Systemcalls - Project

DM510: Operativesystems

Bavith Kirshnakumar [Bakir23],
Emil Vadgaard Nielsen [Emiln17]

March 2025

Contents

1 Introduction

The goal of this project is to create two systemcalls which allows the implementation of passing messages between processes, with the help of the systemcalls: "*dm*510_*msgbox_put* & "*dm*510_*msgbox_get*". With the two systemcalls, we will be able to pass a message from user-space, through kernel-space, and back to user-space. The message system will follow the convention of a stack data-structure; Retrieving messages will follow the convention of the stack, same for adding messages.

2 Implementation

2.1 The System Call

We made the systemcalls, by making a simple stack data structure. This is done by creating *msg*-nodes, which each point to the previous node. Each node then carries the message length, and the message itself. Initially, a *top* node is created, which points to nothing.

We then have the two system calls: *msgbox_put* and *msgbox_get*. Before going into the details of the systemcall implementation, we would like to explain our strategy for error handling. We used *errno.h* library to return an appropriate error, if any potential mistakes occurred, which could crash the kernel. We used three different error flags. EINVAL, which returns 22 in the case that a bad input was given in the parameters. EFAULT, which returns 14 in the case that a mistake happened when allocating memory or problems with the address for the given buffer. Lastly, ENODATA, which returns 61 in the case that the stack is empty, and we are trying to get a message from the empty stack.

Below, we have a step-by-step explanation, of how we created our *msgbox_put* systemcall.

1. The *msgbox_put* systemcall, first checks the parameters, and if a bad input is detected, we return EINVAL if the length is incorrect, and EFAULT if there are problems with the buffer address.
2. Then we allocate space in the kernel for a single stack-node using *kmalloc*.
3. We check if the allocation worked successfully; otherwise we return EFAULT.

4. Then we set up the node, allocate space in the kernel for the message, and again check if the allocation went successfully.

5. We then copy the message from user space, into the allocated memory in the kernel space using the $copy\_from\_user()$ function. If the entire message could not be copied, the function returns an unsigned long, indicating how many bytes remained to be copied.

6. If $copy\_from\_user()$ returns anything other than 0, we return EINVAL.

7. Now we enter the critical section, where we manipulate shared data, so we therefore disable interrupts.

8. We then place the message node in the top of the stack.

9. Lastly, we enable interrupts again and return 0.

Our $msgbox\_get$ systemcall works similarly to our $msgbox\_put$ systemcall. We will still check the input parameters and return EFAULT if an error has been detected with our buffer (The length check occurs later). We then check if the stack is empty, and if it is, we return ENODATA. After the check, we enter out critical section, where we retrieve our message from the top of our stack. We check if our buffer is large enough to hold the message (otherwise, we return EINVAL). We then copy the data from the stack located in kernel space using the $copy\_to\_user()$ function, and we again check the long it returns, and if it is not 0, we return EFAULT. We then remove the node from which we read, from the stack. Then we enable interrupts again, free our messages and our node using $kfree()$, and finally return the length of the message we retrieved.

2.2 The Test Suite

When creating our test suite, we have made several programs to test different aspects of our systemcalls. We have written both programs that work to test how it works, but we have also made some that intentionally return errors, to check if the error handling works. However, all these test programs follow the same structure, best demonstrated in our $test1.c$ program. First we create our message, which we wish to send to kernel space. We create a buffer, for a retrieved message to be stored in. and we have a value to store our errors in, should one be returned. We then make our $msgbox\_put$ systemcall with our message and the length of the message. If an error occurs, we check what the

systemcall returns and, if it is below zero, we call the *perror*() function with the value. Then we do the same for the *msgbox_get* systemcall. Normally, this systemcall returns the length of the message, but we disregard the length in these programs, as we are more interested in whether they return an error. The other programs are simple variations based on this format, as will be explained in the test section.

3 Design Decisions

Throughout the implementation of both systemcalls, we have made some choices in the implementation.

3.1 Critical sections

Our choice of placement to enable/disable interrupts in *msgbox_put* is due to several reasons.

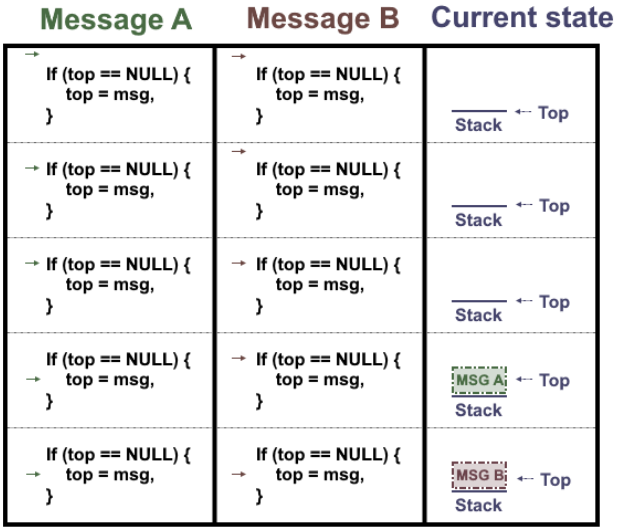

Figure 1: 2 messages inserted on empty stack.

1. To ensure that the structure of the stack is upheld, we ensure that only one message can be put onto the stack at a time. The reasoning behind this, is to ensure that if we have 2 messages being put on at the same time,

then we would like to avoid one message overwriting the other, despite not being the top. This is represented in a visual format, seen in figure 1.

2. The creation of the message node, *kmalloc* and copying from the user space are not included inside the critical section. This is because two processes which are trying to put a message on the stack, will not interfere with each other during these moments. Even if the messages are a copy of each other, they still get copied, and handled, as 2 separate messages.

The critical section in $msgbox\_get$ is designed differently.

1. Because we are changing the stack, we need to ensure that only one process will be touching it at a time. This helps ensure that two processes get two different messages, as once a message has been retrieved, it is then freed from the stack.

2. With processes guaranteed to have two different messages, we can avoid having our $'kfree'$ inside the critical section, as we will not have a situation in which 2 processes free the memory of the same message, and this helps avoid making the critical section too long.

4 Tests

4.1 Description & Motivation

We chose to create three different standard tests, followed by five programs. Each for error testing the systemcalls. Below is a short list of all eight programs, with a timestamp, to when they are shown in the video.

1. test1.c: Puts a msg in, and takes it out. (0:02)

2. test2.c: Puts 3 msg's in, and takes them all out. (0:07)

3. test3.c: Puts and takes 3 msg's interchangeably. (0:13)

4. nullBufferGet.c: Gives null-pointer to get systemcall. (0:21)

5. nullBufferPut.c: Gives null-pointer to put systemcall. (0:28)

6. invalidMessageLength.c: Gives -1 as message length. (0:33)

7. shortMessageLength.c: Gives message length - 5. (0:39)

8. longMessageLength.c: Gives message length + 5. (0:45)
9. shortGetBuffer.c: Gives too short a buffer to get systemcall. (0:56)
10. emptyMsgBox.c: Gets a message, without putting a message. (1:02)

4.1.1 Valid usage

The first test, *test*1.*c*, is the most simple test. It simply puts in the message: "This is an example message." and retrieves it again.

The second test, *test*2.*c*, first places in 3 messages, then retrieves 3 messages. This shows that the stack works, and that multiple messages can be placed in the queue. This also shows that the systemcall retrieves the top of the stack, showing that it runs by the "Last in, first out"-concept.

The last test, *test*3.*c*, of valid programs, first puts a message, then retrieves that message, and does so two more times. This is just to show, there are no problems with switching between putting and then retrieving messages.

4.1.2 Invalid usage

We will now explain the part of our test suite, that tests the error handling. We mainly test errors that occur when we give bad input parameters to the systemcalls, and when the stack is empty. We also have error handling for bad addresses, specifically when allocating memory is unsuccessful, however, it is rather difficult to force the *kmalloc* to fail.

The first error test is the $nullBufferGet.c$ program. Here, instead of giving the buffer to retrieve a message, we give NULL. This then causes the systemcall to have nowhere to copy the message from the kernel space. However, as it is the first thing we test in our systemcall, we immediately return EFAULT, due to the buffer address having no information, which is then printed to the terminal.

The second error test is the $nullBufferPut.c$ program. This mirrors the previous program, but we instead feed NULL into the "put" systemcall. Now it has nothing to copy from, but much like before, we check the parameters as the first thing, and immediately return EFAULT. Because there are no messages on the stack, we also return ENODATA from the "get" systemcall.

invalidMessageLength.c gives -1 as the length of the message. This would of course cause a crash for the kernel, and therefore we need to return EINVAL.

shortMessageLength.c, gives the length of the message to be too short. This does not crash either the kernel nor the user program. It does however fail to copy over the entire message to the kernel space, and therefore can not retrieve the entire message. The message is however, still removed from the stack.

longMessageLength.c gives the length of the message to be too large. This means we copy more than intended into the kernel space. This is a user error that would not cause the kernel to crash, and most likely we are just copying uninitialized memory, which the kernel will also copy back, when the message is retrieved. However, this also shows that it is impossible to copy memory from kernel space that is unintended. In the kernel there could be important data, that we do not want the user to have access to, however since we can only retrieve the the same bytes we copied into, we can not access uninitialized data from the kernel space.

shortGetBuffer.c gives a buffer that is too small, to copy the entire message from kernel. We check for this in the systemcall and return EINVAL.

emptyMsgBox.c tries to get a message before putting anything into the stack. The *msgbox_get* systemcall checks if the stack is empty, and then returns ENODATA. If a process is waiting for some data to be put into the message box, it can then just keep checking until it is not empty, or be put to sleep until we wake it.

5 Conclusion

With the help of our 2 systemcalls, we are able to create a message system through the kernel space to the user space so multiple processes can communicate. We have ensured that the messages are kept safe with our error handling for buffers, arguments and addresses. In addition, we ensure that concurrency can occur, with the help of critical sections, which allow us to avoid the loss of messages during sending and retrieving of these messages.