Operating Systems Project: Topic 3

Enhanced IPC Mechanism: Design & Implementation

Liangsen Wang

224040364@link.cuhk.edu.cn

2026.1.15

Outline

Outline

The Mission: Topic 3 Requirements

Objective: Build a custom Inter-Process Communication (IPC) mechanism in the Linux Kernel.

Requirement 1: Core Implementation (The "Builder")

- **Task:** Create a **Character Device Driver** (e.g., `/dev/myipc`).
- **Logic:** It must function like a standard Pipe (FIFO Ring Buffer).
- **Constraint:** It must support **Blocking I/O** (Readers sleep when empty; Writers sleep when full).
- **Basis:** Modify or mimic `fs/pipe.c`.

Requirement 2: Benchmarking (The "Tester")

- **Latency:** Measure Round-Trip Time (RTT) for 1-byte messages.
- **Bandwidth:** Measure Throughput (MB/s) for bulk transfers.
- **Comparison: Your Driver** vs. **Linux Pipe** vs. **Shared Memory**.

Advanced Options

Option A: Zero-Copy (`mmap`)

- Implement the `.mmap` file operation.
- Map kernel pages directly to user space.
- **Benefit:** Eliminates `copy_to_user` overhead.

Option B: Lock-Free Algorithms

- Implement a Single-Producer Single-Consumer (SPSC) ring buffer.
- Use memory barriers (`smp_wmb`) instead of Mutex locks.

Outline

Theory 1: The OS Paradox (Isolation vs. Cooperation)

The Wall (Isolation)

- Virtual Memory gives every process its own address space.
- Process A (Address 0x1000) $\neq$ Process B (Address 0x1000).
- **Why?** Stability and Security. If Chrome crashes, your OS survives.

The Gate (IPC)

- Processes often need to collaborate (e.g., 'grep | wc', Server-Client).
- **IPC** is the controlled mechanism to punch a hole in this wall.

Your project is to build a new, efficient gate.

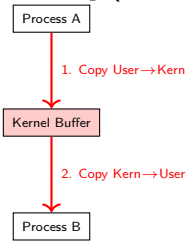
Theory 2: The Linux IPC Landscape

Linux provides a "Zoo" of IPC tools. Which one fits where?

1. **Data Transfer**
   - **Pipes / FIFOs:** Unidirectional byte streams. Simple, thread-safe. (Your Model)
   - **Message Queues:** Linked lists of messages. Preserves boundaries.
   - **Shared Memory:** Zero-copy access to RAM. Fastest, but requires locking.
2. **Communication**
   - **Unix Sockets:** Bi-directional. Can pass File Descriptors.
   - **Signals:** Asynchronous notifications (e.g., SIGINT). No data payload.

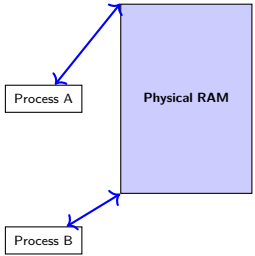
Theory 3: The Physics of Data Movement

1. Message Passing (The Fax Machine)



- **Cost:** 2 Copies per message.
- **Pros:** Safe, implicit synchronization.

2. Shared Memory (The Whiteboard)

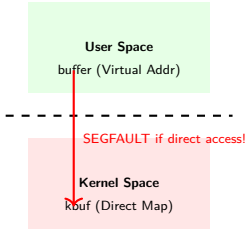


- **Cost:** 0 Copies (after setup).
- **Cons:** Requires explicit locking.

Outline

Kernel Mechanics 1: The User-Kernel Boundary

Critical Concept: You are writing code that lives in Kernel Space.



Why?

- User pointers are virtual addresses.
- They might be swapped out (not in RAM).
- Malicious users might pass invalid pointers.

Solution: MUST use `copy_from_user()` and `copy_to_user()`.

Kernel Mechanics 2: Concurrency & Synchronization

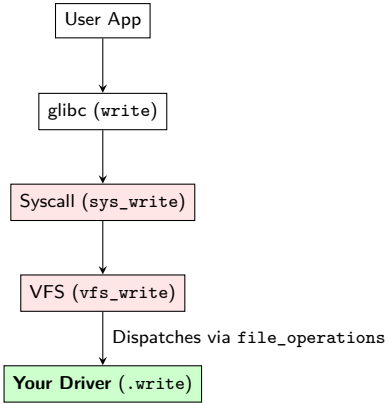
Your driver will be accessed by multiple processes simultaneously.

Lock Type	Behavior	When to use?
Spinlock	Busy-Loops (Spins)	Interrupt Contexts. Tiny critical sections. **Cannot Sleep.**
Mutex	Sleeps	Process Context. Large critical sections. **Can Sleep** (e.g., during `copy_from_user`).

Rule: Since `copy_from_user` might fault (sleep), you **MUST** use a Mutex.

Kernel Mechanics 3: The Virtual File System (VFS)

How does a user calling `write()` end up in your code?



Kernel Mechanics 4: The Lifecycle of a Blocking Read

What happens when you call `read()` on an empty pipe?

1. **Check:** Driver sees buffer is empty.
2. **Prepare:** Set state to `TASK_INTERRUPTIBLE`.
3. **Enqueue:** Add current task to `wait_queue`.
4. **Yield:** Call `schedule()`. CPU switches to another process.
5. **Sleep:** Process is paused. (Takes 0% CPU).
6. **Wake:** Writer calls `wake_up()`. Task moves to Runqueue.

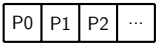
Outline

Reference 1: Core Data Structures

A Linux Pipe is not a simple byte array. It is a **Ring of Pages**.

1. `pipe_inode_info` (The Manager)

- `mutex`: Serializes access.
- `bufs`: Array of `pipe_buffer`.
- `head`: Producer index (ever-increasing).
- `tail`: Consumer index (ever-increasing).



2. `pipe_buffer` (The Container)

- `page`: Pointer to physical RAM page.
- `offset`: Where data starts in this page.
- `len`: How much valid data is in this page.
- **Why Pages?** Enables Zero-Copy Splice!

Reference 2: Efficient Ring Buffer Math

Linux uses a clever trick to avoid expensive Modulo (%) operations.

The "Masking" Logic

Instead of wrapping `head` back to 0, Linux lets `head` grow indefinitely (until integer overflow).

- **Ring Size:** Must be a power of 2 (e.g., 16, 32).
- **Index Calculation:** `index = head & (size - 1)`.
- **Full Check:** `head - tail >= size`.
- **Empty Check:** `head == tail`.

Project Tip: You can use the simple modulo operator (%) for your project to keep it simple, but know that the kernel uses bitwise AND for speed.

Reference 3: The Write Algorithm (Optimization)

When `pipe_write` receives data, it doesn't always allocate a new page.

```
// Pseudocode logic
mutex_lock(&pipe->mutex);

// 1. Check current head page
buf = &pipe->bufs[head & mask];
if (buf has space) {
    // Optimization: Append to existing page!
    copy_to_page(buf->page, ...);
    buf->len += bytes;
    goto out;
}

// 2. If no space, allocate NEW page
page = alloc_page(GFP_KERNEL);
// ... attach page to ring ...
pipe->head++;

wake_up_interruptible(...);
mutex_unlock(...);
```

Key Concept: Merging

- If you write "H", then "E", then "LL", Linux puts them in the **same** 4KB page.
- This reduces memory fragmentation and allocation overhead.

Reference 4: The Read Algorithm

Reading is about consuming data and releasing pages back to the OS.

1. **Lock & Wait:** If `head == tail`, sleep via `pipe_wait()`.
2. **Map:** Get the `pipe_buffer` at `tail`.
3. **Copy:** `copy_page_to_iter()` moves data to User Space.
4. **Update:** Increase `buf->offset`, Decrease `buf->len`.
5. **Release:**
   - If `buf->len == 0` (Page fully consumed):
   - `pipe_buf_release(buf)` (Free the physical page).
   - `pipe->tail++` (Move read pointer).
6. **Wake Writer:** Signal that space is free.

Outline

Step 1: Define Your Device Structure

You need a data structure to maintain state.

```
struct my_ipc_dev {
    char *data;             // The Ring Buffer (use vmalloc)
    int size;               // Buffer Size (e.g., 64KB)
    int head;               // Write Index
    int tail;               // Read Index

    struct mutex lock;      // For thread safety
    wait_queue_head_t wq;   // For blocking I/O
};

```

Ring Buffer Math:

- **Index:** Always use `% size`. e.g., `pos = head % size`.
- **Empty:** `head == tail`
- **Full:** `(head + 1) % size == tail` (Keep one slot open).

Step 2: Implementing Blocking I/O (Crucial)

This pattern allows processes to sleep efficiently.

The Read Function:

```
mutex_lock(&dev->lock);

while (is_empty(dev)) {
    mutex_unlock(&dev->lock); // Release!

    // Sleep until not empty
    if (wait_event_interruptible(
          dev->wq, !is_empty(dev)))
          return -ERESTARTSYS;

    mutex_lock(&dev->lock); // Re-acquire!
}

// ... Copy data to user ...
```

The Write Function:

```
mutex_lock(&dev->lock);

// ... Copy data from user ...
// ... Update head index ...

// Wake up the sleepers!
wake_up_interruptible(&dev->wq);

mutex_unlock(&dev->lock);
```

Step 3: Registration

Use the Character Device API to expose your driver to user space.

```
static struct file_operations fops = {
    .owner = THIS_MODULE,
    .read = my_read,
    .write = my_write,
    .open = my_open,
    .release = my_release,
};

// In module_init:
// 1. alloc_chrdev_region()
// 2. cdev_init(&my_cdev, &fops)
// 3. cdev_add(&my_cdev, ...)

```