

项目设计文档

1. proc文件系统简介

最新的Linux内核通过文件系统接口实现了proc文件系统，它是一个虚拟文件系统，在Linux启动时就被挂接（mount）到了/proc目录上。proc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的存在于磁盘上，而是在内存中形成了一种对Linux内核数据的直观表示，并且随着操作系统的运行自动建立、删除和更新。虽然是虚拟的，但它们都可以通过标准的文件系统调用来访问（包括read、write函数等）。

2. 项目设计思路

Linux0.11内核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类型的接口。所以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proc文件系统。需要在MINIX 1.0文件系统中**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对proc文件系统的读写等功能。

1. 首先，在 MINIX 1.0 文件系统中为 proc 文件增加一个新文件类型。在 include/sys/stat.h 文件的第 25行后面添加新文件类型的宏定义，其代码如下：

```
#define S_IFPROC 0050000
```

2. 为新文件类型添加相应的测试宏。在第 37 行后面添加测试宏，代码如下：

```
#define S_ISPROC(m) ((m) & S_IFMT) == S_IFPROC)
```

3. 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。在 fs/namei.c 文件中只需修改 mknod 函数中的一行代码即可，即将第 647 行代码修改为如下：

```
if(S_ISBLK(mode) || S_ISCHR(mode) || S_ISPROC(mode))
```

4. proc 文件系统的初始化工作应该在根文件系统被挂载之后开始，包括两个步骤：建立/proc 目录和建立/proc 目录下的各个结点。因为在根文件系统加载后，初始化过程已经进入用户态，就不能调用 sys_mkdir 和 sys_mknod 函数建立目录和节点了，而必须调用 mkdir 和 mknod系统调用函数来建立目录和结点。所以，必须在初始化代码所在文件中实现 mkdir 和 mknod 两个系统调用函数的用户态接口。下面是实现这两个接口的具体过程：

在 init/main.c 文件的第 13 行包含头文件 stat.h：

```
#include<sys/stat.h>
```

在第 42 行后面添加如下代码，定义 mkdir 和 mknod 两个系统调用函数：

```
_syscall12(int, mkdir, const char*, name, mode_t, mode)
_syscall13(int, mknod, const char*, filename, mode_t, mode, dev_t, dev)
```

5. 在根文件系统被挂载之后开始proc文件系统初始化：建立/proc 目录和建立/proc 目录下的各个结点。

在第 234 行复制句柄语句的后面添加如下代码，调用 mkdir 函数创建/proc 目录，调用 mknod 函数创建 psinfo 节点：

```
mkdir("/proc",0755);
mknod("/proc/psinfo",S_IFPROC|0444,0);
```

由于 proc 文件系统对用户态程序来说是只读的，所以将 mkdir 函数的第二个参数 mode 的值设为“0755”(rwxr-xr-x)，表示只允许 root 用户改写此目录，其它用户只能进入和读取此目录。将 mknod 函数的第二个参数 mode 的值设为“S_IFPROC|0444”，表示这是一个 proc 文件，权限为 0444 (r--r--r--)，对所有用户只读。

mknod 函数的第三个参数 dev 用来说明结点所代表的设备编号。对于 proc 文件系统来说，此编号可以完全自定义。proc 文件的处理函数可以通过这个编号决定文件包含的是什么信息。例如，可以把 0 对应 psinfo，1 对应 meminfo，2 对应 memap。

6. 找到 fs/read_write.c 文件中的 sys_read 函数，此函数中有一系列的 if 判断语句，再添加一个 if 语句，代码如下：

```
if(S_ISPROC(inode->i_mode))
    return psread(inode->i_zone[0], buf, count, &file->f_pos);
```

这样，当文件类型为 proc 文件时，就会调用 psread 函数进行了。传递给 psread 函数的参数包括：

- inode->i_zone[0]，这就是之前调用 mknod 函数时指定的 dev ——设备编号。
- buf，用户空间缓冲区，就是应用程序在调用 read 函数时传入的第二个参数，用于存放从文件中读取到的数据。
- count，就是应用程序在调用 read 函数时传入的第三个参数，用于说明 buf 指向的缓冲区的大小。
- &file->f_pos，f_pos 是上一次读文件结束时“文件位置指针”的位置。这里必须传递指针，因为 psread 函数还需要根据读取到的数据量修改 f_pos 的值。

7. 由于这里调用了 psread 函数，所以需要在第 32 行添加如下代码声明此函数：

```
extern int psread (int dev, char * buf, int count, off_t * f_pos);
```

8. psread 函数的源代码在fs/procfs.c文件中。函数源代码如下：

```
int psread(int dev,char * buf,int count,off_t * f_pos)
{
    // printk("dev:%d\n",dev);
    struct task_struct * p;
    int i, j = 0;    //变量j为在psbuffer中写数据的位置偏移量
    unsigned long pte;
    unsigned long* page_dir_base = 0;
    unsigned long* page_base = 0;
    unsigned long log_cnt = task_log_id;
    struct task_log* task_logs = get_logs();

    //第一次从psbuffers数组的起始位置0开始，读取完毕后第二次再进入到此函数就
    //不会再为psbuffer赋值，直接执行后面的for循环体
    if(!(*f_pos))
    {
        memset(psbuffer, 0, SIZE);    //将psbuffer数组初始化为0

        switch(dev)
        {
            case PSINFO:
                //添加psinfo节点的标题
```

```

        j += sprintf(psbuffer+j,
"pid\tppid\tcounter\tstate\tstart_time\n");
        //遍历当前的进程，为psinfo节点赋值
        //NR_TASKS为系统最大的任务数的宏定义
        for(i = 0; i < NR_TASKS; i++)
        {
            p = task[i];
            if(p == NULL) continue;

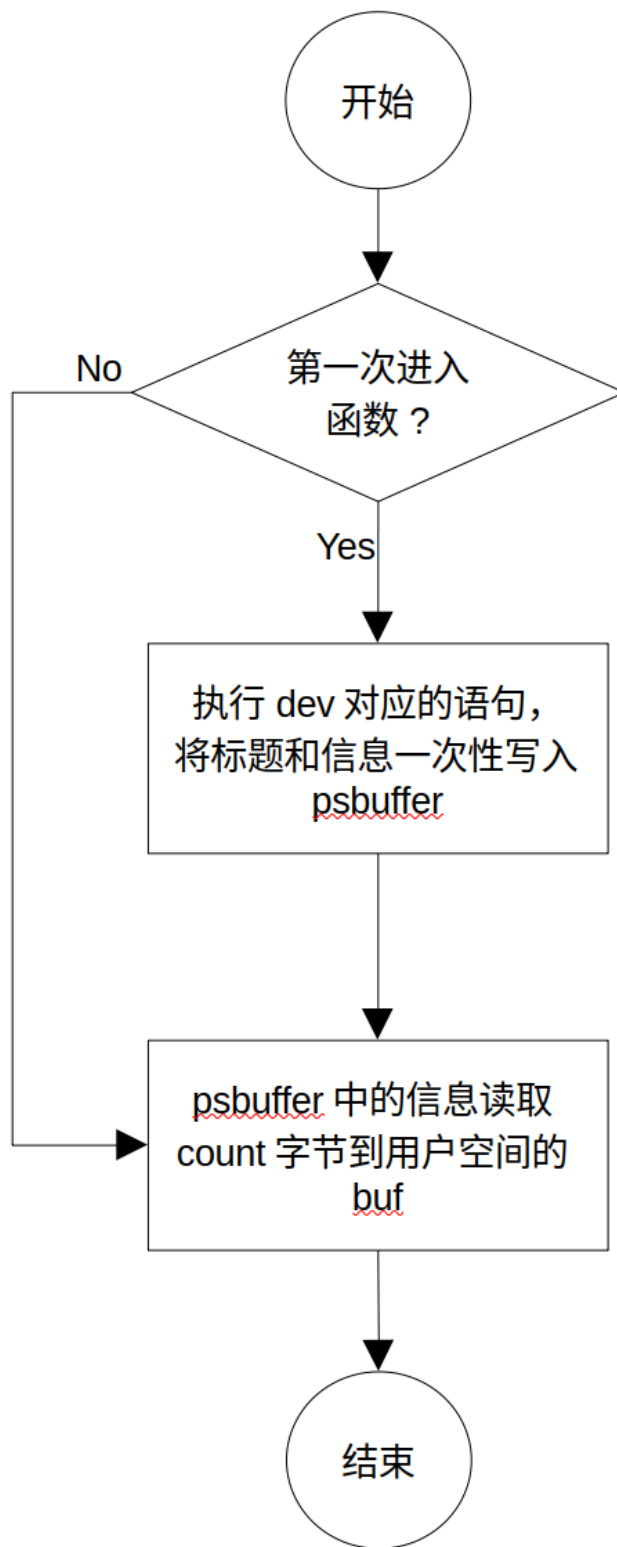
            j += sprintf(psbuffer+j, "%d\t%d\t%d\t%d\t%d\n", p-
>pid, p->father, p->counter, p->state, p->start_time);
        }
        break;

        ... ...
    }
}
//将数组psbuffer中的信息读取到用户空间的buf位置处
for(i = 0; i < count; i++, (*f_pos)++)
{
    if(psbuffer[*f_pos] == '\0')
        break;
    //put_fs_byte函数是将一字节存放在 fs 段中指定内存地址处。
    //其第一个参数是char类型，第二个参数是char*类型
    put_fs_byte(psbuffer[*f_pos], &buf[i]);
}

return i;
}

```

psread函数算法流程图:



题目共要求实现5个proc文件结点，所以采用switch-case控制结构，并且创建枚举类型变量：

```
enum DEV_TYPE
{
    PSINFO, MEMINFO, MMAP, TICKINFO, KMSG,
};
```

在系统空间定义缓冲区，存储节点信息：

```
#define SIZE 4096 //缓冲区大小
char psbuffer[SIZE]; //定义缓冲区来存放结点信息
```

psinfo结点信息获取算法思想：遍历当前的进程控制块列表，将非空的控制块需要的信息写入 **psbuffer+j**

函数中的**sprintf**是参考 **init/main.c** 文件中第 200 行的 **printf** 函数调用内核**vsprintf**接口实现的，并且返回打印的偏移量，源码如下：

```
static int sprintf(char* tar, const char *fmt, ...)
{
    va_list args;
    int i;

    va_start(args, fmt);
    // write(1, printbuf, i=vsprintf(tar, fmt, args));
    i=vsprintf(tar, fmt, args);
    va_end(args);
    return i;
}
```

9. 接着在236行后添加其他四个结点：

```
mknod("/proc/meminfo", S_IFPROC | 0444, 1);
mknod("/proc/mmap", S_IFPROC | 0444, 2);
mknod("/proc/tickinfo", S_IFPROC | 0444, 3);
mknod("/proc/kmsg", S_IFPROC | 0444, 4);
```

10. **psread**函数中**meminfo**部分：

```
case MEMINFO:
    // 添加meminfo节点的标题
    j += sprintf(psbuffer+j, "pageCount\tfreePageCount\tusedPageCount\n");

    int free = get_free_info();

    j += sprintf(psbuffer+j, "%d\t%d\t%d\n", PAGING_PAGES, free,
PAGING_PAGES - free);
    break;
```

其中，**PAGING_PAGES**是内核定义的宏，代表内存总页数：

```
#define PAGING_MEMORY (15*1024*1024) // 分页内存15MB。主内存区最多15M。
#define PAGING_PAGES (PAGING_MEMORY>>12) // 分页后的物理内存页数。
```

get_free_info函数定义在**mm/memory.c**，遍历内核中的**mem_map**数组，记录此时的空闲页数：

```
int get_free_info()
{
    __asm("cli");
    // 添加内存信息
    int i, free = 0;
    for(i = 0; i < PAGING_PAGES; i++)
    {
        if(mem_map[i] == 0)
        {
            free++;
        }
    }
}
```

```

    }
    __asm("sti");

    return free;
}

```

11. psread函数中mmap部分:

```

case MMAP:
    // 参考vm(mm/memory.c)
    j += sprintf(psbuffer+j, "The first 5 values(0x) of
page_dir_table:\n");
    for(i = 0; i < 5; i++)
    {
        pte = page_dir_base[i];
        j += sprintf(psbuffer+j, "%d\t%08lx\n", i, pte);
    }

    j += sprintf(psbuffer+j, "The first 10 values(0x) of page_table mapped
by the second pte:\n");
    pte = page_dir_base[1];
    page_base = (unsigned long*)(pte & 0xFFFFF000);
    for(i = 0; i < 10; i++)
    {
        pte = page_base[i];
        j += sprintf(psbuffer+j, "%d\t%08lx\n", i, pte);
    }

    j += sprintf(psbuffer+j, "The linear address(0x) of physical pages
mapped by these 10 values:\n");
    for(i = 0; i < 10; i++)
    {
        j += sprintf(psbuffer+j, "%d\t%08lx\n", i, (0x00400000 | i<<12));
    }
    break;

```

page_dir_base（其实为0地址）指针记录了页目录表首地址，遍历前5个目录项，格式化记录到psbuffer；page_dir_base[1]为第2个页目录项，其16进制的前5位即为其映射的页表首地址page_base，遍历page_base前10个页表项，格式化记录到psbuffer；0x00400000为第二个页目录项映射的页表的线性地址，即第1个页表项的线性地址，后面的9个页表项只需要在此基础上依次偏移 $i \ll 12$ 。

12. psread函数中tickinfo部分:

```

case TICKINFO:
    j += sprintf(psbuffer+j, "%ld\n", jiffies);

```

直接返回内核中的jiffies变量。

13. psread函数中kmsg部分:

```

...
unsigned long log_cnt = task_log_id;
struct task_log* task_logs = get_logs();
...
case KMSG:

```

```

j += sprintf(psbuffer+j, "pid\tstate\tjiffies\n");

for(i = 0; i < log_cnt; i ++)
{
    if(j >= SIZE)
    {
        return -ENOMEM;
    }
    j += sprintf(psbuffer+j, "%d\t%s\t%d\n", task_logs[i].pid,
map2state(task_logs[i].new_state), task_logs[i].jiffies);
}

break;

```

其中：

```

if(j >= SIZE)
{
    return -ENOMEM;
}

```

是为了防止进程状态转换信息过多，造成指针访问到psbuffer外的内存。

get_logs和task_log_id声明在linux/sched.h。

```

extern int task_log_id;
struct task_log
{
    long pid;
    long new_state;
    long jiffies;
};
extern struct task_log* get_logs(void);

```

要想记录进程状态转换过程，只需要在进程调度过程中，每次状态转换都把起始和结果状态记录到内存中即可。

定义数据结构：

```

struct task_state_entry task_trans_table[MAX_TASK_TRANS];
struct task_log task_logs[MAX_TASK_TRANS];
long task_trans_count = 0;
long real_trans_count = 50;
// int first_five_pids[MAX_P];           // 记录前5个pid
// int pids_init = 0;                    // first_five_pids是否被初始化
int task_log_id = 0;                      // task_logs下一条记录的索引

```

函数get_logs提供给psread函数获取数据结构task_logs一个接口，task_logs记录了需要的信息：

```

struct task_log* get_logs(void)
{
    return task_logs;
}

```

主要的函数逻辑：

```

inline void record_task_log(long pid, long new_state, long jiffies)
{
    // int i = 0;

    // linux0.11前5个进程是固定的, pid为0-4
    if(pid < 0 || pid > 4) return;

    if(task_log_id >= MAX_TASK_TRANS)
        return;

    // 记录日志
    task_logs[task_log_id].pid = pid;
    task_logs[task_log_id].new_state = new_state;
    task_logs[task_log_id].jiffies = jiffies;

    task_log_id++;
}

```

只需要在调度函数转换进程状态后紧接着调用函数record_task_log并且传入正确的参数即可。比如, 进程就绪:

```

RECORD_TASK_LOG((*p)->pid, TS_READY, jiffies); // 宏RECORD_TASK_LOG

```

```

#define RECORD_TASK_LOG(pid, state, jiffies) record_task_log(pid, state,
jiffies); // 定义于linux/sched.h

```

至此, 基本完成了赛题要求的五个proc结点。下面为内核编译后, 使用cat命令输出的结果。

3. 运行结果

1. psinfo

```

[/usr/root]# cat /proc/psinfo
pid      state   father  counter  start_time
0         0       -1      0         0
1         1       0       27        1
4         1       1       5         73
3         1       1       24        67
6         0       4       12       26229

```

2. meminfo

```

[/usr/root]# cat /proc/meminfo
pageCount      memCount      freePageCount  usedPageCount
3840           15728640      2959           881
[/usr/root]# ls -l /proc
total 0
-r--r--r--  1 root    root      0 ??? ??  ??? meminfo
-r--r--r--  1 root    root      0 ??? ??  ??? psinfo
[/usr/root]#

```

3. mmap

```

The first 5 values(0x) of page_dir_table:
0    00001027
1    00002007
2    00003007

```



```

3 00004027
4 00000000
The first 10 values(0x) of page_table mapped by the second pte:
0 00400007
1 00401007
2 00402007
3 00403007
4 00404007
5 00405007
6 00406007
7 00407007
8 00408007
9 00409007
The linear address(0x) of physical pages mapped by these 10 values:
0 00400000
1 00401000
2 00402000
3 00403000
4 00404000
5 00405000
6 00406000
7 00407000
8 00408000
9 00409000

```

4. tickinfo

```

[/usr/root]# ls -l /proc
total 0
?r--r--r-- 1 root root 0 ??? ?? ???? meminfo
?r--r--r-- 1 root root 0 ??? ?? ???? psinfo
?r--r--r-- 1 root root 0 ??? ?? ???? tickinfo
[/usr/root]# cat /proc/tickinfo
42966

```

5. kmsg 设置MAX_TASK_TRANS为300, 即记录前300条状态转换信息

```

pid state jiffies
0 Ready 1
1 Running 1
1 Wait 1
0 Running 1
1 Ready 2
0 Ready 2
1 Running 2
1 Wait 2
0 Running 2
1 Ready 2
0 Ready 2
1 Running 2
1 Wait 2
0 Running 2
1 Ready 2
0 Ready 2
1 Running 2
1 Wait 2
0 Running 2
1 Ready 2
0 Ready 2
1 Running 2

```

1	Wait	2
0	Running	2
1	Ready	2
0	Ready	2
1	Running	2
1	Wait	2
0	Running	2
1	Ready	2
0	Ready	2
1	Running	2
1	Wait	2
0	Running	2
1	Ready	2
0	Ready	2
1	Running	2
1	Wait	2
0	Running	2
1	Ready	2
0	Ready	2
1	Running	2
1	Wait	3
0	Running	3
1	Ready	3
0	Ready	3
1	Running	3
1	Wait	3
0	Running	3
1	Ready	3
0	Ready	3
1	Running	3
1	Wait	3
0	Running	3
1	Ready	3
0	Ready	3
1	Running	3
1	Wait	3
0	Running	3
1	Ready	3
0	Ready	3
1	Running	3
1	Wait	3
0	Running	3
1	Ready	3
0	Ready	3
1	Running	3
1	Wait	3
0	Running	3
1	Ready	3
0	Ready	3
1	Running	3
1	Wait	49
0	Running	49
1	Ready	49
0	Ready	49
1	Running	49
1	Wait	49
0	Running	49
1	Ready	49

0	Ready	49
1	Running	49
1	Wait	49
0	Running	49
1	Ready	49
0	Ready	49
1	Running	49
1	Wait	49
0	Running	49
1	Ready	49
0	Ready	50
1	Running	50
1	Wait	50
0	Running	50
1	Ready	50
0	Ready	50
1	Running	50
1	Wait	50
0	Running	50
1	Ready	50
0	Ready	50
1	Running	50
2	Running	51
2	Wait	51
0	Running	51
2	Ready	51
0	Ready	51
2	Running	51
2	Wait	51
0	Running	51
2	Ready	51
0	Ready	51
2	Running	51
2	Wait	52
0	Running	52
2	Ready	52
0	Ready	52
2	Running	52
2	Wait	52
0	Running	52
2	Ready	52
0	Ready	52
2	Running	52
2	Wait	58
0	Running	58
2	Ready	58
0	Ready	58
2	Running	58
2	Wait	64
0	Running	64
2	Ready	64
0	Ready	64
2	Running	64
1	Ready	73
1	Running	73
3	Running	74
3	Wait	74
4	Running	74

3	Ready	74
4	Wait	80
3	Running	80
4	Ready	80
3	Wait	81
4	Running	81
3	Ready	81
4	Wait	84
3	Running	84
4	Ready	84
3	Wait	85
4	Running	85
3	Ready	85
4	Wait	85
3	Running	85
4	Ready	85
3	Wait	89
4	Running	89
4	Wait	89
0	Running	89
4	Ready	89
0	Ready	89
4	Running	89
4	Wait	94
0	Running	94
4	Ready	94
0	Ready	94
4	Running	94
4	Wait	95
0	Running	95
4	Ready	95
0	Ready	95
4	Running	95
4	Wait	96
0	Running	96
4	Ready	96
0	Ready	96
4	Running	96
4	Wait	97
0	Running	97
4	Ready	97
0	Ready	97
4	Running	97
4	Wait	98
0	Running	98
4	Ready	98
0	Ready	98
4	Running	98
4	Wait	99
0	Running	99
4	Ready	99
0	Ready	99
4	Running	99
4	Wait	111
4	Ready	113
4	Running	114
4	Wait	114
0	Running	114

4	Ready	114
0	Ready	115
4	Running	115
4	Wait	116
0	Running	116
4	Ready	116
0	Ready	116
4	Running	116
4	Wait	116
0	Running	116
4	Ready	116
0	Ready	116
4	Running	116
4	Wait	120
0	Running	120
3	Ready	3088
0	Ready	3088
3	Running	3088
3	Wait	3088
0	Running	3088
4	Ready	249661
0	Ready	249661
4	Running	249661
4	Wait	249661
0	Running	249661
4	Ready	249727
0	Ready	249727
4	Running	249727
4	Wait	249727
0	Running	249727
4	Ready	249863
0	Ready	249863
4	Running	249863
4	Wait	249863
0	Running	249863
4	Ready	249863
0	Ready	249863
4	Running	249863
4	Wait	249863
0	Running	249863
4	Ready	250591
0	Ready	250591
4	Running	250591
4	Wait	250591
0	Running	250591
4	Ready	250669
0	Ready	250669
4	Running	250669
4	Wait	250669
0	Running	250669
4	Ready	250946
0	Ready	250946
4	Running	250946
4	Wait	250946
0	Running	250946
4	Ready	251137
0	Ready	251137
4	Running	251137

4	Wait	251137
0	Running	251137
4	Ready	251443
0	Ready	251443
4	Running	251443
4	Wait	251443
0	Running	251443
4	Ready	251655
0	Ready	251655
4	Running	251655
4	Wait	251655
0	Running	251655
4	Ready	252067
0	Ready	252067
4	Running	252067
4	Wait	252067
0	Running	252067
4	Ready	252112
0	Ready	252112
4	Running	252112
4	Wait	252112
0	Running	252112
4	Ready	252640
0	Ready	252640
4	Running	252640
4	Wait	252641
0	Running	252641
4	Ready	252794
0	Ready	252794
4	Running	252794
4	Wait	252794
0	Running	252794
4	Ready	254080
0	Ready	254080
4	Running	254080
4	Wait	254080
0	Running	254080
4	Ready	254207
0	Ready	254207
4	Running	254207
4	Wait	254207
0	Running	254207
4	Ready	254250
0	Ready	254250
4	Running	254250
4	Wait	254250