

Продолжение про аппаратные прерывания.

Принцип распараллеливания функций

В совр. ОС появилась сетевая подсистема, которая позволяет получать информацию из сети (там тоже есть устр-ва и они тоже явл. внешними) и отправлять информацию из нашего компьютера;

Обработка прер-й — важнейшая задача любой ОС (по крайней мере, общего назначения);

Выполнение аппаратных прер-й должно выполняться на <sup>очень</sup> высоких уровнях приоритетов;

Сложность обработки таких прер-й с течением времени увеличивалась, что привело к увеличению времени выполнения обработчиков прер-й

Прервать вот-е такого обработчика нельзя, он выполняется от начала до конца;

Это очень сильно влияет на отзывчивость системы;

Зальное время — время обслуживания запроса внеш. устр-ва / процесса строго определено — жёсткое реальное время;

Отзывчивость системы  $\neq$  реальное время;  
(реакция на действия пользователя) ( $\neq$  быстро)

См. опр - е реального времени POSIX;

Отзывчивость системы предполагает время ответа на действия пользователя, которое не приводило бы к отсзданию (и это не реальное время);

Обработчики прер-й должны завершаться как можно быстрее, в ОС такие обработчики прер-й стали делить на 2 половины: top half и bottom half;

В рез-те обработчик прер-я должен выполнять минимально необходимые действия. Как правило, это получение данных (если это ввод) из буфера контроллера устр-ва и её передача в буфер ядра;

Планош обработчик прер-я перед своим завершением должен инициализировать отложенное действие;

Отложенные действия в совр. системах бывают 3 видов:

- softirq;
- tasklet;
- workqueue;

softirq определены статически;  
10 обработчиков, представлены в массиве:

```
const char *softirq_to_name[NR_SOFTIRQS] =  
{ "HI", "TIMER", "NET_Tx", "NET_Rx", "BLOCK",  
  "BLOCK_IOPOLL", "TASKLET", "SCHED", "HRTIMER",  
  "RCU" };
```

На softirq определены ф-ции ядра и сист. вызовы;

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

Когда ядро выполняет обработчик отложенного прерывания, ф-ция action() вызывается с указателем на struct softirq\_action в качестве аргумента;

```
extern void open_softirq(int nr,  
    void (*action)(struct softirq_action *));
```

Эта ф-ция инициализирует softirq;

nr - индекс э-та массива softirq\_vec;  
action - указатель на ф-цию softirq,  
которая будет выполняться;

softirq\_to\_vec определен аналогично  
softirq\_to\_name: это массив размером 10  
э-тов, в котором перечисляются имена обра-  
ботчиков, начиная с 0;

Обработчик отложенного действия (softirq), которое зарегистрировано ф-цией `open_softirq()`, в рез-те выт-я этой ф-ции ставится в очередь на выполнение;

Это отложенное действие активируется с помощью ф-ции `raise_softirq`:

```
extern void raise_softirq(unsigned int nr)
{
    unsigned long flags;
    local_irq_save(flags); // сохраняем состояние
                             // флага IF (разрешаем/
                             // запрещаем
                             // прер-я на про-
                             // цессоре)
    raise_softirq_irqoff(nr);
    local_irq_restore(flags);
}
```

Это макросы;

Ф-ция `raise_softirq_irqoff()` может выполняться только с разрешёнными прерываниями;

При этом для каждого процессора существует `ksoftirqd` даемон-демон, работающий с отложенными действиями (softirq);

На примере получения сетевых пакетов продемонстрируем посл-ть действий:

Коротко:

При запуске системы для каждого ядра нажимает выполняться даемон `softirqd`;



? } Это цикл проверки;

Какие-то внешние действия (в частности, инициализация девайса, связанного с сетью) приводит к фактированию события в системе (в данном случае это получение сетевых пакетов), которое должно быть обработано;

Сотв. прер-е, которое придёт от сетевого устр-ва, приведёт к заталкиванию т.н. poll list и затем с помощью softirq daemon, на котором будет выполнена ф-ция krun\_ksoftirqd, эта информация инициализирует вызов ф-ции \_\_do\_softirq();

smpboot.c

① Создание потоков ksoftirqd на CPU

CPU 0

CPU 1

ksoftirqd/0

ksoftirqd/1

② Демон ksoftirqd выполняет циклы обработки в ф-ции run\_ksoftirqd()

⑦ run\_ksoftirqd проверяет softirq-pending bit field

если отсутствует,  
то \_\_do\_softirq()

⑧ \_\_do\_softirq()  
выполняет зарегистрированный обработчик

softnet\_data poll list

□ → □ → □ → □ → □ → ...

③  
⑤ NAPI опрос добавляем в poll\_list

Driver

softirq-pending bit field

□ □ □ □ □

⑥ Драйвер обрабатываем

kernel

net\_dev\_init

④ net\_dev\_init регистрирует softirq обработчик для NET\_RX\_SOFTIRQ

? В перечне функций для сетевой подсистемы присутствует таймер (HI). Всё это выполняется через ksoftirq даемон и вызов соотв. ф-ций ядра. Триходящие аппараты прер-я инициализируют ожидание выполнения зарегистрированного отложенного действия;

NAPI - New API (появилось в версии ядра 2.4)

Появился, видимо, как ответ на возросший объём действий с сетевой подсистемой;

Был разработан для улучшения выполнения (да, улучшения, это цитата) высокоскоростных действий с сетевой подсистемой; (наз. High speed networking)

High speed networking может генерировать (создавать) тысячи прерываний в секунду, и они должны быть обработаны;

Очевидно, что они обрабатываются именно таким образом: сначала возникшее прер-е регистрируется, затем помещается в соотв. очередь и обрабатывается в соотв-ии с этой очередью; курсовая?

Сетевая подсистема явл. канонической для серверов;

TCP - протокол надёжного соединения, т.к. тройного рукопожатия: чтобы начать обрабатывать пакет, клиент запрашивает сервер, сервер посылает ответное сообщение и ждёт подтверждения от клиента. Когда подтверждение от клиента не приходит, соединение наз. полуконкртым. Но такое соединение забивает входную очередь, и в результате легитимные пакеты не обрабатываются;

DDOS - атаки

`raise_softirq_irqoff()` нацелено `softirq` как отключенный путь установки соотв. бита в битовой маске `softirq` (`softirq pending` локального процессора). Это делается в ф-ции `raise_softirq()`;

Если при этом процессор выполняет аппаратное прер-е, то осуществляется выход из ф-ции `raise_softirq_irqoff()` и восстанавливается флаг `IF`. В противном случае (если не в прерывании) вызывается `wakeup_softirqd()`:

```
if (!in_interrupt)
    wakeup_softirqd();
```

Это объясняет, почему каждый процессор имеет собственный `ksoftirqd` daemon;

Итого:



- Каждый `softirq` проходит через след. этапы:
- 1) `softirq` ирригуется ф-цией `open_softirq()`;
  - 2) Обработчик `softirq` отменяется как отложенный с помощью ф-ции `raise_softirq()`;
  - 3) Затем отмененные `softirq` <sup>(перекликаются)</sup> запускаются на выполнение (`triggered`) в каждом следующем цикле выполнения отложенных ф-ций;
  - 4) Выполнение заканчивается завершением;

Цитата из источника (мануал?):

„Попытайтесь избежать выделения новых `softirq's`, если вам не нужно реально высокоскоростное потоковое планирование заданий.“  
↳ в оригинале „высокочастотное“, т. е. если не нужно с высокой частотой выполнять планирование отложенных действий;

Для почти всех нужных целей более чем достаточно `tasklet'ов` (они определены в 10-м `softirq`);

tasklet - спец. тип `softirq`;

Например, все bottom half последовательных устройств должны быть реализованы как `tasklet'ы`, а не как `softirq`;

`tasklet` - многопоточный аналог bottom half;

Ядро версии 6.2.2:

„Этот API устарел. Потсацуйста, рассмолит-те возможность использования потоковых `irq`-запросов вместо этого.“

Что за потоковые `irq`-запросы?

Скорее всего речь идёт об очередях работ (`workqueue`);

В отличие от `softirq`, `tasklet` может выполняться только на одном процессоре, т.е. для `tasklet` недопустимо их параллельное выполнение (речь идёт о параллельном выполнении одного и того же `tasklet`);

Соответственно, `irq` могут выполняться параллельно (см. картинку);

Для `tasklet` установлено ограничение:

один и тот же `tasklet` может выполняться только в единственном экземпляре;

А `softirq` допускают параллельное выполнение  
⇒ увеличение скорости обслуживания таких запросов;

П.к. один и тот же тип `softirq` может выполняться (реально) параллельно, то на них накладываются строгие требования по использованию средств взаимного исключения;

Т.к. один и тот же tasklet не может выполняться параллельно, то таких жестких требований на tasklet'ы не накладывается;

При этом tasklet не может <sup>(застывать)</sup> блокироваться и выполняется от начала до конца;

Для tasklet'ов определена структура (v.6.2.2)

```
struct tasklet_struct // есть в ядре
{
    struct tasklet_struct *next; // ссылается на tasklet'ов
    unsigned long state;
    atomic_t count;
    bool use_callback;
    union {
        void (*func)(unsigned long data);
        void (*callback)(struct tasklet_struct *t);
    };
    unsigned long data;
};
```

в ядре?

tasklet'ы могут регистрироваться в системе как статически, так и динамически; Для статического объявления taskleta существует 2 макроса:

```
DECLARE_TASKLET(name, _callback);
DECLARE_TASKLET_DISABLED(name, _callback);
```

Существуют также объявления т.н. старых tasklet'ов. В них вместо callback используется

func:

```
DECLARE_TASKLET_OLD(name, _func);  
DECLARE_TASKLET_DISABLED_OLD(name, _func);
```

Динамическое объявление tasklet'a:

```
extern void tasklet_init(struct tasklet_struct *t,  
    void (*func)(unsigned long), unsigned long data);
```

После того, как tasklet объявлен, он должен быть  
запланирован на выполнение:

```
static inline void tasklet_schedule(  
    struct tasklet_struct *t)  
{  
    if (!test_and_set_bit(TASKLET_STATE_SCHED,  
        &t->state))  
        __tasklet_schedule();  
}
```

```
static inline tasklet_hi_schedule(...);
```

↳ высокоприоритетные tasklet'ы будут  
выполняться раньше обычных;

В более ранних системах есть ещё tasklet  
schedule\_first, но в совр. ядре этой ф-ции нет;

Ф-цию tasklet\_schedule() можно вызывать  
в обработке прерывания;



## Св-ва tasklet'ов:

- 1) Если вызвана ф-ция `tasklet_schedule()`, то после её вызова `tasklet` гарантированно будет выполнен на каком-либо процессоре хотя бы один раз;
- 2) Если `tasklet` запланирован, но ещё не выполняется, он будет выполнен только один раз;
- 3) Если `tasklet` уже выполняется на другом процессоре или ф-ция `tasklet_schedule()` вызывается из самого `tasklet`'а, то выполнение будет отложено на более поздний срок;
- 4) Если разработчик считает, что в `tasklet`'е необходимо выполнить действия, которые также могут выполняться в другом `tasklet`'е, то в `tasklet`'е надо использовать взаимное исключение с помощью `spin lock`;

## Пояснение:

`tasklet` выполняется от начала до конца только на одном процессоре (параллельно выполняться не может), поэтому для конкретного `tasklet`'а взаимное исключение не нужно.

Но если разработчик предполагает, что два разных `tasklet`'а м.б. заинтересованы в одних и тех же данных, то в кодах этих `tasklet`'ов должно быть реализовано взаимное исключение с помощью `spin lock`;

??? То, что `tasklet`'у не требуется никаких средств взаимного исключения по отношению к самому себе, в принципе

? { как называется сериялизацией;  
где?

На tasklet'ах определены ф-ции  
tasklet\_disable() и tasklet\_enable();

struct tasklet\_struct

В совр. кодах отключение tasklet'ов из аппарат-  
ных (неделимых) контекстов чревато ошиб-  
ками, и этого следует избегать;

tasklet\_kill(struct tasklet\_struct \*);  
tasklet\_setup();

В я/р: tasklet манипулируется в обработ-  
чике аппаратного прерывания;

Планирование tasklet'а может выполняться  
в ф-ции модуля module\_init();

Обработчик аппаратного прер-я надо ре-  
гистривать на разделяемой линии irq;

Флаг IRQF\_SHARED позволяет зарегистри-  
ровать наш обработчик прер-я на линии  
irq, связанной с клавиатурой;

Сейчас IRQ12 не видно, но видно IRQ8(API);  
Раньше существовал спец. флаг быстрого  
прер-я (quick). Сейчас единств. быстрым прер-ем  
осталось прер-е от сист. таймера;

## Итрехишинная архитектура

Показать прер-я можно через `/proc/interrupts`

Необходимо повесить обработчик прер-я на линию `IRQ1` (9-е прер-е);

Необходимо написать обработчик аппаратного прер-я, который будет планировать выполнение отложенного действия в виде `tasklet'a`;

`tasklet` должен выводить время, когда он был запланирован и когда был выполнен;

Как правило, `tasklet` выполняется на том же процессоре, на котором выполнялся обработчик прер-я, который его запланировал;

В наших архитектурах, когда выполняется аппаратное прер-е, на том процессоре, на котором выполняется обработчик прер-я, все прер-я запрещены;

А в системе запрещены прер-я на всех процессорах по данной линии `IRQ`;