因为不是科班出身，写的源码不好懂，所以附上额外的“注释”，我会分几次写完，发给大家。  
  >>>>>>>>>>>>>>>  
  papaya内核源码分析（一）  
  具体介绍之前，我先简述各个模块的功能，它们都放在src文件夹里：  
1,boot.asm        MBR的源文件  
2,kernel.asm        内核的入口文件，负责初始化相关硬件  
3,kernel.c        开启分页机制，启动相关服务进程，从kernel.c以后，papaya尽量都用c实现  
4,mm.c        内存管理模块，提供页错误中断例程，页映射函数。  
5,fs.c            各类文件系统的抽象层  
5.5,fs\_ext.c   只读的ext2文件系统  
6proc.c        进程调度模块  
7,proc.asm        进程调度模块  
8,tty.c            终端，集成了简单的shell  
9,hs.c            IDE磁盘驱动，提供读写接口  
10,kbd\_drv.c        键盘驱动  
11,video\_drv.c        文本模式下，简单的屏幕驱动  
12,utils.asm,utils.c,ku\_utils.c        工具性代码如memcp，或硬件设置代码如init8259A  
13,mw,struinfo.c        用于调试，他俩让c语言支持简单的反射，方便打印结构体  
14,func\_table.c        系统调用接口  
15,disp.c            实现一个printf函数  
1,MBR(boot.asm)  
  bochs虚拟机从从硬盘映像400m.img引导，硬盘映像的MBR有3个功能：  
  1,跳入保护模式，看源码：  
-------------------------------------------------------------------------------------------------------  
     7    [section .gdt]                 ；准备一个精简的gdt，仅为cs和ds准备段描述符  
     8    gdt:Descriptor 0,0,0   
     9    .desc\_plain\_c0:Descriptor 0,0fffffh,DA\_32+DA\_C+DA\_LIMIT\_4K   
    10    .desc\_plain\_d0:Descriptor 0,0fffffh,DA\_DRW|DA\_LIMIT\_4K+DA\_32   
     …  …  
    60    ;load gdt   
    61    mov ax,0   
    62    mov ds,ax   
    63    lgdt [gdtPtr]                 ；gdtr寄存器指向预设的global descriptor table了  
       ...  ...  
    70    switch\_proMode:   
    71        mov eax,cr0   
    72        or eax,1   
    73        mov cr0,eax            ；置PE位，cpu已经运行在保护模式下，寻址机制已切换   
         
    74    mov ax,selector\_plain\_d0   
    75    mov ds,ax   
    76    jmp dword selector\_plain\_c0:base\_entrance\_kernel         ；为jmp提供48位目标地址指针，段选择子被装到cs，偏移量被装到eip，cs的投影寄存器也随之刷新。  
-------------------------------------------------------------------------------------------------------  
  2,担当loader的角色  
  400m.img也就是系统的启动盘，在它的mbr（即第0个扇区）和第一个分区之间有1M的空白，内核镜像文件就写到这1M的空间里：  
  system("dd if=../bin/kernel.elf of=../cmd/400m.img bs=512 conv=notrunc seek=1");  
  kernel.elf被刻录到1号扇区以后的硬盘空间。  
  papaya内核没有专门的loader，加载内核也是由MBR完成的。看源码：  
-------------------------------------------------------------------------------------------------------  
    ；第一阶段，加载内核文件，暂时存放在0x80000处  
    ；用扩展的int 13H中断  
dap:  
    db 16        ;packetsize ==16  
    db 0        ;reserved ==0  
    dw 120\*2        ;sector count,=120KB  
    dw 0        ;offset  
    dw 0x8000    ;seg  
    dd 1        ;start-sector  
    dd 0  
entrance:  
mov ax,0  
mov ds,ax  
  
mov ah,42h  
mov dl,80h  
mov si,dap  
int 13h  
    ；第二阶段，根据elf文件的格式，装载kernel.elf到正确位置：  
resetKernel:  
;load kernel from 0x8000h  
mov ax,base\_kernel\_loaded  
mov ds,ax  
mov bx,0  
mov  dx,[bx+42];e\_phentsz  
mov  cx,[bx+44];e\_phnum  
mov  bx,[bx+28];ERR e\_phoff should be 32-bit,here ignore high-16 bit  
search\_ph\_typeLoaded:  
    cmp dword [bx],1  
    jne doNothing      
cySegment:  
    push cx  
    mov cx,[bx+16]  
    mov si,[bx+4]    ;segment offset,ds=base\_kernel\_loaded             ERR ignore high-16 bit      
    mov eax,[bx+8]  
    mov di,ax  
    and di,1111b    ;get bit 0~3  
    shr eax,4  
    mov es,ax  
      
    cld  
    rep movsb     ;ERR kernel should be smaller than 64kb,for we use cx as loop-seed  
    pop cx              
doNothing:  
    add bx,dx  
loop search\_ph\_typeLoaded  
    ；代码注释中出现ERR的，表示该处不够健壮。  
-------------------------------------------------------------------------------------------------------  
3,打开A20地址总线：  
-------------------------------------------------------------------------------------------------------  
    65    cli   
    66    openA20:   
    67        in al,92h   
    68        or al,00000010b   
    69        out 92h,al  
-------------------------------------------------------------------------------------------------------  
  MBR的3个功能说完了，它的的最后一条指令是：jmp dword selector\_plain\_c0:base\_entrance\_kernel ，“base\_entrance\_kernel”是宏，指定内核入口在0x30400。  
  最后一条指令执行完后，系统的控制权就从MBR转交到系统内核。我们再看看内核初始化模块kernel.asm。  
  
2,内核初始化模块（kernel.asm)  
1,切换到新的gdt  
  MBR虽然把cpu切换到保护模式，但体积所限（毕竟才512个字节），只提供两个描述符，kernel的第一个任务是把GDTR切换到一个完善的global descriptor table，我们看新的全局描述符表：  
-------------------------------------------------------------------------------------------------------  
  ;Descriptor宏的原型：Descriptor base,limit,attr。  
  ;这种宏机制被nasm的预处理器支持，它会把其后给出的base,limit,attr三个参数“打散”，塞到64位描述符的正确位置。  
   327    gdt:   
   328    .desc\_empty: Descriptor 0,0,0   
   329    .desc\_plain\_c0:Descriptor 0,0fffffh,DA\_32|DA\_C|DA\_LIMIT\_4K +DA\_DPL0  
   330    .desc\_plain\_d0:Descriptor 0,0fffffh,DA\_DRW|DA\_LIMIT\_4K+DA\_32   
   331    .desc\_plain\_c1:Descriptor 0,0fffffh,DA\_32|DA\_C|DA\_LIMIT\_4K+DA\_DPL1   
   332    .desc\_plain\_d1:Descriptor 0,0fffffh,DA\_32|DA\_DRW|DA\_LIMIT\_4K+DA\_DPL1   
   333    .desc\_plain\_c3:Descriptor 0,0fffffh,DA\_C|DA\_LIMIT\_4K+DA\_DPL3+DA\_32   
   334    .desc\_plain\_d3:Descriptor 0,0fffffh,DA\_DRW|DA\_LIMIT\_4K+DA\_DPL3+DA\_32   
 -------------------------------------------------------------------------------------------------------  
  上面只列出来gdt的一部分，以第一个有效描述符.desc\_plain\_c0为例,其base=0,指定代码段基地址是0,这就是所谓的“平坦内 存”；limit=fffff,配合attr中的LIMIT\_4K（指定段粒度为4K），段界限达到4G。Attr里的DA\_32指定该段是32位，即指 令使用的操作数和地址默认32位或8位，DA\_C指定该段是代码段，DA\_DPL0指定该段需要cpu运行在rign0下。  
  第二个描述符,desc\_plain\_d0是给ds,es,fs,gs,ss用的，并没有专门的堆栈段描述符。  
  剩下的2组分别是ring1和ring3下的描述符，intel提供的4个ring，papaya用了3个。  
    
  更改gdtptr之后，就要刷新段寄存器了。ds,es,gs之类只需要mov操作，cs寄存器刷新是这样的：  
    54    jmp selector\_plain\_c0:newcs ;update cs   
    55    newcs:nop   
   对比kernel.asm和boot.asm中gdt的布局，对ds和cs的刷新似乎是不必要的，因为选择子恰好都是第2,3个，而且描述符组成也一样。这里刷新是为了提高代码可靠性，这样，将来可以轻易改动GDT。  
  
2,初始化8253  
-------------------------------------------------------------------------------------------------------   
    84    push  11931 ;1193180/100   
    86    call init8253   
    87    add esp,4  
-------------------------------------------------------------------------------------------------------  
  CMOS发出1193180hz的脉冲，设置8253控制字为11931,这样8253会以100hz频率向8259A发送中断请求。  
  系统时钟频率是100hz。  
  
3,初始化8259A  
-------------------------------------------------------------------------------------------------------   
    88    push 11111000b   
    89    call init8259A   
    90    add esp,4   
-------------------------------------------------------------------------------------------------------  
  向中断屏蔽寄存器写入11111000b，这样打开了时钟中断，键盘中断，并打开了级联从片。  
  
4,初始化IDTR和TR  
  -------------------------------------------------------------------------------------------------------   
   104    ;set idt   
   105    cli                 ；关闭中断  
   106    lidt [idtPtr]   
   107    mov ax,selector\_tss   
   108    ltr ax  
-------------------------------------------------------------------------------------------------------  
  和linux一样，papaya也只用一个tss走过场，这里永久的设定tr。  
5,跳入kernel.c模块  
    jmp kernel\_c;   
  
3,kernel.c  
  kernel.c继续做内核的初始化，只是“进化”到c语言实现。  
  1,  为所有的物理页建立对等映射，提供给内核使用。这样，内核在需要时，可以直接访问全部的物理内存。  
-------------------------------------------------------------------------------------------------------   
global\_equal\_map();  
-------------------------------------------------------------------------------------------------------  
  1.5,初始化各个模块  
-------------------------------------------------------------------------------------------------------   
    heap\_init();  
    proc\_init();  
    init\_fs();  
-------------------------------------------------------------------------------------------------------  
  2,打开分页机制。  
-------------------------------------------------------------------------------------------------------  
    24        \_\_asm\_\_("mov $0x101000,%eax\t\n"   
    25                "mov %eax,%cr3\t\n"   
    26                "mov %cr0,%eax\t\n"   
    27                "or $0x80000000,%eax\t\n"   
    28                "mov %eax,%cr0\t\n"   
    29                );  
-------------------------------------------------------------------------------------------------------  
  打开分页前，将cr3指向刚刚建立的对等映射表，页目录表存放在0x101000处。  
  
3,启动基本进程  
-------------------------------------------------------------------------------------------------------  
    create\_kernel\_process((int)&idle,9,0xffff,"idle",0);    //pid must =0  
    create\_kernel\_process((int)hs,2,0xffff,"hs",1);//hs的时间片要非常多，保证在下一轮时间片重置之前不会被挂起 ERR:pid must =1  
    create\_kernel\_process((int)fs\_ext,4,10,"fs\_ext",1);//pid must =2  
    create\_kernel\_process((int)mm,3,10,"mm",1);//ERR mm has great prio,because it shall run and prepare environmnet for other process  
    create\_kernel\_process((int)tty,5,10,"tty",1);  
    create\_kernel\_process((int)&p1,8,10,"p1",1);  
-------------------------------------------------------------------------------------------------------  
  0号进程为idle进程。  
  hs,fs,mm,tty分别为磁盘驱动进程，文件系统进程，内存管理进程和中端进程。这里特殊的是，hs和fs必须紧接idle进程启动，因为它们的进程id被规定为1和2。  
  mm进程优先级较高，它准备好分页模块后，其它进程才能正确运行。  
  除了idle，这些进程都运行在ring1级别，papaya的内核进程都运行在ring1。我起初是想放到ring0,可ring0级别下，进程在时钟 中断时不依tss来切换堆栈，这样一来，就需要为它们增加专门的进程调度。我花了很长时间都解决不了内核进程的堆栈问题，它们公用内核棧，却怎么防止相互 覆盖？当然，linux是做到了，它的内核进程就在ring0。  
  把内核进程放在ring1是取巧：通过eflags的IOPL向ring1开放IO敏感指令，让ring1使用内核页表，这样ring1下的进程活动在内核地址空间，同时它们又能像普通进程一样被调度。唯一的缺陷是，不能使用系统特权指令。  
  因此把idle放在ring0，看idle进程：  
-------------------------------------------------------------------------------------------------------  
     6    ;runing at ring0   
     7    idle:   
     9    hlt             ；系统特权指令  
    11    jmp idle   
-------------------------------------------------------------------------------------------------------  
  hlt指令是系统特权指令。好在idle进程简单，用不到堆栈。  
    
4,开中断，开始进程调度  
\_\_asm\_\_("sti");   
这样cpu控制权就随下一次时钟中断转交给进程调度模块。  
  
4,键盘驱动（kbd\_drv.c）  
  键盘驱动跟pcb是耦合的，每个进程都有一个私有的键盘缓冲区，它是个环形数组：  
-------------------------------------------------------------------------------------------------------  
    26    typedef struct{   
    27        char c[size\_buffer];   
    28        int head;   
    29        int tail;   
    30        int num;//环形数组当前队列长度   
    31  }OBUFFER;  
-------------------------------------------------------------------------------------------------------  
  围绕键盘缓冲区有3个函数：  
   159    void obuffer\_init(OBUFFER\* pt\_obuffer);   
   160    void obuffer\_push(OBUFFER\* pt\_obuffer,char c);   
   161   signed char obuffer\_shift(OBUFFER\* pt\_obuffer);   
  函数原型交代足够多的信息了，主要是队列的增添和删除操作。  
  我们看一个进程是如何接收按键的：  
  进程调用getchar()函数来等待按键：  
-------------------------------------------------------------------------------------------------------  
int getchar(void){   
    int ascii;   
    while((ascii=u\_obuffer\_shift())==-1)    sleep(MSGTYPE\_CHAR,0);   
    return ascii;   
}   
-------------------------------------------------------------------------------------------------------  
  u\_obuffer\_shift()是系统调用，功能是从进程的键盘缓冲区取ascii码，如果缓冲区是空的，进程就挂起，并标识自己在等待 MSGTYPE\_CHAR。  
  用户按一个键，8048检测到按键动作，转化成扫描码发送给8042,8042转换扫描码的编码规格，同时往8259A发送中断，紧接着cpu进入键盘中断例程：  
-------------------------------------------------------------------------------------------------------  
    67    static int ctrl\_down=0;   
    68    static int shift\_down=0;   
    69    void  key\_handler(void){   
    70        assert(!(ctrl\_down&&shift\_down))   
    71        int key\_code=in\_byte(0x60);//禁止ctrl，shift鍵同时按下   
    72    //        dispInt(key\_code);   
    73        //若接收到ctrl,shift鍵释放   
    74        if(key\_code>=NR\_SCAN\_CODES){   
    75            (key\_code==BC\_CTRL\_L)?ctrl\_down=0:0;   
    76            (key\_code==BC\_SHIFT\_L||key\_code==BC\_SHIFT\_R)?shift\_down=0:0;   
    77        }   
    78        //若接收到ctrl,shift鍵按下       
    79        else if(key\_code==MC\_CTRL\_L){   
    80            ctrl\_down=1;   
    81        }   
    82        else if(key\_code==MC\_SHIFT\_L||key\_code==MC\_SHIFT\_R){   
    84            shift\_down=1;   
    85        }   
    86        //处理正常ascii鍵   
    87        else{   
    91            char ascii=keymap[key\_code\*MAP\_COLS+shift\_down\*1+ctrl\_down\*2];//根据状态字节调整ascii   
    94            for(int pid=0;pid<MAX\_TASK;pid++){   
    95                if(pcb\_table[pid].mod==TASKMOD\_SLEEP&&pcb\_table[pid].msg\_type==MSGTYPE\_CHAR){   
    96                    SLEEP\_ACTIVE(pid)   
    97                    obuffer\_push(&pcb\_table[pid].obuffer,ascii);   
    98                }   
    99            }   
   100        }   
   104        proc\_dispatch();//暂时先这样，看处理速度根的上就好   
   105   }   
-------------------------------------------------------------------------------------------------------  
  键盘驱动就只这一个函数，核心的两句是：  
  SLEEP\_ACTIVE(pid)   
  obuffer\_push(&pcb\_table[pid].obuffer,ascii);   
  唤醒等待按键的进程，把按键的扫描码转化成ascii放到进程的键盘缓冲区。立即做新一轮的进程调度。  
  这就是进程获取一个按键的过程。有3点要说明：  
  1,驱动唤醒进程后，立即做proc\_dispathch()，这是必要的。虽然按键交互进程的优先级通常较高，但如果等到下一个时钟中断，延时最多能达 到10ms。papaya的时钟频率是100HZ，退一步讲，即使时钟频率提到1000HZ，仍应该立即做proc\_dispatch()。1Ms的延时 对有些用户程序是可观的。  
  2,进程都私有键盘缓冲区，为什么不共享呢？我没看过其它系统的源码，只是觉得一旦公有，那并发访问起来就麻烦的很，如何知道每个进程读到缓冲区的第几字节？  
  3,如何处理快捷键？键盘驱动负责把按键的扫描码（通常是通码）映射成ascii玛，这种映射通过一个数组：  
-------------------------------------------------------------------------------------------------------  
unsigned char keymap[NR\_SCAN\_CODES \* MAP\_COLS]   = {   
    /\* scan-code !Shift Shift  Ctrl \*/   
    /\* 0x00 - none \*/ 0, 0, 0,   
    /\* 0x01 - ESC \*/ ESC, ESC, 0,   
    …   ...  
    /\* 0x26 - 'l' \*/ 'l', 'L', 128,   
    …  ...  
    /\* 0x2E - 'c' \*/ 'c', 'C', 129,   
    …  ...  
};   
-------------------------------------------------------------------------------------------------------  
  映射只用一句代码就完成了:  
  unsigned char ascii=keymap[key\_code\*MAP\_COLS+shift\_down\*1+ctrl\_down\*2];  
  观察数组，不难发现这句代码的原理。shift\_down和ctrl\_down是状态字节，记录shift键和ctrl键是否按下，如果都没按下，那就是 规规矩矩的ascii码映射；如果shift键按下，那映射成大写的ascii；如果ctrl键按下，那映射成扩展ascii码，例如ctrl+L动作被 映射成128,ctrl+C被映射成129。至于进程怎么处理128,129,驱动就不管了。  
  我们看tty怎么处理128,129的。  
-------------------------------------------------------------------------------------------------------  
    50                if(ascii==128){   
    51                    k\_screen\_reset();   
    52                    break;   
    53                }   
    54                if(ascii==129){   
    55                    kill(front\_pid);  
    56                    oprintf("^C\n");   
    57                    break;   
    58                                        }   
-------------------------------------------------------------------------------------------------------  
  ctrl+L是清屏，ctrl+C是终止前台进程。下图是终止一个打印输出进程：  
  
  总的看来，这种键盘驱动类似消息机制，若干个进程可以同时侦听键盘。我们测试一下，让两个进程侦听按键，并打印ascii。系统运行后，我们依次按a,b,c：  
  其实不止2个进程，tty也在侦听键盘，它屏幕上回显a,b,c。  
  
  
5,写屏模块（disp.c)  
  disp.c实现了一个oprintf()函数。它目前支持打印格式有%u,%x,%c,%s，支持宽度限定，支持的转义符有\n,\b,\s,\t。我们看具体实现：  
-------------------------------------------------------------------------------------------------------  
void oprintf(char\*format,...){  
    char\*pt\_curr=format;  
    unsigned arg\_id=0;  
    int\*pt\_arg0=(int\*)&format;//pt\_arg0指向堆栈的第一个参数  
    while(\*pt\_curr!=0){  
        if(\*pt\_curr=='%'){  
            arg\_id++;  
            pt\_curr++;//跳到下一个字符（标记或宽度字符）  
            //下一个字符是否是\*？  
            if(\*pt\_curr=='\*'){  
                width=\*(pt\_arg0+arg\_id);//取出宽度变量参数  
                arg\_id++;  
                pt\_curr++;  
                goto just\_show\_var;  
            }  
            //如果有宽度限制，读取宽度  
            eat\_dec(pt\_curr,width);  
        just\_show\_var:  
            k\_show\_var(\*(pt\_arg0+arg\_id),\*pt\_curr);//注意，pt\_arg0在是最后一个入栈，所以参数寻址用pt\_arg0+arg\_id  
            pt\_curr++;//跳出当前的标志组合  
        }  
        else{  
            k\_show\_chars(pt\_curr,1);//显示下个标记前的所有字符  
            while((\*pt\_curr!='%')&&(\*pt\_curr!=0)) pt\_curr++;//同时，pt\_curr也跳到下一个标记  
        }  
        //pt\_curr现在处于字符串尾，或字符段首，或标记首  
    }  
}  
-------------------------------------------------------------------------------------------------------  
  oprintf()做的负责解析宽度标记“%\*”或”%xx”(xx指数字)，打印格式标记%u,%x...，具体的打印任务交给 k\_show\_chars（显示字符序列）和k\_show\_var（显示变量），实际上k\_show\_chars是唯一的写屏接口，k\_show\_var 最后也要调用k\_show\_chars。我们看k\_show\_var()：  
-------------------------------------------------------------------------------------------------------  
    83    void k\_show\_var(unsigned x,int val\_type){   
    84        switch(val\_type){   
    85            case 's'://val\_type=string,just call k\_show\_chars   
    86                k\_show\_chars((char\*)x,0);//字符串指针是以unsigned传进来的，不妨   
    87                break;   
    88            default://case 'c','u','d','x',val\_type=digit,use asciis\_buffer to convert   
    89                write\_asciis\_buffer(x,val\_type);   
    90                show\_asciis\_buffer();   
    91                break;   
    92        }   
    93   }  
-------------------------------------------------------------------------------------------------------  
  普通的数字，例如255，会被拆解成一个个的'2','5','5'，放到asciis\_buffer，然后调用k\_show\_chars打印出来。也就是先write\_asciis\_buffer,再show\_asciis\_buffer，一进一出。  
  disp.c里还有滚屏函数，光标移动函数，涉及到硬件端口操作。这样，disp.c也有点儿显示驱动的意思.

IDE硬盘驱动（hs.c)  
  建议先从于渊《一个操作系统的实现》327页（pdf照片348）读，把硬盘驱动读完，他讲的很清楚。这本书最好下载到自己电脑上，以后会常用<http://pan.baidu.com/share/link?shareid=227022&uk=1830497034>。附件里还有一份ide的扩展资料。  
  把这两样读完（很容易就读完了），hs.c就好理解了。下面讲IDE驱动在这个OS上的实现：  
  
  Hs进程通过一个结构体HS\_CMD接收外部命令：  
-------------------------------------------------------------------------------------------------------  
typedef struct{  
    union{  
        struct{  
            u8 low;  
            u8 mid;  
            u8 high;  
            u8 padden;  
        }lba\_stru;  
        u32 lba;  
    };  
    u8 features;  
    u32 reg\_count;//不被用户进程制定，而由hs自动计算出来,1~256,当它是256时，hs\_cmd\_out函数会向REG\_COUNT寄存器写入0  
    u8 device;  
    u8 command;  
    //1,buf并不会传递给硬盘驱动器的端口，而是给hs进程用，所以HD\_CMD这个名字，应该理解为外部传递给hs的工作参数，而非hs传给硬盘驱动器的工作参数  
    //2,改成HS\_CMD，意思就好了  
    char\*buf;  
    u32 count;//record the sector-num usr wants to r/w,it will be decreased when a local-loop of r/w finished,and hs will sleep again when count=0  
  
    char\*pt\_curr;//不被用户进程制定，由hs内部使用  
    int asker;  
}HS\_CMD;  
-------------------------------------------------------------------------------------------------------  
  这个结构体比想象的要复杂，按理说，用户只需告知读/写，起始扇区号，扇区数量，缓冲区地址。为什么有这么多不相关的成员呢？因为这个结构体既面向用户，又面向磁盘驱动器。我们看hs提供给用户的HS\_CMD设置函数：  
-------------------------------------------------------------------------------------------------------  
  void hs\_cmd\_init(u32 lba,u32 count,u8 command,char\*buf){//这个函数对外开放，所以直接操作cmd，不用参数指定cmd，因为外界访问不到hs的cmd。  
    assert(count!=0)  
    cmd.features=0;  
    cmd.device=MAKE\_REG\_DEVICE(1,0,0);//papaya的磁盘驱动，硬代码指定操作0磁盘，使用lba模式，且lba最高4位不用。  
    cmd.lba=lba;//init 3 regs at one blow  
    cmd.count=count;  
    cmd.command=command;  
  
    cmd.buf=buf;  
    cmd.pt\_curr=buf;  
    cmd.asker=pcb\_table\_info.curr\_pid;    //磁盘驱动一般只被文件系统访问，这里文件系统留下自己的pid  
                                                                 //hs完成任务后，会根据这个pid，唤醒相应的文件系统  
}  
-------------------------------------------------------------------------------------------------------  
  用户确实只需要传递进来4个必须的参数，剩余的成员由hs自动设置。  
  接受命令之后，无非就是写端口，等待中断，读端口...，我们看hs怎样把命令传递给硬盘驱动器的：  
-------------------------------------------------------------------------------------------------------  
   119    void hs\_cmd\_out(HS\_CMD\*cmd){   
   120        assert(!STATUS\_BSY)   
   121        //activate the interrupt enable bit   
   122        out\_byte(REG\_CONTROL,0);       
          
   123        out\_byte(REG\_LBA\_LOW,cmd->lba\_stru.low);   
   124        out\_byte(REG\_LBA\_MID,cmd->lba\_stru.mid);   
   125        out\_byte(REG\_LBA\_HIGH,cmd->lba\_stru.high);   
         
   126        out\_byte(REG\_FEATURES,cmd->features);   
   127        out\_byte(REG\_COUNT,cmd->reg\_count==256?0:cmd->reg\_count);   
   128        out\_byte(REG\_DEVICE,cmd->device);   
   129        //write 0x3f6 at tail,this will trigger hd\_drive to work   
   130        out\_byte(REG\_COMMAND,cmd->command);   
   131    }   
-------------------------------------------------------------------------------------------------------  
  out\_byte函数封装了写端口的汇编码。最后一句把命令写入0X3F6控制寄存器,硬盘驱动器就开始工作了。  
  这样看来，hs进程体应该会很简洁，HS\_CMD一读一写，然后等中断就行了。实际不是，因为IDE驱动器一次只能读写256个扇区,hs要向上层隐藏这一细节，即支持多于256扇区的命令。我们看进程体是怎么读写磁盘的：  
-------------------------------------------------------------------------------------------------------  
while(cmd.reg\_count>0){  
                if(cmd.command==COMMAND\_READ){  
                    sleep(MSGTYPE\_HD\_DONE,0);  
                    port\_read(REG\_DATA,cmd.pt\_curr,SIZE\_SECTOR);//break  
                    cmd.pt\_curr+=SIZE\_SECTOR;  
                    send\_hd\_eoi();  
                    in\_byte(REG\_STATUS);//activate the interrupt enable bit  
                    cmd.reg\_count--;  
                }  
                else{//COMMAND\_WRITE  
                    port\_write(REG\_DATA,cmd.pt\_curr,SIZE\_SECTOR);//break  
                    cmd.pt\_curr+=SIZE\_SECTOR;  
                    send\_hd\_eoi();  
                    in\_byte(REG\_STATUS);//activate the interrupt enable bit  
                    cmd.reg\_count--;  
                    sleep(MSGTYPE\_HD\_DONE,0);//等待磁盘向磁缓读入一个扇区  
                }  
            }  
            cmd.count-=reg\_count;  
            cmd.lba+=reg\_count;  
            if(cmd.count==0){//完成用户指定的扇区任务，进入空闲休眠  
                cmd.command=COMMAND\_NULL;  
                SLEEP\_ACTIVE(cmd.asker);    //hs done..inform proc-fs,she is sleeping on MSGTYPE\_HS\_DONE              
            }  
            //没有完成用户制定的扇区任务，继续主循环  
        }  
-------------------------------------------------------------------------------------------------------  
  IDE0号磁盘中断对应8259A上的中断号是0x2f，这个中段的中断门放在kernel.asm：  
--------------------------------------------------  
i2fh:  
    save  
    mov esp,[kernel\_esp]  
    jmp wake\_hs  
---------------------------------------------------