DM510 - Operating Systems
Project 2: System Call

Jeff Gyldenbrand (jegyl16)
Supervisor: Daniel Merkle
Southern University of Denmark

July 20, 2018

Contents

1 Introduction

This project is done in collaboration with Jonas Sørensen (joso216) and Simon D. Jørgensen (simjo16).

The goal of this project is to add system calls to the user-mode linux (UML), that implement a message box in kernel-space. More specific two system calls is made, one that writes a message to kernel-space, and one that reads a message from kernel-space. For simplicity the message box is implemented as a stack, meaning incoming messages are put on top of stack, and outgoing messages are read from top of the stack. When messages are read, they are immediately removed.

2 Design decisions

As recommended in the project description, the message box is implemented as a stack. This message box is implemented in the file message_box.c, and handles the interprocess communication between user- and kernel-space. More specifically, a function has been developed for each of the system calls: sys_put, which push the message send by the user, onto the stack, and sys_get, which reads from the top of the stack, and then removes it from the stack. The two system calls will be elaborated in greater details in section 3.

Furthermore a safety precaution is implemented in the message box, a macro-function named LOCK which ensures that whenever a user is invoking the sys_put system call, and pushing a message onto the stack, no other processes can interrupt. This is illustrated in figure 1 where step one plainly shows the stack. Step two is where the sys_put is invoked, and the lock-macro locks the global stack while a new message is pushed onto the stack. Lastly in step three, global stack now points to the top, and the lock is free'ed. This function is also elaborated in greater details in section 3.

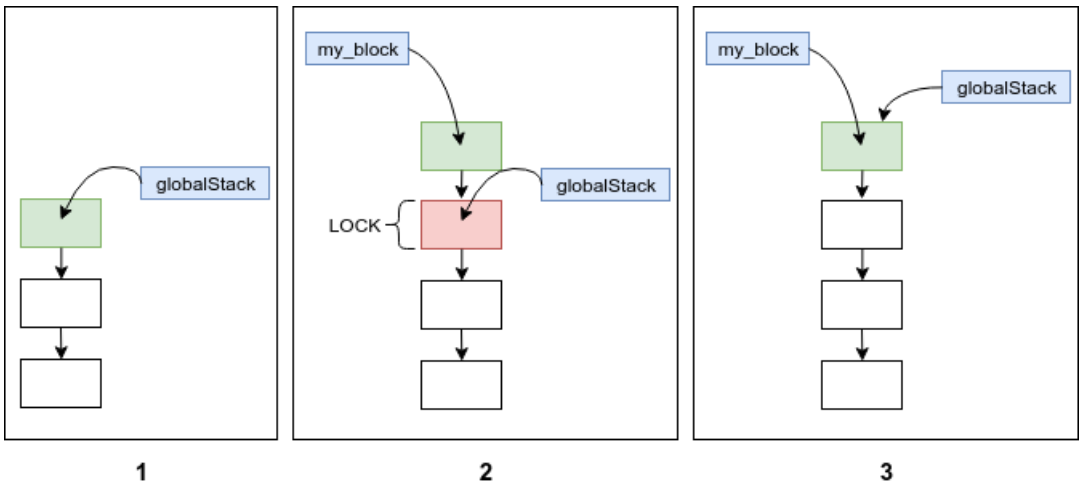

fig:1

3 Implementation

As mentioned in section 2, a LOCK-macro has been implemented. See listing 1 for the code. This macro takes two arguments. First argument is a given lock of type integer, either PUSH_LOCK, POP_LOCK or both. The locks is defined in the header errors.h as binary shifts of 1. See figure 2, where a POP_LOCK is '0 0 0 1' which equals 1 and PUSH_LOCK is '0 0 1 0' which equals 2. The second argument is the code to be encapsulated by the lock(s). When the LOCK-macro is invoked the local_irq_save()-macro is also invoked, to prevent any interupts.

Error name	Binary shift	Binary number
POP_LOCK	1 << 0	0 0 0 1
PUSH_LOCK	1 << 1	0 0 1 0
NOT_FOUND	1 << 2	0 1 0 0
NULL_ERROR	1 << 3	1 0 0 0

fig:2

The LOCK-macro then does a logical AND of the provided lock in the argument and the current lock. If two locks yield a one as result it means that it tries to apply a lock of a certain type to an already existing lock of same type, and thus we would get an logical intersection of the two. When this happens we simply invoke the local_irq_restore()-macro to allow interrupts again, and returns the lock. If the logical AND of the locks yields a zero, the lock is simply applied to the given code.

Listing 1: message_box.c :: LOCK()

```
1  #define LOCK(locks, f) { \
2          unsigned long flags; \
3          local_irq_save(flags); \
4          const int temp_lock = lock; \
5          if((-lock) & (-locks)) { \
6                  local_irq_restore(flags); \
7                  return lock; \
8          }\
9          else lock |= locks;\
10         local_irq_restore(flags); \
11         f \
12         lock = temp_lock;\
13 }while(false) \
```

As explained in section 2, a stack is implemented to handle how the message is written and read. The sys_put system call, which writes the message to kernel-space, uses this stack to write the message. The aforementioned LOCK-macro as seen in listing 2, line 4-8 embraces the code that push the message onto the stack.

Listing 2: message_box.c :: sys_put()

```
1  int sys_put(void *user_message, unsigned int user_length){
2      if(!user_message) return NULL_ERROR;
3      msg_t * my_block;
4      LOCK(POP_LOCK,
5          LOCK(PUSH_LOCK,
6              my_block = stack_push(globalStack, user_length );
```

```
7             if( my_block && my_block-> message ) globalStack = my_block;
8             );
9         if( my_block && my_block-> message  ) my_block -> length = user_length -
              copy_from_user( my_block->message, user_message, user_length );
10    );
```

4 Test

Two test programs have been developed to verify that the solution works correctly with both valid and invalid call parameters. Both programs are implemented such that a user pass as parameter, in the execution of the program, the message to be send. E.g. ./test1.out "a message" and ./test2.out "this is also a message". These messages will be written to kernel-space.

The first test of program 1 is ./test1.out "42", see video at time-frame 0:55. The motivation for this test is to see that a user successfully can send a message from user-space to kernel-space, and then read it from kernel-space. What happens is illustrated in figure 4, where test 1 writes the message to kernel-space, reside there for a second before it is read out to std::out.

The second test of program 1 is ./test1.out, notice no string is provided. See video at time-frame 01:13. Now the motivation is to see what happens if a user sends a NULL-message. As shown in the video, the program yields an error-code 8 (NULL_ERROR) and error-code 4 (NOT_FOUND). Hence, no message was ever written to kernel-space. These error-codes will be elaborated later in this section.

In the second test-program the motivation is to test an interception of another process' message, and replace this message with its own message. Hence, testing for concurrency. See video at time-frame 01:34. Here test-program 2 will run: ./test2.out "hello..." As shown in figure 4, the message will reside in the first white box and wait until another process' message arrives. See video at time-frame 01:49. Here the other process is started with test-program 1: ./test1.out "anybody out there?.." See figure 4 again, where this message will go to the first white box, and then be intersected by test-program 2, which then replaces the message with its own message, and writes it to kernel-space. Lastly Both test-program 1 and test-program 2 will read the new message.

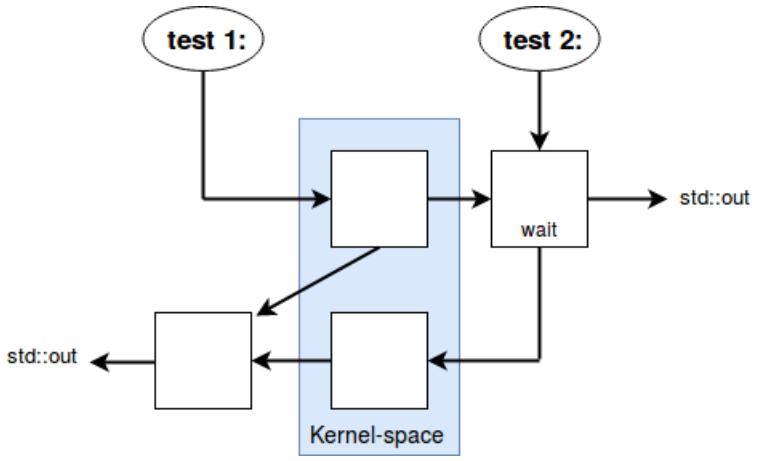

fig:3

To elaborate on the before mentioned error-codes the description of them was mentioned in section 3, figure 2. The actual error-checking is done in the test-files by performing a logical AND of all the error-codes to a specific return code. E.g. in figure 4, here the return code is error: 1 1 1 0. First this error code is AND'ed with the error-code for POP_LOCK which yields a false. Every other case yields true. Hence, three errors was returned.

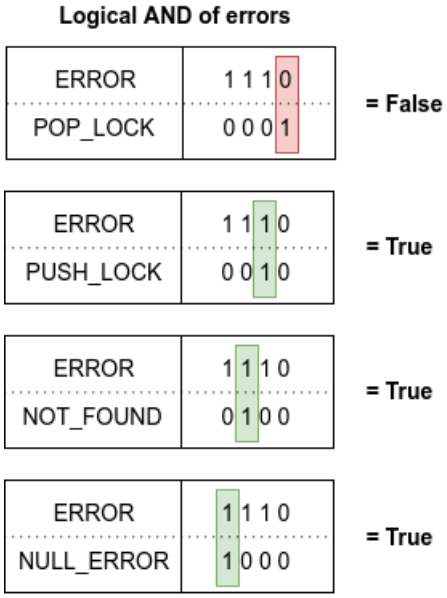

fig:4

5 Discussion

Because of the LOCK-macro implementation in the message box a race-condition would never be able to occur. This statement is tested in the previous section where test-program 2 waits for another process' message to intercept it, and replace it.

6 Conclusion

It is possible to make system calls to the kernel, to write and read messages from kernel-space. The message box handles both concurrency, because of the LOCK-macro, and the corner-case, where a user would send a NULL-message. As recommended the message box was implemented as a stack. Every goal of this project has been met.

7 Appendix

7.1 Source-code

errors.h

```
#define POP_LOCK -(1 << 0)
#define PUSH_LOCK -(1 << 1)

#define NOT_FOUND -(1 << 2)
#define NULL_ERROR -(1 << 3)
```

message_box.h

```
#ifndef TALK_H
#define TALK_H
#include "linux/kernel.h"
#include "linux/unistd.h"

static char * message = NULL;
extern int sys_get(char *, int);
extern int sys_put(char *, int );
extern int sys_length(void);
#endif
```

message_box.c

```
#include "linux/kernel.h"
#include "linux/unistd.h"
#include "linux/slab.h"
#include "linux/uaccess.h"
#include "errors.h"

/*
       locks is which locks that should be applied.
       f is the code that we want to execute during the given locks.
*/
int active_locks = 0;

#define LOCK(new_locks, f) { \
       unsigned long flags; \
       local_irq_save(flags); \
       if((-active_locks) & (-new_locks)) { \
              local_irq_restore(flags); \
              return active_locks; \
       }\
       const int temp_lock = active_locks; \
       active_locks |= new_locks;\
       local_irq_restore(flags); \
       f \
       active_locks = temp_lock;\
}while(false) \

typedef struct msg_t{
       void * message;
       unsigned int length;
       struct msg_t * next;
```

```
31 }msg_t;
32
33 /*
34 * @globalStack : globalStack of a given stack.
35 *
36 * @length : size of the block of memeory to be pushed on the stack.
37 */
38 msg_t * stack_push(msg_t * globalStack, unsigned int length){
39         msg_t * new_msg = kmalloc(sizeof(*new_msg), GFP_KERNEL);
                              // Allocate the header of the message in kernel land.
40         new_msg -> message = kmalloc(length,GFP_KERNEL);
                                                  // Allocate memory for the message
              in kernel land.
41         new_msg -> next = globalStack;

                              // next points to the globalStack of the stack
42         new_msg -> length = length;

                                           // length
43         return new_msg;
44 }
45
46 /*
47 * @ stack_globalStack : globalStack of the given stack
48 */
49 msg_t * free_msg_t(msg_t * stack_globalStack){
50         msg_t * next = stack_globalStack->next;
                                                                              // next
             points to the element below the globalStack of the stack
51         kfree(stack_globalStack->message);

                       // free message
52         kfree(stack_globalStack);

                                            // free globalStack of stack
53         return next;
54 }
55
56 // our stack
57 static msg_t * globalStack = NULL;
58
59
60 int sys_get(void *user_message, unsigned int user_length){
61         if(NULL == globalStack) return NOT_FOUND;
                                                                              // if no
             messages, global stack is null
62         if(!user_message) return NULL_ERROR;
                                                                                  // if
              user gives null as argument instead of a pointer to an array
63         msg_t *my_block;

                                                                              // create
             new instance my_block of type msg_t
64         LOCK(POP_LOCK | PUSH_LOCK,

                                           // pop and push is prevented from being
```

```
            applied by the lock function,
65              my_block = globalStack;
                                                                    // my_block points
                    to the global stack,
66              globalStack = globalStack->next;
                                            // global stack points to the next
                    element
67      );
68      // if length is less than my_block length then length will be returned,
            else my_block length is returned.
69      const int allowed_read_length = user_length < my_block->length ?
            user_length : my_block->length;
70      // subtract the succesfully copied length from the allowed read length
71      const int number_of_read_bytes = allowed_read_length - copy_to_user(
            user_message, my_block->message, allowed_read_length );
72      free_msg_t(my_block);
                                                // free my_block
73
74      return number_of_read_bytes;
75  }
76
77  int sys_put(void *user_message, unsigned int user_length){
78      if(!user_message) return NULL_ERROR;
                                    // if user gives null as argument instead of a
            pointer to an array
79      msg_t * my_block;
80      LOCK(POP_LOCK,
                            // We prevet the stack from being popped.
81              LOCK(PUSH_LOCK,
                                        // We prevent the stack from being
                    pushed.
82                  my_block = stack_push(globalStack, user_length );
                                    // push my_block to top of the stack,
83                  if( my_block && my_block-> message  ) globalStack =
                        my_block;                           // if succeeded then
                        make globalStack point to my_block (my_block is now the
                         top)
84              );
85              // we check again, if it succeeded, then we copy the message from
                    the user to the top of the stack
86              // (if anything of message is missing then it returns how many
                    bytes were copied)
87              if( my_block && my_block-> message  ) my_block -> length =
                    user_length - copy_from_user( my_block->message, user_message,
                    user_length );
88      );
89      if( !my_block ) return NULL_ERROR;
```

```
                                        // if not succeeded return error
90          if( !my_block -> message){

                                                                  // but if my_block
              succeeded but messeage failed, the block is still useless.
91              kfree(my_block);


                                                 // then we free it
92              return NULL_ERROR;
93          }
94          return my_block -> length;
95  }
96  int sys_length(void){

                                                                  // if it is
      requered of a function to be a prototype, but no arguments are given
97          if(NULL == globalStack) return NOT_FOUND;

                              // void can be used.
98          return globalStack->length;
99  }
```

test1.c

```
1   #include <stdio.h>
2   #include <errno.h>
3   #include <unistd.h>
4   #include <string.h>
5   #include <stdlib.h>
6   #include "arch/x86/include/generated/uapi/asm/unistd_64.h"
7   #include "errors.h"
8
9   void print_error(const char * msg, int error){
10    printf("%s Error code : %d\n", msg, error );
11    const char *names[] = {"POP_LOCK","PUSH_LOCK","NOT_FOUND","NULL_ERROR"};
12    for (size_t i = 0; i < sizeof(names)/sizeof(*names); i++) {
13      if(error & (1 << i)) printf("%s Error(%lu) : %s\n",msg , i, names[i]);
14    }
15  }
16
17  int main(int argc, char const *argv[]) {
18    const char * name = "Test 1 :";
19    const char * out_msg = argc < 2 ? NULL : argv[1] ;
20    printf("%s Sending message : %s\n", name, out_msg);
21    int error = syscall(__NR_put, out_msg, out_msg ? strlen(out_msg) : 0 );
22    if(error < 0) print_error(name,-error);
23    sleep(1);
24
25    const int length = syscall(__NR_length);
26    if(length < 0){
27      print_error(name, -length);
28      return 0;
29    }
30
```

```
31   char * in_msg = malloc(length);
32   error = syscall(__NR_get, in_msg, length); // Retive message;
33   if(error < 0) print_error(name,-error);
34   printf("%s Recived message : %s\n", name, in_msg);
35   free(in_msg);
36
37   return 0;
38 }
```

test2.c

```
1  #include <stdio.h>
2  #include <errno.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <stdlib.h>
6  #include "arch/x86/include/generated/uapi/asm/unistd_64.h"
7  #include "errors.h"
8
9  void print_error(const char * msg, int error){
10   printf("%s Error code : %d\n", msg, error );
11   const char *names[] = {"POP_LOCK","PUSH_LOCK","NOT_FOUND","NULL_ERROR"};
12   for (size_t i = 0; i < sizeof(names)/sizeof(*names); i++) {
13     if(error & (1 << i)) printf("%s Error(%lu) : %s\n",msg , i, names[i]);
14   }
15 }
16
17
18 int main(int argc, char const *argv[]) {
19   const char * name = "Test 2 :";
20   int length;
21   do{ // While any error occured
22      length = syscall(__NR_length);
23   }while( length < 0 );
24
25   char * in_msg = malloc(length);
26   while(syscall(__NR_get, in_msg, length) < 0); // Retrive message.
27   printf("%s Recived message : %s\n", name, in_msg);
28   free(in_msg);
29
30   const char * out_msg = argc < 2 ? NULL : argv[1] ;
31   printf("%s Sending message : %s\n", name, out_msg);
32   int error = syscall(__NR_put, out_msg, out_msg ? strlen(out_msg) : 0); // Send
         message.
33   if(error < 0){
34     print_error(name,-error);
35   }
36   return 0;
37 }
```