Interrupts and Drivers

Youjip Won

KAIST EE
Contents

- System calls, exceptions, and interrupts
- X86 protection
- Code: The first system call
- Code: Assembly trap handlers
- Code: C trap handler
- Code: System calls
- Code: Interrupts
- Drivers
- Code: Disk driver
IRQ (Interrupt ReQuest) Line

Diagram showing the flow of an interrupt request through the INTERrupt Controller/Adapter (PIC) to the CPU. The diagram includes:

- Device
- IRQ lines
- PIC
- CPU
- Interrupt message (interrupt vector)

The diagram illustrates how an interrupt request from a device is processed through the PIC to the CPU.
IRQ Line

**Interrupt ReQuest (IRQ) Line**

- a single output line to raise an interrupt
- All existing IRQ lines are connected to the input pins of a hardware circuit called the *Programmable Interrupt Controller*

<table>
<thead>
<tr>
<th>IRQ</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>system timer (cannot be changed)</td>
</tr>
<tr>
<td>1</td>
<td>keyboard controller (cannot be changed)</td>
</tr>
<tr>
<td>2</td>
<td>cascaded signals from IRGs 8–15</td>
</tr>
<tr>
<td>3</td>
<td>second RS-232 serial port (COM2: in Windows)</td>
</tr>
<tr>
<td>4</td>
<td>first RS-232 serial port (COM1: in Windows)</td>
</tr>
<tr>
<td>5</td>
<td>parallel port 2 and 3 or sound card</td>
</tr>
<tr>
<td>6</td>
<td>floppy disk controller</td>
</tr>
<tr>
<td>7</td>
<td>first parallel port</td>
</tr>
<tr>
<td>8</td>
<td>real-time clock</td>
</tr>
<tr>
<td>9</td>
<td>open interrupt</td>
</tr>
<tr>
<td>10</td>
<td>open interrupt</td>
</tr>
<tr>
<td>11</td>
<td>open interrupt</td>
</tr>
<tr>
<td>12</td>
<td>PS/2 mouse</td>
</tr>
<tr>
<td>13</td>
<td>math coprocessor</td>
</tr>
<tr>
<td>14</td>
<td>primary ATA channel</td>
</tr>
<tr>
<td>15</td>
<td>secondary ATA channel</td>
</tr>
</tbody>
</table>
Programmable Interrupt Controller

1. Monitors the IRQ lines, checking for raised signals. If two or more IRQ lines are raised, selects the one having the lower pin number.

2. If a raised signal occurs on an IRQ line:
   1. Converts the raised signal received into a corresponding vector.
   2. Stores the vector in an Interrupt Controller I/O port, thus allowing the CPU to read it via the data bus.
   3. Sends a raised signal to the processor INTR pin—that is, issues an interrupt.
   4. Waits until the CPU acknowledges the interrupt signal by writing into one of the Programmable Interrupt Controllers (PIC) I/O ports; when this occurs, clears the INTR line.

3. Goes back to step 1.
The mapping between IRQs and vectors can be modified by software. IRQ n \(\rightarrow\) Interrupt vector n + 32.

The PIC can be told to stop issuing interrupts that refer to a given IRQ line, or to resume issuing them. Disabled interrupts are not lost; the PIC sends them to the CPU as soon as they are enabled again.

Selective enabling/disabling of IRQs is not the same as global masking/unmasking of maskable interrupts. When the IF flag of the eflags register is clear, each maskable interrupt issued by the PIC is temporarily ignored by the CPU.
Interrupt descriptor table

Gate descriptor (interrupt vector)

System call

interrupt

trap

Interrupt descriptor table

Interrupt disabled
Interrupt

- interrupt: to tell the kernel about the hardware event
  - I/O completion
  - can be generated at any time.
  - System call can be generated only when it is called.

Delivering interrupt

- PIC (Programmable Interrupt Controller): old motherboards
- IO APIC (in IO device, ioapic.c) + local APIC (attached to CPU, lapic.c):

Control interrupt

- Disable interrupts for a certain fragment of code.
- **cli** (clear interrupt): disable interrupt
- **sti** (set interrupt): enable interrupt on a processor.
bootasm.S

- Initialize registers.

```
start:
cli       # BIOS enabled interrupts; disable

# Zero-fill data segment registers DS, ES, and SS.
xorw     %ax,%ax       # Set %ax to zero
movw     %ax,%ds       # -> Data Segment
movw     %ax,%es       # -> Extra Segment
movw     %ax,%ss       # -> Stack Segment
```
IOAPIC and LAPIC

- PIC (Programmable Interrupt Controller) from archaic Intel 8259 processor.
- APIC (Advanced Programmable Interrupt Controller) from 82489DX (80486 and early Pentium)
  - Split architecture of LAPIC and IO APIC
  - LAPIC (Local APIC)
    - integrated into the CPU itself
    - Handles all external interrupts
  - IO APIC
    - Integrated into system bus
    - Contains a redirection table to route the interrupt it receives from peripheral buses to one or more local APICs.
    - CPU can program the entries in the table through memory mapped IO (MMIO).
IOAPIC and LAPIC

CPU
Local APIC
Interrupt messages
IPI’s
Interrupt messages
IPI’s
Interrupt messages
IPI’s
Interrupt messages
IPI’s
Interrupt messages
IPI’s

IO APIC

IPI’s

Disk
Network Card
User device

External interrupts
Drivers

- The code in an OS that manages a device.
  - Tells the device to do something.
  - Configure the device.
  - Handle the interrupt from the device.

- Device driver for disk
  - Copies data to and from the disk.
  - Unit of transfer: 512 byte (sector)
  - Host side data structure for sector: `struct buf`
3850 struct buf {
3851   int flags; // DIRTY bit and VALID bit
3852   uint dev;
3853   uint blockno;
3854   struct sleeplock lock;
3855   uint refcnt;
3856   struct buf *prev; // LRU cache list
3857   struct buf *next;
3858   struct buf *qnext; // disk queue
3859   uchar data[BSIZE];
3860   };
3861 #define B_VALID 0x2
   // buffer has been read from disk
3862 #define B_DIRTY 0x4
   // buffer needs to be written to disk
Structure of IO APIC MMIO

Read and write data.

volatile struct ioapic *ioapic;

// IO APIC MMIO structure: write reg, then read or write data.
struct ioapic {
    uint reg;
    uint pad[3];
    uint data;
};
Program an interrupt handler: REG_TABLE entry

Program REG_TABLE entry.

```c
static void ioapicwrite(int reg, uint data)
{
    ioapic->reg = reg;
    ioapic->data = data;
}
```
void ioapicinit(void) {
    int i, id, maxintr;

    ioapic = (volatile struct ioapic*)IOAPIC; // IO device to memory
    maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
    id = ioapicread(REG_ID) >> 24;
    if(id != ioapicid)
        cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");

    // Mark all interrupts edge-triggered, active high, disabled,
    // and not routed to any CPUs.
    for(i = 0; i <= maxintr; i++){
        ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
        ioapicwrite(REG_TABLE+2*i+1, 0);
    }
}
Program an REG_TABLE to route \texttt{irq} to CPU \texttt{cpunum}.

```c
void ioapicenable(int irq, int cpunum)
{
    // Mark interrupt edge-triggered, active high, enabled, and routed to the given cpunum, which happens to be that cpu's APIC ID.
    ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
    ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
}
```
ideinit(): Initialize the device driver.

```c
void ideinit(void) {
    int i;

    initlock(&idelock, "ide");
    ioapicenable(IRQ_IDE, ncpu - 1);
    idewait(0);

    // Check if disk 1 is present
    outb(0x1f6, 0xe0 | (1<<4));
    for(i=0; i<1000; i++) {
        if(inb(0x1f7) != 0) {
            havedisk1 = 1;
            break;
        }
    }

    // Switch back to disk 0.
    outb(0x1f6, 0xe0 | (0<<4));
}
```

- IO is routed to CPU 1, the highest numbered CPU
- Wait till device is ready: busy bit is clear and ready bit is set.
- Check if disk 1 is present: write to port 0x1F6 to select disk and wait for some time.
Handling Disk I/O

1. ide-rw
2. ide-start
3. ide-infr

1. ide-rw
   - Prepare command
     - struct buf
       - Where, what, how many
       - Read vs. Write
   - Add it to command list.

Host → Device

(2) ide-start()

(3) ide-infr

Interrupt/Trap

Read

Youjip Won
IDE Device Driver

- Four registers
  - Control register (8 bit)
  - Command block registers
  - Status register (8 bit)
  - Error register (8 bit)
The IDE Interface

- Control Register (1 Byte):
  Address 0x3F6 = 0x08 (0000 1RE0): R=reset, E="Enable Interrupt", E=0 means "enable interrupt"

- Command Block Registers:
  Address 0x1F0 = Data Port (128 Byte)
  Address 0x1F1 = Error (1 Byte)
  Address 0x1F2 = Sector Count (1 Byte)
  Address 0x1F3 = LBA low byte (1 Byte)
  Address 0x1F4 = LBA mid byte (1 Byte)
  Address 0x1F5 = LBA hi byte (1 Byte)
  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
  Address 0x1F7 = Command/status (1 Byte)

- Status Register (Address 0x1F7): (1 Byte)
  7  6  5  4  3  2  1  0
  BUSY  READY  FAULT  SEEK  DRQ  CORR  IDDEX  ERROR
The IDE Interface (Cont.)

- Error Register (Address 0x1F1): (check when Status ERROR==1)

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BBK</td>
<td>UNC</td>
<td>MC</td>
<td>IDNF</td>
<td>MCR</td>
<td>ABRT</td>
<td>T0NF</td>
<td>AMNF</td>
</tr>
</tbody>
</table>
```

- BBK  = Bad Block,
- UNC = Uncorrectable data error,
- MC  = Media Changed,
- IDNF = ID mark Not Found,
- MCR = Media Change Requested,
- ABRT = Command aborted,
- T0NF = Track 0 Not Found,
- AMNF = Address Mark Not Found
The basic protocol of IDE device driver

- Wait for the drive to be ready.
- Write parameters to command registers.
  - Sector count
  - logical block address (LBA) of sector
  - Driver number
- Issue read/write to command register.
  - Read or Write
  - For write command, transfer data
- Handle interrupt.
- Handle Error.
void iderw(struct buf *b) {
    if (!holdingsleep(&b->lock))
        panic("iderw: buf not locked");
    if ((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
        panic("iderw: nothing to do");
    if (b->dev != 0 && havedisk1)
        panic("iderw: ide disk 1 not present");
    ...
}

1. The buffer should have been locked (b->lock).
2. In case of write: B_DIRTY should be set. In case of read: BVALID should be not set.
3. Check if the device exists.
IDE device driver in xv6: Big picture for mechanism

```c
void iderw(struct buf *b) {
...

acquire(&idelock);  //DOC:acquire-lock
b->qnext = 0;
for(pp=&idequeue; *pp; pp=&(*pp)->qnext) {
    *pp = b;
if (idequeue == b)
    idestart(b);
while((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
    sleep(b, &idelock); // IMPORTANT !!!
release(&idelock);
}
```

**Mechanism: add the buffer to the ide_queue and perform IO.**

1. Enqueue the buffer to the ide_queue.
2. If the buffer is the only entry in the ide_queue, start the IO right away.
3. Wait for the completion of the IO.
   1. If the IO is READ, interrupt handler resets the B_VALID flag.
   2. If the IO is write, interrupt handler resets the B_DIRTY flag.
Device driver

```c
static void idestart(struct buf *b) {
    idewait(0);
    outb(0x3f6, 0); // generate interrupt
    outb(0x1f2, 1); // the count of sectors
    outb(0x1f3, b->sector & 0xff);
    outb(0x1f4, (b->sector >> 8) & 0xff);
    outb(0x1f5, (b->sector >> 16) & 0xff);
    outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
    if (b->flags & B_DIRTY){
        outb(0x1f7, IDE_CMD_WRITE); // WRITE
        outs1(0x1f0, b->data, 512/4); // transfer data
    } else {
        outb(0x1f7, IDE_CMD_READ); // READ
    }
}
```
Device driver

```c
static void idestart(struct buf *b) {
    idewait(0);
    ...
}
```

```c
static int idewait(int checkerr) {
    int r;
    while (((r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY));
    if (checker && (r & (IDE_DF|IDE_ERR)) != 0) return -1;
    return 0;
}
```

1. Read Status Register(0x1F7) and save it to r.
2. Check if the drive is the busy. (IDE_BSY).
3. Check if the drive is ready.
4. If the device is not busy and ready, get out of the while loop.

# Status Register is also used as a command register. The device uses it to store the status of the register. The host uses it to store the command.
Enable interrupt and set the sector count.

```c
static void idestart(struct buf *b) {
    idewait(0);
    outb(0x3f6, 0); // generate interrupt
    outb(0x1f2, 1); // the number of sectors
...
```

1. Enable interrupt for the device: set the Control Register(0x3F6).
2. Set the number of sectors for an IO: set the sector count register(0x1F2).
Specifying the LBA

```c
static void idestart(struct buf *b) {
    idewait(0);
    outb(0x3f6, 0); // generate interrupt
    outb(0x1f2, 1); // the count of sectors
    outb(0x1f3, b->sector & 0xff);       // 1
    outb(0x1f4, (b->sector >> 8) & 0xff); // 2
    outb(0x1f5, (b->sector >> 16) & 0xff); // 3
    outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f)); // 4
}
```

1. LBA is 28 bit. (example: 512 Byte * 2^28 disk size == 16 GByte)
Write Command and the data transfer

```c
static void idestart(struct buf *b) {
    ...
    outb(0x1f3, b->sector & 0xff);
    outb(0x1f4, (b->sector >> 8) & 0xff);
    outb(0x1f5, (b->sector >> 16) & 0xff);
    outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
    if (b->flags & B_DIRTY){
        outb(0x1f7, IDE_CMD_WRITE); // WRITE
        outsl(0x1f0, b->data, 512/4); // transfer data
    }
}
```

1. If the DIRTY bit is set, perform write. (In xv6, device driver performs write only when the buffer is dirty.)
2. Set the write command at the Command Register (0x1F7).
3. Write the data to the Data Register (0x1F0).
   1. The amount of data to write: 512 Byte
   2. `static inline void outsl (int port, const void *addr, int cnt)`
Read command

static void idestart(struct buf *b) {
    ...
    outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
    if (b->flags & B_DIRTY){
        outb(0x1f7, IDE_CMD_WRITE); // WRITE
        outsl(0x1f0, b->data, 512/4); // transfer data
    } else {
        outb(0x1f7, IDE_CMD_READ); // READ
    }
}

1. If the buffer is not dirty, perform read.
2. Set the command READ at Command Register (0x1F7).
3. Perform read.
void ideintr() {

    struct buf *b;
    acquire(&ide_lock);

    if (!((b->flags & B_DIRTY) && ide_wait_ready() >= 0)) {
        insl(0x1f0, b->data, 512/4); // if READ: get data

        b->flags |=- B_VALID;
        b->flags &= B_DIRTY;
        wakeup(b); // wake up the waiting process

        if (idequeue != 0) // start next request
            idestart(ide_queue); // (if one exists)
    }

    release(&ide_lock);
}

Read

Set the flag properly. →wake up from iderw()

Start the next request in the idequeue.
4272 // Start the request for b. Caller must hold idelock.
4273 static void
4274 idestart(struct buf *b)
4275 {
4276   if(b == 0)
4277     panic("idestart");
4278   if(b->blockno >= FSSIZE)
4279     panic("incorrect blockno");
4280   int sector_per_block = BSIZE/SECTOR_SIZE;
4281   int sector = b->blockno * sector_per_block;
4282   int read_cmd = (sector_per_block == 1) ? IDE_CMD_READ : IDE_CMD_RDMUL;
4283   int write_cmd = (sector_per_block == 1) ? IDE_CMD_WRITE : IDE_CMD_WRMUL;
4284   if (sector_per_block > 7) panic("idestart");
Start disk operation: idestart

4286
4287  idewait(0); // wait for IDE device to be ready.
4288  outb(0x3f6, 0); // generate interrupt
4289  outb(0x1f2, sector_per_block); // number of sectors
4290  outb(0x1f3, sector & 0xff);
4291  outb(0x1f4, (sector >> 8) & 0xff);
4292  outb(0x1f5, (sector >> 16) & 0xff);
4293  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
4294  if(b->flags & B_DIRTY){
4295      outb(0x1f7, write_cmd);
4296      outsl(0x1f0, b->data, BSIZE/4);
4297  } else {
4298      outb(0x1f7, read_cmd);
4299  }
Start disk operation: idestart

```c
4288  outb(0x3f6, 0); // generate interrupt
4289  outb(0x1f2, sector_per_block); // number of sectors
4290  outb(0x1f3, sector & 0xff);
4291  outb(0x1f4, (sector >> 8) & 0xff);
4292  outb(0x1f5, (sector >> 16) & 0xff);
4293  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
4294  if(b->flags & B_DIRTY){
4295      outb(0x1f7, write_cmd);
4296      outsl(0x1f0, b->data, BSIZE/4);
4297  } else {
4298      outb(0x1f7, read_cmd);
4299  }
```
Interrupt handler for disk IO

4302 // Interrupt handler.
4303 void ideintr(void)
4304 {
4305   struct buf *b;
4306
4307   // First queued buffer is the active request.
4308   acquire(&idelock);
4309
4310   if((b = idequeue) == 0){
4311     release(&idelock);
4312     return;
4313   }
4314   idequeue = b->qnext;
4315
4316   // Read data if needed.
4317   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
4318      insl(0x1f0, b->data, BSIZE/4);
4319
4320   // Wake process waiting for this buf.
4321   b->flags |= B_VALID;
4322   b->flags &~ B_DIRTY;
4323   wakeup(b);
4324
4325   // Start disk on next buf in queue.
4326   if(idequeue != 0)
4327      idestart(idequeue);
4328
4329   release(&idelock);
4330 }
4331
If queue is not empty, service the next request.

Perform read.
Summary

- Interrupt, exception and system call
- Protection mode
- Trapframe
- System call
- Interrupt and Device driver