Design And Implementation Of Linux Character Device Driver To Provide Extra IPC

***Nada A.Z. Abdullah* ***

Abstract

ln this paper we will design and Implement a character device driver which uses two device special files to allow a pair of processes to send short variable-length text message to each other. The driver should make sure that multiple readers and multiple writers are not permitted and also that read ( ) s will not block even when there are no messages to read and that write ( )s will not block however many messages are written before the next read ( ) occurs .This device driver is just going to control some system memory as its "hardware device" and effectively provide an extra **IPC** mechanism in addition to those already available.

***Baghdad University/ College of Science/ Computer science Department**

1 Introduction

One of the purposes of an operating system is to hide the peculiarities of the system's hardware devices from its users. For example the Virtual File System presents a uniform view of the mounted filesystems irrespective of the underlying physical devices.

The CPU is not the only intelligent device in the system; every physical device has its own hardware controller. Each hardware controller has its own control and status registers (CSRs) and these differ between devices. The CSRs are used to start and stop the device, to initialize it and to diagnose any problems with it. Instead of putting code to manage the hardware controllers in the system into every application, the code is kept in the Linux kernel. The software that handles or manages a hardware controller is known as a device driver. The Linux kernel device drivers are, essentially, a shared library of privileged, memory resident, low level hardware handling routines. It is Linux's device drivers that handle the peculiarities of the devices they are managing. [Rus 99]

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. Every device in the system is represented by a *device special file.* Linux supports three types of hardware device: character, block and network.

2 Character Device Drivers

Character devices, the simplest of Linux's devices, are accessed as files, applications use standard system calls to open them, read from them, write to them and close them exactly as if the device were a file. As a character device is initialized its device driver registers itself with the Linux kernel by adding an entry into the `chrdevs` vector of `device_struct` data structure as shown in figure (1). The device's major device identifier (for example 4 for the `tty` device) is used as an index into this vector. The major device identifier for a device is fixed [LRU03].

Each entry in the `chrdevs` vector, a `device_struct` data structure contains two elements; a pointer to the name of the registered device driver and a pointer to a block of file operations. This block of file operations is itself the addresses of routines within the device character device driver each of which handles specific file operations such as open, read, write and close. The contents of `/proc/devices` for character devices is taken from the `chrdevs` vector.

When a character special file representing a character device (for example `/dev/cua0`) is opened the kernel must set things up so that the correct character device driver's file operation routines will be called. Just like an ordinary file or directory, each device special file is represented by a VFS inode . The VFS inode for a character special file, indeed for all device special files, contains both the major and minor identifiers for the device. [Aivazian02]

When the character special file is opened by an application the generic open file operation uses the device's major identifier as an index into the `chrdevs` vector to retrieve the file operations block for this particular device. It also sets up the `file-`data

structure describing this character special file, making its file operations pointer point to those of the device driver. Thereafter, all of the application's file operations will be mapped to calls to the character device set of file operations. [Rus 99]

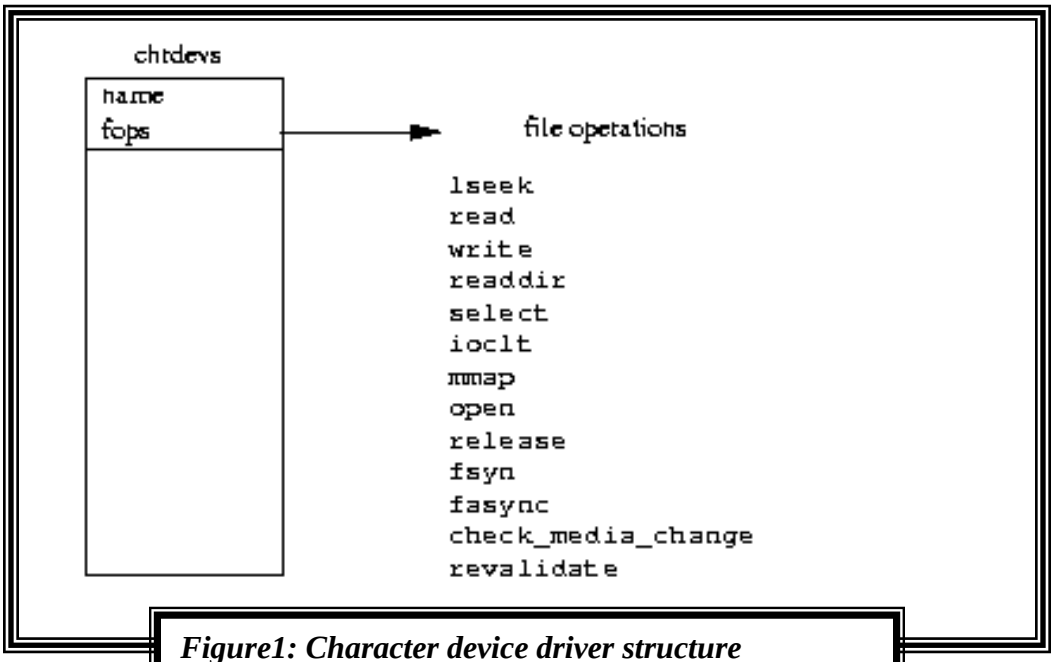

Figure1: Character device driver structure

3 The file operations Structure[Salzman03]

The file_operations structure is defined in linux/fs.h, and holds pointers to functions defined by the driver that perform various operations on the device. Each field of the structure corresponds to the address of some function defined by the driver to handle a requested operation. The file_operations structure holds the address of the module's function that performs that operation. Here is what the definition looks like:

```
struct file_operations {
struct module *owner;
loff_t (*llseek) (struct file *, loff_t, int);
ssize_t (*read) (struct file *, char *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
int (*readdir) (struct file *, void *, filldir_t);
unsigned int (*poll) (struct file *, struct poll_table_struct *);
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, struct dentry *, int datasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
ssize_t (*readv) (struct file *, const struct iovec *, unsigned long,
loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *, unsigned long,
loff_t *);
```

```
};
```

4 Registering Character Devices[LFQ01]

Each device driver gets the opportunity to initialize itself and its hardware at system boot time. For character device drivers, this is achieved by having an initialization function (init ()) in the driver and then placing a call to this function into the kernel chr_dev_init () function..

Suppose our device driver has the name prefix tdd_, then the init () routine would be called tdd_init (). The registration is performed by calling the kernel's register_chrdev () function as follows: [Aivazian02]

register_chrdev (major , name , file_op)

Where major is the major device number to be used by this driver, name is the string that gives the name of the driver, and file_op is the pointer to the driver's file_operations structure.

5 Basic Entry Points[Cornes97]

The main system calls which can be used with a character device or a file are: open (), close (), read (), write (), and ioctl () each of this system calls, when it has been established that they are intended for the proposed driver can result in the appropriate driver function being executed .

5.1 open Function

The driver's open () function is called when a user process executes an open () system call on a device special file associated with this driver. The prototype of the open () function is:

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

where the inode parameter is pointer to the inode structure of the device special file and file is a pointer to the file structure for this device.

5.2 close Function

The driver's close function is called when the last user process which has the device open close ()s it. The prototype of the close() function is:

*void close (struct inode *inode , struct file *file) ;*

where inode is pointer to the inode structure of the devise special file and file is a pointer to the file structure for this device .

5.3 read Function

The drivers read () function is called whenever a read () system call is executed on a device special file associated with this character device driver. The prototype of this function is:

*void read (struct inode *inode , struct file *file, char *buf , int cont) ;*

where inode is a pointer to the inode structure of the device special file, file is a pointer to the file structure for this device, buf is a pointer to a buffer in the user, and count is the number of bytes required by the user process.

5.4 write Function

The write() function is performed on a device special file belonging to this driver. The prototype of this function is:

*void write (struct inode * inode, struct file *file, char *buf, int count) ;*

Where inode is a pointer to the special file's inode structure, file is a pointer to the file structure, buf is a pointer to a buffer in user space passed into the write ( ) system call, from which user characters will be written, and count is the number of bytes to transfer **.**

5.5 Special Control Functions[Cornes97]

In addition to the basic open, close, read and write operations, it is sometime desirable to send control information to the device driver or to take status information from it. In this case (and many others like it) the solution is to use the ioct1() system call. This system call is implemented by just calling an ioct1 () function in the device driver. ioct1 () function has the following prototype:

*int ioct1 (struct inode *inode, strut file *file, usigned int cmd,usigned long arg) ;*

where inode and file are the same as before , cmd is a device driver specific command code to be performed, and arg is any data of 4-byte (typically an int or a struct *) which provides a parameter for the particular cmd value. The cmd and arg parameters are obtained from the second and third parameters to the ioct1 ( ) system call .

6 Implementation

In this paper we implement a character device driver which uses two device special files to allow a pair of processes to send short variable-length text message to each other. The driver should make sure that multiple readers and multiple writers are not permitted and also that read () s will not block even when there are no messages to read and that write () s will not block however many messages are written before the next read ( ) occurs.

Unlimited write capacity is dangerous as it is possible for some kind of error to stop the consumer process from reading any data. This leaves the producer process writing messages unchecked until, eventually, the system runs out of space to store them. Unlimited write capacity also requires more complex physical implementation. In fact for long lived processes there doesn't even need to be any kind of error for this problem to occur eventually. All that is required is that, on average, the producer is generating messages faster than the consumer can use them up. With this in mind, it might be sensible to set some arbitrary, but large, limit on the number of message that can be stored simultaneously within the device driver.

6.1 Header Information

In order to implement the requirement of the specification, the first task is to decide what the device driver's internal data structures will look like. What is required is that the device driver should be capable of storing a number of messages of short text messages which are in transit between two processes. The number of messages the driver is required to hold is supposed to be infinite (infinite on a computer usually means ' until we run out') but in practice it would probably only hold a few messages until they could be read. This means that we really want to use a variable number of dynamically allocated buffers which can be built up a FIFO queue which is best implemented as a linked list . This will required a kernel mechanism for the dynamic allocation and release of blocks of memory. The structure from which the linked list of messages will be built has the following layout:

```
struct  tdd-buf
{
      int buf_size
      char buffer[MAX_BUF]:
      struct tdd_ buf *link:
};
```

where buffer [] is the array that holds one of the short messages, buf_size says how many characters in the buffer [] array, and link is the linked list pointer to the next tdd_buf. The symbolic MAX_BUF will be set to whatever marches our idea of the maximum length of a 'short messages' ,by default is 120 characters. The developed device driver header information is as follows :

```
#define KERNEL
#include ‹Linux / kernel .h  ›
#include ‹Linux /sched .h ›
#include ‹ Linux/ tty .h ›
#include ‹ Linux/ signal .h ›
#include ‹ Linux/errno . h ›
#include ‹Linux /malloc .h ›
#include ‹asm /io .h ›
# include <asm/segment.h>
# include <asm/system.h>
include <asm/irq.h>
# include "tdd.h"
static int tdd_trace ;
static int write_busy
static int read_busy;
static struct tdd_buf *qhead;
static struct tdd_buf *qtail;
static int tdd_read (struct indo *, struct file *, char *, int );
static int tdd_write (struct indo *, struct file *, char *, int );
 static int tdd_ioctl (struct indo *, struct file *, unsigned int unsigned long)
static int tdd_open struct (struct inode *, struct file *).
Static void tdd_realised (struct inode *, struct file *);
Extern void console_print (chare  *);
struct file operation tdd fops =
{
null;
```

```
tdd_read;
tdd_write;
null;
null;
tdd_ioctl;
null;
tdd_open;
tdd_release;
null;
null;
  null;
 null;
};
```

Over and above this header code there is also header file called tdd.h which contains the #defines and structure declaration require by the devise driver and also by user code wishing to use this driver :

```
#ifdef  KERNEL                                          /* if we are in kernel code */
#define trace_txt(txt) \
{   \
if  (tdd_tace)  \
{  \
      console_print(text);\
console_print("\n");\
}  \
}
#define trace_chr(chr) \
{ \
           if (tdd_trace)  \
console_print (chr) ;  \
}
#define TDD_WRITE 0                         /* /dev/tddw minor device number */
#define TDD_READ  1                         /* / dev/tddw minor device number */
#endif
#define FALSE 0
#define TRUE   1
#define MAX_BUF     120                 /* size of struct tdd_buf buffer */
#define TDD_TRON (('m'<<8) |0x01)           /* trace on cmd for ioctl () */
#define TDD_TROFF (('m'<<8)|0x02)           /* trace of cmd for ioctl () */
struct tdd_buf
{
int buf_size;
char buffer [MAX_BUF] ;
struct tdd_buf *link;
};
```

6.2 init Function

Moving on into the driver code proper the first thing to look at is the initializing function tdd_init():

```
void tdd_init(void)
{
        tdd_trace = TRUE;
if  (register_chrdev (30,  "tdd", &tdd_fops))
TRACE_TXT ("cannot register tdd as major device 30 ")
else
trace_txt (" developed device driver registered successfully 30 ")
qhead = 0;
write_busy = false;
read_busy = false;
tdd_trce = false;
return;
}
```

This routine is executed at system boot time remember that a call to the routines needs to be added to the chr_dev_init() function in the file :

/user/src/linux/drivers/char/mem.c

When tdd_init() is executed it calls a kernel function register_chrdev() to add its file_operations structure to the character device routine address table.

6.3 open Function

The device drivers open function (tdd_open()) is called whenever an open () system call is performed on one of the two device special file associated with this driver:

```
static init tdd_open (struct inode *inode, struct file *file)
{
       trace_txt("tdd_open")
switch (MINOR(inode->i_rdev))
{
case TDD-WRITE:
   if  (write_busy)
       return –EBUSY;
else
       write_busy = TRUE;
       returne 0;
case TDD-READ:
if   (read_busy)
     return  –EBUSY;
else
     read_busy = TRUE;
     returne 0;
default:
   return –ENXIO;
   }
}
```

If there where any hardware involved with the device driver then this routine would arrange to bring it into service.

6.4 release Function

The release function tdd_release () is called when the last process that is holding open each of the device special files associated with this device driver closes it with the close() call. In fact since only one process at a time can open each of the device special files for this driver then a close() from this process will also call the drive release routine (tdd_release()):

```
static void tdd_release (struct inode *inode,  struct file *file)
{
            trace_txt("tdd_release")

switch (MINOR(inode->i_rdev))
{
case TDD-WRITE :
              write_busy = FALSE;
               return;
case TDD-READ:
            read_busy = FALSE;
return;
    }
}
```

6.5 write Function

The write function tdd_write () is called every time a process uses the write () system call on an open file descriptor associated with one of the device special files belonging to this devise driver:

```
static init tdd_write (struct inode *inode,  struct file  *file,char  *buffer, int count)
{
         int i,  len;
        struct tdd_buf  *ptr;
        TRACE_TXT ("tdd_write")
If (MINOR (inode->i_rdev) != TDD-WRITE)
return –EINVAL;
if  ((ptr = kmalloc(sizeof(struct tdd_buf) ,  gfp_kernel)) = =0)
retune –ENOMEM;
len = count<max_buf ?count:MAX-BUF;

if  (verify_area (VERIFY-READ , buffer , len ))
return –EFAULT;

for  (I = 0;  i<count  && i<MAX_BUF; ++i)
{
       ptr->buffer [i] = get_user_byte(buffer+i);
        TRACE_CHR("w")
}
ptr->link = 0;
if  (qhead = =0)
      qhead = ptr;
else
  qtail->link = ptr;
 qtail = ptr;
```

```
  TRACE_CHR("\n")
ptr->buf_size =i ;
return i;
}
```

The third and further parameters to tdd_write() are the buffer and character count passed by the user process into the write () system call the contains of this buffer need to be copied into the device drivers internal linked list of messages.

6.6. read Function

the tdd_read function is called when user process calls the read() system call to read from a device special file controlled by this device driver:

```
static int  tdd_read(struct inode *inode,  struct file *file, chr *buffer, int count)
{
        int i,  len;
        struct tdd_buf  *ptr;
TRACE_TXT("tdd_read")
if (MINOR(inode->i_rdev)!=tdd_read)
   return –EINVAL;
if (qhead= =0)
return –ENODATA;
ptr = qhead;
 qhead = qhead->link;
len = count<ptr->buf_size?count : ptr->buf_size;
if  (verify_area(VERIFY_WRITE, buffer, len))
     return –EFAULT;
for  (I = 0;  i<count && i<ptr->buf_size;  ++i)
{
   put_user_byte (ptr->buffer [i] , buffer+i);
TRACE_CHR("r")
}
TRACE_CHR("\n")
kfree_s(ptr, sizeof(struct tdd_buf));
return i;
 }
```

Once the buffer and count parameters into tdd_read () pointer to a buffer in user space and a character count which were passed a parameters into the associated read () system call the obvious difference between this and the tdd_write function being that this time the buffer is to receive character from a message structure in the device driver.

6.7 ioctl Function

The ioctl () function for this driver is very simple but it does serve to illustrate the idea. Two new ioctl() calls are provided by this driver to switch the trace facilities on and off this is done using the TDD_TRON and TDD_TROFF commands to ioctl () respectively:

```
static int tdd_ioctl(struct inode  *inode,  struct file  *file, unsigned int cmd ,  unsigned long arg)
{
TRACE-TXT("tdd_ioctl")
```

```
switch (cmd)
{
case TDD_TRON:
            tdd_trace = TRUE;
                return  0;
case TDD_TROFF:
             tdd_trace = FALSE;
                  return  0;
default :
 return  -EINVAL;
      }
}
```

This cmd and arg parameter to tdd_ioctl () are the same values as were passed in the user process to the ioctl () system call . in this case the value of arg is not significant. Only the two values TDD_TRON and TDD_TROFF have any special significance to this driver and that is when they are used as cmd values the only action performed by this ioctl () commands is to set and reset the tdd_trace flag.

7 Conclusion

From developing the device driver the following concluding remarks can be drawn:

- In this paper a new IPC mechanism is developed.
- A device driver is just a collection of routines with various specific tasks to perform.
- Each device driver has essentially the same set of routines and so some mechanism is required to prevent name clashes between drivers. A simple mechanism is to make all the names unique. This is done by choosing a simple unique prefix for each device driver which will be added to the function names in the driver. This means that even though most device drivers will provide an open () routine, for instance the prefix for each driver will make all the open () routine names unique within the kernel.
- The proposed device driver is going to control some system memory as its 'hardware device' and effectively provide an extra **IPC** mechanism in addition to those already available with Linux Operating System. The **IPC** semantics provided by this new driver, however, are quite different from those available with the existing **IPC** mechanisms.
- The proposed device driver has been tested, and since the developed device driver running in the kernel, thus communication is secure and fast.

References

⎩ [Rus 99] “The Linux Kernel” ,David A Rusling 1999,Internet online.

⎩ [Salzman03]”The Linux Kernel Module Programming Guide” Peter Jay Salzman,Ori Pomerantz, ver. 2.4.0, 2003.

⎩ [LFQ01]”Linux Frequently Asked Questions with Answers”, Robert Kiesling,2001,Internet online.

⎩ [LRU03]“LINUX Rute Users Tutorial and Exposition”**,** Paul Sheer**,** Version 0.9.1 ,2003.

⎩ [Cornes97] “The Linux A-Z”, Phil Cornes ,Prentice Hall ,1997.

⎩ [Aivazian02] “Linux Kernel 2.4 Internals” ,Tigran Aivazian,Internet,2002.