Linux Architecture

Linux is primarily divided into **User Space** & **Kernel Space**. These two components interact through a System Call Interface – which are predefined and matured interface to Linux Kernel for User space applications. The below image will give you the basic understanding.

Kernel Space

Kernel space is where the kernel (i.e., the core of the operating system) executes (i.e., runs) and provides its services.

User Space

User Space is where the user applications are executed.

Linux Kernel Modules

Kernel modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. Custom codes can be added to Linux kernels via two methods.

- The basic way is to add the code to the kernel source tree and recompile the kernel.
- A more efficient way is to do this is by adding code to the kernel while it is running. This process is called loading the module, where module refers to the code that we want to add to the kernel.

Since we are loading these codes at runtime and they are not part of the official Linux kernel, these are called loadable kernel module(LKM), which is different from the “base kernel”. Base kernel is located in /boot directory and is always loaded when we boot our machine whereas LKMs are loaded after the base kernel is already loaded. Nonetheless, these LKM are very much part of our kernel and they communicate with base kernel to complete their functions.

LKMs can perform a variety of task, but basically they come under three main categories,

- Device drivers
- Filesystem drivers
- System calls

Device drivers

A device driver is designed for a specific piece of hardware. The kernel uses it to communicate with that piece of hardware without having to know any details of how the hardware works.

Filesystem drivers

A filesystem driver interprets the contents of a filesystem (which is typically the contents of a disk drive) as files and directories and such. There are lots of different ways of storing files and directories and such on disk drives, on network servers, and in other ways. For each way, you need a filesystem driver. For example, there's a filesystem driver for the ext2 filesystem type used almost universally on Linux disk drives. There is one for the MS-DOS filesystem too, and one for NFS.

System calls

User space programs use system calls to get services from the kernel. For example, there are system calls to read a file, to create a new process, and to shut down the system. Most system calls are integral to the system and very standard, so are always built into the base kernel (no LKM option).

But you can invent a system call of your own and install it as an LKM. Or you can decide you don't like the way Linux does something and override an existing system call with an LKM of your own.

Advantages of LKM

- One major advantage they have is that we don't need to keep rebuilding the kernel every time we add a new device or if we upgrade an old device. This saves time and also helps in keeping our base kernel error free.
- LKMs are very flexible, in the sense that they can be loaded and unloaded with a single line of command. This helps in saving memory as we load the LKM only when we need them.

Differences Between Kernel Modules and User Programs

- **Kernel modules have separate address space.** A module runs in kernel space. An application runs in user space. System software is protected from user programs. Kernel space and user space have their own memory address spaces.
- **Kernel modules have higher execution privilege.** Code that runs in kernel space has greater privilege than code that runs in user space.
- **Kernel modules do not execute sequentially.** A user program typically executes sequentially and performs a single task from beginning to end. A kernel module does not execute sequentially. A kernel module registers itself in order to serve future requests.
- **Kernel modules use different header files.** Kernel modules require a different set of header files than user programs require.

Difference Between Kernel Drivers and Kernel Modules

- A kernel module is a bit of compiled code that can be inserted into the kernel at run-time, such as withinsmod or modprobe.
- A driver is a bit of code that runs in the kernel to talk to some hardware device. It “drives” the hardware. Most every bit of hardware in your computer has an associated driver.

Device Driver

A device driver is a particular form of software application that is designed to enable interaction with hardware devices. Without the required device driver, the corresponding hardware device fails to work.
A device driver usually communicates with the hardware by means of the communications subsystem or computer bus to which the hardware is connected. Device drivers are operating system-specific and hardware-dependent. A device driver acts as a translator between the hardware device and the programs or operating systems that use it.

Types

In the traditional classification, there are three kinds of device:

- Character device
- Block device
- Network device

In Linux everything is a file. I mean Linux treat everything as a File even hardware.

Character Device

A char file is a hardware file which reads/write data in character by character fashion. Some classic examples are keyboard, mouse, serial printer. If a user use a char file for writing data no other user can use same char file to write data which blocks access to other user. Character files uses synchronize Technic to write data. Of you observe char files are used for communication purpose and they can not be mounted.

Block Device

A block file is a hardware file which read/write data in blocks instead of character by character. This type of files are very much useful when we want to write/read data in bulk fashion. All our disks such are HDD, USB and CDROMs are block devices. This is the reason when we are formatting we consider block size. The write of data is done in asynchronous fashion and it is CPU intensive activity. These devices files are used to store data on real hardware and can be mounted so that we can access the data we written.

Network Device

A network device is, so far as Linux’s network subsystem is concerned, an entity that sends and receives packets of data. This is normally a physical device such as an ethernet card. Some network devices though are software only such as the loopback device which is used for sending data to yourself.

First Device Driver

Module Information

- License
- Author
- Module Description
- Module Version

These all informations are present in the linux/module.h as a macros.

License

GPL, or the GNU General Public License, is an open source license meant for software. If your software is licensed under the terms of the GPL, it is free. However, “free” here does not essentially mean freeware—it can also be a paid software. Instead, “free” as per the GPL means freedom. As proponents of GPL proudly proclaim, free as in freedom, not free beer.

The following license idents are currently accepted as indicating free software modules.

"GPL" [GNU Public License v2 or later]

"GPL v2" [GNU Public License v2]

"GPL and additional rights" [GNU Public License v2 rights and more]

"Dual BSD/GPL" [GNU Public License v2 or BSD license choice]

"Dual MIT/GPL" [GNU Public License v2 or MIT license choice]

"Dual MPL/GPL" [GNU Public License v2 or Mozilla license choice]

The following other idents are available

"Proprietary" [Non free products]

There are dual licensed components, but when running with Linux it is the GPL that is relevant so this is a non issue. Similarly LGPL linked with GPL is a GPL combined work.

This exists for several reasons,

1. modinfo can show license info for users wanting to vet their setup is free
2. The community can ignore bug reports including proprietary modules
3. Vendors can do likewise based on their own policies

We can give the License for our driver (module) like below. For this you need to include the Linux/module.h header file.

```
MODULE_LICENSE("GPL"
);
MODULE_LICENSE("GPL
v2");
MODULE_LICENSE("Dual
BSD/GPL");
```

Note : It is not strictly necessary, but your module really should specify which license applies to its code.

Author

Using this Macro we can mention that who is wrote this driver or module. So modinfo can show author name for users wanting to know. We can give the Author name for our driver (module) like below. For this you need to include the Linux/module.h header file.

```
MODULE_AUTHOR("Author");
```

```
MODULE_AUTHOR("Author
");
```

Note: Use “Name <email>” or just “Name”, for multiple authors use multiple MODULE_AUTHOR() statements/lines.

Module Description

Using this Macro we can give the description of the module or driver. So modinfo can show module description for users wanting to know. We can give the description for our driver (module) like below. For this you need to include the linux/module.h header file.

```
MODULE_DESCRIPTION("A sample
driver");
```

Module Version

Using this Macro we can give the version of the module or driver. So modinfo can show module version for users wanting to know.

Version of form [<epoch>:]<version>[-<extra-version>].

<epoch>: A (small) unsigned integer which allows you to start versions anew. If not mentioned, it’s zero. eg. “2:1.0” is after “1:2.0”.

<version>: The <version> may contain only alphanumerics and the character `.’. Ordered by numeric sort for numeric parts, ascii sort for ascii parts (as per RPM or DEB algorithm).

<extraversion>: Like <version>, but inserted for local customizations, eg “rh3” or “rusty1”.

Example

```
MODULE_VERSION("2:1.0");
```

```
1 MODULE_VERSION("2:1.0
");
```

Simple Kernel Module Programming

So as of now we know the very basic things that needed for writing driver. Now we will move into programming. In every programming language, how we will start to write the code? Any ideas? Well, in all programming there would be a starting point and ending point. If you take C Language, starting point would be the main function, Isn’t it? It will start from the starting of the main function and run through the functions which is calling from main function. Finally it exits at the main function closing point. But Here two separate functions used for that starting and ending.

1. Init function
2. Exit function

Kernel modules require a different set of header files than user programs require.And keep in mind, Module code should not invoke user space Libraries or API’s or System calls.

Init function

This is the function which will executes first when the driver is loaded into the kernel. For example when we load the driver using insmod, this function will execute. Please see below to know the syntax of this function.

```
static int __init hello_world_init(void) /* Constructor */ { return 0; }
module_init(hello_world_init);
```

```
1 static int __init hello_world_init(void) /*
2 Constructor */
3 {
4 return 0;
5 }
  module_init(hello_world_init);
```

This function should register itself by using module_init() macro.

Exit function

This is the function which will executes last when the driver is unloaded from the kernel. For example when we unload the driver using rmmod, this function will execute. Please see below to know the syntax of this function.

```
void __exit hello_world_exit(void) { } module_exit(hello_world_exit);
```

```
1 void __exit
2 hello_world_exit(void)
3 {
4
5 }
  module_exit(hello_world_exit);
```

This function should register itself by using module_exit() macro.

Printk()

In C programming how we will print the values or whatever? Correct. Using printf() function. printf() is a user space function. So we cant use this here. So they created one another function for kernel which is printk().

One of the differences is that printk lets you classify messages according to their severity by associating different loglevels, or priorities, with the messages. You usually indicate the loglevel with a macro. I will explain about the macros now. There are several macros used for printk.

KERN_EMERG:

Used for emergency messages, usually those that precede a crash.

KERN_ALERT:

Situation requiring immediate action.

KERN_CRIT:

Critical conditions, often related to serious hardware or software failures.

KERN_ERR:

Used to report error conditions; device drivers often use KERN_ERR to report hardware difficulties.

KERN_WARNING:

Warnings about problematic situations that do not, in themselves, create serious problems with the system.

KERN_NOTICE:

Situations that are normal, but still worthy of note. A number of security-related conditions are reported at this level.

KERN_INFO:

Informational messages. Many drivers print information about the hardware they find at startup time at this level.

KERN_DEBUG:

Used for debugging messages.

Example

```
printk(KERN_INFO "Welcome To EmbeTronicX");
```

```
1 printk(KERN_INFO "Welcome To
  EmbeTronicX");
```

Difference between printf and printk

- Printk() is a kernel level function, which has the ability to print out to different loglevels as defined in . We can see the prints using dmesg command.
- Printf() will always print to a file descriptor – STD_OUT. We can see the prints in STD_OUT console.

Simple Driver

This is the complete code for our simple device driver (hello_world_module.c). You can download the Project By clicking below link.

Compiling our driver

Once we have the C code, it is time to compile it and create the module file hello_world_module.ko. creating a Makefile for your module is straightforward.

With the C code (hello_world_module.c) and Makefile ready, all we need to do is invoke make to build our first driver (hello_world_module .ko).

In Terminal you need to enter sudo make like below image.

Now we got hello_world_module .ko. This is the kernel object which is loading into kernel.

```
[redacted]:/home/driver/simple_driver$ ls -l
total 8
-rwxrwxrwx 1 root root 649 Aug 10 18:24 hello_world_module.c
-rwxrwxrwx 1 root root 169 Aug 10 18:24 Makefile
[redacted]:/home/driver/simple_driver$ sudo make
make -C /lib/modules/4.4.0-59-generic/build  M=/home/driver/simple_driver modules
make[1]: Entering directory `/usr/src/linux-headers-4.4.0-59-generic'
  CC [M]  /home/driver/simple_driver/hello_world_module.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/driver/simple_driver/hello_world_module.mod.o
  LD [M]  /home/driver/simple_driver/hello_world_module.ko
make[1]: Leaving directory `/usr/src/linux-headers-4.4.0-59-generic'
[redacted]:/home/driver/simple_driver$ ls -l
total 32
www.embetronicx.com
-rwxrwxrwx 1 root root  649 Aug 10 18:24 hello_world_module.c
-rw-r--r-- 1 root root 4444 Aug 10 18:32 hello_world_module.ko
-rw-r--r-- 1 root root  713 Aug 10 18:32 hello_world_module.mod.c
-rw-r--r-- 1 root root 2816 Aug 10 18:32 hello_world_module.mod.o
-rw-r--r-- 1 root root 2496 Aug 10 18:32 hello_world_module.o
-rwxrwxrwx 1 root root  169 Aug 10 18:24 Makefile
-rw-r--r-- 1 root root   56 Aug 10 18:32 modules.order
-rw-r--r-- 1 root root    0 Aug 10 18:32 Module.symvers
[redacted]:/home/driver/simple_driver$
```

Loading and Unloading the Device driver

A Kernel Module is a small file that may be loaded into the running Kernel, and unloaded.

Loading

To load a Kernel Module, use the insmod command with root privileges.

For example our module file name is hello_world_module.ko

sudo insmod hello_world_module.ko

```
[redacted]x:/home/driver/simple_driver$ sudo insmod hello_world_module.ko
[redacted]x:/home/driver/simple_driver$ lsmod
Module                  Size  Used by
hello_world_module     16384  0
nls_utf8               16384  1
www.embetronicx.com
isofs                  40960  1
pci_stub               16384  1
vboxpci                24576  0
```

lsmod used to see the modules were inserted. In below image, i’ve shown the prints in init function. Use dmesg to see the kernel prints.

```
[ 1466.637639] Welcome to EmbeTronicX
[ 1466.637646] This is the Simple Module
[ 1466.637648] Kernel Module Inserted Successfully...
```

So when i load the module, it executes the init function.

Listing the Modules

In order to see the list of currently loaded Modules, use the lsmod command. In above image you can see that i have used lsmod command.

Unloading

To un-load a Kernel Module, use the rmmod command with root privileges.

In our case,

sudo rmmod hello_world_module.ko or sudo rmmod hello_world_module

```
x:/home/driver/simple_driver$ sudo rmmod hello_world_module
x:/home/driver/simple_driver$ dmesg
[ 1466.637639] Welcome to EmbeTronicX
[ 1466.637646] This is the Simple Module
[ 1466.637648] Kernel Module Inserted Successfully...
[ 1701.882646] Kernel Module Removed Successfully...
:/home/driver/simple_driver$
```

So when i unload the module, it executes the exit function.

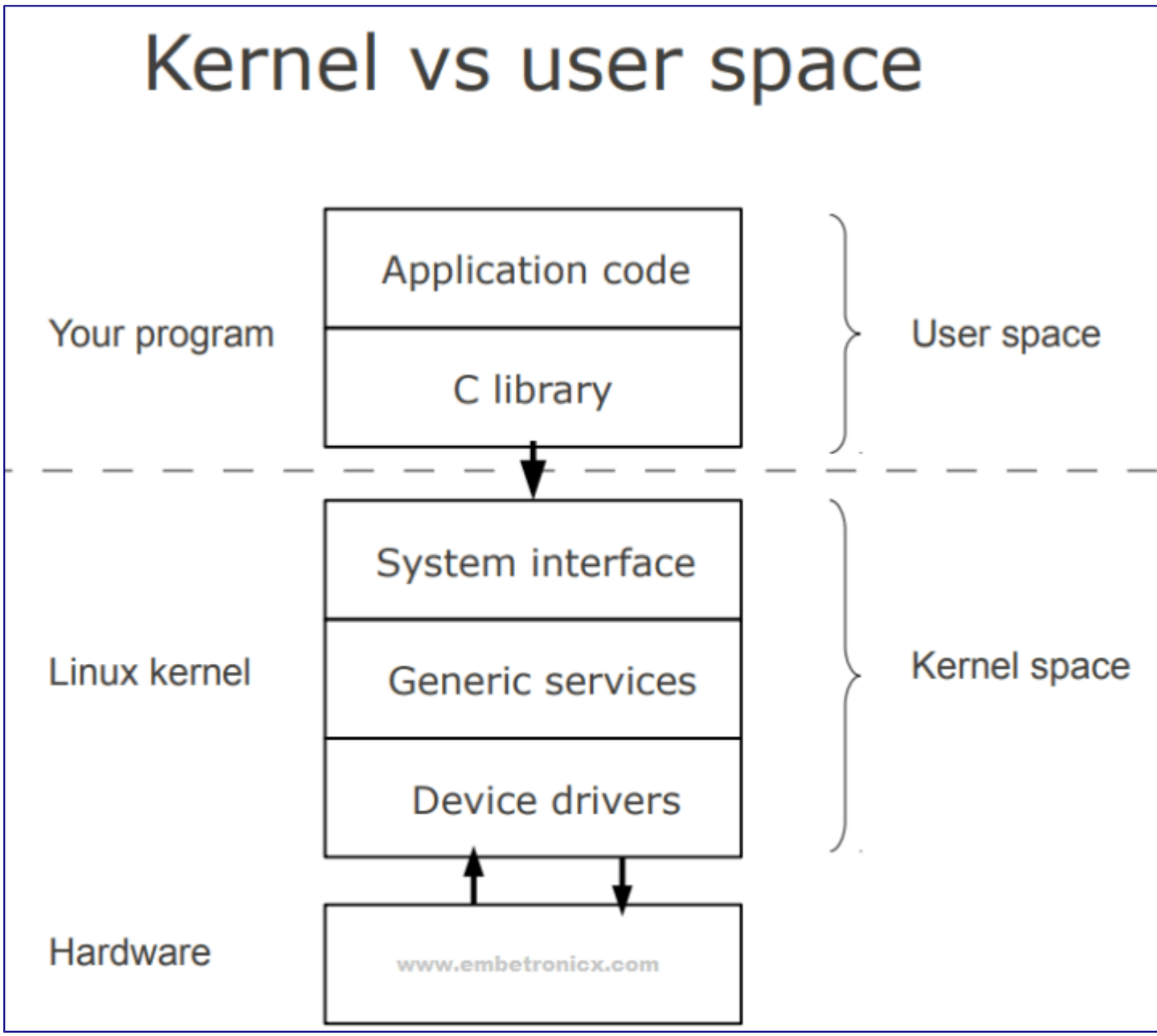
Getting Module Details

In order to get information about a Module (author, supported options), we may use the modinfo command.

For example

modinfo hello_world_module.ko

```
x:/home/driver/simple_driver$ modinfo hello_world_module.ko
filename:       /home/driver/simple_driver/hello_world_module.ko
version:        2:1.0
description:    A simple hello world driver
author:         EmbeTronicX <embetronicx@gmail.com or admin@embetronicx.com>
license:        GPL
srcversion:     A0D3444D86F187EAB75B977
depends:
vermagic:       4.4.0-59-generic SMP mod_unload modversions
x:/home/driver/simple_driver$
```



Passing Arguments to Device Driver

We can pass the arguments to any other functions in same program. But Is it possible to pass any arguments to any program? I think Probably yes. Right? Well, we can. In C Programming we can pass the arguments to the program. For that we need to add argc and argv in main function definition. I hope everyone knows that. Now come to our topic. Is it possible to pass the argument to the Device Driver? Fine. In this tutorial we are going to see that topic.

Module Parameters Macros

- module_param()

- module_param_array()
- module_param_cb()

Before discuss these macros we have to know about permissions of the variable.

There are several types of permissions:

- S_IWUSR
- S_IRUSR
- S_IXUSR
- S_IRGRP
- S_IWGRP
- S_IXGRP

In this S_I is common header.
R = read ,W =write ,X= Execute.
USR =user ,GRP =Group
Using OR '|' (or operation) we can set multiple permissions at a time.

module_param()

This macro used to initialize the arguments. module_param takes three parameters: the name of the variable, its type, and a permissions mask to be used for an accompanying sysfs entry. The macro should be placed outside of any function and is typically found near the head of the source file. module_param() macro, defined in linux/moduleparam.h.

module_param(name, type, perm);

module_param() macro creates the sub-directory under /sys/module. For example

module_param(valueETX, int, S_IWUSR|S_IRUSR);

```
1 module_param(valueETX, int, S_IWUSR|
  S_IRUSR);
```

This will create the sysfs entry. (/sys/module/hello_world_module/parameters/valueETX)

Numerous types are supported for module parameters:

- bool
- invbool

A boolean (true or false) value (the associated variable should be of type int). The invbool type inverts the value, so that true values become false and vice versa.

- charp

A char pointer value. Memory is allocated for user-provided strings, and the pointer is set accordingly.

- int
- long
- short
- uint
- ulong
- ushort

Basic integer values of various lengths. The versions starting with u are for unsigned values.

module_param_array();

This macro is used to send the array as a argument. Array parameters, where the values are supplied as a comma-separated list, are also supported by the module loader. To declare an array parameter, use:

module_param_array(name,type,num,perm);

Where,

name is the name of your array (and of the parameter),

type is the type of the array elements,

num is an integer variable (optional) otherwise NULL, and

perm is the usual permissions value.

module_param_cb()

This macro used to register the callback whenever the argument (parameter) got changed. I think you don’t understand. Let me explain properly.

For Example,

I have created one parameter by using module_param().

```
module_param(valueETX, int, S_IWUSR|S_IRUSR);
```

```
1 module_param(valueETX, int, S_IWUSR|
  S_IRUSR);
```

This will create the sysfs entry. (/sys/module/hello_world_module/parameters/valueETX)

You can change the value of valueETX from command line by

echo 1 > /sys/module/hello_world_module/parameters/valueETX

This will update the valueETX variable. But there is no way to notify your module that “valueETX” has changed.

By using this module_param_cb() macro, we can get notification.

If you want to get notification whenever value got change. we need to register our handler function to its file operation structure.

struct kernel_param_ops { int (*set)(const char *val, const struct kernel_param *kp); int (*get)(char *buffer, const struct kernel_param *kp); void (*free)(void *arg); };

```
struct kernel_param_ops
{
int (*set)(const char
*val, const struct
kernel_param *kp);
int (*get)(char *buffer,
const struct
kernel_param *kp);
void (*free)(void *arg);
};
```

For further explanation, please refer below program.

When we will need this notification?

I will tell you the practical scenario. Whenever value is set to 1, you have to write a something in to a hardware register. How can you do this if the change of value variable is not notified to you? Got it? I think you have understood. If you didn't understood, just see the explanation posted below.

Programming

In this example, i explained all (module_param, module_param_array, module_param_cb).

For module_param(), i have created two variables. One is integer (valueETX) and another one is string (nameETX).

For module_param_array(), i have created one integer array variable ().

For module_param_cb(), i have created one integer variable (cb_valueETX).

You can change the all variable using their sysfs entry which is under /sys/module/hello_world_module/parameters/

But you want get any notification when they got change except the variable which is created by module_param_cb() macro.

Download the code by clicking below link.

Compiling

This is the code of Makefile.

In terminal enter sudo make

```
ox:/home/driver/simple_driver$ sudo make
[sudo] password for optimus:
make -C /lib/modules/4.4.0-59-generic/build  M=/home/driver/simple_driver modules
make[1]: Entering directory `/usr/src/linux-headers-4.4.0-59-generic'
  CC [M]  /home/driver/simple_driver/hello_world_module.o
  Building modules, stage 2.
  MODPOST 1 modules
  LD [M]  /home/driver/simple_driver/hello_world_module.ko
make[1]: Leaving directory `/usr/src/linux-headers-4.4.0-59-generic'
ox:/home/driver/simple_driver$
```


Loading the Driver

sudo insmod hello_world_module.ko valueETX=14 nameETX="EmbeTronicX" arr_valueETX=100,102,104,106

Verify the parameters by using dmesg

Now our module got loaded. now check dmesg. In below picture, every value got passed to our device driver

Now i'm going to check module_param_cb() is weather calling that handler function or not. For that i need to change the variable in sysfs.

echo 13 > /sys/module/hello_world_module/parameters/cb_valueETX

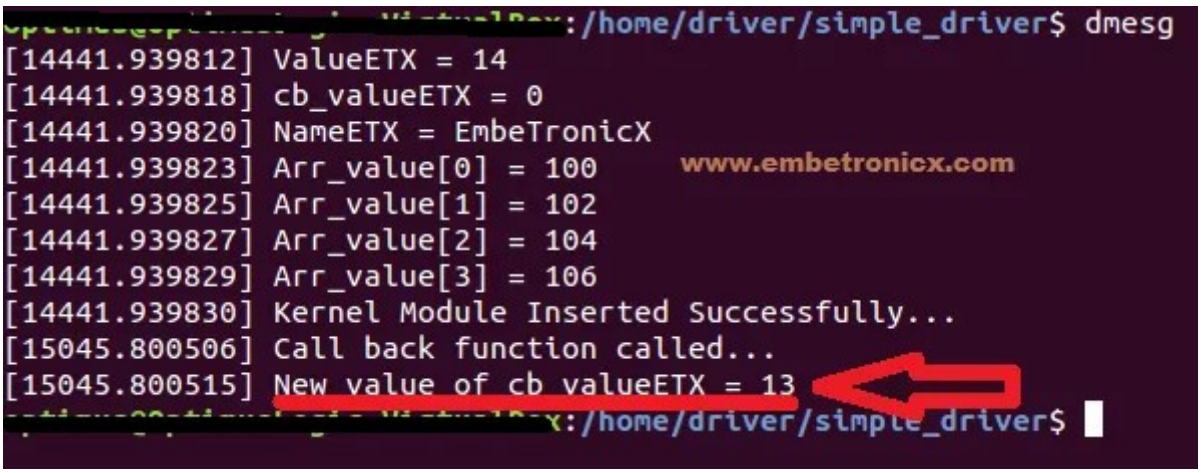
```
:/home/driver/simple_driver$ sudo insmod hello_world_module.ko valueETX=14 nameETX="EmbeTronicX" arr_valueETX=100,102,104,106
:/home/driver/simple_driver$
```



```
:/home/driver/simple_driver$ dmesg
[14441.939812] ValueETX = 14
[14441.939818] cb_valueETX = 0
[14441.939820] NameETX = EmbeTronicX
[14441.939823] Arr_value[0] = 100
[14441.939825] Arr_value[1] = 102
[14441.939827] Arr_value[2] = 104
[14441.939829] Arr_value[3] = 106
[14441.939830] Kernel Module Inserted Successfully...
:/home/driver/simple_driver$
```



```
:/home/driver/simple_driver# ls /sys/module/hello_world_module/parameters/
arr_valueETX  cb_valueETX  nameETX  valueETX
/home/driver/simple_driver# echo 13 > /sys/module/hello_world_module/parameters/cb_valueETX
/home/driver/simple_driver#
```



See the above result. So Our callback function got called. But if you change the value of other variables, you wont get notification.

Unloading the Driver

Finally unload the driver by using sudo rmmod hello_world_module.

Character Device Driver

Introduction

We already know what drivers are, and why we need them. What is so special about character drivers? If we write drivers for byte-oriented operations then we refer to them as character drivers. Since the majority of devices are byte-oriented, the majority of device drivers are character device drivers. Take, for example, serial drivers, audio drivers, video drivers, camera drivers, and basic I/O drivers. In fact, all device drivers that are neither storage nor network device drivers are some type of a character driver

How Applications will communicate Hardware device?

This below diagram will show the full path of the communication.

- First Application will open the device file. This device file is created by device driver).
- Then This device file will find the correspond device driver using major and minor number.
- Then That Device driver will talk to Hardware device.

Character Device Driver Major Number and Minor Number

One of the basic features of the Linux kernel is that it abstracts the handling of devices. All hardware devices look like regular files; they can be opened, closed, read and written using the same, standard, system calls that are used to manipulate files. To Linux, everything is a file. To write to the hard disk, you write to a file. To read from the keyboard is to read from a file. To store backups on a tape device is to write to a file. Even to read from memory is to read from a file. If the file from which you are trying to read or to which you are trying to write is a “normal” file, the process is fairly easy to understand: the file is opened and you read or write data. So device driver also like file. Driver will create the special file for every hardware devices. We can communicate to the hardware using those special files (device file).

If you want to create the special file, we should know about the Major number and minor number in device driver. In this tutorial we will learn that major and minor number.

Major Number and Minor Number

The Linux kernel represents character and block devices as pairs of numbers <major>:<minor>.

Major number

Traditionally, the major number identifies the driver associated with the device. A major number can also be shared by multiple device drivers. See /proc/devices to find out how major numbers are assigned on a running Linux instance.

```
linux@embetronicx-VirtualBox:~/project/devicedriver/devicefile$ cat
/proc/devices

Character devices:

1 mem

4 /dev/vc/0

4 tty

Block devices:

1 ramdisk

259 blkext
```

These numbers are major numbers.

Minor Number

Major number is identify the corresponding driver. Many devices may use same major number. So we need to assign the number to each devices which is using same major number. So this is the minor number. In other words, The device driver uses the minor number <minor> to distinguish individual physical or logical devices.

Allocating Major and Minor Number

We can allocating the major and minor numbers by two ways.

1. Statically allocating
2. Dynamically Allocating

Statically allocating

If you want to set the particular major number to your driver, you can use this method. This method will allocate that major number if it is available. Otherwise it won't.

int register_chrdev_region(dev_t first, unsigned int count, char *name);

Here, first is the beginning device number of the range you would like to allocate.

count is the total number of contiguous device numbers you are requesting. Note that, if count is large, the range you request could spill over to the next major number; but everything will still work properly as long as the number range you request is available.

name is the name of the device that should be associated with this number range; it will appear in /proc/devices and sysfs.

The return value from register_chrdev_region will be 0 if the allocation was successfully performed. In case of error, a negative error code will be returned, and you will not have access to the requested region.

The dev_t type (defined in <linux/types.h>) is used to hold device numbers—both the major and minor parts. dev_t is a 32-bit quantity with 12 bits set aside for the major number and 20 for the minor number.

If you want to create the dev_t structure variable for your major and minor number, please use below function.

MKDEV(int major, int minor);

If you want to get your major number and minor number from dev_t, use below method.

MAJOR(dev_t dev);

MINOR(dev_t dev);

If you pass the dev_t structure to this MAJOR or MONOR fuction, it will return that major/minor number of your driver.

Example,

dev_t dev = MKDEV(235, 0); register_chrdev_region(dev, 1, "Embetronicx_Dev");

```
   dev_t dev = MKDEV(235,
1 0);
2
3 register_chrdev_region(dev,
   1, "Embetronicx_Dev");
```

Dynamically Allocating

If we dont want fixed major and minor number please use this method. This method will allocate the major number dynamically to your driver which is available.

int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);

dev is an output-only parameter that will, on successful completion, hold the first number in your allocated range.

firstminor should be the requested first minor number to use; it is usually 0.

count is the total number of contiguous device numbers you are requesting.

name is the name of the device that should be associated with this number range; it will appear in /proc/devices and sysfs

Difference between static and dynamic method

Static method is only really useful if you know in advance which major number you want to start with. With Static method , you tell the kernel what device numbers you want (the start major/minor number and count) and it either gives them to you or not (depending on availability).

With Dynamic method, you tell the kernel how many device numbers you need (the starting minor number and count) and it will find a starting major number for you, if one is available, of course.

Partially to avoid conflict with other device drivers, it’s considered preferable to use the Dynamic method function, which will dynamically allocate the device numbers for you.

The disadvantage of dynamic assignment is that you can’t create the device nodes in advance, because the major number assigned to your module will vary. For normal use of

the driver, this is hardly a problem, because once the number has been assigned, you can read it from /proc/devices.

Unregister the Major and Minor Number

Regardless of how you allocate your device numbers, you should free them when they are no longer in use. Device numbers are freed with:

void unregister_chrdev_region(dev_t first, unsigned int count);

The usual place to call unregister_chrdev_region would be in your module's cleanup function (Exit Function).

Program for Statically Allocating Major Number

In this program, I'm assigning 235 as a major number.

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

dev_t dev = MKDEV(235, 0);
static int __init hello_world_init(void)
{
register_chrdev_region(dev, 1, "Embetronicx_Dev");
printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
printk(KERN_INFO "Kernel Module Inserted Successfully...\n");
return 0;
}

void __exit hello_world_exit(void)
{
unregister_chrdev_region(dev, 1);
printk(KERN_INFO "Kernel Module Removed Successfully...\n");
}

module_init(hello_world_init);
module_exit(hello_world_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com or
admin@embetronicx.com>");
MODULE_DESCRIPTION("A simple hello world driver");
MODULE_VERSION("1.0");
```

Build the driver by using Makefile (sudo make)

- Load the driver using sudo insmod
- Check the major number using cat /proc/devices

  linux@embetronicx-VirtualBox::/home/driver/driver$ cat /proc/devices | grep "Embetronicx_Dev"

  235 Embetronicx_Dev

- Unload the driver using sudo rmmod

Program for Dynamically Allocating Major Number

This program will allocate major number dynamically.

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

dev_t dev = 0;

static int __init hello_world_init(void)
{
/*Allocating Major number*/
if((alloc_chrdev_region(&dev, 0, 1, "Embetronicx_Dev")) <0){
printk(KERN_INFO "Cannot allocate major number for device 1\n");
return -1;
}
printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
printk(KERN_INFO "Kernel Module Inserted Successfully...\n");
return 0;
}

void __exit hello_world_exit(void)
{
unregister_chrdev_region(dev, 1);
printk(KERN_INFO "Kernel Module Removed Successfully...\n");
}

module_init(hello_world_init);
module_exit(hello_world_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com or
admin@embetronicx.com>");
MODULE_DESCRIPTION("A simple hello world driver");
MODULE_VERSION("1.1");
```

Build the driver by using Makefile (sudo make)

- Load the driver using sudo insmod
- Check the major number using cat /proc/devices

  linux@embetronicx-VirtualBox::/home/driver/driver$ cat /proc/devices | grep "Embetronicx_Dev"

  243 Embetronicx_Dev

- Unload the driver using sudo rmmod

This function allocates major number of 243 for this driver.

Before unloading the driver just check the files in /dev directory using ls /dev/. You wont find our driver file. Because we haven't created yet. In our next tutorial we will see that device file.

Device File Creation

Device File Creation for Character Drivers

In our Last tutorial we have seen how to assign major and minor number. But if you see there it will create major and minor number. But i wont create any device files in /dev/ directory. Device file is important to communicate to hardware. Let's start our tutorial.

Device Files

Device file allows transparent communication between user space applications and hardware.

They are not normal "files", but look like files from the program's point of view: you can read from them, write to them, mmap() onto them, and so forth. When you access such a device "file," the kernel recognizes the I/O request and passes it a device driver, which performs some operation, such as reading data from a serial port, or sending data to a hardware.

Device files (although inappropriately named, we will continue to use this term) provide a convenient way to access system resources without requiring the applications programmer to know how the underlying device works. Under Linux, as with most Unix systems, device drivers themselves are part of the kernel.

All device files are stored in /dev directory. Use ls command to browse the directory:

ls -l /dev/

Each device on the system should have a corresponding entry in /dev. For example, */dev/ttyS0* corresponds to the first serial port, known as COM1 under MS-DOS; /dev/hda2 corresponds to the second partition on the first IDE drive. In fact, there should be entries in /dev for devices you do not have. The device files are generally created during system installation and include every possible device driver. They don't necessarily correspond to the actual hardware on your system. There are a number of pseudo-devices in /dev that don't correspond to any actual peripheral. For example, /dev/null acts as a byte sink; any write request to /dev/null will succeed, but the data written will be ignored.

When using ls -l to list device files in /dev, you'll see something like the following:

crw--w---- 1 root tty **4**, **0** Aug 15 10:40 tty0

brw-rw---- 1 root disk **1**, **0** Aug 15 10:40 ram0

First of all, note that the first letter of the permissions field is denoted that driver type. Device files are denoted either by b, for block devices, or c, for character devices.

Also, note that the size field in the ls -l listing is replaced by two numbers, separated by a comma. The first value is the major device number and the second is the minor device number. This we have discussed in previous tutorial.

Creating Device File

We can create the dive file in two ways.

1. Manually
2. Automatically

We will see one by one.

Manually Creating Device File

We can create the device file manually by using mknod.

mknod -m <permissions><name><devicetype><major><minor>

<name> – your device file name that should have full path (/dev/name)

<device type> – Put **c** or **b**

c – Character Device

b – Block Device

<major> – major number of your driver

<minor> – minor number of your driver

-m<permissions> –optional argument that sets the permission bits of the new device file to permissions

Example:

sudo mknod -m 666 /dev/etx_device c 246 0

If you don’t want to give permission, You can also use chmod to set the permissions for a device file after creation.

Advantages

- Anyone can create the device file using this method.
- You can create the device file even before load the driver.

Programming

I took this program from previous tutorial. I’m going to create device file manually for this driver.

Build the driver by using Makefile (sudo make)

- Load the driver using sudo insmod
- Check the device file using ls -l /dev/. By this time device file is not created for your driver.
- Create device file using mknod and then check using ls -l /dev/.

  linux@embetronicx-VirtualBox:/home/driver/driver$sudo mknod -m 666 /dev/etx_device c 246 0

  linux@embetronicx-VirtualBox::/home/driver/driver$ ls -l /dev/ | grep "etx_device"

  crw-rw-rw- 1 root root 246, 0 Aug 15 13:53 etx_device

- Now our device file got created and registered with major number.
- Unload the driver using sudo rmmod

Automatically Creating Device File

The automatic creation of device files can be handled with udev. Udev is the device manager for the Linux kernel that creates/removes device nodes in the /dev directory dynamically. Just follow the below steps.

1. Include the header file **linux/device.h** and **linux/kdev_t.h**
2. Create the struct Class
3. Create Device with the class which is created by above step

Create the class

This will create the struct class for our device driver. It will create structure under/sys/class/.

struct class * class_create (struct module *owner, const char *name);

owner – pointer to the module that is to “own” this struct class

name – pointer to a string for the name of this class

This is used to create a struct class pointer that can then be used in calls to class_device_create.

Note, the pointer created here is to be destroyed when finished by making a call to class_destroy.

void class_destroy (struct class * cls);

Create Device

This function can be used by char device classes. A struct device will be created in sysfs, registered to the specified class.

struct device *device_create (struct *class, struct device *parent,

dev_t dev, const char *fmt, ...);

class – pointer to the struct class that this device should be registered to

parent – pointer to the parent struct device of this new device, if any

devt – the dev_t for the char device to be added

fmt – string for the device’s name

… – variable arguments

A “dev” file will be created, showing the dev_t for the device, if the dev_t is not 0,0. If a pointer to a parent struct device is passed in, the newly created struct device will be a child of that device in sysfs. The pointer to the struct device will be returned from the call. Any further sysfs files that might be required can be created using this pointer.

Note, you can destroy the device using device_destroy().

void device_destroy (struct class * class, dev_t devt);

If you don’t understand please refer the below program, Then you will understand.

Cdev structure and File Operations

Cdev structure and File Operations of Character drivers

If we want to open, read, write and close we need to register some structures to driver.

cdev structure

In linux kernel struct inode structure is used to represent files. Therefore, it is different from the file structure that represents an open file descriptor. There can be numerous file structures representing multiple open descriptors on a single file, but they all point to a single inode structure.

The inode structure contains a great deal of information about the file. As a general rule, cdev structure is useful for writing driver code:

struct cdev is one of the elements of the inode structure. As you probably may know already, an inode structure is used by the kernel internally to represent files. The struct cdev is the kernel's internal structure that represents char devices. This field contains a pointer to that structure when the inode refers to a char device file.

```
struct cdev {
struct kobject kobj;
struct module *owner;
const struct file_operations *ops;
struct list_head list;
dev_t dev;
unsigned int count;
};
```

This is cdev structure. Here we need to fill two fields,

1. file_operation (This we will see after this cdev structure)
2. owner (This should be THIS_MODULE)

There are two ways of allocating and initializing one of these structures.

1. Runtime Allocation
2. Own allocation

If you wish to obtain a standalone cdev structure at runtime, you may do so with code such as:

```
struct cdev *my_cdev = cdev_alloc( );
my_cdev->ops = &my_fops;
```

Or else you can embed the cdev structure within a device-specific structure of your own by using below function.

void cdev_init(struct cdev *cdev, struct file_operations *fops);

Once the cdev structure is set up with file_operations and owner, the final step is to tell the kernel about it with a call to:

int cdev_add(struct cdev *dev, dev_t num, unsigned int count);

Here,

dev is the cdev structure,

num is the first device number to which this device responds, and

count is the number of device numbers that should be associated with the device. Often count is one, but there are situations where it makes sense to have more than one device number correspond to a specific device.

If this function returns negative error code, your device has not been added to the system. So check the return value of this function.

After a call to cdev_add(), your device is immediately alive. All functions you defined (through the file_operations structure) can be called.

To remove a char device from the system, call:

void cdev_del(struct cdev *dev);

Clearly, you should not access the cdev structure after passing it to cdev_del.

File_Operations

The file_operations structure is how a char driver sets up this connection. The structure, defined in *<linux/fs.h>*, is a collection of function pointers. Each open file is associated with its own set of functions . The operations are mostly in charge of implementing the system calls and are therefore, named open, read, and so on.

A file_operations structure or a pointer to one is called fops. Each field in the structure must point to the function in the driver that implements a specific operation, or be left NULL for unsupported operations. The whole structure is mentioned below snippet

```
struct file_operations {
   struct module *owner;
   loff_t (*llseek) (struct file *, loff_t, int);
   ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
   ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
```

```
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned
int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned
int);
    int (*setlease)(struct file *, long, struct file_lock **, void **);
    long (*fallocate)(struct file *file, int mode, loff_t offset,
              loff_t len);
    void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
    unsigned (*mmap_capabilities)(struct file *);
#endif
    ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
            loff_t, size_t, unsigned int);
    int (*clone_file_range)(struct file *, loff_t, struct file *, loff_t,
            u64);
    ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file *,
            u64);
};
```

This file_operations structure contains many fields. But we will concentrate on very basic functions. Below we will see some fields explanation.

struct module *owner:

The first file_operations field is not an operation at all; it is a pointer to the module that “owns” the structure. This field is used to prevent the module from being unloaded while its operations are in use. Almost all the time, it is simply initialized to THIS_MODULE, a macro defined in *<linux/module.h>*.

ssize_t (*read) (struct file *, char _ _user *, size_t, loff_t *);

Used to retrieve data from the device. A null pointer in this position causes the read system call to fail with -EINVAL (“Invalid argument”). A non negative return value represents the number of bytes successfully read (the return value is a “signed size” type, usually the native integer type for the target platform).

ssize_t (*write) (struct file *, const char _ _user *, size_t, loff_t *);

Sends data to the device. If NULL, -EINVAL is returned to the program calling the write system call. The return value, if non negative, represents the number of bytes successfully written.

int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);

The ioctl system call offers a way to issue device-specific commands (such as formatting a track of a floppy disk, which is neither reading nor writing). Additionally, a few ioctl commands are recognized by the kernel without referring to the fops table. If the device doesn’t provide an ioctl method, the system call returns an error for any request that isn’t predefined (-ENOTTY, “No such ioctl for device”).

int (*open) (struct inode *, struct file *);

Though this is always the first operation performed on the device file, the driver is not required to declare a corresponding method. If this entry is NULL, opening the device always succeeds, but your driver isn’t notified.

int (*release) (struct inode *, struct file *);

This operation is invoked when the file structure is being released. Like open, release can be NULL.

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

If you want to understand the complete flow just have a look at our dummy driver.

Dummy Driver

Here i have added dummy driver snippet. In this driver code, we can do all open, read, write, close operations. Just go through the code.

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

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);
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 * off);
static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * off);

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

static int etx_open(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "Driver Open Function Called...!!!\n");
        return 0;
}

static int etx_release(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "Driver Release Function Called...!!!\n");
        return 0;
}

static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
{
        printk(KERN_INFO "Driver Read Function Called...!!!\n");
        return 0;
```

```
}
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t *off)
{
        printk(KERN_INFO "Driver Write Function Called...!!!\n");
        return len;
}


static int __init etx_driver_init(void)
{
        /*Allocating Major number*/
        if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
                printk(KERN_INFO "Cannot allocate major number\n");
                return -1;
        }
        printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

        /*Creating cdev structure*/
        cdev_init(&etx_cdev,&fops);

        /*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;
        }
        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)
```

```
{
        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("1.3");
```

Build the driver by using Makefile (`sudo make`)

1. Load the driver using *sudo insmod*
2. Do echo 1 > /dev/etx_device

Echo will open the driver and write 1 into the driver and finally close the driver. So if i do echo to our driver, it should call the open, write and release functions. Just check.

linux@embetronicx-VirtualBox:/home/driver/driver# ***echo 1 > /dev/etx_device***

4. Now Check using dmesg

linux@embetronicx-VirtualBox:/home/driver/driver$ **dmesg**
[19721.611967] Major = 246 Minor = 0
[19721.618716] Device Driver Insert...Done!!!
[19763.176347] Driver Open Function Called...!!!
[19763.176363] Driver Write Function Called...!!!
[19763.176369] Driver Release Function Called...!!!

5. Do cat > /dev/etx_device

Cat command will open the driver, read the driver and close the driver. So if i do cat to our driver, it should call the open, read and release functions. Just check.

linux@embetronicx-VirtualBox:/home/driver/driver# ***cat /dev/etx_device***

6. Now Check using dmesg

linux@embetronicx-VirtualBox:/home/driver/driver$ **dmesg**
[19763.176347] Driver Open Function Called...!!!
[19763.176363] Driver Read Function Called...!!!
[19763.176369] Driver Release Function Called...!!!

7. Unload the driver using *sudo rmmod*

Instead of doing echo and cat command in terminal you can also use open(), read(), write(), close() system calls from user space application.

Linux Device Driver Tutorial Programming

Introduction

We already know that in Linux everything is a File. So in this tutorial we are going to develop two applications.

1. User Space application (User program)
2. Kernel Space program (Driver)

User Program will communicate with the kernel space program using device file. Lets Start.

Kernel Space Program (Device Driver)

We already know about major, minor number, device file and file operations of device driver. If you don't know please visit our previous tutorials. Now we are going to discuss more about file operations in device driver. Basically there are four functions in device driver.

1. Open driver
2. Write Driver
3. Read Driver
4. Close Driver

Now we will see one by one of this functions. Before that i will explain the concept of this driver.

Concept

Using this driver we can send string or data to the kernel device driver using write function. It will store those string in kernel space. Then when i read the device file, it will send the data which is written by write by function.

Functions used in this driver

- kmalloc()
- kfree()
- copy_from_user()
- copy_to_user()

kmalloc()

We will see the memory allocation methods available in kernel, in future tutorial. But now we will use only kmalloc in this tutorial.

kmalloc function is used to allocate the memory in kernel space. This is like a malloc() function in user space. The function is fast (unless it blocks) and doesn't clear the memory it obtains. The allocated region still holds its previous content. The allocated region is also contiguous in physical memory.

#include <linux/slab.h>

void *kmalloc(size_t size, gfp_t flags);

Arguments

size_t size– how many bytes of memory are required.

gfp_t flags– the type of memory to allocate.

The *flags* argument may be one of:

GFP_USER – Allocate memory on behalf of user. May sleep.

GFP_KERNEL – Allocate normal kernel ram. May sleep.

GFP_ATOMIC – Allocation will not sleep. May use emergency pools. For example, use this inside interrupt handlers.

GFP_HIGHUSER – Allocate pages from high memory.

GFP_NOIO – Do not do any I/O at all while trying to get memory.

GFP_NOFS – Do not make any fs calls while trying to get memory.

GFP_NOWAIT – Allocation will not sleep.

__GFP_THISNODE – Allocate node-local memory only.

GFP_DMA – Allocation suitable for DMA. Should only be used for kmalloc caches. Otherwise, use a slab created with SLAB_DMA.

Also it is possible to set different flags by OR'ing in one or more of the following additional *flags*:

__GFP_COLD – Request cache-cold pages instead of trying to return cache-warm pages.

__GFP_HIGH – This allocation has high priority and may use emergency pools.

__GFP_NOFAIL – Indicate that this allocation is in no way allowed to fail (think twice before using).

__GFP_NORETRY – If memory is not immediately available, then give up at once.

__GFP_NOWARN – If allocation fails, don't issue any warnings.

__GFP_REPEAT – If allocation fails initially, try once more before failing.

There are other flags available as well, but these are not intended for general use, and so are not documented here. For a full list of potential flags, always refer to linux/gfp.h.

kfree()

This is like a free() function in user space. This is used to free the previously allocated memory.

void kfree(const void *objp)

Arguments

*objp – pointer returned by kmalloc

copy_from_user()

This function is used to Copy a block of data from user space (Copy data from user space to kernel space).

unsigned long copy_from_user(void *to, const void __user *from, unsigned long n);

Arguments

to – Destination address, in kernel space

from – Source address in user space

n – Number of bytes to copy

Returns number of bytes that could not be copied. On success, this will be zero.

copy_to_user()

This function is used to Copy a block of data into user space (Copy data from kernel space to user space).

unsigned long copy_to_user(const void __user *to, const void *from, unsigned long n);

Arguments

to – Destination address, in user space

from – Source address in kernel space

n – Number of bytes to copy

Returns number of bytes that could not be copied. On success, this will be zero.

Open()

This function is called first, whenever we are opening the device file. In this function i am going to allocate the memory using kmalloc. In user space application you can use open() system call to open the device file.

static int etx_open(struct inode *inode, struct file *file) { /*Creating Physical memory*/ if((kernel_buffer = kmalloc(mem_size , GFP_KERNEL)) == 0){ printk(KERN_INFO "Cannot allocate memory in kernel\n"); return -1; } printk(KERN_INFO "Device File Opened...!!!\n"); return 0; }

```
static int etx_open(struct inode *inode, struct file
*file)
{
        /*Creating Physical memory*/
        if((kernel_buffer = kmalloc(mem_size ,
GFP_KERNEL)) == 0){
            printk(KERN_INFO "Cannot allocate
memory in kernel\n");
            return -1;
        }
        printk(KERN_INFO "Device File
Opened...!!!\n");
        return 0;
}
```

write()

When write the data to the device file it will call this write function. Here i will copy the data from user space to kernel space using copy_from_user() function. In user space application you can use write() system call to write any data the device file.

static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t *off) { copy_from_user(kernel_buffer, buf, len); printk(KERN_INFO "Data Write : Done!\n"); return len; }

```
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len,
loff_t *off)
{
        copy_from_user(kernel_buffer, buf, len);
        printk(KERN_INFO "Data Write : Done!\n");
        return len;
}
```

read()

When we read the device file it will call this function. In this function i used copy_to_user(). This function is used to copy the data to user space application. In user space application you can use read() system call to read the data from the device file.

```
static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
{ copy_to_user(buf, kernel_buffer, mem_size); printk(KERN_INFO "Data Read :
Done!\n"); return mem_size; }
```

```
1 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len,
2 loff_t *off)
3 {
4         copy_to_user(buf, kernel_buffer, mem_size);
5         printk(KERN_INFO "Data Read : Done!\n");
6         return mem_size;
  }
```

close()

When we close the device file that will call this function. Here i will free the memory that is allocated by kmalloc using kfree(). In user space application you can use close() system call to close the device file.

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

```
   static int etx_release(struct inode *inode, struct file
1 *file)
2 {
3         kfree(kernel_buffer);
4         printk(KERN_INFO "Device File
5 Closed...!!!\n");
6         return 0;
  }
```

Full Driver Code

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include<linux/slab.h>                 //kmalloc()
```

```
#include<linux/uaccess.h>              //copy_to/from_user()


#define mem_size        1024

dev_t dev = 0;
static struct class *dev_class;
static struct cdev etx_cdev;
uint8_t *kernel_buffer;

static int __init etx_driver_init(void);
static void __exit etx_driver_exit(void);
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 * off);
static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * off);

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

static int etx_open(struct inode *inode, struct file *file)
{
        /*Creating Physical memory*/
        if((kernel_buffer = kmalloc(mem_size , GFP_KERNEL)) == 0){
            printk(KERN_INFO "Cannot allocate memory in kernel\n");
            return -1;
        }
        printk(KERN_INFO "Device File Opened...!!!\n");
        return 0;
}

static int etx_release(struct inode *inode, struct file *file)
{
        kfree(kernel_buffer);
        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 *off)
{
        copy_to_user(buf, kernel_buffer, mem_size);
        printk(KERN_INFO "Data Read : Done!\n");
```

```
        return mem_size;
}
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t *off)
{
        copy_from_user(kernel_buffer, buf, len);
        printk(KERN_INFO "Data Write : Done!\n");
        return len;
}

static int __init etx_driver_init(void)
{
        /*Allocating Major number*/
        if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
                printk(KERN_INFO "Cannot allocate major number\n");
                return -1;
        }
        printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

        /*Creating cdev structure*/
        cdev_init(&etx_cdev,&fops);

        /*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;
        }
        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)
{
        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("1.4");
```

Building the Device Driver

1. Build the driver by using Makefile (*sudo make*) You can download the Makefile Here.

User Space Application

This application will communicate with the device driver. You can download the all codes (driver, Makefile and application) Here.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int8_t write_buf[1024];
int8_t read_buf[1024];
int main()
{
        int fd;
        char option;
        printf("*********************************\n");
        printf("*******WWW.EmbeTronicX.com*******\n");

        fd = open("/dev/etx_device", O_RDWR);
        if(fd < 0) {
                printf("Cannot open device file...\n");
```

```
        return 0;
    }

    while(1) {
        printf("****Please Enter the Option******\n");
        printf("        1. Write             \n");
        printf("        2. Read              \n");
        printf("        3. Exit              \n");
        printf("*********************************\n");
        scanf(" %c", &option);
        printf("Your Option = %c\n", option);

        switch(option) {
            case '1':
                printf("Enter the string to write into driver :");
                scanf("  %[^\t\n]s", write_buf);
                printf("Data Writing ...");
                write(fd, write_buf, strlen(write_buf)+1);
                printf("Done!\n");
                break;
            case '2':
                printf("Data Reading ...");
                read(fd, read_buf, 1024);
                printf("Done!\n\n");
                printf("Data = %s\n\n", read_buf);
                break;
            case '3':
                close(fd);
                exit(1);
                break;
            default:
                printf("Enter Valid option = %c\n",option);
                break;
        }
    }
    close(fd);
}
```

Compile the User Space Application

Use below line in terminal to compile the user space application.

```
gcc -o test_app test_app.c
```

Execution (Output)

As of now, we have driver.ko and test_app. Now we will see the output.

- Load the driver using sudo insmod driver.ko
- Run the application (sudo ./test_app)

```
**********************************
*******WWW.EmbeTronicX.com*******
****Please Enter the Option******
1. Write
2. Read
3. Exit
**********************************
```

- Select option 1 to write data to driver and write the string (In this case i’m going to write “embetronicx” to driver.

```
1
Your Option = 1
Enter the string to write into driver :embetronicx
Data Writing ...Done!
****Please Enter the Option******
1. Write
2. Read
3. Exit
**********************************
```

- That “embetronicx” string got passed to the driver. And driver stored that string in the kernel space. That kernel space was allocated by kmalloc.
- Now select the option 2 to read the data from the device driver.

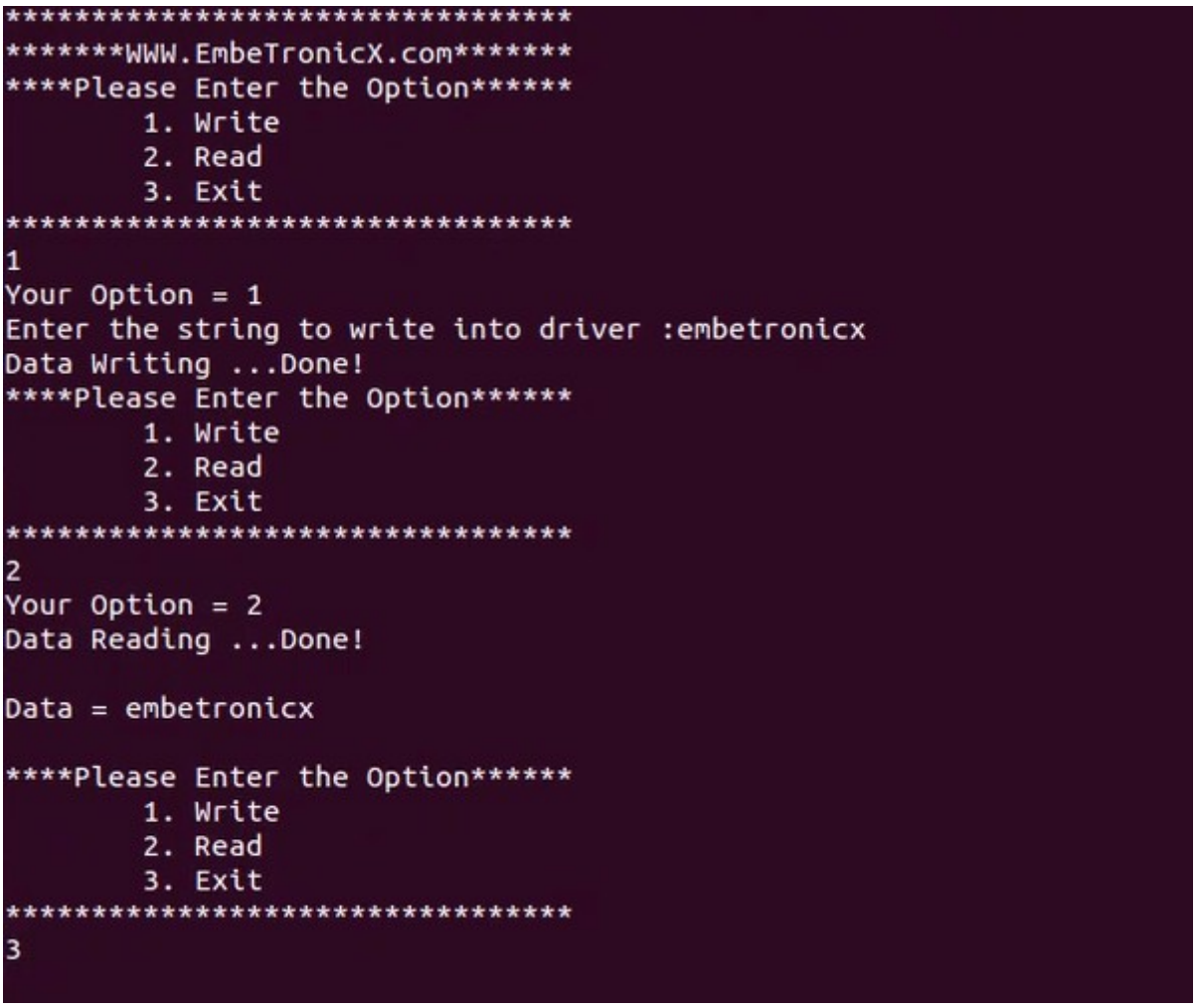
```
2
Your Option = 2
Data Reading ...Done!

Data = embetronicx
```

- See now, we got that string “embetronicx”.

Just see the below image for your clarification.

Note : Instead of using user space application, you can use echo and cat command. But one condition. If you are going to use echo and cat command, please allocate the kernel space memory in init function instead of open() function. I wont say why. You have to find the reason. If you found the reason please comment below. You can use dmesg to see the kernel log.



I/O Control in Linux IOCTL()

Introduction

Operating system segregates virtual memory into kernel space and user space. 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 User space and Kernel Space, they are:

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

- Netlink Sockets

In this tutorial we will see IOCTL.

IOCTL Tutorial in Linux

IOCTL

IOCTL is referred as Input and Output Control, which is used to talking to device drivers. This system call, available in most driver categories. The major use of this is in case of handling some specific operations of a device for which the kernel does not have a system call by default.

Some real time applications of ioctl is Ejecting the media from a “cd” drive, to change the Baud Rate of Serial port, Adjust the Volume, Reading or Writing device registers, etc. We already have write and read function in our device driver. But it is not enough for all cases.

Steps involved in IOCTL

There are some steps involved to use IOCTL.

- Create IOCTL command in driver
- Write IOCTL function in driver
- Create IOCTL command in User space application
- Use IOCTL system call in User space

Create IOCTL Command in Driver

To implement a new ioctl command we need to follow the following steps.

1. Define the ioctl code

#define "ioctl name" __IOX("magic number","command number","argument type")

where *IOX* can be :
“IO”: an ioctl with no parameters
“IOW”: an ioctl with write parameters (copy_from_user)
“IOR”: an ioctl with read parameters (copy_to_user)
“IOWR”: an ioctl with both write and read parameters

- The Magic Number is a unique number or character that will differentiate our set of ioctl calls from the other ioctl calls. some times the major number for the device is used here.
- Command Number is the number that is assigned to the ioctl .This is used to differentiate the commands from one another.
- The last is the type of data.

2. Add the header file linux/ioctl.h to make use of the above mentioned calls.

Example:

#include <linux/ioctl.h> #define WR_VALUE _IOW('a','a',int32_t*) #define RD_VALUE _IOR('a','b',int32_t*)

```
#include
<linux/ioctl.h>

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

Write IOCTL function in driver

The next step is to implement the ioctl call we defined in to the corresponding driver. We need to add the ioctl function which has the prototype.

Where
<file> : is the file pointer to the file that was passed by the application.
<cmd> : is the ioctl command that was called from the user space.
<arg> : are the arguments passed from the user space.

With in the function “ioctl” we need to implement all the commands that we defined above. We need to use the same commands in switch statement which is defined above.

Then need to inform the kernel that the ioctl calls are implemented in the function “etx_ioctl”. This is done by making the fops pointer “unlocked_ioctl” to point to “etx_ioctl” as shown below.

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

Now we need to call the new ioctl command from a user application.

Create IOCTL command in User space application

Just define the ioctl command like how we defined in driver.

Example:

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

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

Use IOCTL system call in User space

Include the header file <sys/ioctl.h>.Now we need to call the new ioctl command from a user application.

long ioctl("file descriptor","ioctl command","Arguments");

<file descriptor>: This the open file on which the ioctl command needs to be executed, which would generally be device files.
<ioctl command>: ioctl command which is implemented to achieve the desired functionality
<arguments>: The arguments that needs to be passed to the ioctl command.

Example:

ioctl(fd, WR_VALUE, (int32_t*) &number); ioctl(fd, RD_VALUE, (int32_t*) &value);

```
1 ioctl(fd, WR_VALUE, (int32_t*)
2 &number);
3 ioctl(fd, RD_VALUE, (int32_t*) &value);
```

Now we will see the complete driver and application.

Device Driver Source Code

In this example we only implemented IOCTL. In this driver, i defined one variable (int32_t value). Using ioctl command we can read or change the variable. So other functions like open, close, read, write, We simply left empty. Just go through the code below.

driver.c

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include<linux/slab.h>                 //kmalloc()
#include<linux/uaccess.h>              //copy_to/from_user()
#include <linux/ioctl.h>


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

int32_t value = 0;

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);
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 * off);
static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * off);
static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg);

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 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 *off)
{
        printk(KERN_INFO "Read Function\n");
        return 0;
}
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t *off)
{
        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){
                printk(KERN_INFO "Cannot allocate major number\n");
                return -1;
        }
        printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
```

```
        /*Creating cdev structure*/
        cdev_init(&etx_cdev,&fops);

        /*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;
        }
        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)
{
        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("1.5");
```

Makefile:

obj-m += driver.o KDIR = /lib/modules/$(shell uname -r)/build all: make -C $(KDIR) M=$ (shell pwd) modules clean: make -C $(KDIR) M=$(shell pwd) clean

```
    obj-m +=
    driver.o

    KDIR =
1   /lib/modules/$
2   (shell uname
3   -r)/build
4
5
6   all:
7       make -C $
8   (KDIR)  M=$
9   (shell pwd)
1   modules
0
    clean:
        make -C $
    (KDIR)  M=$
    (shell pwd) clean
```

Application Source Code

This application is used to write the value to the driver. Then read the value again.

test_app.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include<sys/ioctl.h>

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

int main()
{
        int fd;
```

```
        int32_t value, number;
        printf("*********************************\n");
        printf("*******WWW.EmbeTronicX.com*******\n");

        printf("\nOpening Driver\n");
        fd = open("/dev/etx_device", O_RDWR);
        if(fd < 0) {
                printf("Cannot open device file...\n");
                return 0;
        }

        printf("Enter the Value to send\n");
        scanf("%d",&number);
        printf("Writing Value to Driver\n");
        ioctl(fd, WR_VALUE, (int32_t*) &number);

        printf("Reading Value from Driver\n");
        ioctl(fd, RD_VALUE, (int32_t*) &value);
        printf("Value is %d\n", value);

        printf("Closing Driver\n");
        close(fd);
}
```

Building Driver and Application

- Build the driver by using Makefile (*sudo make*)
- Use below line in terminal to compile the user space application.

```
gcc -o test_app test_app.c
```

Execution (Output)

As of now, we have driver.ko and test_app. Now we will see the output.

- Load the driver using sudo insmod driver.ko
- Run the application (sudo ./test_app)

```
*********************************
*******WWW.EmbeTronicX.com*******

Opening Driver
Enter the Value to send
```

- Enter the value to pass

```
23456
Writing Value to Driver
```

Reading Value from Driver
Value is 23456
Closing Driver

- Now check the value using dmesg

Device File Opened...!!!
Value = 23456
Device File Closed...!!!

- Our value 23456 was passed to the kernel and it was updated.

This is the simple example using ioctl in driver. If you want to send multiple arguments, put those variables into structure and pass the structure.

In our next tutorial we will see other userspace and kernel space communication methods.

Procfs in Linux

Introduction

Operating system segregates virtual memory into kernel space and user space. 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 User space 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.

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.

It can act as a bridge connecting the user space and the kernel space. User space program can use proc files to read the information exported by kernel. Every entry in the proc file system provides some information 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 needs to be writtable in this case.

For example, to enable IP forwarding of iptable, one can use the command below,

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 may be the data that module is handling. In such situations we can create a proc entry for our selves and dump what ever 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)
```

```
1 static inline struct proc_dir_entry *proc_create(const char *name, umode_t
2 mode,
3                                   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 parent, the /proc directory will be set as parent.
<proc_fops> : The structure in which the file operations for the proc entry will be created.

For example to create a proc entry by the name “etx_proc” under /proc the above function will be defined as below,

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

```
1 proc_create("etx_proc",0666,NULL,&proc_fo
  ps);
```

This proc entry should be created in Driver init function.

If you are using kernel version below 3.10, please use 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 to create both read as well as write entries.
create_proc_read_entry is a function specific to create only read entries.

Its 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 };

```
1 static struct file_operations proc_fops
2 = {
3     .open = open_proc,
4     .read = read_proc,
5     .write = write_proc,
6     .release = release_proc
  };
```

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

Next we need to add the all functions to the driver.

Open and Release Function

This 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 a 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 * off)
{ printk(KERN_INFO "proc file write.....\t"); copy_from_user(etx_array,buff,len); return len; }
```

```
static ssize_t write_proc(struct file *filp, const char *buff, size_t len, loff_t
* off)
{
    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_t * offset)
{
   printk(KERN_INFO "proc file read.....\n");
   if(len)
      len=0;
   else{
      len=1;
      return 0;
   }
   copy_to_user(buffer,etx_array,20);

   return length;;
}
```

Remove Proc Entry

Proc entry should be removed in 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);
```

```
1 remove_proc_entry("etx_proc",NUL
  L);
```

Complete Driver Code

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

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 which was passed to the driver using procfs.

This is the simple example using procfs in device driver. This is just a basic. I hope this might helped you.

Sysfs in Linux Kernel

Introduction

Operating system segregates virtual memory into kernel space and user space. 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 User space and Kernel Space, they are:

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

In this tutorial we will see Sysfs.

SysFS in Linux Kernel Tutorial

Introduction

Sysfs is a virtual filesystem exported by the kernel, similar to /proc. The files in Sysfs contain information about devices and drivers. Some files in Sysfs are even writable, for configuration and control of devices attached to the system. Sysfs is always mounted on /sys.

The directories in Sysfs contain the hierarchy of devices, as they are attached to the computer.

Sysfs is the commonly used method to export system information from the kernel space to the user space for specific devices. The sysfs is tied to the device driver model of the kernel. The procfs is used to export the process specific information and the debugfs is used to used for exporting the debug information by the developer.

Before get into the sysfs we should know about the Kernel Objects.

Kernel Objects

Heart of the sysfs model is the Kobject. Kobject is the glue that binds the sysfs and the kernel, which is represented by struct kobject and defined in <linux/kobject.h>. A struct

kobject represents a kernel object, maybe a device or so, such as the things that show up as directory in the sysfs filesystem.

Kobjects are usually embedded in other structures.

It is defined as,

#define KOBJ_NAME_LEN 20 struct kobject { char *k_name; char name[KOBJ_NAME_LEN]; struct kref kref; struct list_head entry; struct kobject *parent; struct kset *kset; struct kobj_type *ktype; struct dentry *dentry; };

```
    #define KOBJ_NAME_LEN
1   20
2
3   struct kobject {
4           char                *k_name;
5           char                name[KO
6   BJ_NAME_LEN];
7           struct kref         kref;
8           struct list_head    entry;
9           struct
1   kobject         *parent;
0           struct kset         *kset;
1           struct
1   kobj_type       *ktype;
1           struct dentry
2   *dentry;
    };
```

Some of the important fields are:

`struct kobject`

|– `name` (Name of the kobject. Current kobject are created with this name in *sysfs*.)

|– `parent` (This iskobject's parent. When we create a directory in sysfs for current kobject, it will create under this parent directory)

|– `ktype` (type associated with a kobject)

|– `kset` (group of kobjects all of which are embedded in structures of the same type)

|– `sd` (points to a sysfs_dirent structure that represents this kobject in sysfs.)

|– `kref` (provides reference counting)

It is the glue that holds much of the device model and its sysfs interface together.

So Kobj is used to create kobject directory in /sys. This is enough. We will not go deep into the kobjects

SysFS in Linux

There are several steps in creating and using sysfs.

1. Create directory in /sys
2. Create Sysfs file

Create directory in /sys

We can use this function (kobject_create_and_add) to create directory.

```
struct kobject * kobject_create_and_add ( const char * name, struct kobject * parent);
```

```
1 struct kobject * kobject_create_and_add ( const char * name, struct kobject
  * parent);
```

Where,

<name> – the name for the kobject

<parent> – the parent kobject of this kobject, if any.

If you pass kernel_kobj to the second argument, it will create the directory under /sys/kernel/. If you pass firmware_kobjto the second argument, it will create the directory under /sys/firmware/. If you pass fs_kobjto the second argument, it will create the directory under /sys/fs/. If you pass NULL to the second argument, it will create the directory under /sys/.

This function creates a kobject structure dynamically and registers it with sysfs. If the kobject was not able to be created, NULL will be returned.

When you are finished with this structure, call kobject_put and the structure will be dynamically freed when it is no longer being used.

```
struct kobject *kobj_ref;
/*Creating a directory in /sys/kernel/ */
kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj); //sys/kernel/etx_sysfs

/*Freeing Kobj*/
kobject_put(kobj_ref);
```

Create Sysfs file

Using above function we will create directory in /sys. Now we need to create sysfs file, which is used to interact user space with kernel space through sysfs. So we can create the sysfs file using sysfs attributes.

Attributes are represented as regular files in sysfs with one value per file. There are loads of helper function that can be used to create the kobject attributes.They can be found in header file sysfs.h

Create attribute

Kobj_attribute is defined as,

```
struct kobj_attribute {
    struct attribute attr;
    ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr, char *buf);
    ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t
count);
};
```

Where,

attr – the attribute representing the file to be created,

show – the pointer to the function that will be called when the file is read in *sysfs*,

store – the pointer to the function which will be called when the file is written in *sysfs*.

We can create attribute using __ATTR macro.

__ATTR(name, permission, show_ptr, store_ptr);

Store and Show functions

Then we need to write show and store functions.

```
ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr, char *buf);
ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t
count);
```

Store function will be called whenever we are writing something to the sysfs attribute. See the example.

Show function will be called whenever we are reading sysfs attribute. See the example.

Create sysfs file

To create a single file attribute we are going to use ‘sysfs_create_file’.

```
int sysfs_create_file ( struct kobject *  kobj, const struct attribute * attr);
```

```
1 int sysfs_create_file ( struct kobject *  kobj, const struct attribute
  * attr);
```

Where,

kobj – object we're creating for.

attr – attribute descriptor.

One can use another function ' sysfs_create_group ' to create a group of attributes.

Once you have done with sysfs file, you should delete this file using sysfs_remove_file

Interrupts in Linux kernel

This tutorial discusses interrupts and how the kernel responds to them, with special functions called interrupt handlers (ISR).

Interrupts

Here are some analogies to everyday life, suitable even for the computer-illiterate. Suppose you knew one or more guests could be arriving at the door. Polling would be like going to the door often to see if someone was there yet continuously.

That's what the doorbell is for. The guests are coming, but you have preparations to make, or maybe something unrelated that you need to do. You only go to the door when the doorbell rings. When doorbell rings, it's time to check the door again. You get more done, and they get quicker response when they ring the doorbell. This is interrupt mechanism.

Another scenario is, Imagine that you are watching TV or doing something. Suddenly you heard someone's voice which is like your Crush's voice. What will happen next? That's it, you are interrupted!! You will be very happy. Then stop your work whatever you are doing now and go outside to see him/her.

Similar to us, Linux also stops his current work and distracted because of interrupts and then it will handle them.

In Linux, interrupt signals are the distraction which diverts processor to a new activity outside from normal flow of execution. This new activity is called interrupt handler or interrupt service routine (ISR) and that distraction is Interrupts.

Polling vs Interrupts

Polling	Interrupt
In polling the CPU keeps on checking all the hardwares of the availablilty of any request	In interrupt the CPU takes care of the hardware only when the hardware requests for some service
The polling method is like a salesperson. The salesman goes from door to door while requesting to buy a product or service. Similarly, the controller keeps monitoring the flags or	An interrupt is like a shopkeeper. If one needs a service or product, he goes to him and apprises him of his needs. In case of interrupts, when the flags or signals are

Polling	Interrupt
signals one by one for all devices and provides service to whichever component that needs its service.	received, they notify the controller that they need to be serviced.

What will happen when interrupt came?

An interrupt is produced by electronic signals from hardware devices and directed into input pins on an interrupt controller (a simple chip that multiplexes multiple interrupt lines into a single line to the processor). These are the process that will be done by kernel.

1. Upon receiving an interrupt, the interrupt controller sends a signal to the processor.
2. The processor detects this signal and interrupts its current execution to handle the interrupt.
3. The processor can then notify the operating system that an interrupt has occurred, and the operating system can handle the interrupt appropriately.

Different devices are associated with different interrupts using a unique value associated with each interrupt. This enables the operating system to differentiate between interrupts and to know which hardware device caused which interrupt. In turn, the operating system can service each interrupt with its corresponding handler.

Interrupt handling is amongst the most sensitive tasks performed by kernel and it must satisfy following:

1. Hardware devices generate interrupts asynchronously (with respect to the processor clock). That means interrupts can come anytime.
2. Because interrupts can come anytime, the kernel might be handling one of them while another one (of a different type) occurs.
3. Some critical regions exist inside the kernel code where interrupts must be disabled. Such critical regions must be limited as much as possible.

Interrupts and Exceptions

Exceptions are often discussed at the same time as interrupts. Unlike interrupts, exceptions occur synchronously with respect to the processor clock; they are often called synchronous interrupts. Exceptions are produced by the processor while executing instructions either in response to a programming error (e.g. divide by zero) or abnormal conditions that must be handled by the kernel (e.g. a page fault). Because many processor architectures handle exceptions in a similar manner to interrupts, the kernel infrastructure for handling the two is similar.

Simple definitions of the two:

Interrupts: asynchronous interrupts generated by hardware.

Exceptions: synchronous interrupts generated by the processor.

System calls (one type of exception) on the x86 architecture are implemented by the issuance of a software interrupt, which traps into the kernel and causes execution of a special system call handler. Interrupts work in a similar way, except hardware (not software) issues interrupts.

There is a further classification of interrupts and exceptions.

Interrupts

Maskable – All Interrupt Requests (IRQs) issued by I/O devices give rise to maskable interrupts. A maskable interrupt can be in two states: masked or unmasked; a masked interrupt is ignored by the control unit as long as it remains masked.

Nonmaskable – Only a few critical events (such as hardware failures) give rise to nonmaskable interrupts. Nonmaskable interrupts are always recognized by the CPU.

Exceptions

Falts – Like Divide by zero, Page Fault, Segmentation Fault.

Traps – Reported immediately following the execution of the trapping instruction. Like Breakpoints.

Aborts – Aborts are used to report severe errors, such as hardware failures and invalid or inconsistent values in system tables.

For a device's each interrupt, its device driver must register an interrupt handler.

Interrupt handler

An interrupt handler or interrupt service routine (ISR) is the function that the kernel runs in response to a specific interrupt:

1. Each device that generates interrupts has an associated interrupt handler.
2. The interrupt handler for a device is part of the device's driver (the kernel code that manages the device).

In Linux, interrupt handlers are normal C functions, which match a specific prototype and thus enables the kernel to pass the handler information in a standard way. What differentiates interrupt handlers from other kernel functions is that the kernel invokes them in response to interrupts and that they run in a special context called interrupt context. This special context is occasionally called atomic context because code executing in this context is unable to block.

Because an interrupt can occur at any time, an interrupt handler can be executed at any time. It is imperative that the handler runs quickly, to resume execution of the interrupted code as soon as possible. It is important that

1. To the hardware: the operating system services the interrupt without delay.
2. To the rest of the system: the interrupt handler executes in as short a period as possible.

At the very least, an interrupt handler's job is to acknowledge the interrupt's receipt to the hardware. However, interrupt handlers can often have a large amount of work to perform.

Process Context and Interrupt Context

The kernel accomplishes useful work using a combination of process contexts and interrupt contexts. Kernel code that services system calls issued by user applications runs on behalf of the corresponding application processes and is said to execute in process context. Interrupt handlers, on the other hand, run asynchronously in interrupt context. Processes contexts are not tied to any interrupt context and vice versa.

Kernel code running in process context is preemptible. An interrupt context, however, always runs to completion and is not preemptible. Because of this, there are restrictions on what can be done from interrupt context. Code executing from interrupt context cannot do the following:

1. Go to sleep or relinquish the processor
2. Acquire a mutex
3. Perform time-consuming tasks
4. Access user space virtual memory

Based on our idea, ISR or Interrupt Handler should be execute very quickly and it should not run for more time (it should not perform time-consuming tasks). What if, I want to do huge amount of work upon receiving interrupt? So it is a problem right? If we do like that this will happen.

1. While ISR run, it doesn't let other interrupts to run (interrupts with higher priority will run).
2. Interrupts with same type will be missed.

To eliminate that problem, the processing of interrupts is split into two parts, or halves:

1. Top halves
2. Bottom halves

Top halves and Bottom halves

Top half

The interrupt handler is the top half. The top half will run immediately upon receipt of the interrupt and performs only the work that is time-critical, such as acknowledging receipt of the interrupt or resetting the hardware.

Bottom half

A bottom half is used to process data, letting the top half to deal with new incoming interrupts. Interrupts are enabled when a bottom half runs. Interrupt can be disabled if necessary, but generally this should be avoided as this goes against the basic purpose of having a bottom half – processing data while listening to new interrupts. The bottom half runs in the future, at a more convenient time, with all interrupts enabled.

For example using the network card:

- When network cards receive packets from the network, the network cards immediately issue an interrupt. This optimizes network throughput and latency and avoids timeouts.
- The kernel responds by executing the network card's registered interrupt.
- The interrupt runs, acknowledges the hardware, copies the new networking packets into main memory, and readies the network card for more packets. These jobs are the important, time-critical, and hardware-specific work.
    - The kernel generally needs to quickly copy the networking packet into main memory because the network data buffer on the networking card is fixed and miniscule in size, particularly compared to main memory. Delays in copying the packets can result in a buffer overrun, with incoming packets overwhelming the networking card's buffer and thus packets being dropped.
    - After the networking data is safely in the main memory, the interrupt's job is done, and it can return control of the system to whatever code was interrupted when the interrupt was generated.
- The rest of the processing and handling of the packets occurs later, in the bottom half.

If the interrupt handler function could process and acknowledge interrupts within few microseconds consistently, then absolutely there is no need for top half/bottom half delegation.

There are 4 bottom half mechanisms are available in Linux:

1. Work-queue
2. Threaded IRQs
3. Softirqs
4. Tasklets

Bottom Half

When Interrupt triggers, Interrupt Handler should be execute very quickly and it should not run for more time (it should not perform time-consuming tasks). If we have the interrupt handler which is doing more tasks then we need to divide into two halves.

1. Top Half
2. Bottom Half

Top Half is nothing but our interrupt handler. If our interrupt handler is doing less task, then top half is more than enough. No need of bottom half in that situation. But if our we have more work when interrupt hits, then we need bottom half. The bottom half runs in the future, at a more convenient time, with all interrupts enabled. So, The job of bottom halves is to perform any interrupt-related work not performed by the interrupt handler.

There are 4 bottom half mechanisms are available in Linux:

1. Work-queue
2. Threaded IRQs
3. Softirqs
4. Tasklets

In this tutorial, we will see Workqueue in Linux Kernel.

Workqueue in Linux Kernel

Work queues are added in linux kernel 2.6 version. Work queues are a different form of deferring work. Work queues defer work into a kernel thread; this bottom half always runs in process context. Because, Work queue is allowing users to create a kernel thread and bind work to the kernel thread. So, this will run in process context and the work queue can sleep.

- Code deferred to a work queue has all the usual benefits of process context.
- Most importantly, work queues are schedulable and can therefore sleep.

Normally, it is easy to decide between using work queues and softirqs/tasklets:

- If the deferred work needs to sleep, work queues are used.
- If the deferred work need not sleep, softirqs or tasklets are used.

There are two ways to implement Workqueue in Linux kernel.

1. Using global workqueue
2. Creating Own workqueue (We will see in next tutorial)

Using Global Workqueue (Global Worker Thread)

In this tutorial we will focus on this method.

In this method no need to create any workqueue or worker thread. So in this method we only need to initialize work. We can initialize the work using two methods.

- Static method
- Dynamic method (We will see in next tutorial)

Initialize work using Static Method

The below call creates a workqueue by the name and the function that we are passing in the second argument gets scheduled in the queue.

DECLARE_WORK(name, void (*func)(void *))

Where,

name: The name of the “work_struct” structure that has to be created.
func: The function to be scheduled in this workqueue.

Example

```
DECLARE_WORK(workqueue,workqueue_fn);
```

```
1 DECLARE_WORK(workqueue,workqueue_
  fn);
```

Schedule work to the Workqueue

These below functions used to allocate the work to the queue.

Schedule_work

This function puts a job in the kernel-global workqueue if it was not already queued and leaves it in the same position on the kernel-global workqueue otherwise.

int schedule_work(struct work_struct *work);

where,

work – job to be done

Returns zero if *work* was already on the kernel-global workqueue and non-zero otherwise.

Scheduled_delayed_work

After waiting for a given time this function puts a job in the kernel-global workqueue.

int scheduled_delayed_work(struct delayed_work *dwork, unsigned long delay);

where,

dwork – job to be done

delay – number of jiffies to wait or 0 for immediate execution

Schedule_work_on

This puts a job on a specific cpu.

```
int schedule_work_on( int cpu, struct work_struct *work );
```

where,

cpu– cpu to put the work task on

work– job to be done

Scheduled_delayed_work_on

After waiting for a given time this puts a job in the kernel-global workqueue on the specified CPU.

```
int scheduled_delayed_work_on(
 int cpu, struct delayed_work *dwork, unsigned long delay );
```

where,

cpu – cpu to put the work task on

dwork – job to be done

delay – number of jiffies to wait or 0 for immediate execution

Delete work from workqueue

There are also a number of helper functions that you can use to flush or cancel work on work queues. To flush a particular work item and block until the work is complete, you can make a call to flush_work. All work on a given work queue can be completed using a call to . In both cases, the caller blocks until the operation is complete. To flush the kernel-global work queue, call flush_scheduled_work.

```
int flush_work( struct work_struct *work );
void flush_scheduled_work( void );
```

Cancel Work from wprkqueue

You can cancel work if it is not already executing in a handler. A call to cancel_work_sync will terminate the work in the queue or block until the callback has finished (if the work is already in progress in the handler). If the work is delayed, you can use a call to cancel_delayed_work_sync.

```
int cancel_work_sync( struct work_struct *work );
int cancel_delayed_work_sync( struct delayed_work *dwork );
```

Check workqueue

Finally, you can find out whether a work item is pending (not yet executed by the handler) with a call to work_pending ordelayed_work_pending.

work_pending(work);
delayed_work_pending(work);

Programming

Driver Source Code

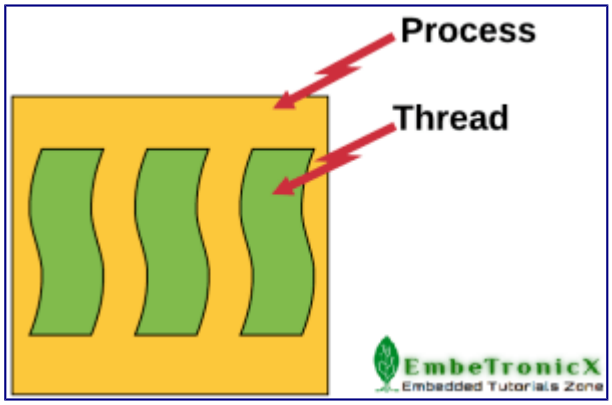
Kernel Thread

Process

An executing instance of a program is called a process. Some operating systems use the term ‘task‘ to refer to a program that is being executed. ***Process*** is a heavy weight process. Context switch between the process is time consuming.

Threads

A *thread* is an independent flow of control that operates within the same address space as other independent flows of control within a process.



One process can have multiple threads, with each thread executing different code concurrently, while sharing data and synchronizing much more easily than cooperating processes. Threads require fewer system resources than processes, and can start more quickly. Threads, also known as light weight processes.

Some of the advantages of the thread, is that since all the threads within the processes share the same address space, the communication between the threads is far easier and less time consuming as compared to processes. This approach has one disadvantage also. It leads to several concurrency issues and require the synchronization mechanisms to handle the same.

Thread Management

Whenever we are creating thread, it has to mange by someone. So that management follows like below.

- A thread is a sequence of instructions.
- CPU can handle one instruction at a time.
- To switch between instructions on parallel threads, execution state need to be saved.
- Execution state in its simplest form is a program counter and CPU registers.
- Program counter tells us what instruction to execute next.
- CPU registers hold execution arguments for example addition operands.
- This alternation between threads requires management.
- Management includes saving state, restoring state, deciding what thread to pick next.

Types of Thread

There are two types of thread.

1. User Level Thread
2. Kernel Level Thread

User Level Thread

In this type, kernel is not aware of these threads. Everything is maintained by the user thread library. That thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. So all will be in User Space.

Kernel Level Thread

Kernel level threads are managed by the OS, therefore, thread operations are implemented in the kernel code. There is no thread management code in the application area.

Anyhow each types of the threads have advantages and disadvantages too.

Now we will move into Kernel Thread Programming. First we will see the functions used in kernel thread.

Kernel Thread Management Functions

There are many functions used in Kernel Thread. We will see one by one. We can classify those functions based on functionalities.

- Create Kernel Thread
- Start Kernel Thread
- Stop Kernel Thread
- Other functions in Kernel Thread

For use the below functions you should include linux/kthread.h header file.

Create Kernel Thread

kthread_create

create a kthread.

struct task_struct * kthread_create (int (* threadfn(void *data),

void *data, const char namefmt[], ...);

Where,

threadfn – the function to run until signal_pending(current).

data – data ptr for threadfn.

namefmt[] – printf-style name for the thread.

... – variable arguments

This helper function creates and names a kernel thread. But we need to wake up that thread manually. When woken, the thread will run threadfn() with data as its argument.

threadfn can either call do_exit directly if it is a standalone thread for which noone will call kthread_stop, or return when ‘kthread_should_stop‘ is true (which meanskthread_stop has been called). The return value should be zero or a negative error number; it will be passed to kthread_stop.

It **Returns**

Start Kernel Thread

wake_up_process

This is used to Wake up a specific process.

int wake_up_process (struct task_struct * p);

Where,

p – The process to be woken up.

Attempt to wake up the nominated process and move it to the set of runnable processes.

It **returns 1** if the process was woken up, **0** if it was already running.

It may be assumed that this function implies a write memory barrier before changing the task state if and only if any tasks are woken up.

Stop Kernel Thread

kthread_stop

It stops a thread created by kthread_create.

int kthread_stop (struct task_struct *k);

Where,

k – thread created by kthread_create.

Sets kthread_should_stop for k to return true, wakes it, and waits for it to exit. Your threadfn must not call do_exit itself if you use this function! This can also be called after kthread_create instead of calling wake_up_process: the thread will exit without calling threadfn.

It **Returns** the result of threadfn, or –**EINTR** if wake_up_process was never called.

Other functions in Kernel Thread

kthread_should_stop

should this kthread return now?

int kthread_should_stop (void);

When someone calls kthread_stop on your kthread, it will be woken and this will return true. You should then return, and your return value will be passed through to kthread_stop.

kthread_bind

This is used to bind a just-created kthread to a cpu.

void kthread_bind (struct task_struct *k, unsigned int cpu);

Where,

k – thread created by kthread_create.

cpu – cpu (might not be online, must be possible) for k to run on.

Implementation

Thread Function

First we have to create our thread which has the argument of void * and should return int value. We should follow some conditions in our thread function. Its advisable.

- If that thread is a long run thread, we need to check kthread_should_stop() every time as because any function may call kthread_stop. If any function called kthread_stop, that time kthread_should_stop will return true. We have to exit our thread function if true value been returned by kthread_should_stop.
- But if your thread function is not running long, then let that thread finish its task and kill itself using do_exit.

In my thread function, lets print something every minute and it is continuous process. So lets check the kthread_should_stop every time. See the below snippet to understand.

```
int thread_function(void *pv) { int i=0; while(!kthread_should_stop())
{ printk(KERN_INFO "In EmbeTronicX Thread Function %d\n", i++); msleep(1000); }
return 0; }
```

```
   int thread_function(void
   *pv)
1 {
2      int i=0;
3      while(!
4 kthread_should_stop()) {
5          printk(KERN_INFO
6 "In EmbeTronicX Thread
7 Function %d\n", i++);
8          msleep(1000);
9      }
       return 0;
   }
```

Creating and Starting Kernel Thread

So as of now, we have our thread function to run. Now, we will create kernel thread using kthread_create and start the kernel thread using wake_up_process.

```
static struct task_struct *etx_thread; etx_thread =
kthread_create(thread_function,NULL,"eTx Thread"); if(etx_thread)
{ wake_up_process(etx_thread); } else { printk(KERN_ERR "Cannot create kthread\n"); }
```

```
   static struct task_struct
   *etx_thread;
1
2 etx_thread =
3 kthread_create(thread_function,N
4 ULL,"eTx Thread");
5 if(etx_thread) {
6     wake_up_process(etx_thread);
7 } else {
8     printk(KERN_ERR "Cannot
   create kthread\n");
   }
```

There is another function which does both process (create and start). That is kthread_run(). You can replace the both kthread_create and wake_up_process using this function.

kthread_run

This is used to create and wake a thread.

kthread_run (threadfn, data, namefmt, ...);

Where,

threadfn – the function to run until signal_pending(current).

data – data ptr for threadfn.

namefmt – printf-style name for the thread.

... – variable arguments

Convenient wrapper for kthread_create followed by wake_up_process.

It **returns** the kthread or ERR_PTR(-ENOMEM).

You can see the below snippet which is using kthread_run.

```
static struct task_struct *etx_thread; etx_thread = kthread_run(thread_function,NULL,"eTx Thread"); if(etx_thread) { printk(KERN_ERR "Kthread Created Successfully...\n"); } else { printk(KERN_ERR "Cannot create kthread\n"); }
```

```
static struct task_struct
*etx_thread;
```

```
etx_thread =
kthread_run(thread_function,NU
LL,"eTx Thread");
if(etx_thread) {
    printk(KERN_ERR "Kthread
Created Successfully...\n");
} else {
    printk(KERN_ERR "Cannot
create kthread\n");
}
```

Stop Kernel Thread

You can stop the kernel thread using kthread_stop. Use the below snippet to stop.

```
kthread_stop(etx_thread);
```

```
kthread_stop(etx_thread
);
```

Mutex in Linux Kernel

Introduction

Before getting to know about Mutex, let’s take an analogy first.

Let us assume we have a car designed to accommodate only one person at any instance of time, while all the four doors of the car are open. But, if any more than one person tries to enter the car, a bomb will set off an explosion!(Quite a fancy car manufactured by EmbeTronicX!!) Now that four doors of the car are open, the car is vulnerable to explosion as anyone can enter through one of the four doors.

Now how we can solve this issue? Yes correct. We can provide a key for the car. So, the person who wants to enter the car must have access to the key. If they don’t have key, they have to wait until that key is available or they can do some other work instead of waiting for key.

Example Problems

Let’s correlate the analogy above to what happens in our software. Let’s explore situations like these through examples.

1. You have one SPI connection. What if one thread wants to write something into that SPI device and another thread wants to read from that SPI device at the same time?
2. You have one LED display. What if one thread is writing data at a different position of Display and another thread is writing different data at a different position of Display at the same time?
3. You have one Linked List. What if one thread wants to insert something into the list and another one wants to delete something in the same Linked List at the same time?

In all the scenarios above, the problem encountered is the same. At any given point two threads are accessing a single resource. Now we will relate the above scenarios to our car example.

1. In SPI example, CAR = SPI, Person = Threads, Blast = Software Crash/Software may get wrong data.
2. In LED display example, CAR = LED Display, Person = Threads, Blast = Display will show some unwanted junks.
3. In Linked List example, CAR = Linked List, Person = Threads, Blast = Software Crash/Software may get wrong data.

The cases above are termed as Race Condition.

Race Condition

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, we don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

To avoid the race conditions, we have many ways like Semaphore, Spinlock and Mutex. In this tutorial we will concentrate on Mutex.

Mutex

A *mutex* is a mutual exclusion lock. Only one thread can hold the lock.

Mutex can be used to prevent the simultaneous execution of a block of code by multiple threads that are running in a single or multiple processes.

Mutex is used as a synchronization primitive in situations where a resource has to be shared by multiple threads simultaneously.

Mutex has ownership. The thread which locks a Mutex must also unlock it.

So whenever you are accessing a shared resource that time first we lock mutex and then access the shared resource. When we are finished with that shared resource then we unlock the Mutex.

I hope you got some idea about Mutex. Now, let us look at Mutex in Linux Kernel.

Mutex in Linux Kernel

Today most major operating systems employ multitasking. Multitasking is where multiple threads can execute in parallel and thereby utilizing the CPU in optimum way. Even though, multitasking is useful, if not implemented cautiously can lead to concurrency issues (Race condition), which can be very difficult to handle.

The actual mutex type (minus debugging fields) is quite simple:

struct mutex { atomic_t count; spinlock_t wait_lock; struct list_head wait_list; };

```
struct mutex
{
   atomic_t
1 count;
2  spinlock_t
3  wait_lock;
4  struct
5 list_head   w
ait_list;
};
```

We will be using this structure for Mutex. Refer to Linux/include/linux/mutex.h

Initializing Mutex

We can initialize Mutex in two ways

1. Static Method
2. Dynamic Method

Static Method

This method will be useful while using global Mutex. This macro is defined below.

DEFINE_MUTEX(name);

This call *defines* and *initializes* a mutex. Refer to Linux/include/linux/mutex.h

Dynamic Method

This method will be useful for per-object mutexes, when mutex is just a field in a heap-allocated object. This macro is defined below.

mutex_init(struct mutex *lock);

Argument:

struct mutex *lock – the mutex to be initialized.

This call *initializes* already allocated mutex. Initialize the mutex to unlocked state.

It is not allowed to initialize an already locked mutex.

Example

struct mutex etx_mutex; mutex_init(&etx_mutex);

```
   struct mutex
1  etx_mutex;
2  mutex_init(&etx_mute
   x);
```

Mutex Lock

Once a mutex has been initialized, it can be locked by any one of them explained below.

mutex_lock

This is used to lock/acquire the mutex exclusively for the current task. If the mutex is not available, the current task will sleep until it acquires the Mutex.

The mutex must later on be released by the same task that acquired it. Recursive locking is not allowed. The task may not exit without first unlocking the mutex. Also, kernel memory where the mutex resides must not be freed with the mutex still locked. The mutex must first be initialized (or statically defined) before it can be locked. memset-ing the mutex to 0 is not allowed.

void mutex_lock(struct mutex *lock);

Argument:

struct mutex *lock – the mutex to be acquired

mutex_lock_interruptible

Locks the mutex like mutex_lock, and returns 0 if the mutex has been acquired or sleep until the mutex becomes available. If a signal arrives while waiting for the lock then this function returns -EINTR.

int mutex_lock_interruptible(struct mutex *lock);

Argument:

struct mutex *lock – the mutex to be acquired

mutex_trylock

This will try to acquire the mutex, without waiting (will attempt to obtain the lock, but will not sleep). Returns 1 if the mutex has been acquired successfully, and 0 on contention.

```
int mutex_trylock(struct mutex *lock);
```

Argument:

struct mutex *lock – the mutex to be acquired

This function must not be used in interrupt context. The mutex must be released by the same task that acquired it.

Mutex Unlock

This is used to unlock/release a mutex that has been locked by a task previously.

This function must not be used in interrupt context. Unlocking of a not locked mutex is not allowed.

```
void mutex_unlock(struct mutex *lock);
```

Argument:

struct mutex *lock – the mutex to be released

Mutex Status

This function is used to check whether mutex has been locked or not.

```
int mutex_is_locked(struct mutex *lock);
```

Argument:

struct mutex *lock – the mutex to check the status.

Returns 1 if the mutex is locked, 0 if unlocked.

Example Programming

This code snippet explains how to create two threads that accesses a global variable (etx_gloabl_variable). So before accessing the variable, it should lock the mutex. After that it will release the mutex.