Множественное блокирование

В системах с большим количеством блокировок (ядро именно такая система), необходимость проведения более чем одной блокировки за раз не является необычной для кода. Если какие-то операции должны быть выполнены с использованием двух различных ресурсов, каждый из которых имеет свою собственную блокировку, часто нет альтернативы, кроме получения обоих блокировок. Однако, получение множества блокировок может быть крайне опасным:

```
DEFINE_SPINLOCK( lock1, lock2 );
...
spin_lock ( &lock1 ); /* 1-й фрагмент кода */
spin_lock ( &lock2 );
...
spin_lock ( &lock2 ); /* где-то в совсем другом месте кода... */
spin_lock ( &lock1 );
```

- такой образец кода, в конечном итоге, когда-то обречён на бесконечное блокирование (dead lock).

Если есть необходимость захвата нескольких блокировок, то единственной возможностью есть а). один тот же порядок захвата, б). и освобождения блокировок, в). порядок освобождения обратный порядку захвата, и г). так это должно выглядеть в каждом из фрагментов кода. В этом смысле предыдущий пример может быть переписан так:

```
spin_lock ( &lock1 ); /* так должно быть везде, где использованы lock1 и lock2 */
spin_lock ( &lock2 );
  /* ... здесь выполняется действие */
spin_unlock ( &lock2 );
spin_unlock ( &lock1 );
```

На практике обеспечить такую синхронность работы с блокировками в различных фрагментах кода крайне проблематично! (потому, что это может касаться фрагментов кода разных авторов).

Предписания порядка выполнения

Механизмы, предписывающие порядок выполнения кода, к синхронизирующим механизмам относятся весьма условно, они не являются непосредственно синхронизирующими механизмами, но рассматриваются всегда вместе с ними (по принципу: надо же их где-то рассматривать?).

Одним из таких механизмов являются определённые в <linux/compiler.h> макросы likely() и unlikely(), например:

```
if( unlikely() ) {
  /* сделать нечто редкостное */
};
```

Или:

```
if( likely() ) {
  /* обычное прохождение вычислений */
}
else {
  /* что-то нетрадиционное */
};
```

Такие предписания а). имеют целью оптимизацию компилированного кода, б). делают код более читабельным, в). недопустимы (не определены) в пространстве пользовательского кода (только в ядре).

Примечание: *подобные оптимизации становятся актуальными с появлением в процессорах конвейерных вычислений с предсказыванием.*

Другим примером предписаний порядка выполнения являются барьеры в памяти, препятствующие в процессе оптимизации переносу операций чтения и записи через объявленный барьер. Например, при записи фрагмента кода:

```
a = 1;
b = 2;
```

- порядок выполнения операция, вообще то говоря, непредсказуем, причём последовательность (во времени) выполнения операций может изменить а). компилятор из соображений оптимизации, б). процессор (периода выполнения) из соображений аппаратной оптимизации работы с шиной памяти. В этом случае это совершенно нормально, более того, даже запись операторов:

```
a = 1;
b = a + 1;
```

- будет гарантировать отсутствие перестановок в процессе оптимизации, так как компилятор «видит» операции в едином контексте (фрагменте кода). Но в других случаях, когда операции производятся из различных мест кода нужно гарантировать, что они не будут перенесены через определённые барьеры. Операции (макросы) с барьерами объявлены в </asm-generic/system.h>, на сегодня все они (rmb(), wmb(), mb(), ...) определены одинаково:

```
#define   mb()  asm volatile ("": : :"memory")
```

Все они препятствуют выполнению операций с памятью после такого вызова до завершения всех операций, записанных до вызова.

Ещё один макрос объявлен в <linux/compiler.h>, он препятствует компилятору при оптимизации переставлять операторы до вызова и после вызова :

```
void barrier( void );
```

Обработка прерываний

«Трудное – это то, что может быть сделано немедленно; невозможное – то, что потребует немного больше времени.»

Сантаяна.

Мы закончили рассмотрение механизмов параллелизмов, для случаев, когда это действительно параллельно выполняющиеся фрагменты кода (в случае SMP и наличия нескольких процессоров), или когда это квази-параллельность, и различные ветви асинхронно вытесняют друг друга, занимая единый процессор. Глядя на сложности, порождаемые параллельными вычислениями, можно было бы попытаться и вообще отказаться от параллельных механизмов в угоду простоте и детерминированности последовательного вычислительного процесса. И так и стараются поступить часто в малых и встраиваемых архитектурах. Можно было бы ..., если бы не один вид естественного асинхронного параллелизма, который возникает в любой, даже самой простой и однозадачной операционной системе, такой, например, как MS-DOS, и это — аппаратные прерывания. И наличие такого одного механизма сводит на нет попытки представить реальный вычислительный процесс как чисто последовательностный, как принято в сугубо теоретическом рассмотрении: параллелизм присутствует всегда!

Примечание: *Есть одна область практических применений средств компьютерной индустрии, которая развивается совершенно автономно, и в которой попытались уйти от асинхронного обслуживания аппаратных прерываний, относя именно к наличию этих механизмов риски отказов, снижения надёжности и живучести систем (утверждение, которое само по себе вызывает изрядные сомнения, или, по крайней мере, требующее доказательств, которые на сегодня не представлены). И область эта: промышленные программируемые логические контроллеры (PLC), применяемые в построении систем АСУ ТП экстремальной надёжности. Такие PLC строятся на абсолютно тех же процессорах общего применения, но обменивающиеся с многочисленной периферией не по прерываниям, а методами циклического программного опроса (пулинга), часто с периодом опроса миллисекундного диапазона или даже ниже. Не взирая на некоторую обособленность этой ветви развития, она занимает (в финансовых объёмах) весьма существенную часть компьютерной индустрии, где преуспели такие мировые бренды как: Modicon (ныне Schneider Electric), Siemens, Allen-Bradley и ряд других. Примечательно, что целый ряд известных моделей PLC работают, в том числе, и под операционной системой Linux, но работа с данными в них основывается на совершенно иных принципах, что, собственно, и делает их PLC. Вся эта отрасль стоит особняком, и к её особенностям мы не будем больше обращаться.*

Общая модель обработки 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. Вот как она выглядела, к примеру, в системе MS-DOS для процессоров x86 и «старого» контроллера прерываний (чип 8259) - на уровне ассемблера это нечто подобное последовательности действий:

- После возникновения аппаратного прерывания управление асинхронно получает функция (ваша функция!), адрес которой записан в векторе (вентиле) прерывания.
- Обработку прерывания функция обработчика выполняет при запрещённых следующих прерываниях.
- После завершения обработки прерывания функция-обработчик восстанавливает контроллер прерываний (чип 8259), посылая сигнал о завершении прерывания. Это осуществляется отправкой команды EOI (End Of Interrupt — код 20h) в командный регистр микросхемы 8259. Это однобайтовый регистр адресуется через порт ввода/вывода 20h.
- Функция-обработчик завершается, возвращая управление командой iret (не ret, как все прочие привычные нам функции, вызываемые синхронно!).

Показанная схема слишком архитектурно зависима (по взаимодействию с контроллером прерываний), даже с более современным чипом APIC контроллера процессора x86 схема взаимодействия в деталях будет выглядеть по-другому. Это недопустимо для много-платформенной операционной системы, которой является Linux. Поэтому вводится логическая модель обработки прерываний, в которой аппаратно зависимые элементы взаимодействия берёт на себя ядро, а обработка прерывания разделяется на две последовательные фазы:

- Регистрируется функция обработчика «верхней половины», который выполняется **при запрещённых прерываниях** локального процессора. Именно этой функции передаётся управление при возникновении аппаратного прерывания. Функция возвращает управление ядру системы традиционным return.
- Перед своим завершением функция-обработчик активирует последующее выполнение «нижней половины», которая и завершит позже начатую работу по обработке этого прерывания...
- В этой точке (после return из обработчика верхней половины) ядро завершает всё взаимодействие с аппаратурой контроллера прерываний, разрешает последующие прерывания, восстанавливает контроллер командой завершения обработки прерывания и возвращает управление из прерывания уже именно командой iret...

- А вот запланированная выше к выполнению функция нижней половины будет вызвана ядром в некоторый момент позже (но часто это может быть и непосредственно после завершения return из верхней половины), тогда, когда удобнее будет ядру системы. Принципиально важное отличие функции нижней половины состоит в том, что она выполняется уже **при разрешённых прерываниях**.

Исторически в Linux сменялось несколько разнообразных API реализации этой схемы (сами названия «верхняя половина» и «нижняя половина» - это дословно названия одной из старых схем, которая сейчас не присутствует в ядре). С появлением параллелизмов в ядре Linux, все новые схемы реализации обработчиков нижней половины (рассматриваются далее) построены на выполнении такого обработчика **отдельным потоком ядра**.

= = = = = = = = = =

здесь Рис. : модель обработки аппаратных прерываний

= = = = = = = = = =

Регистрация обработчика прерывания

Функции и определения, реализующие интерфейс регистрации прерывания, объявлены в <linux/interrupt.h>. Первое, что мы должны всегда сделать — это зарегистрировать функцию обработчик прерываний (все прототипы этого раздела взяты из ядра 2.6.37):

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

- где:

irq - номер линии запрашиваемого прерывания.
handler - указатель на функцию-обработчик.
flags - битовая маска опций (описываемая далее), связанная с управлением прерыванием.
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. Данные по указателю dev требуются для удаления только специфицируемого устройства на разделяемой линии IRQ. Первоначально накладывалось единственное требование, чтобы этот указатель был уникальным, например, при размещении-освобождении N однотипных устройств вполне допустимым могла бы быть конструкция:

```
for( int i = 0; i < N; i++ ) request_irq( irq, handler, 0, const char *name,
(void*)i );
```

```
...
for( int i = 0; i < N; i++ ) free_irq( irq, (void*)i );
```

Но позже оказалось целесообразным и удобным использовать именно в качестве *dev — указатель на специфическую для устройства структуру, которая и содержит все характерные данные экземпляра: поскольку для каждого экземпляра создаётся своя копия структуры, то указатели на них и будут уникальны, что и требовалось. На сегодня это общеупотребимая практика увязывать обработчик прерывания со структурами данных устройства.

Примечание: *прототипы irq_handler_t и флаги установки обработчика существенно меняются от версии к версии, например, радикально поменялись после 2.6.19, все флаги, именуемые сейчас IRQF_* до этого именовались SA_*. В результате этого можно встретиться с невозможностью компиляции даже относительно недавно разработанных модулей-драйверов.*

Флаги установки обработчика:
- группа флагов установки обработчика по уровню (level-triggered) или фронту (edge-triggered):

```
#define IRQF_TRIGGER_NONE    0x00000000
#define IRQF_TRIGGER_RISING  0x00000001
#define IRQF_TRIGGER_FALLING 0x00000002
#define IRQF_TRIGGER_HIGH    0x00000004
#define IRQF_TRIGGER_LOW     0x00000008
#define IRQF_TRIGGER_MASK ( IRQF_TRIGGER_HIGH |
IRQF_TRIGGER_LOW |
                    IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING )
#define IRQF_TRIGGER_PROBE   0x00000010
```

- другие (не все, только основные, часто используемые) флаги:

IRQF_SHARED — разрешить разделение (совместное использование) линии IRQ с другими устройствами (PCI шина и устройства).
IRQF_PROBE_SHARED — устанавливается вызывающим, когда он предполагает возможные проблемы с совместным использованием.
IRQF_TIMER — флаг, маркирующий это прерывание как таймерное.
IRQF_PERCPU — прерывание закреплённое монопольно за отдельным CPU.
IRQF_NOBALANCING — флаг, запрещающий вовлекать это прерывание в балансировку IRQ.

При успешной установке функция request_irq() возвращает нуль. Возврат ненулевого значения указывает на то, что произошла ошибка и указанный обработчик прерывания не был зарегистрирован. Наиболее часто встречающийся код ошибки — это значение -EBUSY (ошибки в ядре возвращаются отрицательными значениями!), что указывает на то, что данная линия запроса на прерывание уже занята (или при текущем вызове, или при предыдущем вызове для этой линии не был указан флаг IRQF_SHARED).

Отображение прерываний в /proc

Но, прежде чем дальше углубляться в организацию обработки прерывания, коротко остановимся на том, как мы можем наблюдать и контролировать то, что происходит с прерываниями. Всякий раз, когда аппаратное прерывание обрабатывается процессором, внутренний счётчик прерываний увеличивается, предоставляя возможность контроля за подсистемой прерываний; счётчики отображаются в /proc/interrupts (последняя колонка это и есть имя обработчика, зарегистрированное параметром name в вызове request_irq()). Ниже показана «раскладка» прерываний в архитектуре x86, здесь источник прерываний — стандартный программируемый контроллер прерываний PC 8259 (XT-PIC):

$ cat /proc/interrupts

```
           CPU0
  0:   33675789         XT-PIC timer
  1:      41076         XT-PIC i8042
  2:          0         XT-PIC cascade
  5:         18         XT-PIC uhci_hcd:usb1, CS46XX
  6:          3         XT-PIC floppy
  7:          0         XT-PIC parport0
  8:          1         XT-PIC rtc
  9:          0         XT-PIC acpi
 11:    2153158         XT-PIC ide2, eth0, mga@pci:0000:01:00.0
 12:     347114         XT-PIC i8042
 14:         38         XT-PIC ide0
...
```

Примечание: *Если точнее, то показано схема с двумя каскодно объединёнными (по линии IRQ2) контроллерами 8259, которая была классикой более 20 лет (чип контроллера прерываний 8259 создавался ещё под 8-бит процессор 8080). Эта «классика» начала постепенно вытесняться только в последние 5-10 лет, в связи с широким наступлением SMP архитектур, и применением для них нового контроллера: APIC. Одним из первых ставших стандартным образцом стал чип 82489DX, но на сегодня функции APIC просто вшиты в чипсет системной платы. Архитектура APIC позволяет обслуживать число линий IRQ больше 16-ти, что было пределом на протяжении многих лет.*

Те линии IRQ, для которых не установлены текущие обработчики прерываний, не отображаются в /proc/interrupts. Вот то же самое, но на существенно более новом компьютере с 2-мя процессорами (ядрами), когда источником прерываний является усовершенствованный контроллер прерываний IO-APIC (отслеживаются прерывания по фронту и по уровню: IO-APIC-edge или IO-APIC-level):

$ cat /proc/interrupts

```
           CPU0       CPU1
  0:   47965733          0  IO-APIC-edge      timer
  1:         10          0  IO-APIC-edge      i8042
  4:          2          0  IO-APIC-edge
  7:          0          0  IO-APIC-edge      parport0
  8:          1          0  IO-APIC-edge      rtc0
  9:      24361          0  IO-APIC-fasteoi   acpi
 12:        157        743  IO-APIC-edge      i8042
 14:     700527          0  IO-APIC-edge      ata_piix
 15:     525957          0  IO-APIC-edge      ata_piix
 16:    1146924          0  IO-APIC-fasteoi   i915, eth0
 18:         78     441659  IO-APIC-fasteoi   uhci_hcd:usb4, yenta
 19:          3        777  IO-APIC-fasteoi   uhci_hcd:usb5, firewire_ohci,
tifm_7xx1
 20:    2087614          0  IO-APIC-fasteoi   ehci_hcd:usb1, uhci_hcd:usb2
 21:        190      11976  IO-APIC-fasteoi   uhci_hcd:usb3, HDA Intel
 22:          0          0  IO-APIC-fasteoi   mmc0
 27:          0          0  PCI-MSI-edge      iwl3945
NMI:          0          0  Non-maskable interrupts
...
```

Ещё одним источником (динамической) информации о произошедших (обработанных) прерываниях является файл /proc/stat:

$ cat /proc/stat

```
cpu  2949061 32182 592004 6337626 301037 8087 4521 0 0
cpu0 1403528 14804 320895 3068116 167380 6043 4235 0 0
cpu1 1545532 17377 271108 3269510 133657 2043 285 0 0
intr 139510185 47968356 10 0 0 2 0 0 0 1 24361 0 0 900 0 700531 525967
1147282 0 441737 780 2087674 12166 0 0 0 0 0 0 ...
```

Здесь строка, начинающаяся с intr содержит суммарные по всем процессорам значения обработанных прерываний для всех последовательно линий IRQ.

Теперь, умея хотя бы наблюдать происходящие в системе прерывания, мы готовы перейти к управлению ними.

Обработчик прерываний, верхняя половина

Прототип функции обработчика прерывания уже показывался выше:

```
typedef irqreturn_t (*irq_handler_t)( int irq, void *dev );
```

где :
- irq — линия IRQ;
- dev — уникальный указатель экземпляра обработчика (именно тот, который передавался последним параметром request_irq() при регистрации обработчика).

Это именно та функция, которая будет вызываться в первую очередь при каждом возникновении аппаратного прерывания. Но это вовсе не означает, что при возврате из этой функции работа по обработке текущего прерывания будет завершена (хотя и такой вариант вполне допустим). Из-за этой «неполноты» такой обработчик и получил название «верхняя половина» обработчика прерывания. Дальнейшие действия по обработке могут быть запланированы эти обработчиком на более позднее время, используя несколько различных механизмов, обобщённо называемых «нижняя половина».

Важно то, что код обработчика верхней половины выполняется при запрещённых последующих прерываниях по линии irq (этой же линии) для того локального процессора, на котором этот код выполняется. А после возврата из этой функции локальные прерывания будут вновь разрешены.

Возвращается значение (<linux/irqreturn.h>):

```
typedef int irqreturn_t;
#define IRQ_NONE        (0)
#define IRQ_HANDLED     (1)
#define IRQ_RETVAL(x)   ((x) != 0)
```

IRQ_HANDLED — устройство прерывания распознано как обслуживаемое обработчиком, и прерывание успешно обработано.
IRQ_NONE — устройство не является источником прерывания для данного обработчика, прерывание должно быть передано далее другим обработчикам, зарегистрированным на данной линии IRQ.

Типичная схема обработчика при этом будет выглядеть так:

```
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;
}
```

Пока мы не углубились в дальнейшую обработку, производимую в нижней половине, хотелось бы отметить следующее: в ряде случаев (при крайне простой обработке обработке, но, самое главное, отсутствии возможности очень быстрых наступлений повторных прерываний) оказывается вполне достаточно простого обработчика верхней половины, и нет необходимости мудрить со сложно диагностируемыми механизмами отложенной обработки.

Управление линиями прерывания

Под управлением линиями прерываний, в этом месте описаний, мы будем понимать запрет-разрешение прерываний поодной или нескольким линиям irq.

Раньше существовала возможность вообще запретить прерывания (на время, естественно). Но сейчас («заточенный» под SMP) набор API для этих целей выглядит так: либо вы запрещаете прерывания по всем линиям irq, но локального процессора, либо на всех процессорах, но только для одной линии irq.

Макросы управления линиями прерываний определены в <linux/irqflags.h>. Управление запретом и разрешением прерываний на локальном процессоре:

local_irq_disable() - запретить прерывания на локальном CPU;
local_irq_enable() - разрешить прерывания на локальном CPU;
int irqs_disabled() - возвратить ненулевое значение, если запрещены прерывания на локальном CPU, в противном случае возвращается нуль ;

Напротив, управление (запрет и разрешение) одной выбранной линией irq, но уже относительно всех процессоров в системе, делают макросы:

void disable_irq(unsigned int irq) -
void disable_irq_nosync(unsigned int irq) - обе эти функции запрещают прерывания с линии irq на контроллере (для всех CPU), причём, disable_irq() не возвращается до тех пор, пока все обработчики прерываний, которые в данный момент выполняются, не закончат работу;
void enable_irq(unsigned int irq) - разрешаются прерывания с линии irq на контроллере (для всех CPU);
void synchronize_irq(unsigned int irq) - ожидает пока завершится обработчик прерывания от линии irq (если он выполняется), в принципе, хорошая идея — всегда вызывать эту функцию перед выгрузкой модуля использующего эту линию IRQ;
Вызовы функций disable_irq*() и enable_irq() должны обязательно быть **парными** - каждому вызову функции запрещения линии должен соответствовать вызов функции разрешения. Только после последнего вызова функции enable_irq() линия запроса на прерывание будет снова разрешена.

Пример обработчика прерываний

Обычно затруднительно показать работающий код обработчика прерываний, потому что такой код должен был бы быть связан с реальным аппаратным расширением, и таким образом он будет перегружен специфическими деталями, скрывающими суть происходящего. Но оригинальный пример приведен в [6], откуда мы его и заимствуем (архив IRQ.tgz):

lab1_interrupt.c :

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

#define SHARED_IRQ 17
```

```
static int irq = SHARED_IRQ, my_dev_id, irq_counter = 0;
module_param( irq, int, S_IRUGO );

static irqreturn_t my_interrupt( int irq, void *dev_id ) {
  irq_counter++;
  printk( KERN_INFO "In the ISR: counter = %d\n", irq_counter );
     return IRQ_NONE;  /* we return IRQ_NONE because we are just observing */
}

static int __init my_init( void ) {
     if ( request_irq( irq, my_interrupt, IRQF_SHARED, "my_interrupt", &my_dev_id ) )
      return -1;
     printk( KERN_INFO "Successfully loading ISR handler on IRQ %d\n", irq );
  return 0;
}

static void __exit my_exit( void ) {
  synchronize_irq( irq );
  free_irq( irq, &my_dev_id );
     printk( KERN_INFO "Successfully unloading, irq_counter = %d\n", irq_counter );
}

module_init( my_init );
module_exit( my_exit );
MODULE_AUTHOR( "Jerry Cooperstein" );
MODULE_DESCRIPTION( "LDD:1.0 s_08/lab1_interrupt.c" );
MODULE_LICENSE( "GPL v2" );
```

Логика этого примера в том, что обработчик вешается в цепочку с существующим в системе, но он не нарушает работу ранее работающего обработчика, фактически ничего не выполняет, но подсчитывает число обработанных прерываний. В оригинале предлагается опробовать его с установкой на IRQ сетевой платы, но ещё показательнее — с установкой на IRQ клавиатуры (IRQ 1) или мыши (IRQ 12) на интерфейсе PS/2 (если таковой используется в компьютере):

```
$ cat /proc/interrupts
           CPU0
  0:   20329441         XT-PIC timer
  1:        423         XT-PIC i8042
...
```

```
$ sudo /sbin/insmod lab1_interrupt.ko irq=1
$ cat /proc/interrupts
           CPU0
  0:   20527017         XT-PIC timer
  1:        572         XT-PIC i8042, my_interrupt
 ...
$ sudo /sbin/rmmod lab1_interrupt
 $ dmesg | tail -n5
 In the ISR: counter = 33
 In the ISR: counter = 34
 In the ISR: counter = 35
 In the ISR: counter = 36
 Successfully unloading, irq_counter = 36
$ cat /proc/interrupts
           CPU0
  0:   20568216         XT-PIC timer
  1:        622         XT-PIC i8042
 ...
```

Оригинальность такого подхода в том, что на подобном коде можно начать отрабатывать код модуля реального устройства, ещё не имея самого устройства, и имитируя его прерывания одним из штатных источников прерываний компьютера, с тем, чтобы позже всё это переключить на реальную линию IRQ, используемую устройством.

Отложенная обработка, нижняя половина

Отложенная обработка прерывания предполагает, что некоторая часть действий по обработке результатов прерывания может быть отложена на более позднее выполнение, когда система будет менее загружена. Главная достигаемая здесь цель состоит в том, что отложенную обработку можно производить не в самой функции обработчика прерывания, и к этому моменту времени может быть уже восстановлено разрешение прерываний по обслуживаемой линии (в обработчике прерываний последующие прерывания запрещены).

Термин «нижняя половина» обработчика прерываний как раз и сложился для обозначения той совокупности действий, которую можно отнести к отложенной обработке прерываний. Когда-то в ядре Linux был один из способов организации отложенной обработки, который так и именовался: обработчик нижней половины, но сейчас он неприменим. А термин так и остался как нарицательный, относящийся к всем разным способам организации отложенной обработки, которые и рассматриваются далее.

Отложенные прерывания (softirq)

Отложенные прерывания определяются статически **во время компиляции ядра**. Отложенные прерывания представлены с помощью структур softirq_action, определенных в файле <linux/interrupt.h> в следующем виде (ядро 2.6.37):

```
// структура, представляющая одно отложенное прерывание
struct softirq_action {
  void (*action)(struct softirq_action *);
};
```

В ядре 2.6.18 (и везде в литературе) определение (более раннее) другое:

```
struct softirq_action {
  void (*action)(struct softirq_action *);
  void *data;
};
```

Для уточнения картины с softirq нам недостаточно хэдеров, и необходимо опуститься в рассмотрение исходных кодов реализации ядра (файл <kernel/softirq.c>, если у вас не установлены исходные тексты ядра, что совершенно не есть необходимостью для всего прочего нашего рассмотрения, то здесь вы будете вынуждены это сделать, если хотите повторить наш экскурс):

```
enum {          /* задействованные номера */
  HI_SOFTIRQ=0,
  TIMER_SOFTIRQ,
  NET_TX_SOFTIRQ,
  NET_RX_SOFTIRQ,
  BLOCK_SOFTIRQ,
  BLOCK_IOPOLL_SOFTIRQ,
  TASKLET_SOFTIRQ,
  SCHED_SOFTIRQ,
  HRTIMER_SOFTIRQ,
  RCU_SOFTIRQ, /* Preferable RCU should always be the last softirq */
  NR_SOFTIRQS  /* число задействованных номеров */
};
static struct softirq_action softirq_vec[NR_SOFTIRQS]
char *softirq_to_name[NR_SOFTIRQS] = {
  "HI", "TIMER", "NET_TX", "NET_RX", "BLOCK", "BLOCK_IOPOLL",
  "TASKLET", "SCHED", "HRTIMER",<>"RCU"
};
```

В 2.6.18 (то, что кочует из одного литературного источника в другой) аналогичные описания были заметно проще и статичнее:

```
enum {
  HI_SOFTIRQ=0,
```

```
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    TASKLET_SOFTIRQ
};
static struct softirq_action softirq_vec[32]
```

Следовательно, имеется возможность создать 32 обработчика softirq, и это количество фиксировано. В этой версии ядра (2.6.18) их было 32, из которых задействованных было 6. Эти определения из предыдущей версии помогают лучше понять то, что имеет место в настоящее время.

Динамическая диагностика использования softirq в работающей системе может производиться так:

```
# cat /proc/softirqs
                CPU0       CPU1
      HI:          0          0
   TIMER:   16940626   16792628
  NET_TX:       4936          1
  NET_RX:      96741       1032
   BLOCK:     176178          2
BLOCK_IOPOLL:      0          0
 TASKLET:        570      50738
   SCHED:     835250    1191280
 HRTIMER:       6286       5457
     RCU:   17000398   16867989
```

В любом случае (независимо от версии), добавить новый уровень обработчика (назовём его XXX_SOFT_IRQ) без перекомпиляции ядра мы не сможем. Максимальное число используемых обработчиков softirq не может быть динамически изменено. Отложенные прерывания с меньшим номером выполняются раньше отложенных прерываний с большим номером (приоритетность). Обработчик одного отложенного прерывания никогда не вытесняет другой обработчик softirq. Единственное событие, которое может вытеснить обработчик softirq, — это аппаратное прерывание. Однако на другом процессоре одновременно с обработчиком отложенного прерывания может выполняться другой (**и даже этот же**) обработчик отложенного прерывания. Отложенное прерывание выполняется **в контексте прерывания**, а значит для него недопустимы блокирующие операции.

Если вы решились на перекомпиляцию ядра и создание нового уровня softirq, то для этого необходимо:

- Определить новый индекс (уровень) отложенного прерывания, вписав (файл <linux/interrupt.h>) его константу XXX_SOFT_IRQ в перечисление, где-то,

очевидно, на одну позицию выше TASKLET_SOFTIRQ (иначе зачем переопределять новый уровень и не использовать тасклет?).

- Во время инициализации модуля должен быть зарегистрирован (объявлен) обработчик отложенного прерывания с помощью вызова open_softirq(), который принимает три параметра: индекс отложенного прерывания, функция-обработчик и значение поля data:

```
/* The bottom half */
void xxx_analyze( void *data ) {
  /* Analyze and do ..... */
}
void __init roller_init() {
  /* ... */
  request_irq( irq, xxx_interrupt, 0, "xxx", NULL );
  open_softirq( XXX_SOFT_IRQ, xxx_analyze, NULL );
}
```

- Функция-обработчик отложенного прерывания (в точности как и рассматриваемого ниже тасклета) должна в точности соответствовать правильному прототипу:

```
void xxx_analyze( unsigned long data );
```

- Зарегистрированное отложенное прерывание, для того, чтобы оно было поставлено в очередь на выполнение, должно быть отмечено (генерировано, возбуждено - rise softirq). Это называется генерацией отложенного прерывания. Обычно обработчик аппаратного прерывания (верхней половины) перед возвратом возбуждает свои обработчики отложенных прерываний:

```
/* The interrupt handler */
static irqreturn_t xxx_interrupt( int irq, void *dev_id ) {
  /* ... */
  /* Mark softirq as pending */
  raise_softirq( XXX_SOFT_IRQ );
  return IRQ_HANDLED;
}
```

- Затем в подходящий (для системы) момент времени отложенное прерывание выполняется. Обработчик отложенного прерывания выполняется при разрешенных прерываниях процессора (особенность нижней половины). Во время выполнения обработчика отложенного прерывания новые отложенные прерывания на данном процессоре запрещаются. Однако на другом процессоре обработчики отложенных прерываний могут выполняться. На самом деле, если вдруг генерируется отложенное прерывание в тот момент, когда ещё выполняется предыдущий его обработчик, то такой же обработчик может быть запущен на другом процессоре одновременно с первым обработчиком. Это означает, что любые совместно используемые данные, которые используются в обработчике отложенного прерывания, и даже глобальные данные, которые используются только внутри самого обработчика, должны соответствующим образом блокироваться.

Главная причина использования отложенных прерываний — **масштабируемость на многие процессоры**. Если нет необходимости масштабирования на многие процессоры, то лучшим выбором будет механизм тасклетов.

\

Тасклеты

Предыдущая схема достаточно тяжеловесная, и в большинстве случаем её подменяют тасклеты — механизм на базе тех же softirq с двумя фиксированными индексами HI_SOFTIRQ или TASKLET_SOFTIRQ. Тасклеты это ни что иное, как частный случай реализации softirq. Тасклеты представляются (<linux/interrupt.h>) с помощью структуры:

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

Поле state может принимать только одно из значений: 0, TASKLET_STATE_SCHED, TASLET_STATE_RUN. Значение TASKLET_STATE_SCHED указывает на то, что тасклет запланирован на выполнение, а значение TASLET_STATE_RUN — что тасклет выполняется.

```
enum {
  TASKLET_STATE_SCHED,   /* Tasklet is scheduled for execution */
  TASKLET_STATE_RUN      /* Tasklet is running (SMP only) */
};
```

Поле count используется как счетчик ссылок на тасклет. Если это значение не равно нулю, то тасклет запрещен и не может выполняться; если оно равно нулю, то тасклет разрешен и может выполняться в случае, когда он помечен как ожидающий выполнения.

Схематически код использования тасклета полностью повторяет структуру кода softirq:

- Инициализация тасклета при инициализации модуля:

```
struct xxx_device_struct { /* Device-specific structure */
 /* ... */
 struct tasklet_struct tsklt;
 /* ... */
```

```
}
void __init xxx_init() {
  struct xxx_device_struct *dev_struct;
  /* ... */
  request_irq( irq, xxx_interrupt, 0, "xxx", NULL );
  /* Initialize tasklet */
  tasklet_init( &dev_struct->tsklt, xxx_analyze, dev );
}
```

Для статического создания тасклета (и соответственно, обеспечения прямого доступа к нему) могут использоваться один из 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). Второй макрос создает тасклет, но устанавливает для него значение поля count, равное единице, и, соответственно, этот тасклет будет запрещен для исполнения. Макрос DECLARE_TASKLET(name, func, data) эквивалентен (можно записать и так):

```
struct tasklet_struct namt = { NULL, 0, ATOMIC_INIT(0), func, data ) ;
```

Используется, что совершенно естественно, в точности тот же прототип функции обработчика тасклета, что и в случае отложенных прерываний (в моих примерах просто использована та же функция).

Для того чтобы запланировать тасклет на выполнение (обычно в обработчике прерывания), должна быть вызвана функция tasklet_schedule(), которой в качестве аргумента передается указатель на соответствующий экземпляр структуры struct tasklet_struct :

```
/* The interrupt handler */
static irqreturn_t xxx_interrupt( int irq, void *dev_id ) {
  struct xxx_device_struct *dev_struct;
  /* ... */
  /* Mark tasklet as pending */
  tasklet_schedule( &dev_struct->tsklt );
  return IRQ_HANDLED;
}
```

После того как тасклет запланирован на выполнение, он выполняется один раз в некоторый момент времени в ближайшем будущем. Для оптимизации тасклет всегда выполняется на том процессоре, который его запланировал на выполнение, что дает надежду на лучшее использование кэша процессора.

Если вместо стандартного тасклета нужно использовать тасклет высокого приоритета (HI_SOFTIRQ), то вместо функции tasklet_schedule() вызываем функцию планирования tasklet_hi_schedule().

Уже запланированный тасклет может быть запрещен к исполнению (временно) с помощью вызова функции tasklet_disable(). Если тасклет в данный

момент уже начал выполнение, то функция не возвратит управление, пока тасклет не закончит своё выполнение. Как альтернативу можно использовать функцию tasklet_disable_nosync(), которая запрещает указанный тасклет, но возвращается сразу не ожидая, 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 выполнить тасклет, созданный с помощью макроса DECLARE_TASKLET_DISABLED (). Из очереди тасклетов, ожидающих выполнения, тасклет может быть удален с помощью функции tasklet_kill().

Так же как и в случае отложенных прерываний (на которых он построен), тасклет не может переходить в блокированное состояние.

Демон ksoftirqd

Обработка отложенных прерываний (softirq) и, соответственно, тасклетов осуществляется с помощью набора потоков пространства ядра (по одному потоку на каждый процессор). Потоки пространства ядра помогают обрабатывать отложенные прерывания, когда система перегружена большим количеством отложенных прерываний.

```
$ ps -ALf | head -n12
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
root         1     0     1  0    1 08:55 ?        00:00:01 /sbin/init
...
root         4     2     4  0    1 08:55 ?        00:00:00 [ksoftirqd/0]
...
root         7     2     7  0    1 08:55 ?        00:00:00 [ksoftirqd/1]
...
```

Для каждого процессора существует свой поток. Каждый поток имеет имя в виде ksoftirqd/n , где п — номер процессора. Например, в двухпроцессорной системе будут запущены два потока с именами ksoftirqd/0 и ksoftirqd/1. То, что на каждом процессоре выполняется свой поток, гарантирует, что если в системе есть свободный процессор, то он всегда будет в состоянии выполнять отложенные прерывания. После того как потоки запущены, они выполняют замкнутый цикл.

Очереди отложенных действий (workqueue)

Очереди отложенных действий (workqueue) — это еще один, но совершенно другой, способ реализации отложенных операций. Очереди отложенных действий позволяют откладывать некоторые операции для последующего выполнения **потоком пространства ядра** (эти потоки ядра называют рабочими потоками - worker threads) — отложенные действия всегда выполняются в **контексте процесса**. Поэтому код, выполнение которого

отложено с помощью постановки в очередь отложенных действий, получает все преимущества, которыми обладает код, выполняющийся в контексте процесса, главное из которых — это возможность переходить в блокированные состояния. Рабочие потоки, которые выполняются по умолчанию, называются events/n, где n — номер процессора, для 2-х процессоров это будут events/0 и events/1 :

```
$ ps -ALf | head -n12
...
root         9     2     9  0    1 08:55 ?        00:00:00 [events/0]
root        10     2    10  0    1 08:55 ?        00:00:00 [events/1]
...
```

Когда какие-либо действия ставятся в очередь, поток ядра возвращается к выполнению и выполняет эти действия. Когда в очереди не остается работы, которую нужно выполнять, поток снова возвращается в состояние ожидания. Каждое действие представлено с помощью struct work_struct (определяется в файле <linux/workqueue.h> - очень меняется от версии к версии ядра!):

```
typedef void (*work_func_t)( struct work_struct *work );
struct work_struct {
  atomic_long_t data;      /* аргумент функции-обработчика */
  struct list_head entry;  /* связанный список всех действий */
  work_func_t func;        /* функция-обработчик */
...
};
```

Для создания статической структуры действия на этапе компиляции необходимо использовать макрос:

```
DECLARE_WORK( name, void (*func) (void *), void *data );
```

Это выражение создает struct work_struct с именем name, с функцией-обработчиком func() и аргументом функции-обработчика data. Динамически отложенное действие создается с помощью указателя на ранее созданную структуру, используя следующий макрос:

```
INIT_WORK( struct work_struct *work, void (*func)(void *), void *data );
```

Функция-обработчика имеет тот же прототип, что и для отложенных прерываний и тасклетов, поэтому в примерах будет использоваться та же функция (xxx_analyze()).

Для реализации нижней половины обработчика IRQ на технике workqueue, выполнем последовательность действий примерно следующего содержания:

При инициализации модуля создаём отложенное действие:

```
#include <linux/workqueue.h>
struct work_struct *hardwork;
void __init xxx_init() {
  /* ... */
```

```
  request_irq( irq, xxx_interrupt, 0, "xxx", NULL );
  hardwork = kmalloc( sizeof(struct work_struct), GFP_KERNEL );
  /* Init the work structure */
  INIT_WORK( hardwork, xxx_analyze, data );
}
```

Или то же самое может быть выполнено статически

```
#include <linux/workqueue.h>
DECLARE_WORK( hardwork, xxx_analyze, data );
void __init xxx_init() {
  /* ... */
  request_irq( irq, xxx_interrupt, 0, "xxx", NULL );
}
```

Самая интересная работа начинается когда нужно запланировать отложенное действие; при использовании для этого рабочего потока ядра по умолчанию (events/n) это делается функциями :

- schedule_work(struct work_struct *work); - действие планируется на выполнение немедленно и будет выполнено, как только рабочий поток events, работающий на данном процессоре, перейдет в состояние выполнения.
- schedule_delayed_work(struct delayed_work *work, unsigned long delay); - в этом случае запланированное действие не будет выполнено, пока не пройдет хотя бы заданное в параметре delay количество импульсов системного таймера.

В обработчике прерывания это выглядит так:

```
static irqreturn_t xxx_interrupt( int irq, void *dev_id ) {
  /* ... */
  schedule_work( hardwork );
  /* или schedule_work( &hardwork ); - для статической инициализации */
  return IRQ_HANDLED;
}
```

Очень часто бывает необходимо ждать пока очередь отложенных действий очистится (отложенные действия завершатся), это обеспечивает функция:

```
void flush_scheduled_work( void );
```

Для отмены незавершённых отложенных действий с задержками используется функция:

```
int cancel_delayed_work( struct work_struct *work );
```

Но мы не обязательно должны рассчитывать на общую очереди (потоки ядра events) для выполнения отложенных действий — мы можем создать под эти цели собственные очереди (вместе с обслуживающим потоком). Создание обеспечивается макросами вида:

```
struct workqueue_struct *create_workqueue( const char *name );
struct workqueue_struct *create_singlethread_workqueue( const char *name );
```

Планирование на выполнение в этом случае осуществляют функции:

```
int queue_work( struct workqueue_struct *wq, struct work_struct *work );
int queue_delayed_work( struct workqueue_struct *wq,
                        struct wesrk_struct *work, unsigned long delay);
```

Они аналогичны рассмотренным выше schedule_*(), но работают с созданной очередью, указанной 1-м параметром. С вновь созданными потоками предыдущий пример может выглядеть так:

```
struct workqueue_struct *wq;
/* Driver Initialization */
static int __init xxx_init( void ) {
  /* ... */
  request_irq( irq, xxx_interrupt, 0, "xxx", NULL );
  hardwork = kmalloc( sizeof(struct work_struct), GFP_KERNEL );
  /* Init the work structure */
  INIT_WORK( hardwork, xxx_analyze, data );
  wq = create_singlethread_workqueue( "xxxdrv" );
  return 0;
}
static irqreturn_t xxx_interrupt( int irq, void *dev_id ) {
  /* ... */
  queue_work( wq, hardwork );
  return IRQ_HANDLED;
}
```

Аналогично тому, как и для очереди по умолчанию, ожидание завершения действий в заданной очереди может быть выполнено с помощью функции :

```
void flush_workqueue( struct workqueue_struct *wq );
```

Техника очередей отложенных действий показана здесь на примере обработчика прерываний, но она гораздо шире по сферам её применения (в отличие, например, от тасклетов), для других целей.