0.1 Introduction

The aim of this project is to implement two system calls to handle interprocess communication. This is achieved by implementing a message box in kernel space. The system calls must be able to retrieve a message from kernel to user space and place a message in the message box from user space. The message box is implemented by a stack where incoming messages are placed on the top and messages are retrieved from the top.

0.2 Design Choices

During the process of implementing the project there where multiple design choices to be made.

For example was it required to implement the message box using a stack were new messages were put on top of the stack by the system call and the top message would always be retrieved by the other system call. Using a stack in this way makes it easier to retrive messages as there is no need to look for specific messages. While this is great from an implementation point of view, because it makes it easier to implement, it also puts restrictions on how we can use the message box.

The two system calls both take a pointer to a buffer. For the system call that retrives a message, the buffer is the pointer that points to where the message should be copied to from. For the system call that stores a message on the stack the pointer points to the message that we want to store. They also both have the parameter length which is an integer that defines the size of the buffer.

Another design choice is how big the messages can be that that can be stored on the stack in kernel space. In this implementation one message is limited to 101 characters. 101 was chosen because it allows to have 100 characters and the 0 terminator. Messages can also not be shorter than one character as a message with less than that would not make any sense and just cause trouble.

For error handling the error codes for system calls are used. These error codes are defined in the file "errno.h" in the Linux source code. These error codes are negative integers that are returned. An example use is for example the use of "-EINVAL" which is used if an invalid argument is passed. For example when the message is to long or to short. Another one is "-ENOMEM" which is used when "kmalloc" was not able to allocate memory for the message. "-EFAULT" is used when "access_ok" checks if it is safe to read or write from user space memory.

One important design choice is how to handle simultanius acces to reasources of the message box. If this is ignored the systemcall could be interrupted while in a critical region, causing problems. So it is important that data is accesed and modified amoticaly to prevent race conditions. To deal with this, interrupts can be temporarily disabled in specific regions of the code. This is done with "local_irq_save(flags);" and "local_irq_restore(flags);". These have to be used with caution and not unnecessarily often as critical systems like the cpu clock run on interrupts. If interrupts are disabled the cpu clock could get out of sync. This could have repercussions for other parts of the system. These function disble interrupts when specific parts of the code are running. In this case specificaly operations involving memory. Disabeling interrupts is in this case probably the more efficent approach if compared to locks. Locks require more overhead but are also safer to use in comparison to disabeling interrupts. But as the critical section in this programm are rather short interrupts are disabled only for a short amount of time.

0.3 Implementation

Because we are working in kernel space the implementation differs slightly from what it would look like in user space. For example "kmalloc" has to be used instead of the normal "malloc". The main differences between "kmalloc" and "malloc" are that "kmalloc" is specifacly for kernel memory. "kmalloc" also requires more paramaeters than "malloc", it needs a flag that tells it how to allocate memory. In this case "GFP_KERNEL" is used. This flag makes "kmalloc" wait until memory is availiable if there is none. There are also flags that would return instantly if no memory was available. Below is an example of how "kmalloc" is used. As can be seen it does not differ much from the normal "malloc".

```
msgStack* msg = kmalloc(sizeof(msgStack), GFP_KERNEL);
if(msg == NULL){
    return -ENOMEM;
}
```

As seen in the example above error handling is also close to normal C code, but system call error codes are used.

With great "kmalloc" there comes great "kfree". As this is real programming we need to free the memory we allocate using "kmalloc". For that we use "kfree", it is the kernel equivalent to the normal "free" it is used just like the normal C "free". It just frees memory in kernel space. Here is an example from the project code

```
kfree(msg -> message);
kfree(msg);
```

Because interacting with user space memory from kernel space can give complications the "access_ok" function is used to prevent memory violations. Before reading from or writing to user space "access_ok" checks if the adress is viable and accesible thus protecting the kernel from accessing illigal memory. Below is an example from the "dm510_msgbox_put" system call where we check if the location is safe before "copy_from_user()" is used to copy from user memory.

```
if(!access_ok(buffer, length)){
    return -EFAULT;
}
int copyUser = copy_from_user(msg -> message, buffer, length);
if(copyUser != 0){
    return -EFAULT;
}
```

the functions "copy_from_user" and "copy_to_user" are used to move data between user and kernel space. "copy_from_user" transfers data from user space to kernal space. "copy_to_user" copies data from kernel to user space. An example from the code is shown below. as arguments both functions take a pointer to were the data should be moved and from were aswell as an integer representing how many bytes should be moved. Both functions return zero on succses which can be used for error handling.

```
int copyUser = copy_from_user(msg -> message, buffer, length);
if(copyUser != 0){
    return -EFAULT;
}
```

```
int cpyToUser = copy_to_user(buffer, msg->message, msgLength);
if(cpyToUser != 0){
    return -EFAULT;
}
```

As mentioned before the message box uses a stack to manage messages new messages are put ontop of the stack. The top message is also always the first one to be retrieved. When pushing a message onto the stack it becomes the new top. We also check if the stack is empty, if it is the pushed message also becomes the bottom. To keep track of the messages below top, every message also refrences the privius message. If the message was the first message on the stack the privius message is set to "NULL" otherwise the new messages privius pointer points to the old top message. If the stack is not empty the pushed message becomes the new top. The push operation is shown below.

```
//if the stack is empty
if(bottom == NULL){
    bottom = msg;
    top = msg;
    msg -> previous = NULL;
}
// if stack is not empty
else{
    msg -> previous = top;
    top = msg;
}
```

For the pop operation the message top points to is removed from the stack. Because every message points its privius message, the privius message top points to becomes the new top. If top becomes "NULL" becuse the last message has been retrieved we return from the function. The implementation is shown below.

```
if(top != NULL) {
    msgStack* msg = top;
    int msgLength = msg->length;
    top = msg->previous;

    // some other stuff
}
```

The code handles multiple error conditions to make the program as safe and resistent as possible. If the message to be put in the message box is to long or to short we gracefully return the appropriat system call error code as shown below.

```
if(length <= 0){
    return -EINVAL;
}
if(length > 101){
    return -EINVAL;
}
```

Memory allocation failures are also handled with the appropiate error code as shown below.

```
msgStack* msg = kmalloc(sizeof(msgStack), GFP_KERNEL);
if(msg == NULL){
    return -ENOMEM;
}
```

Access validation error handling is also implementet to handle if memory access operations fail or are invalid, aswell as protection against reading or writing outside allowed memory locations. This is also shown below

```
if(!access_ok(buffer, length)){
    return -EFAULT;
}
int copyUser = copy_from_user(msg -> message, buffer, length);
if(copyUser != 0){
    return -EFAULT;
}
```

Another important part of the program is the use of flags to disable interrupts. This is done for parts that access memory or shared data to protect data integrety and prevent race conditions. Below is a section that uses the flags to disable interrupts.

```
unsigned long flags;
local_irq_save(flags); //start of critical region

//if the stack is empty
if(bottom == NULL){
    bottom = msg;
    top = msg;
    msg -> previous = NULL;
}
// if stack is not empty
else{
    msg -> previous = top;
    top = msg;
}
local_irq_restore(flags); // critical region end
```

These code snippets show the important and interesting parts of the program

0.4 Testing

0.5 Conclusion

0.6 Apendix