assignment 1: Systems call

February 2nd, 2024

Kasper Mortensen | Kamor22
Diego Fernández | Difer22

University of
Southern Denmark

1 Introduction

Message passing is a method of interprocess communication. Often used in distributed systems or when shared memory is not possible. One example of message passing is a message box. This report will present the design, implementation, and testing of a simple message box residing in kernel-space and its associated system calls.

2 Design

The message box is implemented as a stack, meaning the first element put in the box is the last element taken out. Each element on the stack contains a reference to the element below it.
We have decided to allow both the NULL and empty messages because they could be used to report that some event has happened. We also decided not to allow messages of negative length simply because it doesn't make much sense. The maximum length allowed by the program is 128 because message boxes generally use short messages.
If the system calls are called with parameters that do not follow the above-mentioned criteria, it needs to return a negative number to indicate an error has occurred.
When either of the system calls interact with the message box they need to disable interrupts to ensure they are done before another process gets access to the box. This is to ensure correct functionality in a concurrent environment.

3 Implementation

3.1 Adding a system call

To define an entry point we added an ID for our system call in `unistd_64.h` in the repository
arch/x86/include/generated/uapi/asm
And a reference to the system calls in `syscall_64.tbl`
in the repository /arch/x86/entry/syscalls/
The lines we added to these files are included in the sources, but not included here.

3.2 parameter checking

To check all criteria for the arguments outlined in the design section we used the following code. Such as checking if the buffer is valid and if the length is acceptable.

Listing 1: Checks in msgbox put

```
1 if (buffer == NULL) {
2     return -EINVAL;  //Invalid argument code}
3
4 if (length < 0 || length > 128 ){
5     return -EINVAL;
6 }
7 if (!access_ok(buffer, length)){
8     return -EFAULT;  // Bad address code
9 }
```

We also tried to check if the buffer was too small to contain the message, but it always returned an error. `msgbox_get` has similar tests. If any of the checks fail they return a negative integer code to indicate an error has occurred.

3.3 Memory allocation

To allocate memory in kernel-space we used the kmalloc function. To test if the allocation was successful we checked if the result was a null pointer. If it is we free the memory we allocated up to this point and return an error.

Listing 2

```
msg_t* msg = kmalloc(sizeof(msg_t), GFP_KERNEL);
if (msg == NULL){
    return -ENOMEM;  // Out of memory code
}
```

Listing 3

```
msg->message = kmalloc(length, GFP_KERNEL);
if (msg->message == NULL){
    kfree(msg);
    return -ENOMEM;
}
```

The memory is deallocated using the kfree() function when an element is taken out with `dm510_msgbox_get`, or when an error occurs.

3.4 Address translation

To translate addresses from user space to kernel space we used the following function:

- `copy_from_user()`
- `copy_to_user()`

The functions have a source, destination, and length. We used this to copy from our buffer to our allocated memory in `dm510_msgbox_put`, and the reverse in `dm510_msgbox_get`. The function returns 0 on success, so we check if it succedes and if it doesn't we free the memory and return an error.

Listing 4: Copy from user, from message box put

```
if (copy_from_user(msg->message, buffer, length) != 0) {
    kfree(msg);
    kfree(msg->message);
    return -EFAULT;
  }
```

The copy to user in msgbox get is very similar and included in the sources.

3.5 Concurrency

To avoid concurrency problems we disable/enable interrupts when entering the critical regions of the system calls. It is done with the following 2 macros:

- To disable: `local_irq_save(flags);`

- To enable again: `local_irq_restore(flags);`

The critical region of put and get are after they have checked the validity of the arguments, allocated memory and are ready to interact with the stack. below is the critical region of get. When we retrieve elements we

Listing 5: Critical region of msgbox put

```
local_irq_save(flags);
  if (bottom == NULL) {
    bottom = msg;
    top = msg;
  } else {
    /* not empty stack */
    msg->previous = top;
    top = msg;
  }
  local_irq_restore(flags);
  return 0;
```

likewise disable interrupts when we are done with our tests and ready to interact with the stack. The critical region of `dm510_msgbox_put` is a lot longer and will not be shown because of that, but is available in the source code.

4 Testing

For testing 4 test files were used. The entire log of all tests is included in the sources.

4.1 Test 1

Timestamp: 1 minute 35
`Test_1` was used to test the buffer argument of the 2 system calls. The first test was using a NULL buffer in `dm510_msgbox_get` and `dm510_msgbox_put`. The error code for the put system call was the EINVAL (invalid argument), because we check for NULL buffers directly. With the get command the error code was EFAULT (bad addres), because it failed the `acces_ok` check.
The small buffer test failed because we had trouble implementing a check for it, it gave errors with all input. Meaning it could put a message in and retrieve a too big message for the output buffer. When run with correct arguments the result of the system call is 26(amount of characters read), which was only done to show that it is different from the code given when trying to get from an empty box. When trying to retrieve from the empty box, the result is -1.

4.2 Test 2

Timestamp: 1 minute 40 seconds
`Test_2` was used to test the functionality with correct arguments and that the stack is implemented correctly. This was done by using the `dm510_msgbox_get` with 3 different messages and then checking what order they are retrieved from the message box. The test was a success, the message box correctly implements a stack or FILO(first in, last out) data structure. The messages also came out of the box the same way they came in, meaning the message box doesn't alter the messages in some weird way.

4.3 Test 3

Timestamp: 1 minute 43 seconds.
`Test_3` was used to test the length argument of the 2 system calls. The test was quite simple, make the system calls with a too small length(¡0) and a too big length(¿128). As mentioned in the design we allowed messages of 0 length, because they could be useful. And to disallow messages larger than 128, because message passing usually utilizes small messages. The system calls both failed and gave the EINVAL error 22 (invalid argument).

4.4 Test 4

Timestamp: 1 minute 47 seconds
`Test_4` was used to test the error handling of using the system calls without sufficient memory. This was done by repeatedly putting things on the stack until the output of the system call was negative (meaning an error). As seen in the video the test fails because the process is killed before it returns an error. We are not sure why the program crashed before the memory allocation check.

5 Conclusion

In conclusion, our report details the design, implementation, and testing of a simple message box residing in kernel-space, along with its associated system calls. We have implemented tests to ensure the validity of arguments, addresses, and memory management.
While most tests were successful, some were not. More work should be done to ensure the program doesn't crash if it fills memory or writes a message that is too large for a buffer.
Overall, the implementation demonstrates a (mostly)functional message box with proper error handling and parameter validation.

6 appendix

Listing 6: Lines edited or added in unistd-64.h

```
#define __NR_hellokernel 454
#define __NR_dm510_msgbox_get 455
#define __NR_dm510_msgbox_put 456
#ifdef __KERNEL__
#define __NR_syscalls 457
#endif
```

Listing 7: Lines added in syscall-64.tbl

```
454  common  hellokernel        sys_hellokernel
455  common  dm510_msgbox_get        sys_dm510_msgbox_get
456  common  dm510_msgbox_put        sys_dm510_msgbox_put
```

Listing 8: dm510-msgbox.h

```
#ifndef __DM510_MSGBOX_H__
#define __DM510_MSGBOX_H__

int dm510_msgbox_put( char*, int );
int dm510_msgbox_get( char*, int );

#endif
```

Listing 9: dm510-msgbox.c

```
#include "linux/unistd.h"
#include "linux/kernel.h"
#include "linux/uaccess.h"
#include "linux/slab.h"
unsigned long flags;
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;
asmlinkage
int sys_dm510_msgbox_put( char *buffer, int length ) {
  if (buffer == NULL) {
    return -EINVAL;  //Invalid argument code
  }

  if (length < 0 || length > 128 ){
    return -EINVAL;
  }
  if (!access_ok(buffer, length)){
    return -EFAULT;  // Bad address code
  }
  msg_t* msg = kmalloc(sizeof(msg_t), GFP_KERNEL);
  if (msg == NULL){
    return -ENOMEM;  // Out of memory code
  }
  msg->previous = NULL;
  msg->length = length;
  msg->message = kmalloc(length, GFP_KERNEL);
  if (msg->message == NULL){
    kfree(msg);
    return -ENOMEM;
  }

  if (copy_from_user(msg->message, buffer, length) != 0) {
    kfree(msg);
    kfree(msg->message);
    return -EFAULT;
  }
  local_irq_save(flags);
  if (bottom == NULL) {
    bottom = msg;
    top = msg;
  } else {
    /* not empty stack */
    msg->previous = top;
    top = msg;
  }
  local_irq_restore(flags);
  return 0;
}

```

Listing 10: dm510-msgbox.c pt 2

```
asmlinkage
int sys_dm510_msgbox_get( char* buffer, int length ) {
  if (buffer == NULL) {
    return -EINVAL;
  }
  if (length < 0 || length > 128 ) {
    return -EINVAL;
  }
  if(!access_ok(buffer, length)){
    return -EFAULT;
  }
  if (top != NULL) {
    local_irq_save(flags);
    msg_t* msg = top;
    int mlength = msg->length;
    top = msg->previous;
    if (mlength > length){
      kfree(msg->message);
      kfree(msg);
      return -ENOBUFS;  // Buffer full
    }
    /* copy message */
    if (copy_to_user(buffer, msg->message, mlength) != 0) {
      kfree(msg->message);
      kfree(msg);
      return -EFAULT;
    }
    /* free memory */
    kfree(msg->message);
    kfree(msg);
    local_irq_restore(flags);
    return mlength;
  }
  return -1;
}
```