Linux Kernel
Interrupt Handling

Paul Chu
Hao-Ran Liu
What is interrupt
- A communication mechanism for hardware components to notify CPU of events. E.g. key strokes and timers.
- There may be one or more interrupt request lines (IRQ), which is a physical input to the interrupt controller chip. The number of such inputs is limited. (eg. Classic PC has only 15 IRQ lines)
- Each IRQ has a unique number, which may be used by one or more components.

Basic flow of interrupt handling
- When receiving an interrupt, CPU program counter jumps to a pre-defined address (interrupt vectors)
- The state of interrupted program is saved
- The corresponding service routine is executed
- The interrupting component is served, and interrupt signal is removed
- The state of interrupted program is restored
- Resume the interrupted program at the interrupted address
Interrupts and exceptions are handled by the kernel in a similar way.

- **Interrupts**
  - Asynchronous events generated by external hardware,
  - Interrupt controller chip maps each IRQ input to an interrupt vector, which locates the corresponding interrupt service routine

- **Exceptions (Trap)**
  - Synchronous events generated by the software
  - E.g. divide by zero, page faults
## Interrupt vectors on x86

<table>
<thead>
<tr>
<th>Vector range</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-19 (0x0-0x13)</td>
<td>Nonmaskable interrupts and exceptions</td>
</tr>
<tr>
<td>20-31 (0x14-0x1f)</td>
<td>Intel-reserved</td>
</tr>
<tr>
<td>32-127 (0x20-0x7f)</td>
<td>External interrupts (IRQs)</td>
</tr>
<tr>
<td>128 (0x80)</td>
<td>Programmed exception for system calls</td>
</tr>
<tr>
<td>129-238 (0x81-0xee)</td>
<td>External interrupts (IRQs)</td>
</tr>
<tr>
<td>239 (0xef)</td>
<td>Local APIC timer interrupt</td>
</tr>
<tr>
<td>240-250 (0xf0-0xfa)</td>
<td>Reserved by Linux for future use</td>
</tr>
<tr>
<td>251-255 (0xfb-0xff)</td>
<td>Interprocessor interrupts</td>
</tr>
</tbody>
</table>

The table is from Understanding the Linux kernel, 2nd edition
Interrupt handling in x86 Linux

**Hardware**

- CPU
  - IDTR register
- PIC
  - IRQ x
  - IRQ y
  - INT

**Software**

- idt_table
  - setup by trap_init(), init_IRQ()
- divide_error
- page_fault
- interrupt[]
- system_call
- pushl $vector-256
- common_interrupt:
  - SAVE_ALL
  - mov %esp, %eax
  - call do_IRQ
  - jmp ret_from_intr
- do_IRQ
- handle_IRQ_event
- shared IRQ handler 1
- shared IRQ handler 2
- ...
Each IRQ line is associated with an IRQ descriptor

typedef struct irq_desc {
    unsigned int status;       /* IRQ status: in progress, disabled, ... */
    hw_irq_controller *handler; /* ack, end, enable, disable irq on PIC */
    struct irqaction *action;   /* IRQ action list */
    unsigned int depth;         /* nested irq disables */
    spinlock_t lock;            /* serialize access to this structure */
} __attribute__((cachelineAligned));

irq_desc_t irq_desc[NR_IRQS] __attribute__((cachelineAligned)) = {
    [0 ... NR_IRQS-1] = {
        .handler = &no_irq_type,
        .lock = SPIN_LOCK_UNLOCKED
    }
};
The status field of the IRQ descriptor is used to indicate various conditions that can occur on an IRQ line. The table below lists some of these conditions with their descriptions:

<table>
<thead>
<tr>
<th>Flag name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ_INPROGRESS</td>
<td>A handler for the IRQ is being executed.</td>
</tr>
<tr>
<td>IRQ_DISABLED</td>
<td>The IRQ line has been deliberately disabled by a device driver.</td>
</tr>
<tr>
<td>IRQ_PENDING</td>
<td>An IRQ has occurred on the line; its occurrence has been acknowledged to the PIC, but it has not yet been serviced by the kernel.</td>
</tr>
<tr>
<td>IRQ_AUTODETECT</td>
<td>The kernel uses the IRQ line while performing a hardware device probe.</td>
</tr>
<tr>
<td>IRQ_WAITING</td>
<td>The kernel uses the IRQ line while performing a hardware device probe; moreover, the corresponding interrupt has not been raised.</td>
</tr>
</tbody>
</table>

The table is from Understanding the Linux kernel, 2nd edition.
This describes operations of a interrupt controller

```c
struct hw_interrupt_type {
    /* the name of the PIC, shown in /proc/interrupts */
    const char * typename;
    /* called at first time reg. of the irq */
    unsigned int (*startup)(unsigned int irq);
    /* called when all handlers on the irq unreg’ed */
    void (*shutdown)(unsigned int irq);

    void (*enable)(unsigned int irq); /* enable the specified IRQ */
    void (*disable)(unsigned int irq); /* disable the specified IRQ */
    void (*ack)(unsigned int irq); /* ack. (may disable) the received IRQ */
    void (*end)(unsigned int irq); /* called at termination of IRQ handler */
    void (*set_affinity)(unsigned int irq, cpumask_t dest);
};
```
i8259A is the classic interrupt controller on x86

```c
static struct hw_interrupt_type i8259A_irq_type = {
    "XT-PIC",
    startup_8259A_irq,
    shutdown_8259A_irq,
    enable_8259A_irq,
    disable_8259A_irq,
    mask_and_ack_8259A,
    end_8259A_irq,
    NULL
};
```

- mask_and_ack_8259A() acknowledges the interrupt on the PIC and also disables the IRQ line
- end_8259A_irq() re-enables the IRQ line
Multiple devices can share a single IRQ; each irqaction refers to a specific hardware device and its interrupt handler

```c
struct irqaction {
    /* Points to the interrupt service routine for an I/O device */
    irqreturn_t (*handler)(int, void *, struct pt_regs *);
    /* Describes the relationships between the IRQ line and the I/O device */
    unsigned long flags;
    cpumask_t mask;
    /* the name of the device, shown in /proc/interrupts */
    const char *name;
    /* a private field for the device driver */
    void *dev_id;
    /* points to next irqaction which shared the same IRQ line */
    struct irqaction *next;
    int irq; /* IRQ number */
    struct proc_dir_entry *dir;
};
```
Registering Interrupt Handler

- Requesting to be invoked when a specific IRQ is signaled

```c
int request_irq( unsigned int irq, irq_handler_t *handler, long irqflags, const char* devname, void *dev_id)
```

- **irqflags**
  - SA_INTERRUPT: This is a fast interrupt. All local IRQs are disabled during handler execution
  - SA_SAMPLE_RANDOM: The timing of interrupts from this device are fed to kernel entropy pool. This is for kernel random number generator
  - SA_SHIRQ: the IRQ line can be shared among multiple devices

- **devname**: the name of the device used by /proc/interrupts

- **dev_id**
  - The unique identifier of a handler for a shared IRQ
  - The argument passed to the registered handler (E.g. private structure or device number of the device driver)
  - Can be NULL only if the IRQ is not shared
Unregistering interrupt handler

- Unregister a specified interrupt handler and disable the given IRQ line if this is the last handler on the line.

```c
int free_irq( unsigned int irq, void *dev_id)
```

- If the specified IRQ is shared, the handler identified by the dev_id is unregistered.
Probing Interrupt Line

⌵ Problem to solve
  – Fail to register interrupt handler because of not knowing which interrupt line the device has been assigned to
  – Rarely to use on embedded systems or for PCI devices

’int’’ Probing procedure
  – Clear and/or mask the device internal interrupt
  – Enable CPU interrupt
  – mask = probe_irq_on()
    • return a bit mask of unallocated interrupts
  – Enable device’s interrupt and make it to trigger an interrupt
  – Busy waiting for a while allowing the expected interrupt to be signaled
  – irqs = probe_irq_off(mask)
    • Returns the number of the IRQ that was signaled
    • If no interrupt occurred, 0 is returned; if more than 1 interrupt occurred, a negative value is returned
  – Service the device and clear pending interrupt
Writing an Interrupt Handler

- **Handler prototype**

```c
int irqreturn_t handler(int irq, void *dev_id, struct pt_regs *regs);
```

- dev_id: the dev_id you register at request_irq()
- pt_regs: value of registers before being interrupted

- **Return value**
  - IRQ_NONE: the handler cannot handle it; the originator may be other devices sharing the same IRQ line
  - IRQ_HANDLED: the interrupt is serviced by the handler
  - IRQ_RETVAL(x): if x is nonzero, return IRQ_HANDLED; otherwise, return IRQ_NONE

- **Interrupt handler is not reentrant; while it is executing:**
  - its IRQ line is disabled on PIC
  - IRQ_INPROGRESS flag prevents other CPU from executing it
To share an IRQ with other device, you must
  – register_irq() with SA_SHIRQ flag
    • The registration fails if other handler already register the same IRQ without SA_SHIRQ flag
  – The dev_id argument must be unique to each handler
  – The interrupt handler must be able to find out whether its device actually generate an interrupt
    • Hardware must provide a status register for inquiry
static ata_index_t do_ide_setup_pci_device (struct pci_dev *dev, ...) {
    hwif->irq = dev->irq;
}

#define ide_request_irq(irq,hand,flg,dev,id) \
    request_irq((irq),(hand),(flg),(dev),(id))
static static int init_irq (ide_hwif_t *hwif) {
    int sa = IDE_CHIPSET_IS_PCI(hwif->chipset)?SA_SHIRQ:SA_INTERRUPT;
    ide_request_irq(hwif->irq, &ide_intr, sa, hwif->name, hwgroup);
}

irqreturn_t ide_intr (int irq, void *dev_id, struct pt_regs *regs) {
    ide_hwgroup_t *hwgroup = (ide_hwgroup_t *)dev_id;
    ide_drive_t *drive = choose_drive(hwgroup);
    struct request *rq;

    rq = elv_next_request(drive->queue);
    start_request(drive, rq);
    return IRQ_HANDLED;
}
Interrupt Context

- **Context**
  - The execution environments of a piece of code

- **Process context**
  - Kernel is executing on behalf of a process. E.g. executing a system call.
  - Because of process management mechanisms, code in process context can sleep or be blocked

- **Interrupt context**
  - Time critical; it must finish its job quickly because it may interrupts some real-time job (may be a process or another interrupt handler)
  - No backing process; interrupted process context cannot be used
  - Code in interrupt context cannot sleep or be blocked (i.e. you cannot call some kernel functions that may sleep)
  - Configurable stack: dedicated interrupt stack (4K) or sharing the kernel stack of interrupted process (<8K)
  - Both interrupt handlers and bottom halves (softirq, tasklet) run in interrupt context
Interrupt context is not preemptive; preemption are disabled by increasing preempt_count

Process softirq only when are not in interrupt context
  – Nested execution of interrupt handlers is possible

```
#define HARDIRQ_OFFSET (1UL << HARDIRQ_SHIFT)
#define IRQ_EXIT_OFFSET (HARDIRQ_OFFSET-1)
#define irq_enter() (preempt_count() += HARDIRQ_OFFSET)
void irq_exit(void) {
    preempt_count() -= IRQ_EXIT_OFFSET;
    if (!in_interrupt() && local_softirq_pending()) do_softirq();
    preempt_enable_no_resched();
}
fastcall unsigned int do_IRQ(struct pt_regs *regs) {
    int irq = regs->orig_eax & 0xff;
    irq_enter();
    __do_IRQ(irq, regs);
    irq_exit();
    return 1;
}
```
Implementation of Interrupt Handling

--- __do_IRQ() ---

**IRQ Probing**
- `probe_irq_on()` set IRQ_WAITING for all unallocated IRQs
- IRQ_WAITING flag is cleared when interrupt signals
- `probe_irq_off()` checks this flag to find the IRQ number of expected interrupt

```c
fastcall unsigned int __do_IRQ(unsigned int irq, struct pt_regs **regs)
{
    irq_desc_t *desc = irq_desc + irq;
    struct irqaction *action;
    unsigned int status;

    /* avoid concurrent execution of the same IRQ */
    spin_lock(&desc->lock);
    desc->handler->ack(irq); /* disable IRQ at PIC */
    status = desc->status & ~IRQ_WAITING;
    status |= IRQ_PENDING; /* we _want_ to handle it */
}
```
Implementation of Interrupt Handling

-- __do_IRQ()

- IRQ_INPROGRESS flag prevents handlers of the same IRQ from concurrent execution

```c
action = NULL;
if (likely(!((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 there is no IRQ handler or it was disabled, exit early.
 * Since we set PENDING, if another processor is handling
 * a different instance of this same irq, the other processor
 * will take care of it.
 */
if (unlikely(!action)) goto out;
```
Implementation of Interrupt Handling

--- do_IRQ()

> Take care of other CPUs’ interrupt by checking IRQ_PENDING flag

```c
for (;;) {
    irqreturn_t action_ret;
    spin_unlock(&desc->lock);
    action_ret = handle_IRQ_event(irq, regs, action);
    spin_lock(&desc->lock);
    if (likely(!(desc->status & IRQ_PENDING)))
        break;
    desc->status &= ~IRQ_PENDING;
}

out:
    desc->status &= ~IRQ_INPROGRESS;
    desc->handler->end(irq); /* enable IRQ at PIC */
    spin_unlock(&desc->lock);
    return 1;
```
Invoke all registered handlers of the IRQ line since kernel do not know the origin of the signaled interrupt

```c
fastcall int handle_IRQ_event(unsigned int irq, struct pt_regs *regs, struct irqaction *action) {
    int ret, retval = 0, status = 0;

    if (!(action->flags & SA_INTERRUPT))
        local_irq_enable(); /* fast interrupt */
    do {
        ret = action->handler(irq, action->dev_id, regs);
        if (ret == IRQ_HANDLED)
            status |= action->flags;
        retval |= ret;
        action = action->next;
    } while (action);
    if (status & SA_SAMPLE_RANDOM)
        add_interrupt_randomness(irq);

    local_irq_disable();
    return retval;
}
```
Before returning to the interrupted context, call schedule() for a reschedule when:

- The kernel is returning to user space and need_resched() is true
- The kernel is returning to kernel space and preempt_count() is zero

The value of registers are restored and the kernel resumes whatever was interrupted
References

- Understanding the Linux Kernel, Bovet & Cesati, O’REILLY, 2002
- Linux 2.6.10 kernel source