Assignment 1 OS implementing message box

Kristian Degn Abrahamsen, krabr21

16. March 2023

Introduction

In this assignment we are to implement a **message-passing** systemcall. This is a well known inter process communication strategy, for which two or more processes can communicate useful data through a system call implementation. One reason why one may want to use this message passing system call, is if another process has gotten some data, that it may have spent execution time on getting, then another process can get the data directly, without having to spend execution time on it itself, just by reading the data from the message box. The inter process communication is shown to be a very efficient way of using the machine, when used correctly. The message passing system call which is implemented in this assignment, is implemented via a stack-structure, storing strings. The implementation contains a system call for both writing to the stack, and reading of the stack. Throughout the call, errors may occur, which is passed on to the caller of the system call, which can be used for debugging purposes.

Design decisions

Data structure

In this implementation the stack has been used, to store the messages and to read the messages. This structure is easy to implement, and gives the necessary functionality that is required for a message-box. Other data structues which could be used, could be a simple array, however this would require the need for updating the list size, if the message-box got filled up, i.e this carries some additional overhead. A linked-list could also be used, but as we only want to have functionality for *push* and *pop*, the stack is the perfect data-structure.

Error messages

When using the system call, the user should be able to know, if something went wrong. This can then be used for debugging purposes. Giving the cor-

rect error messages, is then of course crucial, for figuring out what is happening during the underlying system call. Luckily, the linux kernel implementers has already thought of this importance, and has made some standard error-messages. These can be used by including *include/uapi/asm-generic/errno.h* and *include/uapi/asm-generic/errno-base.h*. When using these standard error messages, it is easier for the user, to know exactly what is happening, such that the user does not have to go into some implementation documentation, to figure out what fx. return value -102 means. The user can merrily include f.x *errno.h* and check if the return val is fx. *-EINVAL*, and know exactly what is going on. The user must then indeed be guided throughout the system call via error messages if something went wrong, or the system call may be deemed unreliable and or unusable. Therefore the implementation done in this assignment, makes sure the user is correct-fully informed if errors occur. Under this same topic, the implementation also uses kernel logging, such that additional information is given to the user, that cannot be given via just return values.

Concurrency

As the inter process communication strategy message-passing is expected to be used by multiple users, some security measures must be taken into account. This is necessary as when multiple processes wants to update the used data-structure concurrent, then race-conditions are created, which must be handled, to not leave the shared data-structure in an invalid state. I.e we have to ensure mutual exclusion for the processes. Luckily again, the linux-kernel implementers has created some tools, to let other implementers deal with mutual exclusion, which is used in this implementation. One of these tools is *local_irq_save()* and *local_irq_restore()*, which is used in the implementation. This window of mutual exclusion is preferred to be as small as possible, as it may stall other processes from progressing, if they also want to enter this same area of code, the window size has been taken into account in this implementation. This implementation does not make sure that if fx. process1 calls the *put()*-system call, and process2 calls the *put()*-system call, that process1's message will be put on first. This is a design choice to keep the mutual-exclusion section as small as possible, such that processes are not stalled. This of course has to be kept in mind, when using this message-box.

Implementation

msbbox_put

When a message is received via the sys_dm510_msgbox_put-system call, a pointer to the buffer storing this message, is given to the system call with its length. The length argument is used to know how much memory we have to allocate, for storing this new message onto the stack, and the buffer is of course given such that we can copy the message onto the stack. As this buffer is created

in user-space, which implies the buffer is stored using user-space addressing, it has to be mapped into the physical address in the kernel-space. First of all, it is checked whether it is safe to access this char-pointer, as if it is not, we should not access it from kernel-space. Secondly we also check whether or not, the length argument is less than 0, since a message cannot have a negative length, if it is negative, an error from *errno-base* is thrown corresponding to *"invalid argument"*. This way, if the user checks happened, the user sees that the argument it has provided is not valid. After the checks, the construction of a data-structure containing the message with its length and a pointer to the previous data-structure is created, via allocation of memory to this message, and copying the buffer. If the memory allocation fails one of the errors from errno-base is again thrown, *-ENOMEM*, is thrown to let the user know why the syscall failed. Copying from user-space addressing into kernel-space address is done via *copy_from_user(const char *to, const char *from, int length)*. This returns how many bytes not written into the buffer based on the length argument provided. If this call is unable to copy the entire message from *from*, this implementation chooses to write to the kernel using the log-level *KERN_WARN*, not update the stack, and will return *EAGAIN* which will prompt the user to try again. If however the call to copy the buffer goes through, we put the message on top of the stack. As we are updating the entire stack structure, we use the function *local_irq_save()* which in the specification of the linux-kernel, says that when calling this function, it will disable all interrupts of this process, and no other process can enter this critical section, until *local_irq_restore()* is called again. This will ensure, there will be no race conditions in this section. When the critical section is entered, we update the shared data structure i.e the stack. This will then of course put the message on top of the stack. When we are done updating the stack, we enable interrupts again, and as we successfully have put the message on top of the stacks, we return *0* indicating no errors occured, and the message given was put on top of the stack.

msgbox_get

This function will retrieve the top message in the shared stack structure, copy the message to the buffer given as an argument, and return how many bytes written into the buffer. This method will start by checking if there is a message available, which means that we are doing a check and then will execute something afterwards if the check passes. We disable interrupts before this check, making sure mutual exclusion is ensured. This will of course be done with the same methods used for mutual exclusion in the put-function. We start by making a pointer to the top, and update the top of the stack to the message below the message we just got a pointer to. Next up is checking if should access the user-space address provided with the buffer, from kernel space. If we should not, we return the error code *-EFAULT* which indicates we may not access this memory from kernel-space. If the check passes we will start the copying of the message from the stack into the buffer. If the user buffer provided is too small

for the message, the kernel will stop the application program, since it detects stack-smashing. If we are not able to copy the entire message we warn the kernel, writing to the kernel log file with debug-level KERN_WARN, providing how many bytes we could not copy. Then we restore the previous top and return *EAGAIN* signaling that the application program should try the method again. Since in the code block we enter if the message could not be copied correctly, is the last time we update the shared stack, we enable interrupts, such that another process can enter its critical section. Lastly we free the memory that we used to store the message with *kfree* i.e kernel-free, since the kernel created it with *kmalloc*, and return the number of bytes we copied into the buffer. If the overall stack structure was empty, we will just return -1 indicating the stack is empty.

Tests

For testing the system calls correctness I have made four test-methods and three helper methods.

Helper methods

empty_the_stack(), 0:48 in the video

This will be called at the top of all the test-method calls, such that we ensure that the stack is empty before we begin *putting* and *getting*, this is done since I test if the stack is empty before and after every method, ensuring correctness of the functiononality

put_message(), 0:51 in the video

This is just a helper method for putting a string using the system call, also asserting that the method was put correctly.

assert_empty(), 0:55 in the video

This is again a helper method for asserting that the stack is empty at the time the method is called.

help_concurrency(), 1:16 in the video

This method is used to repeatedly fork processes, creating 1024 processes which will concurently call to put a message on top of the stack, and then retrieve a message from the message box. We then test that the message has the expected length and contains the expected content. When putting the message we don't put unique messages, as this message box does not ensure that if process1 calls to put before process2, it will put its message on top of the stack before process2. This means we just test that no messages of the "hello" has been damaged in

any way. Each process which has created a process will wait for its children processes to end.

Test methods

test_put_and_get_with_output(), 0:58 in the video

This method is mainly for giving some actual visual output. It is asserted at the beginning that the stack is actually empty. As seen in the command prompt, we insert a message with a specific length, the response codes are asserted to be what we expect, i.e the put() gives the response 0 back, and the get message returns 26 as it should, as the message is actually that long. Lastly it is asserted that the message we put in is taken out by calling *assert_empty()*

test_put_gives_right_returnval_on_negative_length(), 1:03 in the video

This is to test the functionality in the syscall of *put()* as if a negative length is provided it should return the error code that an argument was invalid. This is tested by calling the *put()* system call with a negative length as argument, and we assert that we receive the error code *-EINVAL*. Next we assert that the message was not put on top of the stack, by asserting that the stack is indeed still empty.

test_stack_structure_works_as_expected(), 1:07 in the video

This tests is where two messages are put on top of the stack, by one process, i.e no processes are using the stack at the same time as this one. first we assert that the stack is empty, then two messages are put on top of the stack by the system call to *put()*, then we test that the lastly put message is retrieved first using the *get()* system call, asserting that we indeed got the last put message out first. The process is the same for first put message. Lastly we assert that the two messages put in has been taken out, and the stack is empty.

test_concurrency(), 1:20 in the video

This test starts out with ensuring that the stack is empty. We will then fork a process, and this child process will then continue to call *help_concurrency()*, which is explained above. It then waits for the child to finish. Lastly we assert that all messages has been taken out of the message box, ensuring that all messages that *help_concurrency()* has put in, also has been taken out. As *help_concurrency()* also asserts that the messages are as expected, if this test passes, we have ensured that concurrency works as expected.

Conclusion

A message-box system call has been implemented, where an application from user-space can access this message-box via system-calls to kernel-space. The

implementation makes it possible to use the same message-box by multiple processes concurrently, with no insurances of the order the messages will be put on the stack, because of performance measures. Tests were implemented, to make sure the system-call works as expected. This means that users can use this message-box as an inter-process communication, which can end up giving significant performance boosts if used right.