ASSIGNMENT 1 – SYSTEM CALL

DM510 - Mikkel Brix Nielsen (mikke21) in cooperation with Steffen Bach (stbac21) And Danny Nicolai Larsen (Dalar21)

Introduction

Inter-process communication (or IPC) is a very useful mechanism, which is necessary for different processes to be able to communicate. Communication between processes would allow for sharing data used for computation or ongoing operation between multiple processes, speeding up computation time by allowing a task to be broken into subtasks, which will execute in parallel with the others and can lead to a greater degree of modularity by dividing system functions into separate processes or threads.

One method for inter-process communication is message passing. In message passing processes sends small messages (byte array(s)) to each other using either the operating system or through some other message passing interface. This method is generally used in distributes systems were using shared memory for one reason or another is not possible.

The scope of this report will be to cover the design, implementation, and testing process of incorporating message passing as a message box into the 6.1.9 Linux kernel.

Design decisions

When designing additional functionality, features or modules, which are to be included in a kernel a few things must be taken into consideration. Due to the kernel being an essential piece of software that provides the user with the ability to interface with a systems's hardware and essentially holds control over everything in the operating system it is crucial to ensure reliability, robustness, and recoverability for the additions one makes to the kernel, such that a program or process failing does not result in a systemwide crash. Especially when dealing with additions that take user input as part of their execution. These inputs will have to be sanitized and their validity checked to ensure that they do not cause unforeseen effects.

Following the previously mentioned considerations a message passing system could be designed as a small message box residing in the kernel, which processes can write to or read from. This would require two methods, one for putting a message into the message box and one for taking a message out again. Since the message box resides in kernel-space the functionality would have to be implemented using system calls of which two are required, one for putting and another for getting a message from the message box.

For simplicity one could think of the message box itself as a stack of messages, following the last in first out convention (or LIFO), were if the message box is empty the message will be put into the message box with the subsequent following messages being put on top of the previous message. When retrieving a message, the process simply accesses the message box and retrieves the topmost message of the message stack reviling the next message underneath. This message is now the topmost message and the one which will be retrieved by the next process which attempts to retrieve a message from the message box. That is unless another process has put a new message into the message box before a retrieval occurs. Then that message would be the next to be retrieved.

To ensure that the message box can be used by multiple processes at the same time, without processes constantly interrupting each or potentially having two processes overwrite one another's messages while altering the orientation of the underlying stack in the message box.

To prevents this behaviour, one could imagine a process needing a key to unlock the message box before being able to retrieve a message from or put a message into the message box and that only one such key exists, so that only a single process can access the messages inside the message box at a time. This would prevent the scenario where while process (A) is in the middle of retrieving a message and altering the stack another process (B) also wants to retrieve a message and gets access to the message box and tries to retrieve the same message as (A) this could result in both processes retrieving the same message which is unintentional.

Considering two messages both wanting to put a message into the message box at the same time could also result in one of the messages being lost due to both processes setting the top message of the stack to be their own message, which results in the loss of the other message. Having to hold a key to access the message box would result in e.g., process (A) being the only process able to access the messages in the message box. This would ensure that process (B) cannot alter the stack structure in the message box before process (A) is done manipulating the stack and released control of the key.

As mentioned, one needs to be careful regarding input from user-space since calling a system call from kernel-space with improper variables given as parameters can result in undefined behaviour or outright crashes. Therefore, appropriate fail-safes must be put in place, such that the system calls cannot create an irrecoverable state from which the kernel has no other choice but to outright crash and require a reboot. In the context of a message box such fail-safes could include checking that it is okay to read from or write to the given buffer in user-space when retrieving a message from the message box or putting a message into the message box.

Furthermore, having a fail-safe for when the user-space defined length of the given buffer is less than the length of the message stored in the message box one could chose to only retrieve as much of the message that fits into the buffer or chose to return an error. For the design of this message box an error will be returned when calling the system call to get a message from the message box with a buffer that is too small to store the entire message within the message box this is to ensure that a message will always be read in its entirety.

Should the opposite be the case and the length of the buffer is greater than the message stored in the message box. To prevent overwriting other variables in user-space and ensuring that only the message is being retrieved from the message box and nothing else, only the message's size should be used when determining how much to read from kernel-space into the user-space buffer as to prevent one from reading the buffer's size from kernel space, which could result in data being retrieved which was not part of the original message.

Implementation

To implement a system, call firstly it needs to be defined with a unique identifier. This can be done by modifying the file unistd_64.h in the directory arhc/x86/include/generated/uapi/asm. In this file two lines need, be added one identifying the system call to put a message and one identifying the system call to get a message. Making sure to increment __NR_syscalls xxx to be one greater than the identifier for lastly defined system call. In this case that would be __NR_md510_msgbox_get with the identifier 453, so __NR_syscalls's identifier should be 454, which would look like the following:

```
#define __NR_dm510_msgbox_put 452
#define __NR_dm510_msgbox_get 453

#ifdef __KERNEL__
#define __NR_syscalls 454
#endif
```

Then a reference for each of the system calls must be made in the system calls table. This can be done by modifying the file syscall_64.tbl in the directory arch/x86/entry and adding a line defining the name of the system call for both get and put as well as the name of the method, which implements their respective functionality (written in C), which would look like the following:

```
452     common dm510_msgbox_put            sys_dm510_msgbox_put
453     common_dm510_msgbox_get            sys_dm510_msgbox_get
```

The next step is to define a header file e.g., dm510_msgbox.h describing the method signatures for the put and get methods. This file should be in the directory arch/um/include/asm and should include the two following signature: (1) extern int sys_dm510_msgbox_put(char*, int); and (2) extern int sys_dm510_msgbox_get(char*, int);.

Then to implement the actual C code for the message box and its put and get methods. A struct "_msg_t "can be defined, and type defined as "msg_t" to represent a message stored in the stack. This struct includes a "msg_t* previous" which is a pointer to the previous message, an "int length" describing the length of the message and a "char* message", which points to the memory address where the message is stored. The stack can then be implemented as a linked list with a pointer to the top message and the bottom message.

For implementing the put method (sys_dm510_msgbox_put) a check is made at the beginning to check the validity of the arguments if the given buffer pointer is equal to "NULL", or the length of the buffer is less than zero "-EINVAL" is returned to indicate invalid arguments. Then "kmalloc" is used with flag "GFP_KENREL" to allocate a message struct, if this fails "-EADDRNOTAVAIL" is returned to indicate that there is no available address. Otherwise, the message element is then initialized with the length given as a parameter as the message's length, its "previous" pointer is initially set to "NULL" and then "kmalloc" is used again with flag "GFP_KERNEL" to allocate space for storing the message being put into the message box. If the allocation fails "-EADDRNOTAVAIL" is returned. Now that the message has been initialized then access_ok" is used to check if it is safe to copy from users-pace to kernel-space if it is "copy_from_user" is used to copy the contents of the user-space buffer to the kernel-space memory address. If any of these two methods fail "-EFAULT" is returned otherwise interrupts gets disabled to ensure no other process can access the stack of messages in the message box using "local_irq_save", then "the message is put on top of the stack, then interrupts are restored using "local_irq_restore"and 0 is returned.

To retrieve a message the get method (sys_dm510_msgbox_get) checks that the user-space provided buffer is not "NULL" and the length is greater or equal to 0 if this is not the case "-EINVAL" is returned otherwise a check to see whether there is a message in the message box is made if there is no message in the message box "-ENODATA" is returned. Then interrupts are disabled using "local_irq_save", then a "msg_t*" is initialized that points to the topmost message in the message box and the length of the message is saved in a temporary variable, "mlength", used as a return value if "copy_to_user" is successful. Another check to see if the message can fit into the buffer is made. If the message does not fit "-EMSGSIZE" is returned, but if it can fit "access_ok" is used to check if it is okay to write to the buffer defined in user-

space. The method “copy_to_user” is used to copy the message from kernel-space to user-space. If either “access_ok” or “copy_to_user” fails interrupts are restored using “local_irq_restore” and “-EFAULT” is returned otherwise the message is taken out of the message box and the allocated kernel-space memory for the message struct and the buffer used to store the message in the struct is freed using “kfree”. Interrupts are then reenabled using “local_irq_restore” and the length of the read message stored in the temporary variable “mlength” is returned.

Finally, when everything else is in order and the main C implementation is placed in the directory arch/um/kernel the last thing that needs modification is the Makefile in the same directory where dm510_msgbox.o is added as a target. Then the kernel is recompiled from the source directory after which the system calls can be used in any user-space application / program.

Testing

For the first sets of tests, shown at 0:38 in the video, the motivation was to ensure proper behaviour when giving valid and invalid parameters. The first test regarding when the buffer given as a parameter is invalid, with both a valid, >0, and invalid, <0, length given as the other parameter. To represent an invalid buffer “NULL” was selected as the value to assign the buffer. When runninig both of the test trying to use “NULL” as a buffer with either valid or invalid buffer length were not allowed and both tests returned the error “Invalid arguments” (EINVAL) for putting the message into the message box. Trying to get a message out of the message box afterwards results in the error “No data available” (ENODATA).

The second tests, shown at 0:51 in the video, test for giving different buffer lengths as a parameter along with a valid buffer. The first test show the expected and actual results from giving a valid buffer and the buffers length + 1 (for terminating byte) – This results in the putting of the message returning “Success” and the getting of the message returning the full message in the specified recieve buffer along with returning the length of the message.

Giving a length that is less than the actual size of the buffer – This results in the putting of the message returning “Success” but only putting a part of the message into the message box, then getting the message returns that part of the message and not the entire message in the specified return buffer along with its length.

When testing for giving a length that is greater than the actual size of the buffer – This results in the putting of the message returning “Success” and putting the whole message specified by the buffer padded with zero bytes for the amount the specified length is greater than the actual message into the message box. The getting of the message returns the entire message in a specified return buffer and will report the original length given as a parameter to the “put” system call.

The test with a length that is invalid, <0, – This results in the putting of the message returning the error “Invalid arguments” (EINVAL) and the getting of the message, since there in none, returns the error “No data available” (ENODATA).

The last of this category of test was for what happens when a process tries to get a message when its receive buffer is smaller than the actual message in the message box – This results in the putting of the message returning “Success” and the entire message being put into the message box, for getting the message the result is the error “Message too long” (EMSGSIZE), since the message is too large to fit into the specified return buffer.

The third test is to test the behaviour regarding single threaded use, shown at 1:09 in the video, here a list of messages: "nuggets", "violent", "universal", "ending", "before", "zebras", "for", “daylight", "consuming", "inevitably", "are", "mechanics", "quantum", "perhaps", "you", "see", "to", "good", "world", "hello" are put into the message box and retrieved again. This results in the order of which the messages were put into the message box being reversed when retrieved from the message box, which matches the specified LIFO convention described in the design decisions section.

The fourth test, show in the video at 1:31, was made to check whether multiple processes can put messages into the message box without overwriting each other’s messages. In this test first 200 “put” threads were created, which each put 1000 messages into the message box. After all, “put” threads had put the messages into the messages box they were retrieved by a single thread which counted the number of messages in the message box. This number was then compared to the expected number of messages that should have been put into the message box. For good measure the test was repeated with a single thread putting the same number of messages into the message box and taking them out again. Both tests yielded an identical result in line with the expected result, which suggest that there is no sign of multiple threads trying to alter the underlying stack structure at the same time.

The fifth test was to ensure that the strategy used for implementing concurrency and multiple access to the message box was working properly, shown at 01:57 in the video. This test starts six threads three “put” threads, which puts messages into the message box and three “get” threads which retrieves messages from the message box. If the concurrency strategy was working since each “put” thread was started before a “get” thread it would put all messages into the message box before the “get” thread was able to take any of them out. A “get” thread would then get all the messages before the next “put” thread could put any new messages into the message box. But in practice this is not the case and the results range between the “get” threads trying to get messages before anything has been put into the message box to a “put” or “get” executing only to be interrupted by another thread before it is finished.

Multiple access

The intended outcome when two or more processes access the message box at the same time is that the process which got access to the message box first gains authority over it and blocks all other processes from accessing it until that process is done altering the stack within the message box. It then releases control over the message box and the next process to gain access to it blocks it from all other processes to ensure that no messages gets overwritten, lost or retrieved more than once unlike the example given in the design decisions section, where process (A) wants to retrieve or put a message into the message box whiles it does this process (B) does the same and ends up either retrieving the same message as (A) if they are both getting message or it might overwrite (A)’s message if they are both putting messages into the message box. But as the foruth test shows this does not seem to be a problem but another problem did occur as shown by the fifth test even though the fourth test makes it seem like the message box is concurrent regarding putting messages into and taking them out of the message box a process can still be interrupted while it is executing which could results in unwanted behaviour and concurrency bugs.

Conclusion

In closing, it can be said that the incorporation of message passing as a message box into the 6.1.9 Linux kernel was successful in the sense that it is robust against invalid parameters and messages can be put into and retrieved from the message box following the LIFO convention as long as only one thread / process is being used to both put the messages into as well as get the messages from the message box. When multiple threads / processes are involved in accessing the message box at the same time the outcome is non-deterministic as shown by the fifth test and the result can change wildly.