CHARACTER DRIVER

[Document subtitle]

MOSCHIP SEMICONDUCTORS

INDEX

Introduction

Char driver

- Char devices are accessed through names in filesystem. Those names are called special files or device files.
- They are conventionally located in the /dev directory. Special files for character drivers are identified by a ‘c’.
- We use ls -l command to see the information about the character drivers.
- We can also see major number and minor number by using above command

Major and Minor Number

- both major number and minor number are the parts of the device number of type dev_t defined in <linux/types.h> header file
- this device number is 32 bit value 12 bits assigned for major number and 20 bit for minor number
- there are predefined macros in <linux/kdev_t.h> header to find the major number and minor number.
- we need to allocate major number for every driver so we can do this in statically and dynamically
- for statical allocation we use functions defined in <linux/fs.h> like

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

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

- to unregister these, we use function like

  void unregister_chrdev_region(dev_t first,unsigned int count);

- for dynamic allocation of major numbers its strongly suggested to use dynamic allocation to obtain your major number certainly be using alloc_chrdev_region rather than register_chrdev_region

Character device registration

- To do so we need to include <linux/cdev.h> where the structure is associated.
- There are two ways of allocating and initializing one of these structures.

  Void cdev_intit(struct cdev *cdev,struct file_operations *fops);

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

  Void cdev_del(struct cdev * dev);

- He above are the registration calls to innitilize char dev to add the character device and to delete the character device.

Printk

- Printk() is a kernel-level function,which has the ability to print out the different log levels as defined in.we can see the prints using dmesg command.
- Syntax:printk(<macro>"information");

- pr_info -> it will print in information loglevel as printk(KERN_INFO"data");
- pr_cont ->It will continues a previous log message in the same line.
- pr_debug -> Print a debug-level message as printk(KERN_DEBUG"data");
- pr_err -> print an error level message
- pr_warn -> print a warning level message

Kernel Logs

- Kern_emerg:0
- Used for emergency messages usually those that precede a crash.
- Kern_Alert:1
- A situation requiring immediate action.
- Kern_crit:2
- Critical conditions are often related to serious hardware or software issues.
- Kern_err:3
- Used to report error conditions; device drivers often use Kern_err to report hardware difficulties.
- Kern_Warning:4
- Warning about problematic situations that do not, in themselves,create serious problems with the system.
- Kern_Notice:5
- Situations that are normal,but still worthy of note.a number of security-related conditions are reported at this level
- Kern_info:6
- Informational messages many drivers print information about the hardware they find at startup time at this level.
- Kern_debug:7
- Used for debugging messages

Differences between printk and printf

Printk

- Printk() is kernel level function.
- Printk() is not a standard liabrary function.

Printf

- Printf() will always print to an STDOUT file descriptor
- Printf() is standard liabrary function

Basic Driver Programming

```
#include<linux/kernel.h> //this hedrfile for kernelinformation
#include<linux/init.h> //for this hederfile for intilisation and freeup memory
#include<linux/module.h> //this hederfile author and licence
/*module intiliasation function*/
static int __init hello_world_init(void)
{
   printk(KERN_INFO "Welcome to kernel\n");
   printk(KERN_INFO "This is the Simple Module\n");
   printk(KERN_INFO "Kernel Module Inserted Successfully...\n");
   return 0;
}
/*exit function*/
static void __exit hello_world_exit(void)
{
  printk(KERN_INFO "Kernel Module Removed Successfully...\n");
}
module_init(hello_world_init);
module_exit(hello_world_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR(“/<pavan>");
MODULE_DESCRIPTION("A simple hello world driver");
```

Program for statically allocating major number

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

#include<linux/init.h>

#include<linux/module.h>

#include <linux/fs.h> //creating the dev with our custom major and minor number

dev_t dev = MKDEV(42, 0);  //Module Init function

static int __init hello_world_init(void)

{

 register_chrdev_region(dev, 1, “char_Dev")

   printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

printk(KERN_INFO "Kernel Module Inserted Successfully...\n");

  return 0;

}

// Module exit function

static 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(“module author");

MODULE_DESCRIPTION("Simple linux driver (Statically allocating the Major and Minor number)");
```

Program for dynamically allocating major number

```
#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)        //Module Init function

{   /*Allocating Major number*/

if((alloc_chrdev_region(&dev, 0, 1, "char_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;

}

static void __exit hello_world_exit(void)      //Module exit function

{

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("<pavan>");

MODULE_DESCRIPTION("Simple linux driver (Dynamically allocating the Major and Minor number)");
```

What are device Files

- Device files are files allows an application program to interact with a device via standard input/output system calls.
- Using standard system calls simplifies many programming tasks and leads to consistent user space I/O mechanism regardless of device feature and functions.
- There are two general kinds of device files in unix like operating systems known are character special files or block special files.
- They are not normal files but looks like files from program point of view you can read from them, write to them
- All device files are stored in /dev directory use **ls** command to browse the directory

Types of Device file creation

1.Manually creating device file
2.Automatically creating device file

Manually Creating Device file

- We can create the device file manually by using **mknod**
- **->mknod –m <permissions> <name> <device type> <major> <minor>**
- **<name> ->** your device file name that should have a full path (/dev/name)
- **<device type> ->** put c-character device or 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

Disadvantages:

- You need to check whether there is device file present or not

Dynamic device file creation

- The Linux kernel can create the device files dynamically after device driver model the kernel 2.6 offers a very simple way to do so.
- A set of user mode programs, collectively known as the udev toolset, must be installed in the system.
- The automatic creation of device file can be handled with udev

Header files required

- #include<linux/device.h>
- #include<linux/kdev_t.h>

Driver code for Dynamic Device file Creation

```
#include<linux/init.h>
#include<linux/fs.h>
#include<linux/module.h>
#include<linux/kernel.h>
#include<linux/device.h>
#include<linux/kdev_t.h>
#include<linux/err.h>
MODULE_LICENSE("GPL");
static struct class *dev_class;
static struct device *device;
dev_t dev=0;
/******module init function***************/
static int __init _init(void)
{
        /**********allocating major number******/
        if((alloc_chrdev_region(&dev,0,1,"device"))<0)
        {
                pr_err("failed to create major number for the device\n");
                return -1;
        }
        pr_info("MAJOR NO =%d MINOR NO =%d \n",MAJOR(dev),MINOR(dev));
/************creation of struct class**************/
        dev_class=class_create(THIS_MODULE,"class");
if(IS_ERR(dev_class)){
                pr_err("failed to create the struct class for device\n");
                class_destroy(dev_class);
        }
        device=device_create(dev_class,NULL,dev,NULL,"device");
        if(IS_ERR(device)){
                pr_err("failed to create the device files\n");
                unregister_chrdev_region(dev,1);
                return -1;
        }
        pr_info("module is inserted successfully.....\n");
        return 0;
}
static void __exit _exit(void){         //module exit function
        device_destroy(dev_class,dev);
        class_destroy(dev_class);
        unregister_chrdev_region(dev,1);
        pr_info("module is removed successfully .....\n");}
module_init(_init);
module_exit(_exit);
```

File Operations

- file_operations is a pre-defined structure which is present in **linux/fs.h**.
- It is the collection of function pointers.
- A file_operations structure is called **fops**.
- The whole structure of file_operations is mentioned below,

```
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);
};
```

- The file_operations base address is one of the member in **struct cdev**. And this struct cdev base address is one of the member in **struct inode**.
- Kernel uses this file_operations structure to access the functions written in device driver code such as open, read, release(close), write etc. The base address of these functions are assigned to the members of struct file_operations.

For example:

```
struct file_operations my_routines =
{
.open = mydriver_open,
.release = mydriver_close,
.read = mydriver_read,
.write = mydriver_write
};
```

- Basically we concentrate on some basic functions which are mentioned below,

1. open

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

- This is always the first operation performed on the device file.
- In this function we pass two arguments one is the base address of the struct inode i.e inode object's base address and second is the file object base address i.e. struct file base address.

- This inode object is created when we create the device file. When we use open() in our application then a file object is created along with an entry in the fd table nothing but file descriptor table.
- Based on those both inode object base address and file object base address open function in the device driver code is invoked.
- For example: mydriver_open() function is invoked in our device driver code when we use the open() system call in our application.

2. release(close)

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

- Even in the release function we pass two arguments. First one is the inode object's base address and the second argument is the file object's base address.
- This release function is invoked when close() system call is called in our application code.

- For example: mydriver_close() function is invoked in our device driver code when we use the close() system call in our application.

3. read

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

- The return type is ssize_t. In this function we pass four arguments. First one is the file object's base address, second argument is the buffer address, third argument is the size of the buffer and the fourth argument is offset value.
- This function is used to retrieve the data from the device.
- A null pointer causes the read system call to fail with -EINVAL ("Invalid Argument").
- A non-negative return value represents the number of bytes successfully read.
- This read function is invoked when read() system call is called in our application code.
- For example: mydriver_read() function is invoked in our device driver code when we use the read() system call in our application.

4. write

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

- The return type is ssize_t. Even here we pass four arguments in this function as same as the read function. But the second argument we pass in this function is the constant buffer address. First argument is the file object base address, second argument is the constant buffer address, third argument is the size of the buffer and fourth argument is offset value.
- Here the second argument is constant buffer because the written buffer string which is passed by the application should not be changed.
- It is used to send the data to the device.
- If the return value is non-negative it represents the number of bytes successfully written.
- This write function is invoked when write() system call is called in our application code.
- For example: mydriver_write() function is invoked in our device driver code when we use the write() system call in our application.

5. ioctl

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

- Return type is int.
- Here we pass four arguments. First one is the inode object base address, second one is the file object base address, third one is the unsigned integer value and the fourth one is the unsigned long int value.
- ioctl is nothing but the input-output control. The detailed information about ioctl is present under the heading of IOCTL.

CHARACTER DRIVER CODE

```
#include<linux/kernel.h>
#include<linux/module.h>              //header file for module information
#include<linux/init.h>
#include<linux/poll.h>
#include<linux/ioport.h>
#include<linux/errno.h>
#include<linux/cdev.h>
#include<linux/fs.h>                  //header file for struct file operations
MODULE_AUTHOR(“author name"); //this MACRO is used to know which
person have wrote the DD code
MODULE_LICENSE("GPL");            /*this MACRO is used to give liscence
ident to our code
```

- *There are many license idents which are currently accepted by indicating free software modules, GPL is one of the license ident.*
- *Your module really should specify which license applies to it’s code.*/*

```
static int mychar_open(struct inode *inode,struct file *file);   //function
declarations
static int mychar_close(struct inode *inode,struct file *file);
static ssize_t mychar_write(struct file *file,const char *buff,size_t len,loff_t *offset);
static ssize_t mychar_read(struct file *file,char *buff,size_t len,loff_t *offset);
//---------------------------------structure defined by the user-------------------------
struct d_context
{
char buf[5];                    //character buffer of 5 bytes, where it is used as the kernel buffer.
    int no_chars;                       //number of characters read or written
    struct cdev c_dev;                  //variable of type struct cdev
}my_context;                            //variable name of the structure type struct d_context
//---------------------------structure type of struct file_operations----------------------
struct file_operations my_routines=
{       .read=mychar_read,
.open=mychar_open,
.write=mychar_write,
.release=mychar_close  };           /*storing the base addresses of the functions which we are using
in this driver code to the particular member of the structure type struct file_operations defined for
that function type*/
//---------------------------------character driver initialization-----------------------
static int mydev_init(void)   //this function is executed when the linux DD is
loaded into the kernel using insmod
{
   int res=0;                   //taking this res variable of int type to just check the
return values
   dev_t devno=0;               //dev_t is the typedef of int in the kernel
   devno=MKDEV(42,0);   //with the help of Major and Minor numbers, MKDEV
generate device number.
```

```
   res=register_chrdev_region(devno,1,“char_device");   /*in this fn() the 3rd argument is the name of device that should be associated. To set a particular Major number for your driver we use this method.*/
if(res<0)
 {
      printk("<1>"“char_device is not registered successfully\n");  /*here printk() works as printf(), we can see the messages written in Device Driver using printk() under dmesg command*/
      return res;
 }
 else
 {
      printk("<1>"“char_device is loaded successfully\n");
 }
/**************Driver registration with kernel is completed*************/
cdev_init(&my_context.c_dev,&my_routines);  /*here cdev structure gets initialized, as struct file_operations is one among the members of cdev structure, the base address of variable of type struct file_operations gets upadated in the pointer type of struct file_operations (one among the member of cdev structure)*/
my_context.c_dev.owner=THIS_MODULE;  //here we are using MACRO to update  the owner of this module is itself only
res=cdev_add(&my_context.c_dev,devno,1);  /*here the second argument is the number of corresponding minor numbers
if(res<0)
{
       printk("<1>""cdev object was not loaded successfully\n");
       unregister_chrdev_region(devno,1);
}
printk("<1>""cdev object loaded successfuly\n");
//if the cdev object is loaded successfully, all functions you defined through file_operators structure can be called.
/*************Now your device has been added to the system*************/
my_context.no_chars=0;
return res;
}
//----------------------------------character driver exit function-----------------------
static void mydev_release(void) //this function is executed when the linux DD is unloaded using rmmod
{

      dev_t devno=MKDEV(42,0); //with the help of Major and Minor numbers, MKDEV generate device number
      cdev_del(&my_context.c_dev); //removes the char device which we have created from the system
      unregister_chrdev_region(devno,1); //here the unregistration takes place of that particular Major Number
     printk("<1>""char_device  unloaded successfully\n");
}
//--------------------------------character driver open function-----------------------
```

```
/*1. load device file dynamically or statically in to kernel space..(# mknod /dev/mydev c 42 0)*/
/* i_rdev=devno, *i_cdev=&my_context.c_dev are the members of inode object of "/dev/mydev" are stored automatically when a request comes from application (open()) system call*/
static int mychar_open(struct inode *inode, struct file *file)   //1st arg= inode object base address, 2nd arg= base address of file object of "/dev/my_char"
{
    try_module_get(THIS_MODULE);  //it will increase the count of no of devices communicating to driver
    file->private_data=&my_context;  //storing the base address of device context info in file_object private data
    if(file->f_mode & FMODE_READ)  //FMODE_READ is a predefined macro-it has a unique identifier for read_mode
                printk("<1>""open for read mode\n");

    if(file->f_mode & FMODE_WRITE) //FMMODE_WRITE is a predefined macro- it has a unique identifier for write_mode
                printk("<1>""open for write mode\n");
    return 0;     //returning zero for successfull open
}
//--------------------------------character driver close function-----------------------
static int mychar_close(struct inode *inode, struct file *file)
{
    module_put(THIS_MODULE);  //it will decrease the count of no of devices communicating to driver
    printk("<1>""device file is closed\n");
    return 0;      // returning zero for successfull close
}
//---------------------------------character driver write function----------------------
static ssize_t mychar_write(struct file *file, const char *buff, size_t len, loff_t *offset)
{
    struct d_context *tdev;
    tdev=file->private_data;  //storing the address of device context info of particular device file into a device context pointer
    if(len>5)
          len=5;
    copy_from_user(tdev->buf,buff,len);  //1st arg is the destination address of device context info member buf,2nd arg is the source,3rd arg is len, here we are copying data from application to kernel
    tdev->no_chars=len;      //updating no of chars written to kernel device context info member-buf
    printk("<1>""write is called\n");
    pr_info("data received from app %s-%d (no of chars)\n",tdev->buf,tdev->no_chars);
    return (ssize_t)len;
}
//---------------------------------character driver read function----------------------
static ssize_t mychar_read(struct file *file, char *buff, size_t len, loff_t *offset)
{
    struct d_context *tdev;
    tdev=file->private_data;   //storing the address of device context info of particular device file into a
```

```
device context pointer
    if(len>tdev->no_chars)
        len=tdev->no_chars;
    copy_to_user(buff,tdev->buf,len);  //2nd arg is the destination address of device context info
member buf,1st arg is the source,3rd arg is len, here we are copying data from kernel to application
    tdev->no_chars=0;  //updating no of chars written to user device context info
member-buf
    printk("<1>""read is called\n");
    return (ssize_t)len;
}
module_init(mydev_init);    //module_init is predefined macros which hold the address of driver
initialisation routine
module_exit(mydev_release);    //module_exit is predefined macros which hold the address of driver
exit routine
```

MAKE FILE

```
obj-m =  basic_character_driver.o
all:
        make –C /lib/modules/$(shell uname –r)/build M=$(shell pwd) modules
clean:
        make –C /lib/modules/$(shell uname –r)/build M=$(shell pwd) clean
```

#In the above makefile, we are changing the directory to the path /lib/modules/$(shell uname –r)/build and loading the modules which are required for our device driver code in the present directory.

APPLICATIONS

Write application:

```
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
#include<unistd.h>
#include<sys/stat.h>
void main()
{
        int fd,ret;
        char buf[100];
        fd=open("/dev/my_char",O_WRONLY);
        if(fd<0)
        {
                printf("failed to establish the                                communication");
                exit(1);
        }
        printf("enter the input\n");
        scanf("%[^\n]s",buf);
        ret=write(fd,buf,strlen(buf));
```

```
if(ret<0)
        {
                printf("failed to erform write operation");
                exit(2);
        }
        printf("write return value ->%d\n",ret);
        close(fd);
}
```

Read application:

```
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
#include<unistd.h>
#include<sys/stat.h>
void main()
{
        int fd,ret=0;
        char buf[100];
        fd=open("/dev/my_char",O_RDONLY);
        if(fd<0)
        {
                printf("failed to establish the communication");
                exit(1);
        }
        ret=read(fd,buf,sizeof(buf));
        if(ret<0)
        {
                printf("failed to perform        write operation");
                exit(2);
        }
        printf("string ->%s\n",buf);
        close(fd);
}
```

Commands to be performed:

- $sudo make //loading the required modules
- $sudo insmod character_driver.ko //loading the driver code
- $sudo mknod /dev/my_char c 42 0 //creating the device file under /dev dir
- $gcc -o read.c read //executable file for read application
- $gcc -o write.c write //executable file for write application
- $sudo ./write //running ./write executable file
- $sudo ./read //running ./read executable file
- $sudo rmmod character_driver.ko //unloading the driver code

There are some **drawbacks** in this basic character driver code. Those are as follows:

1. If we perform the read application first, the function mychar_read is getting executed. It must not get executed until some data is present in the kernel buffer.
2. If we perform the write application multiple times without performing the read application, the data is over written in the write application. It must not allow to write the data until the previous data is read by the read application.
3. As in our code we have given the buffer size as 5, if we write the data more than 5 bytes, while the read application execution only 5 bytes of data is read.

To overcome these drawbacks blocking mechanism has been introduced. Those are nothing but **waitqueues**.

Waitqueue

Introduction

When you write a Linux Driver or Module or Kernel Program, Some processes should wait or sleep for some event. There are several ways of handling sleeping and waking up in Linux, each suited to different needs. Waitqueue is also one of the methods to handle that case.

Whenever a process must wait for an event (such as the arrival of data or the termination of a process), it should go to sleep. Sleeping causes the process to suspend execution, freeing the processor for other uses.

After some time, the process will be woken up and will continue with its job when the event which we are waiting for has arrived.

Wait queue is a mechanism provided in the kernel to implement the wait. As the name itself suggests, waitqueue is the list of processes waiting for an event. In other words, A wait queue is used to wait for someone to wake you up when a certain condition is true. They must be used carefully to ensure there is no race condition.

There are 3 important steps in Waitqueue:

1. Initializing Waitqueue

2. Queuing (Put the Task to sleep until the event comes)

3. Waking Up Queued Task

Initializing Waitqueue

Use this Header file for Waitqueue (include /linux/wait.h). There are two ways to initialize the waitqueue.

1. Static method

2. Dynamic method

Static Method

DECLARE_WAIT_QUEUE_HEAD(wq);

Where the “wq” is the name of the queue on which task will be put to sleep.

Dynamic Method

wait_queue_head_t wq;

init_waitqueue_head (&wq);

You can create a waitqueue using any one of the above methods. Other operations are common for both static and dynamic method except the way we create the waitqueue.

Queuing

Once the wait queue is declared and initialized, a process may use it to go to sleep. There are several macros are available for different uses. We will see each one by one.

1. wait_event

2. wait_event_timeout

3. wait_event_cmd

4. wait_event_interruptible

5. wait_event_interruptible_timeout

6. wait_event_killable

Old kernel versions used the functions sleep_on() and interruptible_sleep_on(), but those two functions can introduce bad race conditions and should not be used.

Whenever we use the above one of the macro, it will add that task to the waitqueue which is created by us. Then it will wait for the event.

wait_event sleep until a condition gets true.

wait_event(wq, condition);

wq – the waitqueue to wait oncondition – a C expression for the event to wait for the process is put to sleep (TASK_UNINTERRUPTIBLE) until the[condition]evaluates to true.

The[condition]is checked each time the waitqueue[wq]is woken up.

wait_event_timeout sleep until a condition gets true or a timeout elapses

wait_event_timeout(wq, condition, timeout);

wq – waitqueue to wait on condtion – a C expression for the event to wait for timeout – timeout, in jiffies the process is put to sleep (TASK_UNINTERRUPTIBLE) until the[condition]evaluates to true or timeout elapses. The condition is checked each time the waitqueue wq is woken up. It returns 0 if the condition evaluated to false after the timeout elapsed, 1 if the condition evaluated to true after the

timeout elapsed, or the remaining jiffies (at least 1) if the condition evaluated to true before the timeout elapsed.

Waitqueue in Linux

wait_event_cmd sleep until a condition gets true

wait_event_cmd(wq, condition, cmd1, cmd2);

wq – the waitqueue to wait on condtion – a C expression for the event to wait for cmd1 – the command will be executed before sleep

cmd2 – the command will be executed after sleep

The process is put to sleep (TASK_UNINTERRUPTIBLE) until the condition evaluates to true. The condition is checked each time the waitqueue wq is woken up.

wait_event_interruptible sleep until a condition gets true

wait_event_interruptible(wq, condition);

wq – the waitqueue to wait on condtion – a C expression for the event to wait for the process is put to sleep (TASK_INTERRUPTIBLE) until the condition evaluates to true or a signal is received. The condition is checked each time the waitqueue wq is woken up. The function will return -ERESTARTSYS if it was interrupted by a signal and 0 if condition evaluated to true.

wait_event_interruptible_timeout sleep until a condition gets true or a timeout elapses.

wait_event_interruptible_timeout(wq, condition, timeout);

wq – the waitqueue to wait on condtion – a C expression for the event to wait for timeout – timeout, in jiffies the process is put to sleep (TASK_INTERRUPTIBLE) until the condition evaluates to true or a signal is received or timeout elapsed. The condition is checked each time the waitqueue wq is woken up.

It returns, 0 if the condition evaluated to false after the timeout elapsed, 1 if the condition evaluated to true after the timeout elapsed, the remaining jiffies (at least 1) if the condition evaluated to true before the timeout elapsed, or -ERESTARTSYS if it was interrupted by a signal.

wait_event_killable sleep until a condition gets true

wait_event_killable(wq, condition);

wq – the waitqueue to wait on condtion – a C expression for the event to wait for the process is put to sleep (TASK_KILLABLE) until the condition evaluates to true or a signal is received. The condition is checked each time the waitqueue wq is woken up.

The function will return -ERESTARTSYS if it was interrupted by a signal and 0 if condition evaluated to true.

Waking Up Queued Task

When some Tasks are in sleep mode because of the waitqueue, then we can use the below function to wake up those tasks.

1. wake_up

2. wake_up_all

3. wake_up_interruptible

4. wake_up_sync and wake_up_interruptible_sync

wake_up wakes up only one process from the wait queue which is in non-interruptible sleep.

wake_up(&wq);

wq – the waitqueue to wake up

wake_up_all wakes up all the processes on the wait queue

wake_up_all(&wq);

wq – the waitqueue to wake up

wake_up_interruptible wakes up only one process from the wait queue that is in interruptible sleep

wake_up_interruptible(&wq);

wq – the waitqueue to wake up

wake_up_sync and wake_up_interruptible_sync

wake_up_sync(&wq);

wake_up_interruptible_sync(&wq);

Normally, a wake_up call can cause an immediate reschedule to happen, meaning that other processes might run before wake_up returns. The “synchronous” variants instead make any awakened processes runnable but do not reschedule the CPU. This is used to avoid rescheduling when the current process is known to be going to sleep, thus forcing a reschedule anyway. Note that awakened processes could run immediately on a different processor, so these functions should not be expected to provide mutual exclusion.

DRIVER CODE:

```
#include<linux/kernel.h>
#include<linux/module.h>
#include<linux/init.h>
#include<linux/poll.h>
#include<linux/ioport.h>
#include<linux/errno.h>
#include<linux/cdev.h>
#include<linux/fs.h>
#include<linux/wait.h>

  DECLARE_WAIT_QUEUE_HEAD(rq);
  DECLARE_WAIT_QUEUE_HEAD(wq);
MODULE_AUTHOR("ajay");
MODULE_LICENSE("GPL");
```

```
static int mychar_open(struct inode *inode,struct file *file);
static int mychar_close(struct inode *inode,struct file *file);
static ssize_t mychar_write(struct file *file,const char *buff,size_t len,loff_t *offset);
static ssize_t mychar_read(struct file *file,char *buff,size_t len,loff_t *offset);

struct d_context
{
    char buf[5];  //driver own
    int no_chars;
    int base_address; /* HW resources*/
    int IRQ_lineno;  /* HW resources*/
    struct cdev c_dev;

}my_context;
/*  file operations structure  */
/* storing the base address of the driveno_charsr routines in function pointers of structure of
file_operations  */
struct file_operations my_routines=
{
   .read=mychar_read,
   .open=mychar_open,
   .write=mychar_write,
   .release=mychar_close
};

/*****character driver initialisation ****************/
static int mydev_init(void)
{
   int res=0;
   dev_t devno=0; //typedef of int in kernal
   devno=MKDEV(42,0); //with the help of major and minor numbers MKDEV generates device
number
   res=register_chrdev_region(devno,1,"ajay_char");
   if(res<0)
   {
     printk("<1>""ajay_char not registered successfully\n");
     return res;
   }
   else
   {
     printk("<1>""ajay_char is loaded successfully\n");
   }
   /********driver registration with kernal was completed ***********/
   cdev_init(&my_context.c_dev,&my_routines); //linking the cdev object and file_opstatic ssize_t
mychar_read(struct file *file,char *buff,size_t len,loff_t *offset);erations
   my_context.c_dev.owner=THIS_MODULE;//Predefined macro static ssize_t mychar_write(struct
file *file,const char *buff,size_t len,loff_t *offset);used to return the base address of the driver

   res=cdev_add(&my_context.c_dev,devno,1);
   if(res<0)
   {
      printk("<1>""cdev object was not loaded successfully\n");
      unregister_chrdev_region(devno,1);

   }
```

```
      printk("<1>""cdev object loaded successfuly\n");
      my_context.no_chars=0;
      return res;

}
static void mydev_release(void)
{
      dev_t devno=MKDEV(42,0); //with the help of major and minor numbers MKDEV generates
device number
      cdev_del(&my_context.c_dev);
      unregister_chrdev_region(devno,1);
      printk("<1> ajay_char unloaded successfully\n");
}
/*****************************************************************************
*****************************/

/****************************** open driver routine
*************************************/
static int mychar_open(struct inode *inode,struct file *file)
{
   try_module_get(THIS_MODULE);//It will increase the count of no of devices communicating with
the driver
   file->private_data=&my_context;
   if(file->f_mode & FMODE_READ)
   printk("<1>""open for read mode\n");
   if(file->f_mode & FMODE_WRITE)
   printk("<1>""open for write mode\n");
   return 0;
}
/****************************** close driver routine
*************************************/
static int mychar_close(struct inode *inode,struct file *file)
{
  module_put(THIS_MODULE);
  printk("<1>""device file is closed\n");
  return 0;

}
/****************************** write driver routine
*************************************/
static ssize_t mychar_write(struct file *file,const char *buff,size_t len,loff_t *offset)
{
    struct d_context *tdev;
    int i,ret,n=sizeof(tdev->buf);
    tdev=file->private_data;
    for(i=0;i<len;i=i+n)
    {
                if(tdev->no_chars==n)
                        wake_up_interruptible(&rq);
                wait_event_interruptible(wq,tdev->no_chars<n);
    ret=copy_from_user(tdev->buf,&buff[i],n);
                tdev->no_chars=tdev->no_chars+n;
    }
                tdev->no_chars=len;
```

```
    wake_up_interruptible(&rq);
    printk("<1>""write is called\n");
    printk("<1>""data received from app:%s-%d (no of characters)\n",tdev->buf,tdev->no_chars);
    return (ssize_t)len;

}
/****************************** read driver routine
*************************************/
static ssize_t mychar_read(struct file *file,char *buff,size_t len,loff_t *offset)
{
        struct d_context *tdev;
        tdev=file->private_data;
                int n=sizeof(tdev->buf);
                wait_event_interruptible(rq,tdev->no_chars>0);
                                copy_to_user(buff,tdev->buf,n);
        tdev->no_chars=0;
    wake_up_interruptible(&wq);
        printk("<1>""read is called\n");
        return (ssize_t)len;
}
/****************************************************************************
************/
module_init(mydev_init);
module_exit(mydev_release);
```

MAKE FILE

```
obj-m =  basic_character_driver.o
all:
            make –C /lib/modules/$(shell uname –r)/build M=$(shell pwd) modules
clean:
            make –C /lib/modules/$(shell uname –r)/build M=$(shell pwd) clean
```

APPLICATIONS

Write application:

```
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
#include<unistd.h>
#include<sys/stat.h>
void main()
{
            int fd,ret;
            char buf[100];
            fd=open("/dev/my_char",O_WRONLY);
            if(fd<0)
            {
                        printf("failed to establish the                                         communication");
```

```
                exit(1);
        }
        printf("enter the input\n");
        scanf("%[^\n]s",buf);
        ret=write(fd,buf,strlen(buf));
if(ret<0)
        {
                printf("failed to erform write operation");
                exit(2);
        }
        printf("write return value ->%d\n",ret);
        close(fd);
}
```

Read application:

```
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
#include<unistd.h>
#include<sys/stat.h>
void main()
{
        int fd,ret=0;
        char buf[100];
        fd=open("/dev/my_char",O_RDONLY);
        if(fd<0)
        {
                printf("failed to establish the communication");
                exit(1);
        }
        ret=read(fd,buf,sizeof(buf));
        if(ret<0)
        {
                printf("failed to perform          write operation");
                exit(2);
        }
        printf("string ->%s\n",buf);
        close(fd);
}
```

Character Driver code execution flow

I.module loading

step:1 creation of device file: A device file can be created statically or dynamically based on necessity as device file created an correspnding inode object is created in that file system. Device file is created as a psedo files which are stored in virtual file system which is in ram not in physical memory like rom,HDD etc.

```
Inode object
of device file
```

step:2:initialising file operation structure members: In kernel level if you want to perform any operations on kernel objects you need to intialise on corresponding structure which is linked to the kernel object through the base address of the operation structure.

```
Struct file_operations op{
.read=my_read;
.write=my_write;
.open=my_open;
.close=my_close;};
```

Step3:intialise the variables which you want: in this the variables which are needed in kernel was declared and defined.

```
Char rbuf[]
Char wbuff[]
```

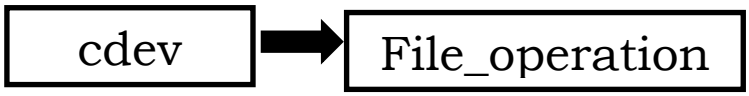
Step:4:declare a cdev structure: allocate a memory for cdev object for device file in static or dynamically

(i)statically: stuct cdev a;

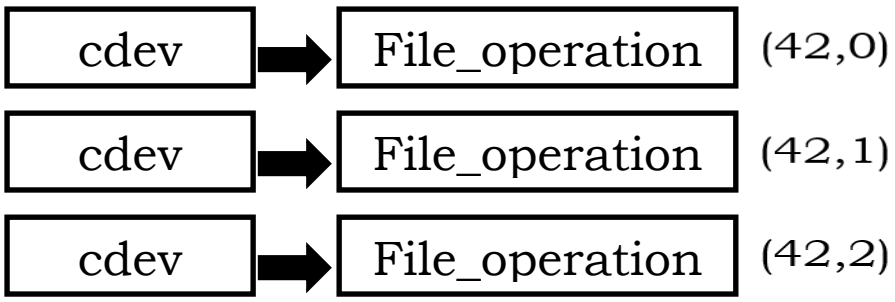
(ii) dynamically: using kalloc function and deallocating through kfree().

```
Struct cdev{
Owner;
Struct file_operation* ops;
Dev_t devno;
```

Step:5:cdev_init() function execution: in this we will pass a cdev object base address and file object base address which will get initialised in a member in cdev structure.

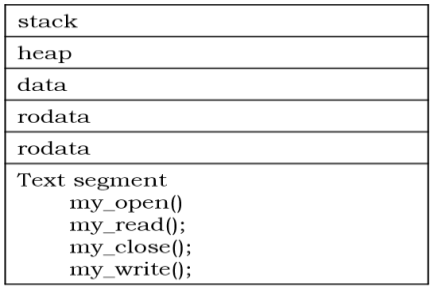


Step:6:cdev_add()function execution: in this function execution the second argument is count which denotes no of device files can access the driver it will grant the permission for that many minor numbers.

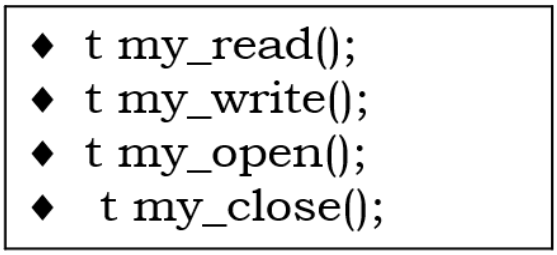


Step:7:adding function to kernel image: after excution of insmod the driver code the funtion definition is updated in text segment and varaibles are updated in corresponding segements which are kernel image and those symbols got updated in symbol table which is known as kallsyms

Vmlinux/kernel image



kernel symbol table/kallsyms



Module execution

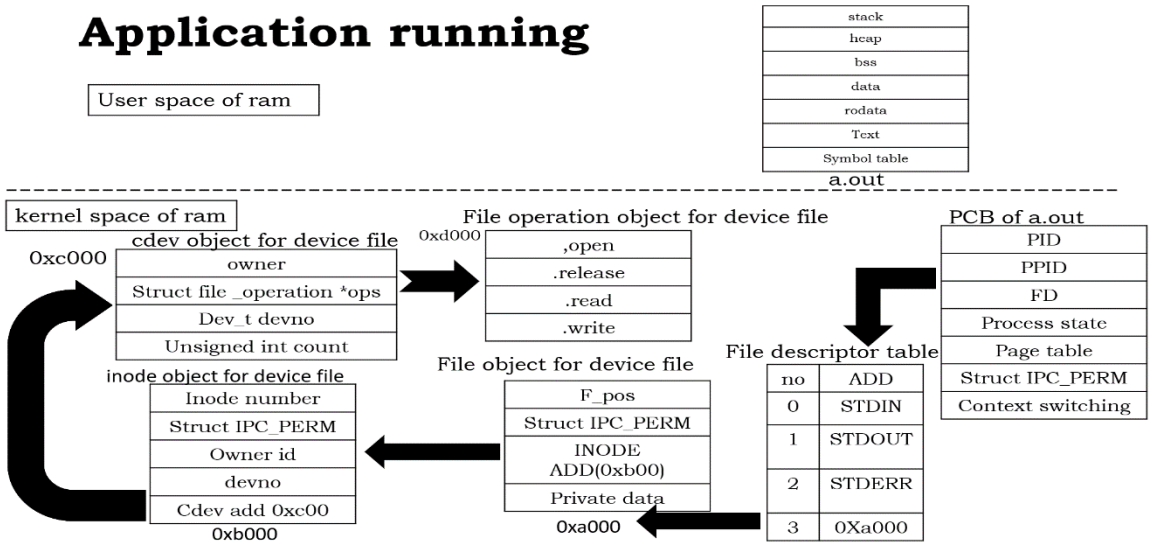
I. If you execute any process in an user space of ram which allocates memory segments that are used in this process.

II. For every process there will be a PCB(process control block) created in kernel space of ram .the pcb is used to store the information about the process which is under execution.

III. When an open system call execute invokes the process checks the file whether if it is present or not. If it is present it takes the inode number and it opens the inode object and creates a file object and its base address is updated in fd table.

IV. If you want to execute a file operation on device file. It is applied through file object. Whose base address is stored in fd table, through file object it goes to inode object whose base address is strode in file object and then process moves to the cdev object whose base address is stored in inode

object and then process moves to the file operation structure whose base address stored in cdev object of device file and the equivalent function is excuted.



IOCTL

Introduction

The operating system segregates virtual memory into kernel space and userspace. Kernel space is strictly reserved for running the kernel, kernel extensions, and most device drivers. In contrast, user space is the memory area where all user-mode applications work, and this memory can be swapped out when necessary.

There are many ways to Communicate between the Userspace and Kernel Space, they are:

- Ioctl
- Procfs
- Sysfs

IOCTL

IOCTL is referred to as Input and Output Control, which is used to talk to device drivers. This system call is 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 are Ejecting the media from a “cd” drive, changing the Baud Rate of Serial port,

Adjusting the Volume, Reading or Writing device registers, etc. We already have the 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 the driver

• Write IOCTL function in the driver

• Create IOCTL command in a Userspace application

• Use the IOCTL system call in a Userspace

Create IOCTL Command in the Driver

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

1. Define the ioctl command

#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.>

#define WR_VALUE _IOW('a','a',int32_t*)

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

Write IOCTL function in the driver

The next step is to implement the ioctl call we defined into the corresponding driver. We need to add the ioctl function to

our driver.

int ioctl(struct inode *inode,struct file *file,unsigned int cmd,unsigned long arg)

Where

<inode>: is the inode number of the file being worked on.

<file>: is the file pointer to the file that was passed by the application.

<cmd>: is the ioctl command that was called from the userspace.

<arg>: are the arguments passed from the userspace

Within the function “ioctl” we need to implement all the commands that we defined above (WR_VALUE, RD_VALUE). We need

to use the same commands in the switch statement which is defined above.

Then we 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.

example

This function will be called when we write IOCTL on the Device file

static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)

```
{
        switch(cmd) {
                case WR_VALUE:
                        if( copy_from_user(&value ,(int32_t*) arg, sizeof(value)) )
                        {
                                pr_err("Data Write : Err!\n");
                        }
                        pr_info("Value = %d\n", value);
                        break;
                case RD_VALUE:
                        if( copy_to_user((int32_t*) arg, &value, sizeof(value)) )
                        {
                                pr_err("Data Read : Err!\n");
                        }
                        break;
                default:
                        pr_info("Default\n");
                        break;
        }
        return 0;
}
```

File operation sturcture

```
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 ioctl command from a user application.

Create IOCTL command in a Userspace application

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

Example

#define WR_VALUE _IOW('a','a',int32_t*)

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

Use IOCTL system call in Userspace

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");

Where,

<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 need to be passed to the ioctl command.

Example:

ioctl(fd, WR_VALUE, (int32_t*) &number)

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

Now we will see the complete driver and application.

IOCTL in Linux – Device Driver Source Code

In this example we only implemented IOCTL. In this driver, I've defined one variable (int32_t value). Using ioctl

command we can read or change the variable. So other functions like open, close, read, and write.

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()
```

```
#include <linux/ioctl.h>
#include <linux/err.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)
{
        pr_info("Device File Opened...!!!\n");
        return 0;
}
 static int etx_release(struct inode *inode, struct file *file)
{
        pr_info("Device File Closed...!!!\n");
```

```
        return 0;
}
 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
{
        pr_info("Read Function\n");
        return 0;
}
 static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t *off)
{
        pr_info("Write function\n");
        return len;
}
static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
         switch(cmd) {
                case WR_VALUE:
                        if( copy_from_user(&value ,(int32_t*) arg, sizeof(value)) )
                        {
                                pr_err("Data Write : Err!\n");
                        }
                        pr_info("Value = %d\n", value);
                        break;
                case RD_VALUE:
                        if( copy_to_user((int32_t*) arg, &value, sizeof(value)) )
                        {
                                pr_err("Data Read : Err!\n");
                        }
                        break;
               default:
                        pr_info("Default\n");
                        break;
        }
```

```
        return 0;
}
static int __init etx_driver_init(void)
{
        /*Allocating Major number*/
        if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
                pr_err("Cannot allocate major number\n");
                return -1;
        }
        pr_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){
            pr_err("Cannot add the device to the system\n");
            goto r_class;
        }
        if(IS_ERR(dev_class = class_create(THIS_MODULE,"etx_class"))){
            pr_err("Cannot create the struct class\n");
            goto r_class;
        }
        if(IS_ERR(device_create(dev_class,NULL,dev,NULL,"etx_device"))){
            pr_err("Cannot create the Device 1\n");
            goto r_device;
        }
        pr_info("Device Driver Insert...Done!!!\n");
        return 0;
r_device:
        class_destroy(dev_class);
r_class:
```

```
        unregister_chrdev_region(dev,1);
        return -1;
}
static void __exit etx_driver_exit(void)
{
        device_destroy(dev_class,dev);
        class_destroy(dev_class);
        cdev_del(&etx_cdev);
        unregister_chrdev_region(dev, 1);
        pr_info("Device Driver Remove...Done!!!\n");
}
module_init(etx_driver_init);
module_exit(etx_driver_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("TEAM”);
MODULE_DESCRIPTION("Simple Linux device driver (IOCTL)");
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
```

application

```
#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("\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);
}
```

Procfs

Procfs is a vitual file system mounted as /proc directory in which every process information is stored in a individual directory whose name as pid which unique for each and every process in /proc directory the directories are created in runtime at process execution and delete at process termination. These information is passed from kernel level to user level for debugging purposes like at time of improper execution of process and it also contains some information about the components which are used in a process like io ports address, interrupts, symbols and memory related information as documented and strored as files.

Sudo Cat /proc/iomem ->command used to check input output memory address

Sudo objdump -s /proc/kallsyms->command used to see symbol table information

Sudo cat /proc/devices ->command used to check the type of devices

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. Userspace programs can use proc files to read the information exported by the 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 need to be writable 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 maybe the data that the module is handling. In such situations, we can create a proc entry for ourselves and dump whatever data we want to look into in the entry.

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

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

An entry that only reads data from the kernel space.

An entry that reads as well as writes data into and from kernel space.

Creating procfs directory

You can create the directory under /proc/* using the below API.

struct proc_dir_entry *proc_mkdir(const char *name, struct proc_dir_entry *parent)

where,

name: The name of the directory that will be created under /proc.

parent: In case the folder needs to be created in a subfolder under /proc a pointer to the same is passed else it can be left as NULL.

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. Let us see how we can create proc entries in version 3.10.

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

The function is defined in proc_fs.h.

Where,

<name>: The name of the proc entry

<mode>: The access mode for proc entry

<parent>: The name of the parent directory under /proc. If NULL is passed as a parent, the /proc directory will be set as a parent.

<proc_fops>: The structure in which the file operations for the proc entry will be created.

Note: The above proc_create is valid in the Linux Kernel v3.10 to v5.5. From v5.6, there is a change in this API. The fourth argument const struct file_operations *proc_fops is changed to const struct proc_ops *proc_ops.

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);

This proc entry should be created in the Driver init function.

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

create_proc_read_entry()

create_proc_entry()

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

The create_proc_entry is a generic function that allows creating both the read as well as the write entries.

create_proc_read_entry is a function specific to create only read entries.

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

Process to create an entry in /proc directory

Creating procfs directory

You can create the directory under /proc/* using the below API.

struct proc_dir_entry *proc_mkdir(const char *name, struct proc_dir_entry *parent)

where,

name: The name of the directory that will be created under /proc.

parent: In case the folder needs to be created in a subfolder under /proc a pointer to the same is passed else it can be left as NULL.

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. Let us see how we can create proc entries in version 3.10.

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

The function is defined in proc_fs.h.

Where,

<name>: The name of the proc entry

<mode>: The access mode for proc entry

<parent>: The name of the parent directory under /proc. If NULL is passed as a parent, the /proc directory will be set as a parent.

<proc_fops>: The structure in which the file operations for the proc entry will be created.

Note: The above proc_create is valid in the Linux Kernel v3.10 to v5.5. From v5.6, there is a change in this API. The fourth argument const struct file_operations *proc_fops is changed to const struct proc_ops *proc_ops.

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);
```

This proc entry should be created in the Driver init function.

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

create_proc_read_entry()

create_proc_entry()

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

The create_proc_entry is a generic function that allows creating both the read as well as the write entries.

create_proc_read_entry is a function specific to create only read entries.

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

Procfs File System

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

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

This is like a device driver file system. We need to register our proc entry filesystem. If you are using the kernel version below 3.10, this will not work. If you are using the Linux kernel v5.6 and above, you should not use this structure. You have to use struct proc_ops instead of struct file_operationslike below.

```
static struct proc_ops proc_fops = {
        .proc_open = open_proc,
        .proc_read = read_proc,
        .proc_write = write_proc,
        .proc_release = release_proc
```

};

Code:

Driver code:

```
#include<linux/kernel.h>
#include<linux/module.h>
#include<linux/init.h>
#include<linux/proc_fs.h>
MODULE_AUTHOR("");
MODULE_LICENSE("GPL");
char kbuff[1024];
struct proc_dir_entry *dir;
static int open(struct inode* inode,struct file* file)
{
printk("<6>""procfs file opend");
return 0;
}
static int close(struct inode* inode,struct file* file)
{
printk("<6>""procfs file close");
return 0;
}
static ssize_t read(struct file* file,char *buf,size_t len,loff_t *offset)
{
copy_to_user(buf,kbuff,len);
printk("<6> read function invoked");
return len;
}
static ssize_t write(struct file* file,const char *buf,size_t len,loff_t *offset)
{
copy_from_user(kbuff,buf,len);
printk("<6> write function invoked");
return len;
```

```
}
struct proc_ops fops={
.proc_open=open,
.proc_read=read,
.proc_write=write,
.proc_release=close};
static int driver_init(void){
dir=proc_mkdir("cdriver",NULL);
if(dir==NULL)
{
printk("<6>""failed to create directory in procfs");
return -1;
}
printk("<6>""to create directory in procfs");
proc_create("my_char",0660,dir,&fops);
printk("<6>""driver is loaided successfully");
return 0;
}
static void driver_exit(void){
remove_proc_entry("my_char",dir);
proc_remove(dir);
printk("<6>""driver is unloaded successfully");
}
module_init(driver_init);
module_exit(driver_exit);
```

Write application:

```
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
#include<unistd.h>
#include<sys/stat.h>
void main()
```

```
{
int fd,ret;
char buf[100];
fd=open("/proc/cdriver/my_char",O_WRONLY);
if(fd<0)
{
printf("failed to establish the communication");
exit(1);
}
printf("enter the input\n");
scanf("%[^\n]s",buf);
ret=write(fd,buf,strlen(buf));
if(ret<0)
{
printf("failed to erform write operation");
exit(2);
}
printf("write return value ->%d\n",ret);
close(fd);
}
```

Read application:

```
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
#include<unistd.h>
#include<sys/stat.h>
void main()
{
int fd,ret=0;
char buf[100];
fd=open("/proc/cdriver/my_char",O_RDONLY);
if(fd<0)
```

```
{
printf("failed to establish the communication");
exit(1);
}
ret=read(fd,buf,sizeof(buf));
if(ret<0)
{
printf("failed to perform read operation");
exit(2);
}
printf("string ->%s\n",buf);
printf("size ->%d\n",strlen(buf));
close(fd);
}
```

Make file:

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

Sysfs

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 use for exporting the debug information by the developer.

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

Kernel Objects

The 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;

};
```

Some of the important fields are:

struct kobject

(I) name (Name of the kobject. Current kobject is created with this name in sysfs.)
(II) parent (This is kobject's parent. When we create a directory in sysfs for the current kobject, it will create under this parent directory)
(III) ktype (the type associated with a kobject)
(IV) kset (a group of kobjects all of which are embedded in structures of the same type)
(V) sd (points to a sysfs_dirent structure that represents this kobject in sysfs.)

(VI) kref (provides reference counting)

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

So kobject is used to create kobject directory in /sys.

There are several steps to creating and using sysfs.

I. Create a directory in /sys
II. Create Sysfs file

Create a directory in /sys

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

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_kobj to the second argument, it will create the directory under /sys/firmware/. If you pass fs_kobj to 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.

Example

```
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 the above function we will create a directory in /sys. Now we need to create a 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 functions that can be used to create the kobject attributes. They can be found in the 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 an 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.

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

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);

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 the sysfs file, you should delete this file using sysfs_remove_file

void sysfs_remove_file (struct kobject * kobj, const struct attribute * attr);

Where,

kobj – object we're creating for.

attr – attribute descriptor.

Code:

Driver code:

#include<linux/kernel.h>

#include<linux/module.h>

```
#include<linux/init.h>
#include<linux/poll.h>
#include<linux/ioport.h>
#include<linux/errno.h>
#include<linux/cdev.h>
#include<linux/fs.h>
volatile char  c;
struct kobject* kadd;
MODULE_AUTHOR("");
MODULE_LICENSE("GPL");
static ssize_t show(struct kobject *kobj,struct kobj_attribute *attr,char *buf)
{
sprintf(buf,"%c",c);
printk("show->%c\n",c);
return 1;
}
static ssize_t store(struct kobject *kobj,struct kobj_attribute *attr,const char *buf,size_t count)
{
sscanf(buf,"%c",&c);
printk("store->%c\n",c);
return 1;
}

struct kobj_attribute kattr=__ATTR(c,0660,show,store);
static int pdriver_init(void){
int ret;
kadd=kobject_create_and_add("characterdriver",kernel_kobj);
if(kadd==NULL)
{
printk("<6>""failed to create a directory\n");
return -1;
}
```

```
ret=sysfs_create_file(kadd,&kattr.attr);
if(ret<0)
{
printk("<6>""failed to create file\n");
return -1;
}
printk("<6>""driver is loaded successfully");
return 0;
}
static void pdriver_exit(void){
kobject_put(kadd);
sysfs_remove_file(kernel_kobj,&kattr.attr);
}
module_init(pdriver_init);
module_exit(pdriver_exit);
```

Write application:

```
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
#include<unistd.h>
#include<sys/stat.h>
void main()
{
int fd,ret;
char a;
fd=open("/sys/kernel/characterdriver/c",O_WRONLY);
if(fd<0)
{
printf("failed to establish the communication");
exit(1);
}
printf("enter a number\n");
```

```
scanf("%c",&a);
ret=write(fd,&a,sizeof(a));
if(ret<0)
{
printf("failed to erform write operation");
exit(2);
}
printf("write return value ->%d\n",ret);
close(fd);
}
```

Read application:

```
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
#include<unistd.h>
#include<sys/stat.h>
void main()
{
int fd,ret=0;
char buf;
fd=open(/sys/kernel/characterdriver/c",O_RDONLY);
if(fd<0)
{
printf("failed to establish the communication");
exit(1);
}
ret=read(fd,buf,sizeof(buf));
if(ret<0)
{
printf("failed to perform read operation");
exit(2);
}
```

```
printf("character->%c\n",buf);
close(fd);
}
```

Make file:

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

Export_symbol()

- Since kernel 2.6 version the function which is not applied static keyword is available to all the .c codes in the kernel source code
- After 2.6 kernel version every function is restricted to its file scope. to utilize a function from one driver to another driver the macro EXPORT_SYMBOL is used.
- In c codes symbols are nothing but a variables or functions which you declared in that code.
- through EXPORT_SYMBOL macro we can increase the function or variables scope from file scope to process scope.

Macro definition

Header file : export.h

```
#define EXPORT_SYMBOL(sym)            __EXPORT_SYMBOL(sym,"")
#define EXPORT_SYMBOL_GPL(sym)  __EXPORT_SYMBOL(sym,"gpl")
#define __EXPORT_SYMBOL(sym,sec,"")     __EXPORT_SYMBOL(sym,sec,"")
#define __EXPORT_SYMBOL(sym,sec,ns) __GENKSYMS_EXPORT_SYMBOL(sys)
```

sym-----→symbol

sec-----→license

ns------→namespace

Syntax:

Through export_symbol the function which is shared between two drivers we must declare in two drivers codes and define in one driver code and another one can directly invoke without definition.

syntax:

EXPORT_SYMBOL(function name or variable name)

Export_SYMBOL_GPL macro is used to export those symbols for particular license

syntax:

EXPORT_SYMBOL_GPL(function name or variable name)

Driver 1 code:

```
#include<linux/kernel.h>
#include<linux/module.h>
#include<linux/init.h>
#include<linux/export.h>
void function1 (void);
int a=10;
MODULE_AUTHOR(“Ajay”);
MODUEL_LICENSE(“GPVL2”);
MODULE_DESCRIPTION(“driver1”);
EXPORT_SYMBOL(&a);
EXPORT_SYMBOL(function1);
static int driver1_init(void)
{
        printk(“<6>””driver1 is loaded”);
        return 0;
}
void function1(void)
{
        printk(“<6>””function from driver1”);
}
static void driver1_exit(void)
{
        printk(“<6>””driver1 is unloaded”);
}
module_init(driver1_init);
module_exit(driver1_exit);
```

Driver 2 code:

```
#include<linux/kernel.h>
#include<linux/module.h>
#include<linux/init.h>
void function1 (void);
extern  int a;
MODULE_AUTHOR(“Ajay”);
MODUEL_LICENSE(“GPVL2”);
MODULE_DESCRIPTION(“driver2”);
static int driver2_init(void)
{
        printk(“<6>””driver2 is loaded”);
        function1();
        printk(“<6>””a=%d”,a);
        return 0;
}
static void driver2_exit(void)
{
        printk(“<6>””driver2 is unloaded”);
}
module_init(driver2_init);
module_exit(driver2_exit);
```