EmbeTronicX
Embedded Tutorials Zone

Sidebar



Device Drivers


# Linux Device Driver Tutorial Part 9 – Procfs in Linux

3

X

carries on the discussion on character drivers and their implementation. This is Part 9 of the Linux device driver tutorial. Now we will discuss ProcFS in Linux.



**Post Contents** [hide]


1 Introduction
2 Procfs in Linux
    2.1 Introduction
3 Creating Procfs Entry
4 Procfs File System
    4.1 Open and Release Function
    4.2 Write Function
    4.3 Read Function
5 Remove Proc Entry
6 Complete Driver Code
7 MakeFile
8 Building and Testing Driver
        8.0.1 Share this:
        8.0.2 Like this:
        8.0.3 Related


3

X

userspace. Kernel space is strictly reserved for running the kernel, kernel extensions, and most device drivers. In contrast, user space is the memory area where all user-mode applications work, and this memory can be swapped out when necessary.
There are many ways to Communicate between the Userspace and Kernel Space, they are:

- IOCTL
- Procfs
- Sysfs
- Configfs
- Debugfs
- Sysctl
- UDP Sockets
- Netlink Sockets

In this tutorial, we will see Procfs.

## Procfs in Linux

## Introduction

Many or most Linux users have at least heard of proc. Some of you may wonder why this folder is so important.

3

X

On the root, there is a folder titled “proc”. This folder is not really on /dev/sda1 or where ever you think the folder resides. This folder is a mount point for the procfs (Process Filesystem) which is a filesystem in memory. Many processes store information about themselves on this virtual filesystem. ProcFS also stores other system information.

3

X

from the kernel.

The entry "***meminfo***" gives the details of the memory being used in the system.
To read the data in this entry just run

```
cat /proc/meminfo
```

Similarly the "***modules***" entry gives details of all the modules that are currently a part of the kernel.

```
cat /proc/modules
```

It gives similar information as `lsmod`. Like this more, proc entries are there.

- `/proc/devices` — registered character and block major numbers
- `/proc/iomem` — on-system physical RAM and bus device addresses
- `/proc/ioports` — on-system I/O port addresses (especially for x86 systems)
- `/proc/interrupts` — registered interrupt request numbers
- `/proc/softirqs` — registered soft IRQs
- `/proc/swaps` — currently active swaps
- `/proc/kallsyms` — running kernel symbols, including from loaded modules
- `/proc/partitions` — currently connected block devices and their partitions
- `/proc/filesystems` — currently active filesystem drivers
- `/proc/cpuinfo` — information about the CPU(s) on the system

Most proc files are read-only and only expose kernel information to user space programs.

proc files can also be used to control and modify kernel behavior on the fly. The proc files need to be writable in this case

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

The proc file system is also very useful when we want to debug a kernel module. While debugging we might want to know the values of various variables in the module or maybe the data that the module is handling. In such situations, we can create a proc entry for our selves and dump whatever data we want to look into in the entry.

We will be using the same example character driver that we created in the previous post to create the proc entry.

The proc entry can also be used to pass data to the kernel by writing into the kernel, so there can be two kinds of proc entries.

1. An entry that only reads data from the kernel space.
2. An entry that reads as well as writes data into and from kernel space.

## Creating Procfs Entry

The creation of proc entries has undergone a considerable change in kernel version 3.10 and above. In this post, we will see one of the methods we can use in Linux kernel version 3.10 and above let us see how we can create proc entries in version 3.10 and above.

```
static inline struct proc_dir_entry *proc_create(const char *name, umode_t mode,
                                     struct proc_dir_entry *parent,
                                     const struct file_operations *proc_fops)
```

The function is defined in proc_fs.h.

Where,

<**name**>: The name of the proc entry
<**mode**>: The access mode for proc entry
<**parent**>: The name of the parent directory under /proc. If NULL is passed as a parent, the /proc directory will be set as a parent.
<**proc_fops**>: The structure in which the file operations for the proc

the above function will be defined as below,

```
proc_create("etx_proc",0666,NULL,&proc_fops);
```

This proc entry should be created in Driver init function.

If you are using the kernel version below 3.10, please use the below functions to create proc entry.

```
create_proc_read_entry()
create_proc_entry()
```

Both of these functions are defined in the file ***linux/proc_fs.h***.

The create_proc_entry is a generic function that allows creating both the read as well as the write entries.
**create_proc_read_entry** is a function specific to create only read entries.

It is possible that most of the proc entries are created to read data from the kernel space that is why the kernel developers have provided a direct function to create a read proc entry.

## Procfs File System

Now we need to create file_operations structure proc_fops in which we can map the read and write functions for the proc entry.

```
static struct file_operations proc_fops = {
    .open = open_proc,
    .read = read_proc,
    .write = write_proc,
    .release = release_proc
};
```

This is like a device driver file system. We need to register our proc entry filesystem. If you are using the kernel version below 3.10, this will not be

## Open and Release Function

These functions are optional.

```
static int open_proc(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "proc file opend.....\t");
    return 0;
}

static int release_proc(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "proc file released.....\n");
    return 0;
}
```

## Write Function

The write function will receive data from the user space using the function copy_from_user into an array "etx_array".

Thus the write function will look as below.

```
static ssize_t write_proc(struct file *filp, const char *buff, size_t len, loff_t * o
{
    printk(KERN_INFO "proc file write.....\t");
    copy_from_user(etx_array,buff,len);
    return len;
}
```

## Read Function

Once data is written to the proc entry we can read from the proc entry using a read function, i.e transfer data to the user space using the function copy_to_user function.

The read function can be as below.

```
static ssize_t read_proc(struct file *filp, char __user *buffer, size_t length,loff_
```

```
    }
    copy_to_user(buffer,etx_array,20);

    return length;;
}
```

## Remove Proc Entry

Proc entry should be removed in the Driver exit function using the below function.

```
void remove_proc_entry(const char *name, struct proc_dir_entry *parent);
```

Example:

```
remove_proc_entry("etx_proc",NULL);
```

## Complete Driver Code

This code will work for the kernel above the 3.10 version. I just took the previous tutorial driver code and update it with procfs.

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
```

X

```
#define WR_VALUE _IOW('a','a',int32_t*)
#define RD_VALUE _IOR('a','b',int32_t*)

int32_t value = 0;
char etx_array[20]="try_proc_array\n";
static int len = 1;


dev_t dev = 0;
static struct class *dev_class;
static struct cdev etx_cdev;

static int __init etx_driver_init(void);
static void __exit etx_driver_exit(void);
/*************** Driver Functions **********************/
static int etx_open(struct inode *inode, struct file *file);
static int etx_release(struct inode *inode, struct file *file);
static ssize_t etx_read(struct file *filp, char __user *buf, size_t len,loff_t * of
static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * o
static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg);

/***************** Procfs Functions *******************/
static int open_proc(struct inode *inode, struct file *file);
static int release_proc(struct inode *inode, struct file *file);
static ssize_t read_proc(struct file *filp, char __user *buffer, size_t length,loff_
static ssize_t write_proc(struct file *filp, const char *buff, size_t len, loff_t *

static struct file_operations fops =
{
        .owner          = THIS_MODULE,
        .read           = etx_read,
        .write          = etx_write,
        .open           = etx_open,
        .unlocked_ioctl = etx_ioctl,
        .release        = etx_release,
};

static struct file_operations proc_fops = {
        .open = open_proc,
        .read = read_proc,
        .write = write_proc,
        .release = release_proc
};

static int open_proc(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "proc file opend.....\t");
    return 0;
}

static int release_proc(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "proc file released.....\n");
    return 0;
}
```

```
        return 0;
    }
    copy_to_user(buffer,etx_array,20);

    return length;;
}

static ssize_t write_proc(struct file *filp, const char *buff, size_t len, loff_t *
{
    printk(KERN_INFO "proc file wrote.....\n");
    copy_from_user(etx_array,buff,len);
    return len;
}

static int etx_open(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "Device File Opened...!!!\n");
        return 0;
}

static int etx_release(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "Device File Closed...!!!\n");
        return 0;
}

static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *of
{
        printk(KERN_INFO "Readfunction\n");
        return 0;
}
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, lof
{
        printk(KERN_INFO "Write Function\n");
        return 0;
}

static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
         switch(cmd) {
                case WR_VALUE:
                        copy_from_user(&value ,(int32_t*) arg, sizeof(value));
                        printk(KERN_INFO "Value = %d\n", value);
                        break;
                case RD_VALUE:
                        copy_to_user((int32_t*) arg, &value, sizeof(value));
                        break;
        }
        return 0;
}


static int __init etx_driver_init(void)
{
        /*Allocating Major number*/
        if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
```

X

```

        /*Adding character device to the system*/
        if((cdev_add(&etx_cdev,dev,1)) < 0){
            printk(KERN_INFO "Cannot add the device to the system\n");
            goto r_class;
        }

        /*Creating struct class*/
        if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
            printk(KERN_INFO "Cannot create the struct class\n");
            goto r_class;
        }

        /*Creating device*/
        if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
            printk(KERN_INFO "Cannot create the Device 1\n");
            goto r_device;
        }

        /*Creating Proc entry*/
        proc_create("etx_proc",0666,NULL,&proc_fops);

        printk(KERN_INFO "Device Driver Insert...Done!!!\n");
    return 0;

r_device:
        class_destroy(dev_class);
r_class:
        unregister_chrdev_region(dev,1);
        return -1;
}

void __exit etx_driver_exit(void)
{
        remove_proc_entry("etx_proc",NULL);
        device_destroy(dev_class,dev);
        class_destroy(dev_class);
        cdev_del(&etx_cdev);
        unregister_chrdev_region(dev, 1);
    printk(KERN_INFO "Device Driver Remove...Done!!!\n");
}

module_init(etx_driver_init);
module_exit(etx_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com or admin@embetronicx.com>");
MODULE_DESCRIPTION("A simple device driver");
MODULE_VERSION("1.6");
```

## MakeFile

```
obj-m += driver.o
```

```
clean:
    make -C $(KDIR)  M=$(shell pwd) clean
```

# Building and Testing Driver

- Build the driver by using Makefile (***sudo make***)
- Load the driver using **`sudo insmod driver.ko`**
- Check our procfs entry using ls in procfs directory

```
linux@embetronicx-VirtualBox:ls /proc/

filesystems      iomem      kallsyms      modules     partitions
```

- Now our procfs entry is there under /proc directory.
- Now you can read procfs variable using **`cat`**.

```
linux@embetronicx-VirtualBox:  cat /proc/etx_proc

try_proc_array
```

- We initialized the etx_array with “try_proc_array”. That’s why we got “try_proc_array”.
- Now do proc write using **`echo`** command and check using **`cat`**.

```
linux@embetronicx-VirtualBox: echo "device driver proc" > /proc/etx_proc

linux@embetronicx-VirtualBox:  cat /proc/etx_proc

device driver proc
```

- We got the same string that was passed to the driver using procfs.

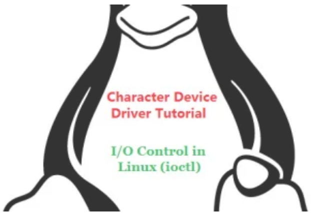
This is a simple example using procfs in the device drivers. This is just basic. I hope this might helped you. In our next tutorial, we will discuss waitqueue in the Linux device drivers.

**Like this:**

Loading...

**Related**

[Linux Device Driver Tutorial Part 11 – Sysfs in Linux Kernel]
In "Device Drivers"




[Linux Device Driver Tutorial Part 8 – I/O Control in Linux IOCTL()]
In "Device Drivers"

[Linux Device Driver Part 1 : Introduction]
In "Device Drivers"

3

X

Subscribe

Connect with | Login

*Join the discussion*

B I U S {} [+]

This site uses Akismet to reduce spam. [Learn how your comment data is processed].

**3 COMMENTS**

Oldest

Anonymous

January 21, 2018 5:37 PM

Nice tutorial. Thanks for taking the time to share in simple format.

3

Loading...

X

nguyen tiendat

January 25, 2018 2:16 AM

thanks. Your tutorial make me understand how to debug device driver by using procfs

Loading...

0 Reply

Sana Srikar

August 13, 2018 9:00 AM

also guys do let us know how to check updates in the API.
Say in future if this is going to change then how to check .. could you please also explain that.

Loading...

1 Reply

3

X

3

X