BLG 413E
System Programming

Project 2

Group No: 33

Members'

Name	**ID**
Mertcan Yasakçı	150140051
Süheyl Emre Karabela	150140109

Date of Delivery
29.11.2017

1. Introduction

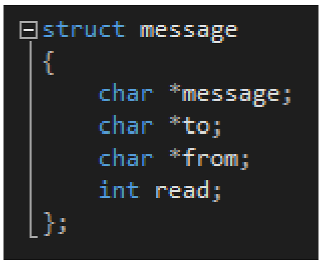
In this project, we are wanted to implement a device driver that will serve as a message box to users on a system. Write operation to the device will place the given message to message box and read operation will show messages sent to reading user.

For this project, we modified scull device to achieve goals stated in the project file. Main parts that we modified are device layout, write operation, read operation and ioctl function.

2. Implementation

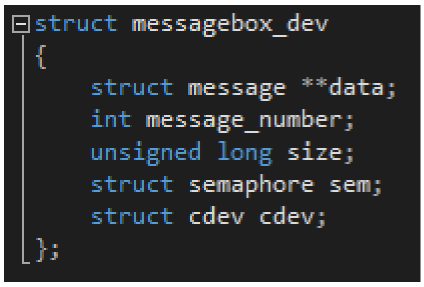
2.1. Device Layout

Messages will be sent to the message box in this format: "@<to> <message>". We have to implement a structure to hold sender, receiver, message and a read flag for further processes. In order to achieve this, we implement following structure.



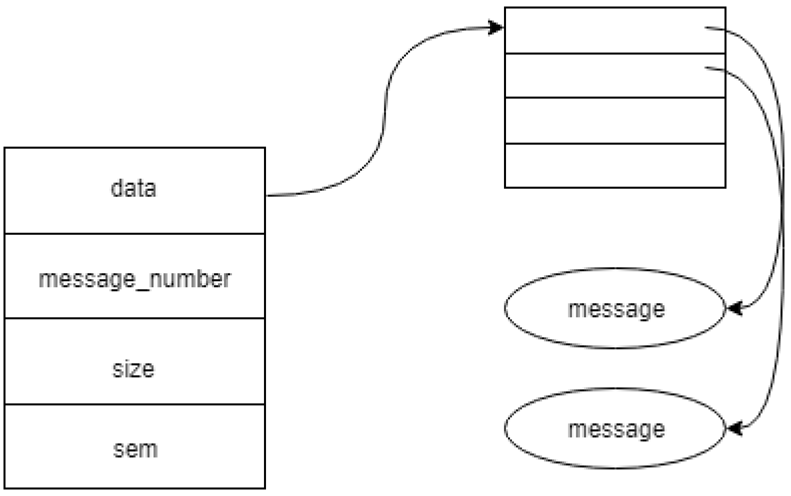
```
struct message
{
    char *message;
    char *to;
    char *from;
    int read;
};
```

We also made necessary changes in the message box device structure. We used instances of message structure as quantum. That's why we changed type of data pointer.



```
struct messagebox_dev
{
    struct message **data;
    int message_number;
    unsigned long size;
    struct semaphore sem;
    struct cdev cdev;
};
```

You can see the relationship between these structures below.



2.2. Write Operation

Writing a message to the message box device can be done using echo command from the shell. Example command for write operation can be seen below.

```
echo "@joe hello" > /dev/messagebox
```

When a message sent to the device messagebox_write method will work. The algorithm we implemented to store the message in this function can be seen below.

```
Using copy_from_user method, get fetch message to a buffer
Get message-sending user’s username
Get receiver’s username from message
IF reciever’s unread message box is full
    THEN return “quota exceeded” error
ENDIF
Get message body
Place message, sender, receiver information to memory and set read flag to 0
```

After a successful run of this method, message sent using echo command placed in a next available data slot of the device with additional information like sender, receiver and read flag.

Since receiver user’s unread message count will be checked, there is no way to send a message to a user whose unread message box is full.

2.3. Read Operation

Users can read messages sent to them. Users use cat command to achieve this functionality. An example command for reading a message box can be found below.

```
cat /dev/messagebox
```

When a reading request is done to the device, messagebox_read method will work. The algorithm we implemented to display messages to a user in this function can be seen below.

```
Get message-reading user’s username
FOR every message in the device
    IF message’s receiver is not message-reading user
        THEN move to next message
    ENDIF
    IF device is not set to include_read and the message is read
        THEN move to next message
    ENDIF
    Copy the message to a buffer
    Set read flag of the message
END
Copy buffer to the user
```

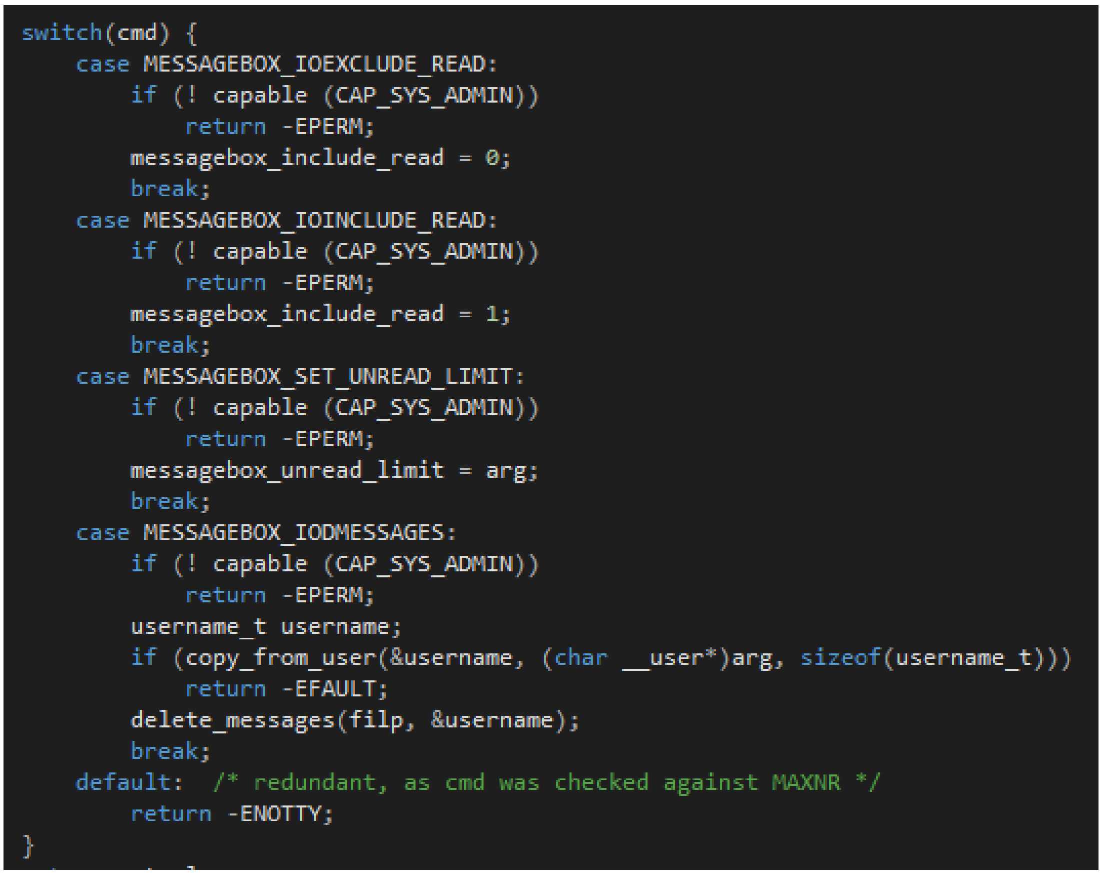
After a successful run of this method, only all of the unread messages of the message-reading user are sent to him/her if include_read flag is not set. If include_read flag is set, then reading-user receives all messages sent to him/her.

Since read flag of a message is checked along with include_read flag, there is no way to read an already read message if include_read flag is not set.

2.4. ioctl Function

ioctl command is used to set different functionalities of the device. Our ioctl command takes predefined command values to set functionalities like including read messages, excluding read messages, setting maximum unread message number and deleting all of the messages that sent to a user.

All of these commands can only called by super user privileges.



```
switch(cmd) {
    case MESSAGEBOX_IOEXCLUDE_READ:
        if (! capable (CAP_SYS_ADMIN))
            return -EPERM;
        messagebox_include_read = 0;
        break;
    case MESSAGEBOX_IOINCLUDE_READ:
        if (! capable (CAP_SYS_ADMIN))
            return -EPERM;
        messagebox_include_read = 1;
        break;
    case MESSAGEBOX_SET_UNREAD_LIMIT:
        if (! capable (CAP_SYS_ADMIN))
            return -EPERM;
        messagebox_unread_limit = arg;
        break;
    case MESSAGEBOX_IODMESSAGES:
        if (! capable (CAP_SYS_ADMIN))
            return -EPERM;
        username_t username;
        if (copy_from_user(&username, (char __user*)arg, sizeof(username_t)))
            return -EFAULT;
        delete_messages(filp, &username);
        break;
    default:  /* redundant, as cmd was checked against MAXNR */
        return -ENOTTY;
}
```

In every command case, we check if the user has the necessary privileges. According to this privilege check, we continue to process the request or reject it.

MESSAGEBOX_IOEXCLUDE_READ and MESSAGEBOX_IOINCLUDE_READ commands are used to set or reset the include_read flag of the device.

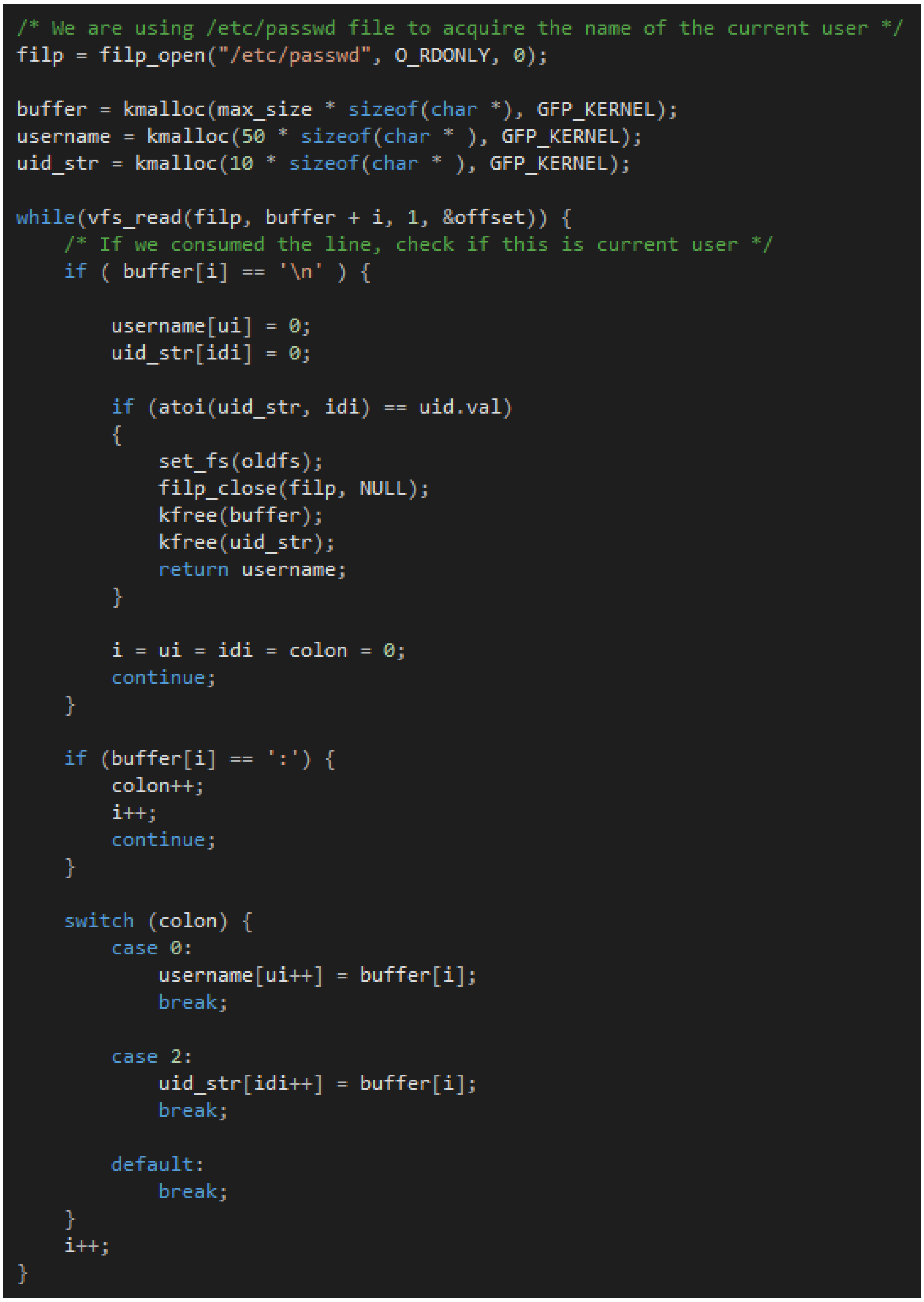
MESSAGEBOX_SET_UNREAD_LIMIT command sets the unread_limit number according to supplied argument.

MESSAGEBOX_IODMESSAGES command deletes all of the messages of the user supplied as an argument to ioctl function.

2.5. Additional Functions

2.5.1. get_user_name

In order to get a user's username from uid of that user, we need to parse the file "/etc/passwd". This is the file that users are kept with their uids. We wrote get_user_name function to return a char array which is the username of the current user.



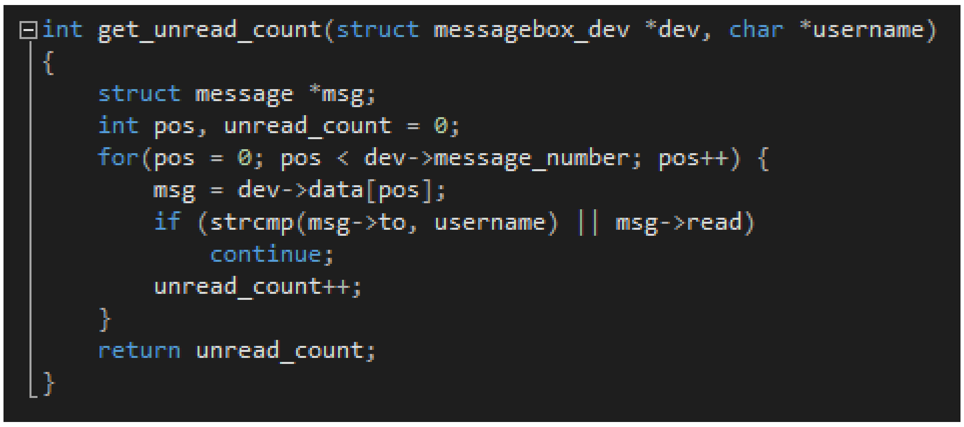
```
/* We are using /etc/passwd file to acquire the name of the current user */
filp = filp_open("/etc/passwd", O_RDONLY, 0);

buffer = kmalloc(max_size * sizeof(char *), GFP_KERNEL);
username = kmalloc(50 * sizeof(char * ), GFP_KERNEL);
uid_str = kmalloc(10 * sizeof(char * ), GFP_KERNEL);

while(vfs_read(filp, buffer + i, 1, &offset)) {
    /* If we consumed the line, check if this is current user */
    if ( buffer[i] == '\n' ) {

        username[ui] = 0;
        uid_str[idi] = 0;

        if (atoi(uid_str, idi) == uid.val)
        {
            set_fs(oldfs);
            filp_close(filp, NULL);
            kfree(buffer);
            kfree(uid_str);
            return username;
        }

        i = ui = idi = colon = 0;
        continue;
    }

    if (buffer[i] == ':') {
        colon++;
        i++;
        continue;
    }

    switch (colon) {
        case 0:
            username[ui++] = buffer[i];
            break;

        case 2:
            uid_str[idi++] = buffer[i];
            break;

        default:
            break;
    }
    i++;
}
```

You can see the main part of this function above.

2.5.2. get_unread_count

We implemented this function to get the number of messages that are unread of a user. We used this function in write method in order to check whether a user can receive a message or not.

This function basically counts the messages that are sent to a specific username. You can see the whole method below.

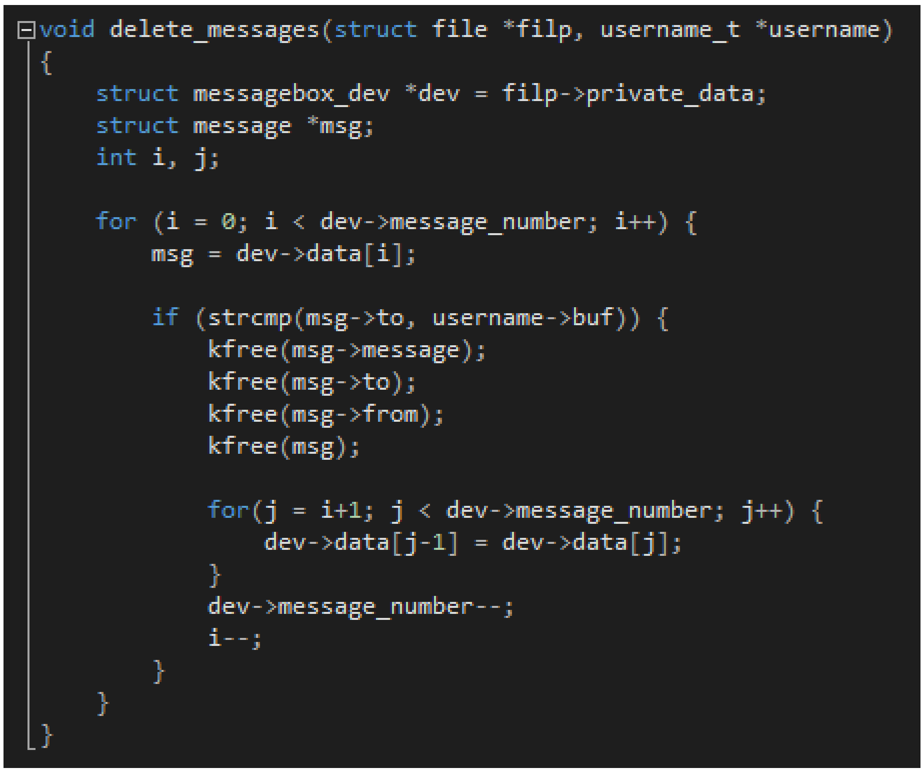


```c
int get_unread_count(struct messagebox_dev *dev, char *username)
{
    struct message *msg;
    int pos, unread_count = 0;
    for(pos = 0; pos < dev->message_number; pos++) {
        msg = dev->data[pos];
        if (strcmp(msg->to, username) || msg->read)
            continue;
        unread_count++;
    }
    return unread_count;
}
```

2.5.3. delete_messages

We implemented this function to use it in message-deleting ioctl command. This function basically takes a username and searches the whole device in order to find messages sent to the username and delete them by freeing sources.

You can see full implementation of this method below.

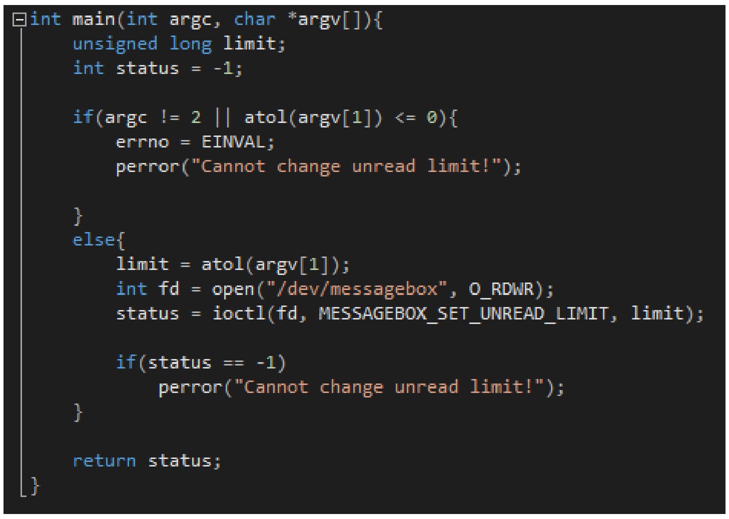


```c
void delete_messages(struct file *filp, username_t *username)
{
    struct messagebox_dev *dev = filp->private_data;
    struct message *msg;
    int i, j;

    for (i = 0; i < dev->message_number; i++) {
        msg = dev->data[i];

        if (strcmp(msg->to, username->buf)) {
            kfree(msg->message);
            kfree(msg->to);
            kfree(msg->from);
            kfree(msg);

            for(j = i+1; j < dev->message_number; j++) {
                dev->data[j-1] = dev->data[j];
            }
            dev->message_number--;
            i--;
        }
    }
}
```

3. Test Programs

3.1. unread_limit_test

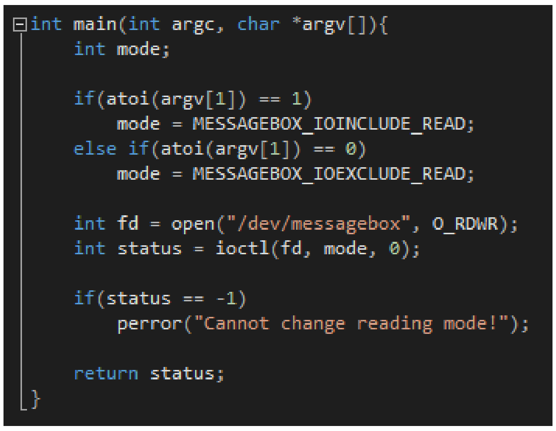
This program takes an integer input to set the maximum unread messages that a user's inbox can hold. If a user's unread message box is full, further message sending request will be blocked and a quota error will be displayed to the user.



```c
int main(int argc, char *argv[]){
    unsigned long limit;
    int status = -1;

    if(argc != 2 || atol(argv[1]) <= 0){
        errno = EINVAL;
        perror("Cannot change unread limit!");

    }
    else{
        limit = atol(argv[1]);
        int fd = open("/dev/messagebox", O_RDWR);
        status = ioctl(fd, MESSAGEBOX_SET_UNREAD_LIMIT, limit);

        if(status == -1)
            perror("Cannot change unread limit!");
    }

    return status;
}
```

3.2. read_mode_test

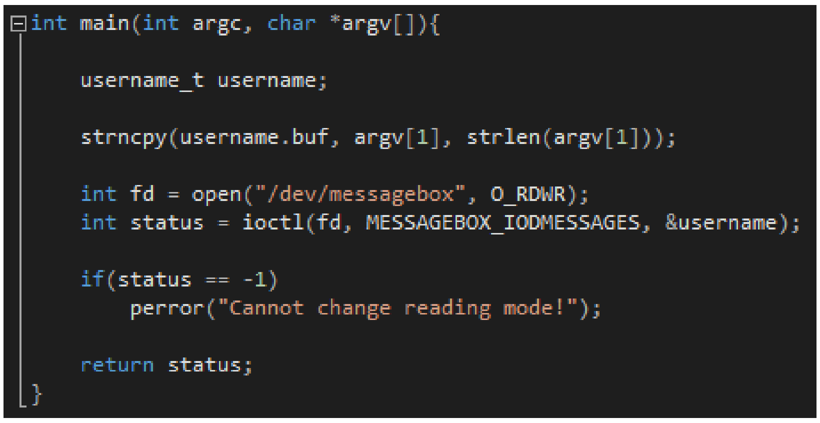
This program takes a flag input which can be 1 or 0 to set or reset include_read flag.



```c
int main(int argc, char *argv[]){
    int mode;

    if(atoi(argv[1]) == 1)
        mode = MESSAGEBOX_IOINCLUDE_READ;
    else if(atoi(argv[1]) == 0)
        mode = MESSAGEBOX_IOEXCLUDE_READ;

    int fd = open("/dev/messagebox", O_RDWR);
    int status = ioctl(fd, mode, 0);

    if(status == -1)
        perror("Cannot change reading mode!");

    return status;
}
```

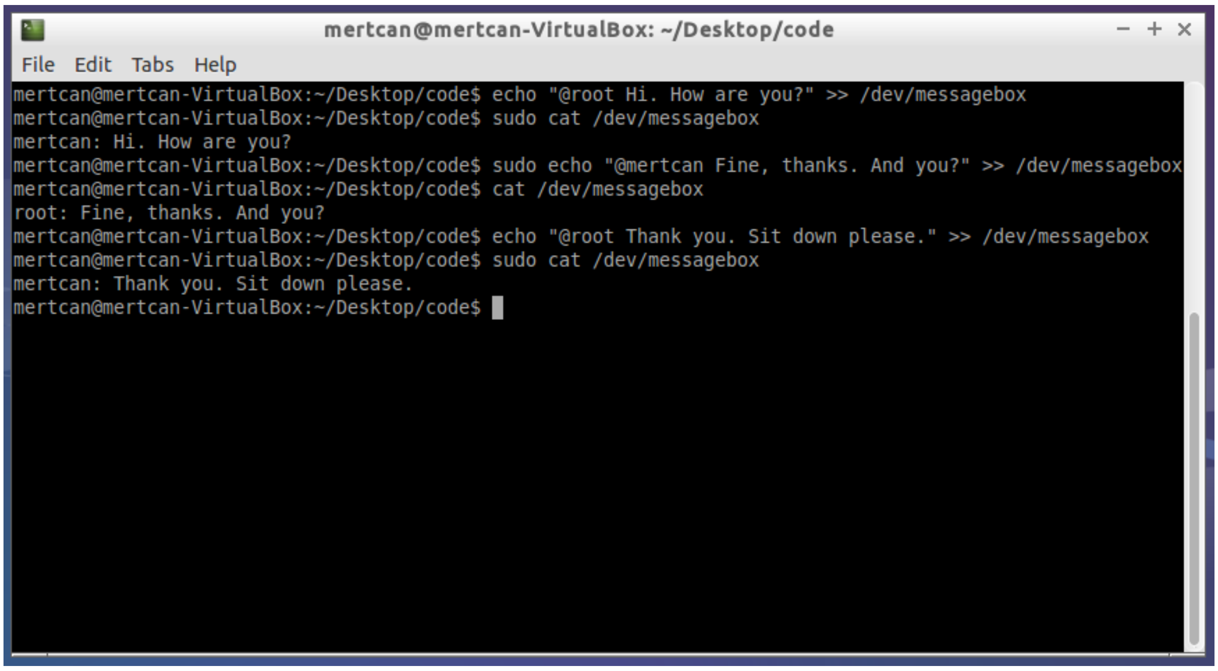
3.3. delete_test

This program takes a username in order to delete all messages sent to that user.

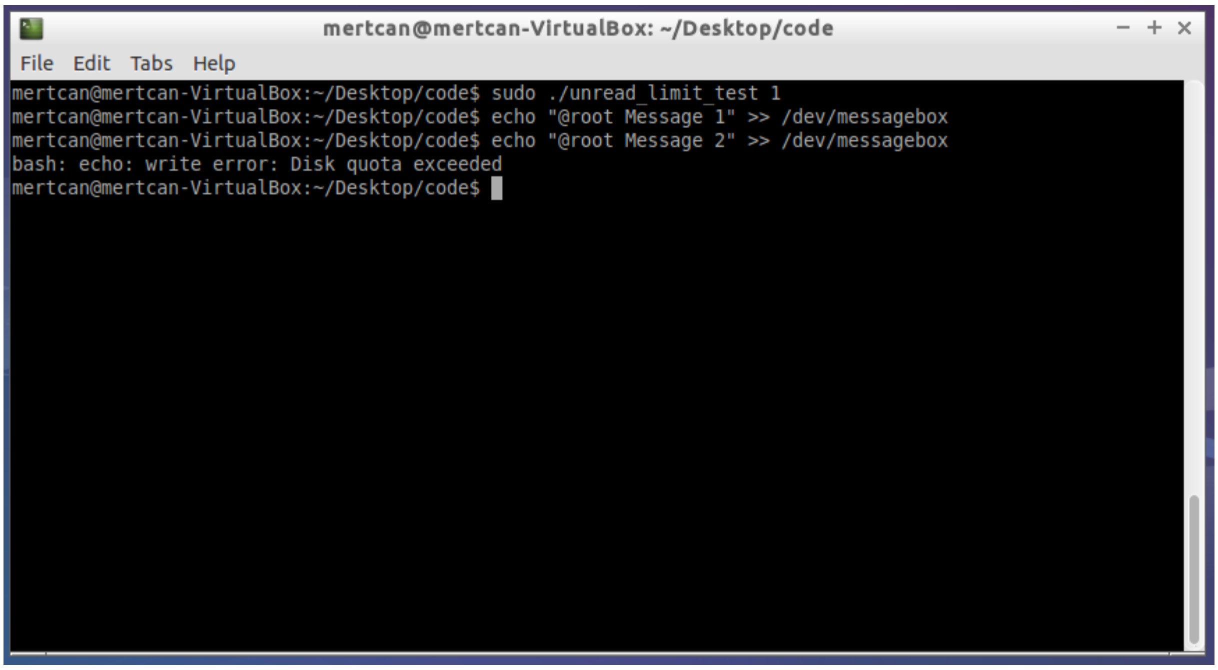


```c
int main(int argc, char *argv[]){

    username_t username;

    strncpy(username.buf, argv[1], strlen(argv[1]));

    int fd = open("/dev/messagebox", O_RDWR);
    int status = ioctl(fd, MESSAGEBOX_IODMESSAGES, &username);

    if(status == -1)
        perror("Cannot change reading mode!");

    return status;
}
```

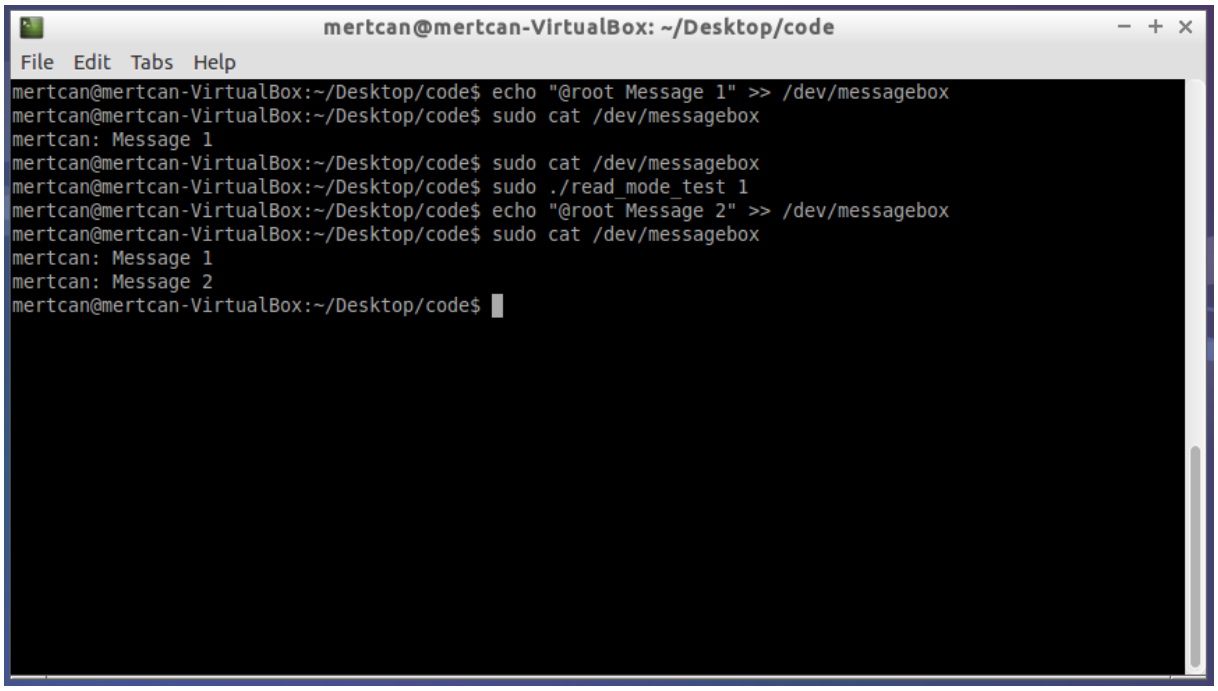
4. Screenshots



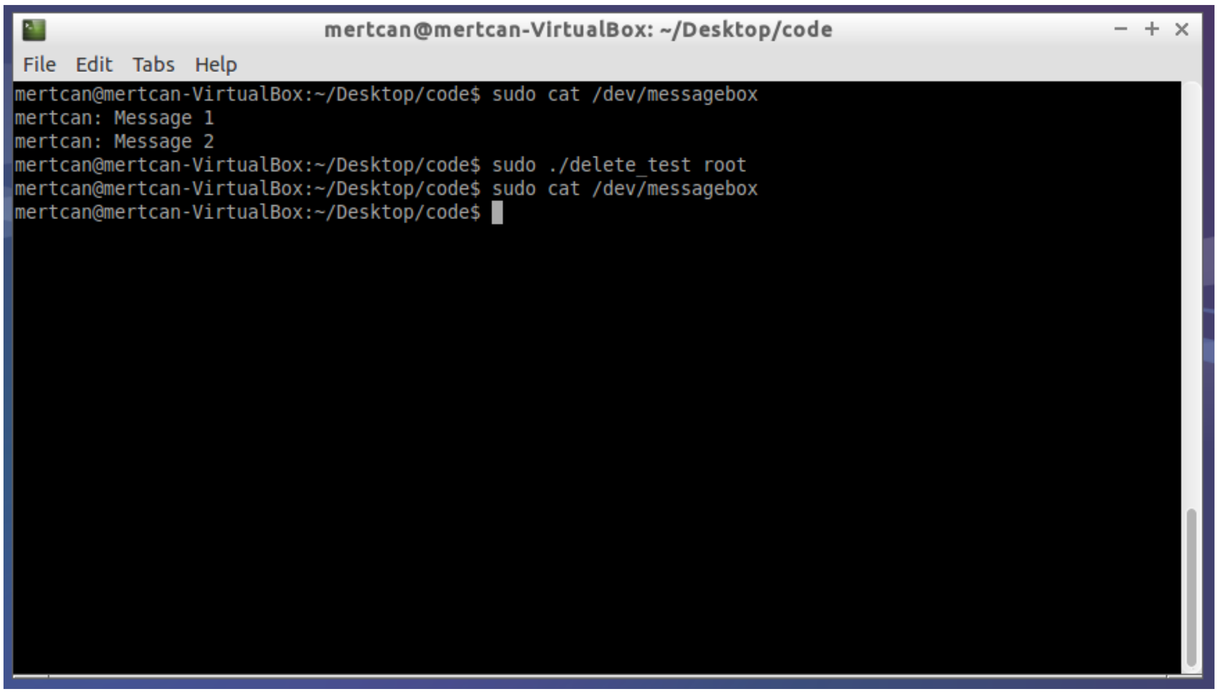
```
mertcan@mertcan-VirtualBox:~/Desktop/code$ echo "@root Hi. How are you?" >> /dev/messagebox
mertcan@mertcan-VirtualBox:~/Desktop/code$ sudo cat /dev/messagebox
mertcan: Hi. How are you?
mertcan@mertcan-VirtualBox:~/Desktop/code$ sudo echo "@mertcan Fine, thanks. And you?" >> /dev/messagebox
mertcan@mertcan-VirtualBox:~/Desktop/code$ cat /dev/messagebox
root: Fine, thanks. And you?
mertcan@mertcan-VirtualBox:~/Desktop/code$ echo "@root Thank you. Sit down please." >> /dev/messagebox
mertcan@mertcan-VirtualBox:~/Desktop/code$ sudo cat /dev/messagebox
mertcan: Thank you. Sit down please.
mertcan@mertcan-VirtualBox:~/Desktop/code$
```



```
mertcan@mertcan-VirtualBox:~/Desktop/code$ sudo ./unread_limit_test 1
mertcan@mertcan-VirtualBox:~/Desktop/code$ echo "@root Message 1" >> /dev/messagebox
mertcan@mertcan-VirtualBox:~/Desktop/code$ echo "@root Message 2" >> /dev/messagebox
bash: echo: write error: Disk quota exceeded
mertcan@mertcan-VirtualBox:~/Desktop/code$
```



```
mertcan@mertcan-VirtualBox:~/Desktop/code$ echo "@root Message 1" >> /dev/messagebox
mertcan@mertcan-VirtualBox:~/Desktop/code$ sudo cat /dev/messagebox
mertcan: Message 1
mertcan@mertcan-VirtualBox:~/Desktop/code$ sudo cat /dev/messagebox
mertcan@mertcan-VirtualBox:~/Desktop/code$ sudo ./read_mode_test 1
mertcan@mertcan-VirtualBox:~/Desktop/code$ echo "@root Message 2" >> /dev/messagebox
mertcan@mertcan-VirtualBox:~/Desktop/code$ sudo cat /dev/messagebox
mertcan: Message 1
mertcan: Message 2
mertcan@mertcan-VirtualBox:~/Desktop/code$
```



```
mertcan@mertcan-VirtualBox:~/Desktop/code$ sudo cat /dev/messagebox
mertcan: Message 1
mertcan: Message 2
mertcan@mertcan-VirtualBox:~/Desktop/code$ sudo ./delete_test root
mertcan@mertcan-VirtualBox:~/Desktop/code$ sudo cat /dev/messagebox
mertcan@mertcan-VirtualBox:~/Desktop/code$
```