操作系统

第6章 输入/输出系统

朱小军, 教授
https://xzhu.info
南京航空航天大学
计算机科学与技术学院
2025年春

回忆冯诺依曼计算机五大部件（与前面章节的对应关系？）

输入输出管理

- IO硬件
  - IO硬件的发展、构成
  - IO设备的控制方式
- IO软件
  - 中断处理程序、驱动程序
    - xv6驱动程序案例
  - 设备独立软件
  - 用户层IO软件
- 磁盘
  - 结构
  - 磁盘调度方法

I/O 设备的发展

Typewriter 打字机

- 1880s开始流行

Teletype, TTY, 电传打字机

- 按键--> 信号-->电报网络--->信号-->打印机

Siemens t37h (1933) from Wikipedia

TTY作为计算机终端

Teletype Model 33 ASR teleprinter, with punched tape reader and punch, usable as a computer terminal. from Wikipedia

tty的后续发展

1950s
model 28 by Teletype

Intelligent terminal

1978
VT100
Video Terminal

2025

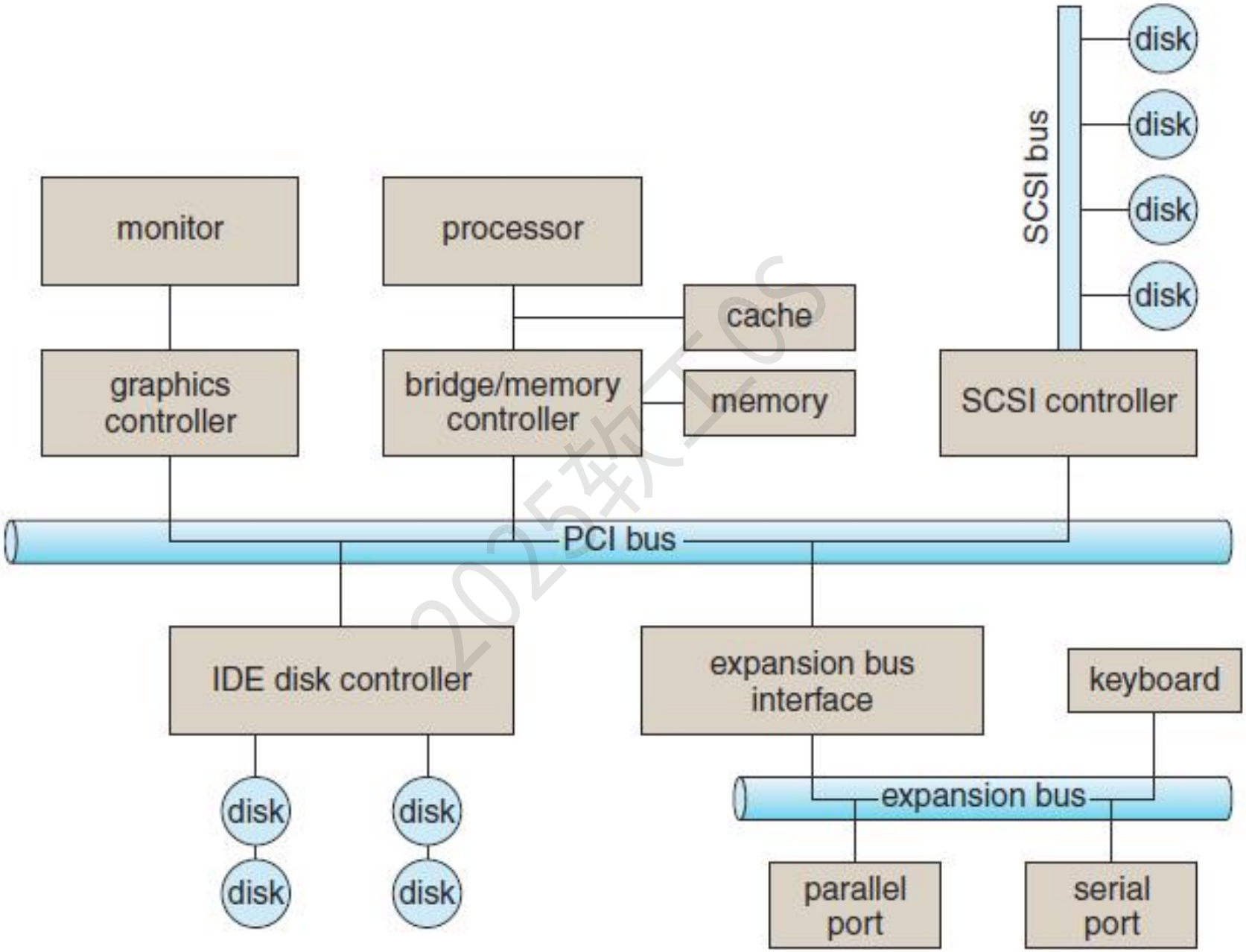
```
ubuntu@VM-16-15-ubuntu:~$ ls /dev/tty*
/dev/tty     /dev/tty26  /dev/tty44  /dev/tty62     /dev/ttyS21
/dev/tty0    /dev/tty27  /dev/tty45  /dev/tty63     /dev/ttyS22
/dev/tty1    /dev/tty28  /dev/tty46  /dev/tty7      /dev/ttyS23
/dev/tty10   /dev/tty29  /dev/tty47  /dev/tty8      /dev/ttyS24
/dev/tty11   /dev/tty3   /dev/tty48  /dev/tty9      /dev/ttyS25
/dev/tty12   /dev/tty30  /dev/tty49  /dev/ttyprintk /dev/ttyS26
/dev/tty13   /dev/tty31  /dev/tty5   /dev/ttyS0     /dev/ttyS27
/dev/tty14   /dev/tty32  /dev/tty50  /dev/ttyS1     /dev/ttyS28
/dev/tty15   /dev/tty33  /dev/tty51  /dev/ttyS10    /dev/ttyS29
/dev/tty16   /dev/tty34  /dev/tty52  /dev/ttyS11    /dev/ttyS3
```

https://www.yabage.me/2016/07/08/tty-under-the-hood/

现在：屏幕、键盘、鼠标、语音、手势

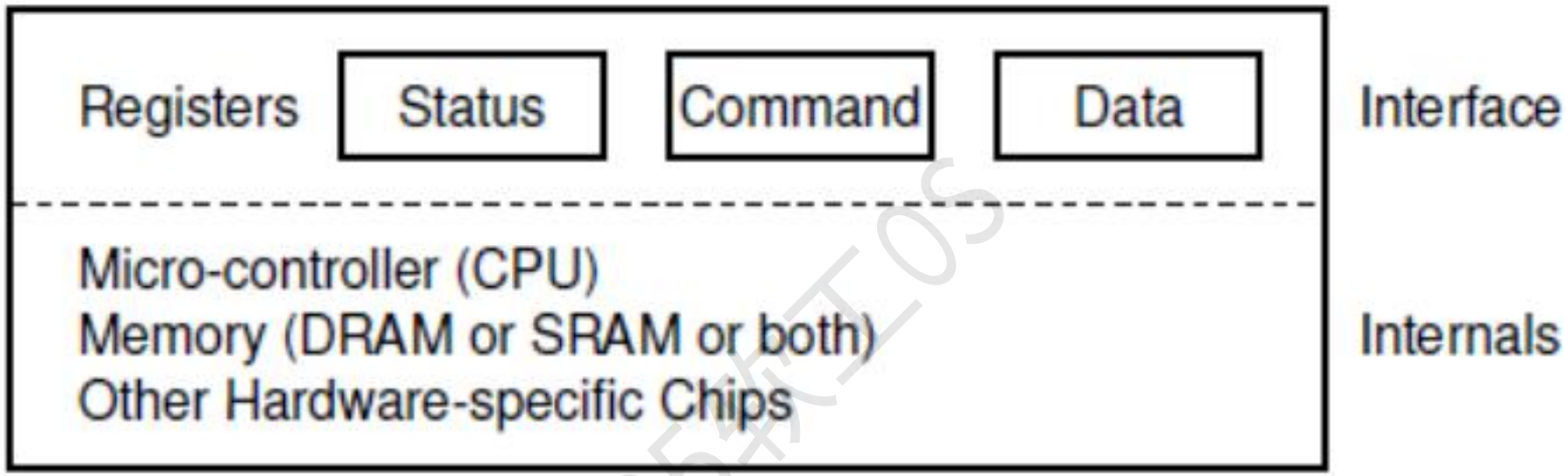
将来?

IO设备与CPU、内存的连接


monitor
processor
cache
graphics controller
bridge/memory controller
memory
SCSI bus
disk
disk
disk
disk
SCSI controller
PCI bus
IDE disk controller
expansion bus interface
keyboard
disk
disk
disk
disk
expansion bus
parallel port
serial port

I/O 设备的典型构成

两部分



- 硬件接口
  - 系统通过接口控制该设备
- 内部实现
  - 实现设备的具体功能

读写控制器寄存器的两种方法

- 特殊的指令，称为I/O 指令
  - x86机器的 in 和 out 指令,如 out <port_num>

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

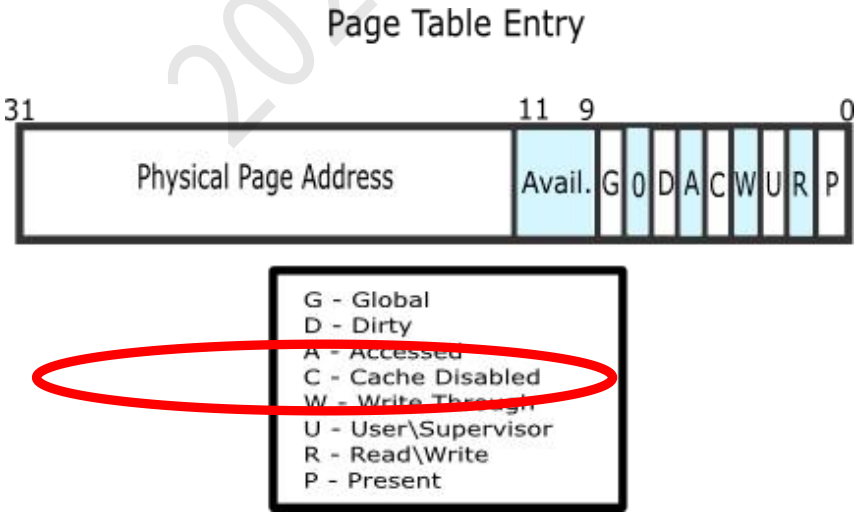
部分I/O端口地址

- 如何避免用户程序错误操作I/O设备？
  - 将I/O指令设置为特权指令，只能由内核执行
- 特殊I/O指令在使用时有何缺点？
  - 只能使用汇编语言编写程序

法2：内存映射I/O

- 将部分内存地址分配给I/O设备（的寄存器）
- 向这部分地址读写时做I/O操作
  - 不需要特殊的指令
  - 比如， movl %eax, [78EA63] 可能就是输出eax的内容到I/O 设备
  - 可以用c语言编程，而不是汇编，比如将3写入一个设备控制器的寄存器中： *a=3
- 如何避免用户程序错误操作I/O设备？
  - 将地址设置为用户程序不可访问（how？）
  - 一般的做法，地址映射到内核区

- 如何避免内容被缓存?
  - 现代计算机一般会缓存内存内容
  - 若I/O设备控制器寄存器的内容被缓存。。。
    - 比如： while（status==BUSY）；
    - 若在第一次测试时缓存status，则发生死循环。（为何缓存正常内存没关系？）



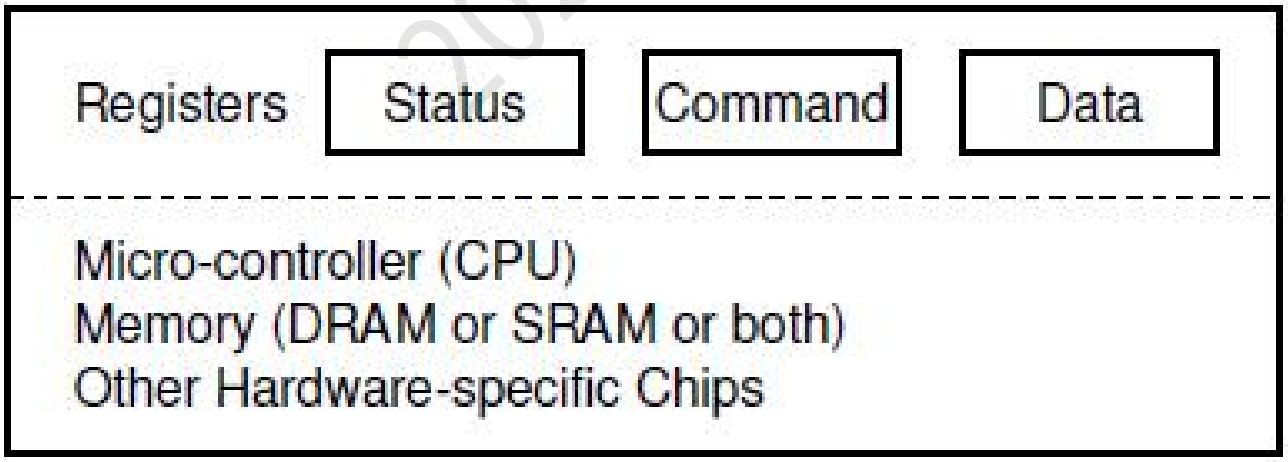
输入输出管理

- IO硬件
  - IO硬件的发展、构成
  - IO设备的控制方式
- IO软件
  - 中断处理程序、驱动程序
    - xv6驱动程序案例
  - 设备独立软件
  - 用户层IO软件
- 磁盘
  - 结构
  - 磁盘调度方法

第一种方式：轮询

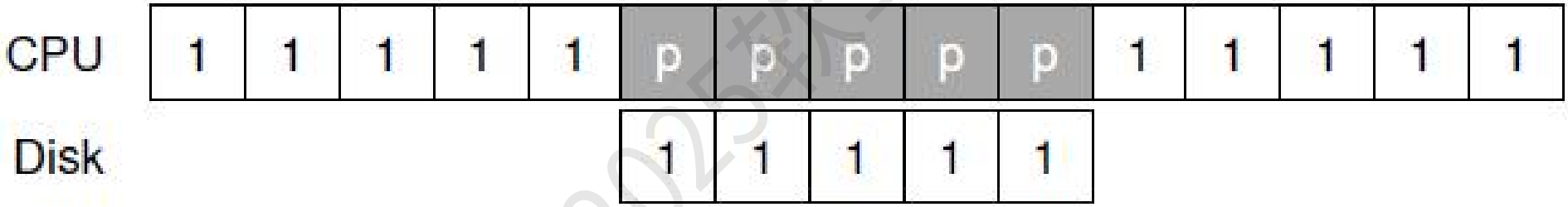
轮询

```
while(STATUS==BUSY);          //轮询（polling）
write data to DATA register
write command to COMMAND register
while(STATUS==BUSY);          //轮询（polling）
```

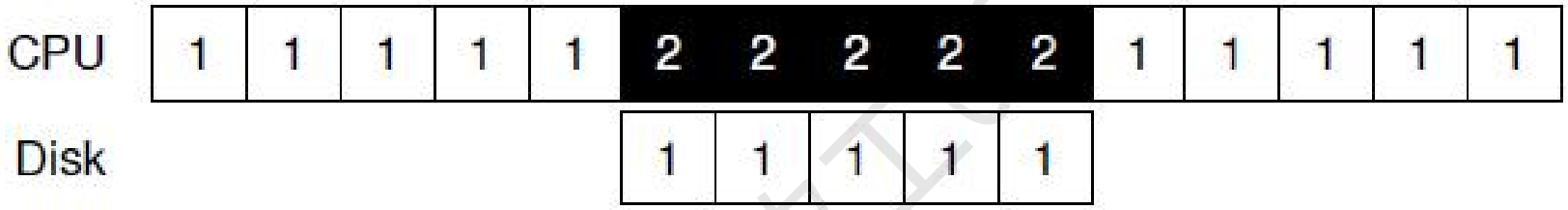
轮询方式的优劣

- 优点：简单、有效；OS在启动时需要使用
- 缺点：CPU被占用

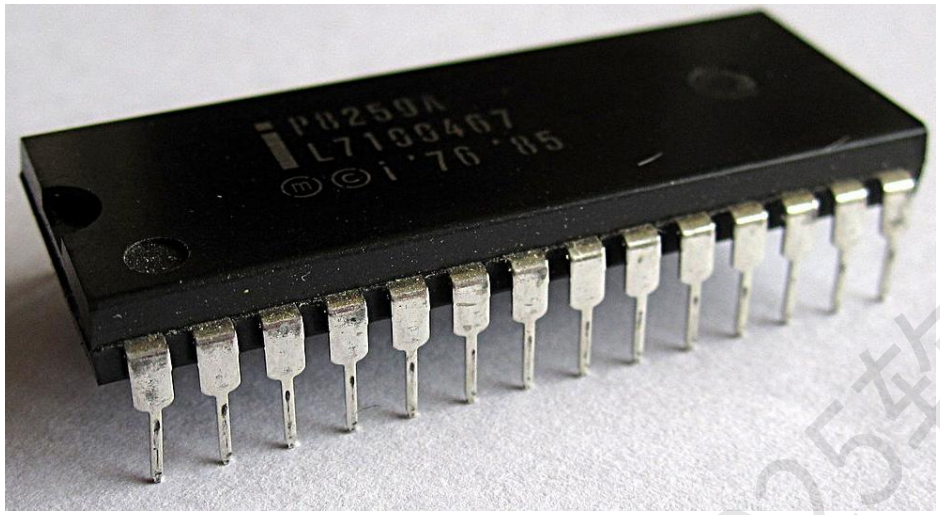


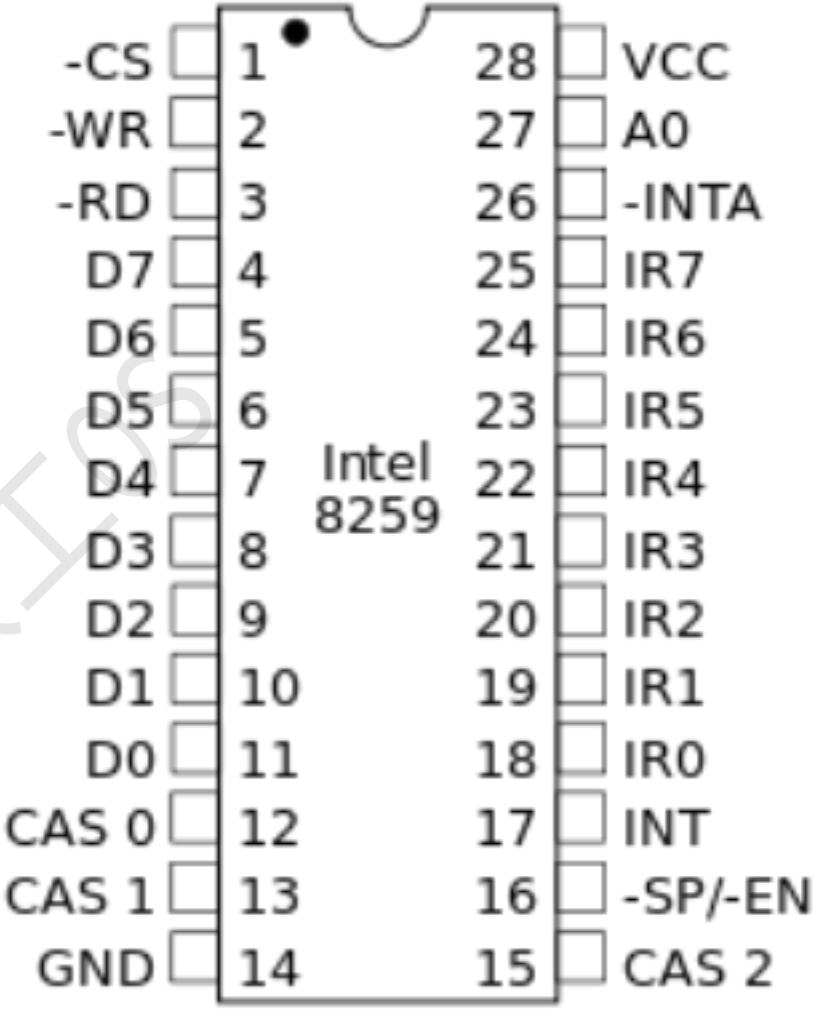
图中数字1表示进程1，p表示polling

第二种方式：中断

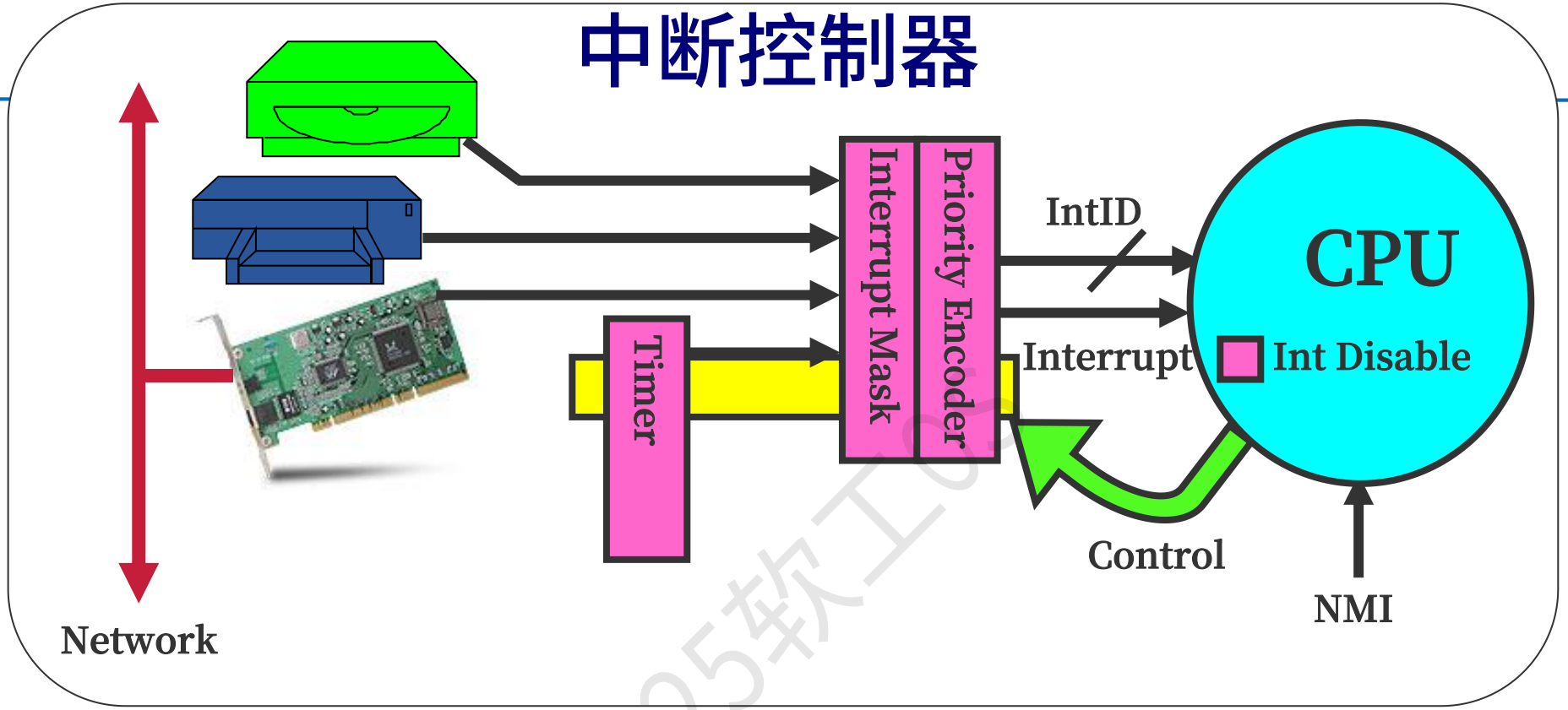


图中数字1、2分别表示进程1、进程2





中断控制器（Intel 8259A IRQ）



- 设备可以从中断线路发出中断
- 中断控制器选择一个中断输出
  - CPU 可以屏蔽中断（内部有一个标志位）被特殊指令所控制，cli

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

英特尔奔腾处理器中断号（0-31不可被屏蔽）

中断协议

- OS 使用设备 时，
  - set_up_controller( );
  - scheduler( );
- 设备控制器结束任务时发出中断
- OS 在收到中断信号后，
  - unblock_user( );
  - return_from_interrupt( );

中断协议的优劣

- 中断一定优于轮询吗?
  - 如果I/O设备处理命令的速率较低，中断确实可节省大量时间;
  - 但如果I/O可以迅速完成呢?
- 当系统启动、中断向量尚未初始化时，只能采用轮询

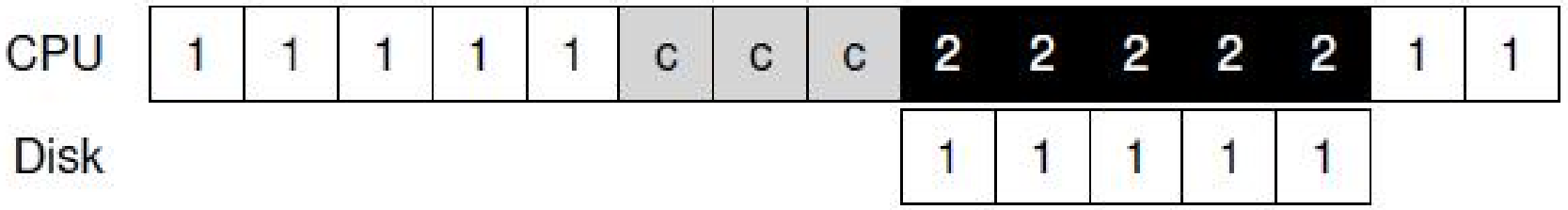
回顾：输入输出管理

- IO硬件
  - IO硬件的发展、构成
    - 硬件接口（各种寄存器）、寄存器的编址方式（独立编址（特殊IO指令）、统一编址（内存映射IO））
  - IO设备的控制方式
    - 轮询（不停查看状态）
      轮询消耗CPU
    - 中断（状态更新时发出中断信号）
      中断方式依赖中断处理过程，计算机启动时无法使用中断
    - 各有优劣

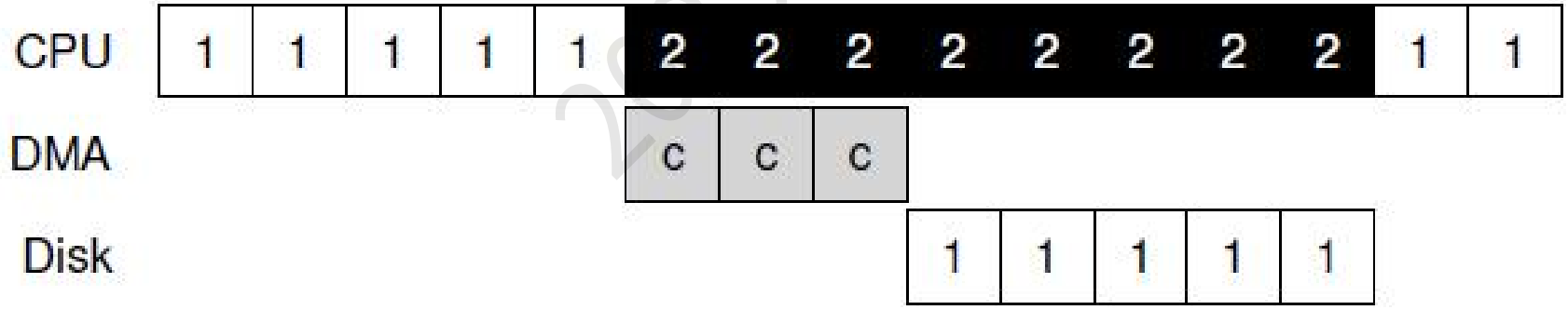
中断驱动的I/O以需要CPU深度参与

- 以字节（或字）为单位在内存和设备控制器的寄存器之间传输
- 中断频繁，让CPU处理有点浪费

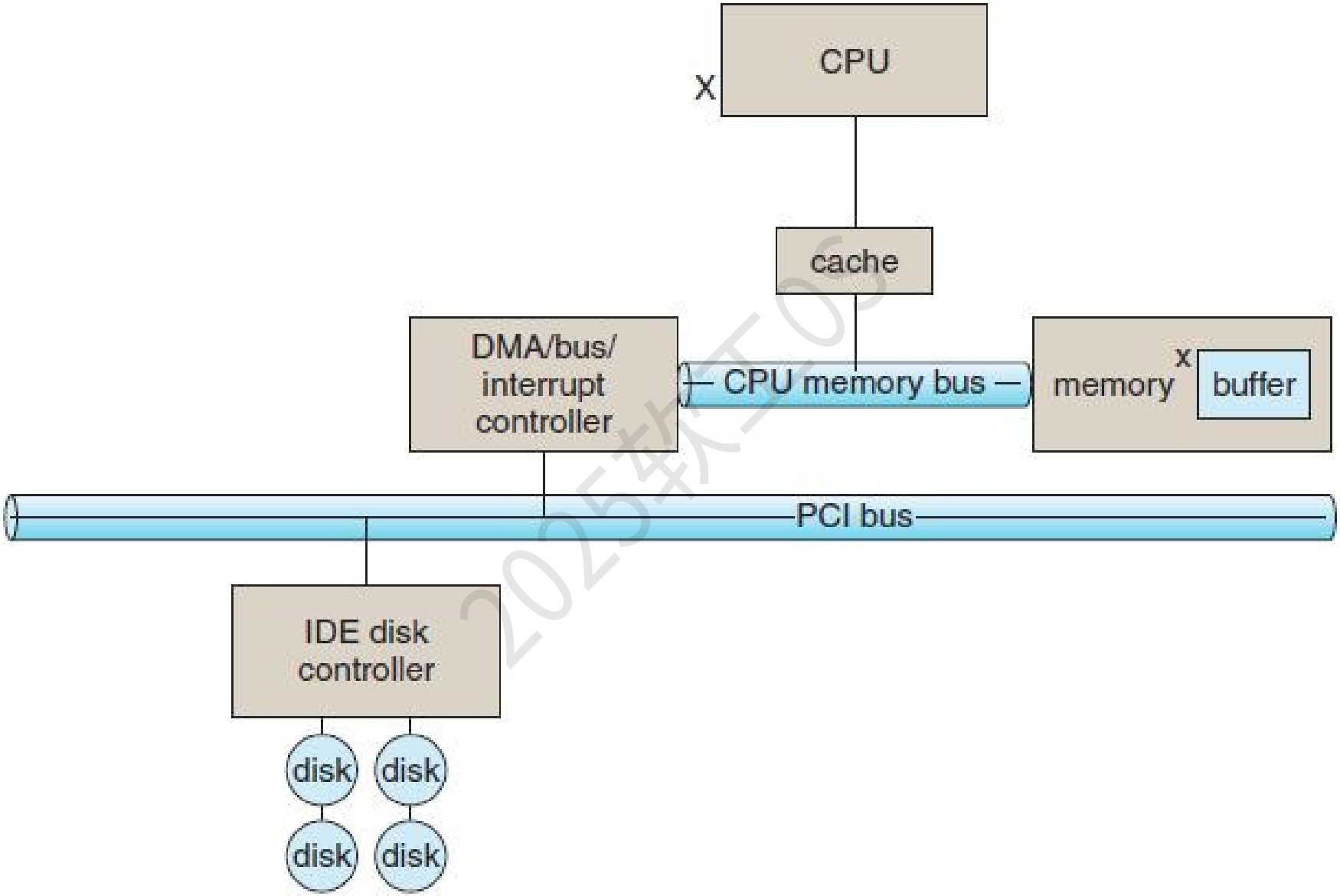
第三种方式：
批量传输数据时采用：
DMA（direct memory access）



未采用DMA时，数据传输由CPU来完成



采用DMA后，数据传输由DMA来完成


CPU
x
cache
DMA/bus/
interrupt
controller
CPU memory bus
memory
x
buffer
PCI bus
IDE disk
controller
disk
disk
disk
disk

DMA 控制器

- DMA方式需要借助DMA控制器，可以看做是一个特殊功能的CPU
- OS 使用DMA 时，
  - set_up_DMA_controller( );
  - scheduler( );
- DMA 处理任务，结束时发出中断
- OS 在收到中断信号后，
  - unblock_user( );
  - return_from_interrupt( );

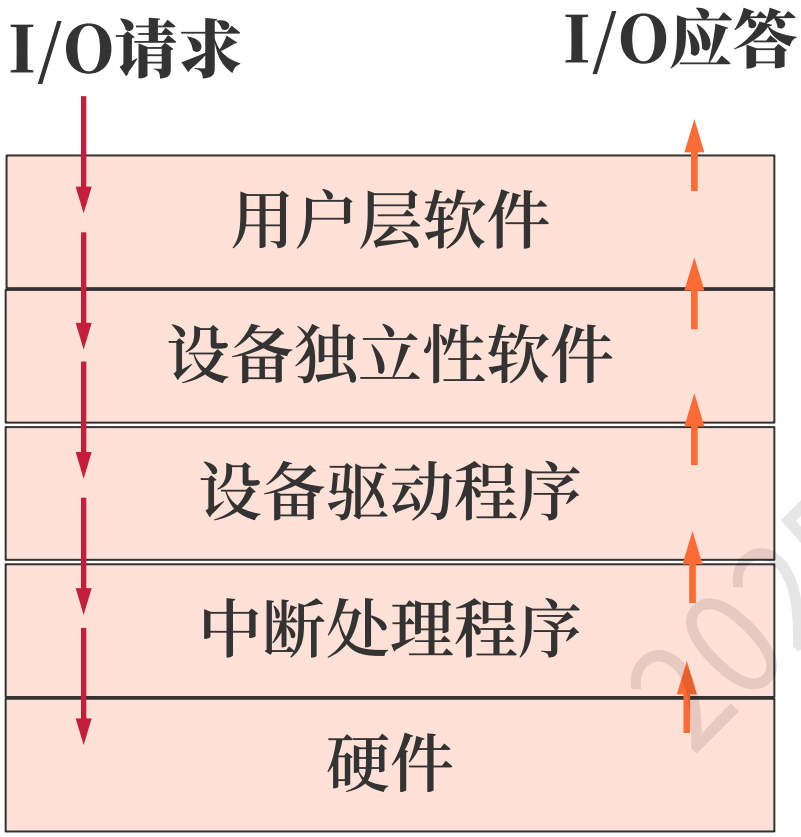
DMA的优劣

- DMA一定比中断好吗?
  - 当CPU较忙时，确实可以节省CPU时间
  - 当CPU较闲时，DMA的效率低于CPU（why？）
  - 当然，大多数时候，还是值得的

输入输出管理

- IO硬件
  - IO硬件的发展、构成
  - IO设备的控制方式
- IO软件
  - 中断处理程序、驱动程序
    - xv6驱动程序案例
  - 设备独立软件
  - 用户层IO软件
- 磁盘
  - 结构
  - 磁盘调度方法

IO软件的层次结构



系统调用、库函数

缓冲区管理、分配与回收、假脱机

设置设备寄存器；检查状态

发出操作指令、驱动设备工作

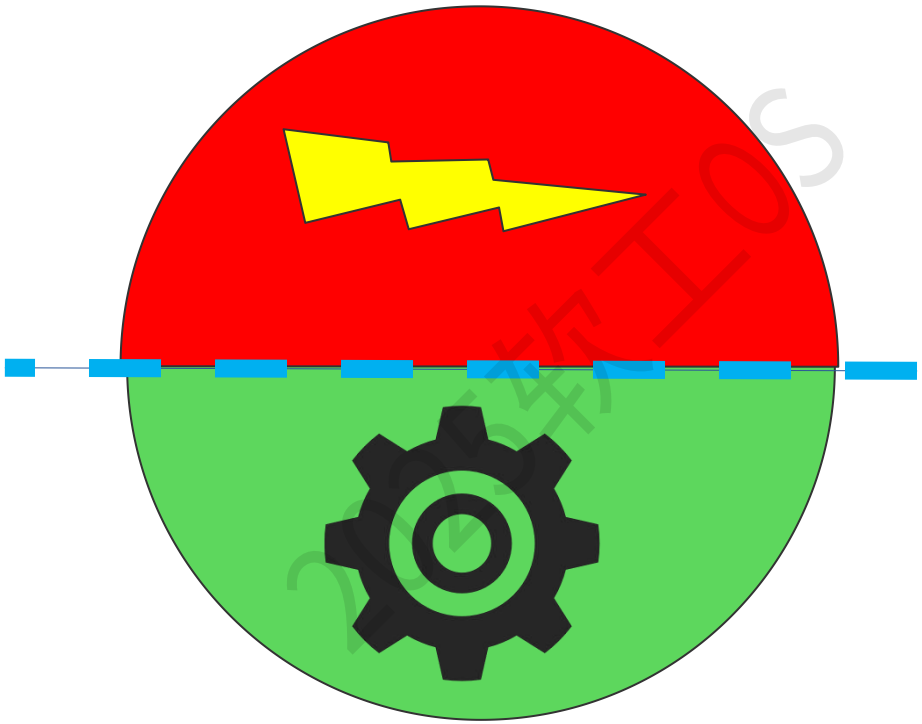
执行I/O操作

输入输出管理

- IO硬件
  - IO硬件的发展、构成
  - IO设备的控制方式
- IO软件
  - 中断处理程序、驱动程序
    - xv6驱动程序案例
  - 设备独立软件
  - 用户层IO软件
- 磁盘
  - 结构
  - 磁盘调度方法

中断处理过程

Top half：快速响应

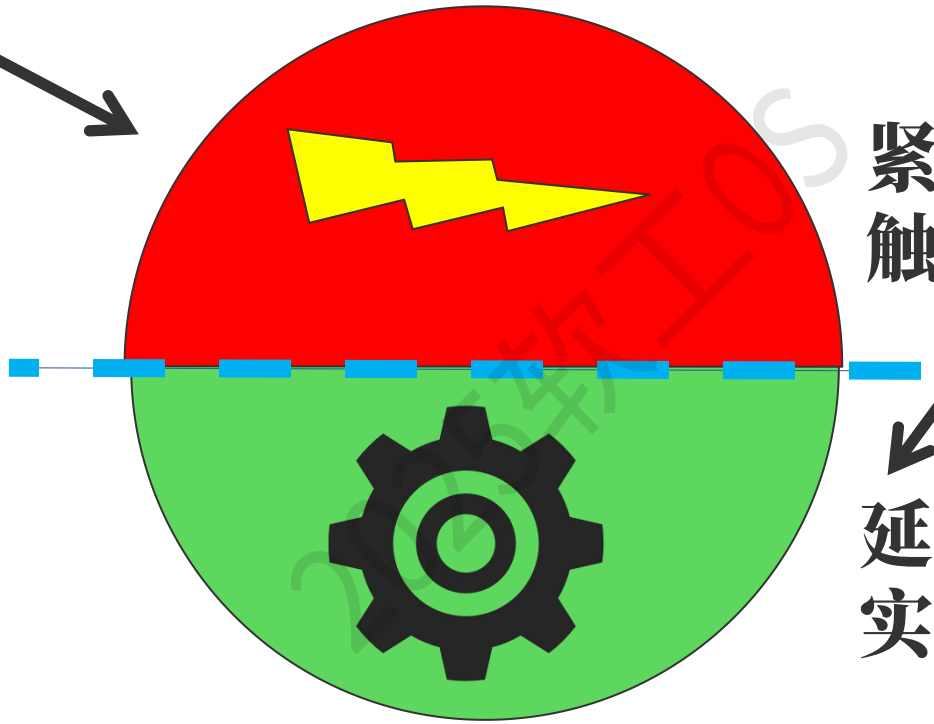


Bottom half：慢慢处理

中断处理过程

中断

Top half：快速响应



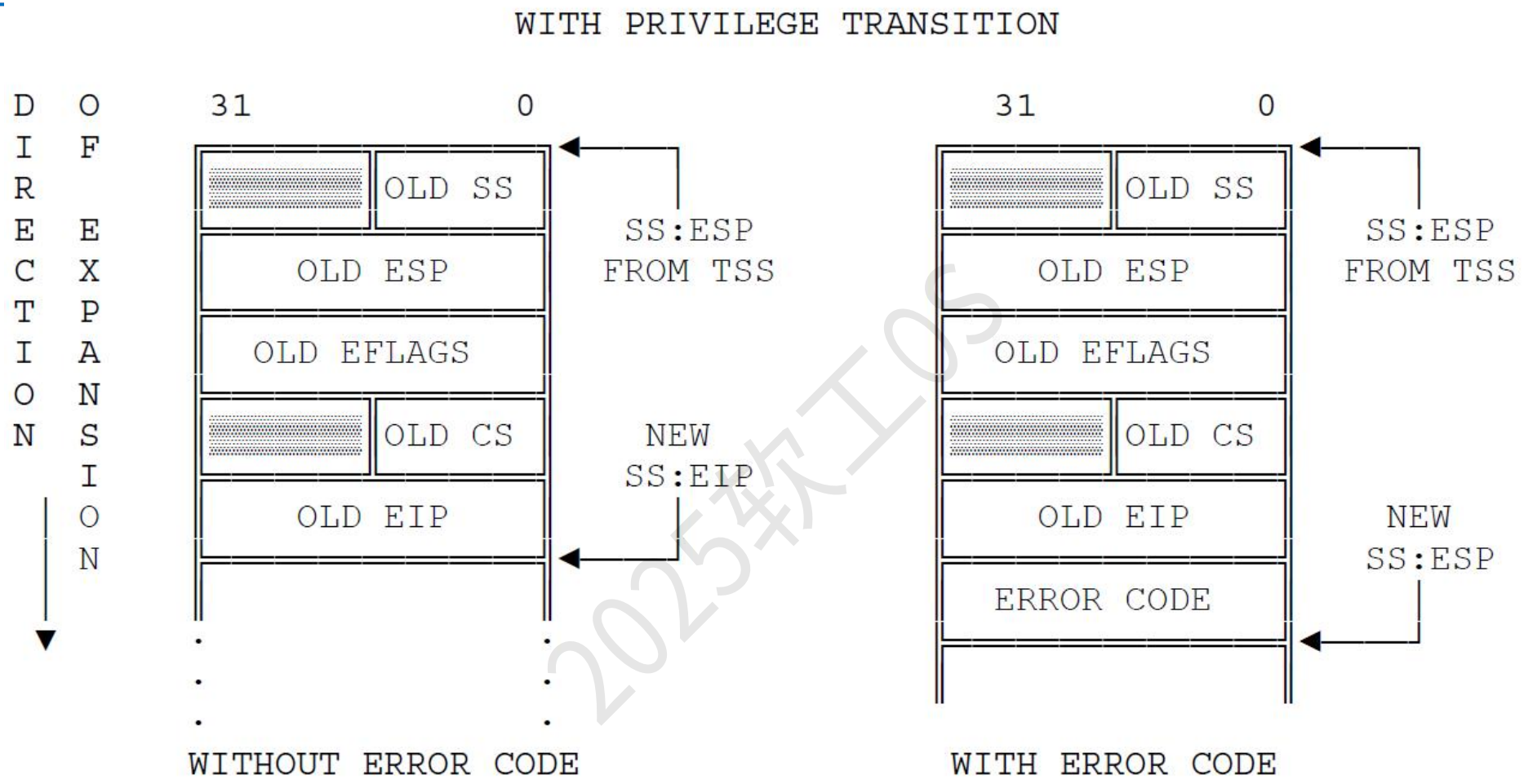
紧急响应
触发Bottom half

延迟工作
实质处理

Bottom half：慢慢处理

附：操作系统的中断处理过程

- x86 硬件自动执行（内核中找不到指令）
  - 保存esp和ss到CPU内部的寄存器中
  - 从一个任务段描述符中加载新值到ss和esp
    （由task register指示，此寄存器可通过ltr指令修改）
  - push （原来的）ss　only on privilege change
  - push （原来的）esp only on privilege change
  - push eflags，cs，eip
  - push error code （部分中断）
  - 修改eflags （外部中断清IF）
  - 根据描述符修改 cs 和 eip



CPU处理中断时栈的变化(硬件自动完成)

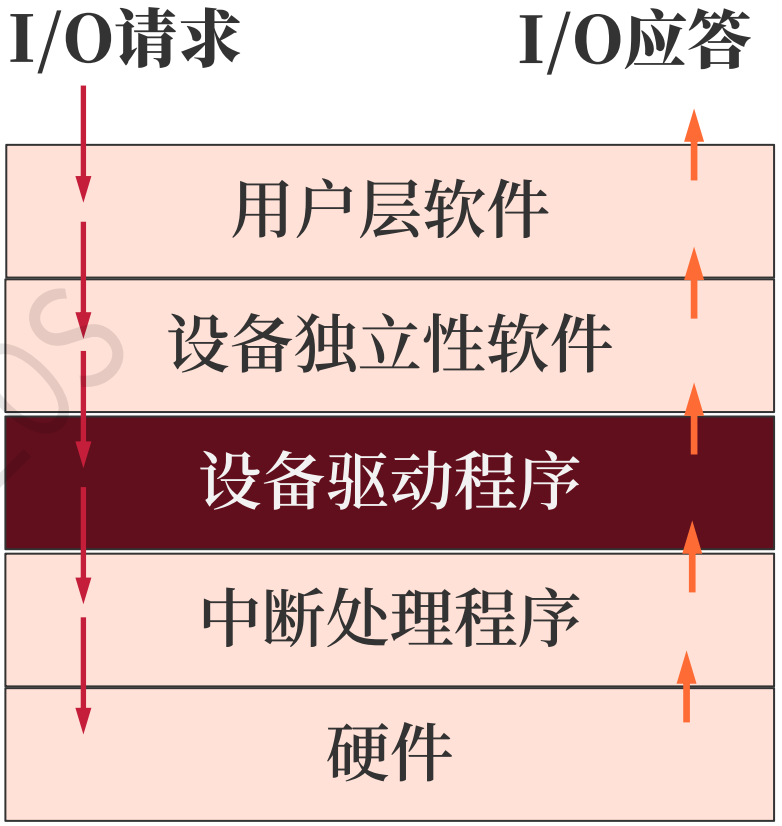
pp159,《Intel 80386 programmer's reference manual》

xv6操作系统中的中断处理过程

- 软件部分
  - vectors.S
    - 对没push error code的中断push 0（why?)
    - jmp alltraps
  - alltraps　//trapasm.S
    - 保存其他寄存器值
    - call trap(tf)
    - trapret
  - trap（）　//trap.c
    - 对不同中断做不同处理

设备驱动程序

- 对上提供服务
  - 实现设备独立性软件发出的调用，比如open，read，write
  - 根据设备状态发出相应IO指令
- 对下响应需求
  - 中断处理程序移交的处理任务（bottom half）
  - Linux中，中断处理程序是驱动程序的一部分



输入输出管理

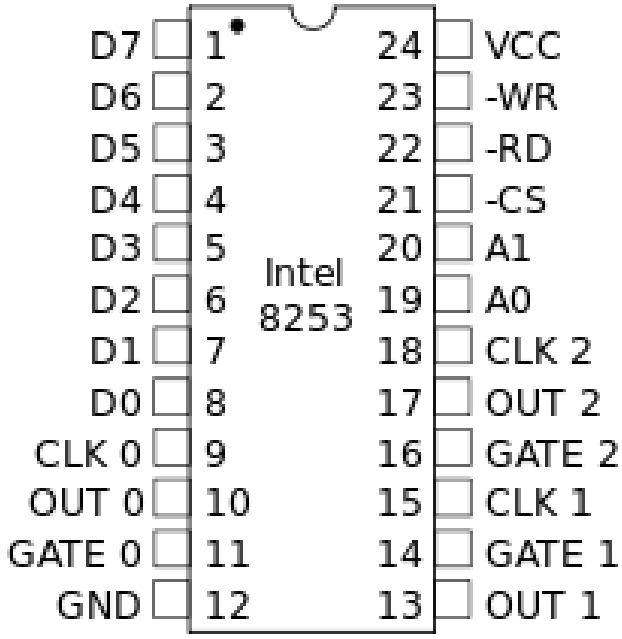
- IO硬件
  - IO硬件的发展、构成
  - IO设备的控制方式
- IO软件
  - 中断处理程序、驱动程序
    - xv6驱动程序案例
  - 设备独立软件
  - 用户层IO软件
- 磁盘
  - 结构
  - 磁盘调度方法

实例：xv6的时钟、键盘

时钟

- 硬件时钟
- 中断处理程序

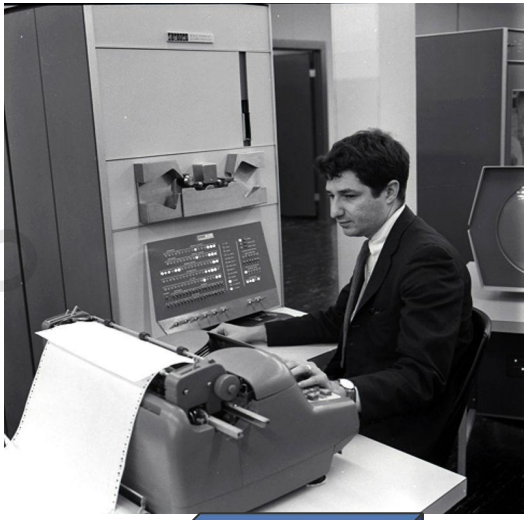

```
case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
//...
//...
 //...
if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
    exit();

  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();

  // Check if the process has been killed since we yielded
if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
    exit();
```



输入输出设备terminal、console、shell

Terminal（tty，打印输出，允许多个）

Console（直连计算机，只有1个）



电缆

shell
（软件）

电缆

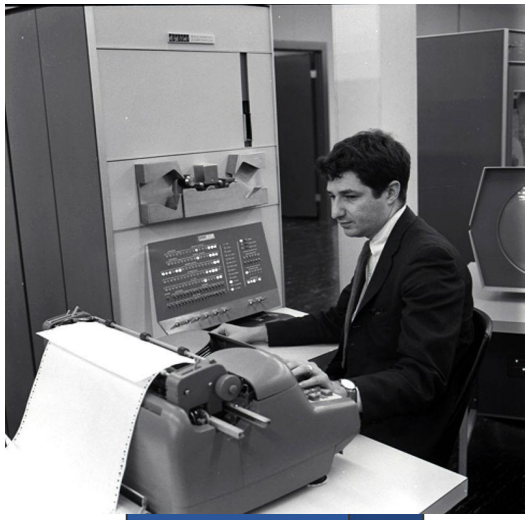
计算机

Terminal（tty，屏幕输出，允许多个）

xv6的输入和输出

- 除了磁盘外，输入输出有两类
  - keyboard（输入）、cga（输出）
  - uart（输入和输出）
- shell在启动时，打开console作为输入输出设备

Console（直连计算机，只有1个）



Terminal（tty，屏幕输出，允许多个）

电缆
uart协议

shell
（软件）

计算机

输入：键盘、uart

- 硬件
  - press和release时均会产生中断
  - OS需要从控制器端口读取代码
- 中断处理程序
- 缓冲区

```
#define INPUT_BUF 128
struct {
  char buf[INPUT_BUF];
  uint r;  // Read index
  uint w;  // Write index
  uint e;  // Edit index
} input;
```

```
void
kbdintr(void)
{
  consoleintr(kbdgetc);
}
```

```
void
uartintr(void)
{
  consoleintr(uartgetc);
}
```

```
kbdgetc(void)
{
  static uint shift;
  static uchar *charcode[4] = {
    normalmap, shiftmap, ctlmap, ctlmap
  };
  uint st, data, c;

  st = inb(KBSTATP);
  if((st & KBS_DIB) == 0)
    return -1;
  data = inb(KBDATAP);

  if(data == 0xE0){
    shift |= E0ESC;
    return 0;
  } else if(data & 0x80){
    // Key released
    data = (shift & E0ESC ? data : data & 0x7F);
    shift &= ~(shiftcode[data] | E0ESC);
    return 0;
  } else if(shift & E0ESC){
    // Last character was an E0 escape; or with 0x80
    data |= 0x80;
    shift &= ~E0ESC;
  }

  shift |= shiftcode[data];
  shift ^= togglecode[data];
  c = charcode[shift & (CTL | SHIFT)][data];
  if(shift & CAPSLOCK){
    if('a' <= c && c <= 'z')
      c += 'A' - 'a';
    else if('A' <= c && c <= 'Z')
      c += 'a' - 'A';
  }
  return c;
```

```
void
consoleintr(int (*getc)(void))
{
  int c;

  acquire(&input.lock);
  while((c = getc()) >= 0){
    switch(c){
    case C('P'):  // Process listing.
      procdump();
      break;
    case C('U'):  // Kill line.
      while(input.e != input.w &&
            input.buf[(input.e-1) % INPUT_BUF] != '\n'){
        input.e--;
        consputc(BACKSPACE);
      }
      break;
    case C('H'): case '\x7f':  // Backspace
      if(input.e != input.w){
        input.e--;
        consputc(BACKSPACE);
      }
      break;
    default:
      if(c != 0 && input.e-input.r < INPUT_BUF){
        c = (c == '\r') ? '\n' : c;
        input.buf[input.e++ % INPUT_BUF] = c;
        consputc(c);
        if(c == '\n' || c == C('D') || input.e == input.
          input.w = input.e;
          wakeup(&input.r);
        }
      }
      break;
    }
```

输入输出管理

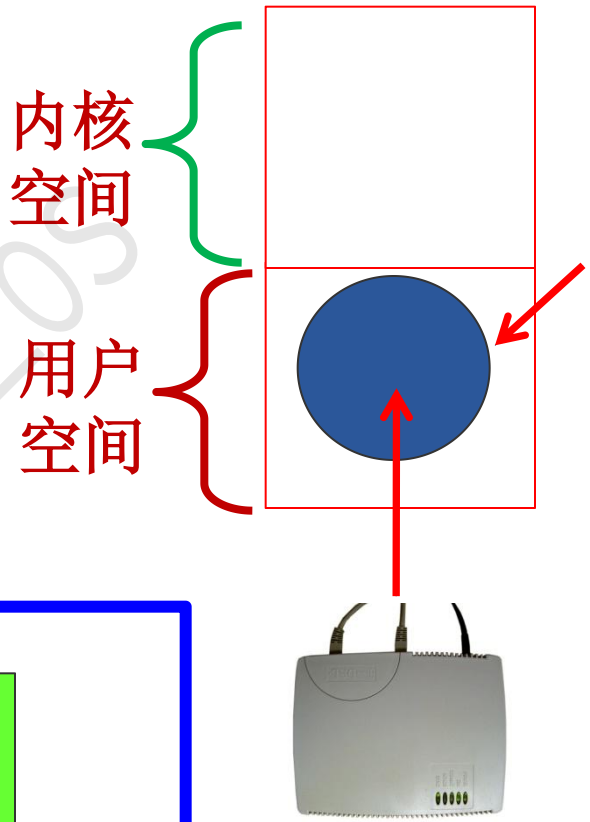
- IO硬件
  - IO硬件的发展、构成
  - IO设备的控制方式
- IO软件
  - 中断处理程序、驱动程序
    - xv6驱动程序案例
  - 设备独立软件
  - 用户层IO软件
- 磁盘
  - 结构
  - 磁盘调度方法

设备独立软件

- 在驱动程序之上，设置一层软件，称为与设备无关的IO软件，或设备独立性软件
- 统一驱动程序的接口
  - 设备分类：字符设备、块设备、网络设备等
  - read，write，ioctl【所有设备均提供这些操作】
  - 输入输出重定向成为可能
- 缓冲区管理
- 假脱机(spooling)

缓冲区：为什么需要缓冲区？

- 考虑如下例子
  - 用户程序需要从调制解调器读取数据
  - 调制解调器收到一个字符后产生一个中断




```
用户程序
char c;
while(c=read()){
  deal_with(c);
}
```

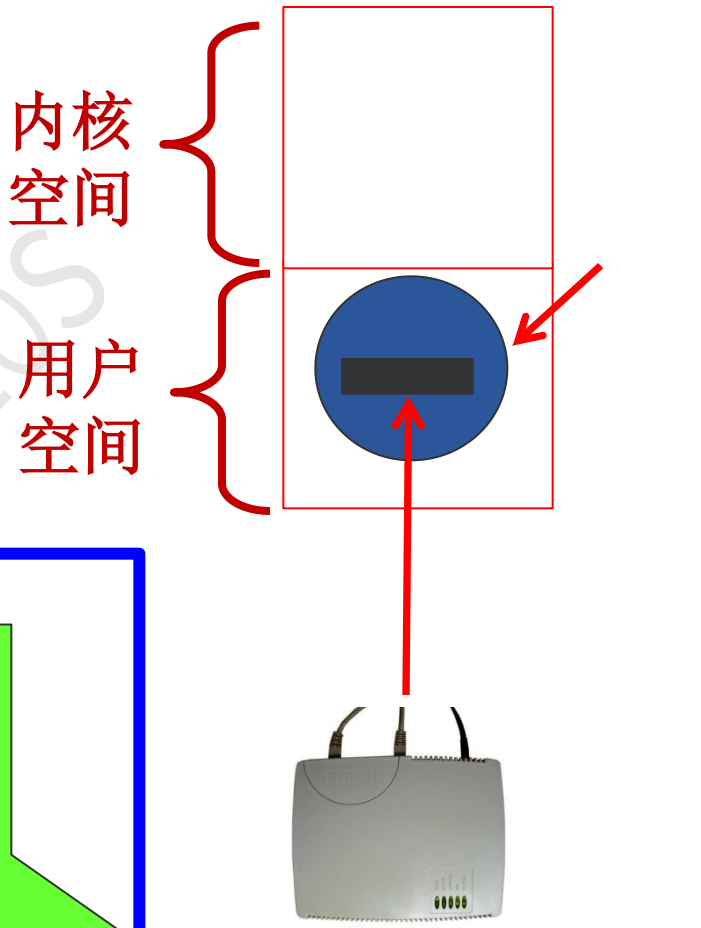
```
char read(){
    IO_control();
    block( );
    return IO_read();
}
```

请写出中断处理程序

过于低效

用户区的缓冲区

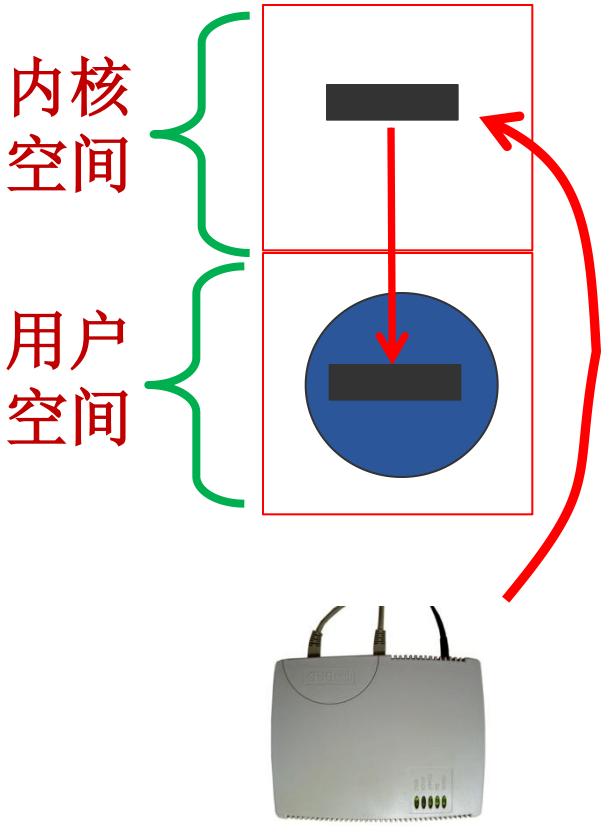
- 程序指定用户缓冲区
- 中断服务程序将收到数据放入缓冲区
- 收到n个字符后唤醒进程
- 缺点?



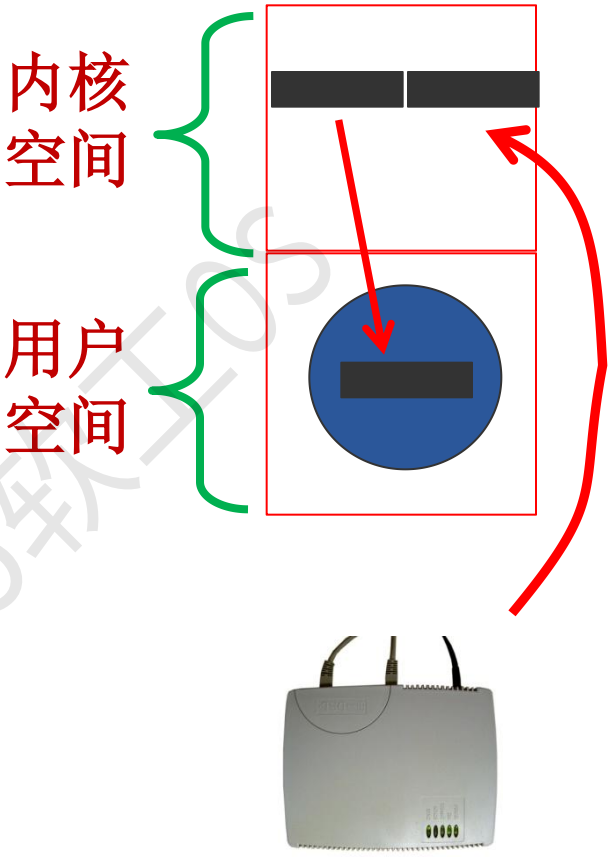

```
用户程序
char c[n];
read(c,n)
```

```
read(char *c, int n){
    IO_control();
    block( );
    return;
}
尝试写出中断处理程序
```

内核区建立缓冲区



内核中建立缓冲区，随后
复制到用户空间缓冲区



内核中建立双缓冲区

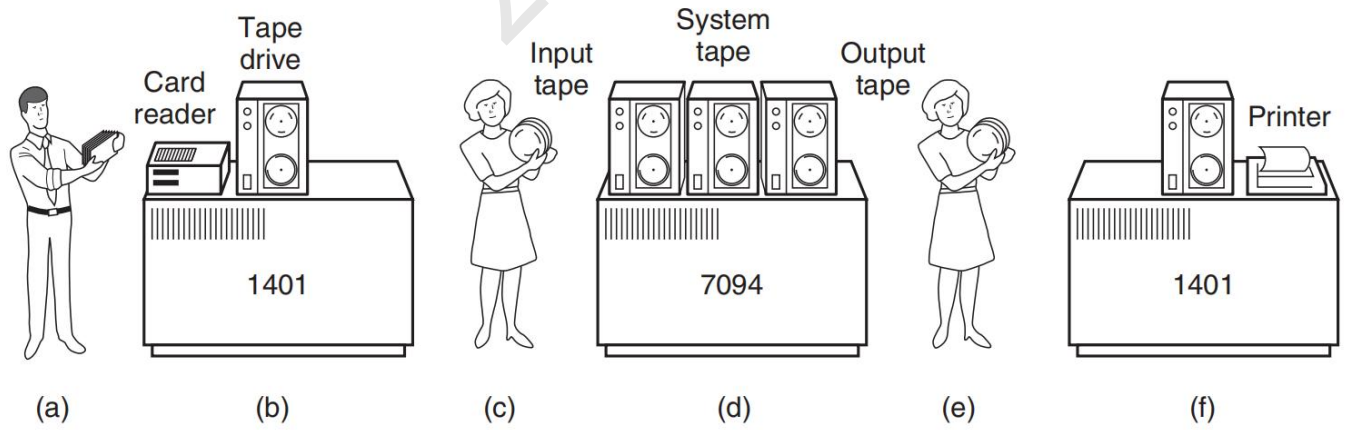
关于缓冲区

- 单缓冲区、双缓冲区都有应用
- 还有一种环形缓冲区
- 输出也可以使用缓冲区

设备独立软件之 spooling

为了效率，从联机到脱机

- 联机
  - 打孔纸带--->计算机--->打孔纸带
- 脱机
  - 打孔纸带--->外围控制机-->磁带
  - 磁带-->计算机-->磁带
  - 磁带--->外围控制机--->打孔纸带

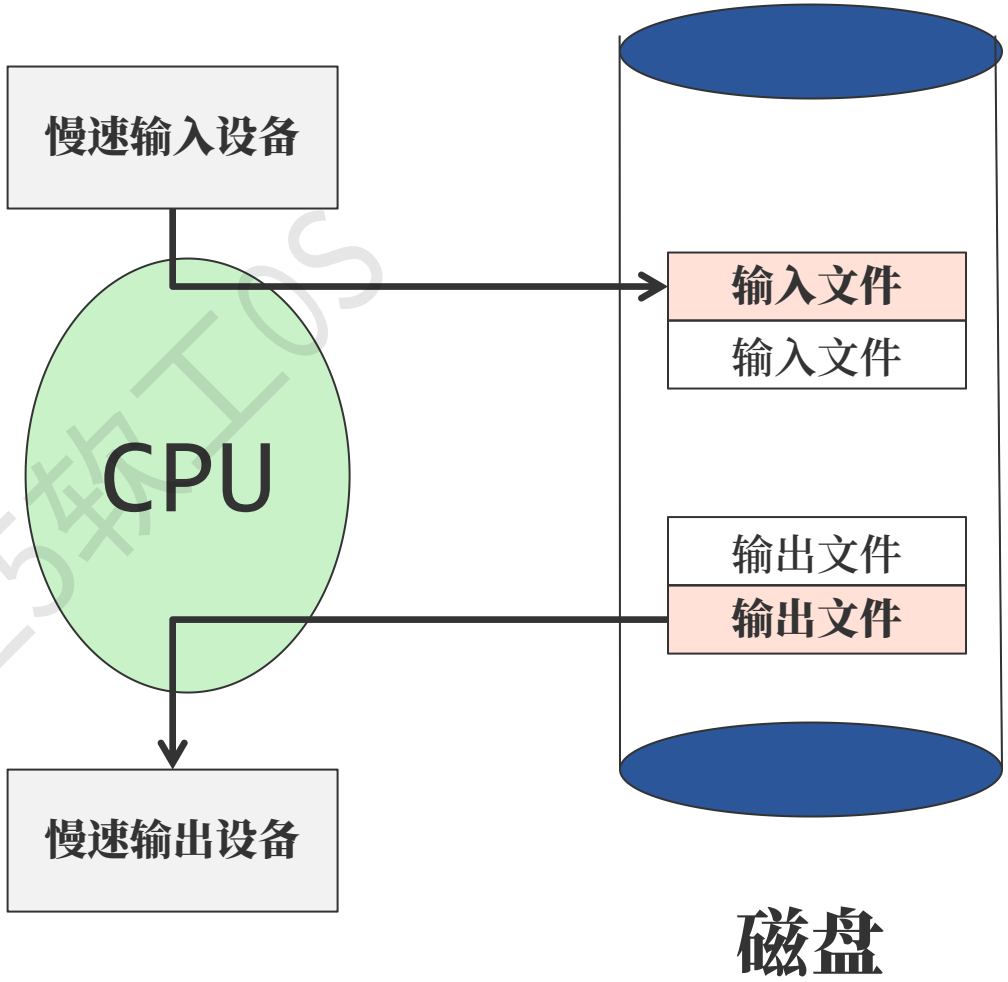


还是为了效率，从脱机到假脱机spooling

- 假脱机

  spooling：用一道程序代替外围控制机

  - 缓冲区
    - 数据交给慢速设备前，放入磁盘缓冲
  - 队列
    - 排队使用，像是脱机一样



假脱机spooling

- Spooling技术，是做什么的，有什么优点?
  - 中文名：假脱机技术，Simultaneous Periphernal Operations Online,把一台IO设备虚拟成多台
  - 联机---（太慢）--->脱机
    - 联机：主机参与的IO
    - 脱离主机：没有主机参与的情况下完成IO，利用外围控制机将数据IO进磁盘
  - 脱机--（多道程序设计）--->假脱机
    - 假脱机：没有外围设备参与的脱机，用软件模拟外围控制机，实际上还是联机，但与开始的联机不同
  - 主要应用领域：打印服务，共享打印机

输入输出管理

- IO硬件
  - IO硬件的发展、构成
  - IO设备的控制方式
- IO软件
  - 中断处理程序、驱动程序
    - xv6驱动程序案例
  - 设备独立软件
  - 用户层IO软件
- 磁盘
  - 结构
  - 磁盘调度方法

用户层IO软件

- 系统调用
  - open, read，write，close, ioctl，等
- 库函数
  - glibc
  - fopen, fread, fwrite, fclose
  - 带了缓冲区，速度差异大，同时也引起一些问题

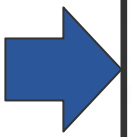
```
void main(){
  printf("a");
  fork();
  printf("b");
}
```

```
ubuntu@VM-16-15-ubuntu:~$
ab
ab
```

用户层IO软件

- 系统调用
  - open, read，write，close, ioctl，等
- 库函数
  - glibc
  - fopen, fread, fwrite, fclose
  - 带了缓冲区，速度差异大，同时也引起一些问题

```c
void main(){
    write(1,"a",1);
    fork();
    write(1,"b\n",2);
}
```

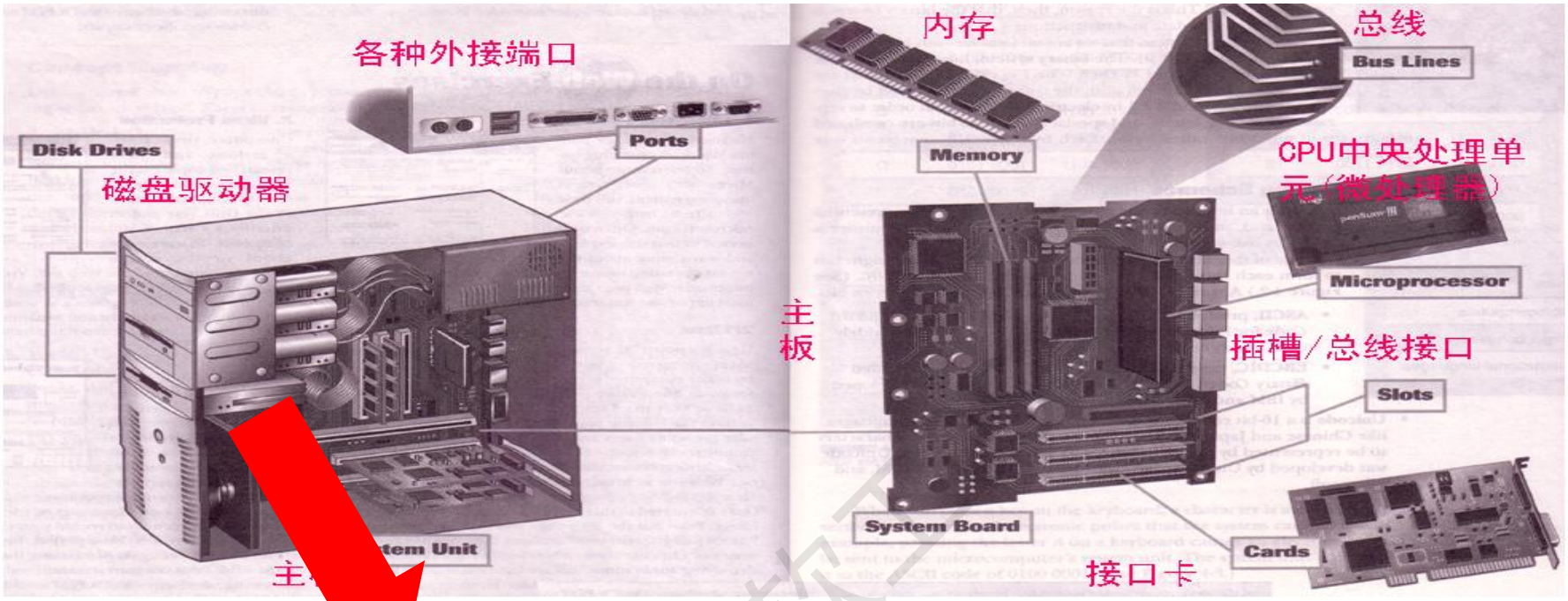


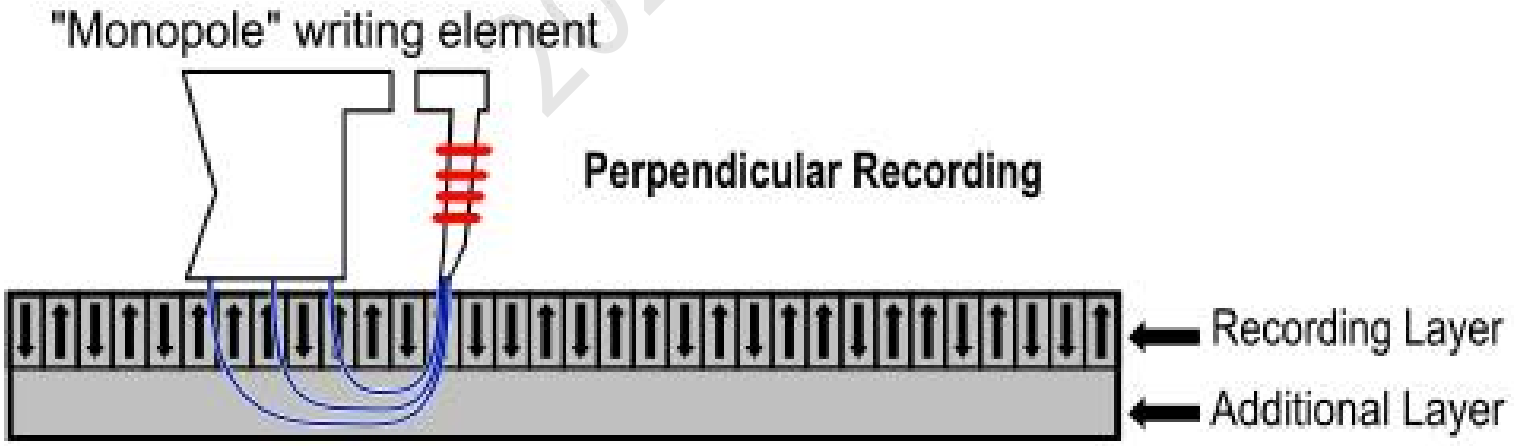
```
ubuntu@VM-16-15-ubuntu:
ab
b
```

输入输出管理

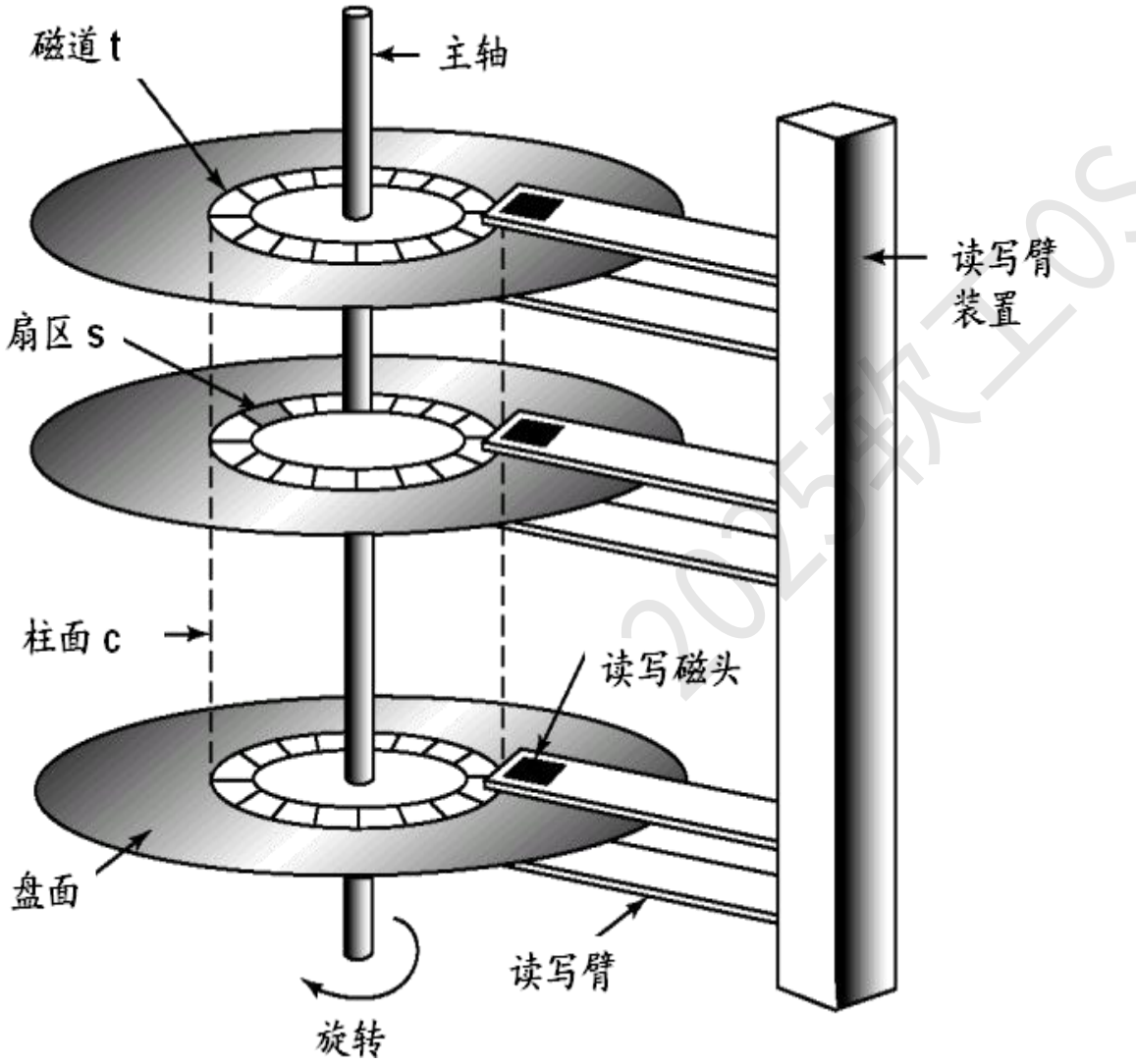
- IO硬件
  - IO硬件的发展、构成
  - IO设备的控制方式
- IO软件
  - 中断处理程序、驱动程序
    - xv6驱动程序案例
  - 设备独立软件
  - 用户层IO软件
- 磁盘
  - 结构
  - 磁盘调度方法

磁盘设备


各种外接端口
Ports
Disk Drives
磁盘驱动器
主
板
内存
Memory
总线
Bus Lines
CPU中央处理单元（微处理器）
Microprocessor
插槽/总线接口
Slots
System Board
Cards
接口卡


"Monopole" writing element
Perpendicular Recording
Recording Layer
Additional Layer

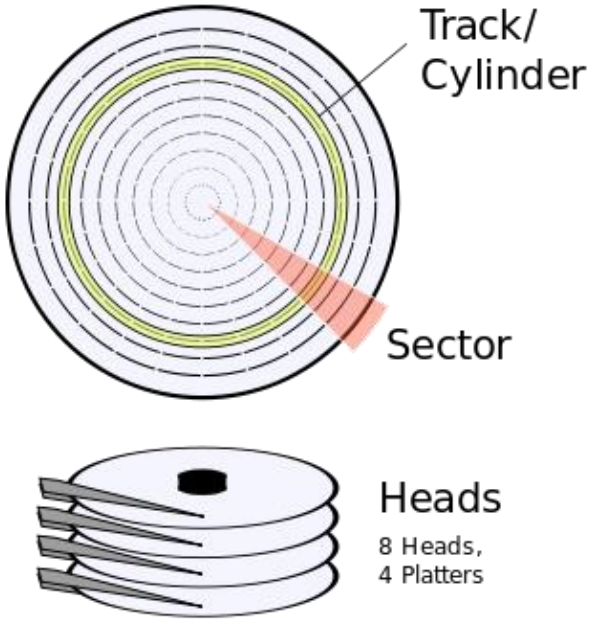
CHS寻址方式 磁道:盘面:扇区



一个扇区存放512个字节。

现代OS不使用CHS

- 早期OS使用CHS寻址方式
  - 三维地址
- 技术发展后，不同磁道的扇区数不同
  - 临时解决方案：remapping
  - 地址位数确定后容量不易扩展



LBA寻址方式

- logical block address
- 寻址的基本单位是块（block）
- 地址线性增长，0,1,2,3，。。。
- 块的大小是扇区大小的整数倍。
  - ➢假如移动硬盘的大小是1TB，地址28位，则每块至少需要多大?
- 在OS看来，一块磁盘可以当成block类型的一维数组。（内存呢？）

磁盘（磁臂）调度算法

（回想以前一系列调度算法）

背景：从磁盘上读写数据

- 移动磁头到指定磁道上
  - 寻道时间
- 等待磁盘，直到需要读写的扇区在磁头下面
  - 旋转延迟时间
- 传输数据
  - 把数据从磁盘读出或写入
- 前两者的耗时较长
- 磁盘调度问题的目的是减少寻道时间
  - 更确切地说是磁臂调度问题

磁臂调度问题

- 假设给定一组磁盘读写请求访问的磁道是
  - 55,58,39,18,90,160,150,38,18
  - 当前磁头处于100号磁道上
- 问:应当如何满足这些访问磁盘的请求，使得
  - 磁臂的移动距离尽量短
  - 尽量公平（至少不发生饿死）

磁臂调度算法：先来先服务FCFS

- 公平、低效
- 应用于前面的例子，则为
  - 55,58,39,18,90,160,150,38,18
- 磁臂移动距离（寻道长度）
  - 总距离：45+3+19+21+72+70+10+112+20

回顾：输入输出管理

- IO硬件
  - IO硬件的发展、构成（CPU和硬件的寄存器交互，特殊IO指令、内存映射IO）
  - IO设备的控制方式（中断、轮询、DMA）
- IO软件
  - 中断处理程序、驱动程序（硬件相关）
    - xv6驱动程序案例
  - 设备独立软件（硬件无关、缓冲区、Spooling）
  - 用户层IO软件
- 磁盘
  - 结构（盘面、磁道、扇区，CHS寻址方式，LBA寻址方式）
  - 磁盘调度方法（磁盘的读取过程、磁臂移动距离、FCFS）

磁臂调度算法：最短寻道时间优先SSTF

- 服务距离当前磁头最近的请求
- 高效、不公平（磁臂易停留在中间区域）
- 应用于前面的例子，为

  90,58,55,39,38,18,150,160,184
- 磁臂移动距离
  - 总距离：10+32+3+16+1+20+132+10+24

磁臂调度算法：电梯调度算法SCAN

- 算法介绍
  - 沿着某个方向移动，直到该方向没有请求，逆转方向
  - 需要提前告知磁头移动方向
- 应用于前面的例子
  - 55,58,39,18,90,160,150,38,18
- 假设先向磁道增加的方向移动
  - 150,160,184,90,58，55,39,38,18
- 磁臂移动距离
  - 总距离：50+10+24+94+32+3+16+1+20

磁臂调度算法：循环扫描算法CSCAN

- 算法介绍
  - 只沿一个方向移动（没有逆转的操作）
  - 折中，降低最坏情况下的时延
- 应用于前面的例子，假设向磁道增加的方向扫描
  - 150,160，184,18,38,39,55,58,90
- 磁臂移动距离
  - 总距离：50+10+24+166+20+1+16+3+32

磁臂调度算法

- 先来先服务（FCFS）
  - 公平、低效
- 最短寻道时间优先（SSTF）
  - 高效、不公平
- 电梯调度算法（SCAN）
  - 沿着某个方向移动，直到该方向没有请求，逆转
- 循环扫描算法（CSCAN）
  - 只沿一个方向移动（没有逆转的操作）
  - 折中，降低最坏情况下的时延

输入输出系统总结（2025考研大纲）

- 设备的基本概念
  - 设备的基本概念、IO接口、IO端口
- IO控制方式
  - 轮询方式、中断方式、DMA方式
- IO软件层次结构
  - 中断处理程序，驱动程序，设备独立软件，用户层IO软件
- 设备独立软件（缓冲区管理、假脱机）
- 外存管理（磁盘结构，磁盘调度方法）