Department for Mathematics & Computer Science
University of Southern Denmark

DM510 Operating Systems

Department for Mathematics & Computer Science

IMADA

Project 2: System Call

Torvald Johnson – tjohn16

March 18, 2017

Table of Contents

1 Introduction 3

2 Design 3

3 Implementation 4

4 Testing 6

5 Concurrency discussion 9

6 Conclusion 10

7 Appendices 10

7.1 Appendix I: Source code for msgbox.c 10

7.2 Appendix II: Source code for testmsgbox.c 12

7.3 Appendix III: Full output for test 1 15

1 Introduction

The task at hand was to add system calls to User-Mode Linux, implementing a message box in kernel space. Two system calls were required in this project, one to write messages to the message box, and one to retrieve them. The message box was to be implemented in the form of a stack. The project also required, as the task dealt with reading and writing to kernel memory, that all parameters should be checked and precautions should be taken regarding concurrent calls of these system calls.

2 Design

Due to the nature of this project and my complete inexperience in regards to working within the kernel space, the decision was made early on to include abundant instances of the printk() function, to print various notes to the console and help keep track of exactly how far the program had gotten before running into problems. This was deemed necessary due to the ramifications of programming errors when dealing with the kernel space. As stated in the introduction, small errors can cause the kernel to crash or even lead to User-Mode Linux being inoperable. This would often even require a complete redownload and compiling of the entire setup, which could take quite some time. As multiple mistakes were inevitable, and their combined total would result in a large amount of development time lost to waiting, it was quite valuable to have information printed consistently to the console during runtime, allowing me to more accurately pinpoint how far the program had run before crashing, and speeding up the debugging process.

The kmalloc and kfree functions were used to allocate memory within the kernel. Because addresses in user-space do not map to the same physical address in kernel-space, the functions copy_to_user() and copy_from_user() were utilized when transferring data. Although it would be prudent to use the access_ok() function to verify that it is safe to read or write from a given user-space address, this function was not explicitly employed within the program. After researching the copy_from_user and copy_to_user functions, it was found that both of them call the access_ok function before transferring any data, so for it to be included in the program would be redundant.

Of course, it was important to check all parameters and handle any errors, and special attention was paid to this issue. Any instance in which memory was allocated or data transferred within the program, the result was verified to be correct and variables were checked to be within acceptable parameters. In the event of an unacceptable outcome, a short descriptive error message would be printed to the console, and the program would be stopped, returning the negative value of the relevant error code, thus preventing the system call from failing uncontrolled and bringing down the kernel. Originally the plan was to implement an error handling function, but since time became an issue during the development cycle this idea had to be dropped. This will be further discussed in the implementation and conclusion sections of the report.

To handle concurrent system calls, the local_irq_save() and local_irq_restore() functions were used save the current processor state and disable hard interrupts while in a critical region of the system call. Were this precaution not taken, an interrupt occurring while message box data is being updated by another system call could result in unexpected results. This will be discussed more in depth later in the concurrency discussion section of the report.

As far as the actual structure of the program is concerned, the original plan had been to divide the large tasks of writing user data to the message box and reading data from the message box into smaller subtasks. This would make the program more versatile and easier to test and read. However, once again due to unexpected time constraints and my relative inexperience, this was not able to be fully implemented. This will be discussed further in the implementation and conclusion sections of this report.

3 Implementation

In lines 20 through 27 of the msgbox.c program, we can see a good example of the implementation of the previously described use of printk, kmalloc, error checking, and concurrency handling.

```
local_irq_save(flags);
msg_t* msg = kmalloc(sizeof(msg_t), GFP_KERNEL);
if (!msg) {
      printk("Unable to allocate memory for message! Out of memory!\n");
      printk("errno = -12 ENOMEM.\n");
      local_irq_restore(flags);
      return(-12);
}
```

Figure 3.1. Lines 20-27 of msgbox.c, from the function sys_msgbox_put()

Here we can see the implementation of, specifically, local_irq_save() disabling interrupts, kmalloc allocating memory for the message, and printk() outputting error messages to the console. This implementation of error handling, as previously mentioned, was not the first choice for the project. It is inelegantly shoehorned in as part the actual sys_msgbox_put() function, with printk() printing out static lines of text. It does accomplish the necessary error handling, informs the user of what went wrong, and stops the program before any damage is done to the kernel. However, the previous plan had been to design and implement a separate function to handle errors, which could dynamically produce error messages and codes based on the specific error encountered by the system. Unfortunately, this proved to be too complicated an undertaking, and the deadline for this project was unexpectedly moved to an earlier date, which forced me to rethink my priorities and produce, ultimately, a messier and less elegant solution in order to finish on time.

Another aspect of the implementation that was affected by my need for more time, was the structure of functions in the program. As the project stands now, it consists of two giant functions, namely sys_msgbox_put() and sys_msgbox_get() which handle every aspect of the task at hand. They are relatively messy and difficult to read, since the creation of messages, handling of errors and concurrency, and reading and writing of data are all handled in these two functions. Unfortunately, a good deal of work had been put into separating these two large functions out into three auxiliary functions called create_msg(), copy_msg_from_user(), and read_msg_from_kernel(), but in order to speed up development and meet the deadline I was forced to drop these features from the final build.

They were almost entirely implemented, but untested, so I have included them in this report in the following figures for reference.

```
msg_t* create_msg(int length) {
  msg_t* msg = kmalloc(sizeof(msg_t), GFP_KERNEL);
  if (!msg) {
        printk("Unable to allocate memory! Out of memory!\n");
        printk("errno = -12 ENOMEM\n");
       msg = NULL;
        return msg;
  }
  msg->previous = NULL;
  msg->length = length;
  msg->message = kmalloc(length, GFP_KERNEL);
  if (!msg->message) {
       printk("Unable to allocate memory! Out of memory!\n");
        printk("errno = -12 ENOMEM.\n");
       msg = NULL;
        return msg;
  }
  return msg;
}
```

Figure 3.2. Unfinished function create_msg(), not included in final build

```
int copy_msg_from_user(msg_t* msg, char buffer, int length) {
  printk("Message copied. THE LENGTH IS:  %i \n", length);
  if(buffer == NULL){
    printk("Buffer is NULL! Invalid argument!\n");
    printk("errno = -22 EINVAL.\n");
         local_irq_restore(flags);
    return(-22);
  }
  ssize_t res;
  res = copy_from_user(msg->message, buffer, length);
  if(res) {
    printk("Whole message not copied! Bad address!   %d bytes not copied \n", res);
    printk("errno = -14 EFAULT.\n");
    local_irq_restore(flags);
    return(-14);
  }
  printk("Message copied. THE MESSAGE IS:  %s \n", msg->message);
  return 0;
}
```

Figure 3.3. Unfinished function copy_msg_from_user(), not included in final build

```
int read_msg_from_kernel(msg_t* msg, char buffer, int length, int mlength) {
  if(buffer == NULL) {
        printk("Buffer is NULL! Invalid argument!\n");
        printk("errno = -22 EINVAL.\n");
             local_irq_restore(flags);
        return(-22);
  }
    if(msg == NULL||msg->message == NULL) {
        printk("Message is NULL! Invalid argument!\n");
        printk("errno = -22 EINVAL.\n");
             local_irq_restore(flags);
        return(-22);
    }
    if(length < mlength) {
        printk("Buffer is smaller than message! Invalid argument!\n");
        printk("errno = -22 EINVAL.\n");
             local_irq_restore(flags);
        return(-22);
    }
    copy_to_user(buffer, msg->message, mlength);

    /* free memory */
    kfree(msg->message);
    kfree(msg);
    return 0;
}
```

Figure 3.4. Unfinished function read_msg_from_kernel(), not included in final build

4 Testing

A test class was created consisting of five tests, to ensure that the program could handle the various negative conditions described in the project description. A video of these tests running will also be included, and this section of the report will go over four of the tests and their expected and actual output, providing a reference to the relevant time in the video for each test.

```
int test1(void){
  char *in = "This is a stupid message.";

  printf("This test will attempt to create a message in which the length is incorrectly
specified as a negative value. \n");

  /* Send a message containing 'in' */
  int result = syscall(__NR_msgbox_put, in, -1);

  return result;
}
```

Figure 4.1. test1() from testmsgbox.c

Test 1 is an attempt to create a message in which the length is incorrectly specified as a negative value. For this test to pass, an error should be thrown after kmalloc() fails to allocate the negative amount of memory. It is crucial that the test returns an error code and stops the program at this point, so as to prevent unpredictable results when attempting to copy the message.

This test begins at 33 seconds into the video. The expected outcome for this test, is that the console will print “Unable to allocate memory for message! Out of memory!” “errno = -12 ENOMEM.” and return -12. The actual output of the program was slightly different, and as it takes up quite a few lines, it will be included in the appendices after this report. The system outputted quite a few lines of unintelligible text which I think were generated during the kmalloc() call on a negative value, but as you can see after that the program exited with the expected lines printed to the console.

Test 2 is an attempt to create a message with a positive specified length, but a NULL value as the buffer parameter. For this test to pass, an error should be thrown just before the program attempts to copy the message, reading “Buffer is NULL! Bad address!” “errno = -14 EFAULT.” and returning -14.

```
int test2(void){
  char *in = NULL;

  printf("This test will attempt to create a message, passing a NULL value as the buffer
parameter \n");

  /* Send a message containing 'in' */
  int result = syscall(__NR_msgbox_put, in, 50);

  return result;
}
```

Figure 4.2. test2() from testmsgbox.c

This test begins at 44 seconds into the video. The expected outcome is printed to the console, and has been included in the following figure 4.3.

```
Running test 2.
This test will attempt to create a message, passing a NULL value as the buffer parameter
msgboxPut start.
msgboxPut BEGIN COPY.
Buffer is NULL! Bad address!
errno = -14 EFAULT.
```

Figure 4.3. The output of test2()

Test 3 is an attempt to read a message with a positive specified length, but a NULL value as the buffer parameter. The expected outcome of this test would be for the system to throw an error just before beginning the copy_to_user() function, writing to the console “Buffer is NULL! Bad address!” “errno = -14 EFAULT.” and returning -14.

```
int test3(void){
  char *in = "This is a stupid message.";
  char msg = NULL;

  printf("This test will attempt to read a message, passing a NULL value as the buffer
parameter \n");

  /* Send a message containing 'in' */
  syscall(__NR_msgbox_put, in, strlen(in)+1);

  /* Read a message */
  int result = syscall(__NR_msgbox_get, msg, 50);

  return result;
}
```

Figure 4.4. test3() from testmsgbox.c

This test can be seen at 50 seconds into the video, and the outcome is as expected. It has been included in the following figure 4.5.

```
Running test 3.
This test will attempt to read a message, passing a NULL value as the buffer parameter
msgboxPut start.
msgboxPut BEGIN COPY.
Message copied. THE MESSAGE IS:  This is a stupid message.
msgboxPut SUCCESS.
msgboxGet start.
The top message is:  This is a stupid message.
msgboxGet BEGIN COPY.
Buffer is NULL! Bad address!
errno = -14 EFAULT.
```

Figure 4.5. The output of test3()

Test 4 attempts to read a message with a specified length of smaller than the message at the top of the stack's length. The expected output of this test is "Buffer is smaller than message! Invalid argument!" "errno = -22 EINVAL." followed by the system returning -22. .56 sec.

```
int test4(void){
  char *in = "This is a stupid message.";
  char msg[50];

  printf("This test will attempt to read a message, passing a value smaller than the length
of the message on the stack as the length parameter \n");

  /* Send a message containing 'in' */
  syscall(__NR_msgbox_put, in, strlen(in)+1);

  /* Read a message */
  int result = syscall(__NR_msgbox_get, msg, 1);

  return result;
}
```

Figure 4.6. test4() from testmsgbox.c

This test can be viewed 56 seconds into the video, and the output is exactly what is expected. It has been included in the following figure 4.7.

```
Running test 4.
This test will attempt to read a message, passing a value smaller than the length of the
message on the stack as the length parameter
msgboxPut start.
msgboxPut BEGIN COPY.
Message copied. THE MESSAGE IS:  This is a stupid message.
msgboxPut SUCCESS.
msgboxGet start.
The top message is:  This is a stupid message.
msgboxGet BEGIN COPY.
Buffer is smaller than message! Invalid argument!
errno = -22 EINVAL.
```

Figure 4.7. The output of test3()

5 Concurrency discussion

In this project, I was perhaps a bit generous in the amount of time I left interrupts disabled. Basically shortly after the moment either sys_msgbox_put() or sys_msgbox_get() is called, the system will then call local_irq_save() and disable interrupts. Only at the conclusion of either function, right before returning, is local_irq_release() finally called, enabling interrupts again. This was a conscious decision on my part, I decided to implement it this way because of the large potential for problems caused if interrupts were allowed. For example, the first thing sys_msgbox_get() does after disabling interrupts, is to find the top message of the stack. Even though this may seem safe, in my opinion it is

important that interrupts be disabled before even this is allowed to happen, to ensure that another system call doesn't come in and read or change the top message after it has been read by the current system call, thereby causing unexpected results.

It is important to note, however, that after calling local_irq_save(), all interrupt handling is disabled for the current processor. This means that if two processes wish to access the message box, the first one will disable interrupts so that the second one will not be able to interrupt, and will be run once the first one is finished. This means that as long as interrupts are disabled during a system call, that system call will be able to peacefully execute all the way through without worry of another system call changing the data it is working with, and once it's done the next system call will get its own turn.

6 Conclusion

Looking back at this entire project, I am satisfied with what I have been able to accomplish. The project meets the requirements outlined in the provided project description; it is able to use system calls to read and write from the user-space to the kernel-space and handle errors safely. As previously discussed in the implementation section of this report, I do feel that certain elements of my program could be more fully-fleshed out, however given the extenuating circumstances of my having planned around a deadline that got moved and trying to juggle this project and projects in my other classes, I believe that this was a successful first time developing a program to interact with the kernel. I have also learned a lot about the necessity of careful error-handling code when dealing with sensitive parts of the system like the kernel, and it was even an educational experience to have to cut down on the features I wanted in the project in order to meet the deadline. I actually really enjoyed this project in general, I've never done anything like completely crashing the kernel and leaving it inoperable, then restoring it to a functional state before. This was genuinely a lot of fun, and I am much more interested in the inner-workings of an operating system now that I have experienced it firsthand.

7 Appendices

7.1 Appendix I: Source code for msgbox.c

```
#include "uapi/asm-generic/errno.h"
#include "uapi/asm-generic/errno-base.h"
#include <linux/slab.h>

typedef struct _msg_t msg_t;

struct _msg_t{
  msg_t* previous;
  int length;
  char* message;
};
```

```
static msg_t *bottom = NULL;
static msg_t *top = NULL;

unsigned long flags;

int sys_msgbox_put( char *buffer, int length ) {
  printk("msgboxPut start.  \n");
  local_irq_save(flags);
  msg_t* msg = kmalloc(sizeof(msg_t), GFP_KERNEL);
  if (!msg) {
        printk("Unable to allocate memory for message! Out of memory!\n");
        printk("errno = -12 ENOMEM.\n");
        local_irq_restore(flags);
        return(-12);
  }
  msg->previous = NULL;
  msg->length = length;
  msg->message = kmalloc(length, GFP_KERNEL);
  if (!msg->message) {
        printk("Unable to allocate memory! Out of memory!\n");
        printk("errno = -12 ENOMEM.\n");
        local_irq_restore(flags);
        return(-12);
  }

  printk("msgboxPut BEGIN COPY.  \n");
  if(buffer == NULL){
    printk("Buffer is NULL! Bad address!\n");
    printk("errno = -14 EFAULT.\n");
          local_irq_restore(flags);
    return(-14);
  }
  ssize_t res;
  res = copy_from_user(msg->message, buffer, length);
  if(res) {
    printk("Whole message not copied! Bad address!   %d bytes not copied \n", res);
    printk("errno = -14 EFAULT.\n");
    local_irq_restore(flags);
    return(-14);
  }
  printk("Message copied. THE MESSAGE IS:  %s \n", msg->message);

  if (bottom == NULL) {
    bottom = msg;
    top = msg;
  } else {
    /* not empty stack */
    msg->previous = top;
    top = msg;
  }
  printk("msgboxPut SUCCESS.  \n");
  local_irq_restore(flags);
  return 0;
}
```

```
int sys_msgbox_get( char* buffer, int length ) {
  printk("msgboxGet start.  \n");
  if (top != NULL) {
    local_irq_save(flags);
    msg_t* msg = top;
    int mlength = msg->length;
    top = msg->previous;

    /* copy message */
    printk("The top message is:  %s \n", msg->message);
    printk("msgboxGet BEGIN COPY.  \n");
    if(buffer == NULL) {
        printk("Buffer is NULL! Bad address!\n");
        printk("errno = -14 EFAULT.\n");
        local_irq_restore(flags);
        return(-14);
    }
    if(msg == NULL||msg->message == NULL) {
        printk("Message is NULL! Bad address!\n");
        printk("errno = -14 EFAULT.\n");
        local_irq_restore(flags);
        return(-14);
    }
    if(length < mlength) {
        printk("Buffer is smaller than message! Invalid argument!\n");
        printk("errno = -22 EINVAL.\n");
        local_irq_restore(flags);
        return(-22);
    }

    copy_to_user(buffer, msg->message, mlength);

    /* free memory */
    kfree(msg->message);
    kfree(msg);

    printk("msgboxGet SUCCESS.  \n");
    local_irq_restore(flags);
    printk("msgboxGet local_irq_restore complete.  mlength= %d \n",mlength);
    return mlength;
  }
  return -1;
}
```

7.2 Appendix II: Source code for testmsgbox.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "arch/x86/include/generated/uapi/asm/unistd_64.h"
```

```
int test1(void);
int test2(void);
int test3(void);
int test4(void);
int test5(void);

int main(int argc, char** argv) {

  int res = 0;
  int choice;

  printf("Which test would you like to run? \n");
  scanf(" %i", &choice);
  printf("Running test %i.\n", choice);

  if(choice == 1) {
       res = test1();
  }

  if(choice == 2) {
       res = test2();
  }

  if(choice == 3) {
       res = test3();
  }

  if(choice == 4) {
       res = test4();
  }

  if(choice == 5) {
       res = test5();
  }

return res;
}

int test1(void){
  char *in = "This is a stupid message.";

  printf("This test will attempt to create a message in which the length is
incorrectly specified as a negative value. \n");

  /* Send a message containing 'in' */
  int result = syscall(__NR_msgbox_put, in, -1);

  return result;
}

int test2(void){
  char *in = NULL;

  printf("This test will attempt to create a message, passing a NULL value as the
buffer parameter \n");
```

```
  /* Send a message containing 'in' */
  int result = syscall(__NR_msgbox_put, in, 50);

  return result;
}

int test3(void){
  char *in = "This is a stupid message.";
  char msg = NULL;

  printf("This test will attempt to read a message, passing a NULL value as the
buffer parameter \n");

  /* Send a message containing 'in' */
  syscall(__NR_msgbox_put, in, strlen(in)+1);

  /* Read a message */
  int result = syscall(__NR_msgbox_get, msg, 50);

  return result;
}

int test4(void){
  char *in = "This is a stupid message.";
  char msg[50];

  printf("This test will attempt to read a message, passing a value smaller than the
length of the message on the stack as the length parameter \n");

  /* Send a message containing 'in' */
  syscall(__NR_msgbox_put, in, strlen(in)+1);

  /* Read a message */
  int result = syscall(__NR_msgbox_get, msg, 1);

  return result;
}

int test5(void){
  char *in = "This is a stupid message.";
  char msg[50];
  int msglen;

  printf("This test will simply attempt to create a message, and then read that
message off of the stack, using acceptable data as parameters. \n");

  /* Send a message containing 'in' */
  syscall(__NR_msgbox_put, in, strlen(in)+1);

  /* Read a message */
  msglen = syscall(__NR_msgbox_get, msg, 50);

  return 0;
}
```

7.3 Appendix III: Full output for test 1

```
Running test 1.
This test will attempt to create a message in which the length is incorrectly
specified as a negative value.
msgboxPut start.
------------[ cut here ]------------
WARNING: CPU: 0 PID: 1711 at mm/slab_common.c:657 kmalloc_slab+0x47/0xa4()
Modules linked in:
CPU: 0 PID: 1711 Comm: testmsgbox Not tainted 3.18.48 #58
Stack:
 68289d50 6005f474 00000000 602778ac
 00000009 602edd64 68289d60 60279958
 68289dc0 60036246 100000000 6009e650
Call Trace:
 [<602778ac>] ? printk+0x0/0xa0
 [<6001af94>] show_stack+0x132/0x18d
 [<6005f474>] ? dump_stack_print_info+0xe1/0xea
 [<602778ac>] ? printk+0x0/0xa0
 [<60279958>] dump_stack+0x2a/0x2c
 [<60036246>] warn_slowpath_common+0x98/0xc4
 [<6009e650>] ? kmalloc_slab+0x47/0xa4
 [<6002d4c2>] ? set_signals+0x0/0x40
 [<600363b0>] warn_slowpath_null+0x1c/0x1e
 [<6009e650>] kmalloc_slab+0x47/0xa4
 [<602778ac>] ? printk+0x0/0xa0
 [<600b732f>] __kmalloc+0x1a/0x103
 [<602778ac>] ? printk+0x0/0xa0
 [<6002d4c2>] ? set_signals+0x0/0x40
 [<6001ca2b>] sys_msgbox_put+0xb3/0x1e2
 [<6001d2d4>] handle_syscall+0x65/0x7c
 [<600305d0>] userspace+0x441/0x547
 [<60019ef2>] ? interrupt_end+0x0/0x80
 [<6001d317>] ? copy_chunk_to_user+0x0/0x29
 [<6001d447>] ? do_op_one_page+0x0/0x1e3
 [<6002c38a>] ? save_registers+0x1f/0x39
 [<6003323e>] ? arch_prctl+0xf9/0x173
 [<60019da6>] fork_handler+0x85/0x87

---[ end trace b1f3759162807c51 ]---
Unable to allocate memory! Out of memory!
errno = -12 ENOMEM.
```