

操作系统

Operating Systems

L26 I/O与显示器

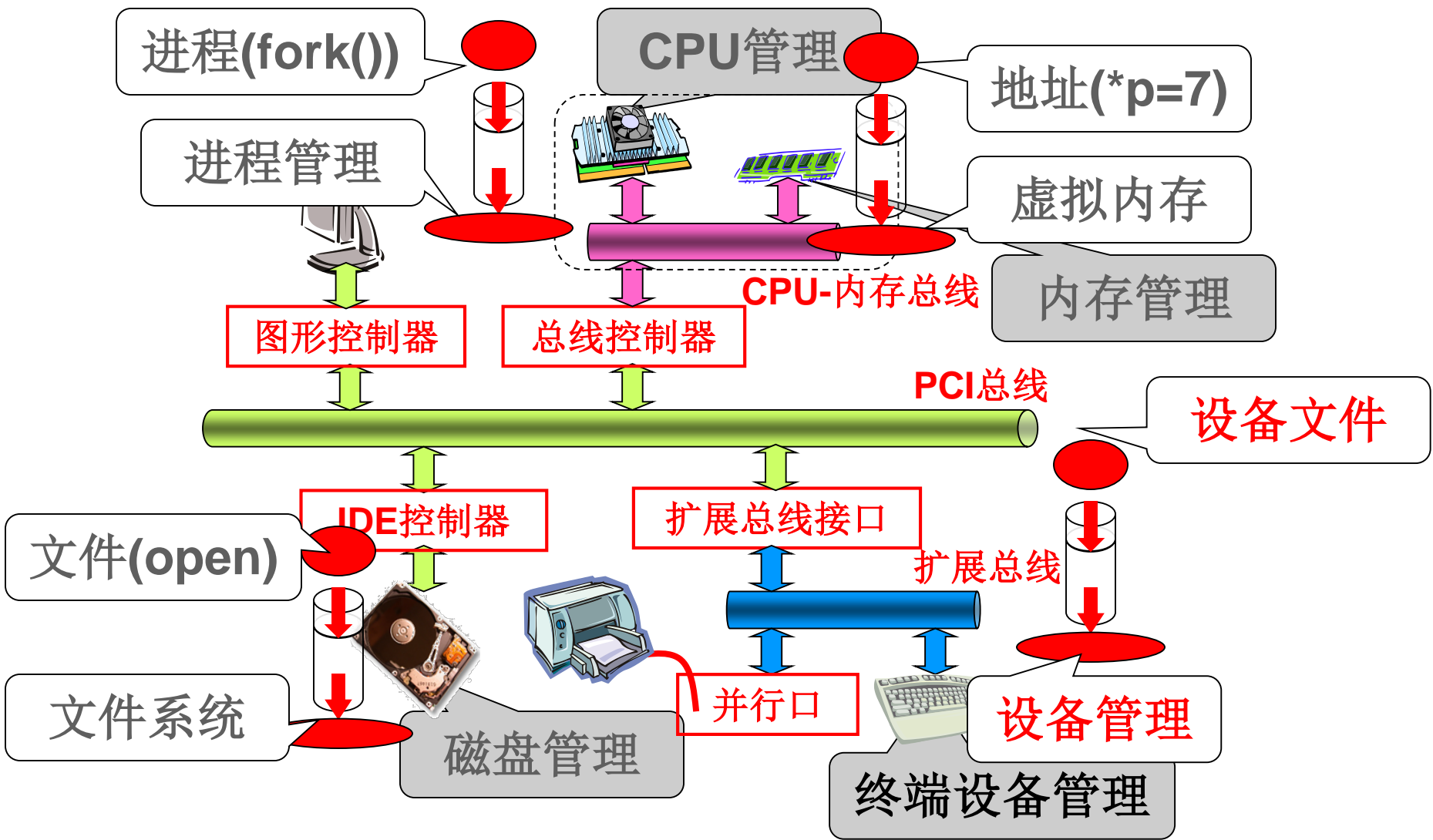
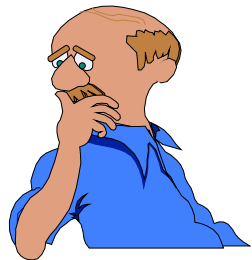
printf(Display)

授课教师：李治军

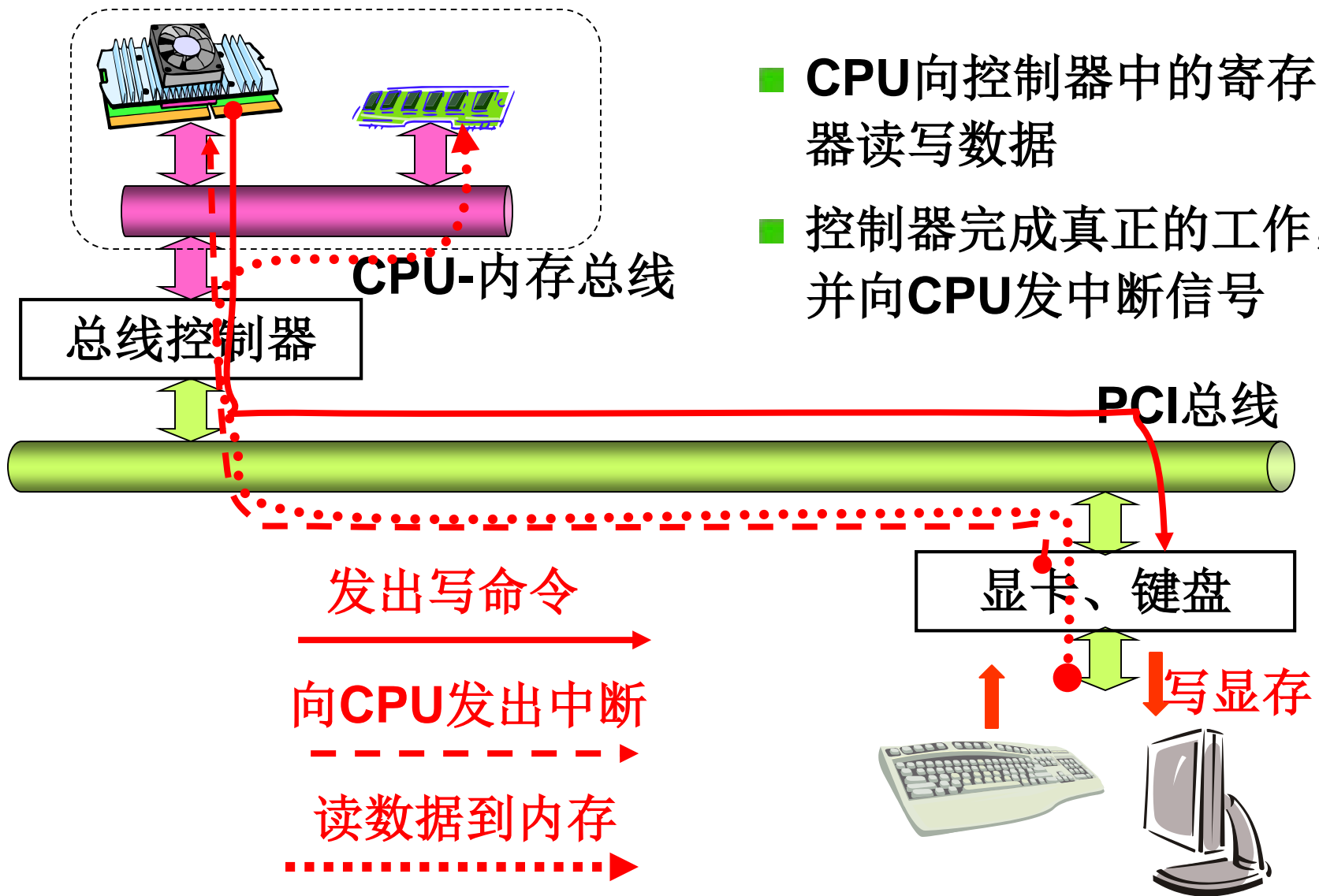
lizhijun_os@hit.edu.cn

综合楼411室

继续那台“计算机”

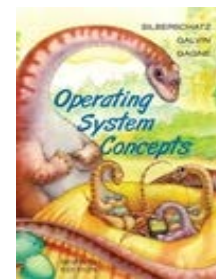


让外设工作起来

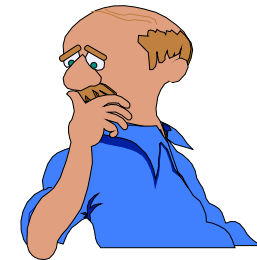


向设备控制器的寄存器写不就可以了吗？

需要查寄存器地址、内容的格式和语义...操作系统要给用户提供一个简单视图—**文件视图**，这样方便



一段操纵外设的程序



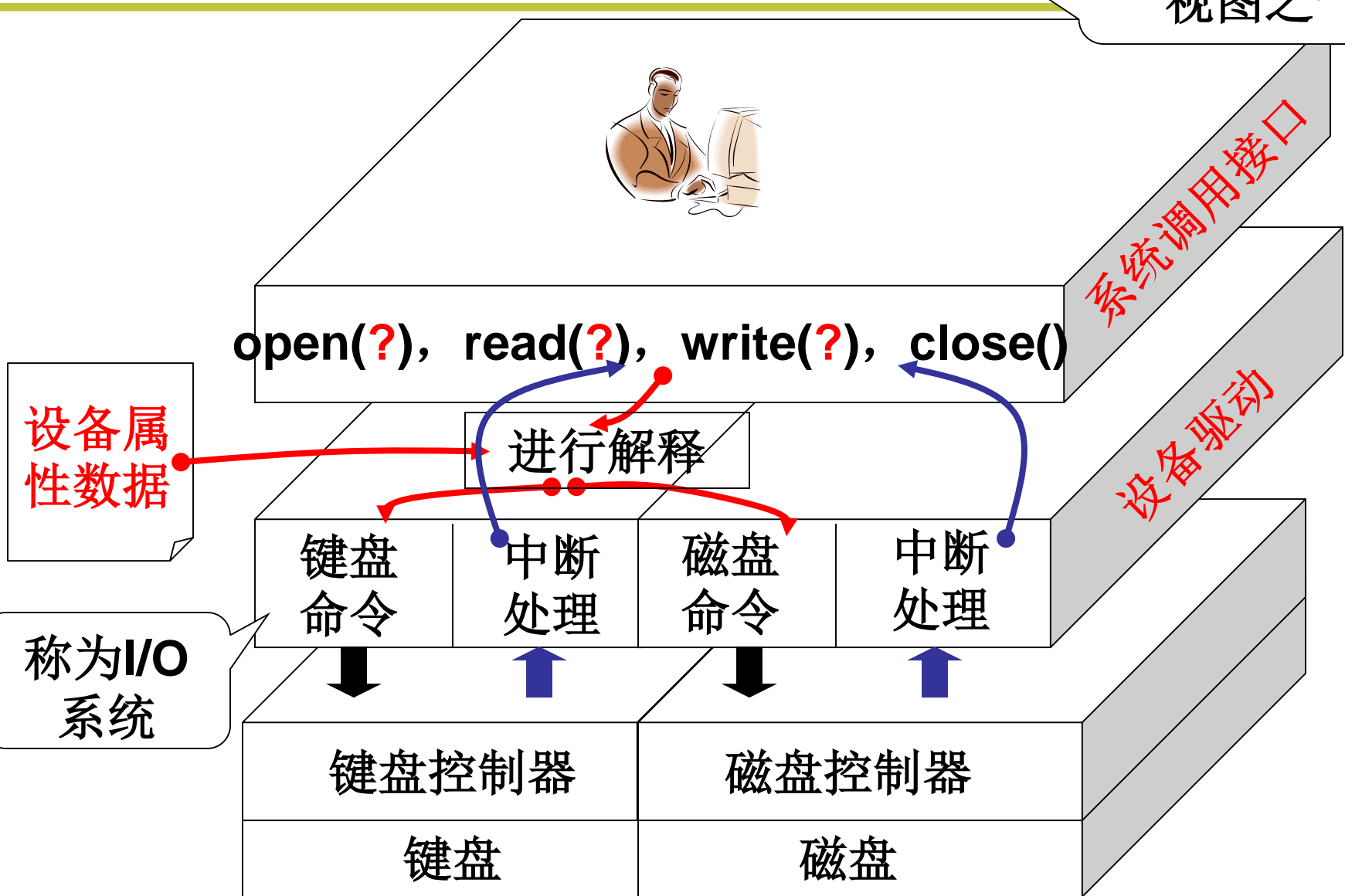
```
int fd = open("/dev/xxx");  
for (int i = 0; i < 10; i++) {  
    write(fd, i, sizeof(int));  
}  
close(fd);
```

- (1) 不论什么设备都是open, read, write, close
操作系统为用户提供统一的接口!
- (2) 不同的设备对应不同的设备文件(/dev/xxx)
根据设备文件找到控制器的地址、内容格式等等!



一个统一的视图-文件视图

操作系统两大
视图之一



概念有了，开始给显示器输出...

■ 从哪里开始这个故事呢？

printf("Host Name: %s", name);

- **printf**库展开的部分我们已经知道：先创建缓存**buf**将格式化输出都写到那里，然后再**write(1,buf,...)**

在linux/fs/read_write.c中

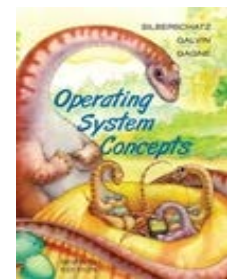
```
int sys_write(unsigned int fd, char *buf,  
int count)
```

```
{ struct file* file;  
  file = current->filp[fd];  
  inode = file->f_inode;
```

fd是找到file的索引!

current不陌生吧，进程
带动整个系统的视图

- **file**的目的是得到**inode**，显示器信息应该就在这里



fd=1的filp从哪里来？

- 因为是被**current**指向，所以是从**fork**中来

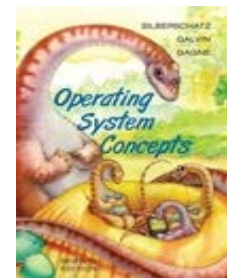
```
int copy_process(...){  
    *p = *current;  
    for (i=0; i<NR_OPEN;i++)  
        if ((f=p->filp[i])) f->f_count++;  
}
```

- 显然是拷贝来的，那么是谁一开始打开的？

shell进程启动了**whoami**命令，**shell**是其父进程

```
void main(void)  
{ if(!fork()){ init(); }  
}
```

```
void init(void)  
{ open("dev/tty0", O_RDWR, 0); dup(0); dup(0);  
  execve("/bin/sh", argv, envp) }
```



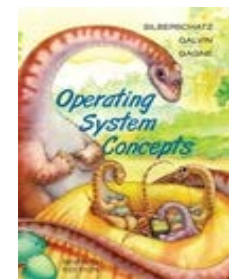
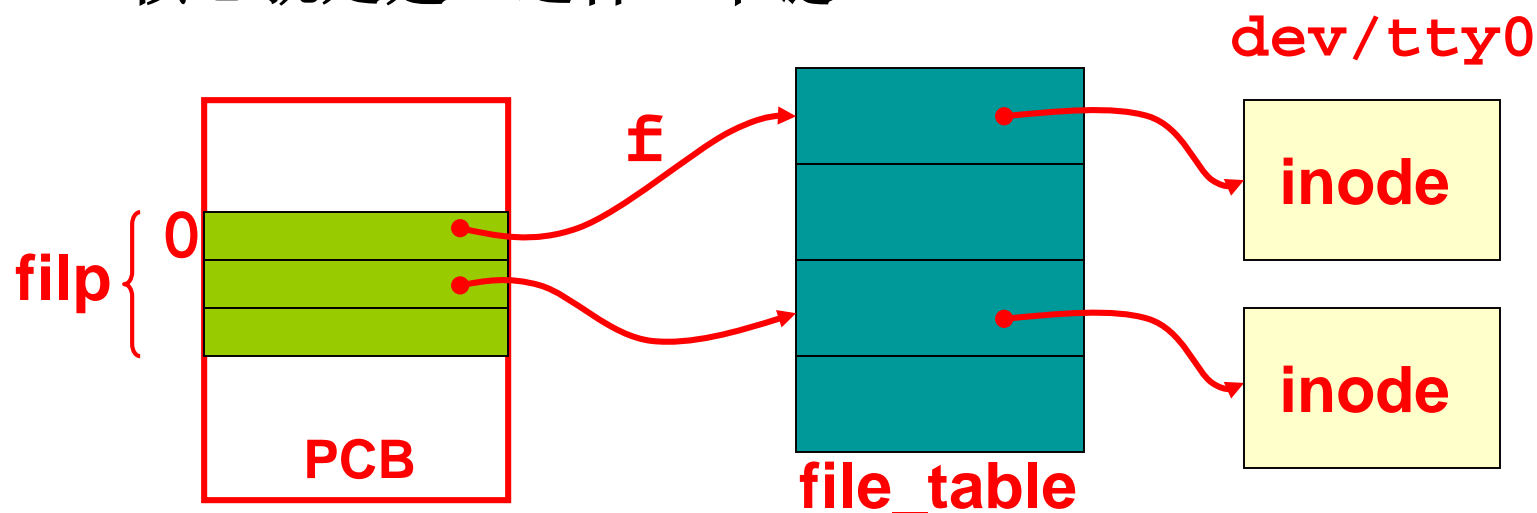
open系统调用完成了什么？

在linux/fs/open.c中

```
int sys_open(const char* filename, int flag)
{
    i=open_namei(filename,flag,&inode);
    cuurent->filp[fd]=f; //第一个空闲的fd
    f->f_mode=inode->i_mode; f->f_inode=inode;
    f->f_count=1; return fd; }
```

解析目录，找到inode!

- 核心就是建立这样一个链



准备好了，真正向屏幕输出！

■ 继续sys_write!

在linux/fs/read_write.c中

```
int sys_write(unsigned int fd, char *buf, int cnt)
{ inode = file->f_inode;
  if(S_ISCHR(inode->i_mode))
    return rw_char(WRITE, inode->i_zone[0], buf,
cnt); ...
```

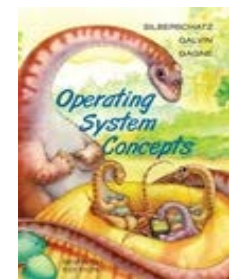
/dev/tty0的inode中的
信息是字符设备

```
[/dev]# ls -l
crw-rw-rw-  1 root    root      4,  0 Mar  4  2004 tty0
```

■ 转到rw_char!

在linux/fs/char_dev.c中

```
int rw_char(int rw, int dev, char *buf, int cnt)
{ crw_ptr call_addr=crw_table[MAJOR(dev)];
  call_addr(rw, dev, buf, cnt); ...}
```



看看crw_table!

第4个!

```
static crw_ptr crw_table[]={...,rw_ttyx,};  
typedef (*crw_ptr)(int rw, unsigned minor, char  
*buf, int count)
```

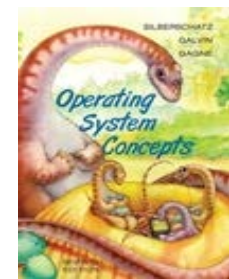
```
static int rw_ttyx(int rw, unsigned minor, char  
*buf, int count)  
{ return ((rw==READ)? tty_read(minor,buf):  
    tty_write(minor,buf));}
```

■ 再转到**tty_write**! //实现输出的核心函数

在linux/kernel/tty_io.c中

```
int tty_write(unsigned channel,char *buf,int nr)  
{ struct tty_struct *tty;tty=channel+tty_table;  
    sleep_if_full(&tty->write_q);  
    ...}
```

可以猜测:输出就是
放入队列!



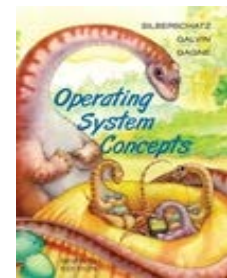
继续tty_write这一核心函数

在linux/kernel/tty_io.c中

```
int tty_write(unsigned channel, char *buf, int nr)
{ ...
    char c, *b=buf;
    while(nr>0&&!FULL(tty->write_q)) {
        c = get_fs_byte(b);
        if(c=='\r'){PUTCH(13,tty->write_q);continue;}
        if(O_LCUC(tty)) c = toupper(c);
        b++; nr--; PUTCH(c,tty->write_q);
    } //输出完事或写队列满!
    tty->write(tty);
}
```

fs:从用户缓存区读!

■ **tty->write**应该就是真的开始输出屏幕了!



看看tty->write

在include/linux/tty.h中

```
struct tty_struct{ void (*write)(struct tty_struct
*tty); struct tty_queue read_q, write_q; }
```

■ 需要看tty_struct结构的初始化!

```
struct tty_struct tty_table[] = {
{ con_write, {0,0,0,0,""}, {0,0,0,0,""} }, {}, ...};
```

■ 到了con_write, 真正写显示器!

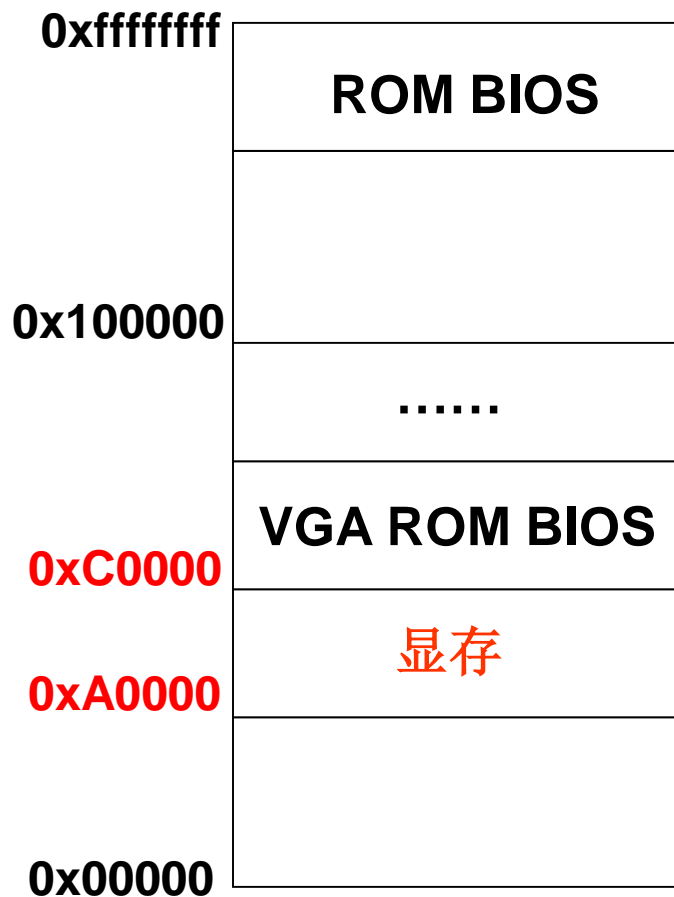
在linux/kernel/chr_drv/console.c中

```
void con_write(struct tty_struct *tty)
{
    GETCH(tty->write_q,c);
    if(c>31&& c<127){__asm__( "movb _attr,%%ah\n\t"
        "movw %%ax,%1\n\t"::"a"(c),
        "m" (*(short*)pos):"ax"); pos+=2;}
```



只有一句话: `mov pos`

PC/AT机内存区域图



■ 完成显示中最核心的秘密就是

`mov pos, c`

■ `pos`指向显存: `pos=0xA0000`

```
con_init();
```

```
void con_init(void)
{ gotoxy(ORIG_X, ORIG_Y); }
```

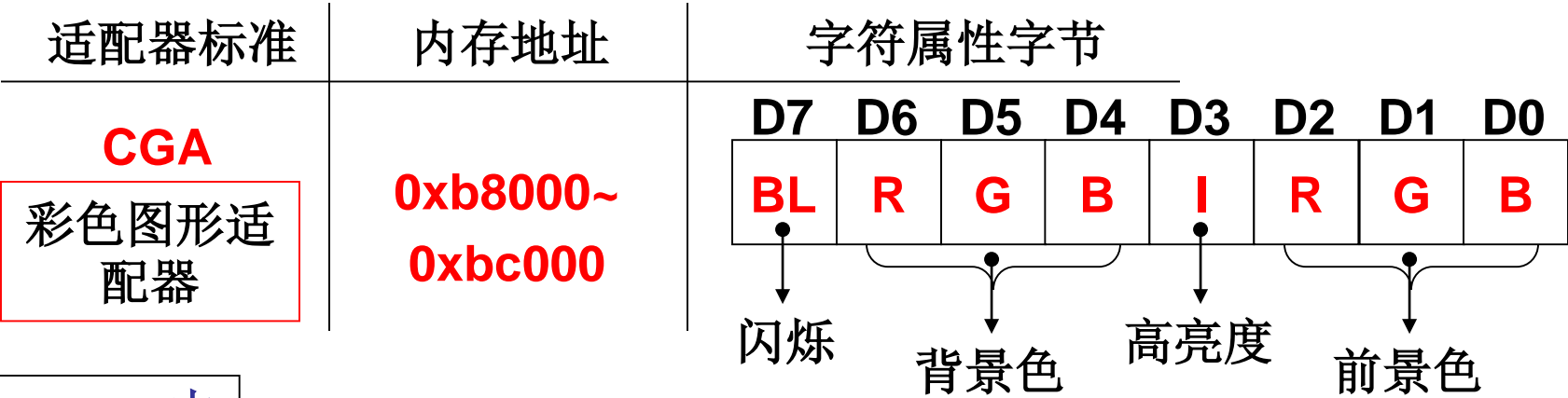
```
static inline void gotoxy()
{ pos=origin+y*video_size_row
  +(x<<1); }
```

```
#define ORIG_X (*(unsigned char*)0x90000) //初始光标列号
#define ORIG_Y (*(unsigned char*)0x90001) //初始光标行号
```



pos的修改

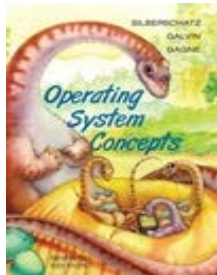
- pos的修改: `pos+=2` 为什么加2?
- 屏幕上的一个字符在显存中除了字符本身还应该有什么属性(如颜色等)



console.c中

```
Static unsigned char attr=0x07;  
__asm__( "movb _attr,%%ah\n\t" "movw %%ax,  
%1\n\t"::"a"(c),"m"(*(short *)pos):"ax");
```

黑底白字!



printf的整个过程!

