Operating Systems
Assignment 2 Report

Advanced IPC Mechanism
Priority Message Queue

(Option B)

Submitted by:

Taimoor Safdar	463920
Abdul Raheem	469273
Ahmed Raza	465607
Hashmat Raza	463025

Course: Operating Systems
Class: BSCS-13
Section: E

Date: November 30, 2025

GitHub Repository:

`https://github.com/wizz-ctrl/Ipc-priority-message-queue-.git`

Contents

1 Setup Guide

1.1 Development Environment

1.1.1 Operating System and Kernel

The implementation was developed and tested on the following platform:

- **Operating System:** Ubuntu 24.04 LTS (Long Term Support)
- **Kernel Version:** Linux 6.14.0-35-generic
- **Architecture:** x86_64 (64-bit)

The kernel version was verified using the command:

```
uname -r
# Output: 6.14.0-35-generic
```

1.1.2 Tools and Utilities Used

The following development tools were utilized:

- **Compiler:** GCC (GNU Compiler Collection) version 13.3.0
- **Build System:** GNU Make 4.3
- **Kernel Headers:** linux-headers-6.14.0-35-generic
- **Version Control:** Git 2.43.0

1.1.3 Required Dependencies

Before building the project, the following packages must be installed:

```
# Update package database
sudo apt-get update

# Install build essentials and kernel headers
sudo apt-get install build-essential
sudo apt-get install linux-headers-$(uname -r)
```

Verification of installed dependencies:

```
# Verify GCC installation
gcc --version

# Verify kernel headers
ls /lib/modules/$(uname -r)/build
```

1.2 Compilation Instructions

1.2.1 Building the Kernel Module

The kernel module is compiled using the standard kernel build system (kbuild):

```
# Navigate to project directory
cd /path/to/os-assignment2

# Compile kernel module
make
```

This generates the file `safe_lkm.ko`, which is the compiled loadable kernel module.

1.2.2 Building Test Programs

The test programs are compiled using GCC:

```
# Compile individual test programs
gcc -o test_basic test_basic.c
gcc -o test_stress test_stress.c
gcc -o test_edge test_edge.c

# Or use the provided script
./compile_tests.sh
```

1.3 Running the System

1.3.1 Module Installation

```
# Load the kernel module
sudo insmod safe_lkm.ko

# Verify module is loaded
lsmod | grep safe_lkm

# Check kernel logs
sudo dmesg | grep safe_lkm | tail -10
```

1.3.2 Executing Tests

Run the three test programs in sequence:

```
# Test 1: Basic functionality
sudo ./test_basic

# Test 2: Stress testing
sudo ./test_stress

# Test 3: Edge cases
sudo ./test_edge
```

1.3.3 Module Removal

After testing, unload the module:

```
# Unload kernel module
sudo rmmod safe_lkm

# Verify clean removal
sudo dmesg | grep safe_lkm | tail -5
```

2 Implementation Summary

2.1 Implementation Approach: Loadable Kernel Module (LKM)

2.1.1 Reasoning for LKM Approach

For this assignment, we chose to implement the IPC mechanism as a **Loadable Kernel Module (LKM)** rather than modifying the kernel source code directly and recompiling the entire kernel. This approach was selected for several critical reasons:

1. Development Efficiency and Safety:

Modifying the kernel source code and recompiling the entire kernel is a time-intensive process that can take 30-60 minutes per compilation. Additionally, any errors in the code could render the entire system unbootable, requiring recovery procedures. LKMs allow us to develop, test, and debug iteratively without risking system stability.

2. Dynamic Loading and Unloading:

LKMs can be loaded into the running kernel using `insmod` and unloaded using `rmmod` without requiring a system reboot. This enables rapid testing cycles and immediate verification of code changes, which is essential for educational purposes and development workflows.

3. Equivalent Functionality:

Despite being external modules, LKMs operate with full kernel privileges and have complete access to kernel APIs, data structures, and hardware. Our IPC implementation using an LKM provides *exactly the same functionality* as if it were compiled directly into the kernel. The module can:

- Access kernel memory space and use kernel allocators (`kmalloc`/`kfree`)
- Utilize kernel synchronization primitives (spinlocks, mutexes)
- Create interfaces in `/proc` filesystem
- Interact with process structures and scheduling
- Perform all operations that built-in kernel code can perform

4. Modularity and Maintainability:

LKMs promote modular design by keeping new functionality separate from the core kernel. This makes the code easier to maintain, update, and distribute. The module can be shared and tested on different systems without requiring each system to rebuild its kernel.

5. Educational and Production Use:

Many production systems use LKMs for device drivers, filesystem modules, and network protocols. This approach mirrors real-world kernel development practices, making our implementation both educationally valuable and practically relevant.

Conclusion: The LKM approach serves the same purpose as kernel source modification while providing superior development agility, safety, and maintainability. Our implementation demonstrates that complex kernel features can be added dynamically without compromising functionality or performance.

2.2 Feature Selection

For this assignment, we implemented **Option B: Advanced Inter-Process Communication (IPC) Mechanism** with a priority-based message queue system. This option was selected because it provides practical experience with:

- Kernel-level data structure design

- Thread synchronization mechanisms
- User-kernel space communication
- Dynamic memory management in kernel space

2.3 High-Level Design

2.3.1 Architecture Overview

The implementation consists of a loadable kernel module (LKM) that provides an IPC mechanism accessible through the `/proc` filesystem. The system uses dual-priority queues to manage messages:

- **High Priority Queue:** For messages with type ≥ 5
- **Normal Priority Queue:** For messages with type < 5

Messages are delivered in priority order, with FIFO ordering within each priority level.

2.3.2 Key Components

1. Data Structures

Two primary structures were designed:

```
struct demo_msg {
    pid_t pid;           // Process ID
    int type;            // Message priority type
    char text[256];      // Message content
    struct list_head list; // Kernel list node
};

struct demo_msg_queue {
    struct list_head high_priority;   // High priority list
    struct list_head normal_priority; // Normal priority list
    spinlock_t lock;                  // Synchronization lock
};
```

2. Core Operations

- `demo_send_msg()`: Adds message to appropriate queue based on priority
- `demo_receive_msg()`: Retrieves highest priority message (non-blocking)
- `proc_write()`: Handles user commands (S for send, R for receive)
- `proc_read()`: Displays queue status and statistics

3. User Interface

The module creates `/proc/safe_lkm` for user interaction:

```
# Send message: S <pid> <type> <message>
echo "S␣1001␣10␣HighPriorityMessage" > /proc/safe_lkm

# Receive message: R
echo "R" > /proc/safe_lkm

# View queue status
cat /proc/safe_lkm
```

2.4 Implementation Details

2.4.1 Thread Safety

All queue operations are protected using spinlocks with interrupt safety:

```
unsigned long flags;
spin_lock_irqsave(&msg_queue.lock, flags);
// Critical section
spin_unlock_irqrestore(&msg_queue.lock, flags);
```

This ensures safe concurrent access from multiple processes.

2.4.2 Memory Management

Dynamic memory allocation using kernel allocators:

- `kmalloc(GFP_KERNEL)` for message allocation
- `kfree()` for proper deallocation
- Automatic cleanup on module unload prevents memory leaks

2.4.3 Priority Logic

Messages are classified and queued based on type value:

```
if (msg->type >= 5) {
    list_add_tail(&msg->list,
                  &msg_queue.high_priority);
} else {
    list_add_tail(&msg->list,
                  &msg_queue.normal_priority);
}
```

Retrieval always checks high priority queue first:

```
if (!list_empty(&msg_queue.high_priority)) {
    node = msg_queue.high_priority.next;
} else if (!list_empty(&msg_queue.normal_priority)) {
    node = msg_queue.normal_priority.next;
}
```

2.5 Testing Strategy

Three comprehensive test programs were developed:

1. **test_basic.c**: Validates core functionality (6 tests)
    - Send and receive operations
    - Priority ordering verification
    - Empty queue handling
    - Status reading
2. **test_stress.c**: Tests system under load (5 tests)
    - Rapid send operations (100+ messages)
    - Rapid receive operations

- Priority ordering under stress
- Concurrent operations

3. **test_edge.c**: Validates error handling (9 tests)
   - Invalid commands
   - Boundary conditions
   - Extreme priority values
   - Message truncation
   - Special characters

2.6 Challenges and Solutions

2.6.1 Challenge 1: Thread Synchronization

Problem: Initial implementation had race conditions when multiple processes accessed the queue simultaneously.

Solution: Implemented spinlocks with interrupt-safe operations (`spin_lock_irqsave`) to ensure atomic queue operations. All critical sections are now properly protected.

2.6.2 Challenge 2: User-Kernel Communication

Problem: Parsing user commands from `/proc` interface required careful buffer handling to prevent kernel crashes.

Solution: Implemented robust input validation using `sscanf()` with format checking and buffer size limits. Added comprehensive error handling for malformed input.

2.6.3 Challenge 3: Memory Management

Problem: Memory leaks occurred when module was unloaded with pending messages in queues.

Solution: Implemented cleanup function in `module_exit()` that iterates through both queues and properly frees all allocated messages:

```
static void cleanup_queue(struct list_head *queue) {
    struct demo_msg *msg, *tmp;
    list_for_each_entry_safe(msg, tmp, queue, list) {
        list_del(&msg->list);
        kfree(msg);
        msg_count++;
    }
}
```

2.6.4 Challenge 4: Priority Queue Design

Problem: Deciding between a single sorted list versus dual separate queues for performance.

Solution: Chose dual queue approach for O(1) insertion and O(1) retrieval of highest priority message, rather than O(n) insertion in a sorted list. This provides better performance under load.

2.6.5 Challenge 5: Testing Methodology

Problem: Ensuring comprehensive testing coverage for a kernel module.
Solution: Developed three-tier testing strategy:

- Basic functionality tests for correctness
- Stress tests for performance and stability
- Edge case tests for robustness

This approach caught several boundary condition bugs during development.

3 Conclusion

The implementation successfully demonstrates a working Advanced IPC mechanism with priority-based message queuing. The module is stable, thread-safe, and provides practical experience with kernel programming concepts including synchronization, memory management, and user-kernel communication.

All test programs pass successfully, confirming correct implementation of the required functionality. The modular design allows for easy extension to support additional features such as message priorities, queue size limits, or blocking operations.

3.1 Learning Outcomes

This assignment provided hands-on experience with:

- Linux kernel module development
- Kernel synchronization primitives (spinlocks)
- Dynamic memory allocation in kernel space
- /proc filesystem interface design
- Comprehensive testing strategies for kernel code

3.2 Repository Information

Complete source code, documentation, and test programs are available at:
`https://github.com/wizz-ctrl/Ipc-priority-message-queue-`