BLG413E System Programing

Project 2

Yunus Güngör – 150150701

Duhan Cem Karagöz – 150130066

Objective

This project helps to gain experience with Linux environment and drivers in Linux operating systems. Objective of this project is to implement a character device that acts as a message box between users in the system.

Environment

In this project Lubuntu 14.04 with GCC 4.7 is used with C programing language. Program created in this project runs in kernel space as a module.

Task Distribution

Required tasks has been distributed among two members of our team. Yunus Güngör handled with driver installation, driver initialization, message input outputs and message data. Duhan Cem handled with setting up environment for a basic driver, writing test program, ioctl commands and memory cleanup.

Implementation

Implementing a message box requires a data structure to keep messages and users in. In this project we have implemented a general structure for keeping all the data we need. This structure called messagebox_dev and used with device variable in the program. This structure contains, an user array and user array contains a linked list to keep all messages send to that user. Message's sender information kept directly in the message text. For example, if root users send message "hello", this message kept as "root: hello". Data structure can be seen at figure 1. Also, user array used as quantum array and messages in message lists used as quantums.

```c
struct message {
  char *data;//message itself with sender information
  int length;//length of message to prevent overflow
  struct message *next;//next message
};

struct m_user {
  char *name;//user name of user
  struct message *messages; //pointer to message list
  int message_r; //how many messages have been read
  int message_l; //how many characters should be displayed
  int message_s; //how many messages this user has
  int name_size; //size of username to prevent overflow
};

struct messagebox_dev {
    struct m_user **data; //pointer to user array
    unsigned long size; //size (required for devices)
    struct semaphore sem; //semaphore of driver (required for devices)
    struct cdev cdev; //cdev structure (required for devices)
    int max_messages; //maximum messages that user can read (can be set with iotcl commands)
    bool inc_read; //should we include read message to output (can be set with iotcl commands)
    int user_count; //how many users system has
    int max_user; //maximum number of users
    char *tmp; //message to display
    int tmpLength; //length of message
    int current_user_no; //user no of the active user
    char current_user_name[33]; // username of the active user
    int current_user_name_l; //length of username
};

struct messagebox_dev device;
```

Figure 1-Data Structure

After setting required information like major and minor numbers, required commands to initialize module has been send to operating system. Additionally, required commands for creating device and device node has been send to the operating system to provide automated creation of device. Setup process can be examined in figure 2 and figure 3.

```
int messagebox_init_module(void)
{
    printk(KERN_INFO "init messagebox\n");
    int result;
    int i;
    int err;
    dev_t devno = 0;
    struct messagebox_dev *dev;

    if (messagebox_major) {
        devno = MKDEV(messagebox_major, messagebox_minor);
        result = register_chrdev_region(devno, messagebox_nr_devs, "messagebox");
    } else {
        result = alloc_chrdev_region(&devno, messagebox_minor, messagebox_nr_devs,
                                     "messagebox");
        messagebox_major = MAJOR(devno);
    }
    if (result < 0) {
        printk(KERN_WARNING "messagebox: can't get major %d\n", messagebox_major);
        return result;
    }

    printk(KERN_NOTICE "Major: ");
    printk(KERN_NOTICE "%d \n",messagebox_major);
```

Figure 2-Module and device initializing code

```
      c = class_create(THIS_MODULE, DEVICE_NAME);
      dev = &device;
      dev->max_messages=100;
      dev->inc_read=false;
      dev->user_count=0;
      dev->max_user=10;
      sema_init(&dev->sem,1);
      devno = MKDEV(messagebox_major, messagebox_minor);
      cdev_init(&dev->cdev, &messagebox_fops);
      dev->cdev.owner = THIS_MODULE;
      dev->cdev.ops = &messagebox_fops;
      dev->data = NULL;
      err = cdev_add(&dev->cdev, devno, 1);
      if (err) {
        printk(KERN_NOTICE "Error %d adding device%d \n", err, i);
        cdev_del(&dev->cdev);
      }
      d = device_create(c, NULL,  devno, NULL, DEVICE_NAME);
      if (IS_ERR(d)) {
        err = PTR_ERR(d);
        printk(KERN_WARNING "[target] Error %d while adding device %s%d \n",
          err, DEVICE_NAME, messagebox_minor);
          cdev_del(&dev->cdev);
        }
      return 0; /* succeed */
    goto fail;
    fail:
      messagebox_cleanup_module();
      return result;
  }
```

Figure 3 – Device node initializing code

Since all messages has been send with username only, we must implement a mechanism for acquiring username in kernel space. We have achieved that by processing /etc/passwd file and acquiring username section with the matching uid. Uid have been provided by current_uid function in linux. After getting username, we must match user's username within our data structure. To accomplish that program traverses user array until it finds a match. Matching uid, username and user id in our data structure has been done in the open function of the device. This function can be examined in figure 4.

Each time device opened which happens before every write and read process, current user's uid, user name and user id in our data structure has been found and kept in device structure for easy access when write and read functions called. And a temporary string that has user's unread massages or all messages created. Since read function can be called more than once and there is no way to know that open operation acquired for writing or reading, our implementation creates temporary string in open function. This temporary string only used in the read function. Adding unread messages only or all messages to the temporary string decided by inc_read variable which can be set by iotcl commands. Maximum unread messages can read by a user implemented in this step too. This value can also be set by iotcl commands.

Write process invoked by a user using "echo @root message >/dev/messagebox" notation. User must have necessary permissions to successfully write on the device. Write function we have implemented gets message as an input from the operating system and process that input to store message. First of all, message in user space, copied to kernel space with necessary memory allocation. After that message processed to get username which is right after '@' symbol. After acquiring user name, program searches on saved data to find user id in our data. If there is no such user, a new user created. Consider that, active user and user that receiver message are different users. They can be same user without a problem, but data structure we implemented keeps messages by message receiver. After finding user to send message, message added to that user's message list with current user's username and message. Current user's name acquired from the data created in open function. If there is unallocated pointers for writing data, necessary allocation have been made. Write function can be examined in figure 5.

Each user can display own messages by simply using "cat /dev/messagebox". This notation invokes read function. In the read function, we have used tmp pointer in device data structure. This pointer has user's readable messages in a string form. Read function copies necessary characters to the given pointer with copy_to_user function. While coping, global value r_index used for keeping a cursor, and count variable used for how much read function can copy to pointer. Pointer and count variable provided by operating system. Read function can be examined in figure 6.

When module is removed operating system calls cleanup_module function. This function frees all the memory acquired before, removes device node, removes device and terminates the module.

More information can be found in the source code.

Conculision

In this project, we have learned to implement module and driver in linux operating system. Also we have gained experience with concepts like buffer, node and so on. Writing read function and llseek function helped us to understand linux's stream logic. We have implemented memory allocation, file proces, basic data structures in kernel space which helped us gain experience with working in kernel space.

```c
int messagebox_open(struct inode *inode, struct file *filp)
{
    struct messagebox_dev *dev;
    struct message *m;
    int i,length,j,user_no=0;
    char *tmp;
    r_index=0;
    printk(KERN_INFO "Opened");
    dev = container_of(inode->i_cdev, struct messagebox_dev, cdev);
    dev->size=device.max_user*sizeof(struct m_user);
    dev->max_messages = device.max_messages;
    dev->inc_read = device.inc_read;
    dev->user_count = device.user_count;
    dev->max_user = device.max_user;
    dev->data=device.data;
    dev->current_user_name_l=0;

    //get username
    tmp=kmalloc(33 * sizeof(struct m_user *), GFP_KERNEL);
    char *userName=tmp;
    char userIDs[33];
    int k=0;
    while(k<33)
    {
      userName[k]=' ';
      userIDs[k]=' ';
      k++;
    }
    struct file *file;
    mm_segment_t fsegment;
    char c=' ';
    long userID=-1;
    int uid;
    uid=current_uid().val;
    printk(KERN_INFO "Uid:%d",uid);

    file = filp_open("/etc/passwd", O_RDONLY, 0);
    if(file == NULL)
        printk(KERN_ALERT "Can not open user file\n");
    else{
        printk(KERN_INFO "Getting user name\n");
        fsegment = get_fs();
        set_fs(get_ds());
        while(c!=0)
        {
          i=0;
          while(c!=':')
          {
            vfs_read(file, &c, 1, &file->f_pos);
            userName[i]=c;
            i++;
          }
          j=i;
          userName[i-1] = '\0';

          //pass wd
          c=' ';
          while(c!=':')
          {
            vfs_read(file, &c, 1, &file->f_pos);
          }

          c=' ';
          i=0;
          while(c!=':')
          {
            //file->f_op->read(file, &c, 1, );
            vfs_read(file, &c, 1, &file->f_pos);
            userIDs[i]=c;
            i++;
          }
          userIDs[i-1] = '\0';
          kstrtol(userIDs, 10, &userID);
           if(userID==uid)
           break;

          //groupid
          c=' ';
          while(c!=':')
          {
            vfs_read(file, &c, 1, &file->f_pos);
          }

          //idinfo
          c=' ';
          while(c!=':')
          {
            vfs_read(file, &c, 1, &file->f_pos);
          }

          //homedr
          c=' ';
          while(c!=':')
          {
            vfs_read(file, &c, 1, &file->f_pos);
          }

          //commandshell
          c=' ';
          while(c!='\n')
          {
            vfs_read(file, &c, 1, &file->f_pos);
          }
          c=' ';
      }
    set_fs(fsegment);
    printk(KERN_INFO "username:%s\n",userName);
    }
    filp_close(file,NULL);

    for(i=0;i<=j;i++){
      dev->current_user_name[i]=userName[i];
    }
    dev->current_user_name_l=j+1;

    //find user whom opened
    printk(KERN_INFO "User count:%d",dev->user_count);
    i=0;
    while(i<=dev->user_count){
      j=0;
      if(dev->data!=NULL && dev->data[i]!=NULL && dev->data[i]->name!=NULL){
        while(j<dev->data[i]->name_size-1 && dev->data[i]->name[j]!='\0' && tmp[j]!='\0' && dev->data[i]->name[j]==tmp[j]){
            //compare string
            j++;
        }
        if(j==dev->data[i]->name_size-1){
          printk(KERN_INFO "matched!");
          user_no=i;
          i=dev->user_count;
        }
      }
      i++;
    }
    kfree(tmp);

    if(dev->data!=NULL && dev->data[user_no]!=NULL && j!=dev->data[user_no]->name_size-1){
      printk(KERN_INFO "no match!");
      user_no=-1;
    }else{...
    dev->current_user_no=user_no;
    filp->private_data = dev;
    return 0;
}


int messagebox_release(struct inode *inode, struct file *filp)
{...


ssize_t messagebox_read(struct file *filp, char __user *buf, size_t count,
```

Figure 4 – Open Function handles with username, uid, user id and creates messages string

```c
ssize_t messagebox_write(struct file *filp, const char __user *buf, size_t count,
                    loff_t *f_pos)
{
    printk(KERN_INFO "Written\n");

    struct messagebox_dev *dev = filp->private_data;
    struct message *m;
    int quantum = sizeof(struct m_user);
    int i,j;
    char *tmp;
    ssize_t retval=0;

    printk(KERN_INFO "Allocating string:%d\n",count);
    tmp=kmalloc(count*sizeof(char),GFP_KERNEL);
    if (!tmp)
        goto out;

    printk(KERN_INFO "Coping from user:");
    if (copy_from_user(tmp, buf, count)) {
        retval = -EFAULT;
        goto out;
    }
    printk(KERN_INFO "%s\n",tmp);

    if(tmp[0]!='@'){
      printk(KERN_INFO "No @");
        retval = -EFAULT;
        goto out;
    }

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

    if (*f_pos >= quantum * dev->max_user) {
        retval = 0;
        goto out;
    }

    if (dev->data==NULL) {
        printk(KERN_INFO "Allocating quantum array: ");
        dev->data = kmalloc(dev->max_user * sizeof(struct m_user *), GFP_KERNEL);
        printk(KERN_INFO "%p\n",dev->data);
        if (!dev->data)
            goto out;
        for(i=0;i<dev->max_user;i++){
          dev->data[i]=NULL;
        }
    }

    //find user to send
    int user_no=0;
    i=0;
    while(i<=dev->user_count){
      j=0;
      printk(KERN_INFO "i: %d",i);
      if(dev->data[i]!=NULL && dev->data[i]->name!=NULL){
        while(j<dev->data[i]->name_size && dev->data[i]->name[j]!='\0' && tmp[j+1]!=' ' && dev->data[i]->name[j]==tmp[j+1]){
            //compare string
            j++;
        }
        if(j==dev->data[i]->name_size-1){
          printk(KERN_INFO "matched!");
          user_no=i;
          i=dev->user_count;
        }
      }
      i++;
    }

    if(dev->data[user_no]!=NULL && j!=dev->data[user_no]->name_size-1){
      printk(KERN_INFO "no match!");
      user_no=dev->user_count+1;
    }

    printk(KERN_INFO "User no:%d",user_no);

    if (dev->data[user_no]==NULL) {
        printk(KERN_INFO "Allocating quantum: ");
        dev->data[user_no] = kmalloc(quantum, GFP_KERNEL);
        printk(KERN_INFO "%p\n",dev->data[user_no]);
        if (!dev->data[user_no])
            goto out;
        dev->data[user_no]->messages=NULL;
        dev->data[user_no]->message_l=0;
        dev->data[user_no]->message_r=0;
        dev->data[user_no]->message_s=0;
        printk("Assiging name:");
        i=0;
        while(tmp[i]!=' ')
          i++;
        dev->data[user_no]->name=kmalloc(i*sizeof(char), GFP_KERNEL);
        memcpy(dev->data[user_no]->name,tmp+1,i);
        dev->data[user_no]->name_size=i;
        printk("%s\n",dev->data[user_no]->name);
        printk("Name count:%d\n",i);
        device.user_count++;
    }

    m=dev->data[user_no]->messages;
    for(i=1;i<dev->data[user_no]->message_l;i++){
      m=m->next;
    }
    if(m!=NULL && m->next==NULL){
      printk(KERN_INFO "Allocating new message:\n");
      m->next = kmalloc(sizeof(struct message), GFP_KERNEL);
      if (!(m->next))
          goto out;
      m->next->next=NULL;
      m=m->next;
    }
    if(m==NULL){
      dev->data[user_no]->messages=kmalloc(sizeof(struct message), GFP_KERNEL);
      if (!dev->data[user_no]->messages)
        goto out;
      dev->data[user_no]->messages->length=0;
      dev->data[user_no]->messages->next=NULL;
      m=dev->data[user_no]->messages;
    }
    printk(KERN_INFO "Coping from tmp\n");
    m->length = (count-dev->data[user_no]->name_size-1+dev->current_user_name_l+2);
    printk(KERN_INFO "data size:%d\n",m->length);
    m->data = kmalloc(m->length*sizeof(char), GFP_KERNEL);
    if (!m->data)
      goto out;
    printk(KERN_INFO "coping data:%s\n",tmp+dev->data[user_no]->name_size+1);
    memcpy(m->data,dev->current_user_name,dev->current_user_name_l-1);
    memcpy(m->data+dev->current_user_name_l-1,": ",2);
    memcpy(m->data+dev->current_user_name_l+1,tmp+dev->data[user_no]->name_size+1,count-dev->data[user_no]->name_size-1);

    kfree(tmp);

    dev->data[user_no]->message_s+=m->length;
    printk(KERN_INFO "Got data:\n%s\n",m->data);

    dev->data[user_no]->message_l++;

    retval = count;

    /* update the size */
    if (dev->size < *f_pos)
        dev->size = *f_pos;

  out:
    up(&dev->sem);

  return retval;
}
```

Figure 5 – Write Function handles with incoming messages and memory allocations

```c
ssize_t messagebox_read(struct file *filp, char __user *buf, size_t count,
                   loff_t *f_pos)
{
    size_t length;
    struct message *m;
    struct messagebox_dev *dev = filp->private_data;
    int user_no = dev->current_user_no;

    printk(KERN_INFO "Read!\n");

        if (dev->data == NULL || user_no<0)
            goto out;

    printk(KERN_INFO "User no:%d\nUser count:%d\n",user_no,dev->user_count);

    if(! dev->data[user_no])
      goto out;

    ssize_t retval = 0;

    if (down_interruptible(&dev->sem)){
        printk(KERN_INFO "interrupt!");
        return -ERESTARTSYS;
    }

    m=dev->data[user_no]->messages;
    if(m==NULL)
      goto out;

    length=dev->tmpLength;
    length=length-r_index;
    if(length>count)
      length=count;

    printk(KERN_INFO "tmp: %s\nlenght:%d\n",dev->tmp,length);
    if (copy_to_user(buf, dev->tmp+r_index, length)) {
        printk(KERN_INFO "copy_to_user fail");
        retval = -EFAULT;
        goto out;
    }

    r_index+=length;
    retval = length;
    dev->data[user_no]->message_r=dev->data[user_no]->message_l;
  out:
    up(&dev->sem);
    return retval;
}
```

Figure 6 – Read function copies required part of message string