设备管理

蒋炎岩

南京大学 | 计算机软件研究所 | 系统与软件分析研究组

回到我们“认识”的计算机

- 在上计算机系以前，很多同学都有一个认识
  - “计算机系就是装(修)电脑的”

- 但是进了坑发现不是这么回事
  - 从来没有学过修电脑
  - 每天对着终端编程

为什么?

- 我们用电脑是和I/O设备交互
  - CPU、内存 (这是我们学的比较多的)
  - 但我们不直接和CPU/内存交互
    - 显卡连接显示器
    - USB鼠标、键盘
    - ……
- 计算机系学习“造计算机”，而I/O设备并不是最难造的

在计算机中访问I/O设备

I/O设备无非是用线连接到计算机上的

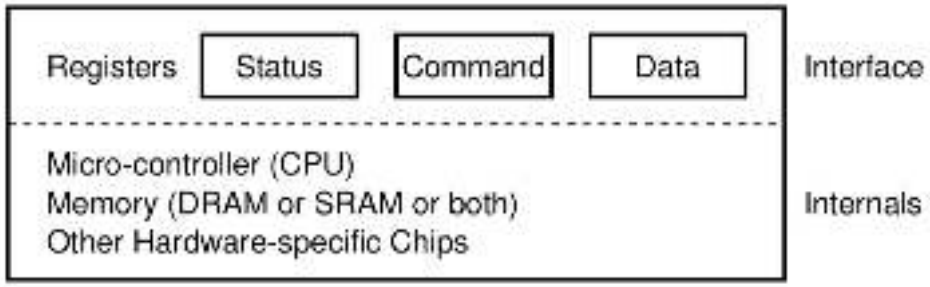
- 它们也必须提供接口 (类似API)实现访问

- 在Lab0中大家已经玩过各种I/O设备了
  - 每个设备是一系列的寄存器(可读/可写)
  - 读写数值有I/O设备约定的格式 – 这就搞定了!

- 操作系统 = C程序，所以一定可以通过指令访问I/O设备
  - I/O端口(port)，通过专门指令(in/out in x86)实现
  - 内存映射I/O，设备寄存器位于约定好的物理内存地址，直接C语言就能实现

例子

- IBM PC/AT 8042 Keyboard Controller
  - 有若干个端口(0x64, 0x60)
  - 端口I/O访问设备
- 典型的“Canonical Device”




```
size_t input_read(uintptr_t reg, ...)
  _KbdReg *kbd = (_KbdReg *)buf;
  int status = inb(0x64);
  kbd->keydown = 0;
  kbd->keycode = _KEY_NONE;
  if ((status & 0x1) == 0) {
  } else {
    if (status & 0x20) {
      // 这是个鼠标事件
    } else {
      int code = inb(0x60) & 0xff;
      // 得到了按键的扫描码
    }
  }
  …
}
```

/proc/iomem – procfs里什么好东西都有

- 在内核(操作系统实验)里，访问内存就能访问各种系统中的总线、控制器、I/O设备……

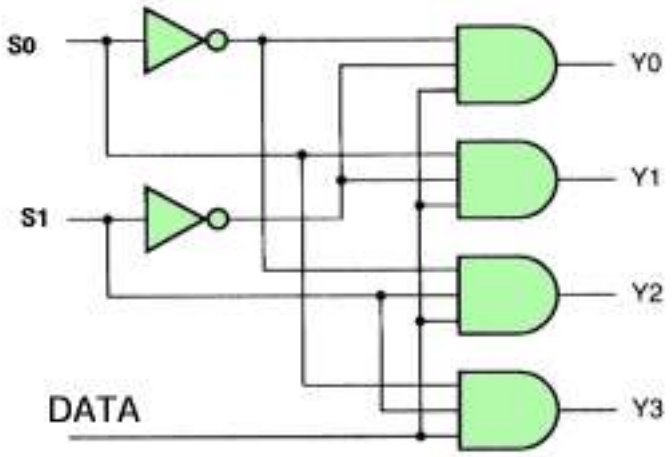
```
000c0000-000c8fff : Video ROM
...
00100000-bfaaffff : System RAM
  01000000-0184804e : Kernel code
  0184804f-01f439bf : Kernel data
  020c3000-0220bfff : Kernel bss
bfac4000-bfac5fff : System RAM
bfac6000-bfad5bff : ACPI Tables
c0000000-fdffffff : PCI Bus 0000:00
  ...
fe000000-ffffffff : reserved
  fec00000-fec003ff : IOAPIC 0
  fec81000-fec813ff : IOAPIC 1
  fed00000-fed003ff : HPET 0
    fed00000-fed003ff : PNP0103:00
  fed1f410-fed1f414 : iTCO_wdt.0.auto
  fed40000-fed44fff : PCI Bus 0000:00
  fee00000-fee00fff : Local APIC
```

内存映射I/O：实现

- 去年参加“龙芯杯”的计算机系统习作

```
#define VMEM_ADDR ((volatile void *)0x50000000)
```

  - 我们可以随便指定一个地址
  - 然后在实现地址翻译的时候用一个DEMUX
    - 把数据送到正确的I/O设备就行(内存/显存/设备控制器...)



内存映射I/O：麻烦

- 还记得之前sum++的例子么
  - -O0: 不确定; -O1: N; -O2: 2N
  - 防止编译器作出优化：volatile

- 如果是内存映射的I/O，例如我向控制寄存器里写128个0，被gcc优化成写一个0，就错了

- 还有更大的麻烦
  - 系统里有缓存，movl $0, (addr)写入的不是设备而是缓存
  - i386本身没有缓存，但对于有缓存的情况呢(比如PCI Conf)?
  - 我们自己的CPU也有缓存，像素是否可能写不到显存里?

缓存控制

- 对于固定的物理内存映射，物理页可以强行规定no-cache
  - 我们CPU中采取的粗暴手段，直接解决问题

- 在各个级别都有
  - CR3/页目录/页表的PCD (page cache disable) bit
  - CR3/页目录/页表的PWT (page write-through) bit
  - CLFLUSH addr (cache-line flush)

中断

回顾：刚才的机制

- 通过端口/MMIO实现设备行为的各类控制

- 但有一个问题
  - 设备可能主动产生事件(时钟、按键、鼠标、网络包……)
  - 系统中设备可能很多，如果逐个轮询，非常浪费时间

- 中断：I/O设备的通知
  - 中断自带一些信息(#IRQ)
  - 设备可能共享#IRQ，因此仍然需要轮询IRQ上注册的设备

中断控制器：本身也是一个I/O设备！

- 8259 PIC (Programming Interrupt Controller)
  - 通过PIO访问()
- APIC (Advanced PIC)
  - 通过内存映射访问(刚才在iomap里已经见到过了)
  - IO APIC: 负责I/O中断(磁盘、PIC设备……)
  - Local APIC: 负责本地中断(时钟, IPI, ...)
- 功能
  - 屏蔽中断、响应中断、……

/proc/interrupts – How Old Are You?

- 各种中断的统计都有(截取了一些)

```
           CPU0
  0:         43   IO-APIC   2-edge      timer
  1:         10   IO-APIC   1-edge      i8042
  8:          0   IO-APIC   8-edge      rtc0
  9:          0   IO-APIC   9-fasteoi   acpi
 12:        156   IO-APIC  12-edge      i8042
 15:    2459445   IO-APIC  15-edge      ata_piix
 19:    2802399   IO-APIC  19-fasteoi   enp0s3
NMI:          0   Non-maskable interrupts
LOC:   50395912   Local timer interrupts
PMI:          0   Performance monitoring interrupts
RTR:          0   APIC ICR read retries
TRM:          0   Thermal event interrupts
THR:          0   Threshold APIC interrupts
```

还记得i8042吗?
为什么这么少?

中断的好处

- 处理器再也不用轮询设备的状态了

- 设备在发生事件后会以中断的方式告诉处理器
  - 有数据来了，可以读走了！
  - 你上次让我传送的数据已经完成了！
  - 设备从计算机上拔掉了！ (PnP interrupt)
  - CPU过热了！我在报警！ (ACPI interrupt)

- 处理器还可以根据负载动态调节
  - 有些中断可以暂时屏蔽(mask)，优先处理重要的事件

中断 + DMA

另一个节约CPU时间的机制：DMA

- I/O设备的名字不是乱取的
  - 作用就是input/output
- 因此在内存和设备之间传送数据是非常普遍的
  - 把材质/贴图/...传送给显卡
  - 把一大段数据送到磁盘/网络
- 不如我们
  - 把内存准备好，用一个特殊的I/O设备
    - 从设备读取/写入设备
    - 读写完毕后用中断通知处理器

Intel 8237 DMA控制器

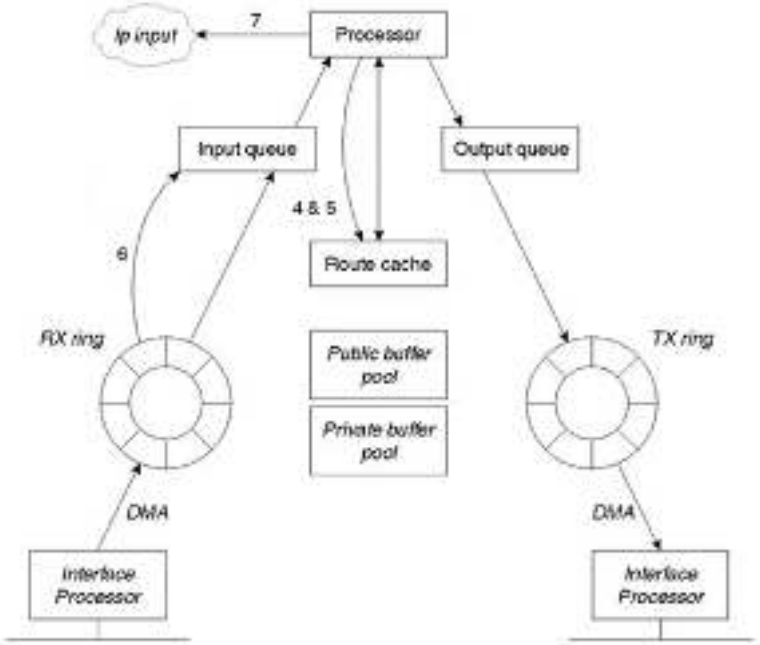
- 系统中的另一个处理器
  - 接到指令(通过Port IO设置)以后开始执行程序
  - 执行完毕后发送中断

```
for (int i = 0; i < nbytes; i += b) {
  SRC[i] := DST[i]; // SRC/DST can be a memory
                    // or an I/O port
}
send_interrupt();
```

  - 能实现内存-I/O设备；内存-内存之间的DMA

例子：NIC

- 网卡有两个接口TX (transmit)和RX (receive)
  - 内核维护环形缓冲区TX/RX buf (与网卡共享)
  - 将TX包写入TX buf，DMA到网卡
  - RX包自动读入RX buf (缓冲区网卡自行丢弃)，中断通知



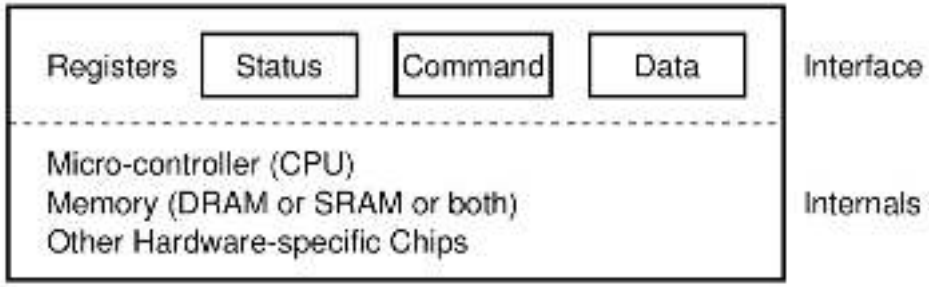
DMA: Memory Coherence

- 场景1
  - 我刚把新鲜的内存写好，还在缓存里，立即送去DMA，I/O设备能得到正确的数据吗?
  - 如果一边修改，一边DMA会发生什么?
- 场景2
  - 我mmap一个文件，文件是不会载入内存的(只是标记)，如果我要把这块映射的内存传递给网卡/显卡.....？ 如何处理?

设备驱动程序

小结

- 我们已经知道了用于管理I/O设备的机制
  - 设备模型
  - 中断
  - DMA



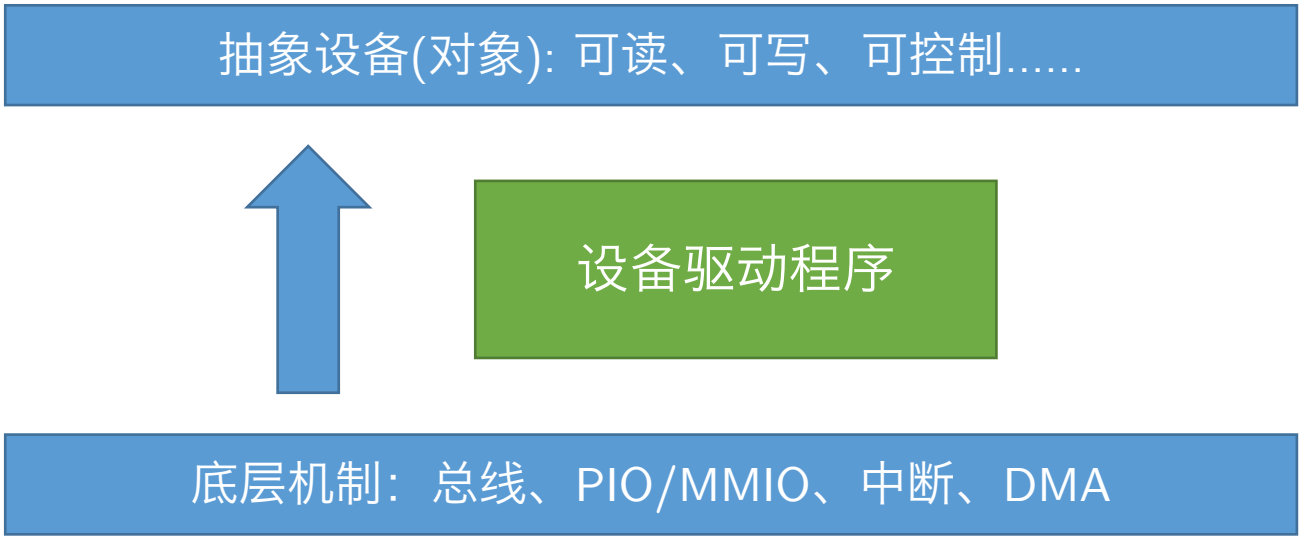
- 计算机系统遍布着处理器
  - 每个I/O设备处理指令/数据都需要处理器
  - DMA控制器就是个执行特定指令的循环
  - ……

如何用这些机制有效地管理设备?

- 非常显而易见的问题：I/O设备是用来做什么的?

I/O设备是用来Input/Output的

- 因此设备可以抽象成是一个数据流或数组，支持
  - read (读), write (写), ioctl, mmap, ...



I/O设备：例子

- 终端 (/dev/tty)
  - read (读出字符), write (写入字符串), ioctl (获取信息/设置)
- 帧缓冲 (/dev/fb0)
  - write (写入像素)，mmap (映射)
- 磁盘 (/dev/sda)
  - read, write, ... 但底层是块设备(不直接实现read/write)
- 如果对性能有额外要求，处理会更复杂一些
  - PowerVR - Driver – OpenGL ES (Vulkan?) – Unity 3D – 王者荣耀

设备 ≠ 某个设备 (只是一个对象)

- 只要支持file_operations，就可以是设备
  - /dev/zero, /dev/null
  - /dev/random, /dev/urandom – 随机数生成器
  - /dev/rtc – 时钟
  - /dev/kvm – 虚拟化接口 (可以实现libvirt)

- 更有趣的
  - /dev/stdin, /dev/stdout – 当前进程的stdin/stdout
  - /dev/input/{event, mice, ...} -

Everything is a File：设备也是文件

- 例子：Linux中的字符设备驱动
  - 打开/dev/xxx，最终就会调用这些操作(你也可以写驱动哦)

```
struct file_operations {
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t (*read) (struct file *, char __user *
                   size_t, loff_t *);
  ssize_t (*write) (struct file *, const char __user *,
                   size_t, loff_t *);
  int (*ioctl) (struct inode *, struct file *,
                unsigned int, unsigned long);
  int (*open) (struct inode *, struct file *);
  int (*release) (struct inode *, struct file *);
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*flush) (struct file *);
  ...
};
```