**On The first post we built a**[**Simple Kernel Module**](https://devarea.com/linux-kernel-development-and-writing-a-simple-kernel-module/)**with init and exit functions and covered the basic concepts in kernel programming**

Next, We added a [Kernel Module Parameters](https://devarea.com/linux-kernel-development-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  2 | # cat /proc/cpuinfo  # 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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | #include <linux/module.h>  #include <linux/moduleparam.h>  #include <linux/init.h>  #include <linux/kernel.h>  #include <linux/proc\_fs.h>  #include <asm/uaccess.h>  #define BUFSIZE  100      MODULE\_LICENSE("Dual BSD/GPL");  MODULE\_AUTHOR("Liran B.H");      static **struct** proc\_dir\_entry \*ent;    static ssize\_t mywrite(**struct** file \*file, **const** **char** \_\_user \*ubuf,size\_t count, loff\_t \*ppos)  {  printk( KERN\_DEBUG "write handler\n");  **return** -1;  }    static ssize\_t myread(**struct** file \*file, **char** \_\_user \*ubuf,size\_t count, loff\_t \*ppos)  {  printk( KERN\_DEBUG "read handler\n");  **return** 0;  }    static **struct** file\_operations myops =  {  .owner = THIS\_MODULE,  .read = myread,  .write = mywrite,  };    static **int** simple\_init(**void**)  {  ent=proc\_create("mydev",0660,**NULL**,&myops);  **return** 0;  }    static **void** simple\_cleanup(**void**)  {  proc\_remove(ent);  }    module\_init(simple\_init);  module\_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  2  3  4  5  6  7 | # echo "test" > /proc/mydev  bash: echo: write error: Operation **not** permitted    # cat /proc/mydev  # dmesg | tail -2  [  694.640306] write handler  [  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:



|  |  |
| --- | --- |
| 1  2  3  4 | **int** fd = open("/proc/mydev", O\_RDWR);    len = read(fd,buf,100);  len = 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:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | static **int** irq=20;  module\_param(irq,**int**,0660);    static **int** mode=1;  module\_param(mode,**int**,0660);      static ssize\_t myread(**struct** file \*file, **char** \_\_user \*ubuf,size\_t count, loff\_t \*ppos)  {  **char** buf[BUFSIZE];  **int** len=0;  printk( KERN\_DEBUG "read handler\n");  **if**(\*ppos > 0 || count < BUFSIZE)  **return** 0;  len += **sprintf**(buf,"irq = %d\n",irq);  len += **sprintf**(buf + len,"mode = %d\n",mode);    **if**(copy\_to\_user(ubuf,buf,len))  **return** -EFAULT;  \*ppos = len;  **return** len;  } |

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



|  |  |
| --- | --- |
| 1  2  3 | include <**asm**/uaccess.h>  **unsigned** **long** copy\_to\_user(**void** \_\_user \***to**,**const** **void** \*from, **unsigned** **long** n);  **unsigned** **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:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | static ssize\_t mywrite(**struct** file \*file, **const** **char** \_\_user \*ubuf,size\_t count, loff\_t \*ppos)  {  **int** num,c,i,m;  **char** buf[BUFSIZE];  **if**(\*ppos > 0 || count > BUFSIZE)  **return** -EFAULT;  **if**(copy\_from\_user(buf,ubuf,count))  **return** -EFAULT;  num = sscanf(buf,"%d %d",&i,&m);  **if**(num != 2)  **return** -EFAULT;  irq = i;  mode = m;  c = strlen(buf);  \*ppos = c;  **return** c;  } |

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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75 | #include <linux/module.h>  #include <linux/moduleparam.h>  #include <linux/init.h>  #include <linux/kernel.h>  #include <linux/proc\_fs.h>  #include <asm/uaccess.h>  #define BUFSIZE  100      MODULE\_LICENSE("Dual BSD/GPL");  MODULE\_AUTHOR("Liran B.H");    static **int** irq=20;  module\_param(irq,**int**,0660);    static **int** mode=1;  module\_param(mode,**int**,0660);    static **struct** proc\_dir\_entry \*ent;    static ssize\_t mywrite(**struct** file \*file, **const** **char** \_\_user \*ubuf, size\_t count, loff\_t \*ppos)  {  **int** num,c,i,m;  **char** buf[BUFSIZE];  **if**(\*ppos > 0 || count > BUFSIZE)  **return** -EFAULT;  **if**(copy\_from\_user(buf, ubuf, count))  **return** -EFAULT;  num = sscanf(buf,"%d %d",&i,&m);  **if**(num != 2)  **return** -EFAULT;  irq = i;  mode = m;  c = strlen(buf);  \*ppos = c;  **return** c;  }    static ssize\_t myread(**struct** file \*file, **char** \_\_user \*ubuf,size\_t count, loff\_t \*ppos)  {  **char** buf[BUFSIZE];  **int** len=0;  **if**(\*ppos > 0 || count < BUFSIZE)  **return** 0;  len += **sprintf**(buf,"irq = %d\n",irq);  len += **sprintf**(buf + len,"mode = %d\n",mode);    **if**(copy\_to\_user(ubuf,buf,len))  **return** -EFAULT;  \*ppos = len;  **return** len;  }    static **struct** file\_operations myops =  {  .owner = THIS\_MODULE,  .read = myread,  .write = mywrite,  };    static **int** simple\_init(**void**)  {  ent=proc\_create("mydev",0660,**NULL**,&myops);  printk(KERN\_ALERT "hello...\n");  **return** 0;  }    static **void** simple\_cleanup(**void**)  {  proc\_remove(ent);  printk(KERN\_WARNING "bye ...\n");  }    module\_init(simple\_init);  module\_exit(simple\_cleanup); |

Note : to implement more complex proc entries , use the [seq\_file](https://elixir.bootlin.com/linux/latest/source/include/linux/seq_file.h) 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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #include <stdio.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <unistd.h>    **void** main(**void**)  {  **char** buf[100];  **int** fd = open("/proc/mydev", O\_RDWR);  read(fd, buf, 100);  puts(buf);    lseek(fd, 0 , SEEK\_SET);  write(fd, "33 4", 5);    lseek(fd, 0 , SEEK\_SET);  read(fd, buf, 100);  puts(buf);  } |