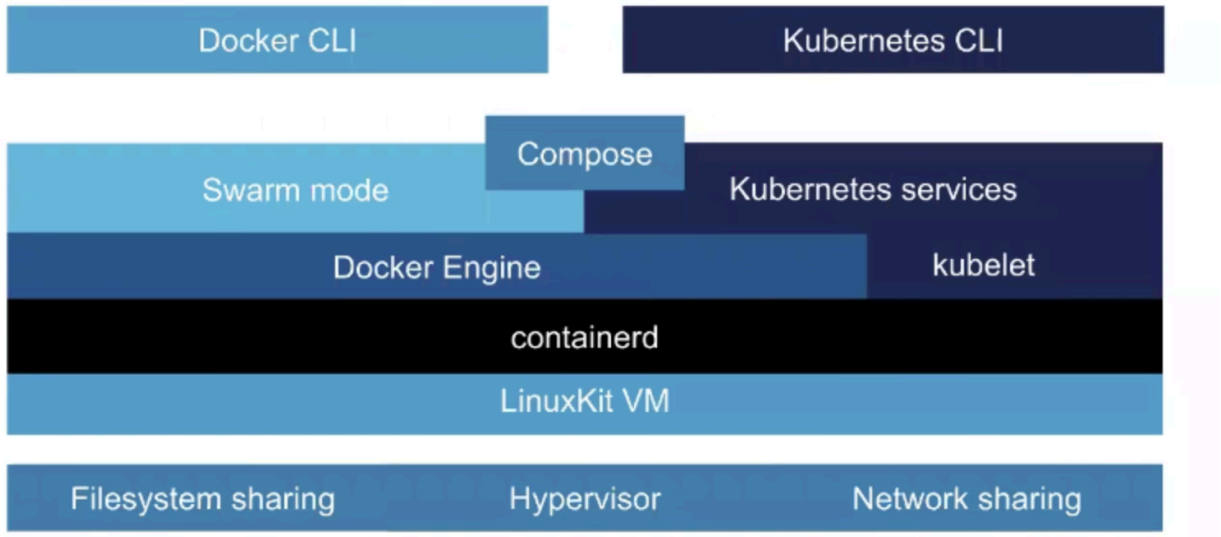
操作系统实验八

前言

这个实验实际上是在完成字符显示那个实验之后完成的，因为相对起来，这个又算一个简单的，相对简单，虽然还是研究了很长时间，其实还是有很多问题的，在实验之前我还试图解决一下不能挂载MINIX文件的问题，但是失败了：

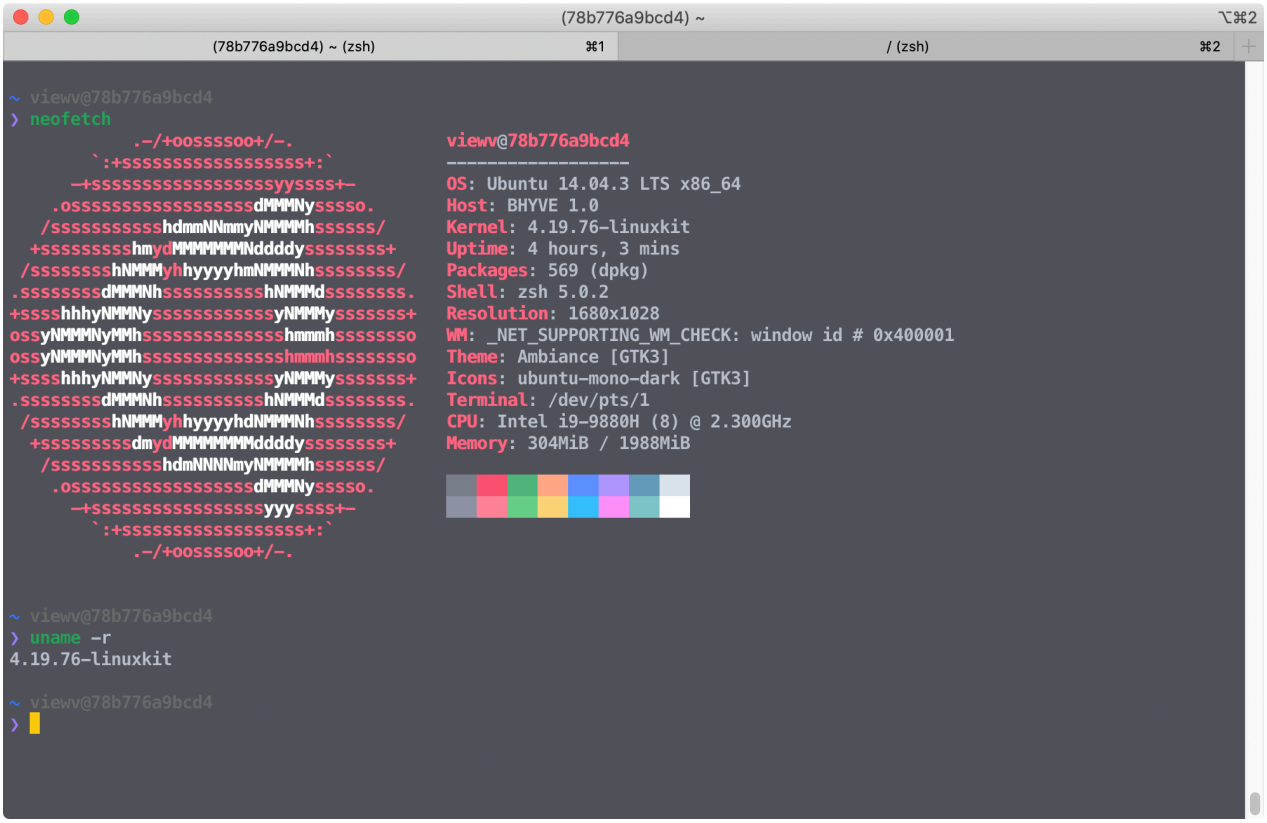
首先docker在Mac和Windows这样不是Linux的系统上，实际上是依赖虚拟机层的（也许不太准确，因为微软重视云市场，应该有个windows的容器支持），因为应该只有Linux支持这样的空间裁剪之类的我不算很懂的操作，而且实际上docker的内核是和宿主机是一样的，所以要想实现Mac下面运行docker就需要一层虚拟机层，现在的架构差不多是这样：



底层是系统，之后再上一层实际上现在Mac用的虚拟机叫hyperkit，我这里说的应该不准确，这个hyperkit应该就是图中的Hypervisor，之后在这个虚拟机上面会跑一个叫Linuxkit的小型Linux，我看了，应该是个定制内核的Alpine Linux，可用用下面的指令访问到这一层去在Mac系统下面：

```
screen ~/Library/Containers/com.docker.docker/Data/vms/0/tty
```

实际上也不用访问到这层测试，在docker任何一个容器中看一下内核版本我们会发现：



这时候我们就会发现其运行的内核4.19.76-linuxkit（话说截止我写这篇报告的时候Linux稳定版本应该出到5.4了），这个内核有什么问题呢?

在linuxkit中的默认编译内核参数的配置文件中，我们发现有这么两行：

```
# CONFIG_MINIX_SUBPARTITION is not set
# CONFIG_MINIX_FS is not set
```

真相大白，这种古老的文件系统，默认编译的系统已经取消支持了，那么我们自己编译一份替换呢，我试过了，非常不幸的失败了，感觉不是那么简单的这个docker for Mac desktop程序，历史上基于virtualbox也许可能会简单，但是这里不想回去试了，所以知道了原因，但是我没有办法解决，对了，我还试图编译了对应的minix文件系统的插件，ko文件那种的，但是不能加载，应该也是有些限制，总是就是失败了，不过这一波操作还是让我理解了其运行架构，也算是熟悉了操作系统。

值得一提的是在windows现在推出wsl2之后，其相当于使用系统的hyperv虚拟化一个真的Linux，之后docker for windows也提供了wsl2的支持，这样看着是会运行一个非常完整的Linux内核，现在发现windows这样运行docker做实验都比Mac好了，当然不能否认使用docker这样进行实验，这比跑一个大虚拟机又快又爽。

Proc文件系统的实现

实际上是实现一个假的文件系统，比较实际上其是在每次调用的时候返回系统当前信息给用户空间，而不是从外部存储器上取得一些数据送出去，看起来很简单但是这个实验我还是写了一点时间，因为不熟悉文件系统，实际上现在整完了也不算很明晰。

这里实现手册的帮助很大，但是也有坑，后面会说到，首先是创建一个文件系统并且在开机的时候就挂载上去：

如何增加一种新文件类型

首先找到include/sys/stat.h，这个地方定义了文件系统的宏和测试宏，添加结果如下：

```
//*文件类型
#define S_IFMT 00170000 //文件类型屏蔽码
#define S_IFREG 0100000 //常规文件
#define S_IFBLK 0060000 //块特殊文件
#define S_IFDIR 0040000 //目录文件
#define S_IFCHR 0020000 //字符设备文件
#define S_IFIFO 0010000 //FIFO特殊文件
/*
* 增加需要的文件系统 S_IFPRC
* 这里不用PROC是因为好像人家上面写的都是IF+三个字母
* --viewvos
*/
#define S_IFPRC 0070000 //Proc文件

//*文件属性位
#define S_ISUID 0004000 //执行时设置用户ID
#define S_ISGID 0002000 //执行时设置组ID
#define S_ISVTX 0001000 //对于目录，受限删除标志

//*测试
#define S_ISREG(m) (((m)&S_IFMT) == S_IFREG)  //测试是否为正常文件
#define S_ISDIR(m) (((m)&S_IFMT) == S_IFDIR)  //是否目录文件
#define S_ISCHR(m) (((m)&S_IFMT) == S_IFCHR)  //是否字符设备文件
#define S_ISBLK(m) (((m)&S_IFMT) == S_IFBLK)  //是否块设备文件
#define S_ISFIFO(m) (((m)&S_IFMT) == S_IFIFO) //是否FIFO特殊文件
//*viewvos PROC->PRC
#define S_ISPRC(m) (((m)&S_IFMT) == S_IFPRC) //是否Proc文件
```

这里我都根据内核解释那本书加上了commit，不同实验手册这里我加上的文件是PRC，因为上面看着Linus写的都是这样的三位，其他的我还阅读了下面的一点，下面说明了不同成员的权限设置，已经在代码中加了commit了。

之后要让mknod()支持新增加的文件类型，在fs/namei.c中mknod的系统调用中加一句变成：

```
int sys_mknod(const char * filename, int mode, int dev)
{
    const char * basename;
    int namelen;
    struct m_inode * dir, * inode;
    struct buffer_head * bh;
    struct dir_entry * de;

    if (!suser())
        return -EPERM;
    if (!(dir = dir_namei(filename,&namelen,&basename)))
```

```
        return -ENOENT;
    if (!namelen) {
        iput(dir);
        return -ENOENT;
    }
    if (!permission(dir,MAY_WRITE)) {
        iput(dir);
        return -EPERM;
    }
    bh = find_entry(&dir,basename,namelen,&de);
    if (bh) {
        brelse(bh);
        iput(dir);
        return -EEXIST;
    }
    inode = new_inode(dir->i_dev);
    if (!inode) {
        iput(dir);
        return -ENOSPC;
    }
    inode->i_mode = mode;
  //*增加新文件类型
    if (S_ISBLK(mode) || S_ISCHR(mode) || S_ISPRC(mode))
        inode->i_zone[0] = dev;
    inode->i_mtime = inode->i_atime = CURRENT_TIME;
    inode->i_dirt = 1;
    bh = add_entry(dir,basename,namelen,&de);
    if (!bh) {
        iput(dir);
        inode->i_nlinks=0;
        iput(inode);
        return -ENOSPC;
    }
    de->inode = inode->i_num;
    bh->b_dirt = 1;
    iput(dir);
    iput(inode);
    brelse(bh);
    return 0;
}
```

这样就完成了新建，跟着实验手册走还是非常简单轻松的。

在开机的时候新建文件

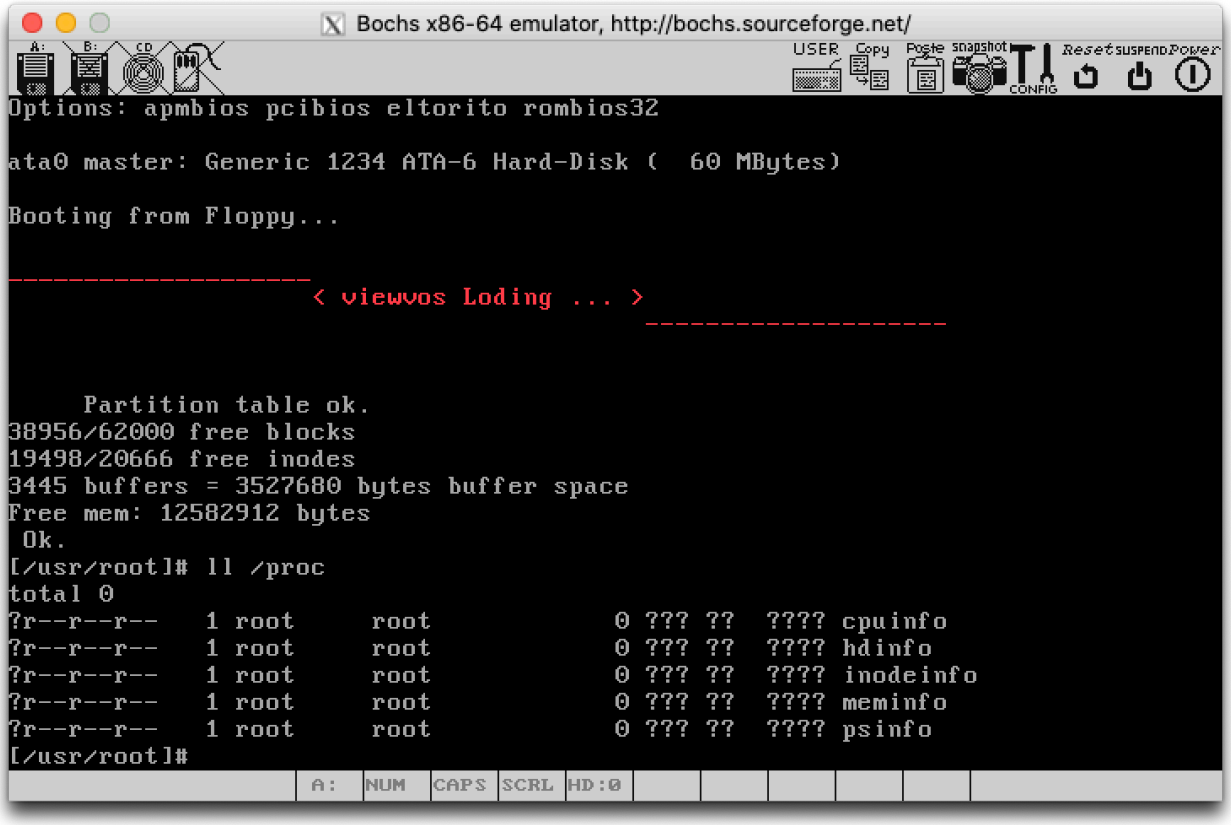
要想实现这个东西，首先要看main函数是怎么创建的，实际上前面的实验比如创建process.log来看进程转换就已经提示了怎么做，于是这里就仿照着写来挂载这个文件。

```
    //* viewvos Add
    setup((void *)&drive_info);
    (void)open("/dev/tty0", O_RDWR, 0);
    (void)dup(0);
    (void)dup(0);
    (void)open("/var/process.log", O_CREAT | O_TRUNC | O_WRONLY, 0666);
    (void)mkdir("/proc", 0755);
    (void)mknod("/proc/psinfo",S_IFPRC|0444,0);
    (void)mknod("/proc/meminfo",S_IFPRC|0444,1);
    (void)mknod("/proc/cpuinfo",S_IFPRC|0444,2);
    (void)mknod("/proc/hdinfo",S_IFPRC|0444,3);
    (void)mknod("/proc/inodeinfo",S_IFPRC|0444,4);
    //* --viewvos
```

在move_to_user_mode()后面，在fork前面，加入这段话，这里是我已经在前面加process的基础上写的，实际上原来的代码的话，应该算则在init()里面加，其实那个时候相当于是在1进程里面加，差不多啦。当然要是想调用mkdir和mknod，就需要前面学习到的系统调用的知识，在前面加一些宏：

```
/*
* Add mkdir and mknod
* int sys_mknod(const char * filename, int mode, int dev)
* int sys_mkdir(const char * pathname, int mode)
*/
static inline _syscall2(int, mkdir, const char *, name, mode_t, mode)
static inline _syscall3(int, mknod, const char *, filename, mode_t, mode,
dev_t, dev)
```

这样就实现了开机的时候新建这种文件类型，之后按照手册写的，来简单读一下这个文件，不过我这里已经都写完实验了：



很好，实验手册这次说的很详细，成功的建立了文件。

让Proc文件可读

既然是虚拟的，当用户访问的时候，肯定需要让其可用读，不是去存储设备上读取文件，而是通过函数给出系统参数，不过还是要理解怎么想办法把数据给用户空间。

前面的实验也说过，用户要是想读文件本质上是需要系统调用的，调用sys_read来读写文件，找到这个函数，让它发现在读我们需要的proc文件就调用我们写好的对应的函数给信息传递到用户空间就可以了，我们就需要在这个函数中加入这么一个东西：

```
int sys_read(unsigned int fd,char * buf,int count)
{
    //*这里应该会新建一个文件，应该每次读文件的pos都是新建的0
    struct file * file;
    struct m_inode * inode;

    if (fd>=NR_OPEN || count<0 || !(file=current->filp[fd]))
        return -EINVAL;
    if (!count)
        return 0;
    verify_area(buf,count);
    inode = file->f_inode;
    if (inode->i_pipe)
        return (file->f_mode&1)?read_pipe(inode,buf,count):-EIO;
    if (S_ISCHR(inode->i_mode))
        return rw_char(READ,inode->i_zone[0],buf,count,&file->f_pos);
    if (S_ISBLK(inode->i_mode))
```

```
        return block_read(inode->i_zone[0],&file->f_pos,buf,count);
    if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
        if (count+file->f_pos > inode->i_size)
            count = inode->i_size - file->f_pos;
        if (count<=0)
            return 0;
        return file_read(inode,file,buf,count);
    }
    //*viewvos proc file
    if (S_ISPRC(inode->i_mode)){
        return proc_read(inode->i_zone[0],&file->f_pos,buf,count);
    }
    printk("(Read)inode->i_mode=%06o\n\r",inode->i_mode);
    return -EINVAL;
}
```

这个地方就看到了那些个宏的作用，之后在这里加入我们写的函数就可以了，这个地方本质上是模仿的读取块设备的调用方法，看参数和读S_ISBLK那行是不是很相似，实际上就是模仿直接读取块文件来写的，这里的i_zone是什么呢，回到前面创建的时候，我们发现这就是文件的dev号，在main函数中我是这样定义的：

i_zone[0]	代表文件
0	psinfo
1	memento
2	cpuinfo
3	hdinfo
4	inodeinfo

实际上最后只是实现了psinfo，hdinfo和inodeinfo，meminfo这个好难，要想实现，需要读取当前应用程序是什么，之后给出每个程序占了多少内存，这很难，cpuinfo在这里没啥意义，应为Linus应该是在IBM当时的AT机上写的，故事是这样，当时Linus的祖父给上大学的他买了一台IBM AT机作为礼物（说实话还是有钱），IBM当时就是找的比尔盖茨他们也就是微软开发的DOS系统，也就是这里搭载的时候DOS系统，Linus就非常不满这个系统，DOS系统其实也不错，但是他是一个单用户单工系统，也就是没啥进程调度这种东西，一次就一个进程，之后当时CMU的操作系统还没开发出来，而GNU的内核，到今天2020年也还没成熟，没啥人用，BSD系统当时还有法律问题，伯克利估计忙着打官司和改代码，Minix系统教学用的系统，很好但是也不能说完善，所以Linus就啃Minix操作系统的书，和伙伴合作最后开发出来了Linux，宏内核架构，我们也可以在汇编读出来，这代码当时还是很依赖这一种处理器的，至于宏内核和微内核之争是另一个故事了，所以这里cpuinof也没啥用，就那一个cpu，而读取制造商这又太难了，我看了需要汇编一堆操作读出来，完全没思路。

这里还是介绍一下参数：

1. inode->i_zone[0]，这就是mknod()时指定的dev——设备编号
2. buf，指向用户空间，就是read()的第二个参数，用来接收数据
3. count，就是read()的第三个参数，说明buf指向的缓冲区大小

4. &file->f_pos，f_pos是上一次读文件结束时“文件位置指针”的指向。这里必须传指针，因为处理函数需要根据传给buf的数据量修改f_pos的值。

主要目的就是向buf里面放数据，之后的操作都是想办法向buf里放参数。

向buf里面放数据

这里按照实验要求放在proc.c文件中。

首先要有几个辅助的函数，实验手册中也给出来了，我没有使用malloc和free，我总害怕用不好内存泄漏，而且也不直观，[] 这也算是指针操作：

sprintf

```
#include <stdarg.h>
……
int sprintf(char *buf, const char *fmt, ...)
{
    va_list args; int i;
    va_start(args, fmt);
    i=vsprintf(buf, fmt, args);
    va_end(args);
    return i;
}
```

其实主要是这个了，之后就是怎么获取信息。首先我在这里我首先在这里定义了一个全局的算缓存的pbuffer来保存读取到的信息：

```
int proc_read(int dev, unsigned long * pos, char * buf, int count)
{
    int i;
    int chars;
    int read = 0;
    register char *p;

      if (dev == 0)
        read = getpsInfo();
    if (dev == 3)
        read = getHdinfo();
    if (dev == 4)
        read = getInodeinfo();
    p = pbuffer + *pos;
    buf += *pos;
    chars = MIN(read,count);
    for (i = 0; i < chars; i++)
    {
        if(*(p+i) == '\0')
            break;
        put_fs_byte(*(p+i),buf + i);
```

```
    }
    *pos += i;
    return i;
}
```

首先来说明这个函数，这个函数其实是学习同fs下面的block_dev中的块设备读取函数block_read的，大致功能就是先获取对应的信息，之后把pbuffer里面的东西放到buffer里面，下面一个一个的说怎么获取这些需要的信息。

psinfo

获取当前的进程信息，这个地方实际上算是简单，毕竟前面那个跟踪进程看了很多了，基本上了解怎么得到这个信息了：

```
int getpsInfo(){
    int loc = 0;
    struct task_struct **p;
    //* 提示信息
    loc += sprintf(pbuffer+loc,"%s","RUNNING-0, INTERRUPTIBLE-1, UNINTERRUPTIBLE-2, ZOMBIE-3, STOPPED-4\n");
    loc += sprintf(pbuffer+loc,"%s","pid\tuser\tstate\tfather\tcounter\tstart_time\n");
    for (p = &LAST_TASK;p > &FIRST_TASK;--p){
        if (*p){
            loc += sprintf(pbuffer + loc,
                           "%ld\t%ld\t%ld\t%ld\t%ld\t%ld\n",
                           (*p)->pid,(*p)->uid,(*p)->state,
                           (*p)->father,(*p)->counter,(*p)->start_time);
        }
    }
    return loc;
}
```

其实就是便利进程数组，有这个位置的进程就放东西进去，这里我放进去的参数有进程id，进程用户id，进程状态，进程的父进程id，进程的counter，还有启动时间。

hdinfo

硬盘的信息读取，这个前面没有见过，但是可以看看系统是怎么做的，之后移花接木，修改一下，首先来看超级块里面有啥：

```
/*
* 在Linux 0.11中被夹在的文件系统超级快被保存在超级块数组super_block[]中
* 该表最多有8项，因此最多同时加载8个文件系统
* 硬盘总共有多少块，多少块空闲，有多少inode等信息都放在super块中
*/
struct super_block
{
```

```
    unsigned short s_ninodes;        //* i 节点数
    unsigned short s_nzones;         //* 逻辑块数
    unsigned short s_imap_blocks;    //* i节点位图所占块数
    unsigned short s_zmap_blocks;    //* 逻辑块位图所占块数
    unsigned short s_firstdatazone; //* 数据块中第一个逻辑块块号
    unsigned short s_log_zone_size; //* Log2（磁盘块数/逻辑块）
    unsigned long s_max_size;        //* 最大文件长度
    unsigned short s_magic;          //* 文件系统幻数
    /* These are only in memory */
    struct buffer_head *s_imap[8];
    struct buffer_head *s_zmap[8];
    unsigned short s_dev;        //* 超级块所在的设备号
    struct m_inode *s_isup;      //* 被安装文件系统根目录i节点
    struct m_inode *s_imount;    //* 该文件系统被安装到的i节点
    unsigned long s_time;        //*修改时间
    struct task_struct *s_wait; //*等待本超级块的进程指针
    unsigned char s_lock;        //*锁定标志
    unsigned char s_rd_only;     //*只读标志
    unsigned char s_dirt;        //*已被修改（脏位置）
};
```

这里就是超级块里面的所有信息，已经写了注释，操作系统在开机的时候，也会去查询超级块和空余信息，可以学习一下，实际上操作系统在启动的时候，会调用mount_root来挂载根文件目录，这时候就需要统计一下，这一段代码可以使用：

```
/*
* 挂载根目录，这个函数是初始化操作的一部分，函数首先初始化文件表数组
* file_table[]和超级块表，然后读取根文件系统根i节点，然后统计显示出
* 根文件系统上的可用资源（空闲块数和空闲i节点）。该函数会在系统开机的时候
* 进行初始化的时候进行设置
*/
void mount_root(void)
{
    int i,free;
    struct super_block * p;
    struct m_inode * mi;

    if (32 != sizeof (struct d_inode))
        panic("bad i-node size");
    //*首先初始化文件表数组（共64项，即系统每次最多只能打开64个文件）和超级块表
    //*这里将所有文件结构中的引用技术都设置为0（表示空闲），之后把所有的超级块中
    //*各个结构的设备字段初始化为0，这上古系统，如果用系统文件所在的设备是软盘的话
    //*这里还会提示请插入软盘之后按下回车。
    for(i=0;i<NR_FILE;i++)
        file_table[i].f_count=0;
    if (MAJOR(ROOT_DEV) == 2) {
        printk("Insert root floppy and press ENTER");
        wait_for_keypress();
    }
```

```
    for(p = &super_block[0] ; p < &super_block[NR_SUPER] ; p++) {
        p->s_dev = 0;
        p->s_lock = 0;
        p->s_wait = NULL;
    }
    //*做好以上额外的初始化工作之后，会开始安装根文件系统，于是从根设备上读取文件
    //*文件系统块，并且取得文件系统的根inode（1号节点）在内存i节点表中的指针，如果
    //*读根设备上超级块失败或者取根节点失败，就只能pacic停机
    if (!(p=read_super(ROOT_DEV)))
        panic("Unable to mount root");
    if (!(mi=iget(ROOT_DEV,ROOT_INO)))
        panic("Unable to read root i-node");
    //*现在对超级块和根i节点进行设置，把i节点引用次数递增三次
    mi->i_count += 3 ;  /* NOTE! it is logically used 4 times, not 1 */
    p->s_isup = p->s_imount = mi;
    current->pwd = mi;
    current->root = mi;
    //*这里应该就是proc实验中可用用到的代码，这里我们对根文件系统上对资源进行统计
    //*统计该设备上空闲块数和空闲的inode数。首先令i等于超级块中表明的设备逻辑块总数
    //*然后根据逻辑块位图中对应比特位的占用情况统计出空闲块数，这里的宏函数set_bit()
    //*只是在测试比特位 i>>13是将i/8192，也就是除一个磁盘块包含的比特位数。
    free=0;
    i=p->s_nzones;
    while (-- i >= 0)
        if (!set_bit(i&8191,p->s_zmap[i>>13]->b_data))
            free++;
    //*之后再这里统计设备上空闲i节点数，首先令i等于超级块中表明的设备上的inode
    //*总数+1，+1是将0节点也统计进去，然后根据i节点位图中相应的比特位的占用情况
    //*计算出空闲的inode数，最后显示设备上可用空闲i节点数和i节点总数
    printk("%d/%d free blocks\n\r",free,p->s_nzones);
    free=0;
    i=p->s_ninodes+1;
    while (-- i >= 0)
        if (!set_bit(i&8191,p->s_imap[i>>13]->b_data))
            free++;
    printk("%d/%d free inodes\n\r",free,p->s_ninodes);
}
```

最下面那个地方就可以被拿来应用，来确定硬盘信息：

```
int getHdinfo(){
    int read = 0;
    int i, free,step;
    struct super_block *p;
    step = 0;
    //*这个地方非常我写的很不优雅，我不喜欢，应该还是我对结构不熟悉
    while (p = get_super(0x301 + step)){
        if (p){
            free = 0;
```

```
            read += sprintf(pbuffer+read,"Hd%d Block Info:\n",step/5);
            read += sprintf(pbuffer+read,"Total blocks on Hd%d:
%d\n",step/5,p->s_nzones);
            i = p->s_nzones;
            while (-- i >= 0){
                if (!set_bit(i&8191,p->s_zmap[i>>13]->b_data))
                    free++;
            }
            read += sprintf(pbuffer+read,"Free blocks on Hd%d:
%d\n",step/5,free);
            read += sprintf(pbuffer+read,"Used blocks on Hd%d:
%d\n",step/5,p->s_nzones-free);
            //*Inodes 信息
            read += sprintf(pbuffer+read,"Hd%d Inodes Info:\n",step/5);
            read += sprintf(pbuffer+read,"Total Inodes on Hd%d:
%d\n",step/5,p->s_ninodes);
            free = 0;
            i=p->s_ninodes+1;
            while(--i >= 0)
            {
                if(!set_bit(i&8191,p->s_imap[i>>13]->b_data))
                    free++;
            }
            read += sprintf(pbuffer+read,"Free Inodes on Hd%d:
%d\n",step/5,free);
            read += sprintf(pbuffer+read,"Used Inodes on Hd%d:
%d\n",step/5,p->s_ninodes-free);
        }else{
            break;
        }
        step += 5;
    }
    return read;
}
```

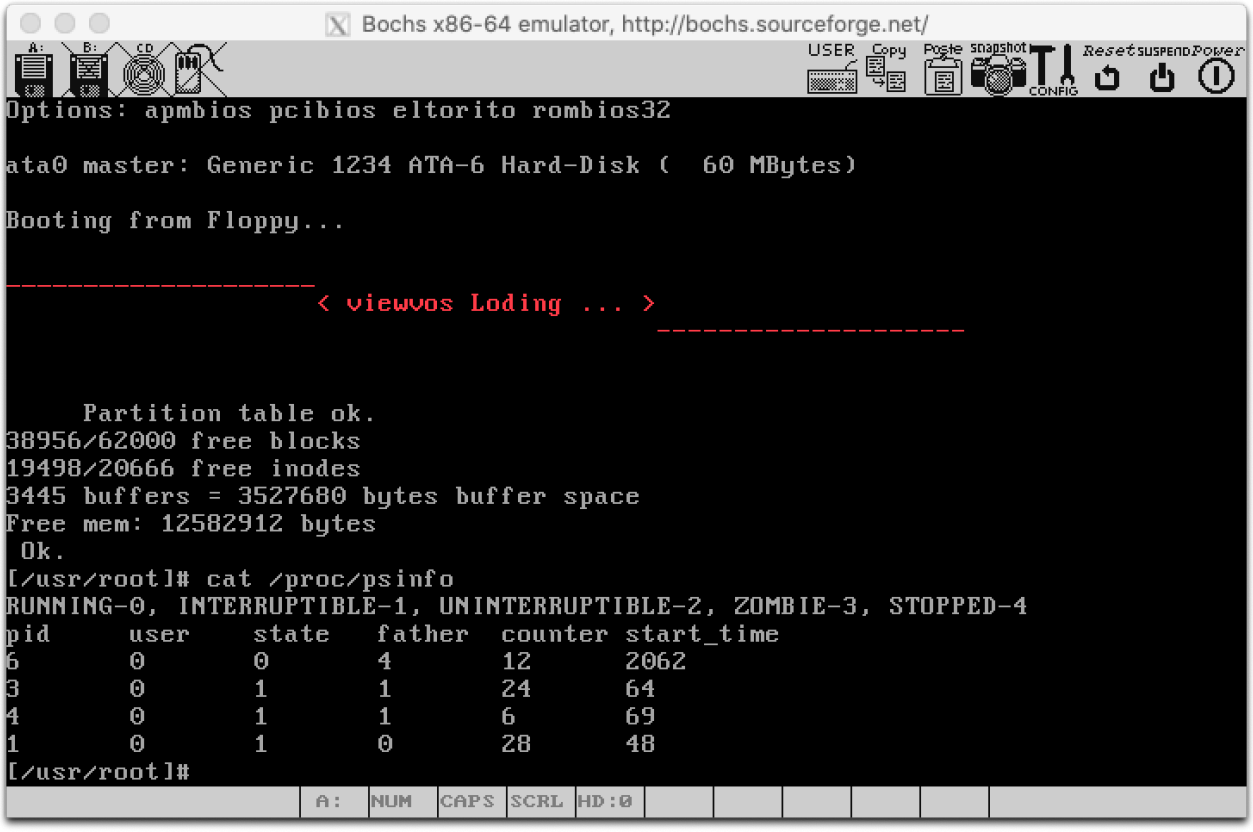
这里可能我想实现读取多个硬盘，问题来了301是啥，这个其实在实验一中其实也遇到过这个问题，这里就不再说了，还是老的Linux磁盘命名，301就是第一块磁盘的第一个分区。之后类似的有获取innode信息：

```
int getInodeinfo(){
    int i;
    int read = 0;
    struct super_block * p;
    struct m_inode *mi;
    p=get_super(0x301);
    i=p->s_ninodes+1;
    i=0;
    //printk("D-ninodes: %d",p->s_ninodes);
    while(++i < p->s_ninodes+1){
```

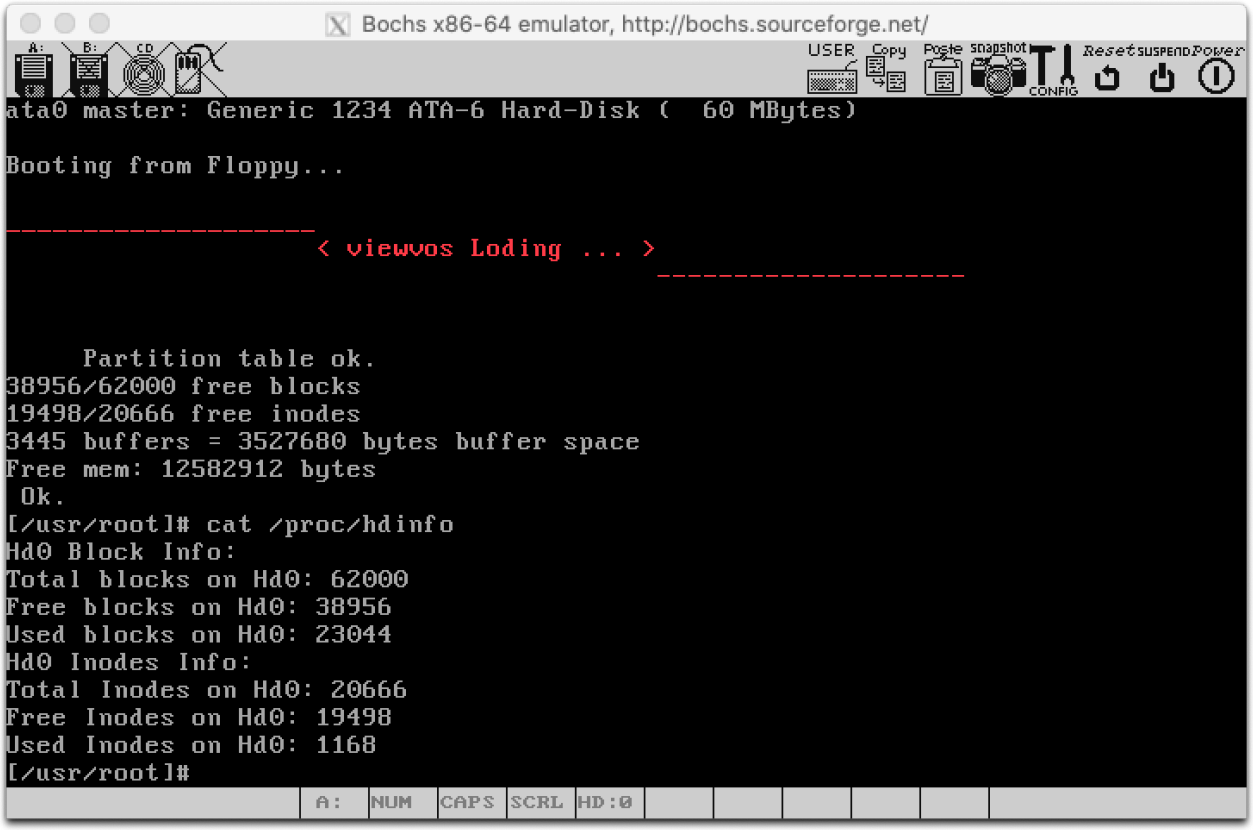
```
        if(set_bit(i&8191,p->s_imap[i>>13]->b_data)){
            mi = iget(0x301,i);
            read += sprintf(pbuffer+read,"inr:%d;zone[0]:%d\n",mi->i_num,mi->i_zone[0]);
            iput(mi);
        }
        //*这里不能读太多，否则我连文件都输出不出去（干，我512都段错误）
        //*只能500了，才能cat /proc/inodeinfo > inode.txt 成功
        if(read >= 500)
                        break;
    }
    return read;
}
```

这样就获取到了信息，说的不是很详细，具体都在代码中写了。

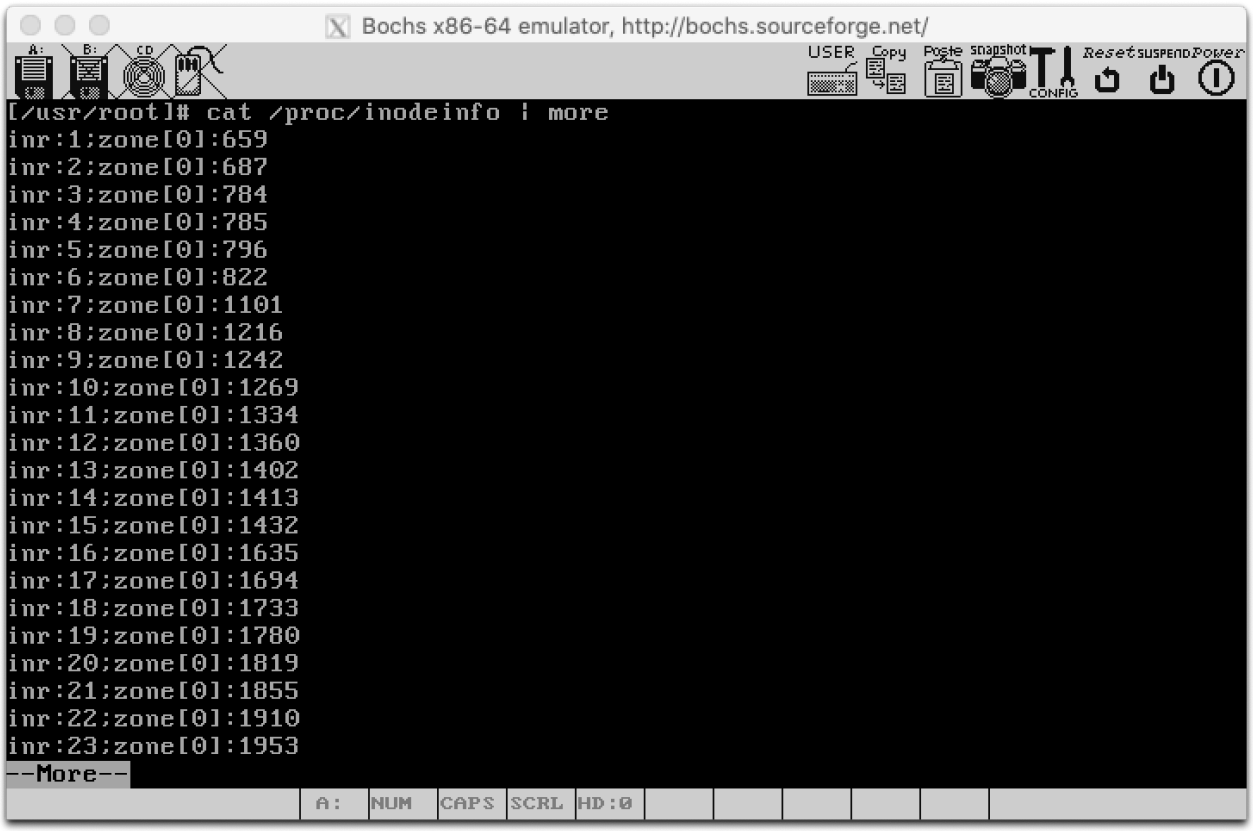
运行结果



psinfo 结果



hdinfo 结果



inodeinfo 结果，这里只能用more这样看一部分了

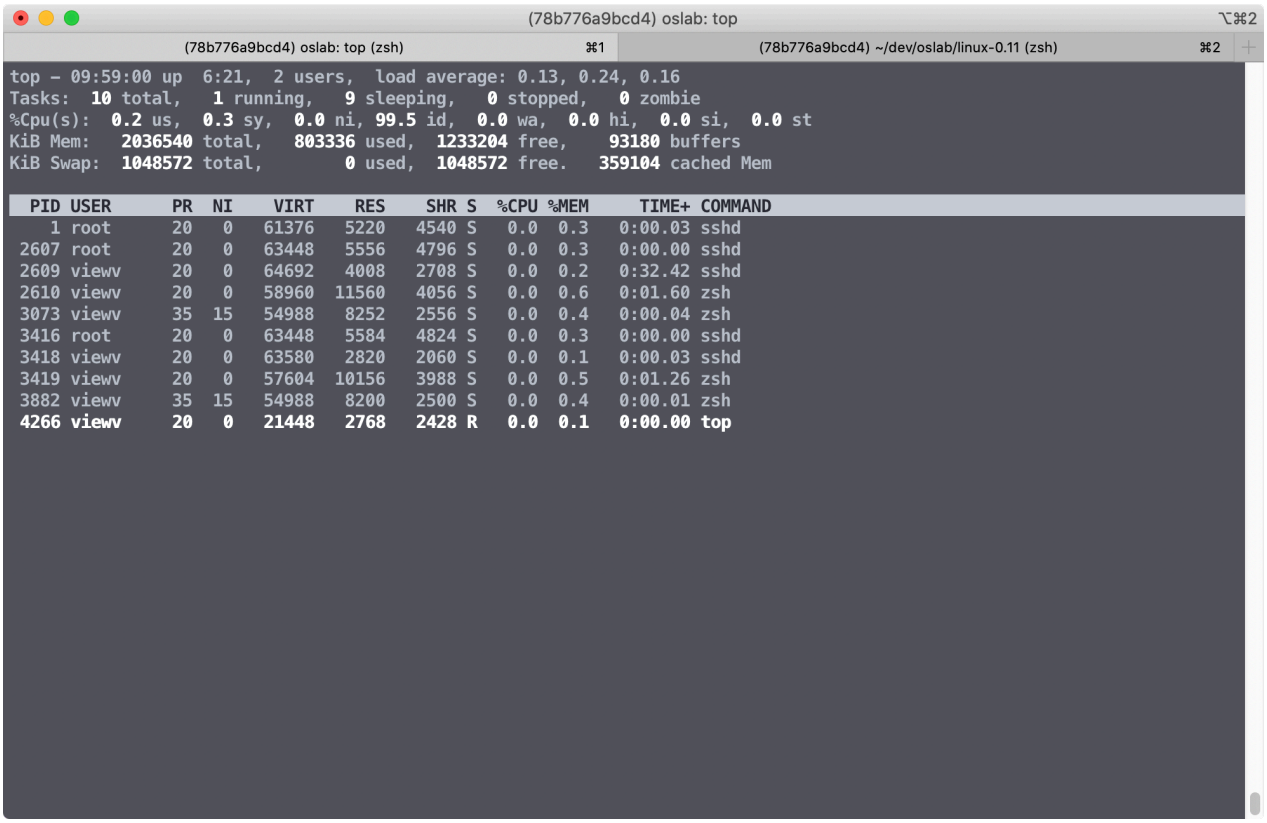
回答

1. 如果要求你在psinfo之外再实现另一个结点，具体内容自选，那么你会实现一个给出什么信息的结

点？为什么？

2. 一次read()未必能读出所有的数据，需要继续read()，直到把数据读空为止。而数次read()之间，进程的状态可能会发生变化。你认为后几次read()传给用户的数据，应该是变化后的，还是变化前的？
3. 如果是变化后的，那么用户得到的数据衔接部分是否会有混乱？如何防止混乱？
4. 如果是变化前的，那么该在什么样的情况下更新psinfo的内容？
5. 删除文件以后，/proc/inodeinfo那个inode号的inode，你发现了什么，为什么会这样？

如果我希望增加的话，我希望增加cpuinfo和meminfo和系统版本info，如果看真的现在的比如ubuntu的top指令是这样的：



这就很完善，很强大。

read还是读取前的，只有等读取位置f_pos为0时才更新psinfo内容，前面有个地方的注释提到了这一点，每次其实都相当于新建了文件，只有这个pos为0才说明要读新文件。

删除文件之后，该inode对应的i_zone[0]依然存在。只是从inode映射中取消映射该inode，硬盘上的数据还是存在的。