DM510 OPERATING SYSTEMS ASSIGNMENT 1

ESRA DURAN

I have completed this assignment with Tugra Demirel.

MARCH 16, 2023

SDU

In this project, we are asked to add a system call to our existing Kernel. Our aim is to send to or get some messages from other machines that use a different operating system or trough MPI (Message Passing Interface). MPI makes the communication trough messages in a standardized way between multiple computers that are running a parallel program. Our aim is to create 2 system calls.

- Get the message from the message box into user space

- Put any message to the message box trough the system-calling process.

We will implement the message box as a stack. As the purpose of the stack implementation requires, the last come message will be read at first. The first put message will be read at last. Also, each new message will be the top message of that stack.

The steps of that project can be listed as follows:

1. Define Entry Points

2.Implementation of Code

- Header File
- dm510_msgbox.c
- Makefile

3.Creating some tests for our system call

Before starting all the steps, we need to be sure that we have a running UML kernel system.

DEFINE ENTRY POINTS

For the first step, we need to put corresponding starting lines to the directory arch/x86/include/generated/uapi/asm/unistd_64.h . That is necessary for the kernel to recognize our system calls. In my case, I had 451 defined entries. I defined the putting message call as “__NR_msg_put” 452 and getting message call as “__NR_msg_get 453”. I have also increased the number of calls in kernel to 453 from 451.

Then, I updated the system call table located in /arch/x86/entry/syscalls/syscall_64.tbl .

I added “msg_put” as my first function named “dm510_msgbox_put”. And similirlay, I added “msg_get” for the second function “dm510_msgbox_get”.

So that, we have all the necessary entry points defined. Now, we can start the implementation of those functions.

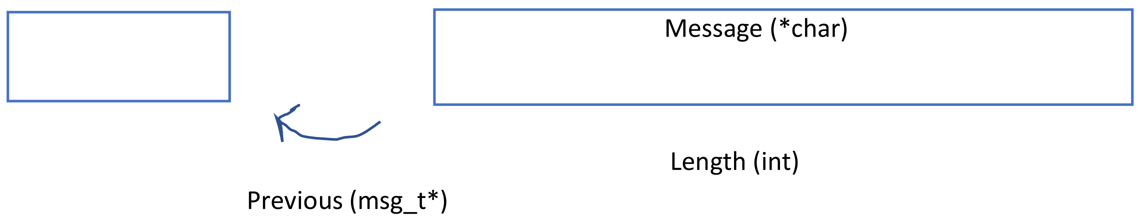
IMPLEMENTATION OF CODE

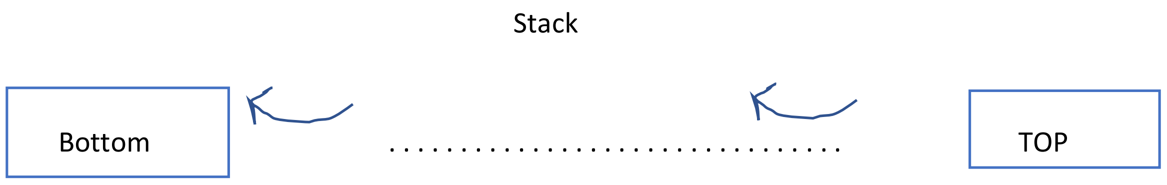
- *Header File*

As the first step of this part, we need to create a header file. I have created "dm510_msgbox.h" and bounded with my functions.

- *dm510_msgbox.c*

In this part, I followed the given C file structure with some changes. We have a type of struct named "msg_t"





We have set the bottom and top as null to create an empty stack.

-int dm510_msgbox_put is a function to write a message to a stack with parameters char* (message) and int (length of the message as characters).

Before taking the message we need to check whether getting the message that has "n" bytes from the user space is safe or not. To be sure of the safety, we used the "access_ok" function that takes address of the message and its size as bytes.

-If this function returns false, we exit the "msg_put" call by returning -EACCESS (negative value).

-If the translation is safe, we continue by allocating a space for the message.

For allocation step, we do not use the standard “malloc()” function. Instead of this, we used “kzalloc()” that takes an extra integer flags parameter. The two used parameters for this function are the size of the struct and the “GFP_KERNEL” for flag parameter. The allocation becomes a pointer shows to the beginning of that space.

In order to use “kzalloc()” we declared <linux/slab.h> . We did not use “kmalloc” because we did not want the messages overlap.

After that, we need to check whether the returned value of the allocation is null or not. If it was null, we exit the function by returning an invalid negative value.

If the space is allocated successfully, we have to define the components of this struct. Since this message will be the TOP, its previous should be null, and we set its size to the message size. Then, allocate a new space to the message itself. We apply the same steps for checking whether this allocation is successfully completed or not. If our requirements are not satisfied, we delete the allocated space by using “kfree()” function.

Then, we rearrange the bottom if this is the first message, or create a bound between the previous one if exists.

While those updates are happening, we need to be sure that there is not any interrupts. So, we used local_irq_save and local_irq_restore to emphasize the critical regions.

-int dm510_msgbox_get is a function to get a message from the stack with parameters char* and int (length).

This function returns the size of the last message (the TOP of the stack). If top is null (stack is empty), it returns an error value (-ENOMSG). If not, we need to further control the given length. It should not be less than the length of the TOP message. If so, we return -EMSGSIZE.

If all the conditions above are satisfied, we check the address safety as we did in the previous function by using access_ok() function. If the translation is safe, we need to delete this message from the stack. We arrange the TOP again and free the memory allocated for that last message.

- *Makefile*

In this file, we added “dm510_msgbox.o”.

TESTS

We have tested our systemcall with some scenarios.

1. In the first test case, we put 2 messages and get them respectively. When they are put to the stack, the 0 value is returned and whenever the “dm510_msgbox_get” function is called, their lengths are returned.

2. For the empty message, it successfully put and 0 value for the length is returned.

3. In test3.c file, we tried to put “NULL” for the message, but it gave us a buffer overflow.

4.In test4.c, I tried to enter some negative value instead of its length, and it returns very high numbers.

Multiprocessing systems are sharing the same memory while they have their own CPU’s. In that case, our program can be used by a multiprocessing system. Accessing to the message box requires to use put or get functions. In the time of using those, we have local_irq_save and local_irq_restore implementations. We can temporarily disable all the interrupts happens at that accessing time. So, if one machine is currently writing or getting any message, this process cannot be interrupted. The others will be waiting to access or make changes on the message box.

In this project, we saw how to use systemcall on linux. I fully understand how those processes works and what can be used in order to organize the orders of the calls by users and machines (such as multiprocessing systems). In user mode, we can easily see and use the functions directly without giving harm to them. OS is the bridge of the user and computer, and we practiced its fundamentals.