一、实验目的和内容

Linux 0.11只支持简单的页面换入功能，而要实现完整的虚拟内存，必须给操作系统分配交换分区，即用来存放换出页面的磁盘空间。只有基于交换分区才能实现完整的内存页换入、换出，由于Linux 0.11中没有交换分区，所以Linux 0.11上的页面换入只能看作是虚拟内存管理的部分实现。

本实验目的就是扩展Linux 0.11，使之可以驱动和管理交换分区，并在此基础上实现完整的虚拟内存管理。

主要内容分为三部分：

\1. 交换分区的驱动和管理；

\2. 进程页面的换出，采用基于clock算法的全局置换策略；

\3. 进程页面的换入，在缺页处理函数do_no_page基础上修改。

二、 操作方法与实验步骤

\1. 交换分区的驱动和管理

a) 给bochs增加一块硬盘

首先修改bochs配置文件0.11.bxrc，告诉bochs新增了一块硬盘，以及硬盘的位置、柱面数、磁头数和每磁道扇区数。

```
nudt@uvm:~/os/linux-0.11-lab/conf$ ls
0.11.bxrc  0.11.bxrc~  0.11-gdbstub.bxrc
```

```
17 ata0-master: type=disk, path="images/rootimage-0.11", mode=flat, cylinders=512, heads=2, spt=20
18 ata0-slave: type=disk, path="images/swap.img",mode=flat,cylinders=4,heads=10,spt=50
```

然后用终端打开准备存放新硬盘的文件夹，输入下列命令新建一块硬盘swap，设置扇区大小和总扇区数。

```
nudt@uvm:~/os/linux-0.11-lab/images$ dd if=/dev/zero of=swap.img bs=512 count=20
00
```

即可在images文件夹下看到swap.img：

```
nudt@uvm:~/os/linux-0.11-lab/images$ ls
rootimage-0.11  swap.img
```

再修改操作系统启动代码setup.s将swap硬盘信息放到内存0x90090处，使得后面swap初始化时可以找到swap硬盘。

```
77    ! Get hd1 data
78
79        mov ax,#0x0000
80        mov ds,ax
81        lds si,[4*0x46]
82        mov ax,#0x9000
83        mov es,ax
84        mov di,#0x0090
85        mov cx,#0x10
86        rep
87        movsb
```

b) 初始化swap

```
106  void swap_init(void* BIOS)
107  {
108      int i;
109      swap_dev.cyl = *(unsigned short *)BIOS;//cylinders
110      swap_dev.head = *(unsigned short *)(2+BIOS);//heads
111      swap_dev.wpcom = *(unsigned short *)(5+BIOS);
112      swap_dev.ctl = *(unsigned char *)(8+BIOS);
113      swap_dev.lzone = *(unsigned short *)(12+BIOS);
114      swap_dev.sect = *(unsigned char *)(14+BIOS);//sectors per track
115      printk("swap dev:cyls: %d, heads: %d, sects: %d\n",
116      swap_dev.cyl,swap_dev.head,swap_dev.sect);
117      swapblocks= swap_dev.cyl*swap_dev.head*swap_dev.sect /
118                      SWAP_BLOCK_SIZE;
119      swap_hash_table=(struct swap_hash_node*)malloc(swapblocks*sizeof(struct swap_hash_node));
120      for(i=0;i<swapblocks;i++)
121          swap_hash_table[i].valid=0;
122  }
```

c) 实现交换分区的数据结构

交换分区分为若干个块，每个块大小为4KB，对应一个编号swapno。

i. 散列表

功能为根据页面的进程号pid和页号pageno映射到交换分区的块号swapno，在页面换出时为页面分配swapno，在散列表增加一项；页面换入时为根据页号和进程号在交换分区找到对应块，并在散列表中删去该页面。发生冲突时采用线性探测法。

```
51    struct swap_hash_node{
52        int valid;
53        int pid;
54        int pageno;
55        int next;
56    };
```

ii. Clock队列

本实验采用基于clock算法的换出策略，将所有缓存在内存中的页面组织成一个环形表，定义扫描指针scan和换出指针swap在环形表中移动探测。基本思想就是优先换出最近没有被访问过的页面，与LRU策略类似。间隔一定时间（我设置为每经过10次时钟中断）同时移动两个指针，使其保持一定夹角，scan指针在前，将页面的访问为R置零，swap指针落后两个页面，检测页面的R位，若为1不换出；为0则换出。换出时将该页面从clock队列中删除。

\2. 进程页面的换出

换出是依靠clock队列的两个指针移动实现的，如前所述，当swap指针指向的页面R位为0时，将该页面换出。具体操作如下：

将页表项的有效位设置为0，将物理页释放回mem_map数组中，将该页面写到交换分区中（这需要用散列表为该页面分配一个交换分区中空闲的块号swapno），将该页面从clock队列中删除，页面数减1。

```
92  void clock_swap()
93  {
94      unsigned long swap_address = swap_pos->pageno;
95      unsigned long *dir = (swap_address>>20) & 0xffc;
96      unsigned long *pg_table = (0xfffff000) & (*dir);
97      unsigned long pg_entry = pg_table[(swap_address>>12) & 0x3ff];
98      unsigned long phys_addr;
99      if(pg_entry & 0x20 == 0)
100     {
101         phys_addr=pg_entry & 0xfffff000;//get physical address
102         phys_addr-=LOW_MEM;
103         phys_addr>>=12;
104         mem_map[phys_addr]--;
105         pg_table[(swap_address>>12)&0x3ff]=pg_entry & 0;
106         swap_pos->next->prev=swap_pos->prev;
107         swap_pos->prev->next=swap_pos->next;
108         pages_in_clock--;
109         write_to_swap((pg_entry & 0xfffff000),get_new_bucket(swap_pos->pid,swap_pos->pageno));
110     }
111     swap_pos=swap_pos->next;
112 }
293 void write_to_swap(unsigned long phys_addr,int swapno)
294 {
295     struct buffer_head* bh[4];
296     int i;
297     for(i=0;i<4;i++)
298     {
299         bh[i]=getblk(SWAP_DEV,i+swapno*SWAP_BLOCK_SIZE/BUFFER_BLOCK_SIZE);
300         COPYBLK(phys_addr+i*1024,bh[i]->b_data);
301         bh[i]->b_dirt=1;
302         ll_rw_block(WRITE,bh[i]);
303         wait_on_buffer(bh[i]);
304         brelse(bh[i]);
305     }
306 }
```

\3. 进程页面的换入

其实Linux 0.11本身已经具有换入功能了，当访问内存出现缺页时，调用do_no_page函数，将磁盘中的页面写入分配的物理内存位置，并修改页表项。我在do_no_page基础上做了修改：

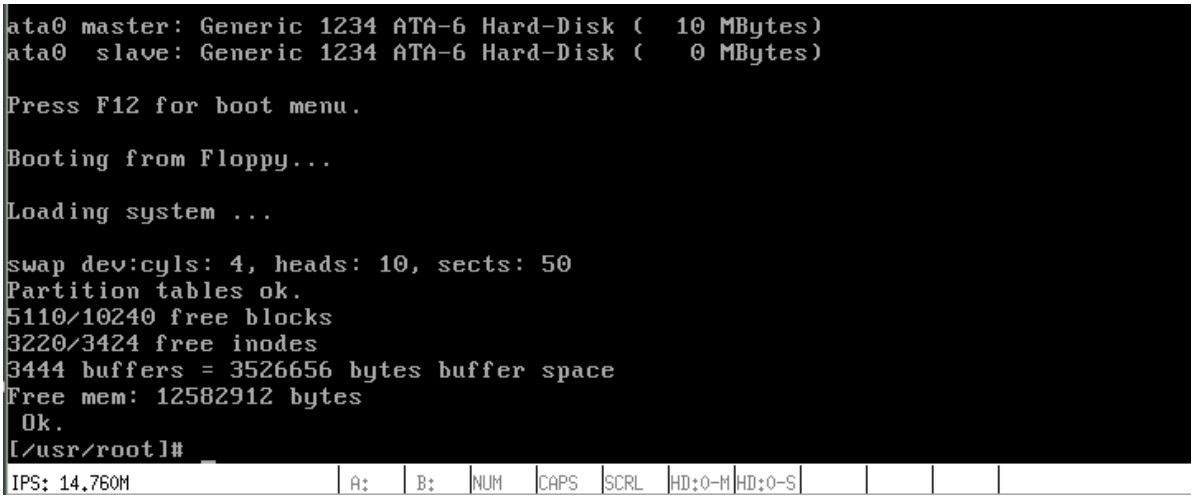
缺页处理函数要在磁盘中寻找数据块，我先在交换分区中寻找该页面所在的块swapno，如果找到，则从交换分区中写入内存，并修改散列表，将该页面去除；将该页面加入到clock队列中，放到swap指针的前一个，保持两个指针的间隔不变。

```
487     swapno=find_bucket(current->pid,address);
488     if(swapno>=0)
489     {
490         read_from_swap(page,swapno);
491         put_page(page,address);
492         available_blocks++;
493         swapped_in++;
494         struct clock_ring_node* new_node=(struct clock_ring_node*)malloc(sizeof(struct clock_
495         new_node->pid=current->pid;
496         new_node->pageno=address;
497         new_node->prev=swap_pos->prev;
498         new_node->next=swap_pos;
499         swap_pos->prev->next=new_node;
500         swap_pos->prev=new_node;
501         swap_hash_table[swapno].valid=0;
502         pages_in_clock++;
503     }
```

如果在交换分区中没有找到所需数据块，则还得从第一块磁盘中寻找，原有的do_no_page函数已经实现了这个功能，我们还需要将页面加入到clock队列中，这里需要根据clock队列中的页面数分情况讨论，后面会给出详细代码，这里不展开讲了。

三、 实验结果与分析

初始化效果如下图：



为了监测实验结果，我在每次执行一个可执行文件时(修改execve函数)，输出当前的交换分区总块数swapblocks，换入换出的页面数swapped_in blocks，swapped_out blocks，当前clock队列中页面总数pages in clock，当前scan,swap指针指向的页面。

```
swapblocks:250
available blocks:0
swapped_in blocks:0
swapped_out blocks:0
pages in clock:0
scan:12295
swap:12295
 Ok.
swapblocks:250
available blocks:0
swapped_in blocks:0
swapped_out blocks:0
pages in clock:46
scan:16732160
swap:16732880
swapblocks:250
available blocks:0
swapped_in blocks:0
swapped_out blocks:0
pages in clock:47
scan:16732160
swap:16732896
[/usr/root]#
IPS: 14.790M | A: | B: | NUM | CAPS | SCRL | HD:0-M | HD:0-S
```

四、问题与建议(可选)

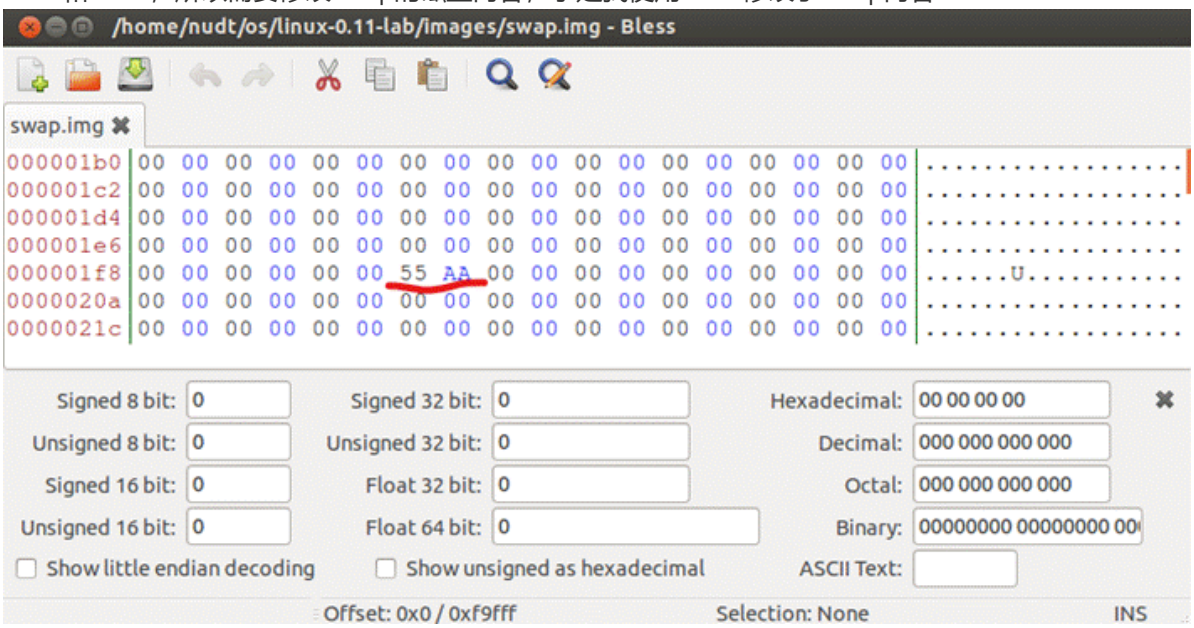
\1. 最开始增加一块硬盘时，会出现分区表错误“bad partition table on drive 1”:

```
ata0 master: Generic 1234 ATA-6 Hard-Disk (   10 MBytes)
ata0  slave: Generic 1234 ATA-6 Hard-Disk (    0 MBytes)

Press F12 for boot menu.

Booting from Floppy...

Loading system ...

swap dev:cyls: 4, heads: 10, sects: 50
Bad partition table on drive 1
Kernel panic:
IPS: 15.225M | A: | B: | NUM | CAPS | SCRL | HD:0-M | HD:0-S
```

于是我查询了输出该语句的代码如下:

```
136     for (drive=0 ; drive<NR_HD ; drive++) {
137         if (!(bh = bread(0x300 + drive*5,0))) {
138             printk("Unable to read partition table of drive %d\n\r",
139                 drive);
140             panic("");
141         }
142         if (bh->b_data[510] != 0x55 || (unsigned char)
143             bh->b_data[511] != 0xAA) {
144             printk("Bad partition table on drive %d\n\r",drive);
145             panic("");
146         }
```

发现输出该语句的判断条件是在1号驱动，也就是305设备，即swap磁盘的第511和512个字节需要是0x55和0xAA，所以需要修改swap的磁盘内容，于是我使用bless修改了swap内容：



再次启动就没有问题了

\2. scan和swap指针的初始化问题

既然我们使用两个指针scan和swap指向页面，但在最开始内存中并没有缓存页面，两个指针无法初始化。经过思考采用以下方法：

最初CPU访问某个虚拟地址，其所在的虚拟页没有缓存，触发缺页处理函数do_no_page，将页面从第一块磁盘中换入内存。由于实现clock算法至少需要4个页面，于是定义一个变量pages_in_clock，记录clock中页面数，在do_timer函数中每次移动两个指针前，检测一次页面数，只有当页面数超过4时，才会执行移动的动作。直到执行了4次缺页处理函数，才完成对scan和swap指针的初始化。

```
487     if(pages_in_clock==0)
488     {
489         scan_pos=new_node;
490         swap_pos=new_node;
491         new_node->next=new_node;
492         new_node->prev=new_node;
493     }
494     else if(pages_in_clock==1)
495     {
496         scan_pos->next=new_node;
497         scan_pos->prev=new_node;
498         new_node->next=scan_pos;
499         new_node->prev=scan_pos;
500         swap_pos=new_node;
501     }
502     else if(pages_in_clock==2)
503     {
504         scan_pos->prev=new_node;
505         swap_pos->next=new_node;
506         new_node->next=scan_pos;
507         new_node->prev=swap_pos;
508     }
509     else if(pages_in_clock==3)
510     {
511         new_node->prev=swap_pos;
512         new_node->next=swap_pos->next;
513         swap_pos->next->prev=new_node;
514         swap_pos->next=new_node;
515     }
```