Department for Mathematics & Computer Science
University of Southern Denmark

DM510 Operating Systems

Department for Mathematics & Computer Science

IMADA

Project 3: Kernel Module

Torvald Johnson – tjohn16

April 15, 2017

Table of Contents

1 Introduction

The task at hand was to create a kernel module, to be run in User-Mode Linux, implementing a that exposed two character drivers to user-space. Instead of interfacing with hardware, as a normal device driver normally does, the point of this project was to make two processes communicate through a device driver. More specifically, the module was to expose two character drivers to user space, one of which wrote to Bounded Buffer 1 and read from Bounded Buffer 2, and the other of which wrote to Bounded Buffer 2 and read from Bounded Buffer 1, thus enabling these two devices to communicate with each other while accepting read and write commands from user space.

2 Design

A large problem faced when trying to structure this module, was the fact that there were specified to be two devices interacting with two buffers. Although this seems simple enough, it became very complex when considering the fact that one device's read buffer needed to be the other device's write buffer, and vice versa. After examining the scull module example given to base this module on, I had at first assumed that the buffers should be contained within the device structure itself. This would make sense normally, in a situation wherein a device would be reading from and writing to the same buffer. However, I could not come up with a good way to structure the module such that the two devices could access each other to read and write information to these buffers without being needlessly confusing, so the decision was made that the buffers would not be held within the device structures, but instead be held separately. See the implementation section of this report for more information regarding this decision within the system.

One important design decision made regarding this project, was that of how to handle requests which are unfulfillable at the time that the request is made. For example, what if a process attempts to read from an empty buffer? Or what if a process attempts to write to a full buffer? Since this would lead to unexpected results, the decision was made to employ blocking, and put such processes to sleep until a future time when the system would be able to proceed with these requests. See the implementation section of this report for a more detailed discussion regarding when and how processes were put to sleep, and woken up.

Another important aspect of this module, was the allowance of concurrency such that multiple processes could execute simultaneously. However, this led to the fact that data within the buffers was likely to change at any time depending on read and write requests to the devices. Since this could lead to race conditions, or unexpected results if the buffer being read from is being written to at the same time, the decision was made to use semaphores to lock down the system and avoid such problems.

As usual, the system was designed to check all parameters and handle any errors. Any instance in which memory was allocated or data transferred within the program, the result is verified to be correct and variables are checked to be within acceptable parameters. In the event of an unacceptable outcome, a short descriptive error message is printed to the console, and the program is stopped, returning the negative value of the relevant error code, thus preventing module from failing uncontrolled and bringing down the kernel.

3 Implementation

As previously specified, the device structure and the buffers were to be held separately within the module. As such, two structs were created, one to hold device data and one to hold buffer data. The device struct contains two pointers to specify a read buffer and a write buffer, while the buffer struct holds all other data pertaining to the buffers. Both the buffer and the device struct were given a semaphore, so that both could be locked or unlocked depending on the state of the module.

```
/* structure for our char buffers */
struct my_buffer
{
    char *buffer, *end;           /* pointers to the beginning and end of the buffer */
    int buffersize;
    int count;                    /* keeps track of how far into the buffer we are */
    int rp, wp;
    int numreaders, numwriters;   /* number of readers and writers */
    struct semaphore sem;
    wait_queue_head_t queue;      /* queue of sleeping processes waiting for access to
buffer */
};

static struct my_buffer *buffers;

/* structure for our device */
struct my_device {
  struct my_buffer *rbuf, *wbuf;  /* pointers to the read-buffer and the write-buffer */
  struct semaphore sem;
  struct cdev cdev;
}

static struct my_device *dm510_device;
```

Figure 3.1: Buffer Struct and Device Struct

Specifically, in this module the device is locked using its semaphore when the device is opened, and is only unlocked when it is released. This decision was made because as long as the device is opened, there is no reason for it to be available for other processes to change its data. If a process attempts to access a device that is locked, that process will have to wait until that device is unlocked when it is released before it is able. The buffers, however, need to be available any time they're not in use so that devices can read from and write to them when necessary. To this end, the buffers are only locked directly before they are read from or written to, and are unlocked directly after being read from or written to, or if there was an error. This allows them to be available for any device that wants to access them most of the time, but avoids race conditions or unwanted modification of data while other processes are accessing them.

The implementation of blocking i/o in this module makes use of the Simple Sleeping technique described in the *Linux Device Drivers, Third Edition* book. When a process attempts to read from an

empty buffer, that process is put to sleep using the wait_event_interruptible() function. The same applies to any process attempting to write to a full buffer. This is kept track of using the wait queues stored in each buffer. Processes are woken up using the wake_up_interruptible() function any time data within the buffers has been changed, namely after a successful read or write. wake_up_interruptible() is also called when a write process first detects that the buffer is full, to ensure that if there is a read process sleeping, it will then wake up and clear a portion of the buffer to make room for the write process.

```
if(down_interruptible(&(dev->rbuf->sem))) {
        return -ERESTARTSYS;
      }

      while (dev->rbuf->count == 0) {
        /* buffer is empty, prepare for sleep */
        up(&(dev->rbuf->sem));
        wake_up_interruptible(&(dev->rbuf->queue));
        if (filp->f_flags & O_NONBLOCK) {
          return -EAGAIN;
        }
        /* Sleepy time! Put process to sleep and wait for more data. */
        if (wait_event_interruptible(dev->rbuf->queue, (dev->rbuf->count != 0))) {
          return -ERESTARTSYS;
        }
        /* reacquire lock and loop */
        if(down_interruptible(&(dev->rbuf->sem))) {
        return -ERESTARTSYS;
        }
      }
      while((data < count) && dev->rbuf->count > 0) {
      /* loop and read until the specified amount is read or the buffer is empty, one
character at a time */
      ret = copy_to_user(buf+data,dev->rbuf->buffer+dev->rbuf->rp,1);
      if (ret) {
        up(&(dev->rbuf->sem));
        return -EFAULT;
      }
      data++;
      dev->rbuf->rp++;
      dev->rbuf->count--;
      if(dev->rbuf->rp == dev_buffer_size) {
        dev->rbuf->rp = 0;
      }
      }

      up(&(dev->rbuf->sem));
      /* Data has been cleared from buffer! Wake up any sleepy processes */
      wake_up_interruptible(&(dev->rbuf->queue));
```

Figure 3.2: Excerpt from the dm510_read() function, showing implementation of semaphores and sleeping

It is also important to note, that besides the read and write functions, simple device control was also made available using the ioctl function. The implementation is limited to retrieving and adjusting the buffer size, as well as retrieving and setting the maximal number of processes that are allowed to read from the device at a time. I did not feel that there were any more relevant functionalities to be included in the ioctl function. In this respect, it basically acts as a getter and setter for these parameters within the module.

4 Testing

The test file moduletest.c, which was provided along with the project description was used to test the module. A second file, moduletest2.c, was also created consisting of five tests, to ensure that the module could handle various negative conditions, and to run the ioctl function. A video of these tests running will also be included, and this section of the report will go over the tests and their expected and actual output, providing a reference to the relevant time in the video for each test. Below, for your convenience, is a table of the tests listing their names, a short description of each one, and their timestamps in the included video.

Name:	moduletest.c: Concurrency	moduletest2.c: Test 1	moduletest2.c: Test 2	moduletest2.c: Test 3	moduletest2.c: Test 4	moduletest2.c: Test 5 ioctl
Description:	Opens two devices, reads and writes with multiple processes	Write a negative number of bytes to the buffer	Write a NULL value to the buffer	Read from a buffer, passing NULL as the userspace buffer parameter	Write with one device, read with the other, using acceptable data.	Get and set buffersize, get and set amount of readers.
Result:	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED
Timestamp:	0:30	0:39	0:44	0:50	0:55	1:00

Figure 4.1: Table of tests

The first test file, which was included in the files given to the students with the project description, attempts to read and write several times with different processes. For this test to succeed, the devices need to be able to sleep when there is no data within the buffer and when the buffers are full, as well as wake up processes to read from or write to the buffer as necessary. In order to pass, the expected value and actual value printed should be the same, for both buffers. Each buffer finished the test with the same value as its expected value, so this test has passed.

From my own test file, Test 1 is an attempt to write to the buffer, in which the number of bytes to write is incorrectly specified as a negative value. For this test to pass, an EFAULT error should be thrown after copy_from_user() fails to copy to the buffer, and an error message should be displayed explaining that there was a “write: Bad address,” returning -14. Following this, both devices should be released. The test begins at 39 seconds into the video, the expected error message is displayed, the devices are released, and the test has passed.

Test 2 is an attempt to write a message with a positive number of bytes, but a NULL value as the buffer parameter. For this test to pass, an error should be thrown when the program attempts to copy_from_user(), displaying the error “write: Bad address” and returning -14. The test can be seen at 44 seconds into the video. The expected outcome is printed to the console, and the test passed.

Test 3 is designed to test bad data passed to the read function. First one device successfully writes some data to its buffer. Then another device attempts to read with a positive specified length, but a NULL value as the buffer parameter. The expected outcome for this test, is that when the device attempts to copy_to_user(), it will display the error “read: Bad address” and return -14. The test begins at 50 seconds into the video, and passes because the expected outcome is printed.

Test 4 attempts to write successfully with one device to the buffer, and then read successfully with another device from the buffer, using acceptable data for all parameters. For this test to pass, no error messages should be printed, and both devices should be released at the end of it. At 55 seconds into the video, you can see that this is exactly what happens, and the test passed.

Test 5 attempts to use the ioctl function to get, and set data within the module. First it attempts to get the current buffer size, which should display as 2000. Then it attempts to set the buffer size to 5000, and display the new buffer size. Then it attempts to get the current amount of allowed read processes, which should be 1. It then attempts to set that amount to 50, and display the new amount of read processes. This test begins at 1:00 minute into the video, and does exactly as expected, which means that it passes.

6 Conclusion

Looking back at this entire project, I am satisfied with what I have been able to accomplish. The module meets the requirements outlined in the provided project description; it is able to expose two character devices to user-space, which read and write to alternate buffers. I am very glad that I was able to figure out how to hold the buffers and their data in a separate struct so that they could be assigned to pointers in the device struct. This was a very perplexing problem for me, and I actually lost a lot of time designing and building the system originally around the idea of having the buffers contained within the device struct. This was a positive experience, however, because my struggles in that implementation helped me to think about the problem from a different angle, and come up with a better design. I have also learned even more about concurrency, and it was interesting when I realized that I should probably lock the devices and buffers separately, to keep the buffers available as often as possible. I also feel that the solution implemented for putting processes to sleep and waking them up is solid, and I learned a lot about blocking and non-blocking i/o when putting this module together. I do feel that my ioctl function is very simplistic, but I could not think of what else to put in there besides a few getters and setters. I actually really enjoyed this project in general, I’ve never built a device driver before, and it was interesting to explore a very new aspect of software development. This was genuinely a lot of fun, and I am much more interested in the inner-workings of an operating system now that I have experienced designing and implementing a Linux kernel module.

7 Appendices

7.1 Appendix I: Source code for dm510_dev.c

```
/* Prototype module for third mandatory DM510 assignment */
#ifndef __KERNEL__
#  define __KERNEL__
#endif
#ifndef MODULE
#  define MODULE
#endif

#include <linux/module.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/types.h>
#include <linux/wait.h>
#include <asm/uaccess.h>
#include <linux/semaphore.h>
/* #include <asm/system.h> */
#include <asm/switch_to.h>
#include <linux/cdev.h>
#include <linux/sched.h>
#include "linux/stddef.h"

#define init_MUTEX(LOCKNAME) sema_init(LOCKNAME,1);

/* Prototypes - this would normally go in a .h file */
static int dm510_open( struct inode*, struct file* );
static int dm510_release( struct inode*, struct file* );
static ssize_t dm510_read( struct file*, char*, size_t, loff_t* );
static ssize_t dm510_write( struct file*, const char*, size_t, loff_t* );
long dm510_ioctl(struct file *filp, unsigned int cmd, unsigned long arg);

#define DEVICE_NAME "dm510_dev" /* Dev name as it appears in /proc/devices */
#define MAJOR_NUMBER 254
#define MIN_MINOR_NUMBER 0
#define MAX_MINOR_NUMBER 1

#define DEVICE_COUNT 2
#define BUFFER_SIZE 2000
#define PROC_NUMBER 1 /* number of processes allowed to read */

/*
 * Ioctl definitions
 */

#define DM510_IOC_MAGIC  82

#define DM510_IOCBUFFERSET    _IO(DM510_IOC_MAGIC, 0)
```

```
#define DM510_IOCBUFFERGET    _IO(DM510_IOC_MAGIC, 1)
#define DM510_IOCPROCSET      _IO(DM510_IOC_MAGIC, 2)
#define DM510_IOCPROCGET      _IO(DM510_IOC_MAGIC, 3)


#define DM510_IOC_MAXNR 4
/* end of what really should have been in a .h file */

/* structure for our char buffers */
struct my_buffer
{
    char *buffer, *end;           /* pointers to the beginning and end of the
buffer */
    int buffersize;
    int count;                    /* keeps track of how far into the buffer
we are */
    int rp, wp;
    int numreaders, numwriters;   /* number of readers and writers */
    struct semaphore sem;
    wait_queue_head_t queue;      /* queue of sleeping processes waiting for
access to buffer */
};

static struct my_buffer *buffers;

/* structure for our device */
struct my_device {
  struct my_buffer *rbuf, *wbuf;  /* pointers to the read-buffer and the
write-buffer */
  struct semaphore sem;
  struct cdev cdev;
};

static struct my_device *dm510_device;

/*other variables */
int ret;                            /* return value, for error checking */
int dev_buffer_size = BUFFER_SIZE;
int dev_number_proc = PROC_NUMBER;

/* file operations struct */
static struct file_operations dm510_fops = {
            .owner    = THIS_MODULE,
            .read     = dm510_read,
            .write    = dm510_write,
            .open     = dm510_open,
            .release = dm510_release,
        .unlocked_ioctl   = dm510_ioctl
};

static void setup_cdev(struct my_device *dev, int count) {
  cdev_init(&dev->cdev, &dm510_fops);
  dev->cdev.owner = THIS_MODULE;
  dev->cdev.ops = &dm510_fops;

  int devno = MKDEV(MAJOR_NUMBER,MIN_MINOR_NUMBER+count);
```

```
  ret = cdev_add(&dev->cdev, devno, 1);
  if (ret) {
    printk(KERN_ALERT "DM510: Unable to add cdev to kernel.\n");
    return ret;
  }
}

/* called when module is loaded */
int dm510_init_module( void ) {
  int i;
      /* initialization code belongs here */
  printk("About to init_module\n");

      ret = register_chrdev_region(MAJOR_NUMBER,DEVICE_COUNT,DEVICE_NAME);
  if (ret < 0) {
    printk(KERN_ALERT "DM510: Failed to allocate a device region. Error
%d\n", ret);
    return ret;
  }

  /* allocate memory for the devices */
  dm510_device = kmalloc(DEVICE_COUNT*sizeof(struct my_device), GFP_KERNEL);
  if (dm510_device == NULL) {
    printk(KERN_NOTICE "Unable to allocate memory! Out of memory!\n");
    printk("errno = -12 ENOMEM\n");
    unregister_chrdev_region(MAJOR_NUMBER, DEVICE_COUNT);
    return -ENOMEM;
  }
  memset(dm510_device, 0, DEVICE_COUNT*sizeof(struct my_device));

  printk("RegionAlloc finished. About to set up cdev.\n");

  /* allocate memory for the buffers */
  buffers = kmalloc(DEVICE_COUNT*sizeof(struct my_buffer), GFP_KERNEL);
  if (buffers == NULL) {
    printk(KERN_NOTICE "Unable to allocate memory! Out of memory!\n");
    printk("errno = -12 ENOMEM\n");
    return -ENOMEM;
  }

  for (i = 0; i < DEVICE_COUNT; i++) {
    setup_cdev(&dm510_device[i], i);
    buffers[i].buffer = kmalloc(dev_buffer_size, GFP_KERNEL);
    if (!buffers[i].buffer) {
      printk(KERN_NOTICE "Unable to allocate memory! Out of memory!\n");
      printk("errno = -12 ENOMEM\n");
      return -ENOMEM;
    }
    buffers[i].count = 0;
    buffers[i].wp = 0;
    buffers[i].rp = 0;
    buffers[i].numwriters = 0;
    buffers[i].numreaders = 0;
    buffers[i].buffersize = dev_buffer_size;
    buffers[i].end = buffers[i].buffer + buffers[i].buffersize;
    init_waitqueue_head(&buffers[i].queue);
    init_MUTEX(&buffers[i].sem);
```

```
    init_MUTEX(&dm510_device[i].sem);
  }

  dm510_device[0].rbuf = dm510_device[1].wbuf = &buffers[0];
  dm510_device[1].rbuf = dm510_device[0].wbuf = &buffers[1];

      printk(KERN_INFO "DM510: Hello from your device!\n");
      return 0;
}

/* Called when module is unloaded */
void dm510_cleanup_module( void ) {

  int i;

  if(!dm510_device) {
    return;
  }
  for(i = 0; i<DEVICE_COUNT; i++) {
    cdev_del(&dm510_device[i].cdev);
    kfree(buffers[i].buffer);
    kfree(buffers);
  }
  kfree(dm510_device);
  unregister_chrdev_region(MAJOR_NUMBER, DEVICE_COUNT);
  dm510_device = NULL;

      printk(KERN_INFO "DM510: Module unloaded.\n");
}


/* Called when a process tries to open the device file */
static int dm510_open( struct inode *inode, struct file *filp ) {

      /* device claiming code belongs here */
      struct my_device *dev; /* device information */

      dev = container_of(inode->i_cdev, struct my_device, cdev);

      if(down_interruptible(&dev->sem)) {
        printk(KERN_ALERT "DM510: Could not lock device during open.\n");
        return -ERESTARTSYS;
      }

      filp->private_data = dev; /* for other methods */

      filp->private_data = dev;
      if(filp->f_mode & FMODE_READ) {
        dev->rbuf->numreaders++;
        if(dev->rbuf->numreaders > dev_number_proc) {
          return -EACCES;
        }
      }
      if(filp->f_mode & FMODE_WRITE) {
        dev->wbuf->numwriters++;
        if(dev->wbuf->numwriters > 1) {
          return -EACCES;
```

```
        }
      }

  printk(KERN_INFO "DM510: Device opened.\n");

      return nonseekable_open(inode,filp);
}


/* Called when a process closes the device file. */
static int dm510_release( struct inode *inode, struct file *filp ) {

      struct my_device *dev = filp->private_data;

      if (filp->f_mode & FMODE_READ) {
        dev->rbuf->numreaders--;
      }
      if (filp->f_mode & FMODE_WRITE) {
        dev->wbuf->numwriters--;
      }
      up(&dev->sem);

      printk(KERN_INFO "DM510: Released device.\n");

      return 0;
}


/* Called when a process, which already opened the dev file, attempts to read
from it. */
static ssize_t dm510_read( struct file *filp,
    char *buf,      /* The buffer to fill with data     */
    size_t count,   /* The max number of bytes to read  */
    loff_t *f_pos )  /* The offset in the file           */
{

      int data = 0;
      struct my_device *dev = filp->private_data;


      if(down_interruptible(&(dev->rbuf->sem))) {
        return -ERESTARTSYS;
      }

      while (dev->rbuf->count == 0) {
        /* buffer is empty, prepare for sleep */
        up(&(dev->rbuf->sem));
        wake_up_interruptible(&(dev->rbuf->queue));
        if (filp->f_flags & O_NONBLOCK) {
          return -EAGAIN;
        }
        /* Sleepy time! Put process to sleep and wait for more data. */
        if (wait_event_interruptible(dev->rbuf->queue, (dev->rbuf->count !=
0))) {
          return -ERESTARTSYS;
        }
        /* reacquire lock and loop */
```

```
        if(down_interruptible(&(dev->rbuf->sem))) {
        return -ERESTARTSYS;
        }
      }
      while((data < count) && dev->rbuf->count > 0) {
        /* loop and read until the specified amount is read or the buffer is
empty, one character at a time */
    ret = copy_to_user(buf+data,dev->rbuf->buffer+dev->rbuf->rp,1);
      if (ret) {
        up(&(dev->rbuf->sem));
        return -EFAULT;
      }
      data++;
      dev->rbuf->rp++;
      dev->rbuf->count--;
      if(dev->rbuf->rp == dev_buffer_size) {
        dev->rbuf->rp = 0;
      }
      }

      up(&(dev->rbuf->sem));
      /* Data has been cleared from buffer! Wake up any sleepy processes */
      wake_up_interruptible(&(dev->rbuf->queue));

      return count;
}

/* Called when a process writes to dev file */
static ssize_t dm510_write( struct file *filp,
    const char *buf,/* The buffer to get data from      */
    size_t count,   /* The max number of bytes to write */
    loff_t *f_pos )  /* The offset in the file           */
{

      int data = 0;
      struct my_device *dev = filp->private_data;

      if(down_interruptible(&(dev->wbuf->sem))) {
        return -ERESTARTSYS;
      }

      while (data < count) {
        /* not all data is written to buffer yet */
        while(dev->wbuf->count == dev_buffer_size) {
          /* buffer is full, prepare for sleep */
          up(&(dev->wbuf->sem));
          /* data is in buffer, wake up any sleepy processes */
          wake_up_interruptible(&(dev->wbuf->queue));

          if(filp->f_flags & O_NONBLOCK) {
            return -EAGAIN;
          }
          /* Sleepy time! Put process to sleep and wait for more space in
buffer. */
          if(wait_event_interruptible(dev->wbuf->queue, (dev->wbuf->count <
dev_buffer_size))) {
            return -ERESTARTSYS;
```

```
            }
            /* reacquire lock and loop */
            if (down_interruptible(&(dev->wbuf->sem))) {
              return -ERESTARTSYS;
            }
          }
          /* loop and write until there's no more data or no more space, one
character at a time */
          ret = copy_from_user(dev->wbuf->buffer + dev->wbuf->wp, buf + data,
1);
        if(ret) {
          up(&(dev->wbuf->sem));
          return -EFAULT;
        }
        data++;
        dev->wbuf->wp++;
        dev->wbuf->count++;
        if(dev->wbuf->wp == dev_buffer_size) {
          dev->wbuf->wp = 0;
        }
        }
        up(&(dev->wbuf->sem));
        /* data has been written to buffer, wake up any sleepy processes */
        wake_up_interruptible(&dev->wbuf->queue);

        return data;
}

/* called by system call icotl */
long dm510_ioctl(
    struct file *filp,
    unsigned int cmd,   /* command passed from the user */
    unsigned long arg ) /* argument of the command */
{
        /* ioctl code belongs here */
        printk(KERN_INFO "DM510: ioctl called.\n");
        int i;
        int err = 0;
        if (_IOC_TYPE(cmd) != DM510_IOC_MAGIC) return -ENOTTY;
        if (_IOC_NR(cmd) > DM510_IOC_MAXNR) return -ENOTTY;
        /*
         * the direction is a bitmask, and VERIFY_WRITE catches R/W
         * transfers. `Type' is user-oriented, while
         * access_ok is kernel-oriented, so the concept of "read" and
         * "write" is reversed
         */
        if (_IOC_DIR(cmd) & _IOC_READ)
              err = !access_ok(VERIFY_WRITE, (void __user *)arg,
_IOC_SIZE(cmd));
        else if (_IOC_DIR(cmd) & _IOC_WRITE)
              err =  !access_ok(VERIFY_READ, (void __user *)arg,
_IOC_SIZE(cmd));
        if (err) return -EFAULT;

        switch(cmd) {

      case DM510_IOCBUFFERSET:
```

```
      dev_buffer_size = arg;
      for (i = 0; i<DEVICE_COUNT; i++) {
        kfree(buffers[i].buffer);
        buffers[i].buffersize = dev_buffer_size;
        buffers[i].buffer = kmalloc(dev_buffer_size, GFP_KERNEL);
        if(!buffers[i].buffer) {
          printk(KERN_NOTICE "Unable to allocate memory! Out of memory!\n");
          printk("errno = -12 ENOMEM\n");
          return -ENOMEM;
        }
      }
      break;

    case DM510_IOCBUFFERGET:
      return dev_buffer_size;

    case DM510_IOCPROCSET:
      dev_number_proc = arg;
      break;

            case DM510_IOCPROCGET:
      return dev_number_proc;

            default:
              return -ENOTTY;
      }
      //return retval;

      return 0; //has to be changed
}

module_init( dm510_init_module );
module_exit( dm510_cleanup_module );

MODULE_AUTHOR( "Torvald Johnson" );
MODULE_LICENSE("GPL");
```

7.2 Appendix II: Source code for moduletest2.c

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

#define DM510_IOC_MAGIC  82

#define DM510_IOCBUFFERSET    _IO(DM510_IOC_MAGIC, 0)
#define DM510_IOCBUFFERGET    _IO(DM510_IOC_MAGIC, 1)
#define DM510_IOCPROCSET      _IO(DM510_IOC_MAGIC, 2)
```

```
#define DM510_IOCPROCGET      _IO(DM510_IOC_MAGIC, 3)

#define DM510_IOC_MAXNR 4

int test1(int fd);
int test2(int fd);
int test3(int fd, int rd);
int test4(int fd, int rd);
int test5(int fd);

int main(int argc, char** argv) {

  int res = 0;
  int choice;



  int fd = open("/dev/dm510-0", O_RDWR);
  int rd = open("/dev/dm510-1", O_RDWR);
      perror("w open");

  printf("Which test would you like to run? \n");
  scanf(" %i", &choice);
  printf("Running test %i.\n", choice);

  if(choice == 1) {
      res = test1(fd);
  }

  if(choice == 2) {
      res = test2(fd);
  }

  if(choice == 3) {
      res = test3(fd, rd);
  }

  if(choice == 4) {
      res = test4(fd, rd);
  }

  if(choice == 5) {
      res = test5(fd);
  }

return res;
}

int test1(int fd){
  int buf = 22;

  printf("This test will attempt to write a message in which the number of
bytes to write is incorrectly specified as a negative value. \n");

  /* Write */
  int ret;
        ret = write(fd, &buf, -1);
```

```
        if (ret == -1) {
            perror("write");
            exit(1);
        }

  return ret;
}

int test2(int fd){
  int buf = NULL;

  printf("This test will attempt to write a message, passing a NULL value as
the buffer parameter \n");

  /* Write */
  int ret;
        ret = write(fd, NULL, 5);
        if (ret == -1) {
            perror("write");
            exit(1);
        }

  return ret;
}

int test3(int fd, int rd){
  int buf = 1;

  printf("This test will first write a message with one device, then attempt
to read that message with another device, passing a NULL value as the buffer
parameter \n");

  /* Write */
  int ret;
  ret = write(fd, &buf, 5);
        if (ret == -1) {
            perror("write");
            exit(1);
        }

  /* Read */
  ret = read(rd, NULL, 5);
        if (ret == -1) {
            perror("read");
            exit(1);
        }

  return ret;
}

int test4(int fd, int rd){
  int buf = 123;
  int retbuf;

  printf("This test will simply attempt to write a message with one device,
and then read using another device, using acceptable data as parameters.
\n");
```

```
  /* Write */
  int ret;
  ret = write(fd, &buf, 4);
        if (ret == -1) {
            perror("write");
            exit(1);
        }

  /* Read */
  ret = read(rd, &retbuf, 4);
        if (ret == -1) {
            perror("read");
            exit(1);
        }
  return ret;
}

int test5(int fd){

  printf("Now the ioctl function will be tested, which includes the
functionality to get and set the buffer size, as well as get and set the
number of allowed read processes.  \n");
  printf("\n");
  printf("Running ioctl buffer tests.\n");
  printf("\n");
  printf("Using ioctl to get the original buffersize. Current buffersize is:
%d.\n", ioctl(fd,DM510_IOCBUFFERGET));
  printf("Using ioctl to set a new buffersize. \n");
  ioctl(fd,DM510_IOCBUFFERSET,5000);
  printf("The new buffersize is: %d,\n", ioctl(fd,DM510_IOCBUFFERGET));

  printf("\n");
  printf("Running ioctl allowed-read-processes tests.\n");
  printf("Using ioctl to get the original number of allowed read processes.
The current number of allowed read processes is: %d,\n",
ioctl(fd,DM510_IOCPROCGET));
  printf("Using ioctl to set a new amount of allowed read processes. \n");
  ioctl(fd,DM510_IOCPROCSET,50);
  printf("The new amount of allowed read processes is: %d,\n",
ioctl(fd,DM510_IOCPROCGET));

  return 0;


}
```