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
> - 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
  - 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，时内核的清理函数，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。

```
static int __init proc_init(void) //程序入口
{
    /* TODO */
}

static void __exit proc_exit(void) //程序出口
{
    /* TODO */
}
```

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

```
static int operand1;
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`madule_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。

```
/* 字符串 方法一：不需要指定字符串大小 */
static char *operator;
module_param(operator, charp, 0);

/* 字符串 方法二：指定字符串长度 */
const int LEN = 10;
static char operator[LEN];
module_param_string(<outer_name>, operator, LEN, 0)
```

添加数组参数：将数组传入`module_para_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。**

```
/* 数组 */
static int operand2[128];
static int count;
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时不可见的. 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)

```
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的信息, 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：

```
/* TODO */
DIR *dir; //定义DIR指针用于接受打开的路径
char proc_path[]="/proc"; //要打开的路径名
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：

```
struct dirent
   {
    __ino_t d_ino;    //索引节点号
    __off_t d_off;    //节点在目录中的偏移量

    unsigned short int d_reclen; //文件名的长度
    unsigned char d_type;        //文件类型
    char d_name[256];            //文件名
   };
```

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.

```
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）

```
/***readdir***/
struct dirent * filename_1;
struct dirent * filename_2;
filename_1 = readdir(dir_pointer); // 第一次读取
filename_2 = readdir(dir_pointer); // 第二次读取
printf("filename:%-10s\td_type:%d\t d_reclen:%us\n",
       filename_1->d_name,filename_1->d_type,filename_1->d_reclen);
printf("filename:%-10s\td_type:%d\t d_reclen:%us\n",
       filename_2->d_name,filename_2->d_type,filename_2->d_reclen);

```

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：

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

任务一：简易计算器

`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`

任务二：模拟ps指令

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

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

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>]

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-procops-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。

```
static ssize_t proc_read(struct file *fp, char __user *ubuf, size_t len,
loff_t *pos)
{
    /* TODO */
    pr_info("pos 1:%lld", *pos);
    char s[13] = "HelloWorld!\n";
    int l = sizeof(s);
    ssize_t ret = l;
    // pr_info("len: %ld", len);
    if (*pos >= 26 || copy_to_user(ubuf, s, l)) {
        pr_info("pos 2:%lld", *pos);
        pr_info("copy_to_user failed\n");
        ret = 0;
    } else {
        pr_info("procfile read %s\n", fp->f_path.dentry->d_name.name);
        pr_info("pos 3:%lld", *pos);
        *pos += l;
    }
    pr_info("pos 4:%lld", *pos);
    return ret;
}
```

按序执行

```
sudo insmod calc.ko
cat /proc/519021911058/calc
dmesg #观察到正常退出，最后一个输出为pos2
cat /proc/519021911058/cal
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。

```
static ssize_t proc_read(struct file *fp, char __user *ubuf, size_t len,
loff_t *pos)
{
    /* TODO */
    if(*pos>=26){
        return 0
    }
    pr_info("pos 1:%lld", *pos);
    char s[13] = "HelloWorld!\n";
    int l = sizeof(s);
    ssize_t ret = l;
    // pr_info("len: %ld", len);

    if (copy_to_user(ubuf, s, l))
        pr_info("error");
    *pos+=l;

    pr_info("pos 2:%lld", *pos);

```

```
    if (copy_to_user(ubuf, s, l))
        pr_info("error");
    *pos+=l;

    pr_info("pos 3:%lld", *pos);

    pr_info("pos 4:%lld", *pos);
    pr_info("pos 5:%lld", *pos);
    pr_info("pos 6:%lld", *pos);
    return 26;
    // 猜想proc_read的return代表了一次copy_to_user的总大小【猜的不对】
    // 使用vim打开calc后观察到是一串[helloworld^@]后面跟了13个^@
}
```

按序执行

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

使用这个代码:

```
static ssize_t proc_read(struct file *fp, char __user *ubuf, size_t len,
loff_t *pos)
{
    pr_info("pos 1:%lld", *pos);
    char s[13] = "HelloWorld!\n";
    int l = sizeof(s);
    ssize_t ret = l;
    /* TODO */
    if(*pos>=26){
        return 0;
    }
    if (*pos >=13)
    {
        copy_to_user(ubuf, s, 13);
        *pos+=13;
        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
    }

    copy_to_user(ubuf, s, 13);
    *pos+=13;

    return ret;
}
```

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

```
static ssize_t proc_read(struct file *fp, char __user *ubuf, size_t len,
loff_t *pos)
```

```
{
    pr_info("pos 1:%lld", *pos);
    char s[13] = "HelloWorld!\n";
    int l = sizeof(s);
    ssize_t ret = l;
    /* TODO */
    if(*pos>=26){
        return 0;
    }
    if (*pos >=13)
    {
        copy_to_user(ubuf, s, 13);
        *pos+=13;
        return ret;
    }

    copy_to_user(ubuf, s, 13);
    *pos+=13;

    return ret;
}
```

可以完整插入。`proc_read`的返回值很关键。