On The first post we built a Simple Kernel Module with init and exit functions and covered the basic concepts in kernel programming

Next, We added a Kernel Module Parameters to configure the kernel module data. In this post, We will create the first interface to user space application using procfs (/proc) file

Proc File System

Proc is a pseudo file system for interfacing with the kernel internal data structures. As a user, you can use proc files for system diagnostics – CPU, memory, Interrupts and many more. You can also configure a lot of parameters like scheduler parameters, kernel objects, memory and more

The common interaction with proc is using cat and echo from the shell. For example:

```
1# cat /proc/cpuinfo
2# echo "50"> /proc/sys/kernel/sched_rr_timeslice_ms
```

Creating a new Proc file

To create a proc file system we need to implement a simple interface – file_operation. We can implement more than 20 functions but the common operations are read, write. To register the interface use the function proc_create

The basic structure is:

```
 1#include <linux/module.h>
 2#include <linux/moduleparam.h>
 3#include <linux/init.h>
 4#include <linux/kernel.h>
 5#include <linux/proc_fs.h>
 6#include <asm/uaccess.h>
 7#define BUFSIZE  100
 8
 9
10MODULE_LICENSE("Dual BSD/GPL");
11MODULE_AUTHOR("Liran B.H");
12
13
14 static struct proc_dir_entry *ent;
15
16 static ssize_t mywrite(struct file *file, const char __user *ubuf,size_t count, loff_t *ppos)
17 {
```

```
18          printk( KERN_DEBUG "write handler\n");
19          return -1;
20}
21
22static ssize_t myread(struct file *file, char __user *ubuf,size_t count, loff_t *ppos)
23{
24          printk( KERN_DEBUG "read handler\n");
25          return 0;
26}
27
28static struct file_operations myops =
29{
30          .owner = THIS_MODULE,
31          .read = myread,
32          .write = mywrite,
33};
34
35static int simple_init(void)
36{
37          ent=proc_create("mydev",0660,NULL,&myops);
38          return 0;
39}
40
41static void simple_cleanup(void)
42{
43          proc_remove(ent);
44}
45
46module_init(simple_init);
47module_exit(simple_cleanup);
```

If you build and insert the module, you will see a new file /proc/mydev , You can test the read and write operations using cat and echo (only see the kernel log messages)

```
1# echo "test" > /proc/mydev
2bash: echo: write error: Operation not permitted
3
4# cat /proc/mydev
5# dmesg | tail -2
6[  694.640306] write handler
7[  714.661465] read handler
```

Implementing The Read Handler

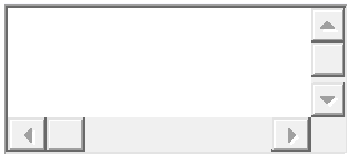
The read handler receives 4 parameters:

- File Object – per process structure with the opened file details (permission , position, etc.)
- User space buffer
- Buffer size
- Requested position (in and out parameter)

To implement the read callback we need to:

- Check the requested position
- Fill the user buffer with a data (max size <= Buffer size) from the requested position
- Return the number of bytes we filled.

For example, the user run the following code:



```
1int fd = open("/proc/mydev", O_RDWR);
2
3len = read(fd,buf,100);
4len = read(fd,buf,50);
```

On the first call to read we get the user buffer, size = 100, position = 0 , we need to fill the buffer with up to 100 bytes from position 0, update the position and return the number of bytes we wrote. If we filled the buffer with 100 bytes and returned 100 the next call to read we get the user buffer, size=50 and position=100

Suppose we have 2 module parameters and we want to return their values on proc read handler we write the following:

```
 1static int irq=20;
 2module_param(irq,int,0660);
 3
 4static int mode=1;
 5module_param(mode,int,0660);
 6
 7
 8static ssize_t myread(struct file *file, char __user *ubuf,size_t count, loff_t *ppos)
 9{
10          char buf[BUFSIZE];
11          int len=0;
```

```
12          printk( KERN_DEBUG "read handler\n");
13          if(*ppos > 0 || count < BUFSIZE)
14                  return 0;
15          len += sprintf(buf,"irq = %d\n",irq);
16          len += sprintf(buf + len,"mode = %d\n",mode);
17
18          if(copy_to_user(ubuf,buf,len))
19                  return -EFAULT;
20          *ppos = len;
21          return len;
22}
```

This is a simple implementation, We check if this is the first time we call read (pos=0) and the user buffer size is bigger than BUFSIZE , otherwise we return 0 (end of file)

Then, we build the returned buffer, copy it to the user , update the position and return the number we wrote

Build and insert the module, you can test it with cat command:

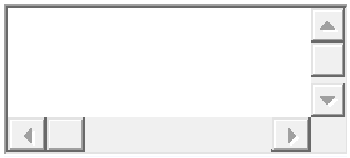
```
1# sudo insmod ./simproc.ko irq=32 mode=4
2# cat /proc/mydev
3irq = 32
4mode = 4
```

Exchanging Data With User-Space

In the kernel code, you can't just use memcpy between an address supplied by user-space and the address of a buffer in kernel-space:

- Correspond to completely different address spaces (thanks to virtual memory).
- The user-space address may be swapped out to disk.
- The user-space address may be invalid (user space process trying to access unauthorized data).

You must use dedicated functions in your read and write file operations code:



```
1include <asm/uaccess.h>
2unsigned long copy_to_user(void __user *to,const void *from, unsigned long n);
3unsigned long copy_from_user(void *to,const void __user *from,unsigned long n);
```

Implementing the Write Handler

The write handler is similar to the read handler. The only difference is that the user buffer type is a const char pointer. We need to copy the data from the user buffer to the requested position and return the number of bytes copied

In this example we want to set both values using a simple command:

```
1# echo "32 6" > /proc/mydev
```

The first value is the irq number and the second is the mode.

The code for the write handler:

```
 1static ssize_t mywrite(struct file *file, const char __user *ubuf,size_t count, loff_t *ppos)
 2{
 3        int num,c,i,m;
 4        char buf[BUFSIZE];
 5        if(*ppos > 0 || count > BUFSIZE)
 6                return -EFAULT;
 7        if(copy_from_user(buf,ubuf,count))
 8                return -EFAULT;
 9        num = sscanf(buf,"%d %d",&i,&m);
10        if(num != 2)
11                return -EFAULT;
12        irq = i;
13        mode = m;
14        c = strlen(buf);
15        *ppos = c;
16        return c;
17}
```

Again, we check if this is the first time we call write (position 0) , then we use copy_from_user to memcpy the data from the user address space to the kernel address space. We extract the values, check for errors , update the position and return the number of bytes we received

The complete module code:

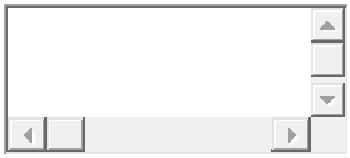
```
 1#include <linux/module.h>
 2#include <linux/moduleparam.h>
 3#include <linux/init.h>
 4#include <linux/kernel.h>
 5#include <linux/proc_fs.h>
 6#include <asm/uaccess.h>
 7#define BUFSIZE  100
 8
 9
10MODULE_LICENSE("Dual BSD/GPL");
11MODULE_AUTHOR("Liran B.H");
12
13static int irq=20;
14module_param(irq,int,0660);
15
16static int mode=1;
17module_param(mode,int,0660);
18
19static struct proc_dir_entry *ent;
20
21static ssize_t mywrite(struct file *file, const char __user *ubuf, size_t count, loff_t *ppos)
22{
23        int num,c,i,m;
24        char buf[BUFSIZE];
25        if(*ppos > 0 || count > BUFSIZE)
26                return -EFAULT;
27        if(copy_from_user(buf, ubuf, count))
28                return -EFAULT;
29        num = sscanf(buf,"%d %d",&i,&m);
30        if(num != 2)
31                return -EFAULT;
32        irq = i;
33        mode = m;
34        c = strlen(buf);
35        *ppos = c;
36        return c;
37}
38
39static ssize_t myread(struct file *file, char __user *ubuf,size_t count, loff_t *ppos)
40{
41        char buf[BUFSIZE];
42        int len=0;
43        if(*ppos > 0 || count < BUFSIZE)
44                return 0;
45        len += sprintf(buf,"irq = %d\n",irq);
46        len += sprintf(buf + len,"mode = %d\n",mode);
47
48        if(copy_to_user(ubuf,buf,len))
```

```
49                      return -EFAULT;
50              *ppos = len;
51              return len;
52}
53
54static struct file_operations myops =
55{
56              .owner = THIS_MODULE,
57              .read = myread,
58              .write = mywrite,
59};
60
61static int simple_init(void)
62{
63              ent=proc_create("mydev",0660,NULL,&myops);
64              printk(KERN_ALERT "hello...\n");
65              return 0;
66}
67
68static void simple_cleanup(void)
69{
70              proc_remove(ent);
71              printk(KERN_WARNING "bye ...\n");
72}
73
74module_init(simple_init);
75module_exit(simple_cleanup);
```

Note : to implement more complex proc entries , use the seq_file wrapper

User Space Application

You can open the file and use read/write functions to test the module. Don't forget to move the position bask to 0 after each operation:



```
 1#include <stdio.h>
 2#include <sys/types.h>
 3#include <sys/stat.h>
 4#include <fcntl.h>
 5#include <unistd.h>
 6
 7void main(void)
 8{
 9              char buf[100];
10              int fd = open("/proc/mydev", O_RDWR);
11              read(fd, buf, 100);
```

```
12          puts(buf);
13
14          lseek(fd, 0 , SEEK_SET);
15          write(fd, "33 4", 5);
16
17          lseek(fd, 0 , SEEK_SET);
18          read(fd, buf, 100);
19          puts(buf);
20}
```