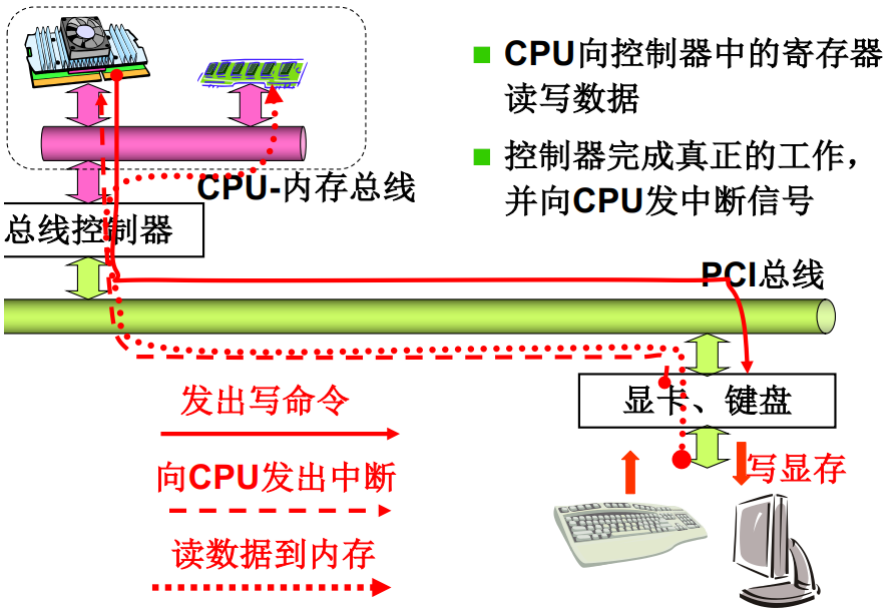
哈工大操作系统-L26IO与显示器

计算机是如何使用外设的?

1.让外设工作的三件事



- 对外设的使用，实际上就是**对外设的控制器发指令。**(可能是一堆out指令)
- 外设工作完成之后，进行中断处理就行。
- **为了让使用外设变得简单，需要提供一个统一的视图--文件视图。**

2.一个向屏幕输出的实例

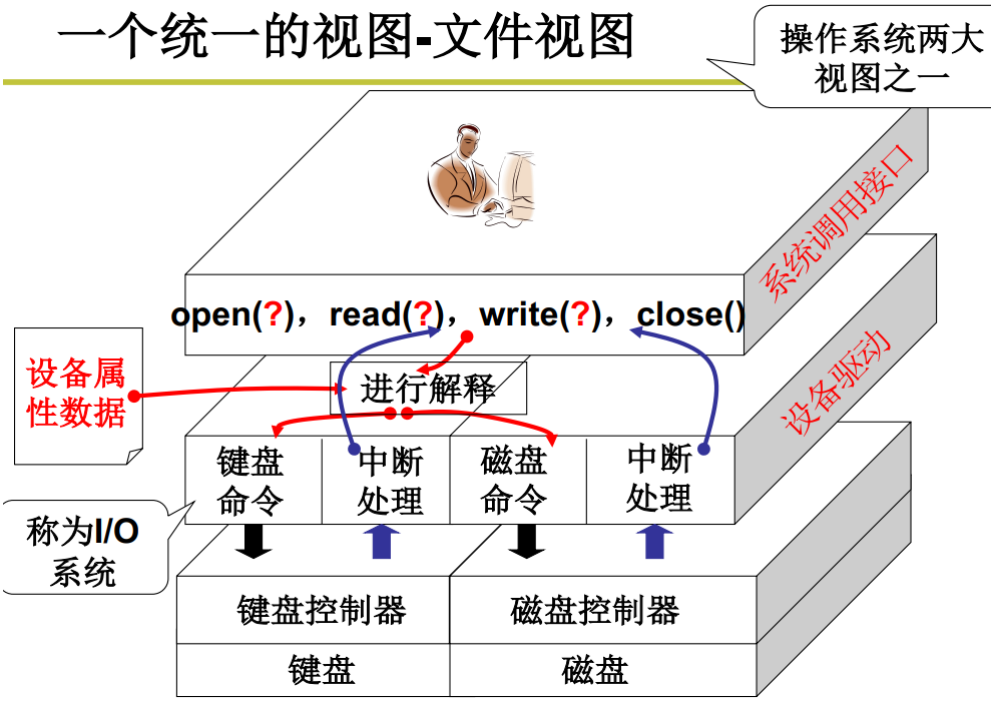
一段操纵外设的程序

```
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!

2.1文件视图--统一的视图

一个统一的视图-文件视图



2.2如何知道输出的是什么设备?

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,...)

1决定了向显示器输出，那么是如何得到这个1的

```
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

得到当前进程的inode

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)}
```

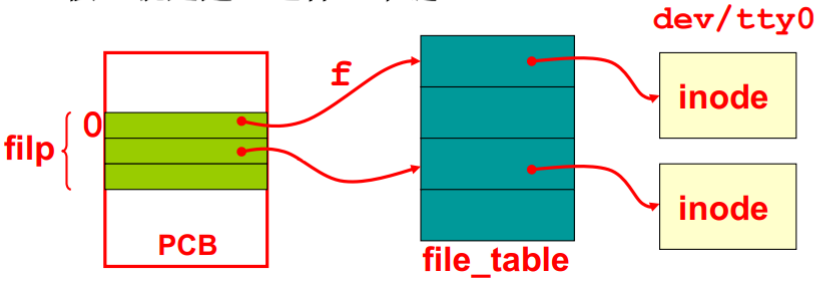
从父进程拷而来，因此filp[1]也是为filp[0]为dev/tty0

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



2.3如何向外部写

准备好了，真正向屏幕输出!

- 继续sys_write!

sys write 会检查 inode中是什么设备 根据不同的设备 调用不同函数

```
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

检查是否是字符设备，是字符设备，就调用 rw char 读写字符设备

```
[/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);  ...}
```

又查表，根据表格信息调用不同的函数

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));}
```

先写到缓冲里面，再写到IO

因为CPU快，IO慢，因此先写到缓冲，然后中断写IO

- 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);
  ...}
```

队列满 进程睡眠

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!

把buf中的东西放到tty的写队列write_q中

- tty->write应该就是真的开始输出屏幕了!

使用tty结构体的函数进行输出

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;}
```

高字节是属性

低字节是字符

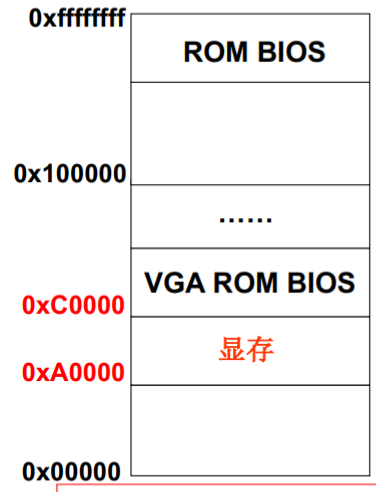
往某个位置上输出东西

因此写设备驱动，就是做好这些读写函数，并且把一些信息注册到注册表里面。

2.4输出的位置

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_ro
w +(x<<1);}
```

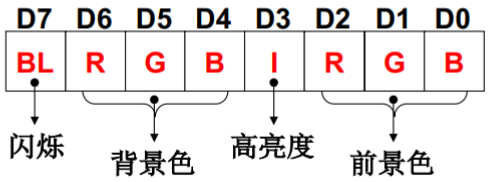
```
#define ORIG_X (*(unsigned char*)0x90000) //初始光标列号
#define ORIG_Y (*(unsigned char*)0x90001) //初始光标行号
```

我们在setup.s的那节课学过，把指针位置保存在90000和90001的位置

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字符的属性(如颜色等)

适配器标准	内存地址	字符属性字节
CGA 彩色图形适配器	0xb8000~ 0xbc000	D7 D6 D5 D4 D3 D2 D1 D0



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!

3.把一系列流程包装成一个统一的视图

printf的整个过程!