DM510 – Assignment 2

Andreas Lynge (anlyn11)
171287-1985

February 18, 2019

Contents

1 Introduction

In this report we are going to discuss the design decisions and implementation of the *sys_ dm510_ msgbox_ put* and *sys_ dm510_ msgbox_ get* linux system calls. These system calls can be used for interprocess communication (IPC). The *sys_ dm510_ msgbox_ put* system call is used to push a message on top of a stack in kernel space, and the *sys_ dm510_ msgbox_ get* is used to pop and return the message on top of that stack.

The first thing we are going to discuss is some important design decisions which was made before implementing the system calls. Next we will discuss the implementation details and lastly we will discuss how the system calls was tested.

2 Design Decisions

The system calls are declared as follows

```
asmlinkage long sys_dm510_msgbox_put(const char __user *buf, long size);
asmlinkage long sys_dm510_msgbox_get(char __user *const buf, long size);
```

The first parameter (*buf*) of *sys_ dm510_ msgbox_ put* is the message the user wants to push on top of the stack and the second parameter (*size*) is the length of the message. The first parameter (*buf*) of *sys_ dm510_ msgbox_ get* is a pointer to a buffer to store the messsage from the top of the stack, and the second argument (*size*) is the size of the buffer.

Since there can be multiple processes trying to push and pop messages from the stack at any given time, to avoid corruption of the stack and memory leaks, we have to synchronize access to the stack. A possible implementation of *sys_ dm510_ msgbox_ get* might be

```
1. call access_ok(VERIFY_WRITE, buf, size);
2. if access is not ok return error code
3. start synchronized code
4. pop message from stack
5. end synchronized code
6. copy message to buf and free the message from kernel memory
7. Return success
```

If we ingnore verification of parameters, this looks quite ok, right?

The first thing we do is call *access_ ok* to check whether *buf* is a valid user space pointer. On the x86-32 architecture essentially what *access_ ok* does is check whether the virtual addresses from *buf* to *buf + size* is within the user process address space, which normaly is virtual addresses ranging from 0 to 0xBFFFFFFF, that is the first 3GB of virtual memory. In steps 3 to 5 we pop the message from the top of the stack in a safe synchronized way. And lastly in steps 6 and 7 we copy the message to the user and return success.

However there is a problem. The kernel function *access_ ok* does not check whether *buf* is accually allocated to the process. Thus copy messege to user space might fail even if *access_ ok* returned success. And if this is the case the message is simply lost. One way to solve this problem is to make sure the message is copied to user space before we pop it from the top of the stack.

On unicore CPUs a common synchronization method is to disable interrupts. Thus assuming the target CPU is unicore, another implementation might look like

```
1. disable interrupts
2. retreve the message from top of stack without removing it
3. call copy_to_user(buf, message, size)
4. if copy failed enable interrupts and return error
5. pop message from top of stack to free it from kernel memory
6. enable interrupts
7. return success
```

Again we will ignore parameter verification. The first thing we do here is disable interrupts so the kernel control path will not get preempted/suspended by an interrupt. Afterwards (in steps 2 to 5) we make sure the message is copied to user space before popping the message from the stack. We do not need to call *access_ ok* because *copy_ to_ user* does this for us. Assuming the message was succesfully copied to the user, the next thing we do (in steps 6 to 7) is enable interrupts and return success.

If the target CPU is unicore, this might look ok at first glance. However not all x86 interrupts are maskable (disableable). These interrupts are often refered to as traps or exceptions because programmers can determine when they will occur. An example of such a trap is "Page Fault".

The problem is that *copy_ to_ user* might issue a "Page Fault" and thus preempt the current kernel control path. A "Page Fault" will occur when a process tries to access virtual memory which is not part of the processes address space, if the process tries to write to a read-only page or when a requested page is not present in memory.

If *copy_ to_ user* issues a "Page Fault" because a requested page is is not present in memory, the current kernel control path will be preempted while the requested page is being swapped in. In the meantime another process might call *sys_ dm510_ msgbox_ get* to get a message. This process will get the same message as the previous process, which was not the intention!

While disabling interrupts might be an easy and efficient way to protect critical sections on unicore CPUs, it does not work in this case. The solution is a more sophisticated synchronization method.

One simple way to achieve this is to use a single semaphore, call the *down* function in the beginning of the system calls, and call the *up* function before returning from the system calls. This surely guarantees atomicity, and might be a good solution, depending on the intent of the system calls.

Our solution is going to be slightly more complex however. We will use a spin lock to ensure atomic operations on the stack, and use a semaphore to synchronize processes calling the *sys_ dm510_ msgbox_ get* system call. Without error handling *sys_ dm510_ msgbox_ get* should do the following

```
1. semaphore down
2. atomically retreve the message from top of stack without removing it
3. call copy_to_user
4. atomically remove the message from top of stack to free it from
    kernel memory
```

```
5. semaphore up
6. return success
```

We use a semaphore to synchronize here because, as we have discussed earlier, *copy_ to_ user* might block, and thus busy waiting is out of the question.

The *sys_ dm510_ msgbox_ put* does not need to be protected by the semaphore. It will just push a new message on top of the stack, so we only need to make sure all stack operations are atomic. A spin lock is perfect for this purpose since all the stack operations we are going to perform will not block and they will take short constant time.

This last example reminds of the final implementation of *sys_ dm510_ msgbox_ get* which we will discuss in the next section.

3 Implementation

We have 1 file ("dm510_msgbox.c") which defines the system calls. We want to be able to test and use the system calls with x86-32 user mode linux and also with x86-32 linux running on a virtual machine. Thus we have placed "dm510_msgbox.c" in the directory "arch/x86/kernel/" and edited "arch/x86/kernel/Makefile" and "arch/x86/um/Makefile" so they will compile and link the system calls into the x86-32 and the x86-32 user mode linux kernels.

We have also added the system calls to the x86-32 specific "arch/x86/syscalls/syscall_32.tbl" file. And declared the system calls in the "arch/x86/include/asm/syscalls.h" file.

Now let us discuss the "dm510_msgbox.c" file. The *struct dm510_ msg* (*dm510_ msg_ t*) is used to store messages from user space, it contains the following fields

Type	Name	Description
*const char **	*buf*	Buffer containing the messasge from user space
long	*size*	Size of the buffer
struct list_ head	*stack_ entry*	Pointers to previous and next message on the stack

There are 3 stack operations and they are implemented as follows

```
static inline void dm510_list_atomic_push(dm510_msg_t *msg)
{
        spin_lock(&list_lock);
        list_add(&msg->stack_entry, &list_head);
        spin_unlock(&list_lock);
}

static inline void dm510_list_atomic_remove(dm510_msg_t *msg)
{
        spin_lock(&list_lock);
        list_del(&msg->stack_entry);
        spin_unlock(&list_lock);
}
```

```
static inline dm510_msg_t *dm510_list_peek(void)
{
	/* No need to lock this. Either we use an old list_head.next or
	 * we use a new list_head.next. Doesn't matter it's valid */
	return dm510_msg_of(list_head.next);
}
```

The operation of the functions should be self explanatory. All the stack operations are thread safe, because access to *dm510_list_atomic_remove* and *dm510_list_peek* is synchronized as we will see later. The macro *dm510_msg_of* returns the *dm510_msg_t* containing the *struct list_head* given as argument.

Without error handling the operations made by *sys_dm510_msgbox_get* system call is as follows

```
1. down(&dm510_sem)
2. msg = dm510_list_peek()
3. copy_to_user(buf, msg->buf, msg->size)
4. dm510_list_atomic_remove(msg)
5. up(&dm510_sem)
6. ret = msg->size
7. dm510_msg_destroy(msg)
8. return ret
```

In line 5 we release the semaphore when we are sure the message has been copied to user space. In line 7 we free the *dm510_msg_t* from kernel memory. And line 8 returns the size of the message. Note that *dm510_list_peek* and *dm510_list_atomic_remove* will not be called concurrently, the calls are synchronized by the *dm510_sem*. There is a huge lack of error handling here, but it serves as a good explanation of the function. The following error codes can be returned from the function

- -*EINVAL* if the size of the buffer is less than 0
- -*EMSGSIZE* if the size of the message on top of the stack is bigger than the size of the buffer (*buf*).
- -*EFAULT* on segmentation violation
- -*ENOMSG* if there are no messages on the stack

Even though we do not do so, it might be a good idea to send a *SIGSEGV* signal to the process (with *force_sig*) when segmentation violations occur.

Without error handling the *sys_dm510_msgbox_put* system call does the following

```
1. msg = dm510_msg_init(size)
2. copy_from_user((char *) msg->buf, buf, size)
3. dm510_list_atomic_push(msg)
4. return 0
```

In line 1 it allocates and initializes a new *dm510_msg_t*. In line 2 we copy the user space message to the *dm510_msg_t* , and in line 3 pushes the *dm510_msg_t* on the stack. And in line 4 returns 0 (success). The function may return the following error codes

- -*EINVAL* if the size of the buffer is less than 0
- -*EFAULT* on segmentation violation
- -*ENOMEM* when out of kernel memory

For more information about the implementation please take a look at the *dm510_ msgbox.c* file.

4 Testing

After the through design and implementation phases, if the kernel compiles correctly we are pretty confident the code works correctly.

However we do have two C programs to test the system calls. The first test program ("testit.c"), is a simple program to put and get messages from the stack. And the second program ("fork_test.c") is a program implemented to stress test the system calls, by forking processes which put and get messages.

In the supplementary video there is a demonstration of these test programs. User mode linux starts to crash whenever we fork many processes. Thus we will use a Qemu virtual machine with 4 CPUs to test the "fork_test.c" program. The video also demonstrates what happens when 500 processes are forked in user mode linux with the "fork500.c" program.

5 Conclusion

Through this process we have learned that kernel programming is at least an order of magnitude harder than user mode programming. Kernel programming requires more knowledge and better understanding.