proc文件系统的实现

难度系数：★★★☆☆

实验目的

- 掌握虚拟文件系统的实现原理
- 实践文件、目录、索引节点等概念

实验内容

在Linux 0.11上实现procfs（proc文件系统）内的psinfo结点。当读取此结点的内容时，可得到系统当前所有进程的状态信息。例如，用cat命令显示/proc/psinfo的内容，可得到：

```
# cat /proc/psinfo
pid     state   father  counter start_time
0       1       -1      0       0
1       1       0       28      1
4       1       1       1       73
3       1       1       27      63
6       0       4       12      817
```

procfs及其结点要在内核启动时自动创建。相关功能实现在fs/proc.c文件内。

实验报告

完成实验后，在实验报告中回答如下问题：

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？

    1. 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？
    2. 如果是变化前的，那么该在什么样的情况下更新psinfo的内容？

评分标准

- 自动创建/proc，20%
- 自动创建/proc/psinfo，20%
- psinfo内容可读，20%
- psinfo内容符合题目要求，20%
- 实验报告，20%

实验提示

procfs简介

正式的Linux内核实现了procfs，它是一个虚拟文件系统，通常被mount到/proc目录上，通过虚拟文件和虚拟目录的方式提供访问系统参数的机会，所以有人称它为“了解系统信息的一个窗口”。这些虚拟的文件和目录并没有真实地存在在磁盘上，而是内核中各种数据的一种直观表示。虽然是虚拟的，但它们都可以通过标准的系统调用

（open()、read()等）访问。

例如，/proc/meminfo中包含内存使用的信息，可以用cat命令显示其内容：

```
$ cat /proc/meminfo
MemTotal:       384780 kB
MemFree:         13636 kB
Buffers:         13928 kB
Cached:         101680 kB
SwapCached:        132 kB
Active:         207764 kB
Inactive:        45720 kB
SwapTotal:      329324 kB
SwapFree:       329192 kB
Dirty:               0 kB
Writeback:           0 kB
……
```

其实，Linux的很多系统命令就是通过读取/proc实现的。例如uname -a 的部分信息就来自/proc/version，而uptime的部分信息来自/proc/uptime和/proc/loadavg。

关于procfs更多的信息请访问：http://en.wikipedia.org/wiki/Procfs

基本思路

Linux是通过文件系统接口实现procfs，并在启动时自动将其mount到/proc目录上。此目录下的所有内容都是随着系统的运行自动建立、删除和更新的，而且它们完全存在于内存中，不占用任何外存空间。

Linux 0.11还没有实现虚拟文件系统，也就是，还没有提供增加新文件系统支持的接口。所以本实验只能在现有文件系统的基础上，通过打补丁的方式模拟一个procfs。

Linux 0.11使用的是Minix的文件系统，这是一个典型的基于inode的文件系统，《注释》一书对它有详细描述。它的每个文件都要对应至少一个inode，而inode中记录着文件的各种属性，包括文件类型。文件类型有普通文件、目录、字符设备文件和块设备文件等。在内核中，每种类型的文件都有不同的处理函数与之对应。我们可以增加一种新的文件类型——proc文件，并在相应的处理函数内实现procfs要实现的功能。

增加新文件类型

在include/sys/stat.h文件中定义了几种文件类型和相应的测试宏：

```
#define S_IFMT  00170000
#define S_IFREG  0100000        //普通文件
#define S_IFBLK  0060000        //块设备
#define S_IFDIR  0040000        //目录
#define S_IFCHR  0020000        //字符设备
#define S_IFIFO  0010000
……

#define S_ISREG(m)      (((m) & S_IFMT) == S_IFREG)     //测试m是否是普通文件
#define S_ISDIR(m)      (((m) & S_IFMT) == S_IFDIR)     //测试m是否是目录
#define S_ISCHR(m)      (((m) & S_IFMT) == S_IFCHR)     //测试m是否是字符设备
#define S_ISBLK(m)      (((m) & S_IFMT) == S_IFBLK)     //测试m是否是块设备
#define S_ISFIFO(m)     (((m) & S_IFMT) == S_IFIFO)
```

增加新的类型的方法分两步：

1. 定义一个类型宏S_IFPROC，其值应在0010000到0100000之间，但后四位八进制数必须是0（这是S_IFMT的限制，分析测试宏可知原因），而且不能和已有的任意一个S_IFXXX相同；
2. 定义一个测试宏S_ISPROC(m)，形式仿照其它的S_ISXXX(m)

注意，C语言中以“0”直接接数字的常数是八进制数。

让mknod()支持新的文件类型

psinfo结点要通过mknod()系统调用建立，所以要让它支持新的文件类型。直接修改fs/namei.c文件中的sys_mknod()函数中的一行代码，如下：

```
if (S_ISBLK(mode) || S_ISCHR(mode) || S_ISPROC(mode))
     inode->i_zone[0] = dev;
```

文件系统初始化

内核初始化的全部工作是在main()中完成，而main()在最后从内核态切换到用户态，并调用init()。init()做的第一件事情就是挂载根文件系统：

```
void init(void)
{
        ……
        setup((void *) &drive_info);
        ……
}
```

procfs的初始化工作应该在根文件系统挂载之后开始。它包括两个步骤：

1. 建立/proc目录；
2. 建立/proc目录下的各个结点。本实验只建立/proc/psinfo。

建立目录和结点分别需要调用mkdir()和mknod()系统调用。因为初始化时已经在用户态，所以不能直接调用sys_mkdir()和sys_mknod()。必须在初始化代码所在文件中实现这两个系统调用的用户态接口，即API：

```
#include
#define __LIBRARY__
#include

_syscall2(int,mkdir,const char*,name,mode_t,mode)
_syscall3(int,mknod,const char*,filename,mode_t,mode,dev_t,dev)
```

mkdir()时mode参数的值可以是“0755”（rwxr-xr-x），表示只允许root用户改写此目录，其它人只能进入和读取此目录。

procfs是一个只读文件系统，所以用mknod()建立psinfo结点时，必须通过mode参数将其设为只读。建议使用“S_IFPROC|0444”做为mode值，表示这是一个proc文件，权限为0444（r--r--r--），对所有用户只读。

mknod()的第三个参数dev用来说明结点所代表的设备编号。对于procfs来说，此编号可以完全自定义。proc文件的处理函数将通过这个编号决定对应文件包含的信息是什么。例如，可以把0对应psinfo，1对应meminfo，2对应cpuinfo。

如此项工作完成得没有问题，那么编译、运行0.11内核后，用“ll /proc”可以看到：

```
# ll /proc
total 0
?r--r--r--   1 root     root             0 ??? ??  ???? psinfo
```

此时可以试着读一下此文件：

```
# cat /proc/psinfo
(Read)inode->i_mode=XXX444
cat: /proc/psinfo: EINVAL
```

inode->i_mode就是通过mknod()设置的mode。信息中的XXX和你设置的S_IFPROC有关。通过此值可以了解mknod()工作是否正常。这些信息说明内核在对psinfo进行读操作时不能正确处理，向cat返回了EINVAL错误。因为还没有实现处理函数，所以这是很正常的。

这些信息至少说明，psinfo被正确open()了。所以我们不需要对sys_open()动任何手脚，唯一要打补丁的，是sys_read()。

让proc文件可读

open()没有变化，那么需要修改的就是sys_read()了。首先分析sys_read（在文件fs/read_write.c中）：

```
int sys_read(unsigned int fd,char * buf,int count)
{
        struct file * file;
        struct m_inode * inode;
        ……
        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);
        }

        printk("(Read)inode->i_mode=%06o\n\r",inode->i_mode);   //这条信息很面善吧？
        return -EINVAL;
}
```

显然，要在这里一群if的排比中，加上S_IFPROC()的分支，进入对proc文件的处理函数。需要传给处理函数的参数包括：

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

proc文件的处理函数

proc文件的处理函数的功能是根据设备编号，把不同的内容写入到用户空间的buf。写入的数据要从f_pos指向的位置开始，每次最多写count个字节，并根据实际写入的字节数调整f_pos的值，最后返回实际写入的字节数。当设备编号表明要读的是psinfo的内容时，就要按照psinfo的形式组织数据。

实现此函数可能要用到如下几个函数：

malloc()和free()

包含linux/kernel.h头文件后，就可以使用malloc()和free()函数。它们是可以被核心态代码调用的，唯一的限制是一次申请的内存大小不能超过一个页面。

sprintf()

Linux 0.11没有sprintf()，可以参考printf()自己实现一个，如下：

```
#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;
}
```

cat命令

cat是Linux下的一个常用命令，功能是将文件的内容打印到标准输出。它核心实现大体如下：

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char* argv[])
{
        char buf[513] = {'\0'};
        int nread;

        int fd = open(argv[1], O_RDONLY, 0);
        while(nread = read(fd, buf, 512))
        {
                buf[nread] = '\0';
                puts(buf);
        }

        return 0;
}
```