DM510 – Operating Systems

Assignment 1: System Call

Yunus Emre Balci (yubal22)

[yubal22@student.sdu.dk](mailto:yubal22@student.sdu.dk)

Jenin Imad El-Merie (jeelm22)

[jeelm22@student.sdu.dk](mailto:jeelm22@student.sdu.dk)

Salma Rashid Jamale Khair (sahka22)

[sahka22@student.sdu.dk](mailto:sahka22@student.sdu.dk)

Southern University of Denmark

March 2nd. 2024

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Contents

[1. Introduction 3](#_Toc160201949)

[2. Design decisions 3](#_Toc160201950)

[3. Implementation 3](#_Toc160201951)

[- Message Structure 4](#_Toc160201952)

[- Initialization and Concurrency Control 4](#_Toc160201953)

[- Adding Messages to the Message Box 4](#_Toc160201954)

[- Retrieving Messages from the Message Box 5](#_Toc160201955)

[- Cleanup 6](#_Toc160201956)

[4. Test 7](#_Toc160201957)

[- Valid Parameter Testing 7](#_Toc160201958)

[- Invalid Parameter Testing 7](#_Toc160201959)

[5. Conclusion 8](#_Toc160201960)

[6. Appendex 9](#_Toc160201961)

# Introduction

This assignment is made in collaboration with Jenin Imad Merie (jeelm22) and Salma Rashid (sakha22).

Our goal for this assignment is to add two system calls to User-mode Linux (UML) to implement a message box in the kernel space. These system calls are designed to work with a message box located within the kernel’s memory space, allowing different processes to send and receive short messages or byte arrays to one another. This message box is implemented as a stack, meaning that messages to be stacked on top of each other as they come in. They’re also read from the top as well. It is a straightforward, simple, and yet effective way to manage interprocess communication (IPC).

# Design decisions

As mentioned in the assignment, the decision to implement the message box as a stack is because of the simplicity and efficiency. The design we use, messages are added to the top of the stack with **sys\_dm510\_msgbox\_put** system call by pushing and retrieved from the top using the **sys\_dm510\_msgbox\_get** system call by popping. This is called a LIFO, Last In, First Out. This approach ensures that the most recent message is always accessed first, making it easy and quick to access messages and simplifying how messages are managed.

An important element of our design is our concurrency management. It is implemented through a spinlock, **msgbox\_lock.** This ensures that when one process executes the **sys\_dm510\_msgbox\_put,** to push a message, no other process can interrupt or access the stack simultaneously. Spinlock mechanism is important when multiple processes simultaneously attempt to access the message box. By making sure only one process can access the message box at a time, we prevent messages from corruption, mixing or even loss that could occur with concurrent accesses. Choosing a spinlock helps us deal with these tasks quickly and keeps messages safe and in order, even when lots of processes are working at same time.

# Implementation

The implementation of the message box module in the Linux kernel provides an efficient way for processes to exchange messages.

## Message Structure

We define a message structure, **msg\_t**, to hold the individual messages in the messages in the message box. This structure includes pointers to manage a stack-based list, the message length, and the message content:



## Initialization and Concurrency Control

Upon module initialization, we set up a spinlock, **msgbox\_lock**, to control access to the message box. This spinlock is important for ensuring that operations on the message box are mutually exclusive, preventing data loss and corruption:



## Adding Messages to the Message Box

**sys\_dm510\_msgbox\_put** system call adds a new message to the top of the stack. It first checks for valid message length and user-space pointer, allocates memory for the message structure and content, and then copies the message from user space to kernel space. It uses the spinlock to safely add the message to the stack:



## Retrieving Messages from the Message Box

**sys\_dm510\_msgbox\_get** system call retrieves and removes the top message from the stack. It locks the message box, takes the top message, updates the stack pointers, and then copies the message back to the user space:



## Cleanup

When the module is at it last steps to exit, we are ensuring that any remaining messages are freed properly to prevent any memory leaks. This involves navigating the message stack and freeing each message’s structure and content:



# Test

To ensure that our message box module, **dm510\_msgbox\_module.c,** functions correctly, we’ve created a .c-file called **dm510\_test.c.** Our test is implemented in a way that is handling both valid and invalid call parameters effectively.

## Valid Parameter Testing

1. **Functionality:** **dm510\_test.c** is a testing file to verify the essential operations of message sending, **sys\_dm510\_msgbox\_put**, and retrieving the message, **sys\_dm510\_msgbox\_get**. We send a predefined message and ensuring that the message is accurately stored and could be retrieved.
2. **Concurrent Access:** To access the module’s behavior under concurrent usage, we have created multiple processes that simultaneously sent and retrieved messages. This test is also important for validating the effectiveness of the spinlock mechanism that is implemented, due to ensuring data consistency, so we do not lose any data.

## Invalid Parameter Testing

1. **Negative Length and Oversized Messages:** In such cases, where there is occurring a negative message length or oversized message, our test is designed to reject the operation immediately, returning an error code about invalid input parameter. Our test verifies the module’s ability to handle errors, ensuring that it does not attempt to process or allocate resources based on incorrect specifications of the message.
2. **Exceeding Maximum Message Size:** When attempting to send messages that surpass the predefined maximum allowable size, which is 4096 bytes. The expected outcome is denying accepting the message and gives an error indicating that the input exceeds allowable bounds. Our test ensures the module does not over-allocate memory or compromise system stability due to oversized inputs.
3. **Invalid Memory Addresses (NULL Pointers):** When submitting a NULL pointer as the message buffer, it simulates a programming error. The test’s response is validating the memory address before proceeding with any operating. Therefore, it is preventing attempts to access or modify memory locations that are undefined or reserved. This is important due to system security and stability.
4. **Retrieval from an Empty Message Box:** The test tests the system’s behavior when a retrieval operation, **sys\_dm510\_msgbox\_get,** is requested, but the message box is empty. When a scenario like this occurs, the system returns an error code.

# Conclusion

As of conclusion, the DM510 message box module works, allowing messages to be sent and read within the kernel. It handles multiple users at once without any problems, due to spinlock mechanism, and deals with errors, empty messages correctly. Built as a stack, it makes sure the latest message is always at the top. Our project and test goals have been reached and therefore we can conclude that our project meets every goal.

# Appendex

**dm510\_msgbox\_module.c:**

****

****

**dm510\_test.c:**

****

****

**dm510\_msgbox.h:**

****