dm510 project 3

Danny Jensen
danje14

Group:
Lea Fog-Fredsgaard
Michelle Dung Hoang

April 2018

1 Introduction

In this project a small driver is to be implemented, using the *scull pipe.c* as a reference. since much of the code is basically copy paste from *scull pipe.c* this rapport will focus on the different places.

2 Design

As *pipe.c* already has a working device it is used as a base and then simply split the struct up into a device struct and a buffer struct. For example the pointers to the places that can be read and wrote from was first designed to be in the devices, but then a problem would occur because device 0 would have a write pointer to buffer 1 buffer and a read pointer to buffer 0 buffer. But then a problem would occur because it was not possible to compare these two pointers because device 0 and device 1 could not talk together. So the pointers was then to be placed in the buffers instead, which made it all easier.
Now that the structs are designed, the init has no real design choices everything just needs to initialized. The only design choices is what needs to be returned if kmalloc fails, and that is that there is not enough memory. And if $cdev_add$ fails it just says try again, there is a very small chance it fails so if it does the process should just try again.
In the open function there needs to be a check that says if there are to many processes trying to open, then only the number of processes that was defined can have access and the rest gets an error saying tat they need to try again. the most important part here is that there is at least one designated spot for a reader, so that there wont be a deadlock where all the processes are readers that just waits on input but there are no space for a writer.
In the read function there needs to be a way that moves the read pointer in the buffer and the way it needs to move it is by simply moving it the number of bytes that was read.
The write function needs to do the same as the read, just with the write pointer in the buffer.

3 Implementation

First off the buffer struct and the device struct is made splitting most of the *pipe.c* device struct up. The buffer gets the wait ques because it makes sense that it is the buffer that wakes up the next process that can either write or read to it. The buffer also needs some pointer for the start and the end of the buffer. The buffer also keeps track of how many processes are interacting with it, the numbers are stored in nreaders and nwriters. The last thing that the buffer have are read and write pointers, to point where the write function can start writing and the read function can start reading.

```
struct dm510_buffer {
  wait_queue_head_t inq, outq; /* read and write queues */
  char *buffer, *end;          /* begin of buf, end of buf */
  int nreaders, nwriters;      /* number of openings for r/w */
  char *buffer_rp, *buffer_wp; /* where to read, where to write */
};
```

Now the device struct needs pointers to the buffers it writes from an reads from. It also holds the *cdev* struct. And ofcause a mutex lock which locks the device so it cant be interrupted.

```
struct dm510_dev {
  struct dm510_buffer *read_buffer;  /*pointer to buffer for
    reading*/
  struct dm510_buffer *write_buffer; /*pointer to buffer for
    writing*/
  struct cdev cdev;                  /* Char device structure */
  struct mutex mutex;                /* mutual exclusion semaphore
    */
};
```

Now in the init function *kmalloc* is used to allocate space in the kernel for both devices and the buffers and the buffer space. Of cause kmalloc can fail and the an error is returned, and if anything had successfully gotten allocated some space it needs to be freed so as not to get dead parts of the memory.

```
dm510_dev_1 = kmalloc(sizeof(struct dm510_dev), GFP_KERNEL);
if (!dm510_dev_1){
  kfree(dm510_dev_0);
  return -ENOMEM;
}
```

Then all the integers is set to the right stating value and pointers are set to the right places. (This was actually something we forgot to do with the read and write pointers and it took us almost an entire day to figure that mistake out. Lesson learned :))

```
dm510_buffer_0->nwriters = 0;
dm510_buffer_0->buffer_rp = dm510_buffer_0->buffer;
```

The wait queues, the mutex locks and the cdev needs to be initialized for later use to.

```
  init_waitqueue_head(&dm510_buffer_0->inq);

  mutex_init(&dm510_dev_0->mutex);

  cdev_init(&(dm510_dev_0->cdev), &dm510_fops);
```

Ofcause the owner in cdev also needs to be set.

```
  dm510_dev_0->cdev.owner = THIS_MODULE;
```

The release function must undo everything the init function has created, and it does so in revers order, starting with the last thing init did. It is done in revers so as to not delete any pointers that should be used to find

```
if (!dev_holder){
  return;
}
/* cleans up in reverse order */
cdev_del(&dm510_dev_1->cdev);   /* removes devices from the dev_t holder */
cdev_del(&dm510_dev_0->cdev);
kfree(dm510_buffer_1->buffer);  /*free content of buffer*/
kfree(dm510_buffer_0->buffer);
kfree(dm510_buffer_1);      /*free buffer*/
kfree(dm510_buffer_0);
kfree(dm510_dev_1);       /*free device*/
kfree(dm510_dev_0);

/* removes the dev_t place holder */
unregister_chrdev_region(dev_holder, DEVICE_COUNT);
```

Now in the open function there is a check build in that ensures that the number of processes dont get larger that specified and also that a deadlock doesn't happen where there is max number of processes running and they all are readers. This would mean that all the processes sleep an one process waits for something to be written but there is no space for a writer. This check is a if statement where if the max number of processes is running the new process gets an error stating to try again, and hopefully by the time the processes reties some other process got done and a space opened up.

```
if (dev->read_buffer->nreaders >= number_proc-1 && (filp->f_mode & FMODE_READ)) {
  mutex_unlock(&dev->mutex);
  return -EAGAIN;
}else if (dev->read_buffer->nreaders + dev->write_buffer->nwriters >= number_proc){ /* if there is a writer and max # of processes is running */
  mutex_unlock(&dev->mutex);
  return -EAGAIN;
```

now if theres already a writing processes running the new writing processes is put to sleep and the old writing process will wake a new writing process when done.

```
dev->write_buffer->nwriters++;
/* if there are one writer allready the rest is put to sleep */
if(wait_event_interruptible(dev->write_buffer->inq, (dev->write_buffer->nwriters >= 1))){
  mutex_unlock(&dev->mutex);
  return -ERESTARTSYS;
}
```

In read an *access_ok* is done to ensure the user space pointer is valid, if not an error is returned and the processes stopped.

```
/* if access is not ok return error */
if (!access_ok(VERIFY_WRITE, buf, count)){
  mutex_unlock (&dev->mutex);
  return -EACCES;
}
```

Now if the user space pointer is ok, the *copy_to_user* gets the requested number of bytes and puts them into user space. Then the read pointer is moved to the right place and if the right place was at the end of the buffer the read pointer is moved to the start of the buffer again.

```
/* copies to user and the remaining # of bytes not copied is stored in remaining */
remaining = copy_to_user(buf, dev->read_buffer->buffer_rp, count);

/* moving the read pointer to the next non read data */
dev->read_buffer->buffer_rp += count;

/* move read pointer from end to start */
if (dev->read_buffer->buffer_rp >= dev->read_buffer->end){
  dev->read_buffer->buffer_rp = dev->read_buffer->buffer; /* wrapped */
}
```

The write function does the exact same same thing just instead of *copy_to_user* it uses *copy_from_user* and the int variable *remaining* is used to subtract from the count to ensure that a read process does not read garbage.

```
dev->write_buffer->buffer_wp += count - remaining;
```

Also the *remaining* variable is also used to subtract from the count in the return value so as to know how much actually was written to the buffer.

```
  return count - remaining; //return number of bytes written
```

Now for the IO control, the two cases are used to either change the buffer size or to change the max number of processes.
The first case $IOC_RESETBUFFER$first ensures that the argument handed to it is not 0 or negative, because it would be impossible to allocate negative space and it makes no sense trying to change the buffer size to 0.
If the argument is larger than 0, the two buffers are freed and two new spaces are allocated for the buffers, then the different pointer like end and the read and write pointers are reset, if the read and write pointers are not reset they would point to the old places in memory and that would be a problem, it would mean that a process would write into another process' memory and maybe destroy that process. And if a process read from the old pointer it would only get garbage.
Now that the buffer sizes has been changed and the pointer set at the right places the new buffer size is returned.

```
case IOC_RESETBUFFER:
  buffer_size = arg;
  if (buffer_size <= 0) {
    return -EINVAL;
  }
  kfree(dm510_buffer_1->buffer);
  kfree(dm510_buffer_0->buffer);
  dm510_buffer_0->buffer = kmalloc(sizeof(char*) *buffer_size, GFP_KERNEL);
  if (!dm510_buffer_0->buffer){
    return -ENOMEM;
  }
  dm510_buffer_1->buffer = kmalloc(sizeof(char*) *buffer_size ,
    GFP_KERNEL);
  if (!dm510_buffer_1->buffer){
    return -ENOMEM;
  }

  dm510_buffer_0->end = dm510_buffer_0->buffer + buffer_size;
  dm510_buffer_1->end = dm510_buffer_1->buffer + buffer_size;

  dm510_buffer_0->buffer_rp = dm510_buffer_0->buffer;
  dm510_buffer_0->buffer_wp = dm510_buffer_0->buffer;

  dm510_buffer_1->buffer_rp = dm510_buffer_1->buffer;
  dm510_buffer_1->buffer_wp = dm510_buffer_1->buffer;

  return buffer_size;
```

$IOC_RESETPROC$ resets the max number of processes and this one is quite easy, it simply asures that there can be at least 2 processes so that it is assured that a writer and a reader can be at the same time. If this check was set to at least 1 then if the max processes was set to 1 there would be no space for a read process because of the check in the open function.
Now if the argument passed onto $IOC_RESETPROC$ is valid, $number_proc$ is simply changed to the new value

```
case IOC_RESETPROC:
  if (arg <=1) {
    return -EINVAL;
  }
  number_proc = arg;
  return number_proc;
```

4 Test

The test was video captured so it will be refereed to with time stamps here instead of showing pictures.
For the first test is to show that *dm510_load* works, this is shown at 0.36 in the video, and it loads just fine.
The next test is then *moduletest.c*, as shown at 0.41, this test prints out 2 tests and what the expected result should be in these 2 tests. And as shown the first test returns -930547960 as it should. The second returned -830547960, this is also equal to the expected value so from this it is concluded that moduletest was passed successfully.
Then just to be sure the devices is unloaded and loaded back again, this is done so the next test is started with everything at the starting points.
Now to test the ioctl, the first test is to change te buffer size, this happens at 0.55 in video. The new buffer size is set to 500 and the new buffer size is printed out just to be sure that it was changed.
The last ioctl test is to test if max number of processes can be changed, this

happens at 1.00. And just like the buffer size change the max number of processes value is also printed out so that it is clear that it was changed.
Now since the buffer size and max number of processes got tampered with the devices are unloaded and loaded back again for the last test.
The last test is started at 1.12 and the first test is to write to device 0 and read the just inserted text from device 1, the result is written out at 1.14. A simple ”hello” string was written and read back to the user which is a success. The buffer size was changed to 15 for the last test.
Now device 1 is written to and device 0 is read from, this happen at 1.15. This time the string was ”world” just to show that it is a new string and not the old that was returned again.
So now both devices was checked with both write and read functions and passed this test to.
The last test is to show that the write function wont write over data that has not yet been read. This test is at 1.16 in the video. As shown the writer tries to insert 15 bytes into the buffer, but since the previous test inserted 5 already (which has been read to), there is only space for 10 bytes until the write pointer hits the end of the buffer. The next writer is also trying to insert 15 but the read pointer is standing at 5 so it can only insert 4 so as not to get the write and read pointer standing in the same place. If that would happen it would look like there is nothing to read in the buffer.

5 Discussion

If multiple processes tried to use the devices at the same time they would be put to sleep using *wait_event_interruptible* that one process can do what it needs and when this process is done it should wake another process up from its sleep using *wake_up_interruptible*.
Now if to many processes tries use the device and the max number of processes is reached every other process get an error code returned stating it should try again.

6 Conclusion

All in all this project was successful and fun to work with, even though there where many times where it felt like it was impossible to finish because of some little bug which was impossible to find.
Thanks to this project a new found respect was found for the ones who work with drivers and suc every day.

7 Appendix

7.1 dm510_dev.c

```c

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

#include "ioct.h"
#include <linux/sched.h>
#include <linux/sched/signal.h>

#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/uaccess.h>
#include <linux/semaphore.h>
// #include <asm/system.h>
#include <asm/switch_to.h>
#include <linux/cdev.h>


/* 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 DM510_IOC_MAGIC 9

#define IOC_RESETBUFFER _IO(DM510_IOC_MAGIC, 0)
#define IOC_RESETPROC _IO(DM510_IOC_MAGIC, 1)

#define IOC_NUMCASES 2*/
/* end of what really should have been in a .h file */

/* 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
};

struct dm510_buffer {
  wait_queue_head_t inq, outq;       /* read and write queues */
  char *buffer, *end;                /* begin of buf, end of buf */
  int nreaders, nwriters;            /* number of openings for r/w */
  char *buffer_rp, *buffer_wp;       /* where to read, where to write */
};

struct dm510_dev {
  struct dm510_buffer *read_buffer;   /*pointer to buffer for reading*/
  struct dm510_buffer *write_buffer;  /*pointer to buffer for writing*/
  struct cdev cdev;                   /* Char device structure */
  struct mutex mutex;                 /* mutual exclusion semaphore */
};

/*initialising*/
struct dm510_buffer *dm510_buffer_0;
struct dm510_buffer *dm510_buffer_1;

struct dm510_dev *dm510_dev_0;
struct dm510_dev *dm510_dev_1;

int  buffer_size = 3000;
int number_proc=10;

/*type that is defined and is used to hold device numbers (major, minor)*/
dev_t dev_holder;
/* called when module is loaded */
int dm510_init_module( void ) {

  /* initialization code belongs here */
  int err;
  /*creates the first device*/
  dev_holder = MKDEV(MAJOR_NUMBER,MIN_MINOR_NUMBER);
  /*registration takes a pointer and a name*/
  err = register_chrdev_region(dev_holder,DEVICE_COUNT,DEVICE_NAME);
  if(err != 0){
    printk(KERN_NOTICE "Unable to get region, error %d\n", err);
    return -ENODEV;
  }

  /* allocating space for devices and buffers
   * and freeing already allocated space if error occurs */
  dm510_dev_0 = kmalloc(sizeof(struct dm510_dev), GFP_KERNEL);
  if (!dm510_dev_0){
    return -ENOMEM;
  }
  dm510_dev_1 = kmalloc(sizeof(struct dm510_dev), GFP_KERNEL);
  if (!dm510_dev_1){
    kfree(dm510_dev_0);
    return -ENOMEM;
  }
  dm510_buffer_0 = kmalloc(sizeof(struct dm510_buffer), GFP_KERNEL);
  if (!dm510_buffer_0){
    kfree(dm510_dev_0);
    kfree(dm510_dev_1);
    return -ENOMEM;
  }
  dm510_buffer_1 = kmalloc(sizeof(struct dm510_buffer), GFP_KERNEL);
  if (!dm510_buffer_1){
    kfree(dm510_dev_0);
    kfree(dm510_dev_1);
    kfree(dm510_buffer_0);
    return -ENOMEM;
  }
  /*allocating space for text in buffer*/
  dm510_buffer_0->buffer = kmalloc(sizeof(char*) *buffer_size, GFP_KERNEL);
  if (!dm510_buffer_0->buffer){
    kfree(dm510_dev_0);
    kfree(dm510_dev_1);
    kfree(dm510_buffer_0);
    kfree(dm510_buffer_1);
    return -ENOMEM;
  }
  dm510_buffer_1->buffer = kmalloc(sizeof(char*) *buffer_size, GFP_KERNEL);
  if (!dm510_buffer_1->buffer){
    kfree(dm510_dev_0);
    kfree(dm510_dev_1);
    kfree(dm510_buffer_0->buffer);
    kfree(dm510_buffer_0);
    kfree(dm510_buffer_1);
    return -ENOMEM;
  }
  /* initialising all int's in the buffer to the right size */
  dm510_buffer_0->nreaders = 0;
  dm510_buffer_0->nwriters = 0;
  dm510_buffer_0->buffer_rp = dm510_buffer_0->buffer;
  dm510_buffer_0->buffer_wp = dm510_buffer_0->buffer;

  dm510_buffer_1->nreaders = 0;
  dm510_buffer_1->nwriters = 0;
  dm510_buffer_1->buffer_rp = dm510_buffer_1->buffer;
  dm510_buffer_1->buffer_wp = dm510_buffer_1->buffer;

  dm510_buffer_0->end = dm510_buffer_0->buffer + buffer_size;
  dm510_buffer_1->end = dm510_buffer_1->buffer + buffer_size;

  /* initialize read and write queues */
  init_waitqueue_head(&dm510_buffer_0->inq);
  init_waitqueue_head(&dm510_buffer_0->outq);
  init_waitqueue_head(&dm510_buffer_1->inq);
  init_waitqueue_head(&dm510_buffer_1->outq);

  /* initialise the mutex locks */
  mutex_init(&dm510_dev_0->mutex);
  mutex_init(&dm510_dev_1->mutex);

  /* initialize a cdev structure */
  cdev_init(&(dm510_dev_0->cdev), &dm510_fops);
  cdev_init(&(dm510_dev_1->cdev), &dm510_fops);

  /* setting the owner */
  dm510_dev_0->cdev.owner = THIS_MODULE;
  dm510_dev_1->cdev.owner = THIS_MODULE;

  /* setting the right read and write buffers in the devices */
  dm510_dev_0->read_buffer = dm510_buffer_0;
  dm510_dev_0->write_buffer = dm510_buffer_1;
  dm510_dev_1->read_buffer = dm510_buffer_1;
  dm510_dev_1->write_buffer = dm510_buffer_0;

  /* add the devices to the dev_t place holder */
  err = cdev_add(&dm510_dev_0->cdev,dev_holder,1);
  if (err != 0) {
    printk(KERN_NOTICE "Error %d adding cdev", err);
    return -EAGAIN;
  }

    err = cdev_add(&dm510_dev_1->cdev,dev_holder+1,1);
  if (err != 0) {
    printk(KERN_NOTICE "Error %d adding cdev", err);
    return -EAGAIN;
  }

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

/* Called when module is unloaded */
void dm510_cleanup_module( void ) {
  /* clean up code belongs here */

  if (!dev_holder){
    return;
  }
  /* cleans up in reverse order */
  cdev_del(&dm510_dev_1->cdev);   /* removes devices from the dev_t holder */
  cdev_del(&dm510_dev_0->cdev);
  kfree(dm510_buffer_1->buffer);  /*free content of buffer*/
```

```c
  kfree(dm510_buffer_0->buffer);
  kfree(dm510_buffer_1);           /*free buffer*/
  kfree(dm510_buffer_0);
  kfree(dm510_dev_1);              /*free device*/
  kfree(dm510_dev_0);

  /* removes the dev_t place holder */
  unregister_chrdev_region(dev_holder, DEVICE_COUNT);

  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 dm510_dev *dev;

  /* puts the devices into filp */
  dev = container_of(inode->i_cdev, struct dm510_dev, cdev);
  filp->private_data = dev;

  /* locks the process */
  if (mutex_lock_interruptible(&dev->mutex)){
    return -ERESTARTSYS;
  }

  /* make checks to ensure that number of processes is kept */
  if (dev->read_buffer->nreaders >= number_proc-1 && (filp->f_mode & FMODE_READ)) {
    mutex_unlock(&dev->mutex);
    return -EAGAIN;
  }else if (dev->read_buffer->nreaders + dev->write_buffer->nwriters >= number_proc){ /* if there is a writer and max # of processes is running */
    mutex_unlock(&dev->mutex);
    return -EAGAIN;
  }else{  /* if its either a write, or there are fewer than max processes */
    if (filp->f_mode & FMODE_READ){
      dev->read_buffer->nreaders++;
    }
    if (filp->f_mode & FMODE_WRITE){
      if (filp->f_flags & O_NONBLOCK) {
        mutex_unlock(&dev->mutex);
        return -EAGAIN;
      }
      dev->write_buffer->nwriters++;
      /* if there are one writer allready the rest is put to sleep */
      if(wait_event_interruptible(dev->write_buffer->inq, (dev->write_buffer->nwriters >= 1))){
        mutex_unlock(&dev->mutex);
        return -ERESTARTSYS;
      }
    }
  }
  mutex_unlock(&dev->mutex);

  return 0;
}

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

  struct dm510_dev *dev = filp->private_data;

  mutex_lock(&dev->mutex);
  if (filp->f_mode & FMODE_READ){
    dev->read_buffer->nreaders--;
  }
  if (filp->f_mode & FMODE_WRITE){
    dev->write_buffer->nwriters--;
  }
  mutex_unlock(&dev->mutex);

  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           */
{
  /* read code belongs here */

  struct dm510_dev *dev = filp->private_data;

  int remaining;

  if (mutex_lock_interruptible(&dev->mutex)){
    return -ERESTARTSYS;
  }
  while (dev->read_buffer->buffer_rp == dev->read_buffer->buffer_wp) { /* nothing to read */
    mutex_unlock(&dev->mutex); /* release the lock */
    if (filp->f_flags & O_NONBLOCK){
      return -EAGAIN;
    }
    if (wait_event_interruptible(dev->read_buffer->inq, (dev->read_buffer->buffer_rp != dev->read_buffer->buffer_wp))){
      return -ERESTARTSYS; /* signal: tell the fs layer to handle it */
    }
    /* otherwise loop, but first reacquire the lock */
    if (mutex_lock_interruptible(&dev->mutex)){
      return -ERESTARTSYS;
    }
  }

  /* ok, data is there, return something */
  if (dev->read_buffer->buffer_wp > dev->read_buffer->buffer_rp){
    count = min(count, (size_t)(dev->read_buffer->buffer_wp - dev->read_buffer->buffer_rp));
  }else{ /* the write pointer has wrapped, return data up to dev->end */
    count = min(count, (size_t)(dev->read_buffer->end - dev->read_buffer->buffer_rp));
  }

  /* if access is not ok return error */
  if (!access_ok(VERIFY_WRITE, buf, count)){
    mutex_unlock (&dev->mutex);
    return -EACCES;
  }

  /* copies to user and the remaining # of bytes not copied is stored in remaining */
  remaining = copy_to_user(buf, dev->read_buffer->buffer_rp, count);

  /* moving the read pointer to the next non read data */
  dev->read_buffer->buffer_rp += count;

  /* move read pointer from end to start */
  if (dev->read_buffer->buffer_rp >= dev->read_buffer->end){
    dev->read_buffer->buffer_rp = dev->read_buffer->buffer; /* wrapped */
  }
  mutex_unlock (&dev->mutex);

  /* finally, awake any writers and return */
  wake_up_interruptible(&dev->read_buffer->outq);

  /* return number of bytes read */
  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            */
{
  /* write code belongs here */
  struct dm510_dev *dev = filp->private_data;
  int remaining;

  /* if trying to read bytes under 1 */
  if (count < 1)
  {
    return -EINVAL;
  }

  if (mutex_lock_interruptible(&dev->mutex)){
    return -ERESTARTSYS;
  }

  if (dev->write_buffer->buffer_wp >= dev->write_buffer->buffer_rp) {
    count = min(count, (size_t)(dev->write_buffer->end - dev->write_buffer->buffer_wp)); /* to end-of-buf */
  }else{ /* the write pointer has wrapped, fill up to rp-1 */
    count = min(count, (size_t)(dev->write_buffer->buffer_rp - dev->write_buffer->buffer_wp - 1));
  }

  if (!access_ok(VERIFY_WRITE, buf, count)){
    mutex_unlock (&dev->mutex);
    return -EACCES;
  }

  remaining = copy_from_user((dev->write_buffer->buffer_wp), buf, count);

  dev->write_buffer->buffer_wp += count - remaining;

  if (dev->write_buffer->buffer_wp >= dev->write_buffer->end){
    dev->write_buffer->buffer_wp = dev->write_buffer->buffer; /* wrapped */
  }
  mutex_unlock(&dev->mutex);
  //dev->write_buffer->buffer_wp = dev->write_p;

  wake_up_interruptible(&dev->write_buffer->inq);  /* blocked in read() and select() */

  return count - remaining; //return number of bytes written
}

/* 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");

  if (_IOC_TYPE(cmd) != DM510_IOC_MAGIC){
    return -ENOTTY;
  }
  if (_IOC_NR(cmd) > IOC_NUMCASES){
    return -ENOTTY;
  }

  switch(cmd){

    case IOC_RESETBUFFER:
      buffer_size = arg;
      if (buffer_size <= 0) {
        return -EINVAL;
      }
      kfree(dm510_buffer_1->buffer);
      kfree(dm510_buffer_0->buffer);
```

```c
      dm510_buffer_0->buffer = kmalloc(sizeof(char*) *buffer_size, GFP_KERNEL);
      if (!dm510_buffer_0->buffer){
        return -ENOMEM;
      }
      dm510_buffer_1->buffer = kmalloc(sizeof(char*) *buffer_size, GFP_KERNEL);
      if (!dm510_buffer_1->buffer){
        return -ENOMEM;
      }

      dm510_buffer_0->end = dm510_buffer_0->buffer + buffer_size;
      dm510_buffer_1->end = dm510_buffer_1->buffer + buffer_size;

      dm510_buffer_0->buffer_rp = dm510_buffer_0->buffer;
      dm510_buffer_0->buffer_wp = dm510_buffer_0->buffer;

      dm510_buffer_1->buffer_rp = dm510_buffer_1->buffer;
      dm510_buffer_1->buffer_wp = dm510_buffer_1->buffer;

      return buffer_size;

    case IOC_RESETPROC:
      if (arg <=1) {
        return -EINVAL;
      }
      number_proc = arg;
      return number_proc;

    default:
      return -ENOTTY;
  }
  return 0;
}

module_init( dm510_init_module );
module_exit( dm510_cleanup_module );

MODULE_AUTHOR( "Michelle Dung Hoang, Lea Fog-Fredsgaard, Danny Rene Jensen" );
MODULE_LICENSE( "GPL" );
```

7.2 iotest.h

```c
#ifndef IOCTHEADER_h
#define IOCTHEADER_h

#define DM510_IOC_MAGIC 9

#define IOC_RESETBUFFER _IO(DM510_IOC_MAGIC, 0)
#define IOC_RESETPROC _IO(DM510_IOC_MAGIC, 1)

#define IOC_NUMCASES 2

#endif
```

7.3 iotest.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <sys/ioctl.h>
#include "ioct.h"


int main(int argc,char **argv){
  char *filename = argv[1];
  int fd = open(filename, O_RDWR);
  int arg = atoi(argv[3]);
  int result = 0;
  char *IO_arg=argv[2];
  if (!strcmp("buffersize", IO_arg)) {
    result = ioctl(fd, IOC_RESETBUFFER, arg);
    printf("changed buffer size to: %d\n", result);
  }else if (!strcmp("processes", IO_arg)){
    result = ioctl(fd, IOC_RESETPROC, arg);
    printf("changed number of max processes to: %d\n", result);
  }else{
    printf("input error\n");
  }
  close(fd);
}
```

7.4 extratests.c

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






void writing(){
  printf("write to buffer:\n");
  int fd = open("/dev/dm510-0", O_RDWR);
  printf("buffer size changed to %d\n",ioctl(fd, IOC_RESETBUFFER,
    15));

  //close(fd);
  //fd = open(filename, O_RDWR);
  char * fillText = "hello";
  int count = 5, write_result = 0;
  write_result = write(fd, fillText, count);
  printf("the result of insert to buffer 0: %d\n", write_result);
  printf("\n");
  close(fd);
```

```

}

void writing1(){
  printf("write to buffer:\n");
  int fd1 = open("/dev/dm510-1", O_RDWR);
  //printf("buffer size changed to %d\n",ioctl(fd, IOC_RESETBUFFER, 15));

  //close(fd);
  //fd = open(filename, O_RDWR);
  char * fillText1 = "world";
  int count = 5, write_result = 0;
  write_result = write(fd1, fillText1, count);
  printf("the result of insert to buffer 1: %d\n", write_result);

  printf("\n");
  close(fd1);


}




void reading(){
  printf("read from buffer0:\n");
  int fd = open("/dev/dm510-1", O_RDWR);
  int count = 5, read_result = 0;
  char *text;

  read_result = read(fd, text, count);
  printf("result from buffer 0 is %d and line: %s\n", read_result, text);
  printf("\n");
  close(fd);

}


void reading1(){
  printf("read from buffer1:\n");
  int fd = open("/dev/dm510-0", O_RDWR);
  int count = 5, read_result = 0;
  char *text1;

  read_result = read(fd, text1, count);
  printf("result from buffer 0 is %d and line: %s\n", read_result, text1);
  printf("\n");
  close(fd);


}


void fullBuffers(){
```

```c
  printf("buffer is full:\n");
  printf("trying to write 15 bytes to buffer twice:\n");
  int fd = open("/dev/dm510-0", O_RDWR);
  //printf("buffer size changed to %d\n",ioctl(fd, IOC_RESETBUFFER, 15));
  //int fd = open(filename, O_RDWR);
  char * fillText = "theBufferIsFull";
  int count = 15, write_result = 0;

  for (size_t i = 0; i < 2; i++) {
    write_result = write(fd, fillText, count);
    if (write_result < 0) {
      printf("%s \n",strerror(errno));
    }else{
      printf("the result of %d insert: %d\n", i, write_result);
    }
  }
  printf("\n");
  close(fd);
}

void readEmptyBuffers(){
  printf("read empty buffer:\n");
  int fd = open("/dev/dm510-0", O_RDWR);
  int count = 15, read_result = 0;
  char *text;
    read_result = read(fd, text, count);
  printf("result is %d\n", read_result);
  printf("\n");
  close(fd);
}

void writeNothing(){
  printf("write nothing:\n");
  int fd = open("/dev/dm510-0", O_RDWR);
  char * fillText = "";
  int count = 0, write_result = 0;

  printf("buffer size changed to %d\n",count);
  write_result = write(fd, fillText, count);

  if (write_result < 0) {
    printf("%s \n",strerror(errno));
  }else{
    printf("result of writing: %d\n", write_result);
  }
  printf("\n");
  close(fd);
}

void readNothing(){
  printf("read nothing:\n");
  int fd = open("/dev/dm510-0", O_RDWR);
  int count = 0, read_result = 0;
  char *text;
  read_result = read(fd, text, count);
  if (read_result < 0) {
    printf("%s \n",strerror(abs(read_result)));
  }else{
    printf("result is %d\n", read_result);
  }
  printf("\n");
  close(fd);
}


int main (int argc,char **argv){
  writing();
  reading();
  sleep(1);
  writing1();
  reading1();
  sleep(1);
    fullBuffers();

  return 0;
}
```