

# Project 1

---

报告主要分为四个部分：

- 任务介绍
- 必备知识（Linux内核代码结构，`/proc` 文件系统，C语言操作文件与目录：`<dirent.h>`
- 任务实现与结果
- 遇到的问题及解决方案
- `proc_read` 函数的机制测试。这部分提出了我自己仍有的一些困惑。

## Project 1

任务介绍

任务一：简易计算器

任务二：模拟ps指令

Linux 内核代码结构

内核模块的出入口：`__init` 和 `__exit` 宏

添加命令行参数

`/proc` 文件系统

`/proc` 简介

`proc_ops` 结构

操作 `/proc` 文件系统的内核代码

在 `/proc` 下操作文件

用户空间和内核空间

`/proc` 文件系统的读写处理函数

C语言操作文件与目录：`<dirent.h>`

开路径 (`cd <- DIR *opendir (const char *filename)`)

`struct DIR`

打开路径函数 `opendir`

读取路径 (`ls <- readdir`)

`dirent` 结构体

`readdir` 函数

任务实现与结果

任务一：简易计算器

任务二：模拟ps指令

遇到的问题及解决方案

`proc_read` 函数的机制测试

## 任务介绍

---

### 任务一：简易计算器

1. 编写带命令行参数的内核模块
2. 可以通过 `\proc` 文件系统进行读取 和写入

### 任务二：模拟ps指令

模拟 `ps` 指令，读取进程的 `PID`，进程状态，进程的命令行参数 三列信息

输出效果类似 `ps -e -ww -o pid:5,state,cmd`。

## Linux 内核代码结构

---

## 内核模块的出入口：\_\_init 和 \_\_exit 宏

带 \_\_init 的函数是程序的入口，是内核的初始化函数，内核在被加载时会执行它。对于内置模块（不可加载的）模块来说，带 \_\_init 的函数执行完以后会释放内存。

带 \_\_exit 的函数是程序的出口，是内核的清理函数，内核在被删除时会执行它。对于内置模块（不可加载的）模块来说，带 \_\_exit 的函数不会得到执行。

```
1 static int __init proc_init(void) //程序入口
2 {
3     /* TODO */
4 }
5
6 static void __exit proc_exit(void) //程序出口
7 {
8     /* TODO */
9 }
```

## 添加命令行参数

可以向内核添加命令行参数，但是内核程序的命令行参数并不是使用 argc 和 argv 来添加，而是需要使用 make\_param() 来向内核“注册”，如下所示

```
1 static int operand1;
2 module_param(operand1, int, 0);
```

module\_param 接受三个参数，分别为：1.变量名 2.变量类型 3.权限（一个数字代表一个权限）。

同样可以使用数组 或 字符串 作为命令行参数，此时需要使用 module\_param\_array() 或 module\_param\_string() 来“注册”。

**添加字符串参数**可以使用两种方法：1) **将字符串作为字符指针注册**，定义 char \* 变量，作为 charp(char pointer)类型传入 module\_param；2) **显式注册字符串**，定义 char 数组，指定长度，使用 module\_param\_string：

module\_param\_string(name, C string, len, perm)，接受四个参数：

- name 命令行参数名
- C string 程序种实际存储命令行参数的变量名
- len 字符串的长度
- perm 权限

**实验种实际使用第一种方法，不需要考虑字符串越界的情况。**

```
1 /* 字符串 方法一：不需要指定字符串大小 */
2 static char *operator;
3 module_param(operator, charp, 0);
4
5 /* 字符串 方法二：指定字符串长度 */
6 const int LEN = 10;
7 static char operator[LEN];
8 module_param_string(<outer_name>, operator, LEN, 0)
```

**添加数组参数：**将数组传入 module\_param\_array 函数，数组“注册”时需要多传入一个指向 count 的指针，count 变量用于记录 operand2 的元素个数。

```
1  /* 数组 */
2  static int operand2[128];
3  static int count;
4  module_param_array(operand2, int, &count, 0);
```

## /proc 文件系统

### /proc 简介

/proc 文件系统的名字是process（进程）的简写，是内核（内核模块）与其他进程进行通信的机制。不同于一般位于磁盘(Disk)上的文件系统，/proc 文件系统完全位于内存(Memory)中，因此当计算机重启时 /proc 文件系统中的内容都会清空。

### proc\_ops 结构

proc\_ops 结构体的作用是告诉内核：当对应的文件系统的被读取（或写入）时执行什么对应的函数。注意：proc\_ops 是在内核版本v5.6+定义，之前版本使用 file\_operations，它比 proc\_ops 有更多冗余。

## 操作 /proc 文件系统的内核代码

### 在 /proc 下操作文件

在 /proc 文件系统下创建文件（例如 /proc/helloworld）需要在内核模块的入口 函数中使用 proc\_create 函数进行创建，它返回一个结构体(struct) proc\_dir\_entry，这个文件可以被用来配置 /proc/helloworld，如果 proc\_create 返回了 NULL 就说明创建不成功。

### 用户空间和内核空间

Linux 内存(在Intel体系结构上)是分段的，这意味着指针本身并不指向内存中的唯一位置，只指向内存段中的一个位置，而不同的内存段有不同权限。内核有一个属于内核的段，每个进程都有自己的段，他们只能访问自己的段。

这一点在写单个线程的代码时无关紧要，但是在写内核代码时就需要考虑到。用户读写 /proc 文件系统时是在用户空间发出信息，换言之是将一个指向用户段的指针传给了内核，此时内核无法访问指针指向的空间，就需要使用 copy\_from\_user 或者 get\_user 来授予内核访问权限。

### /proc 文件系统的读写处理函数

1. proc\_read() 函数在 proc\_ops 结构中“注册”，会在用户读取（例如 cat 指令）/proc 文件系统的对应模块时得到执行。需要注意的是用户读取 /proc 时要使用 copy\_to\_user 将变量从内核空间移动到用户空间。
2. proc\_write() 函数在 proc\_ops 结构中“注册”，会在用户写入（例如 echo 指令）/proc 文件系统的对应模块的时候得到执行。需要注意的是用户读取 /proc 时要使用 copy\_from\_user 或 get\_user 函数将变量从用户空间移动到内核空间。

## C语言操作文件与目录：<dirent.h>

## 开路径 (`cd <- DIR *opendir (const char *filename)`)

### struct DIR

结构体 DIR 是操作一个路径的句柄 (handler)，它的具体结构对于用户来说是不可见的。在 <dirent.h> 的源文件中是这样说的：

### 打开路径函数 opendir

函数声明如下(易理解的版本)

```
1 | DIR *opendir (const char *filename)
```

opendir 函数将一个路径打开，它提取出了传入路径的相关信息，输入参数为一个表示路径的 C 字符串，返回一个 DIR 结构体的地址。

以下这段代码调用 opendir 函数打开 /proc 路径：

```
1 | /* TODO */
2 | DIR *dir; //定义DIR指针用于接受打开的路径
3 | char proc_path[]="/proc"; //要打开的路径名
4 | dir = opendir(proc_path); //打开路径
```

## 读取路径 (`ls <- readdir`)

### dirent 结构体

dirent 结构体是 readdir (read directory) 函数的返回值，它包含了从路径中读取到的信息。

dirent 结构体所含的内容如下：

```
1 | struct dirent
2 | {
3 |     __ino_t d_ino;    //索引节点号
4 |     __off_t d_off;    //节点在目录中的偏移量
5 |
6 |     unsigned short int d_reclen; //文件名的长度
7 |     unsigned char d_type;        //文件类型
8 |     char d_name[256];           //文件名
9 | };
```

每一个 dirent 结构体保存了一个文件（或路径）的信息，其中路径的文件类型为 `dtype=='EOF'` (ASCII码为4)，文件的文件类型为 `d_type=='BS'` (ASCII为8)。

### readdir 函数

readdir 函数读取一个路径（代表路径的 DIR 结构体的地址），输入参数为一个表示路径的 DIR 结构体指针，返回一个 dirent 结构体的地址。

```
1 | struct dirent *readdir (DIR *dir_pointer)
```

readdir 读取一个路径时流式读取，即第一次调用读取第一个文件，第二次调用读取第二个文件，以此类推，如果连读读取两次（如下图）

```

1  /**readdir***/
2  struct dirent * filename_1;
3  struct dirent * filename_2;
4  filename_1 = readdir(dir_pointer); // 第一次读取
5  filename_2 = readdir(dir_pointer); // 第二次读取
6  printf("filename:%-10s\td_type:%d\t d_reclen:%us\n",
7         filename_1->d_name,filename_1->d_type,filename_1->d_reclen);
8  printf("filename:%-10s\td_type:%d\t d_reclen:%us\n",
9         filename_2->d_name,filename_2->d_type,filename_2->d_reclen);
10

```

则会读取到路径下的第一、第二个文件。如果要**读取所有文件**，一般使用一个 `while` 循环：

```

1  /* some code */
2      /** readdir ***/
3      struct dirent * filename;
4      while ((filename = readdir(dir_pointer))) //读取完路径以后会跳出while循环
5      {
6          printf("filename:%-10s\td_type:%d\t d_reclen:%us\n",
7                 filename->d_name,filename->d_type,filename->d_reclen);
8      }
9  /* some code */

```

注意：while中的赋值语句一定要再用一个括号括起来，这是为了显示告诉编译器：**我真的是要使用赋值语句，而不是把==写成了=。**

## 任务实现与结果

### 任务一：简易计算器

#### 实现方法

##### proc\_init

1. 在 `/proc` 下创建以学号命名的路径: `proc_mkdir`
2. 在 `/proc/519021911058` 下创建创建文件 `calc`: `proc_create`
3. 返回 0, 表示成功初始化模块

##### proc\_exit

1. 删除 `/proc/519021911058` 整个路径: `proc_remove`

##### proc\_read

1. 递归结束判断：如果偏移量 `*pos` 超过了应输出字符长度，表示读取完成，则打印日志并返回
2. 判断操作符 `operator`，执行对应操作或输出错误信息
3. 将计算出来的结果（int数组）转换为字符串
4. 计算输出字符串的长度 `out_len`
5. 将结果从内核空间拷贝到用户空间并自增偏移量 `*pos`: `copy_to_user`

##### proc\_write

1. 定义本地字符串并将输入从用户空间拷贝到内核空间: `copy_from_user`
2. 处理拷贝后的字符串
3. 将字符串转为整数 `int` 并赋值给操作数1 `operand1`: `simple_strtoll`

#### 结果

```

→ project1 git:(main) X dmesg
→ project1 git:(main) X sudo insmod calc.ko operand1=2 operand2=1,2,3,4,5 operator=add
→ project1 git:(main) X cat /proc/519021911058/calc
3,4,5,6,7
→ project1 git:(main) X echo 3 > /proc/519021911058/calc
→ project1 git:(main) X cat /proc/519021911058/calc
4,5,6,7,8
→ project1 git:(main) X sudo rmmod calc
→ project1 git:(main) X sudo insmod calc.ko operand1=2 operand2=1,2,3,4,5 operator=mul
→ project1 git:(main) X cat /proc/519021911058/calc
2,4,6,8,10
→ project1 git:(main) X echo 3 > /proc/519021911058/calc
→ project1 git:(main) X cat /proc/<ID>/calc
zsh: no such file or directory: ID
→ project1 git:(main) X cat /proc/519021911058/calc
3,6,9,12,15
→ project1 git:(main) X sudo rmmod calc
→ project1 git:(main) X

```

## 任务二：模拟ps指令

### 实现方法

1. 定义 `DIR*` 并使用 `opendir()` 函数打开 `/proc` 路径。
2. 重复使用 `readdir()` 函数读取路径下的文件名，即执行如下循环。循环完成后执行5.输出进程信息

```

1 while (/*读取/proc下的一个文件*/) {
2     /*处理 `/proc`下的每个文件*/
3 }

```

3. 对于2.中的每一个循环读取到的文件名，首先判断它是不是一个代表进程的路径，如果不是则进行下一次循环，如果是则执行4.
4. 从 `cmdline` 文件中读取命令行信息，从 `status` 文件中读取状态 `state` 和文件名 `Name` .
5. 输出进程信息。输出时判断 `cmdline` 中读取到的内容是否为空，不为空则输出，为空则输出 [`<Name>`]

结果：仅部分

```

→ project1 git:(main) X ./ps
PID S CMD
  1 S /sbin/init splash
  2 S [kthreadd]
  3 I [rcu_gp]
  4 I [rcu_par_gp]
  6 I [kworker/0:0H-events_highpri]
  7 I [kworker/0:1-events]
  8 I [kworker/u24:0-events_unbound]
  9 I [mm_percpu_wq]
 10 S [rcu_tasks_rude_]
 11 S [rcu_tasks_trace]

```

## 遇到的问题及解决方案

1. 内核中无法使用 `atoi` 和 `iota`，可以使用 `snprintf` 和 `simple_strtoll`（及其一系列函数[link](#)）来替代。
2. 错误判断时不能返回-EFAULT，因为不知道调用者会得到错误码以后会干什么。可以使用 `pr_err` 打印错误信息后正常返回。
3. `proc_read` 要返回0，`proc_write` 不能返回0，否则都会持续执行

4. `proc_write`会导致最后带有一个'\n', `len`变量也计算了这个换行符, 需要在 `copy\_from\_user` 以后显示换为'\0', `len`=有效长度+'\0'的长度。 `strlen`会计入\0
5. `cmdline` 中会将字符串不分行存储, 例如 `"string1\0string2\0"`, 需要按字符读取并使用空格替换 '\0'。

最大的问题: `proc_read`提前`return`了但是后面的还是得到执行

第一次`cat`的时候, `pr_err`语句得不到执行

为什么官方代码不直接返回0, 而是等到第二次再返回0; 即为什么`proc_read`一定要执行两次

<https://sysprog21.github.io/lkmpg/#the-procps-structure>

## proc\_read 函数的机制测试

我在学习 `proc_read` 的机制时并未找到讲解清楚的文档, [The Linux Kernel Module Programming Guide](#)中也没有系统如何调用 `proc_read` 的整个机制的讲解。于是我不得不多次实验来探索 `proc_read` 的运行机制。

但是, 我在运行时遇到了一些无法理解的情况, 因此记录在下。

异常情况可以总结为

1. `proc_read` 返回了以后, `return` 之后的 `pr_info` 会在下一次 `cat` 的时候得到执行
2. 无法在一次 `proc_read` 中两次使用 `copy_to_user`
3. 猜想 `proc_read` 的返回值表示读取的总字节数。因此不能第一次 `proc_read` 就返回0, 而要先返回总字节数, 然后系统会再调用一次 `proc_read`, 此时再返回0。是否返回0通过 `*pos` 来判断。

```
1 static ssize_t proc_read(struct file *fp, char __user *ubuf, size_t len,
2   loff_t *pos)
3 {
4     /* TODO */
5     pr_info("pos 1:%lld", *pos);
6     char s[13] = "Helloworld!\n";
7     int l = sizeof(s);
8     ssize_t ret = l;
9     // pr_info("len: %ld", len);
10    if (*pos >= 26 || copy_to_user(ubuf, s, l)) {
11        pr_info("pos 2:%lld", *pos);
12        pr_info("copy_to_user failed\n");
13        ret = 0;
14    } else {
15        pr_info("procfile read %s\n", fp->f_path.dentry->d_name.name);
16        pr_info("pos 3:%lld", *pos);
17        *pos += l;
18    }
19    pr_info("pos 4:%lld", *pos);
20    return ret;
21 }
```

按序执行

```
1 sudo insmod calc.ko
2 cat /proc/519021911058/calc
3 dmesg #观察到正常退出, 最后一个输出为pos2
4 cat /proc/519021911058/cal
5 dmesg # 观察到先输出了pos4, 然后再输出pos1。提前return了但是后面的pr_info依然得到执行
```

使用这个代码，cat没有输出，而且最后一个打印\*pos的指令总是被忽略掉。

```
1 static ssize_t proc_read(struct file *fp, char __user *ubuf, size_t len,
2 loff_t *pos)
3 {
4     /* TODO */
5     if(*pos>=26){
6         return 0
7     }
8     pr_info("pos 1:%lld", *pos);
9     char s[13] = "HelloWorld!\n";
10    int l = sizeof(s);
11    ssize_t ret = l;
12    // pr_info("len: %ld", len);
13
14    if (copy_to_user(ubuf, s, l))
15        pr_info("error");
16    *pos+=1;
17
18    pr_info("pos 2:%lld", *pos);
19
20    if (copy_to_user(ubuf, s, l))
21        pr_info("error");
22    *pos+=1;
23
24    pr_info("pos 3:%lld", *pos);
25
26    pr_info("pos 4:%lld", *pos);
27    pr_info("pos 5:%lld", *pos);
28    pr_info("pos 6:%lld", *pos);
29    return 26;
30    // 猜想proc_read的return代表了一次copy_to_user的总大小【猜的不对】
31    // 使用vim打开calc后观察到是一串[helloworld^@]后面跟了13个^@
32 }
```

按序执行

```
1 sudo insmod calc.ko
2 cat /proc/519021911058/calc # 观察到只输出了一个helloworld
3 dmesg #观察pos6没有输出
4 cat /proc/519021911058/cal # 观察到只输出了一个helloworld
5 dmesg # 观察到先输出了pos6，然后再输出pos1
```

使用这个代码:

```
1 static ssize_t proc_read(struct file *fp, char __user *ubuf, size_t len,
2 loff_t *pos)
3 {
4     pr_info("pos 1:%lld", *pos);
5     char s[13] = "HelloWorld!\n";
6     int l = sizeof(s);
7     ssize_t ret = l;
8     /* TODO */
9     if(*pos>=26){
10         return 0;
11     }
12 }
```



```

11     if (*pos >=13)
12     {
13         copy_to_user(ubuf, s, 13);
14         *pos+=13;
15         return 1; // 主要问题应该在这里，这也是我猜想proc_read返回值表示一次从user，
copy到kernel的字节数的原因
16     }
17
18     copy_to_user(ubuf, s, 13);
19     *pos+=13;
20
21     return ret;
22 }

```

第一次helloworld完整插入，第二次只返回了一个H，但是使用vim打开/proc文件系统calc文件时发现两个helloworld都完整插入

使用这个代码

```

1  static ssize_t proc_read(struct file *fp, char __user *ubuf, size_t len,
loff_t *pos)
2  {
3      pr_info("pos 1:%lld", *pos);
4      char s[13] = "Helloworld!\n";
5      int l = sizeof(s);
6      ssize_t ret = 1;
7      /* TODO */
8      if(*pos>=26){
9          return 0;
10     }
11     if (*pos >=13)
12     {
13         copy_to_user(ubuf, s, 13);
14         *pos+=13;
15         return ret;
16     }
17
18     copy_to_user(ubuf, s, 13);
19     *pos+=13;
20
21     return ret;
22 }

```

可以完整插入。proc\_read 的返回值很关键。