System Programming Project Report

Project No	Project 2
Group Number	G 58
Group Members	Cemal Türkoğlu - 150140719 Rıdvan Sırma -
Demo Date	21.11.2017

Introduction

In this project our aim was developing a device deriver that will act as a message box between all users in the operating system. For this device which acts as a driver is placed in the /dev/messagebox file. Therefore sending message and reading is done via file operations upon this file. Message format is like “@to message”. Users can read only messages that sent to them. And there are 2 different modes in our device, EXCLUDE_READ that will not show already written messages and INCLUDE_READ that will show all messages. There is also a limit on the number of unread messages. These properties can change but only by superuser. So superuser can change the number of limit, read mode and delete all messages of a user.

Implementation

First of all, we have created a struct that will store a message and after a linked list consist of these struct type elements. Our struct is in this form :

```
typedef struct Node{
    int read_flag;
    char to[LIM_TO];
    char from[LIM_TO];
    char message[LIM_MSG];
    struct Node* next;
}node;
```

So with this struct type we have created linked list. In global scope we have a head node which is assigned NULL at the beginning. In every write operation we are creating a new message node and adding to the list. All linked list implementations are done like standart data structure operations that we have learn in previous courses. So we have :

- **node *create_node(char to[ ],char from [ ], char message [ ])**
- **int add_node(char to[ ],char from [ ], char message [ ])**
- **int traverse(void)**
- **int delete_node(node *n)**

functions to implement linked list operations. And we have some auxilary functions that are used whenever needed in linked list and scull functions.

- **Check_mailbox_size**

```
253 int check_mailbox_size(char username[]){
254     int count = 0;
255     node *traverse = head;
256     while(traverse != NULL){
257         if(strcmp(traverse->to,username) == 0 && traverse->read_flag == 0){
258             count=count+1;
259             printk(KERN_ALERT "COUNT = %d\n",count);
260         }
261         traverse = traverse->next ;
262     }
263     if(count >= mailbox_size ){
264         printk(KERN_ALERT "Mailbox is full \n");
265         return 0;   // error
266     }
267     printk(KERN_ALERT "Mailbox is not full \n");
268     return 1;       // OK.
269 }
270
```

Function takes username as a parameter and travers on the list to find messages to this user. There is a global variable mailbox_size, so it compares the number of messages to this user with this limit.

- **Find_username**

```
168 char* find_username(char* uid){
169         struct file *f;
170         mm_segment_t fs;
171         char str[5000];
172         loff_t offset;
173         offset = 0;
174
175         f = filp_open("/etc/passwd",O_RDONLY,0644);
176         fs = get_fs();
177         set_fs(get_ds());
178
179         vfs_read(f,str,5000,&offset);
180         set_fs(fs);
181         filp_close(f,NULL);
182
183         char *token, *username,*userid;
184         char *arg = str;
185
186         while(token = strsep(&arg,"\n")){
187             username = strsep(&token,":");
188             strsep(&token,":");
189             userid = strsep(&token,":");
190             if(strcmp(userid,uid) == 0){
191                 return username;
192             }
193         }
194         return "null";
195 }
196
```

In the program we can get the user id from current pointer, but we could not find any way to get username directly. So this function gets the uid and read /etc/passwd file. The content of file is stored in arg variable. Variable is parsed line by line via strsep function and we are

trying to catch the username of given user id. So function basically function takes uid of user and returns username.

- **Read_message**

```
197 char *read_message(char uname[]){
198     char *result = " ";
199     char *del;
200     int size = strlen(result);
201     node *tmp = head;
202     while(tmp != NULL ){
203         if(strcmp(tmp->to,uname)==0){
204             if(read_mode == 0 && tmp->read_flag == 1){
205                 tmp = tmp->next;
206                 continue ;
207             }
208
209             del = (char *)kmalloc(strlen(result)*sizeof(char),GFP_KERNEL);
210             strcpy(del,result);
211             size += strlen(tmp->from) + strlen(tmp->message) + 2;
212             result = (char *)kmalloc(size*sizeof(char),GFP_KERNEL);
213
214             strcpy(result,del);
215             strcat(result,tmp->from);
216
217             strcat(result,":");
218             strcat(result,tmp->message);
219             strcat(result,"\n ");
220
221
222             kfree(del);
223             tmp->read_flag = 1;
224
225         }
226         tmp = tmp->next;
227     }
228     return result;
229 }
```

Function takes username, travers over the list, catches messages to this user. There is an important point at line 204, if read_mode == 0 which means in EXCLUSIVE mode and read_flag == 1 which means message is already read, in this case pass this message, go to next node and continue the loop.

Every message is read in the format “FROM : MESSAGE \n” , and it is concataneted all the time in result string. So it creates a string holding all unread/read messages of user in line by line style. And returns this result string.

The key point on the project is impelementing **scull_read** and **scull_write** functions which are operated when there is echo or cat commands for our device.

- **Scull_read**

```
330     char uid[LIM_FROM];
331     sprintf(uid,"%d",current_uid());
332
333     char *username;
334     int from_size = strlen(uid);
335     username = (char *) kmalloc(from_size * sizeof(char),GFP_KERNEL);
336     strcpy(username,find_username(uid));        // username -> will be current username
337
338     char *messages = read_message(username);
339
340     count = strlen(messages);
341     if(dev->size > count){
342         dev->size = count;
343     }
344
345     if (copy_to_user(buf, messages , count)) {
346         retval = -EFAULT;
347         goto out;
348     }
349     *f_pos += count;
350     retval = count;
351
```

In scull_read function we have made these changes. First of all we are taking current uid and sending it find_username function and we reach the username of current user. After we are sending this username to read_message function. As a result we get a string consist of messages to this user line by line format. Also we are changing the count variable with the length of message. After that copy_to_user function sends our messages string to buffer and it is printed to screen.

- **Scull_write**

```
426     char *to,*from,*message;
427     char temp[LIM_FROM];
428     char *temp2 ;
429     char *temp3;
430     sprintf(temp,"%d",current_uid());
431
432     int from_size = strlen(temp);
433     from = (char *) kmalloc(from_size * sizeof(char),GFP_KERNEL);
434     strcpy(from,find_username(temp));
435
436     char * bufTemp = (char *)kmalloc(strlen(buf)*sizeof(char),GFP_KERNEL);
437     strcpy(bufTemp,buf);
438     temp2 = strsep(&bufTemp," ");  // temp2 => @bob
439     int to_size = strlen(temp2)-1 ;  // to get rid of @ charachter from the beginning
440     to = (char *) kmalloc(to_size * sizeof(char),GFP_KERNEL);
441     to = &temp2[1];     // to => bob
442
443     temp3 = strsep(&bufTemp,"\n");
444     int size_msg = strlen(temp3);
445     message = (char *)kmalloc(size_msg * sizeof(char),GFP_KERNEL);
446     strcpy(message,temp3);
447
448     if(check_mailbox_size(to) == 0){
449         printk(KERN_ALERT "MAIL BOX IS FULL \n");
450         retval = -EFAULT;
451         goto out;
452     }else{
453         add_node(to,from,message);
454         kfree(message);
455     }
```

In this function we are taking the message in the format of “@To Message” and parsing it. We took it directly from buffer which is coming from userspace. First we need to know who is sending the message, so again via using uid and find_username function we are determining the value of FROM variable in node struct.

After that via cropping the string from 1st index to the index that will be space, and this gives us TO variable for message. The rest is the body of message so we copy it to MESSAGE variable.

Before adding the message to mailbox, we need to check whether it is full or not. It is done via check_mailbox_size function. It returns zero in the failure. So if there is not any failure, there is space in the mailbox, we can send our variables to add_node function to add the node to list.

IOCTL Commands and Super-User Functionality

- **Read_mode**

```
577     case READ_MODE:
578       if (! capable (CAP_SYS_ADMIN)){
579           printk(KERN_ALERT "ONLY SUPERUSER CAN CHANGE THE READ_MODE \n");
580           return -EPERM;
581       }
582       if(arg == 0 || arg ==1){
583           read_mode = arg;
584
585       }else{
586           return -EBADMSG;
587       }
588     break;
589
```

If user has admin permissions it is allowed to change the read_mode flag with the arg sent.

- **Delete_messages**

```
590   case DELETE_MESSAGES:
591     if (! capable (CAP_SYS_ADMIN)){
592         printk(KERN_ALERT "ONLY SUPERUSER CAN DELETE MESSAGES \n");
593         return -EPERM;
594     }
595     copy_from_user(&buf,(char *)arg,20*sizeof(char));
596     // buf username ismi oldu.. loop içinde o mesajlar silinmeli
597     node *temp = head ;
598     while(temp->next != NULL){          // traverse the list
599         if(strcmp(temp->to,buf) == 0){  // find messages to be deleted for the user(->buf in icinde)
600             delete_node(temp);
601         }
602         temp = temp->next;
603     }
604     if(strcmp(temp->to,buf) == 0){      // for last element
605         delete_node(temp);
606     }
607
```

First control is again about administrative permissions. After that we are taking the username as parameter and traverse the list to see all nodes that has this username. And for this nodes we are sending this node to delete_node function to be removed from the list.

Test Program

Our test program works like this:

Usage : ./test filename | -r [0-1] | -d [username]

So filename is the path of our device, with r option we can change the read_mode, and with d option we can delete messagebox of a user.

```
62      switch (option)
63      {
64          case change_read_mode:
65              if (ioctl(fd, READ_MODE,read_mode) == -1)
66              {
67                  perror("set read mode: ");
68              }
69              break;
70          case delete_messages:
71              if (ioctl(fd, DELETE_MESSAGES,user) == -1)
72              {
73                  perror("delete messages: ");
74              }
75              break;
76          case e_set:
77              break;
78          default:
79              break;
80      }
```