

Interrupts and Time

Jaye-Jyh Tsai

Inst. Of CIS, NCTU

March 11, 2002



Outline

- What are interrupts?
- Data Structures
- Actions and IRQs
- Hardware Interrupt Handlers
- Softirqs, Tasklets, and Bottom Halves
- Time
- References

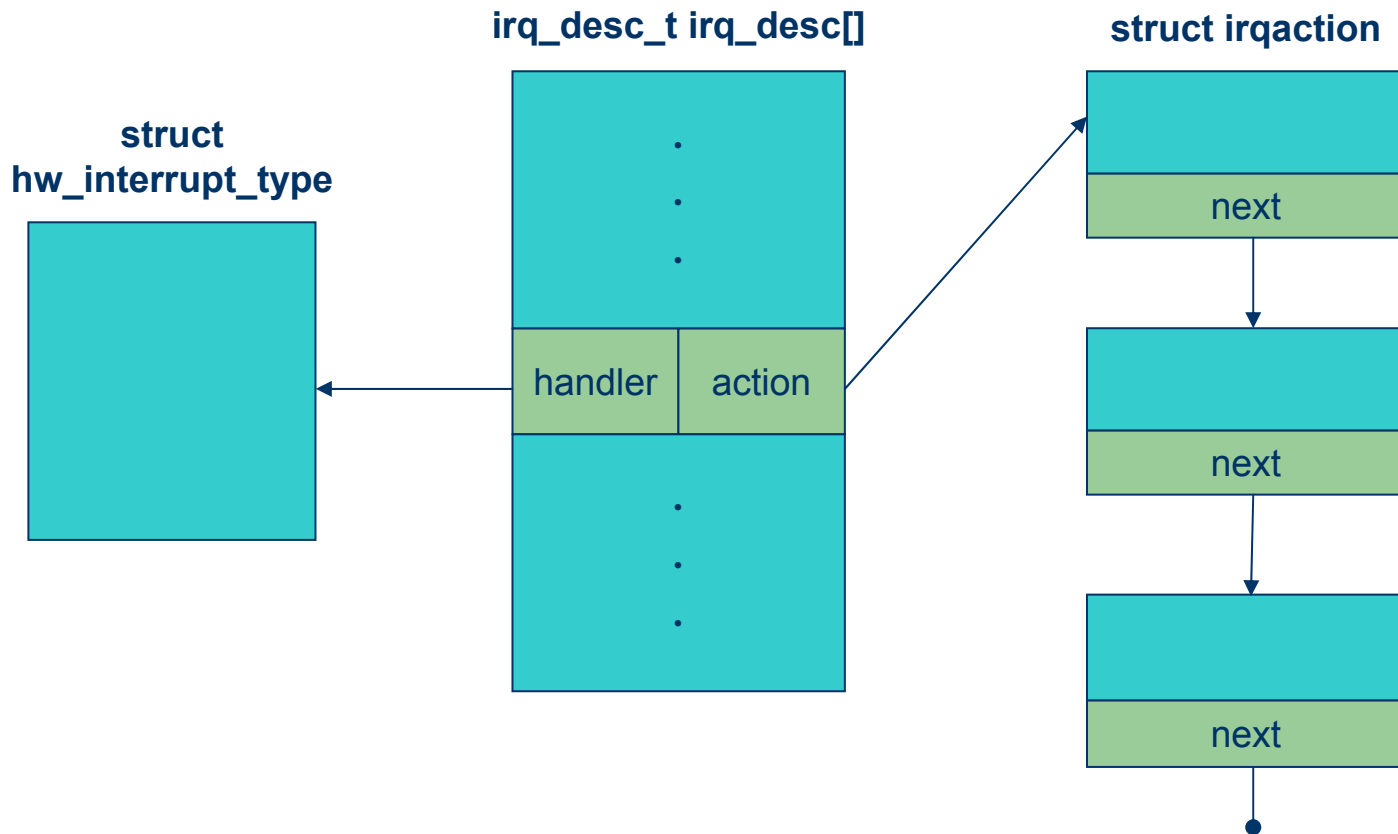
Interrupts – What are interrupts? (1)

- Like signals for the kernel
- Normally sent to the kernel from a **hardware device**
- Used to interrupt the system's normal processing
- Might cause a transition to kernel mode and back
- Can occur while the kernel is already running in kernel mode
- **Clear the interrupts flag – cli**
- **Set the interrupts flag - sti**

Interrupts – What are interrupts? (2)

- Interrupt request (IRQ)
 - an interrupt notification sent from a hardware device to the CPU
- In response to the IRQ
 - the CPU jumps to an interrupt service routine (ISR)
 - More commonly called an interrupt handler
- IRQs are numbered
 - Each hardware device is associated with an IRQ number
 - On IBM PC architecture, IRQ 0 is the hardware timer
- Two kinds of interrupts – fast and slow
 - Fast interrupts cannot be interrupted

Interrupts – Data Structures (1)



Interrupts – Data Structures (2)

include/linux/interrupts.h

```
struct irqaction {  
    void (*handler)(int, void *, struct pt_regs *);  
    unsigned long flags;  
    unsigned long mask;  
    const char *name;  
    void *dev_id;  
    struct irqaction *next;  
};
```

include/asm-i386/signal.h

```
#define SA_INTERRUPT          0x20000000  
#define SA_SAMPLE_RANDOM    SA_RESTART  
#define SA_SHIRQ             0x04000000
```

Interrupts – Data Structures (3)

```
include/linux/irq.h
```

```
struct hw_interrupt_type {
    const char * typename;
    unsigned int (*startup)(unsigned int irq);
    void (*shutdown)(unsigned int irq);
    void (*enable)(unsigned int irq);
    void (*disable)(unsigned int irq);
    void (*ack)(unsigned int irq);
    void (*end)(unsigned int irq);
    void (*set_affinity)(unsigned int irq, cpumask_t mask);
};

typedef struct {
    unsigned int status;
    hw_irq_controller *handler;
    struct irqaction *action;
    unsigned int depth;
    spinlock_t lock;
} ____cacheline_aligned irq_desc_t;

#define IRQ_INPROGRESS 1 /* IRQ handler active - do not enter! */
#define IRQ_DISABLED 2 /* IRQ disabled - do not enter! */
#define IRQ_PENDING 4 /* IRQ pending - replay on enable */
#define IRQ_REPLAY 8 /* IRQ has been replayed but not acked yet */
#define IRQ_AUTODETECT 16 /* IRQ is being autodetected */
#define IRQ_WAITING 32 /* IRQ not yet seen - for autodetection */
#define IRQ_LEVEL 64 /* IRQ level triggered */
#define IRQ_MASKED 128 /* IRQ masked - shouldn't be seen again */
#define IRQ_PER_CPU 256 /* IRQ is per CPU */

/* IRQ action list */
/* nested irq disables */
```

Interrupts – Act

arch/i386/kernel/i8259.c – init_IRQ()

```
void __init init_IRQ(void)
{
    int i;

    #ifndef CONFIG_X86_VISWS_APIC
        init_ISA_irqs();
    #else
        init_VISWS_APIC_irqs();
    #endif

    for (i = 0; i < NR_IRQS; i++) {
        int vector = FIRST_EXTERNAL_VECTOR + i;
        if (vector != SYSCALL_VECTOR)
            set_intr_gate(vector, interrupt_handlers[i]);
    }
    ...
}
```

arch/i386/kernel/i8259.c

```
void __init init_ISA_irqs (void)
{
    int i;

    #ifdef CONFIG_X86_LOCAL_APIC
        init_bsp_APIC();
    #endif
    init_8259A(0);

    for (i = 0; i < NR_IRQS; i++) {
        irq_desc[i].status = IRQ_DISABLED;
        irq_desc[i].action = 0;
        irq_desc[i].depth = 1;

        if (i < 16) {
            /*
             * 16 old-style INTA-cycle interrupts:
             */
            irq_desc[i].handler = &i8259A_irq_type;
        } else {
            /*
             * 'high' PCI IRQs filled in on demand
             */
            irq_desc[i].handler = &no_irq_type;
        }
    }
}
```

Interrupts – Actions and IRQs (1)

arch/i386/kernel/i8259.c – init_IRQ()

```
void __init init_IRQ(void)
{
    int i;

#ifdef CONFIG_X86_VISWS_APIC
    init_ISA_irqs();
#else
    init_VISWS_APIC_irqs();
#endif

    for (i = 0; i < NR_IRQS; i++) {
        int vector = FIRST_EXTERNAL_VECTOR + i;
        if (vector != SYSCALL_VECTOR)
            set_intr_gate(vector, interrupt_handlers[i]);
    }
    ...
}
```

arch/i386/kernel/traps.c

```
/*
 * This needs to use 'idt_table' rather than 'idt', and
 * thus use the _nonmapped_ version of the IDT, as the
 * Pentium F0 0F bugfix can have resulted in the mapped
 * IDT being write-protected.
 */
void set_intr_gate(unsigned int n, void *addr)
{
    _set_gate(idt_table+n,14,0,addr);
}
```

Interrupts – Actions and IRQs (1)

arch/i386/kernel/i8259.c – init_IRQ()

```
...
#ifdef CONFIG_SMP
...
#endif

#ifdef CONFIG_X86_LOCAL_APIC
    /* self generated IPI for local APIC timer */
    set_intr_gate(LOCAL_TIMER_VECTOR, apic_timer_interrupt);

    /* IPI vectors for APIC spurious and error interrupts */
    set_intr_gate(SPURIOUS_APIC_VECTOR, spurious_interrupt);
    set_intr_gate(ERROR_APIC_VECTOR, error_interrupt);
#endif

...
```

Interrupts – Actions and IRQs (1)

`arch/i386/kernel/i8259.c – init_IRQ()`

```
...
/*
 * Set the clock to HZ Hz, we already have a valid
 * vector now:
 */
outb_p(0x34,0x43);           /* binary, mode 2, LSB/MSB, ch 0 */
outb_p(LATCH & 0xff , 0x40); /* LSB */
outb(LATCH >> 8 , 0x40);    /* MSB */

#ifdef CONFIG_VISWS
    setup_irq(2, &irq2);
#endif

    if (boot_cpu_data.hard_math && !cpu_has_fpu)
        setup_irq(13, &irq13);
}
```

Interrupts – Actions and IRQs (1)

arch/i386/kernel/i8259.c – init_IRQ()

...

/*

* Set the clock to HZ Hz, we already have

* vector now:

*/

outb_p(0x34,0x43);

outb_p(LATCH & 0xff , 0x40);

outb(LATCH >> 8 , 0x40);

include/asm-i386/param.h

#define HZ 100

include/asm-i386/timex.h

#define CLOCK_TICK_RATE 1193180

include/linux/timex.h

#define LATCH ((CLOCK_TICK_RATE) + HZ/2) / HZ

/* MSB */

#ifndef CONFIG_VISWS

setup_irq(2, &irq2);

#endif

arch/i386/kernel/i8259.c

#ifndef CONFIG_VISWS

static struct irqaction irq2 = { no_action, 0, 0, "cascade", NULL, NULL};

#endif

if (boot_cpu_data.hard_math && !cpu_has_fpu)

setup_irq(13, &irq13);

}

Interru

arch/i386/kernel/i8259.c -

```
...
/*
 * Set the clock to
 * vector now:
 */
outb_p(0x34,0x4)
outb_p(LATCH)
outb(LATCH >

#ifdef CONFIG
setup_irq(
#endif

if (boot_cpu_data
    setup_irq(13
}
}
```

arch/i386/kernel/irq.c

```
int setup_irq(unsigned int irq, struct irqaction * new)
{
    int shared = 0;
    unsigned long flags;
    struct irqaction *old, **p;
    irq_desc_t *desc = irq_desc + irq;

    if (new->flags & SA_SAMPLE_RANDOM) {
        rand_initialize_irq(irq);
    }

    spin_lock_irqsave(&desc->lock,flags);
    p = &desc->action;
    if ((old = *p) != NULL) {
        /* Can't share interrupts unless both agree to */
        ...
        /* add new interrupt at end of irq queue */
        ...
    }
    *p = new;

    if (!shared) {
        desc->depth = 0;
        desc->status &= ~(IRQ_DISABLED | IRQ_AUTODETECT | IRQ_WAITING);
        desc->handler->startup(irq);
    }
    spin_unlock_irqrestore(&desc->lock,flags);

    register_irq_proc(irq);
    return 0;
}
```

Interrupts – Actions and IRQs (2)

`arch/i386/kernel/irq.c – request_irq()`

```
int request_irq(unsigned int irq, void (*handler)(int, void *, struct pt_regs *), unsigned long irqflags,
               const char * devname, void *dev_id)
{
    int retval;
    struct irqaction * action;

#ifdef 1
    if (irqflags & SA_SHIRQ) {
        if (!dev_id)
            printk("Bad boy: %s (at 0x%x) called us without a dev_id!\n", devname, (&irq)[-1]);
    }
#endif
    if (irq >= NR_IRQS)
        return -EINVAL;
    if (!handler)
        return -EINVAL;
    ...
}
```

Interrupts – Actions and IRQs (2)

arch/i386/kernel/irq.c – request_irq()

```
...
action = (struct irqaction *)
    kmalloc(sizeof(struct irqaction), GFP_KERNEL);
if (!action)
    return -ENOMEM;
action->handler = handler;
action->flags = irqflags;
action->mask = 0;
action->name = devname;
action->next = NULL;
action->dev_id = dev_id;
retval = setup_irq(irq, action);
if (retval)
    kfree(action);
return retval;
}
```

Interrupts – Actions and IRQs (3)

arch/i386/kernel/irq.c – free_irq()

```
void free_irq(unsigned int irq, void *dev_id)
{
    irq_desc_t *desc;
    struct irqaction **p;
    unsigned long flags;

    if (irq >= NR_IRQS)
        return;
    desc = irq_desc + irq;
    spin_lock_irqsave(&desc->lock, flags);
    p = &desc->action;
    for (;;) {
        struct irqaction * action = *p;
        if (action) {
            struct irqaction **pp = p;
            p = &action->next;
            if (action->dev_id != dev_id)
                continue;

```

Interrupts – Actions and IRQs (3)

`arch/i386/kernel/irq.c – free_irq()`

```
    ...
    *pp = action->next;
    if (!desc->action) {
        desc->status |= IRQ_DISABLED;
        desc->handler->shutdown(irq);
    }
    spin_unlock_irqrestore(&desc->lock, flags);
#ifdef CONFIG_SMP
    ...
#endif
    kfree(action);
    return;
}
 printk("Trying to free free IRQ%d\n", irq);
 spin_unlock_irqrestore(&desc->lock, flags);
 return;
}
}
```

Interrupts – Autoprobing for IRQs (1)

1. clear and/or mask the device's internal interrupt
2. `sti();`
3. `irqs = probe_irq_on();`
4. enable the device and cause it to trigger an interrupt
5. wait for the device to interrupt, using non-intrusive polling or a delay
6. `irq = probe_irq_off(irqs);`
7. service the device to clear its pending interrupt
8. loop again if paranoia is required

Interrupts – Autoprobing for IRQs (2)

`arch/i386/kernel/irq.c – probe_irq_on()`

```
unsigned long probe_irq_on(void)
```

```
{
```

```
    unsigned int i;
```

```
    irq_desc_t *desc;
```

```
    unsigned long val;
```

```
    unsigned long delay;
```

```
    down(&probe_sem);
```

```
    for (i = NR_IRQS-1; i > 0; i--) {
```

```
        desc = irq_desc + i;
```

```
        spin_lock_irq(&desc->lock);
```

```
        if (!irq_desc[i].action)
```

```
            irq_desc[i].handler->startup(i);
```

```
        spin_unlock_irq(&desc->lock);
```

```
    }
```

```
    ...
```

```
/*
```

```
 * something may have generated an irq long ago and we want to  
 * flush such a longstanding irq before considering it as spurious.
```

```
*/
```

Interrupts – Autoprobing for IRQs (2)

`arch/i386/kernel/irq.c – probe_irq_on()`

...

`/* Wait for longstanding interrupts to trigger. */`

`for (delay = jiffies + HZ/50; time_after(delay, jiffies);)`

`/* about 20ms delay */ synchronize_irq();`

`for (i = NR_IRQS-1; i > 0; i--) {`

`desc = irq_desc + i;`

`spin_lock_irq(&desc->lock);`

`if (!desc->action) {`

`desc->status |= IRQ_AUTODETECT | IRQ_WAITING;`

`if (desc->handler->startup(i))`

`desc->status |= IRQ_PENDING;`

`}`

`spin_unlock_irq(&desc->lock);`

`}`

`for (delay = jiffies + HZ/10; time_after(delay, jiffies);)`

`/* about 100ms delay */ synchronize_irq();`

...

`/*`

`* enable any unassigned irqs`

`* (we must startup again here because if a longstanding irq`

`* happened in the previous stage, it may have masked itself)`

`*/`

Interrupts – Autoprobing for IRQs (2)

`arch/i386/kernel/irq.c – probe_irq_on()`

```
...
val = 0;
for (i = 0; i < NR_IRQS; i++) {
    irq_desc_t *desc = irq_desc + i;
    unsigned int status;

    spin_lock_irq(&desc->lock);
    status = desc->status;
    if (status & IRQ_AUTODETECT) {
        if (!(status & IRQ_WAITING)) {
            desc->status = status & ~IRQ_AUTODETECT;
            desc->handler->shutdown(i);
        } else
            if (i < 32) val |= 1 << i;
    }
    spin_unlock_irq(&desc->lock);
}
return val;
```

Interrupts – Autoprobing for IRQs (3)

arch/i386/kernel/irq.c – probe_irq_off()

```
int probe_irq_off(unsigned long val)
{
    int i, irq_found, nr_irqs;

    nr_irqs = 0;
    irq_found = 0;
    for (i = 0; i < NR_IRQS; i++)
        ...
}
up(&probe_sem);

if (nr_irqs > 1)
    irq_found = -irq_found;
return irq_found;
}
```

```
irq_desc_t *desc = irq_desc + i;
unsigned int status;

spin_lock_irq(&desc->lock);
status = desc->status;
if (status & IRQ_AUTODETECT) {
    if (!(status & IRQ_WAITING)) {
        if (!nr_irqs)
            irq_found = i;
        nr_irqs++;
    }
    desc->status = status & ~IRQ_AUTODETECT;
    desc->handler->shutdown(i);
}
spin_unlock_irq(&desc->lock);
```

Interrupts – Hardware Interrupt Handlers (1)

1. The CPU jumps to the **IRQ0xNN_interrupt** routine (where *NN* is the interrupt number)
2. **common_interrupt** calls **do_IRQ** and sees to it that when **do_IRQ** returns, control will pass to **ret_from_intr**
3. **do_IRQ** calls code that is specific to the interrupt controller chip
4. **do_IRQ** calls **handle_IRQ_event**
 - **mask_and_ack_8259A()**
5. **do_IRQ** calls the controller-specific **end** function
 - **end_8259Airq()**
6. **do_IRQ** services any pending softirqs, and then returns

Interrupts – Hardware Interrupt Handlers (2)

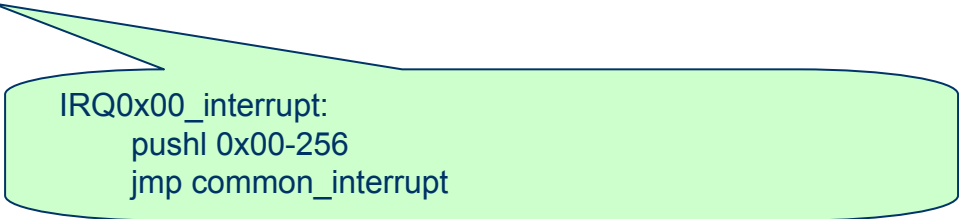
```
include/asm-i386/hw_irq.h
```

```
#define BUILD_IRQ(nr) \  
asmlinkage void IRQ_NAME(nr); \  
__asm__( \  
"\n"__ALIGN_STR"\n" \  
SYMBOL_NAME_STR(IRQ) #nr "_interrupt:\n\t" \  
"pushl $"#nr"-256\n\t" \  
"jmp common_interrupt");
```

```
arch/i386/kernel/i8259.c
```

```
#define BI(x,y) BUILD_IRQ(x##y) \  
#define BUILD_16_IRQS(x) \  
BI(x,0) BI(x,1) BI(x,2) BI(x,3) \  
BI(x,4) BI(x,5) BI(x,6) BI(x,7) \  
BI(x,8) BI(x,9) BI(x,a) BI(x,b) \  
BI(x,c) BI(x,d) BI(x,e) BI(x,f)
```

```
BUILD_16_IRQS(0x0)
```



```
IRQ0x00_interrupt: \  
pushl 0x00-256 \  
jmp common_interrupt
```

Interrupts – Hardware Interrupt Handlers (3)

```
include/asm-i386/hw_irq.h
```

```
#define BUILD_COMMON_IRQ() \  
asmlinkage void call_do_IRQ(void); \  
__asm__( \  
    "\n" __ALIGN_STR "\n" \  
    "common_interrupt:\n\t" \  
    SAVE_ALL \  
    SYMBOL_NAME_STR(call_do_IRQ) "\n\t" \  
    "call " SYMBOL_NAME_STR(do_IRQ) "\n\t" \  
    "jmp ret_from_intr\n");
```

Interrupts – Hardware Interrupt Handlers (4)

arch/i386/kernel/irq.c – do_IRQ()

```
asmlinkage unsigned int do_IRQ(struct pt_regs regs)
{
    int irq = regs.orig_eax & 0xff; /* high bits used in ret_from_code */
    int cpu = smp_processor_id();
    irq_desc_t *desc = irq_desc + irq;
    struct irqaction * action;
    unsigned int status;

    kstat.irqs[cpu][irq]++;
    spin_lock(&desc->lock);
    desc->handler->ack(irq);
    /*
     * REPLAY is when Linux resends an IRQ that was dropped earlier
     * WAITING is used by probe to mark irqs that are being tested
     */
    status = desc->status & ~(IRQ_REPLAY | IRQ_WAITING);
    status |= IRQ_PENDING; /* we _want_ to handle it */
    ...
}
```

Interrupts – Hardware Interrupt Handlers (4)

arch/i386/kernel/irq.c – do_IRQ()

```
...
/*
 * If the IRQ is disabled for whatever reason, we cannot
 * use the action we have.
 */
action = NULL;
if (!(status & (IRQ_DISABLED | IRQ_INPROGRESS))) {
    action = desc->action;
    status &= ~IRQ_PENDING; /* we commit to handling */
    status |= IRQ_INPROGRESS; /* we are handling it */
}
desc->status = status;

if (!action)
    goto out;
...
```

Interrupts – Hardware Interrupt Handlers (4)

arch/i386/kernel/irq.c – do_IRQ()

```
...
for (;;) {
    spin_unlock(&desc->lock);
    handle_IRQ_event(irq, &regs, action);
    spin_lock(&desc->lock);
    if (!(desc->status & IRQ_PENDING))
        break;
    desc->status &= ~IRQ_PENDING;
}
desc->status &= ~IRQ_INPROGRESS;
out:
desc->handler->end(irq);
spin_unlock(&desc->lock);

if (softirq_pending(cpu))
    do_softirq();
return 1;
}
```

Interrupts – Hardware Interrupt Handlers (4)

arch/i386/kernel/irq.c – handle_IRQ_event()

```
int handle_IRQ_event(unsigned int irq, struct pt_regs * regs, struct irqaction * action)
{
    int status;
    int cpu = smp_processor_id();

    irq_enter(cpu, irq);
    status = 1; /* Force the "do bottom halves" bit */
    if (!(action->flags & SA_INTERRUPT))
        __sti();
    do {
        ...
    } while (action);
    if (status & SA_SAMPLE_RANDOM)
        add_interrupt_randomness(irq);
    __cli();
    irq_exit(cpu, irq);
    return status;
}
```

status |= action->flags;
action->handler(irq, action->dev_id, regs);
action = action->next;

Interrupts – Softirqs, Tasklets, and Bottom Halves (1)

- Softirqs
 - A softirq is just a function pointer and an argument to pass to that function
 - For each softirq, there is a one-bit flag that indicates whether it should be run
 - The kernel polls the softirq flags regularly
 - Normally, once a softirq's flag has turned on, the kernel will invoke the softirq in a short time

Interrupts – Softirqs, Tasklets, and Bottom Halves (2)

```
include/linux/interrupt.h
```

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

```
include/asm-i386/hardirq.h
```

```
typedef struct {  
    unsigned int __softirq_pending;  
    unsigned int __local_irq_count;  
    unsigned int __local_bh_count;  
    unsigned int __syscall_count;  
    struct task_struct * __ksoftirqd_task; /* waitqueue is too large */  
    unsigned int __nmi_count; /* arch dependent */  
} ____cacheline_aligned irq_cpustat_t;
```

Interrupts – Softirqs, Tasklets, and Bottom Halves (3)

- Tasklets
 - A tasklet is little more than a glorified function pointer, not all that different from a softirq
 - A given tasklet can be scheduled for execution on at most one CPU at a time
 - Two softirqs, **HI_SOFTIRQ** and **TASKLET_SOFTIRQ**, are devoted to run tasklets
 - The **HI_SOFTIRQ** set is devoted to run bottom halves

Interrupts – Softirqs, Tasklets, and Bottom Halves (4)

```
include/linux/interrupt.h
```

```
struct tasklet_struct  
{  
    struct tasklet_struct *next;  
    unsigned long state;  
    atomic_t count;  
    void (*func)(unsigned long);  
    unsigned long data;  
};
```

Interrupts – Softirqs, Tasklets, and Bottom Halves (5)

- Bottom halves
 - A given interrupt handler is divided into a top half and a bottom half
 - The bottom half of an interrupt handler is the part that not need to be done right away
 - Why bottom halves?
 - Minimize overall interrupt latency
 - The interrupt controller chip is told to disable the specific IRQ be serviced while the kernel is executing the top half
 - Handler's bottom half consists of things that don't need to be done on ever single interrupt

Interrupts – Time (1): Initialization

init/main.c

```
asmlinkage void __init start_kernel(void)
{
    ...
    init_IRQ();
    sched_init();
    time_init();
    softirq_init();
    ...
}
```

arch/i386/kernel/i8259.c

```
void __init sched_init(void)
{
    ...
    outb_p(0x34, 0x43);           /* binary, mode 2, LSB/MSB, ch 0 */
    outb_p(LATCH & 0xff, 0x40); /* LSB */
    outb(LATCH >> 8, 0x40);     /* MSB */
    ...
}
```

kernel/sched.c

```
void __init sched_init(void)
{
    ...
    init_bh(TIMER_BH, timer_bh);
    init_bh(TQUEUE_BH, tqueue_bh);
    ...
}
```

Interrupts – Time (1): Initialization

init/main.c

```
asmlinkage void __init start_kernel(void)
{
    ...
    init_IRQ();
    sched_init();
    time_init();
    softirq_init();
    ...
}
```

kernel/softirq.c

```
static void (*bh_base[32])(void);
struct tasklet_struct bh_task_vec[32];

void init_bh(int nr, void (*routine)(void))
{
    bh_base[nr] = routine;
    mb();
}

void remove_bh(int nr)
{
    tasklet_kill(bh_task_vec+nr);
    bh_base[nr] = NULL;
}
```

Interrupts – Time (1): Initialization

init/main.c

```
asmlinkage void __init  
{  
    ...  
    init_IRQ();  
    sched_init();  
    time_init();  
    softirq_init();  
    ...  
}
```

arch/i386/kernel/time.c

```
static struct irqaction irq0 = { timer_interrupt, SA_INTERRUPT, 0, "timer", NULL, NULL};  
  
void __init time_init()  
{  
    ...  
    setup_irq(0, &irq0);  
}
```

kernel/softirq.c

```
void __init softirq_init()  
{  
    int i;  
  
    for (i=0; i<32; i++)  
        tasklet_init(bh_task_vec+i, bh_action, i);  
    open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);  
    open_softirq(HI_SOFTIRQ, tasklet_hi_action, NULL);  
}
```

Interrupts – Time (1): Initialization

init/main.c

```
asmlinkage void __init start_k  
{  
    ...  
    init_IRQ();  
    sched_init();  
    time_init();  
    softirq_init();  
    ...  
}
```

kernel/softirq.c

```
void tasklet_init(struct tasklet_struct *t,  
                 void (*func)(unsigned long), unsigned long data)  
{  
    t->next = NULL;  
    t->state = 0;  
    atomic_set(&t->count, 0);  
    t->func = func;  
    t->data = data;  
}
```

kernel/softirq.c

```
void open_softirq(int nr, void (*action)(struct softirq_action*), void *data)  
{  
    softirq_vec[nr].data = data;  
    softirq_vec[nr].action = action;  
}
```

Interrupts – Time (2): Timer Interrupt

arch/i386/kernel/time.c

```
static void timer_interrupt(int irq, void *dev_id, struct pt_regs *regs)
```

```
{
```

```
    ...
```

```
    do_timer_interrupt(irq, NULL, regs);
```

```
    ...
```

```
}
```

```
static inline void do_timer_interrupt(int irq, void *dev_id, struct pt_regs *regs)
```

```
{
```

```
    ...
```

```
    do_timer(regs);
```

```
    ...
```

```
}
```

Interrupts – Time (3): Timer Interrupt

kernel/timer.c

```
void do_timer(struct pt_regs *regs)
{
    (*(unsigned long *)&jiffies)++;
#ifdef CONFIG_SMP
    /* SMP process accounting uses the local APIC timer */
    update_process_times(user_mode);
#endif
    mark_bh(TIMER_BH);
    if (TQ_ACTIVE(tq_timer))
        mark_bh(TQUEUE_BH);
}
```

include/linux/interrupt.h

```
static inline void mark_bh(int nr)
{
    tasklet_hi_schedule(bh_task_vec+nr);
}
```

include/linux/interrupt.h

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

Interrupts – Time (3): Timer Interrupt

kernel/timer.c

```
void do_timer(struct pt_regs *regs)
{
    (*(unsigned long *)&jiffies)++;
#ifdef CONFIG_SMP
    /* SMP process accounting uses t

    update_process_times(user_mode

#endif
    mark_bh(TIMER_BH);
    if (TQ_ACTIVE(tq_timer))
        mark_bh(TQUEUE_BH);
}
```

kernel/softirq.c

```
void __tasklet_hi_schedule(struct tasklet_struct *t)
{
    int cpu = smp_processor_id();
    unsigned long flags;

    local_irq_save(flags);
    t->next = tasklet_hi_vec[cpu].list;
    tasklet_hi_vec[cpu].list = t;
    cpu_raise_softirq(cpu, HI_SOFTIRQ);
    local_irq_restore(flags);
}
```

Interrupts – Time (4): Do Softirq

arch/i386/kernel/irq.c

```
asmlinkage unsigned int do_IRQ(struct pt_regs regs)
{
    ...
    if (softirq_pending(cpu))
        do_softirq();
    return 1;
}
```

Interrupts – Time (5): Do Softirq

kernel/softirq.c – do_softirq()

```
asmlinkage void do_softirq()
{
    int cpu = smp_processor_id();
    __u32 pending;
    long flags;
    __u32 mask;

    if (in_interrupt())
        return;
    local_irq_save(flags);

    pending = softirq_pending(cpu);
    if (pending) {
        struct softirq_action *h;

        mask = ~pending;
        local_bh_disable();
        ...
    }
}
```

Interrupts – Time (5): Do Softirq

kernel/softirq.c – do_softirq()

```
...
restart:
    /* Reset the pending bitmask before enabling irqs */
    softirq_pending(cpu) = 0;
    local_irq_enable();

    h = softirq_vec;
    do {
        if (pending & 1)
            h->action(h);
        h++;
        pending >>= 1;
    } while (pending);
    ...
```

Interrupts – Time (5): Do Softirq

kernel/softirq.c – do_softirq()

```
...
local_irq_disable();

pending = softirq_pending(cpu);
if (pending & mask) {
    mask &= ~pending;
    goto restart;
}
__local_bh_enable();

if (pending)
    wakeup_softirqd(cpu);
}

local_irq_restore(flags);
}
```

Interrupts – Time (6): Tasklet Action

kernel/softirq.c – tasklet_hi_action()

```
static void tasklet_hi_action(struct softirq_action *a)
{
    int cpu = smp_processor_id();
    struct tasklet_struct *list;

    local_irq_disable();
    list = tasklet_hi_vec[cpu].list;
    tasklet_hi_vec[cpu].list = NULL;
    local_irq_enable();

    while (list) {
        struct tasklet_struct *t = list;

        list = list->next;
        ...
    }
}
```

Interrupts – Time (6): Tasklet Action

kernel/softirq.c – tasklet_hi_action()

```
...
if (tasklet_trylock(t)) {
    if (!atomic_read(&t->count)) {
        if (!test_and_clear_bit(TASKLET_STATE_SCHED, &t->state))
            BUG();
        t->func(t->data);
        tasklet_unlock(t);
        continue;
    }
    tasklet_unlock(t);
}
local_irq_disable();
t->next = tasklet_hi_vec[cpu].list;
tasklet_hi_vec[cpu].list = t;
__cpu_raise_softirq(cpu, HI_SOFTIRQ);
local_irq_enable();
}
}
```

Interrupts – Time (7): Bottom Halves

kernel/softirq.c

```
static void bh_action(unsigned long nr)
{
    int cpu = smp_processor_id();

    if (!spin_trylock(&global_bh_lock))
        goto resched;
    if (!hardirq_trylock(cpu))
        goto resched_unlock;
    if (bh_base[nr])
        bh_base[nr]();
    hardirq_endlock(cpu);
    spin_unlock(&global_bh_lock);
    return;
resched_unlock:
    spin_unlock(&global_bh_lock);
resched:
    mark_bh(nr);
}
```

Interrupts – Time (8): Bottom Halves

kernel/timer.c

```
void timer_bh(void)
{
    update_times();
    run_timer_list();
}
```

kernel/timer.c

```
static inline void update_times(void)
{
    unsigned long ticks;

    /*
     * update_times() is run from the raw timer_bh handler so we
     * just know that the irqs are locally enabled and so we don't
     * need to save/restore the flags of the local CPU here. -arca
     */
    write_lock_irq(&xtime_lock);

    ticks = jiffies - wall_jiffies;
    if (ticks) {
        wall_jiffies += ticks;
        update_wall_time(ticks);
    }
    write_unlock_irq(&xtime_lock);
    calc_load(ticks);
}
```

Interrupts

kernel/timer.c

```
void timer_bh(void)
{
    update_times();
    run_timer_list();
}
```

kernel/timer.c

```
static inline void run_timer_list(void)
{
    spin_lock_irq(&timerlist_lock);
    while ((long)(jiffies - timer_jiffies) >= 0) {
        struct list_head *head, *curr;
        if (!tv1.index) {
            int n = 1;
            do {
                cascade_timers(tvecs[n]);
            } while (tvecs[n]->index == 1 && ++n < NOOF_TVECS);
        }
        repeat:
        head = tv1.vec + tv1.index;
        curr = head->next;
        if (curr != head) {
            struct timer_list *timer;
            void (*fn)(unsigned long);
            unsigned long data;

            timer = list_entry(curr, struct timer_list, list);
            fn = timer->function;
            data = timer->data;

            detach_timer(timer);
            timer->list.next = timer->list.prev = NULL;
            timer_enter(timer);
            spin_unlock_irq(&timerlist_lock);
            fn(data);
            spin_lock_irq(&timerlist_lock);
            timer_exit();
            goto repeat;
        }
        ++timer_jiffies;
        tv1.index = (tv1.index + 1) & TVR_MASK;
    }
    spin_unlock_irq(&timerlist_lock);
}
```

References

- Linux Core Kernel Commentary, 2nd Edition
- The 80x86 IBM PC & Compatible Computers, Volume II: Design and Interfacing of the IBM PC, PS and Compatibles