.bootlin.com/linux/latest/ident/mutex); */\* protects this wq \*/*

int work\_color; */\* WQ: current work color \*/*

int flush\_color; */\* WQ: current flush color \*/*

[**atomic\_t**](https://elixir.bootlin.com/linux/latest/ident/atomic_t) nr\_pwqs\_to\_flush; */\* flush in progress \*/*

struct [**wq\_flusher**](https://elixir.bootlin.com/linux/latest/ident/wq_flusher) \*first\_flusher; */\* WQ: first flusher \*/*

struct [**list\_head**](https://elixir.bootlin.com/linux/latest/ident/list_head) flusher\_queue; */\* WQ: flush waiters \*/*

struct [**list\_head**](https://elixir.bootlin.com/linux/latest/ident/list_head) flusher\_overflow; */\* WQ: flush overflow list \*/*

struct [**list\_head**](https://elixir.bootlin.com/linux/latest/ident/list_head) maydays; */\* MD: pwqs requesting rescue \*/*

struct [**worker**](https://elixir.bootlin.com/linux/latest/ident/worker) \*rescuer; */\* MD: rescue worker \*/*

int nr\_drainers; */\* WQ: drain in progress \*/*

int saved\_max\_active; */\* WQ: saved pwq max\_active \*/*

struct [**workqueue\_attrs**](https://elixir.bootlin.com/linux/latest/ident/workqueue_attrs) \*unbound\_attrs; */\* PW: only for unbound wqs \*/*

struct [**pool\_workqueue**](https://elixir.bootlin.com/linux/latest/ident/pool_workqueue) \*dfl\_pwq; */\* PW: only for unbound wqs \*/*

#[**ifdef**](https://elixir.bootlin.com/linux/latest/ident/ifdef) CONFIG\_SYSFS

struct [**wq\_device**](https://elixir.bootlin.com/linux/latest/ident/wq_device) \*wq\_dev; */\* I: for sysfs interface \*/*

#endif

#[**ifdef**](https://elixir.bootlin.com/linux/latest/ident/ifdef) CONFIG\_LOCKDEP

char \*lock\_name;

struct [**lock\_class\_key**](https://elixir.bootlin.com/linux/latest/ident/lock_class_key) [**key**](https://elixir.bootlin.com/linux/latest/ident/key);

struct [**lockdep\_map**](https://elixir.bootlin.com/linux/latest/ident/lockdep_map) [**lockdep\_map**](https://elixir.bootlin.com/linux/latest/ident/lockdep_map);

#endif

char name[[**WQ\_NAME\_LEN**](https://elixir.bootlin.com/linux/latest/ident/WQ_NAME_LEN)]; */\* I: workqueue name \*/*

*/\**

*\* 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**](https://elixir.bootlin.com/linux/latest/ident/rcu_head) rcu;

*/\* hot fields used during command issue, aligned to cacheline \*/*

unsigned int flags [**\_\_\_\_cacheline\_aligned**](https://elixir.bootlin.com/linux/latest/ident/____cacheline_aligned); */\* WQ: WQ\_\* flags \*/*

struct [**pool\_workqueue**](https://elixir.bootlin.com/linux/latest/ident/pool_workqueue) [**\_\_percpu**](https://elixir.bootlin.com/linux/latest/ident/__percpu) \*cpu\_pwqs; */\* I: per-cpu pwqs \*/*

struct [**pool\_workqueue**](https://elixir.bootlin.com/linux/latest/ident/pool_workqueue) [**\_\_rcu**](https://elixir.bootlin.com/linux/latest/ident/__rcu) \*numa\_pwq\_tbl[]; */\* PWR: unbound pwqs indexed by node \*/*

};

Рабочую очередь можно удалить позже (при необходимости) с помощью вызова функции destroy\_workqueue.

struct workqueue\_struct \* create\_workqueue (name);

void destroy\_workqueue (struct workqueue\_struct \*);

Нужно использовать create\_singlethread\_workqueue () для создания очереди работ, когда создается только один поток для всех процессоров.

Так как create\_workqueue и create\_singlethread\_workqueue () являются макросами. Оба используют функцию alloc\_workqueue в фоновом режиме.

Пример:

static struct workqueue\_struct \*queue = NULL;

…

queue = create\_workqueue(“my\_workqueue”);

…

**queue\_work(queue, work); /\*добавляет задачу в очередь\*/**

**Использованные и дополнительные источники**

## Linux Device Drivers, 2nd Edition [By Alessandro Rubini & Jonathan Corbet](http://www.oreilly.com/catalog/linuxdrive2/author.html) 2nd Edition June 2001 0-59600-008-1, Order Number: 0081, 586 pages

1. Цирюлик О.И. Модули ядра Linux. Внутренние механизмы ядра

<http://rus-linux.net/MyLDP/BOOKS/Moduli-yadra-Linux/06/kern-mod-06-29.html>

# Обслуживание периферии в коде модулей ядра: Часть 57. Регистрация обработчика прерываний

Олег Цилюрик Опубликовано 30.05.2013 <https://www.ibm.com/developerworks/ru/library/l-linux_kernel_57/>

# Обслуживание периферии в коде модулей ядра: Часть 58. Создание "верхней половины" обработчика прерываний. Олег Цилюрик Опубликовано 04.06.2013 <https://www.ibm.com/developerworks/ru/library/l-linux_kernel_58/>

# Обслуживание периферии в коде модулей ядра: Часть 56. Обработка прерываний Олег Цилюрик Опубликовано 28.05.2013 <https://www.ibm.com/developerworks/ru/library/l-linux_kernel_56/>

# Обслуживание периферии в коде модулей ядра: Часть 60. Тасклеты и очереди отложенных действий

Олег Цилюрик Опубликовано 06.06.2013 <https://www.ibm.com/developerworks/ru/library/l-linux_kernel_60/>

# 7. [torvalds](https://github.com/torvalds)/[linux](https://github.com/torvalds/linux)https://github.com/torvalds/linux/blob/master/include/linux/interrupt.h

1. NUMA – Non-Uniform Memory Access – «неравномерный доступ к памяти» или Non-Uniform Memory Architecture – архитектура с неравномерной памятью – схема реализации оперативной памяти, которая используется в мультипроцессорных системах, когда время доступа к памяти определяется ее расположением относительно процессора. Практически все архитектуры процессоров используют кэши. В NUMA поддержка когерентности кэша приводит к тому, что более один кэш может хранить содержимое одной и той же ячейки памяти. Системы с когерентностью кэша называются ccNUMA. Kernel Linux начиная с версии 2.5 содержит базовые NUMA. Версия 3.8 поддерживает новую NUMA и позволяет более эффективно развивать NUMA политику в следующих версиях. [↑](#footnote-ref-1)