Project 2

Report

Course Name: BLG413E – System Programming
Date: 5.12.2017

Group Information		
Student ID	**Name**	**CRN**
150140119	Mahmut Lutfullah Özbilen	13332
150140123	*Muhammed Kadir Yücel*	*13331*

*This report was prepared by Mahmut Lutfullah Özbilen (150140123).

Table of Contents

Introduction

This report was prepared for show our works in Project 2. Report was prepared by Mahmut Lutfullah Özbilen. In this project, we implemented a character driver that will be using as a message box between users on the system. We used Scull driver as a base in our project. We used linked list to keep records of messages. Write operation is added for updating linked list and read operation is added for showing messages of the user. We also added ioctl commands for three operations. These are changing read mode, message limit and deleting all messages of a user.

We have worked on Linux Kernel 3.13.0 version on this project with Lubuntu 14.04.5 system. Source files and tests programs can be found inside *.zip file provided through Ninova system.

Compiling

1. sudo su
2. make
3. insmod ./driver.ko
4. grep mydevice /proc/devices (output of this command is using for next step)
5. mknod /dev/mydevice0 c ### 0 (output of previous command instead of ###)
6. chmod 666 /dev/mydevice0

Project Details

In this project we used scull driver as a base, but we construct new structure for keeping message records which is linked lists. Nodes of linked list contains source user id, destination user id, message content, message length, read flag and pointer of next node (*Figure 1*).

```
struct user_message { // linked list struct to hold user message
    int fromUser; // user id of message sender
    int toUser; // user id of message receiver
    char *messageContent; // content of message
    size_t length; // length of the messsage
    int read; // if message is read
    struct user_message *next; // linked list!
};
```

Figure 1: Message struct

Write Operation

When a user sends message to another user with “echo” command, *device_write* function is called by device driver. First, we lock the device with semaphore to prevent dirty read, dirty write etc. of the device (*Figure 2*). Then we create a new message node and get user id of writer with *get_current_user()->uid.val* method. Message sent from user is copying with *copy_from_user()* function (*Figure 3*).

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

Figure 2: Locking device

```
if(copy_from_user(incomeMessage->messageContent, buf, count)){
    up(&dev->sem);
    return -EFAULT;
}
```

Figure 3: Copying message of the user

We extracted user name by taking characters until ":" in the message. Since we hold user id in the struct, we convert user name to user id with parsing /etc/passwd file and assign destination user id. We wrote a function named *username_to_userid*, that opens */etc/passwd* file and parses (*Figure 4*).

```
while(index < BUF_SIZE && !found){
    for(i = 0; i < username_length; i++){
        found = 1;
        if(username[i] != buffer[index + i]){
                found = 0;
                break;
        }
    }
    if(!found){
        while(buffer[index] != '\n' && index < BUF_SIZE)
            index++;
        if(index < BUF_SIZE)
            index++;
    }
}
if(found){
    index = index + username_length + 3;
    int uid = 0;
    for(i = index; buffer[i] != ':'; i++){
        uid = 10 * uid + buffer[i] - '0';
    }
    printk(KERN_INFO "uid is %d\n", uid);
    return uid;
}
else{
    printk(KERN_ALERT "User cannot found\n");
    return -1;
}
```

Figure 4: Parsing part of the username_to_userid function

Since we have message limit in driver, we controlled unread messages before adding new message to linked list (*Figure 5*). Then we take the original message with iterating whole income message and assign to allocated node. At the end, we add node to head of the linked list, release semaphore and return the count which is came as argument (*Figure 6*).

```
// check for unread messages count for user
struct user_message *unread_check = dev->head;
int unread_counter = 0;
while(unread_check != NULL){
    if(unread_check->toUser == incomeMessage->toUser && unread_check->read == 0)
        unread_counter++;

    unread_check = unread_check->next;
}
if(unread_counter >= unread_limit){
    printk("Too many unread messages for user %d\n", incomeMessage->toUser);
    kfree(incomeMessage);
    up(&dev->sem);
    return -ENOSPC;
}
```

Figure 5: Checking unread message of current user

```
struct user_message *temporary = dev->head;
incomeMessage->next = NULL;

if(dev->head == NULL)
    dev->head = incomeMessage;
else{
    while(temporary->next != NULL)
        temporary = temporary->next;

    temporary->next = incomeMessage;
}

temp_message = dev->head;

up(&dev->sem);
return count;
```

Figure 6. Adding new node to linked list

Read Operation

When users want to see their message, they enter "cat /dev/mydevice0" command and driver calls *driver_read* function. First, we lock the device with semaphore and get the current user id same as write operation. Then we control the device whether there is message on device or not. Then we iterate the linked list. If there are a node which its destination id is same with user id and read mode is 1 or the message is unread, message is printed. Then we get user id and convert it to user name again parsing "/etc/passwd" file with *userid_to_username* function (*Figure 7*). Then we construct message with source user name. At the end, we print the message with *copy_to_user()* function (*Figure 8*).

```
while(index < BUF_SIZE && !found){
    while(colonCounter < 2){
        if(buffer[index] == ':')
            colonCounter++;
        if(colonCounter == 1){
            usernameSize = lineCounter-1;
        }
        index++;
        lineCounter++;
    }
    for(i = 0; i < uidSize; i++){
        found = 1;
        if(uid[i] != buffer[index + i]){
                found = 0;
                break;
        }
    }
    if(!found){
        while(buffer[index] != '\n' && index < BUF_SIZE){
            index++;
            lineCounter++;
        }
        if(index < BUF_SIZE)
            index++;
        lineCounter = 0;
        colonCounter = 0;
    }
}
if(found){
    index = index - lineCounter;
    (*size) = usernameSize;
    char* username = kmalloc(usernameSize * sizeof(char), GFP_KERNEL);
    for(i = 0; i < usernameSize; i++){
        username[i] = buffer[index + i];
    }
    printk(KERN_INFO "username size is %d\n", usernameSize);
    printk(KERN_INFO "username is %s\n", username);
    return username;
}
```

Figure 7: Parsing part of the userid_to_username function

```
while(return_size == 0){
    if(temp_message == NULL){
        printk("Temp message is null\n");
        temp_message = dev->head;
        break;
    }
    if(temp_message->toUser == user_id){ // if message belongs to user
        if(read_mode == 0 && temp_message->read == 1){ // EXCLUDE_READ and read message
            printk("Read message is only unread mode\n");
            temp_message = temp_message->next;
            continue;
        }
        int user_id_length = 0;
        int username_length = 0;
        printk("Converting id to char\n");
        char* user_id = int_to_char(temp_message->fromUser, &user_id_length);
        printk("Id in char %s\n", user_id);
        char* username = userid_to_username(user_id, &username_length, user_id_length);
        if(username == NULL){
            up(&dev->sem);
            return -EINVAL;
        }
        int return_msg_length = username_length + temp_message->length + 3; // id length, message length and ': ...\0'
        char* return_msg = kmalloc(return_msg_length * sizeof(char), GFP_KERNEL);
        int i;
        for(i = 0; i < username_length; i++){
            return_msg[i] = username[i];
        }
        return_msg[i++] = ':';
        return_msg[i++] = ' ';
        printk("i value is %d\n", i);
        int j = 0;
        while(i < return_msg_length){
            return_msg[i++] = temp_message->messageContent[j++];
        }
        return_msg[return_msg_length - 2] = '\n';
        return_msg[return_msg_length - 1] = '\0';

        return_size = return_msg_length;
        if(copy_to_user(buf, return_msg, return_msg_length)){ // send to user
            up(&dev->sem);
            return -EFAULT;
        }
        printk("Message printed on users screen\n");
        temp_message->read = 1;
    }
    temp_message = temp_message->next;
}
```

Figure 8: Traversing part of the read function

Ioctl Commands

There are three ioctl command in this project which are changing read mode, setting unread message limit and deleting message of users. User need to be root to access ioctl commands.

Set Read Mode

```
case SET_READ_MODE:
    read_mode = (int) arg;
    printk("GOT READ_MODE %d\n", read_mode);
    break;
```

We defined a global read mode variable and this command change this variable (*Figure 9*).

Figure 9: Set read mode command

Set Unread Limit

We defined a variable to unread limit to for new messages. Root users can change this limit with ioctl command (*Figure 10*).

```
case SET_UNREAD_LIMIT:
    unread_limit = (int) arg;
    printk("GOT UNREAD_LIMIT %d\n", unread_limit);
    break;
```

Figure 10: Set unread limit command

Clear user messages

In this command, we lock the device with semaphore. Then, we start deleting. First, we delete head nodes until head node is not specified user. Then start tracing the linked list for deleting (*Figure 11*).

```
case CLEAR_MESSAGE_OF:
    user_id = (int) arg;
    printk("GOT USER_ID %d\n", user_id);

    struct device_dev *dev = filp->private_data;

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

    struct user_message *del_temp = dev->head;
    struct user_message *prev;

    while(del_temp != NULL && del_temp->toUser == user_id){
        dev->head = del_temp->next;
        if(del_temp->messageContent != NULL)
            kfree(del_temp->messageContent);
        kfree(del_temp);
        del_temp = dev->head;
    }

    while(del_temp != NULL){
        while(del_temp != NULL && del_temp->toUser != user_id){
            prev = del_temp;
            del_temp = del_temp->next;
        }

        if(del_temp == NULL){
            up(&dev->sem);
            return 0;
        }
        prev->next = del_temp->next;
        if(del_temp->messageContent != NULL)
            kfree(del_temp->messageContent);
        kfree(del_temp);

        del_temp = prev->next;
    }
    up(&dev->sem);

    break;
```

Figure 11: Clear all messages of user

Tests

We provided a test program to run ioctl commands. This program expects three arguments which are device name, ioctl command and argument. Then we run command with *ioctl()* function, which is function of <*sys/ioctl.h*> library. Test takes user name and convert it to user id from /etc/passwd file.