目 录

1 硬盘驱动程序

对硬盘控制器的操作通过 I/O 端口来进行，分为两组端口，分别是命令块寄存器和控制块寄存器。如下图所示：

表 9.1 硬盘 I/O 端口及寄存器

组别	I/O 端口		读时	写时
	Primary	Secondary		
Command Block Registers	1F0h	170h	Data	Data
	1F1h	171h	Error	Features
	1F2h	172h	Sector Count	Sector Count
	1F3h	173h	LBA Low	LBA Low
	1F4h	174h	LBA Mid	LBA Mid
	1F5h	175h	LBA High	LBA High
	1F6h	176h	Device	Device
	1F7h	177h	Status	Command
Control Block Register	3F6h	376h	Alternate Status	Device Control

一块 PC 主板上有两个 IDE 口，分为 Primary 和 Secondary。对于 IDE 口的访问通过 I/O 端口地址来区分。每个 IDE 口上可以连接两个设备，分别为主设备和从设备。通过 Device 寄存器可以选择哪个设备，如果 Device 寄存器第四位为 0，就是操作主设备，如果为 1 就是操作从设备。

1.1 对命令块寄存器的操作

1.1.1 硬盘操作的端口

```
1  #define REG_DATA      0x1F0
2  #define REG_FEATURES      0x1F1
3  #define REG_ERROR     REG_FEATURES
4  #define REG_NSECTOR  0x1F2
5  #define REG_LBA_LOW  0x1F3
6  #define REG_LBA_MID  0x1F4
7  #define REG_LBA_HIGH      0x1F5
8  #define REG_DEVICE    0x1F6
9  #define REG_STATUS    0x1F7
10 #define REG_CMD       0x1F7
```

状态寄存器有如下几位：

```
1  #define STATUS_BSY   0x80
2  #define STATUS_DRDY  0x40
```

```
#define STATUS_DFSE 0x20
#define STATUS_DSC  0x10
#define STATUS_DRQ  0x08
#define STATUS_CORR 0x04
#define STATUS_IDX  0x02
#define STATUS_ERR  0x01
```

知道端口以后，我们当然想着如何操作它。首先我们定义一个结构体，用于存放操作端口的信息：

```
struct hd_cmd {
    u8  features;
    u8  count;
    u8  lba_low;
    u8  lba_mid;
    u8  lba_high;
    u8  device;
    u8  command;
};
```

随后编写操作端口的函数：

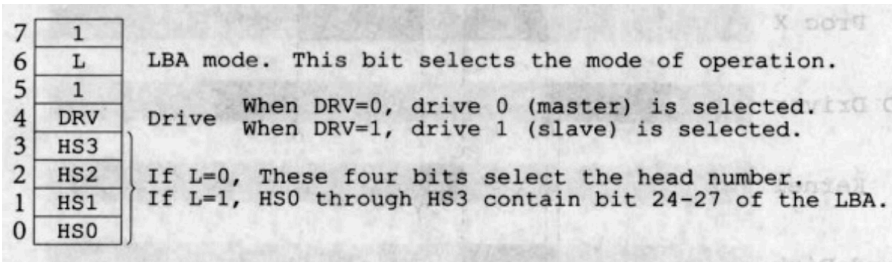
```
PRIVATE void hd_cmd_out(struct hd_cmd* cmd)
{
    if(!waitfor(STATUS_BSY, 0, HD_TIMEOUT))
        panic("hd error");

    // 填充命令块寄存器的信息
    out_byte(REG_DEV_CTRL, 0);
    out_byte(REG_FEATURES, cmd->features);
    out_byte(REG_NSECTOR, cmd->count);
    out_byte(REG_LBA_LOW, cmd->lba_low);
    out_byte(REG_LBA_MID, cmd->lba_mid);
    out_byte(REG_LBA_HIGH, cmd->lba_high);
    out_byte(REG_DEVICE, cmd->device);

    // 写入命令
    out_byte(REG_CMD, cmd->command);
}
```

读端口中的数据的函数如下：

```
; void port_read(u16 port, void* buf, int n);
port_read:
    mov edx, [esp + 4]      ; port
    mov edi, [esp + 4 + 4]  ; buf
    mov ecx, [esp + 4 + 4 + 4]  ; n
    shr ecx, 1
    cld
    rep insw ; 从DX指出的外设端口输入一个字节或字到由ES: DI指定的存储器中
    ret
```

1.1.2 Device 寄存器与 LBA 三个寄存器

在操作端口的时候，我们通过 Device 寄存器指定设备。Device 寄存器如下图所示：



寄存器有三部分：

1. DRV 位，用于指定主设备还是从设备。
2. LBA 模式位，用于指定操作模式。如果 LBA 位为 0，对磁盘的操作使用 CHS 模式。如果 LBA 位为 1，对磁盘的操作使用 LBA 模式。
3. 低四位，在 CHS 模式中表示磁头号，在 LBA 模式中表示 LBA 的 24 ～ 27 位。

LBA 模式：用于寻址硬盘，0 ～ 7 位由 LBA Low 寄存器指定，8 ～ 15 位由 LBA Mid 寄存器指定，16 ～ 23 位由 LBA High 指定，24 ～ 27 位由 Device 寄存器的低四位指定。

为了更简单地表示 Device 寄存器，我使用一个宏来表示寄存器：

```
1  #define MAKE_DEVICE_REG(lba, drv, lba_highest) (((lba) << 6) | \
2                                                  ((drv) << 4) | \
3                                                  (lba_highest & 0xF) | 0xA0)
```

1.2 硬盘驱动程序

首先写一个硬盘驱动程序的系统进程，这个驱动程序接收消息类型如下：

1. 接收 DEV_IDENTIFY，用于得到硬盘信息。
2. 接收 DEV_OPEN，用于遍历所有分区。
3. 接收 DEV_CLOSE，用于关闭硬盘。
4. 接收 DEV_READ，用于读取硬盘内容。
5. 接收 DEV_WRITE，用于写入硬盘。
6. 接收 DEV_IOCTL，用于把请求的设备的起始扇区和扇区数目返回给调用者。

```
PUBLIC void task_hd()
{
    MESSAGE msg;

    init_hd();

    while(1)
    {
        send_rec(RECEIVE, ANY, &msg);
        int src = msg.source;
        switch(msg.type)
        {
            case DEV_IDENTIFY:
                hd_identify(0);
                break;
            case DEV_OPEN:
                hd_open(msg.DEVICE);
                break;
            default:
                dump_msg("HD driver::unknown msg", &msg);
                spin("FS::main_loop (invalid msg.type)");
                break;
        }
        send_rec(SEND, src, &msg);
    }
}
```

1.2.1 初始化硬盘中断的函数

这里的硬盘中断是为了和之后的硬盘驱动程序进行交互。硬盘驱动程序操作硬盘，随即进入阻塞等待硬盘工作的完成。硬盘工作完成时，将触发硬盘中断处理程序hd_handler()，随即通知硬盘驱动程序。

```
PRIVATE void init_hd()
{
    u8 * pNrDrives = (u8*)(0x475);
    printl("NrDrives:%d.\n", *pNrDrives);
    assert(*pNrDrives);

    put_irq_handler(AT_WINI_IRQ, hd_handler);
    enable_irq(CASCADE_IRQ);
    enable_irq(AT_WINI_IRQ);
}
```

硬盘中断处理程序如下：

```
PUBLIC void hd_handler(int irq)
{
    hd_status = in_byte(REG_STATUS);
    inform_int(TASK_HD);
}

```

```
PUBLIC void inform_int(int task_nr)
{
    struct proc* p = proc_table + task_nr;

    // 如果TASK_HD进程正在等待硬盘中断
    if ((p->p_flags & RECEIVING) &&
        ((p->p_recvfrom == INTERRUPT) || (p->p_recvfrom == ANY))) {
        p->p_msg->source = INTERRUPT;
        p->p_msg->type = HARD_INT;
        p->p_msg = 0;
        p->has_int_msg = 0;

        // 解除TASK_HD进程的阻塞
        p->p_flags &= ~RECEIVING;
        p->p_recvfrom = NO_TASK;
        assert(p->p_flags == 0);
        unblock(p);

        assert(p->p_flags == 0);
        assert(p->p_msg == 0);
        assert(p->p_recvfrom == NO_TASK);
        assert(p->p_sendto == NO_TASK);
    }
    else {
        p->has_int_msg = 1;
    }
}
```

1.2.2 得到硬盘信息

编写得到硬盘信息的函数，实现它的想法如下：

1. 首先向 Device 寄存器的第四位指定驱动器，0 表示主设备，1 表示从设备。
2. 然后填充命令块寄存器的其他寄存器。
3. 然后向 Command 寄存器写入十六进制 ECh，从而获得 Data 数据。
4. 等待硬盘中断，原因随后解释。
5. 最后我们通过 Data 寄存器读取数据，总共 256 个数据。

```
PRIVATE void hd_identify(int drive)
{
    struct hd_cmd cmd;
    cmd.device = MAKE_DEVICE_REG(0, drive, 0);
    cmd.command = ATA_IDENTIFY;
    hd_cmd_out(&cmd);
    interrupt_wait();
    port_read(REG_DATA, hdbuf, SECTOR_SIZE);
    print_identify_info((u16*)hdbuf);
}
```

当我们硬盘驱动程序发出指令后，需要等待硬盘工作的完成。我们这里设计硬盘完成工作的时候，会触发中断，执行中断处理程序 hd_handler()，然后调用 inform_int() 函数，解除硬盘驱动的阻塞。等待硬盘中断的函数如下：

```
PRIVATE void interrupt_wait()
{
    MESSAGE msg;
    send_rec(RECEIVE, INTERRUPT, &msg);
}
```

现在我们的 hdbuf 就存放着 256 字节的硬盘信息，具体参数可以从 AT Attachment with Packet Interface 文档中茶的，我们这里只打印出如下的参数：

表 9.2 通过 IDENTIFY 命令得到的硬盘参数（节选）

偏移	描述
10～19	序列号 (20 个 ASCII 字符)
27～46	型号 (40 个 ASCII 字符)
60～61	用户可用的最大扇区数
49	功能（Capabilities） (bit 9 为 1 表示支持 LBA)
83	支持的命令集 (bit 10 为 1 表示支持 48 位寻址)

打印硬盘信息的函数如下：

```
PRIVATE void print_identify_info(u16* hdinfo)
{
    int i, k;
    char s[64];
    struct iden_info_ascii{
        int idx;
        int len;
        char* desc;
    } iinfo[] = {
        {10, 20, "HD SN"}, // 定义序列号
        {27, 40, "HD Model"} // 定义型号
    }

    for (k = 0; k < sizeof(iinfo)/sizeof(iinfo[0]); k++)
    {
        char* p = (char*)&hdinfo[iinfo[k].idx];
```

```
17              for(i = 0; i < iinfo[k].len/2; i++)
18              {
19                  s[i*2 + 1] = *p++;
20                  s[i*2] = *p++;
21              }
22              s[i*2] = 0;
23              printl("%s: %s\n", iinfo[k].desc, s);
24          }
25
26          // 得到支持的功能
27          int capabilities = hdinfo[49];
28          printl("LBA supported: %s\n", (capabilities & 0x0200) ? "Yes" : "No");
29
30          // 得到支持的命令集
31          int cmd_set_supported = hdinfo[83];
32          printl("LBA48 supported: %s\n", (cmd_set_supported & 0x0400) ? "Yes" : "No"
                );
33
34          // 得到用户可用的最大扇区数
35          int sectors = ((int)hdinfo[61] << 16) + hdinfo[60];
36          printl("HD size: %dMB\n", sectors * 512 / 1000000);
37      }
```

1.2.3 遍历所有分区

函数如下：

```
1   PRIVATE void hd_open(int device)
2   {
3       int drive = DRV_OF_DEV(device);
4       assert(drive == 0); /* only one drive */
5
6       // 读取硬盘驱动器的信息
7       hd_identify(drive);
8
9       // 打印分区表
10      if (hd_info[drive].open_cnt++ == 0) {
11          partition(drive * (NR_PART_PER_DRIVE + 1), P_PRIMARY);
12          print_hdinfo(&hd_info[drive]);
13      }
14  }
```

定义一个硬盘信息的结构体，每个硬盘都应该有一个 hd_info 结构体，其中 primary 成员用来记录所有主分区的起始扇区和扇区数目，logical 成员用来记录所有逻辑分区的起始扇区和扇区数目。

```
1   struct part_info {
2       u32 base;   // 起始扇区
3       u32 size;   // 扇区数目
4   };
5
6   /* main drive struct, one entry per drive */
```

```
struct hd_info
{
    int          open_cnt;
    struct part_info     primary[NR_PRIM_PER_DRIVE];
    struct part_info     logical[NR_SUB_PER_DRIVE];
};
```

在实现 partition() 函数之前，先来了解主分区和扩展分区：

```
磁盘分区有三种形式：主分区、扩展分区和逻辑分区。

主分区最多有四个。如果要在硬盘上安装操作系统，那么这个必须有一个主分区。
主分区中不能再划分其他类型的分区，每个主分区相当于一个逻辑磁盘。

扩展分区不能直接使用，必须将它划分为若干个逻辑分区才能使用。
逻辑分区相当于一个逻辑磁盘，逻辑分区必须在扩展分区中划分。
扩展分区中可能又有一个扩展分区。

由主分区和逻辑分区构成的逻辑磁盘称为驱动器(Driver)或卷(Volume)。
```

再来看一下硬盘分区表，硬盘分区表是一个结构体数组，数组的每个成员是一个 16 字节的结构体，如下所示：

表 9.3 分区表结构

偏移	长度	描述
0	1	状态（80h= 可引导, 00h= 不可引导, 其他 = 不合法）
1	1	起始磁头号
2	1	起始扇区号（仅用了低 6 位，高 2 位为起始柱面号的第 8,9 位）
3	1	起始柱面号的低 8 位
4	1	分区类型（System ID）
5	1	结束磁头号
6	1	结束扇区号（仅用了低 6 位，高 2 位为结束柱面号的第 8,9 位）
7	1	结束柱面号的低 8 位
8	4	起始扇区的 LBA
12	4	扇区数目

这块分区表位于分区起始扇区的 1BEh 处，我们可以通过类似于 hd_identify() 函数的方法得到它。我们可以根据分区表来判断分区类型、起始扇区 LBA 和扇区数目，其中关键信息是分区类型，如果分区类型是 5，说明它是扩展分区，需要继续递归，否则直接打印分区信息。定义一个分区表的结构体：

```
struct part_ent {
    u8 boot_ind;      // 表示是否可以引导
    u8 start_head;    // 起始柱头号
    u8 start_sector;  // 起始扇区号
    u8 start_cyl;     // 起始柱面号的低八位
    u8 sys_id;        // 分区类型
```

```
        u8 end_head;     // 结束柱头号
        u8 end_sector;   // 结束扇区号
        u8 end_cyl;      // 结束柱面号的低八位
        u32 start_sect;  // 起始扇区的LBA
        u32 nr_sects;    // 扇区数目
    } PARTITION_ENTRY;
```

我们现在对遍历一块硬盘的分区有了基本的头绪，但还有一件事需要知道，就是向 command 寄存器发送命令时，需要指定扇区的 LBA，这个需要填充 LBA low、LBA mid、LBA high 和 Device 寄存器。这样我们就需要分区的设备号。在操作系统中，有两种设备号：主设备号和次设备号。主设备号告诉操作系统应该用哪个驱动程序来处理，次设备号告诉驱动程序具体是哪个设备。我们这里遍历分区用的当然是次设备号。给定一个次设备号，我们可以很容易地计算出它是主分区还是扩展分区。给定一个分区的名称，我们也很容易计算出其次设备号。

定义一些与设备号相关的宏：

```
#define MAX_DRIVES          2   // 支持硬盘的数目
#define NR_PART_PER_DRIVE    4  // 每个硬盘最多有多少主分区
#define NR_SUB_PER_PART      16 // 每个扩展分区最多有多少个逻辑分区
#define NR_SUB_PER_DRIVE     (NR_SUB_PER_PART * NR_PART_PER_DRIVE)
#define NR_PRIM_PER_DRIVE    (NR_PART_PER_DRIVE + 1)

#define MAX_PRIM          (MAX_DRIVES * NR_PRIM_PER_DRIVE - 1)
#define MAX_SUBPARTITIONS    (NR_SUB_PER_DRIVE * MAX_DRIVES)

#define MINOR_hd1a        0x10 // 扩展分区最小设备号
#define MINOR_hd2a        (MINOR_hd1a+NR_SUB_PER_PART)
```

得到一块分区在哪个硬盘上：

```
#define DRV_OF_DEV(dev) (dev <= MAX_PRIM ? \
            dev / NR_PRIM_PER_DRIVE : \
            (dev - MINOR_hd1a) / NR_SUB_PER_DRIVE)
```

partition() 函数用于获取硬盘分区表，实现的想法如下：

1. 首先判断是主分区还是扩展分区。
2. 然后用 get_part_table() 函数，也就是读取该分区的信息。
3. 因为主分区中可以包含扩展分区，所以继续递归 partition() 函数。

```
PRIVATE void partition(int device, int style)
{
    int i;
    int drive = DRV_OF_DEV(device);
    struct hd_info * hdi = &hd_info[drive];

    struct part_ent part_tbl[NR_SUB_PER_DRIVE];
```

```
8
9       // 如果是主分区
10      if (style == P_PRIMARY) {
11          get_part_table(drive, drive, part_tbl);
12
13          int nr_prim_parts = 0;
14          for (i = 0; i < NR_PART_PER_DRIVE; i++) { /* 0~3 */
15              if (part_tbl[i].sys_id == NO_PART)
16                  continue;
17
18              nr_prim_parts++;
19              int dev_nr = i + 1;       /* 1~4 */
20              hdi->primary[dev_nr].base = part_tbl[i].start_sect;
21              hdi->primary[dev_nr].size = part_tbl[i].nr_sects;
22
23              if (part_tbl[i].sys_id == EXT_PART) /* extended */
24                  partition(device + dev_nr, P_EXTENDED);
25          }
26          assert(nr_prim_parts != 0);
27      }
28      // 如果是扩展分区
29      else if (style == P_EXTENDED) {
30          int j = device % NR_PRIM_PER_DRIVE; /* 1~4 */
31          int ext_start_sect = hdi->primary[j].base;
32          int s = ext_start_sect;
33          int nr_1st_sub = (j - 1) * NR_SUB_PER_PART; /* 0/16/32/48 */
34
35          for (i = 0; i < NR_SUB_PER_PART; i++) {
36              int dev_nr = nr_1st_sub + i;/* 0~15/16~31/32~47/48~63 */
37
38              get_part_table(drive, s, part_tbl);
39
40              hdi->logical[dev_nr].base = s + part_tbl[0].start_sect;
41              hdi->logical[dev_nr].size = part_tbl[0].nr_sects;
42
43              s = ext_start_sect + part_tbl[1].start_sect;
44
45              /* no more logical partitions
46              in this extended partition */
47              if (part_tbl[1].sys_id == NO_PART)
48                  break;
49          }
50      }
51      else {
52          assert(0);
53      }
54  }
55
56  PRIVATE void get_part_table(int drive, int sect_nr, struct part_ent * entry)
57  {
58      struct hd_cmd cmd;
59
60      // 先填充命令块寄存器
61      cmd.features    = 0;
62      cmd.count   = 1;
63      cmd.lba_low = sect_nr & 0xFF;
```

```
64          cmd.lba_mid = (sect_nr >>  8) & 0xFF;
65          cmd.lba_high    = (sect_nr >> 16) & 0xFF;
66          cmd.device  = MAKE_DEVICE_REG(1, /* LBA mode*/
67                        drive,
68                        (sect_nr >> 24) & 0xF);
69
70          // 再指定命令ATA_READ
71          cmd.command = ATA_READ;
72          hd_cmd_out(&cmd);
73          interrupt_wait();
74
75          // 读取数据到hdbuf缓冲区
76          port_read(REG_DATA, hdbuf, SECTOR_SIZE);
77
78          memcpy(entry,
79              hdbuf + PARTITION_TABLE_OFFSET,
80              sizeof(struct part_ent) * NR_PART_PER_DRIVE);
81      }
```

打印分区信息：

```
1   PRIVATE void print_hdinfo(struct hd_info * hdi)
2   {
3       int i;
4       for (i = 0; i < NR_PART_PER_DRIVE + 1; i++) {
5           printl("%sPART_%d: base %d(0x%x), size %d(0x%x) (in sector)\n",
6               i == 0 ? " " : "     ",
7               i,
8               hdi->primary[i].base,
9               hdi->primary[i].base,
10              hdi->primary[i].size,
11              hdi->primary[i].size);
12      }
13      for (i = 0; i < NR_SUB_PER_DRIVE; i++) {
14          if (hdi->logical[i].size == 0)
15              continue;
16          printl("          "
17              "%d: base %d(0x%x), size %d(0x%x) (in sector)\n",
18              i,
19              hdi->logical[i].base,
20              hdi->logical[i].base,
21              hdi->logical[i].size,
22              hdi->logical[i].size);
23      }
24  }
```

1.2.4 关闭硬盘

关闭硬盘的函数很简单，也就是将 hd_info 的结构体成员 open_cnt 减一，如下所示：

```
1   PRIVATE void hd_close(int device)
2   {
3       int drive = DRV_OF_DEV(device);
```

```
4           assert(drive == 0); /* only one drive */
5
6           hd_info[drive].open_cnt--;
7       }
```

1.2.5 读写硬盘内容

实现这个函数的想法如下：

1. 根据 Message 得到我们所要操作的扇区的扇区号。
2. 根据 Message 的类型，决定是向硬盘发出 ATA_READ 命令还是 ATA_WRITE 命令。
3. 如果是 DEV_READ，就调用 port_read() 函数，然后将数据拷贝到消息体中的缓存区。
4. 如果是 DEV_WRITE，就调用 port_write() 函数，将消息体中附带的数据写入硬盘。

```
1       PRIVATE void hd_rdwt(MESSAGE * p)
2       {
3           int drive = DRV_OF_DEV(p->DEVICE);
4
5           u64 pos = p->POSITION;
6           assert((pos >> SECTOR_SIZE_SHIFT) < (1 << 31));
7
8           assert((pos & 0x1FF) == 0);
9
10          // 得到扇区号
11          u32 sect_nr = (u32)(pos >> SECTOR_SIZE_SHIFT); /* pos / SECTOR_SIZE */
12          int logidx = (p->DEVICE - MINOR_hd1a) % NR_SUB_PER_DRIVE;
13          sect_nr += p->DEVICE < MAX_PRIM ?
14              hd_info[drive].primary[p->DEVICE].base :
15              hd_info[drive].logical[logidx].base;
16
17          // 向硬盘发出命令
18          struct hd_cmd cmd;
19          cmd.features    = 0;
20          cmd.count   = (p->CNT + SECTOR_SIZE - 1) / SECTOR_SIZE;
21          cmd.lba_low = sect_nr & 0xFF;
22          cmd.lba_mid = (sect_nr >>  8) & 0xFF;
23          cmd.lba_high    = (sect_nr >> 16) & 0xFF;
24          cmd.device  = MAKE_DEVICE_REG(1, drive, (sect_nr >> 24) & 0xF);
25          cmd.command = (p->type == DEV_READ) ? ATA_READ : ATA_WRITE;
26          hd_cmd_out(&cmd);
27
28          int bytes_left = p->CNT;
29          void * la = (void*)va2la(p->PROC_NR, p->BUF);
30
31          while (bytes_left) {
32              int bytes = min(SECTOR_SIZE, bytes_left);
33              if (p->type == DEV_READ) {
```

```
34                  interrupt_wait();
35                  port_read(REG_DATA, hdbuf, SECTOR_SIZE);
36                  phys_copy(la, (void*)va2la(TASK_HD, hdbuf), bytes);
37              }
38              else {
39                  if (!waitfor(STATUS_DRQ, STATUS_DRQ, HD_TIMEOUT))
40                      panic("hd writing error.");
41
42                  port_write(REG_DATA, la, bytes);
43                  interrupt_wait();
44              }
45              bytes_left -= SECTOR_SIZE;
46              la += SECTOR_SIZE;
47          }
48      }
```

这里再写一个 port_write() 函数：

```
1   port_write:
2       mov edx, [esp + 4]      ; port
3       mov esi, [esp + 4 + 4]  ; buf
4       mov ecx, [esp + 4 + 4 + 4]  ; n
5       shr ecx, 1
6       cld
7       rep outsw
8       ret
```

1.2.6 处理 IOCTL

目前这个函数只处理一个消息类型，也就是 DIOCTL_GET_GEO，只是把硬盘的起始扇区和扇区数目返回给调用者：

```
1   PRIVATE void hd_ioctl(MESSAGE * p)
2   {
3       int device = p->DEVICE;
4       int drive = DRV_OF_DEV(device);
5
6       struct hd_info * hdi = &hd_info[drive];
7
8       if (p->REQUEST == DIOCTL_GET_GEO) {
9           void * dst = va2la(p->PROC_NR, p->BUF);
10          void * src = va2la(TASK_HD,
11                  device < MAX_PRIM ?
12                  &hdi->primary[device] :
13                  &hdi->logical[(device - MINOR_hd1a) %
14                          NR_SUB_PER_DRIVE]);
15
16          phys_copy(dst, src, sizeof(struct part_info));
17      }
18      else {
19          assert(0);
20      }
21  }
```