KERNEL/HD.C

hongjie

先看一下关于硬盘的两个数据结构

```
static struct hd_i_struct{
    int head,sect,cyl,wpcom,lzone,ctl;
    } hd_info[]= { HD_TYPE };
```

定义了硬盘的基本参数：磁头，扇区，柱面

```
static struct hd_struct {
    long start_sect;
    long nr_sects;
} hd[5*MAX_HD]={{0,0},};
```

好像是用于分区的数据结构。MAX_HD 为 2，即 linux0.01 最多支持两块硬盘。由 hd[5*MAX_HD]可知，linux0.01 一块硬盘最多可分为 5 个分区。

对硬盘的读写请求都被转化到结构体 hd_request 中，下面是它的定义

```
struct hd_request {
    int hd;          /* -1 if no request */ //是哪块硬盘，-1 表示此结构没有被占用
    int nsector;                            //要读取扇区数
    int sector;                             //起始扇区
    int head;                               //磁头
    int cyl;                                 //柱面
    int cmd;                                //读还是写
    int errors;
    struct buffer_head * bh;                //缓冲区头指针
    struct hd_request * next;               //下一请求结构
}
```

linux0.01 最多接受 32 个此结构体。所有的请求都被链结在一个链表中，并按一定方式排列，下面会讲到这种方式。程序中定义了 struct hd_request * this_request 指针，其指向当前要处理的请求。

在有了上述分析的基础上，代码就好理解了。

rw_hd（ ）函数为对外调用的接口。它和 rw_abs_hd（ ）函数一起完成了将读写请求转化为 hd_request 结构体的工作。

然后，调用 add_request（ ）函数将此结构体链入链表中。代码如下：

```
static void add_request(struct hd_request * req)
{
    struct hd_request * tmp;

    if (req->nsector != 2)    //一个缓冲区为 1024 字节，为两个扇区大小。一次读写只能
```

处理一个缓冲区

```
            panic("nsector!=2 not implemented");

        sorting=1;   //类似于一个二进制信号量。使得在进行链表添加操作时，不能进行删除操作
        if (!(tmp=this_request))
            this_request=req;   //如果请求链表为空，则将 req 设为当前请求
        else {
            if (!(tmp->next))
                tmp->next=req; //如果当前请求为最后一个请求的话，将 req 链入队末
            else {
                tmp=tmp->next;
                for ( ; tmp->next ; tmp=tmp->next)
                    if ((IN_ORDER(tmp,req) ||
                        !IN_ORDER(tmp,tmp->next)) &&
                        IN_ORDER(req,tmp->next))
    //上面的这个 if 语句相当于：设 tmp 为 a,tmp->next 为 b,req 为 x。当 a<x<b 或 a>b>x 时，
//满足条件,这样做，使得读写操作效率更高
                        break;
                req->next=tmp->next;
                tmp->next=req;
            }
        }
        sorting=0;
        if (!do_hd)
            do_request();
    }
```

实际的向硬盘控制器发送命令的工作是在 do_request( )中完成的。它先调用 hd_out( )函数将读写的参数写入控制器，然后再将要读写的数据写入硬盘缓存。

在硬盘完成读写操作后，就会发出中断。但这里的中断处理方式有些特别，它在 hd_out( )函数中根据读写设置不同的处理函数：读为 read_intr（ ）函数，写为 write_intr( )函数。代码如下：

```
    static void hd_out(unsigned int drive,unsigned int nsect,unsigned int sect,
            unsigned int head,unsigned int cyl,unsigned int cmd,
            void (*intr_addr)(void))
    {
        ∘∘∘∘∘∘∘∘∘
        do_hd = intr_addr; //设为不同的处理函数
        ∘∘∘∘∘∘∘∘∘∘
    }
```

为什么不在中断处理例程中，根据硬盘控制寄存器判断是读还是写来调用不同的函数呢？

类似如下代码：

```
void interrupt(void)
{
   ……
   if( inb(STATUS) & READ_COMPLETE)
         read_intr( );
  if( inb(STATUS) & WRITE_COMPLETE)
        write_intr( );
……..
}
```

我想可能是为了提高效率吧。通过赋值操作可以减少判断语句，有点 c++的思想。

它的中断处理例程在 system_call.s 中，代码如下：

```
_hd_interrupt:
        pushl %eax
        pushl %ecx
        pushl %edx
        push %ds
        push %es
        push %fs
        movl $0x10,%eax
        mov %ax,%ds
        mov %ax,%es    //内核数据段
        movl $0x17,%eax
        mov %ax,%fs
        movb $0x20,%al
        outb %al,$0x20             # EOI to interrupt controller #1 //设为 EOI 方式
        jmp 1f                  # give port chance to breathe
1:      jmp 1f
1:      outb %al,$0xA0             # same to controller #2 //设为 EOI 方式
        movl _do_hd,%eax      //do_hd 在 hd_out 中设置
        testl %eax,%eax
        jne 1f                       //ok,do_hd 不为 NULL
        movl $_unexpected_hd_interrupt,%eax
1:      call *%eax             # "interesting" way of handling intr. //调用处理函数
        pop %fs
        pop %es
        pop %ds
        popl %edx
        popl %ecx
        popl %eax
        iret              //处理完毕
```

在 write_intr 和 read_intr 中又调用 do_request( )函数来处理请求，形成了一条流水线式的操作。