

# FreeBSD 5 中断处理

wheelz

March 21, 2006

## Contents

<b>1</b>	<b>概述</b>	<b>2</b>
<b>2</b>	<b>登记IRQ中断源</b>	<b>2</b>
2.1	数据结构与函数 . . . . .	2
2.2	8259A的登记过程 . . . . .	9
<b>3</b>	<b>IRQ中断的处理过程</b>	<b>12</b>
<b>4</b>	<b>软件中断swi</b>	<b>21</b>
4.1	登记 . . . . .	21
4.2	调度 . . . . .	22

# 1 概述

FreeBSD 5 内核中断处理的最大特点是将中断处理程序在线程的上下文中运行。为此，内核为每个注册的中断源（即vector）准备一个内核线程，即中断线程，其任务就是等待中断的发生，一旦发生，便运行相应的中断处理程序。

FreeBSD 5这样做，有好处也有坏处。好处是可以简化线程和中断的互斥关系，并使得中断处理可以阻塞。坏处是每次响应中断都要进行线程调度，可能有两次线程上下文的切换（从用户线程切到中断线程再切回来）。未来的想法是进行lazy scheduling，即尽可能借用当前线程的上下文，只有在中断要阻塞时才进行真正的调度。

与中断有关的源代码主要在

```
sys/kern/kern_intr.c (与体系结构无关的中断代码)
sys/i386/i386/intr_machdep.c (与i386体系结构相关的中断代码)
sys/i386/isa/atpic.c (与8259A相关的.c代码)
sys/i386/isa/atpic_vector.s (与8259A相关的.s代码)
```

## 2 登记IRQ中断源

### 2.1 数据结构与函数

中断向量表有多个vector，0-31为CPU用，32-32+15对应IRQ0-IRQ15 一个vector对应一个source，数据类型是struct intsrc

```
/*
 * An interrupt source. The upper-layer code uses the PIC methods to
 * control a given source. The lower-layer PIC drivers can store additional
 * private data in a given interrupt source such as an interrupt pin number
 * or an I/O APIC pointer.
 */
struct intsrc {
    struct pic *is_pic;
    struct ithd *is_ithread;
    u_long *is_count;
    u_long *is_straycount;
    u_int is_index;
};
```

其实在vector后面的是中断控制器，如8259A，I/O APIC等，事实上，对中断源的控制实际上就是对中断控制器的操作，因此，在struct intsrc中

有成员`struct pic *is_pic`，即中断控制器的操作函数表，通过这个表，可以为不同的中断控制器定义不同的操作，达到demultiplex的作用。这里pic是 programmable interrupt controller的意思。

```
/*
 * Methods that a PIC provides to mask/unmask a given interrupt source,
 * "turn on" the interrupt on the CPU side by setting up an IDT entry, and
 * return the vector associated with this source.
 */
struct pic {
    void (*pic_enable_source)(struct intsrc *);
    void (*pic_disable_source)(struct intsrc *);
    void (*pic_eoi_source)(struct intsrc *);
    void (*pic_enable_intr)(struct intsrc *);
    int (*pic_vector)(struct intsrc *);
    int (*pic_source_pending)(struct intsrc *);
    void (*pic_suspend)(struct intsrc *);
    void (*pic_resume)(struct intsrc *);
};
```

系统中所有的中断源组成一个数组，由于当采用I/O APIC作为中断控制器时，可以有191个中断号(IRQ)，因此该数组大小定义为191。

```
static struct intsrc *interrupt_sources[NUM_IO_INTS];

/* With I/O APIC's we can have up to 191 interrupts. */
#define NUM_IO_INTS 191
```

所谓登记中断源，就是将实际的中断控制器的对应`struct intsrc`数据结构添加到该数组中去。同时，系统为每个登记的中断源创建一个中断线程，中断处理程序就在该线程的上下文中运行，该线程的入口函数为`ithread_loop()`，

`struct intsrc`结构成员`is_ithread`指向描述中断线程的数据结构`struct ithd`，而`struct ithd`结构成员`it_td`指向真正的线程结构`struct thread`，从而将中断与系统的调度单元线程联系起来。

```
/*
 * Describe an interrupt thread. There is one of these per interrupt vector.
 * Note that this actually describes an interrupt source. There may or may
 * not be an actual kernel thread attached to a given source.
 */
struct ithd {
```

```

struct    mtx it_lock;
struct    thread *it_td;      /* Interrupt process. */
LIST_ENTRY(ithd) it_list;    /* All interrupt threads. */
TAILQ_HEAD(, intrhand) it_handlers; /* Interrupt handlers. */
struct    ithd *it_interrupted; /* Who we interrupted. */
void      (*it_disable)(uintptr_t); /* Enable interrupt source. */
void      (*it_enable)(uintptr_t); /* Disable interrupt source. */
void      *it_md;            /* Hook for MD interrupt code. */
int       it_flags;         /* Interrupt-specific flags. */
int       it_need;         /* Needs service. */
uintptr_t it_vector;
char      it_name[MAXCOMLEN + 1];
};

/*
 * Register a new interrupt source with the global interrupt system.
 * The global interrupts need to be disabled when this function is
 * called.
 */
int
intr_register_source(struct intsrc *isrc)
{
    int error, vector;

    vector = isrc->is_pic->pic_vector(isrc);
    if (interrupt_sources[vector] != NULL)
        return (EEXIST);
    error = ithread_create(&isrc->is_ithread, (uintptr_t)isrc, 0,
        (mask_fn)isrc->is_pic->pic_disable_source,
        (mask_fn)isrc->is_pic->pic_enable_source, "irq%d:", vector);
    if (error)
        return (error);
    mtx_lock_spin(&intr_table_lock);
    if (interrupt_sources[vector] != NULL) {
        mtx_unlock_spin(&intr_table_lock);
        ithread_destroy(isrc->is_ithread);
        return (EEXIST);
    }
    intrcnt_register(isrc);
    interrupt_sources[vector] = isrc;
    mtx_unlock_spin(&intr_table_lock);
}

```

```

    return (0);
}

int
ithread_create(struct ithd **ithread, uintptr_t vector, int flags,
    void (*disable)(uintptr_t), void (*enable)(uintptr_t), const char *fmt, ...)
{
    struct ithd *ithd;
    struct thread *td;
    struct proc *p;
    int error;
    va_list ap;

    /* The only valid flag during creation is IT_SOFT. */
    if ((flags & ~IT_SOFT) != 0)
        return (EINVAL);

    ithd = malloc(sizeof(struct ithd), M_ITHREAD, M_WAITOK | M_ZERO);
    ithd->it_vector = vector;
    ithd->it_disable = disable;
    ithd->it_enable = enable;
    ithd->it_flags = flags;
    TAILQ_INIT(&ithd->it_handlers);
    mtx_init(&ithd->it_lock, "ithread", NULL, MTX_DEF);

    va_start(ap, fmt);
    vsnprintf(ithd->it_name, sizeof(ithd->it_name), fmt, ap);
    va_end(ap);

    error = kthread_create(ithread_loop, ithd, &p, RFSTOPPED | RFHIGHPID,
        0, "%s", ithd->it_name);
    if (error) {
        mtx_destroy(&ithd->it_lock);
        free(ithd, M_ITHREAD);
        return (error);
    }
    td = FIRST_THREAD_IN_PROC(p); /* XXXKSE */
    mtx_lock_spin(&sched_lock);
    td->td_ksegrp->kg_pri_class = PRI_ITHD;
    td->td_priority = PRI_MAX_ITHD;
    TD_SET_IWAIT(td);
}

```

```

mtx_unlock_spin(&sched_lock);
ithd->it_td = td;
td->td_ithd = ithd;
if (ithread != NULL)
    *ithread = ithd;
CTR2(KTR_INTR, "%s: created %s", __func__, ithd->it_name);
return (0);
}

```

中断源登记完成后，便可以登记中断处理程序了。`struct ithd`有一个成员`it_handlers`，指向一个链表，这个链表是中断处理程序的链表。为什么多个中断处理程序会连接在一个链表中呢？这是因为多个设备可以共享同一个IRQ号，即同一个vector可以登记多个设备的中断处理函数。当中断来临时，系统分别调用各个设备的中断处理函数，由他们自己判断是否是自己的中断。

`intr_add_handler()`函数就是用来登记中断处理程序的，它从系统中分配一个描述中断处理程序的数据结构`struct intrhand`，并将传入的参数，即中断处理函数`driver_intr_t handler`保存在结构`struct intrhand`的成员`ih_handler`中。中断发生时真正处理中断事务的就是该函数。

```

/*
 * Describe a hardware interrupt handler.
 *
 * Multiple interrupt handlers for a specific vector can be chained
 * together.
 */
struct intrhand {
    driver_intr_t    *ih_handler;    /* Handler function. */
    void            *ih_argument;    /* Argument to pass to handler. */
    int             ih_flags;
    const char      *ih_name;        /* Name of handler. */
    struct ithd     *ih_ithread;     /* Ithread we are connected to. */
    int             ih_need;         /* Needs service. */
    TAILQ_ENTRY(intrhand) ih_next;   /* Next handler for this vector. */
    u_char          ih_pri;          /* Priority of this handler. */
};

/* Interrupt handle flags kept in ih_flags */
#define IH_FAST      0x00000001    /* Fast interrupt. */
#define IH_EXCLUSIVE 0x00000002    /* Exclusive interrupt. */
#define IH_ENTROPY   0x00000004    /* Device is a good entropy source. */

```

```
#define IH_DEAD      0x00000008  /* Handler should be removed. */
#define IH_MPSAFE   0x80000000  /* Handler does not need Giant. */
```

这里有几个flag值值得一提。

1. IH\_FAST 指示该中断是快速中断，系统将尽快执行该处理函数，并不将它调度到中断线程的上下文中运行，也就是说这种函数的运行是在中断环境下运行，没有线程的上下文，是为历史遗留的还未迁移到新中断模式下的驱动程序提供的。
2. IH\_EXCLUSIVE 指示该中断是独占IRQ的，即不能和其他设备共享IRQ
3. IH\_MPSAFE 表明该中断处理函数是SMP安全的。

```
int
intr_add_handler(const char *name, int vector, driver_intr_t handler,
                void *arg, enum intr_type flags, void **cookiep)
{
    struct intsrc *isrc;
    int error;

    isrc = intr_lookup_source(vector);
    if (isrc == NULL)
        return (EINVAL);
    error = ithread_add_handler(isrc->is_ithread, name, handler, arg,
                               ithread_priority(flags), flags, cookiep);
    if (error == 0) {
        intrcnt_updatename(isrc);
        isrc->is_pic->pic_enable_intr(isrc);
        isrc->is_pic->pic_enable_source(isrc);
    }
    return (error);
}

int
ithread_add_handler(struct ithd* ithread, const char *name,
                   driver_intr_t handler, void *arg, u_char pri, enum intr_type flags,
                   void **cookiep)
{
    struct intrhand *ih, *temp_ih;

    if (ithread == NULL || name == NULL || handler == NULL)
```

```

    return (EINVAL);

    ih = malloc(sizeof(struct intrhand), M_ITHREAD, M_WAITOK | M_ZERO);
    ih->ih_handler = handler;
    ih->ih_argument = arg;
    ih->ih_name = name;
    ih->ih_ithread = ithread;
    ih->ih_pri = pri;
    if (flags & INTR_FAST)
        ih->ih_flags = IH_FAST;
    else if (flags & INTR_EXCL)
        ih->ih_flags = IH_EXCLUSIVE;
    if (flags & INTR_MPSAFE)
        ih->ih_flags |= IH_MPSAFE;
    if (flags & INTR_ENTROPY)
        ih->ih_flags |= IH_ENTROPY;

    mtx_lock(&ithread->it_lock);
    if ((flags & INTR_EXCL) != 0 && !TAILQ_EMPTY(&ithread->it_handlers))
        goto fail;
    if (!TAILQ_EMPTY(&ithread->it_handlers)) {
        temp_ih = TAILQ_FIRST(&ithread->it_handlers);
        if (temp_ih->ih_flags & IH_EXCLUSIVE)
            goto fail;
        if ((ih->ih_flags & IH_FAST) && !(temp_ih->ih_flags & IH_FAST))
            goto fail;
        if (!(ih->ih_flags & IH_FAST) && (temp_ih->ih_flags & IH_FAST))
            goto fail;
    }

    TAILQ_FOREACH(temp_ih, &ithread->it_handlers, ih_next)
        if (temp_ih->ih_pri > ih->ih_pri)
            break;
    if (temp_ih == NULL)
        TAILQ_INSERT_TAIL(&ithread->it_handlers, ih, ih_next);
    else
        TAILQ_INSERT_BEFORE(temp_ih, ih, ih_next);
    ithread_update(ithread);
    mtx_unlock(&ithread->it_lock);

    if (cookiep != NULL)

```

```

        *cookiep = ih;
        CTR3(KTR_INTR, "%s: added %s to %s", __func__, ih->ih_name,
            ithread->it_name);
        return (0);

fail:
    mtx_unlock(&ithread->it_lock);
    free(ih, M_ITHREAD);
    return (EINVAL);
}

```

## 2.2 8259A的登记过程

下面我们以8259A为例，看看系统是如何为其注册中断源的，即注册INTSRC(0)–INTSRC(15)。描述8259A中断控制器的数据结构是`struct atpic_intsrc`，其第一个成员是一个中断源结构，这种类型定义方法是BSD中常用的方法，起到了面向对象编程中继承的作用。

由于两个级连的8259A中断控制器可以控制16个中断，因此系统注册16个`struct atpic_intsrc`。这些中断响应程序的入口地址是`IDTVEC(atpic_intr ## i)`文件中将扩展成`Xatpic_intr0`至`Xatpic_intr15`，即为函数名的引用。而在`.s`文件中将扩展成

```

        ALIGN_TEXT;
        .globl Xatpic_intr0;
        .type Xatpic_intr0,@function;
Xatpic_intr0:

```

等等，即定义一个全局的函数，也就是说在`.c`文件中只是引用该函数，真正定义该函数的是在`sys/i386/isa/atpic_vector.s`中，该函数实际上就是一个对`atpic_handle_intr()`函数的包装，我们后面还将看到该函数。

```

struct atpic_intsrc {
    struct intsrc at_intsrc;
    int    at_irq;        /* Relative to PIC base. */
    inthand_t *at_intr;
    u_long at_count;
    u_long at_straycount;
};

```

```

static struct atpic_intsrc atintrs[] = {

```

```

INTSRC(0),
INTSRC(1),
INTSRC(2),
INTSRC(3),
INTSRC(4),
INTSRC(5),
INTSRC(6),
INTSRC(7),
INTSRC(8),
INTSRC(9),
INTSRC(10),
INTSRC(11),
INTSRC(12),
INTSRC(13),
INTSRC(14),
INTSRC(15),
};

```

```

#define INTSRC(irq) \
    { { &atpics[(irq) / 8].at_pic }, (irq) % 8, \
      IDTVEC(atpic_intr ## irq) }

```

系统启动时，调用8259A的初始化函数atpic\_init()，为非SLAVE IRQ号注册中断源。并在i386初始化时调用atpic\_startup()函数，注册中断向量IDTVEC(atpic\_intr ## irq)，注意，这只是注册总的包装函数，具体IRQ号的中断处理函数将由设备驱动通过intr\_add\_handler()函数来注册。

```

SYSINIT(atpic_init, SI_SUB_INTR, SI_ORDER_SECOND + 1, atpic_init, NULL)

```

```

static void
atpic_init(void *dummy __unused)
{
    int i;

    /* Loop through all interrupt sources and add them. */
    for (i = 0; i < sizeof(atintrs) / sizeof(struct atpic_intsrc); i++) {
        if (i == ICU_SLAVEID)
            continue;
        intr_register_source(&atintrs[i].at_intsrc);
    }
}

```

```

}

void
init386(first)
    int first;
{
    .....

#ifdef DEV_ISA
    atpic_startup();
#endif

    .....
}

void
atpic_startup(void)
{
    struct atpic_intsrc *ai;
    int i;

    /* Start off with all interrupts disabled. */
    imen = 0xffff;
    i8259_init(&atpics[MASTER], 0);
    i8259_init(&atpics[SLAVE], 1);
    atpic_enable_source((struct_intsrc *)&atintrs[ICU_SLAVEID]);

    /* Install low-level interrupt handlers for all of our IRQs. */
    for (i = 0; i < sizeof(atintrs) / sizeof(struct atpic_intsrc); i++) {
        if (i == ICU_SLAVEID)
            continue;
        ai = &atintrs[i];
        ai->at_intsrc.is_count = &ai->at_count;
        ai->at_intsrc.is_straycount = &ai->at_straycount;
        setidt(((struct atpic *)ai->at_intsrc.is_pic)->at_intbase +
            ai->at_irq, ai->at_intr, SDT_SYS386IGT, SEL_KPL,
            GSEL(GCODE_SEL, SEL_KPL));
    }
}

```

### 3 IRQ中断的处理过程

```
/*
 * Macros for interrupt interrupt entry, call to handler, and exit.
 */
#define INTR(irq_num, vec_name) \
    .text ; \
    SUPERALIGN_TEXT ; \
    IDTVEC(vec_name) ; \
    pushl $0 ; /* dummy error code */ \
    pushl $0 ; /* dummy trap type */ \
    pushal ; /* 8 ints */ \
    pushl %ds ; /* save data and extra segments ... */ \
    pushl %es ; \
    pushl %fs ; \
    mov $KDSEL,%ax ; /* load kernel ds, es and fs */ \
    mov %ax,%ds ; \
    mov %ax,%es ; \
    mov $KPSEL,%ax ; \
    mov %ax,%fs ; \
; \
    FAKE_MCOUNT(13*4(%esp)) ; /* XXX late to avoid double count */ \
    pushl $irq_num; /* pass the IRQ */ \
    call atpic_handle_intr ; \
    addl $4, %esp ; /* discard the parameter */ \
; \
    MEXITCOUNT ; \
    jmp doreti
```

IRQ产生时，系统根据产生中断的IRQ号找到相应的中断向量入口，即此处的IDT\_VEC(vec\_name)，再这里，构造好函数atpic\_handle\_intr()的调用栈后，将转到atpic\_handle\_intr()进行处理。同系统调用一样，这里的调用栈struct intrframe既是atpic\_handle\_intr()的参数，也是中断返回时用以恢复现场的寄存器状态。

```
/* Interrupt stack frame */
struct intrframe {
    int if_vec;
    int if_fs;
    int if_es;
    int if_ds;
```

```

int    if_edi;
int    if_esi;
int    if_ebp;
int    :32;
int    if_ebx;
int    if_edx;
int    if_ecx;
int    if_eax;
int    :32;      /* for compat with trap frame - trapno */
int    :32;      /* for compat with trap frame - err */
/* below portion defined in 386 hardware */
int    if_eip;
int    if_cs;
int    if_eflags;
/* below only when crossing rings (e.g. user to kernel) */
int    if_esp;
int    if_ss;
};

void
atpic_handle_intr(struct intrframe iframe)
{
    struct intsrc *isrc;

    KASSERT((uint)iframe.if_vec < ICU_LEN,
            ("unknown int %d\n", iframe.if_vec));
    isrc = &atintrs[iframe.if_vec].at_intsrc;

    /*
     * If we don't have an ithread, see if this is a spurious
     * interrupt.
     */
    if (isrc->is_ithread == NULL &&
        (iframe.if_vec == 7 || iframe.if_vec == 15)) {
        int port, isr;

        /*
         * Read the ISR register to see if IRQ 7/15 is really
         * pending.  Reset read register back to IRR when done.
         */
        port = ((struct atpic *)isrc->is_pic)->at_ioaddr;

```

```

        mtx_lock_spin(&icu_lock);
        outb(port, OCW3_SEL | OCW3_RR | OCW3_RIS);
        isr = inb(port);
        outb(port, OCW3_SEL | OCW3_RR);
        mtx_unlock_spin(&icu_lock);
        if ((isr & IRQ7) == 0)
            return;
    }
    intr_execute_handlers(isrc, &iframe);
}

```

经过简单的有关8259A特有的检查，`atpic_handle_intr()`就转到`intr_execute_handlers()`继续处理。

`intr_execute_handlers()`是一个重要的函数，它先得到IRQ号，然后判断是否是快速中断，如果是，则直接在当前线程的上下文中运行，如果不是，则调度对应的中断线程来运行。这个处理是被`critical_enter()/critical_exit()`保护起来的，以保证不会嵌套调度中断线程。

```

void
intr_execute_handlers(struct intsrc *isrc, struct intrframe *iframe)
{
    struct thread *td;
    struct ithd *it;
    struct intrhand *ih;
    int error, vector;

    td = curthread;
    td->td_intr_nesting_level++;

    /*
     * We count software interrupts when we process them. The
     * code here follows previous practice, but there's an
     * argument for counting hardware interrupts when they're
     * processed too.
     */
    atomic_add_long(isrc->is_count, 1);
    atomic_add_int(&cnt.v_intr, 1);

    it = isrc->is_ithread;
    if (it == NULL)
        ih = NULL;
}

```

```

else
    ih = TAILQ_FIRST(&it->it_handlers);

/*
 * XXX: We assume that IRQ 0 is only used for the ISA timer
 * device (clk).
 */
vector = isrc->is_pic->pic_vector(isrc);
if (vector == 0)
    clkintr_pending = 1;

critical_enter();
if (ih != NULL && ih->ih_flags & IH_FAST) {
    /*
     * Execute fast interrupt handlers directly.
     * To support clock handlers, if a handler registers
     * with a NULL argument, then we pass it a pointer to
     * a trapframe as its argument.
     */
    TAILQ_FOREACH(ih, &it->it_handlers, ih_next) {
        MPASS(ih->ih_flags & IH_FAST);
        CTR3(KTR_INTR, "%s: executing handler %p(%p)",
            __func__, ih->ih_handler,
            ih->ih_argument == NULL ? iframe :
            ih->ih_argument);
        if (ih->ih_argument == NULL)
            ih->ih_handler(iframe);
        else
            ih->ih_handler(ih->ih_argument);
    }
    isrc->is_pic->pic_eoi_source(isrc);
    error = 0;
}

```

凡是总是有例外，fast中断不在中断线程的上下文中运行，而是直接在用户进程的上下文中运行

```

} else {
    /*
     * For stray and threaded interrupts, we mask and EOI the
     * source.
     */
}

```

```

    isrc->is_pic->pic_disable_source(isrc);
    isrc->is_pic->pic_eoi_source(isrc);
    if (ih == NULL)
        error = EINVAL;
    else
        error = ithread_schedule(it, !cold);
}

```

其他的非快速中断则需要调度。这里先应答中断控制器，然后调度。

```

critical_exit();
if (error == EINVAL) {
    atomic_add_long(isrc->is_straycount, 1);
    if (*isrc->is_straycount < MAX_STRAY_LOG)
        log(LOG_ERR, "stray irq%d\n", vector);
    else if (*isrc->is_straycount == MAX_STRAY_LOG)
        log(LOG_CRIT,
            "too many stray irq %d's: not logging anymore\n",
            vector);
}
td->td_intr_nesting_level--;
}

```

中断线程调度函数`ithread_schedule()`处理有关中断线程调度的工作。

```

int
ithread_schedule(struct ithd *ithread, int do_switch)
{
    struct int_entropy entropy;
    struct thread *td;
    struct thread *ctd;
    struct proc *p;

    /*
     * If no ithread or no handlers, then we have a stray interrupt.
     */
    if ((ithread == NULL) || TAILQ_EMPTY(&ithread->it_handlers))
        return (EINVAL);

    ctd = curthread;

```

```

/*
 * If any of the handlers for this ithread claim to be good
 * sources of entropy, then gather some.
 */
if (harvest.interrupt && ithread->it_flags & IT_ENTROPY) {
    entropy.vector = ithread->it_vector;
    entropy.proc = ctd->td_proc;
    random_harvest(&entropy, sizeof(entropy), 2, 0,
        RANDOM_INTERRUPT);
}

```

如果该中断线程有IT\_ENTROPY标志，说明可以当作随机数的来源。

```

td = ithread->it_td;
p = td->td_proc;
KASSERT(p != NULL, ("ithread %s has no process", ithread->it_name));
CTR4(KTR_INTR, "%s: pid %d: (%s) need = %d",
    __func__, p->p_pid, p->p_comm, ithread->it_need);

/*
 * Set it_need to tell the thread to keep running if it is already
 * running. Then, grab sched_lock and see if we actually need to
 * put this thread on the runqueue. If so and the do_switch flag is
 * true and it is safe to switch, then switch to the ithread
 * immediately. Otherwise, set the needresched flag to guarantee
 * that this ithread will run before any userland processes.
 */
ithread->it_need = 1;

```

设置it\_need，可以保证中断线程不会在还有中断的情况下，错过中断而去睡眠，见ithread\_loop()。

```

mtx_lock_spin(&sched_lock);
if (TD_AWAITING_INTR(td)) {
    CTR2(KTR_INTR, "%s: setrunqueue %d", __func__, p->p_pid);
    TD_CLR_IWAIT(td);
    setrunqueue(td);
    if (do_switch &&
        (ctd->td_critnest == 1) ) {
        KASSERT((TD_IS_RUNNING(ctd)),
            ("ithread_schedule: Bad state for curthread."));
    }
}

```

```

        ctd->td_proc->p_stats->p_ru.ru_nivcsw++;
        if (ctd->td_flags & TDF_IDLETD)
            ctd->td_state = TDS_CAN_RUN; /* XXXKSE */
        mi_switch();
    } else {
        curthread->td_flags |= TDF_NEEDRESCHED;
    }

```

如果中断线程正在睡眠，也就是说中断线程正在等待中断的到来，则将它放入runqueue，马上运行。如果参数指示可以调度，并且当前线程的嵌套调度深度为1，即第一次试图调度中断线程，则进行上下文切换，否则，将不立即调度运行中断线程，而要等到正常调度时再运行。

这里需要指出的是，如果决定mi\_switch()，由于中断线程优先级很高，中断线程将会立即执行，中断处理函数完成后也许将回到这里，也可能有变数，不会马上回到这里(FIXME)，因此前面intr\_execute\_handlers()中先应答中断控制器，将中断处理必须做的先做完。

调度回来后，继续运行，完成整个中断的处理。

```

    } else {
        CTR4(KTR_INTR, "%s: pid %d: it_need %d, state %d",
            __func__, p->p_pid, ithread->it_need, td->td_state);
    }

```

否则，由于已经设置了it\_need=1，已经在运行的中断线程将负责处理之。

```

    mtx_unlock_spin(&sched_lock);

    return (0);
}

```

我们再来看看中断线程本身，该函数较为简单，两个嵌套的循环保证不会遗漏中断，如果中断服务完成，则睡眠，调用mi\_switch()

```

/*
 * This is the main code for interrupt threads.
 */
static void
ithread_loop(void *arg)
{
    struct ithd *ithd;      /* our thread context */
    struct intrhand *ih;   /* and our interrupt handler chain */

```

```

struct thread *td;
struct proc *p;

td = curthread;
p = td->td_proc;
ithd = (struct ithd *)arg; /* point to myself */
KASSERT(ithd->it_td == td && td->td_ithd == ithd,
        ("%s: ithread and proc linkage out of sync", __func__));

/*
 * As long as we have interrupts outstanding, go through the
 * list of handlers, giving each one a go at it.
 */
for (;;) {
    /*
     * If we are an orphaned thread, then just die.
     */
    if (ithd->it_flags & IT_DEAD) {
        CTR3(KTR_INTR, "%s: pid %d: (%s) exiting", __func__,
            p->p_pid, p->p_comm);
        td->td_ithd = NULL;
        mtx_destroy(&ithd->it_lock);
        mtx_lock(&Giant);
        free(ithd, M_ITHREAD);
        kthread_exit(0);
    }
}

```

如果已经删除当前IRQ的中断处理程序，则需要退出中断线程。

```

CTR4(KTR_INTR, "%s: pid %d: (%s) need=%d", __func__,
    p->p_pid, p->p_comm, ithd->it_need);
while (ithd->it_need) {
    /*
     * Service interrupts. If another interrupt
     * arrives while we are running, they will set
     * it_need to denote that we should make
     * another pass.
     */
    atomic_store_rel_int(&ithd->it_need, 0);
}

```

清除it\_need标志，当清除后又有中断发生时，it\_need将变成1，从而循环继续。

```

restart:
    TAILQ_FOREACH(ih, &ithd->it_handlers, ih_next) {
        if (ithd->it_flags & IT_SOFT && !ih->ih_need)
            continue;
        atomic_store_rel_int(&ih->ih_need, 0);
        CTR6(KTR_INTR,
            "%s: pid %d ih=%p: %p(%p) flg=%x", __func__,
            p->p_pid, (void *)ih,
            (void *)ih->ih_handler, ih->ih_argument,
            ih->ih_flags);

        if ((ih->ih_flags & IH_DEAD) != 0) {
            mtx_lock(&ithd->it_lock);
            TAILQ_REMOVE(&ithd->it_handlers, ih,
                ih_next);
            wakeup(ih);
            mtx_unlock(&ithd->it_lock);
            goto restart;
        }
        if ((ih->ih_flags & IH_MPSAFE) == 0)
            mtx_lock(&Giant);
        ih->ih_handler(ih->ih_argument);
    }

```

调用设备驱动的中断服务函数。所有注册到该IRQ的函数都将被调用，各个设备的函数将检查自己设备的状态以确定是否是自己的设备产生的中断。

```

        if ((ih->ih_flags & IH_MPSAFE) == 0)
            mtx_unlock(&Giant);
    }
}

/*
 * Processed all our interrupts. Now get the sched
 * lock. This may take a while and it_need may get
 * set again, so we have to check it again.
 */
WITNESS_WARN(WARN_PANIC, NULL, "suspending ithread");
mtx_assert(&Giant, MA_NOTOWNED);
mtx_lock_spin(&sched_lock);
if (!ithd->it_need) {

```

```

    /*
     * Should we call this earlier in the loop above?
     */
    if (ithd->it_enable != NULL)
        ithd->it_enable(ithd->it_vector);
    TD_SET_IWAIT(td); /* we're idle */
    p->p_stats->p_ru.ru_nvcsw++;
    CTR2(KTR_INTR, "%s: pid %d: done", __func__, p->p_pid);
    mi_switch();
    CTR2(KTR_INTR, "%s: pid %d: resumed", __func__, p->p_pid);
}

```

如果此时`it_need==1`，则说明新来了中断，继续for循环为该中断服务，否则挂起调度。

```

    mtx_unlock_spin(&sched_lock);
}
}

```

## 4 软件中断swi

我们将举例说明软件中断swi。

### 4.1 登记

系统启动时，调用`start_softintr()`登记两个重要的软件中断，软时钟中断和VM软中断。当情况需要时，内核将调用`swi_sched()`来调度软件中断的运行。

```

/*
 * Start standard software interrupt threads
 */
static void
start_softintr(void *dummy)
{
    struct proc *p;

    if (swi_add(&clk_ithd, "clock", softclock, NULL, SWI_CLOCK,
              INTR_MPSAFE, &softclock_ih) ||
        swi_add(NULL, "vm", swi_vm, NULL, SWI_VM, INTR_MPSAFE, &vm_ih))
        panic("died while creating standard software ithreads");
}

```

```

    p = clk_ithd->it_td->td_proc;
    PROC_LOCK(p);
    p->p_flag |= P_NOLOAD;
    PROC_UNLOCK(p);
}

int
swi_add(struct ithd **ithdp, const char *name, driver_intr_t handler,
        void *arg, int pri, enum intr_type flags, void **cookiep)
{
    struct ithd *ithd;
    int error;

    if (flags & (INTR_FAST | INTR_ENTROPY))
        return (EINVAL);

    ithd = (ithdp != NULL) ? *ithdp : NULL;

    if (ithd != NULL) {
        if ((ithd->it_flags & IT_SOFT) == 0)
            return(EINVAL);
    } else {
        error = ithread_create(&ithd, pri, IT_SOFT, NULL, NULL,
            "swi%d:", pri);
        if (error)
            return (error);

        if (ithdp != NULL)
            *ithdp = ithd;
    }
    return (ithread_add_handler(ithd, name, handler, arg,
        (pri * RQ_PPQ) + PI_SOFT, flags, cookiep));
}

```

## 4.2 调度

硬件时钟中断，需要处理非紧急时钟事务时，调度softclock，以便在响应完硬件时钟中断后 运行softclock。

```
/*
```

```

    * The real-time timer, interrupting hz times per second.
    */
void
hardclock(frame)
    register struct clockframe *frame;
{
    .....

    if (need_softclock)
        swi_sched(softclock_ih, 0);

    .....
}

/*
 * Schedule a heavyweight software interrupt process.
 */
void
swi_sched(void *cookie, int flags)
{
    struct intrhand *ih = (struct intrhand *)cookie;
    struct ithd *it = ih->ih_ithread;
    int error;

    atomic_add_int(&cnt.v_intr, 1); /* one more global interrupt */

    CTR3(KTR_INTR, "swi_sched pid %d(%s) need=%d",
        it->it_td->td_proc->p_pid, it->it_td->td_proc->p_comm, it->it_need);

    /*
     * Set ih_need for this handler so that if the ithread is already
     * running it will execute this handler on the next pass.  Otherwise,
     * it will execute it the next time it runs.
     */
    atomic_store_rel_int(&ih->ih_need, 1);
    if (!(flags & SWI_DELAY)) {
        error = ithread_schedule(it, !cold && !dumping);
        KASSERT(error == 0, ("stray software interrupt"));
    }
}

```