Topic 3 Final Report

Enhanced IPC Mechanism in the Kernel

Team 5

Chen Jiayi Student ID: 225040157
225040157@link.cuhk.edu.cn

Weiqin Pan Student ID: 225040150
225040150@link.cuhk.edu.cn

May 2, 2026

Contents

List of Tables

List of Figures

Abstract

The standard Linux pipe is a mature byte-stream IPC mechanism, but it offers no built-in support for routing a single kernel object to multiple isolated user groups, and its data-path performance still leaves room for improvement. This project designs and implements an enhanced inter-process communication mechanism, packaged as a Linux character device driver `/dev/hello`, that combines two contributions. First, a single physical device node is transparently turned into multiple per-cgroup logical channels by routing every `open()` call through a red–black tree keyed on the caller's cgroup identifier, mirroring the isolation discipline that container runtimes such as Docker apply to shared kernel resources. Second, a coordinated set of micro-optimisations on the data path—a page-based ring with page recycling, a small-write merge, exclusive wait queues with chained wake-up, and a 64-bit atomic snapshot for a lock-free poll fast path—together reduce the per-operation cost of the read/write/wakeup cycle. Evaluated with the `ipc-bench` framework on Linux 6.8/AArch64, the driver delivers approximately a 25 % reduction in 1-byte round-trip latency and a 28 % increase in 1-KB sustained throughput compared to the standard Linux pipe, while shared memory remains an order of magnitude faster, as expected, because it bypasses the kernel data path entirely.

Keywords: Linux kernel module; character device driver; inter-process communication; pipe; cgroup; container isolation; page-based ring buffer; lock-free synchronisation.

1 Introduction

1.1 Background

In modern operating systems, processes are isolated from one another by virtual memory: each process owns a private address space, which guarantees stability and security but also prevents direct data exchange. To enable cooperation between isolated processes, the kernel must provide controlled Inter-Process Communication (IPC) primitives. Linux offers a rich set of such primitives—pipes, FIFOs, message queues, shared memory, sockets, and signals—each striking a different balance between simplicity, performance, and semantics.

Among them, the pipe is the most classic choice for one-way streaming communication, and its in-kernel implementation (fs/pipe.c) is built around a ring of physical pages with blocking semantics. Linux exposes such kernel objects to user space through the Virtual File System (VFS) under the philosophy that "everything is a file." A device driver only needs to register a file_operations table, and the kernel will dispatch user open/read/write/poll calls to it. This makes a character device driver the most natural vehicle for prototyping a new IPC mechanism.

1.2 Problem Statement

Although the standard Linux pipe is mature and widely used, it presents two limitations when one tries to use it as a general-purpose, system-level IPC service. First, a pipe is anonymous and bound to a parent–child file-descriptor inheritance chain; it cannot be opened by name from arbitrary processes, and a globally accessible character device that simply wraps a single shared buffer would offer no isolation between unrelated process groups, breaking the "private channel" semantics that real applications—especially containerized workloads—depend on. Second, the standard pipe still pays measurable per-operation costs: page allocation on every write, non-exclusive wakeups that trigger the thundering-herd problem under contention, and lock-protected state checks on the fast path.

This project therefore designs and implements an enhanced IPC mechanism, packaged as a Linux character device driver /dev/hello, that simultaneously addresses both limitations. The driver (i) preserves a faithful FIFO pipe semantics with blocking I/O, mimicking fs/pipe.c; (ii) introduces per-cgroup logical-device isolation—a single physical device node transparently

routes each open() to an independent buffer instance based on the caller's cgroup ID, conceptually mirroring how Docker virtualizes shared kernel resources; and (iii) applies several optimizations—page recycling, small-write merging, an atomic 64-bit head_tail snapshot, and exclusive-wait wake-ups—to outperform the standard pipe on both latency and throughput, while also supporting poll for I/O multiplexing.

1.3 Structure of This Report

The rest of this report walks through the project from background to evaluation. We first review the necessary preliminaries on Linux pipes, the character device framework, and cgroup-based isolation, before presenting the overall system architecture and the core data structures that underpin our design. We then describe the implementation of the main file operations—open, release, read, write, and poll—together with the synchronization mechanisms that keep the driver both correct and efficient. After that, we highlight the key optimizations introduced in the design and explain how each contributes to performance, fairness, or scalability. Finally, we report a quantitative evaluation against the standard Linux pipe and shared memory using the ipc-bench framework, discuss the current limitations, and conclude with directions for future work.

2 Key Concepts

This section introduces the three building blocks that the rest of the report relies on: the character-device interface that exposes our driver to user space, the pipe model that defines its semantics, and the cgroup primitive that enables per-group logical isolation.

2.1 VFS and Character Device Drivers

Linux follows the design philosophy that "everything is a file": any kernel object can be exposed to user space through the unified `open/read/write/poll` interface. The Virtual File System (VFS) layer dispatches each system call to the appropriate backend through a file-type-specific operation table. For a character device, this table is `struct file_operations`, and the kernel invokes its callbacks whenever a user process operates on the corresponding device node under `/dev`. Registering a driver therefore reduces to allocating a device number with `alloc_chrdev_region()`, binding a `cdev` to a custom `file_operations` table via `cdev_init()`/`cdev_add()`, and creating the user-visible node with `device_create()`. Listing 1 shows the operation table used in our driver.

```
static const struct file_operations hello_world_fops = {
    .owner   = THIS_MODULE,
    .open    = hello_world_open,
    .release = hello_world_release,
    .read    = hello_world_read,
    .write   = hello_world_write,
    .poll    = hello_world_poll,
};
```

Listing 1: The `file_operations` table dispatched by VFS.

2.2 Pipe Model: FIFO Ring Buffer with Blocking I/O

A pipe is a producer–consumer FIFO byte stream backed by a fixed-capacity *ring buffer*. Two monotonically increasing indices, *head* and *tail*, denote the next slot to be written and the next slot to be read, respectively. Their relation determines the buffer state: *empty* when `head == tail`, and *full* when `head - tail >= size`. Because reads and writes happen in different processes, the driver must enforce two additional properties: *atomicity* of each operation, and *blocking*

when the buffer cannot make progress (empty for a reader, full for a writer). Implementing these properties in kernel space brings two constraints that shape every subsequent design decision. First, user pointers cannot be dereferenced directly—they are virtual addresses that may not be resident in RAM and may even be malicious; data must therefore cross the user–kernel boundary through `copy_to_user()` and `copy_from_user()`, both of which may page-fault and *sleep*. Second, since the critical section can sleep, it must be protected by a `mutex` rather than a spinlock; blocking is then realised through wait queues and `schedule()`, which yield the CPU until a peer issues a `wake_up_*()` call. Table 1 summarises this choice.

Table 1: Why our driver uses a mutex rather than a spinlock.

Primitive	**Behaviour when contended**	**Sleep allowed inside**
Spinlock	Busy-loop on the CPU	No
Mutex	Sleep, scheduled out	Yes

2.3 cgroup v2 as a Logical Isolation Primitive

A character device node is globally unique: every process that opens `/dev/hello` sees the same kernel object. To turn this single *physical* device into multiple *logical* channels—one per group of cooperating processes—we need a stable, kernel-visible identifier for "which group does this caller belong to". *cgroup v2* is exactly that: it is the canonical resource-grouping mechanism in modern Linux, used by `systemd`, Docker, and Kubernetes to isolate containers, and every task belongs to a unique cgroup whose 64-bit identifier can be obtained inside the kernel as shown in Listing 2. By using this identifier as a routing key in our driver, processes that share a cgroup transparently share a pipe instance, while processes in different cgroups are backed by independent buffers—even though they all open the same device file. This is conceptually the same trick Docker uses to virtualise shared kernel resources, and it forms the foundation of the multi-instance design detailed in the next chapter.

```
1 static u64 get_current_cgroup_id(void)
2 {
3     struct cgroup *cgrp = task_dfl_cgroup(current);
4     return cgroup_id(cgrp);
5 }
```

Listing 2: Obtaining the cgroup ID of the current task.

3 System Design and Implementation

This chapter describes how the design of the enhanced IPC driver is realised in code. We first present the overall architecture and the three core data structures, and then walk through the implementation of each `file_operations` callback in the order it is invoked over a typical lifecycle: module load, `open`, `read`/`write`, `release`, and `poll`. A dedicated subsection consolidates the concurrency-control strategy that crosscuts every callback.

3.1 Architecture and Core Data Structures

The driver is organised in two layers. The *character-device layer* handles VFS dispatch and exposes `/dev/hello` to user space. The *pipe-data layer* implements the actual FIFO logic on top of a ring of physical pages and is replicated per cgroup. Three structures are central to the design. `struct data_node` (Listing 3) represents one *logical pipe instance*. Its `head`/`tail` indices are packaged inside a 64-bit `union` so they can be read atomically as a single word; `buffers` is the

ring of page slots; `rq` and `wq` are the per-instance wait queues for blocked readers and writers; `readers`/`writers` count active openers.

```
struct data_node {
    union {
        unsigned long head_tail;          /* atomic snapshot */
        struct { unsigned int head, tail; };
    };
    struct buffer_node *buffers;          /* ring of page slots */
    unsigned int       size;              /* must be power of 2 */
    struct mutex       lock;
    wait_queue_head_t  wq, rq;            /* writers / readers */
    unsigned int       readers, writers;
    unsigned int       cgroup_id;
};
```

Listing 3: Per-instance pipe state.

`struct buffer_node` (Listing 4) is one slot of the ring; each slot owns a 4 KB physical page and tracks a partial range `[offset, offset+len)` of valid bytes, allowing in-place append on the last page (the small-write merge of §3.4).

```
struct buffer_node {
    struct page  *page;
    unsigned int  offset;   /* start of valid data within the page */
    unsigned int  len;      /* length of valid data within the page */
};
```

Listing 4: A single ring-buffer slot.

`struct cgroup_pipe_entry` (Listing 5) is the routing record stored in a global red–black tree keyed by cgroup ID; it maps each cgroup to its private `data_node`, turning a single character-device node into multiple isolated logical pipes.

```
struct cgroup_pipe_entry {
    struct rb_node   node;
    u64              cgroup_id;
    struct data_node *pipe;
};
```

Listing 5: cgroup → pipe-instance map node.

3.2 Driver Skeleton: Module Lifecycle and Character-Device Registration

The module entry point performs the four canonical steps required of any character device: dynamic allocation of a major number, binding a `cdev` to our `file_operations` table, registering it with the kernel, and creating a user-visible node under `/dev`. The unload path mirrors the same sequence in reverse and additionally walks the rbtree to release every per-cgroup instance still alive (Listing 6).

```
static const struct file_operations hello_world_fops = {
    .owner   = THIS_MODULE,
    .open    = hello_world_open,
    .release = hello_world_release,
    .read    = hello_world_read,
    .write   = hello_world_write,
    .poll    = hello_world_poll,
};
static int __init hello_driver_init(void) {
```

```
    alloc_chrdev_region(&hello_devid, 0, 1, "hello");
    cdev_init(&hello_cdev, &hello_world_fops);
    cdev_add(&hello_cdev, hello_devid, 1);
    hello_class = class_create("hello_class");
    device_create(hello_class, NULL, hello_devid, NULL, "hello");
    return 0;
}
```

Listing 6: `file_operations` table and registration sequence.

3.3 Open and Release: Per-cgroup Instance Routing

The `open` callback turns the single physical device into per-cgroup logical channels. It first obtains the caller's cgroup ID, then either locates an existing `data_node` in the rbtree or allocates a new one on first use; the chosen instance is stored in `filp->private_data` so that subsequent `read`/`write`/`poll` calls operate on the correct buffer without re-traversing the tree (Listing 7).

```
cgid = get_current_cgroup_id();         /* task_dfl_cgroup -> cgroup_id
    */
mutex_lock(&cgroup_tree_lock);
entry = find_pipe_by_cgroup(cgid);      /* O(log N) rbtree lookup */
if (!entry) {
    pipe  = kzalloc(sizeof(*pipe), GFP_KERNEL);
    data_node_init(pipe);               /* alloc 16 pages, init wq/rq */
    pipe->cgroup_id = cgid;
    entry = kmalloc(sizeof(*entry), GFP_KERNEL);
    entry->cgroup_id = cgid;
    entry->pipe      = pipe;
    insert_pipe_entry(entry);           /* rb_link_node + rb_insert_color
        */
} else {
    pipe = entry->pipe;
}
mutex_unlock(&cgroup_tree_lock);
filp->private_data = pipe;
```

Listing 7: cgroup-based routing in `open` (core path).

After binding, `open` updates `readers`/`writers` according to the file-mode flags and, mimicking pipe semantics, lets a read-only opener block on `wq` until a writer appears (and vice versa). `release` performs the dual operation: it decrements the counter for the closing role and, when a side reaches zero, wakes the opposite queue so that any waiter learns of the EOF or hang-up. The three open modes lead to slightly different behaviour, summarised in Table 2.

Table 2: Behaviour of `open` under each access mode.

Mode	Counter update	Blocking condition
`O_RDONLY`	`readers++`	wait on `wq` until a writer exists
`O_WRONLY`	`writers++`	wait on `wq` until a reader exists
`O_RDWR`	both counters `++`	none (peer guaranteed within the same fd)

3.4 Read and Write: Page-wise Data Path

Both data-path callbacks operate page-by-page on the ring. Writing first attempts to *merge* a small payload into the unfinished tail of the current head page, avoiding a fresh page allocation

for the common case of short messages (Listing 8). When the merge does not apply, the main loop advances the head: it acquires (or recycles) a page via `anon_pipe_get_page`, maps it with `kmap_local_page`, copies up to one page from user space, and finally publishes the slot by incrementing `pipe->head` (Listing 9). When the ring is full, the writer releases the lock and blocks on `wq`.

```
chars = len & (PAGE_SIZE - 1);
if (chars && !was_empty) {
    struct buffer_node *buf = pipe_buf(pipe, head - 1);
    int offset = buf->offset + buf->len;
    if (offset + chars <= PAGE_SIZE) {
        char *kaddr = kmap_local_page(buf->page);
        copy_from_user(kaddr + offset, user_buf, chars);
        kunmap_local(kaddr);
        buf->len += chars; ret += chars;
        if (ret == len) goto out;
    }
}
```

Listing 8: Small-write merge: append into the last page when it still has room.

```
for (;;) {
    head = pipe->head;
    if (!pipe_full(head, pipe->tail, pipe->size)) {
        page = anon_pipe_get_page(pipe, head);   /* alloc or recycle */
        to_copy = min((size_t)PAGE_SIZE, len - ret);
        char *kaddr = kmap_local_page(page);
        copy_from_user(kaddr, user_buf + ret, to_copy);
        kunmap_local(kaddr);
        pipe->head = head + 1;
        buf_node = pipe_buf(pipe, head);
        buf_node->page = page; buf_node->offset = 0; buf_node->len =
            to_copy;
        ret += to_copy;
        if (ret == len) break;
        continue;
    }
    mutex_unlock(&pipe->lock);
    wait_event_interruptible_exclusive(pipe->wq, pipe_writable(pipe));
    mutex_lock(&pipe->lock);
}
```

Listing 9: Write main loop (one page per iteration).

The read path is symmetric (Listing 10). It examines the slot pointed to by `tail`, maps its page, copies up to `buf->len` bytes into the user buffer, and updates the slot's `offset`/`len`. When a slot is fully consumed, `tail` advances. When the ring is empty and no byte has been copied yet, the reader releases the lock and blocks on `rq`.

```
for (;;) {
    head = pipe->head; tail = pipe->tail;
    if (!pipe_empty(head, tail)) {
        struct buffer_node *buf = pipe_buf(pipe, tail);
        size_t chars = min(buf->len, total_len);
        char *kaddr = kmap_local_page(buf->page);
        copy_to_user(user_buf + ret, kaddr + buf->offset, chars);
        kunmap_local(kaddr);
        ret += chars; buf->offset += chars; buf->len -= chars;
```

```
        if (!buf->len) {                        /* slot fully drained */
            wake_writer |= pipe_full(head, tail, pipe->size);
            pipe->tail = ++tail;
            buf->offset = buf->len = 0;
        }
        total_len -= chars;
        if (!total_len) break;
        if (!pipe_empty(head, tail)) continue;
    }
    if (ret) break;
    mutex_unlock(&pipe->lock);
    if (wait_event_interruptible_exclusive(pipe->rq, pipe_readable(pipe)) < 0)
        return -ERESTARTSYS;
    mutex_lock(&pipe->lock);
}
```

Listing 10: Read main loop (one slot per iteration).

3.5 Concurrency Control

Three coordinated mechanisms protect shared state across all callbacks. A *global mutex* `cgroup_tree_lock` guards the rbtree during `open`/`release`; a *per-instance mutex* `pipe->lock` serialises every read/write critical section, including the sleeping `copy_*_user` call; and *two wait queues* per instance separate blocked readers from blocked writers. The lock-to-state mapping is summarised in Table 3. To avoid the thundering-herd effect, every blocking call

Table 3: Shared state and the primitive that protects it.

Shared state	**Protected by**	**Acquired in**
`cgroup_pipe_tree` (rbtree)	`cgroup_tree_lock` (mutex)	open / release / module exit
`head`, `tail`, `buffers[]`	`pipe->lock` (mutex)	read / write
`readers`, `writers`	`pipe->lock` (mutex)	open / release
`head_tail` (snapshot read)	`READ_ONCE` (lock-free)	poll / wait predicate

uses `wait_event_interruptible_exclusive`, so each `wake_up` releases exactly one waiter; the woken task in turn wakes the next one only when the new state still allows progress, yielding a *chained wake-up* that preserves FIFO fairness. Finally, the wait predicate `pipe_readable()` consults `head` and `tail` as a single atomic 64-bit word via `READ_ONCE`, eliminating the need to take `pipe->lock` on the fast path of `poll` (Listing 11).

```
static inline bool pipe_readable(const struct data_node *pipe) {
    unsigned long ht = READ_ONCE(pipe->head_tail);
    unsigned int head = (unsigned int)(ht >> 32);
    unsigned int tail = (unsigned int)ht;
    return !pipe_empty(head, tail);
}
```

Listing 11: Lock-free state check using the 64-bit `head_tail` union.

3.6 Poll: Supporting I/O Multiplexing

The `poll` callback enables our driver to participate in `select`/`poll`/`epoll` loops, allowing a process to monitor `/dev/hello` alongside other file descriptors without blocking on any single

one. Implementation is deliberately light-weight: it registers the calling task on the relevant wait queues via `poll_wait` (no sleeping happens here—`poll_wait` only *records* the queues so the kernel can re-check later), then computes a readiness mask from the same lock-free 64-bit snapshot used in §3.5 (Listing 12).

```
static __poll_t hello_world_poll(struct file *filp, poll_table *wait) {
    struct data_node *pipe = filp->private_data;
    unsigned long ht; unsigned int head, tail; __poll_t mask = 0;
    if (filp->f_mode & FMODE_READ)  poll_wait(filp, &pipe->rq, wait);
    if (filp->f_mode & FMODE_WRITE) poll_wait(filp, &pipe->wq, wait);
    ht   = READ_ONCE(pipe->head_tail);
    head = (unsigned int)(ht >> 32); tail = (unsigned int)ht;
    if (filp->f_mode & FMODE_READ) {
        if (!pipe_empty(head, tail))              mask |= EPOLLIN |
           EPOLLRDNORM;
        if (!pipe->writers)                       mask |= EPOLLHUP;
    }
    if (filp->f_mode & FMODE_WRITE) {
        if (!pipe_full(head, tail, pipe->size))   mask |= EPOLLOUT |
           EPOLLWRNORM;
        if (!pipe->readers)                       mask |= EPOLLERR;
    }
    return mask;
}
```

Listing 12: The `poll` implementation: lock-free mask computation.

The mapping between events and the returned mask is summarised in Table 4.

Table 4: State transitions and the corresponding poll mask bits.

Trigger	**Wait queue woken**	**Mask bit observed by user**
Writer published a page	`rq`	`EPOLLIN`
Reader drained a page	`wq`	`EPOLLOUT`
Last writer closed the fd	`rq`	`EPOLLHUP`
Last reader closed the fd	`wq`	`EPOLLERR`

4 Optimizations and Advantages

This section consolidates the design choices that distinguish our driver from a textbook FIFO. Each optimization is presented as a self-contained mechanism–problem–benefit triple, building on the implementation already shown in the previous chapter. The discussion ranges from a system-level design choice (cgroup-based logical isolation) to micro-optimizations on the data path (page recycling, write merging, lock-free snapshots, and exclusive wake-ups).

4.1 Per-cgroup Logical Isolation

Mechanism. A single character-device node `/dev/hello` is turned into multiple logical channels by routing every `open()` to a private `data_node` keyed on the caller's cgroup ID, with the mapping stored in a kernel red–black tree. **Problem.** A naive driver that maintains one global ring buffer would expose every byte written by any process to every other reader, breaking the "private channel" semantics that real applications—especially containerised workloads—rely on. Worse, the single global mutex would serialise all callers, eliminating concurrency. **Benefit.** The design grants three properties at once: (i) processes in different cgroups are mutually

invisible, mirroring the isolation that Docker and Kubernetes already establish at the cgroup boundary, with *zero* changes required in user-space code; (ii) per-instance mutexes mean different cgroups never contend on the same lock, so the data path *scales horizontally* with the number of containers; (iii) only a single device node is exposed, so the user-facing API surface stays minimal. The contrast is summarised in Table 5.

Table 5: Global-buffer character device vs. our cgroup-aware design.

Aspect	**Global-buffer driver**	**Our design**
Inter-group isolation	none (data leaks across groups)	full (per-cgroup buffer)
Lock contention	one global mutex for all callers	one mutex per cgroup
Container friendliness	manual coordination required	transparent to user space
Device nodes exposed	one per channel	one for the whole system

4.2 Page-based Ring with Recycling

Mechanism. Each ring slot owns a `struct page` rather than a slice of a contiguous byte array. When a reader fully drains a slot, the page is *not* returned to the kernel; instead, the slot keeps its page pointer, and the next writer that lands on the slot reuses it directly via `anon_pipe_get_page` (cf. `hello_module.h`). **Problem.** Calling `alloc_page`/`__free_page` on every transferred page burdens the buddy allocator, fragments memory, and adds latency to every write that crosses a page boundary. **Benefit.** After an initial warm-up of at most `size` pages (16 in our configuration), the steady-state data path performs *zero* page (de)allocation, regardless of how many bytes are transferred. This is a significant departure from the standard `fs/pipe.c`, which releases each page back to the allocator once consumed, and is one of the contributors to the throughput improvement reported in the evaluation chapter. Because the ring is a true page array, the structure is also natively prepared for the zero-copy `mmap`/`splice` extensions discussed as future work.

4.3 Small-Write Merge

Mechanism. Before entering the main allocation loop, the writer inspects the slot at `head-1` and, if it still has unused trailing space, copies the new payload directly into that page (see the write-merge listing in the Implementation chapter). **Problem.** Workloads that issue many short `write()` calls —typical of shell pipelines, logging, and message-style protocols—would otherwise consume one full 4 KB page per call, wasting both memory and the cost of an extra `alloc_page`. **Benefit.** The optimization compacts a stream of short writes into the fewest possible pages, which reduces both the number of allocator calls and the number of slots the reader must traverse. It is most impactful precisely in the workload our latency benchmark stresses (1-byte messages), contributing to the round-trip-time improvement over the standard pipe reported in the evaluation chapter.

4.4 Lock-free Fast Path via a 64-bit Atomic Snapshot

Mechanism. The two ring indices `head` and `tail` are packed into a single 64-bit `union` field `head_tail`; any code that only needs to *check* the buffer state reads this field via `READ_ONCE` and then unpacks the two 32-bit halves (see the lock-free snapshot listing in the Implementation chapter). **Problem.** A naive predicate that reads `head` and `tail` as two separate words may observe a *torn* state in which one half has been updated and the other has not, leading to spurious wake-ups or, worse, missed events. The standard remedy—taking the per-pipe mutex on every check—would serialise the `poll` fast path that may execute millions of times per second.

Benefit. The fast paths of `poll` and the `wait_event` predicates run completely lock-free yet always observe a consistent state. This is particularly valuable on the AArch64 platform used in our evaluation, whose weak memory model would otherwise demand expensive explicit barriers around any multi-word read.

4.5 Exclusive Wait and Chained Wake-up

Mechanism. Every blocking call uses `wait_event_interruptible_exclusive`, marking the waiter as *exclusive* on the wait queue; each `wake_up_interruptible` therefore releases exactly one waiter, and the woken task itself wakes the next one only when it observes that the new state still allows the next operation to proceed. **Problem.** A traditional `wake_up_all` on a queue with N sleepers triggers N context switches, but only one of them will actually make progress; the remaining $N - 1$ wake up, fail the predicate, and sleep again. This *thundering herd* wastes CPU, thrashes the scheduler, and intensifies lock contention on the protected critical section. **Benefit.** The exclusive-wait pattern reduces context-switch cost from $O(N)$ to $O(1)$ per event, and the chained wake-up preserves FIFO fairness because each new waiter is appended to the tail of the queue and woken in arrival order. Together they guarantee that under contention the driver remains both *efficient* (no wasted wake-ups) and *fair* (no starvation of long-blocked waiters), a property that becomes critical when a single cgroup hosts many cooperating producers and consumers.

5 Testing and Evaluation

This chapter assesses the driver from two complementary angles. We first verify that it functions correctly as a FIFO character device, and then quantify how its latency and throughput compare to two reference IPC mechanisms in the standard Linux toolbox: the kernel pipe and POSIX shared memory. The methodology is anchored in a reproducible setup based on the open-source `ipc-bench` framework.

5.1 Experimental Setup

All experiments were conducted in a virtualised environment whose configuration is summarised in Table 6. We deliberately chose the latest Ubuntu LTS release together with the corresponding 6.8 kernel, both to demonstrate compatibility with current upstream Linux and to surface any portability issues introduced since the original `fs/pipe.c` reference (§5.2). For benchmarking

Table 6: Hardware and software configuration of the test environment.

Component	Configuration
Architecture	AArch64 (virtual machine)
vCPU / RAM	2 vCPUs / 4 GB
Operating system	Ubuntu 24.04 LTS
Kernel	Linux 6.8.0-101-generic
Compiler	GCC 13.3.0

we adopted `goldsborough/ipc-bench`, an open-source suite widely used to measure single-node IPC primitives. We chose it for three reasons. First, it provides a *standardised* ping-pong harness so that every IPC type is exercised under the same protocol. Second, it *automates* the message-passing loop, the timing instrumentation, and the warm-up phase, eliminating common sources of measurement bias. Third, it records timestamps with microsecond resolution, which matches the natural scale of in-kernel data-path operations. The benchmark runs as

two processes connected through the IPC mechanism under test. A *writer* sends a fixed-size payload, the data is copied into the kernel buffer, and a *reader* reads it back; the reader then echoes a reply, completing one *round trip*. Two metrics are reported: the *average round-trip time (RTT)* measured on 1-byte messages, which isolates the per-operation overhead, and the *message rate* (throughput) measured on 1-KB messages, which captures sustained bandwidth. Each test consists of 1000 iterations.

5.2 Implementation Challenges: Porting to Linux 6.8

The driver was originally prototyped against the Linux 5.x API, on which several of the kernel-internal interfaces have since changed. Three concrete issues arose when first compiling against the 6.8 headers; the fixes are summarised in Table 7. None of them required redesigning the data path—all were strictly mechanical adjustments—but they confirm that out-of-tree modules require active maintenance to track upstream API churn.

Table 7: Portability issues encountered on Linux 6.8 and their fixes.

Category	**Symptom**	**Fix**
API change	`class_create()` prototype changed in 6.4 (no longer takes `THIS_MODULE`)	Removed the `THIS_MODULE` argument from the call site.
Type safety	Custom `pipe_write` signature mismatched the new `file_operations` prototype	Added the `const` qualifier to the user-buffer pointer.
Format strings	`printk`'s `%llu` on `u64` cgroup IDs warned on AArch64	Cast the value to `unsigned long long` before printing.

5.3 Functional Verification

Before measuring performance we confirmed that the driver loads cleanly, exposes the expected user interface, and survives basic I/O. Two pieces of evidence were collected:

- The kernel log (`dmesg | tail`) reports `hello_driver_init` immediately followed by the dynamically allocated major number, indicating that `alloc_chrdev_region`, `cdev_add`, and `device_create` all succeeded without error.
- `ls -l /dev/hello` confirms that the device node exists with the expected character-device type and access permissions.

We then ran the basic read/write smoke tests located under `test/src/`: a writer process opens `/dev/hello` and writes a known string, a reader process opens the same node and recovers the bytes in order, and the module unloads via `rmmod` without leaking memory (verified by re-loading and observing fresh allocations). The driver is therefore functionally correct as a FIFO character device.

5.4 Performance Results

Table 8 reports the headline measurements obtained on our driver. The average 1-byte RTT of 431.9 establishes the baseline cost of a round trip through the read/write/wakeup path; the message rate of 2209 1-KB messages per second translates the same behaviour into sustained throughput. Two observations are worth highlighting. *Variance is large.* The standard deviation is comparable in magnitude to the mean, and the worst-case RTT exceeds the minimum by

Table 8: Headline performance of the proposed driver (1000 iterations).

Metric	Value
Average RTT (1-byte messages)	431.897
Minimum RTT (1-byte messages)	153.6
Maximum RTT (1-byte messages)	4847.104
Standard deviation	418.9
Sustained message rate (1 KB messages)	2209 msg/s

more than an order of magnitude. This is a known artefact of running benchmarks inside a virtual machine: scheduler preemption, host-side noise, and hypervisor exit overheads all inject millisecond-scale outliers that dominate the tail. The minimum RTT (153.6) therefore better characterises the driver's intrinsic cost—the unavoidable expense of two system calls, two `copy_*_user` crossings, and a wake-up. *The bottleneck is the syscall path, not memory.* For 1-byte messages, the time spent moving the single byte is negligible compared with entering and leaving the kernel; the fact that the driver still sustains over 2000 round trips per second demonstrates that its locking and notification machinery handles context-switch-bound workloads reliably.

5.5 Comparative Analysis

To put the absolute numbers in context we ran the same `ipc-bench` harness against the standard Linux pipe and POSIX shared memory. Table 9 reports the comparison.

Table 9: Comparison against standard Linux IPC mechanisms.

IPC mechanism	1-byte RTT ()	1-KB throughput (msg/s)
Our driver	431.897	2209
Linux pipe	577.768	1730
Shared memory	30.266	33039

Versus the Linux pipe. Our driver is the more interesting comparison, because it offers the same byte-stream semantics and traverses the same VFS path. We observe a reduction of approximately 146 in average RTT (a 25 % improvement) and a ~28 % increase in sustained message rate. The two figures are consistent with the optimizations introduced in the design: page recycling avoids the per-write `alloc_page` that the standard pipe still pays, the small-write merge collapses successive short writes into the same page, and exclusive wait combined with the lock-free poll predicate trims the wake-up cost. None of these changes alter the API surface; the same `open`/`read`/`write` calls drive both implementations.

Versus shared memory. Shared memory remains roughly an order of magnitude faster on both metrics, which is the expected outcome: it bypasses the kernel entirely on the data path and eliminates the two `copy_*_user` crossings that any character-device-based pipe must perform. The gap therefore quantifies the cost of staying within the "everything is a file" abstraction; closing it would require a zero-copy extension, discussed as future work.

Take-away. For workloads that need byte-stream semantics, process isolation across cgroups, and the convenience of file-descriptor multiplexing, our driver delivers measurably better latency and throughput than the stock Linux pipe while remaining transparent to user-space code. For

workloads that can tolerate manual synchronisation and shared addressing, shared memory remains the absolute speed champion.

5.6 Comparison with a Concurrent Implementation

For additional perspective we briefly compare our driver to another contemporaneous implementation completed by a different team for the same assignment. That implementation, hereafter denoted *MyIPC*, exposes two character-device nodes (`/dev/myipc0` and `/dev/myipc1`) backed by independent contiguous byte buffers, uses a per-device mutex together with default wait queues, and does not implement `poll`. Table 10 summarises the salient design and outcome differences. The MyIPC report notes that its driver achieves competitive 1-byte latency but

Table 10: Design and outcome differences between MyIPC and our driver.

Dimension	**MyIPC (baseline)**	**Our driver**
Device nodes	2 (one per direction)	1, with cgroup routing
Multi-tenant isolation	none	per-cgroup buffer
Buffer storage	contiguous byte array	16-page recycling ring
`head`/`tail` read	mutex-protected	lock-free 64-bit snapshot
Wait-queue policy	default broadcast	exclusive + chained wake-up
Write-side optimisation	none	small-write merge
`poll` support	no	yes
1-KB throughput vs. Linux pipe	slower	~28 % faster

is bandwidth-limited by the small static buffer and frequent allocations. Our results in the comparative analysis above (cf. the evaluation chapter) suggest that the combination of page recycling, write merging, and a lock-free fast path addresses precisely those bottlenecks, allowing the same VFS-based character-device approach to outperform the standard Linux pipe on bulk throughput as well.

6 Discussion, Limitations, and Future Work

The evaluation in the previous chapter demonstrates that the proposed driver is a viable middle ground between the standard Linux pipe and POSIX shared memory. This chapter critically examines the design choices that remain unresolved, and outlines concrete extensions that would address each of them.

6.1 Limitations

Lazy reclamation of pipe instances. A new `data_node` is created on the first `open()` from a previously unseen cgroup, but it is not destroyed when the corresponding cgroup itself disappears. The `release` callback only decrements the reader/writer counters, and the rbtree is fully walked and freed only at module unload. In a long-running system that creates and destroys many short-lived containers, this manifests as a slow but steady leak of `struct data_node` objects and their associated 16-page rings. The fix requires hooking into the cgroup-destruction path, which is outside the scope of the current prototype.

Two unavoidable user–kernel copies. Because the driver lives behind the standard `read`/`write` interface, every byte crosses the user–kernel boundary twice: once into the kernel ring on `write`, and once back out on `read`. This explains the order-of-magnitude gap to shared memory observed in the evaluation. No amount of in-kernel optimisation can close that gap; only a different data-path API—e.g. `mmap`—can.

Single channel per cgroup. All processes that share a cgroup share one pipe instance. This matches the "container = one private channel" model that motivated the design, but it is not appropriate when multiple unrelated services within the same cgroup need independent streams. A finer-grained keying scheme (e.g. a tuple of cgroup ID and user-supplied channel name passed via `ioctl`) would generalise the design without abandoning the per-cgroup default.

Measurement noise. All performance numbers were collected inside a virtual machine on a shared host. The standard deviation observed in the latency experiment (∼420 µs on a mean of ∼432 µs) confirms that hypervisor scheduling jitter contributes a large fraction of the tail. Confirming the steady-state throughput numbers on bare-metal hardware would strengthen the comparison.

Minor type-safety issues. The `cgroup_id` field stored inside `data_node` is currently declared `unsigned int`, whereas the kernel API `cgroup_id()` returns `u64`. The truncation does not cause incorrect behaviour because the rbtree entry keeps the full 64-bit value, but the inconsistency should be cleaned up for robustness. A similar minor issue is the field-size argument passed to `kmalloc_array` when allocating the buffer ring, which is larger than strictly needed and over-allocates a small constant amount of memory per pipe instance.

6.2 Future Work

Zero-copy data path via `mmap`. Because each ring slot already owns a `struct page`, exposing the ring through a custom `.mmap` handler would allow user processes to read and write directly into kernel memory, eliminating both `copy_*_user` crossings. We expect this single change to bring our throughput close to shared memory while preserving FIFO semantics and `poll` support.

Single-producer single-consumer lock-free ring. The current mutex-based design is correct under arbitrary concurrency but pays a fixed cost for every read/write. When the workload is restricted to one producer and one consumer—a common pattern in pipelines—the mutex can be replaced by paired memory barriers (`smp_wmb`, `smp_rmb`), trimming both the lock-acquisition cost and the cache-line ping-pong on the lock word.

Cgroup-lifecycle integration. Registering a callback on cgroup destruction would let the driver eagerly release the corresponding `data_node` and its pages, eliminating the lazy-reclamation leak discussed above. The kernel exposes the necessary hooks via the cgroup v2 subsystem interface.

Splice and `sendfile` integration. With page-based storage already in place, supporting `splice`/`vmsplice` would let the driver participate in zero-copy pipelines that route data between sockets, files, and pipes without ever materialising it in user space.

Per-CPU page caches. Allocation contention on the buddy allocator can be further reduced by caching freed pages in per-CPU lists, mirroring the approach taken by the kernel's slab allocator. This becomes relevant when the small-write merge is bypassed by long, page-aligned writes that genuinely consume new pages each iteration.

7 Conclusion

This project set out to design and implement an enhanced inter-process communication mechanism inside the Linux kernel, packaged as a character device driver and offered through the same VFS

interface that user programs already use for pipes and files. Beyond satisfying the mandatory requirements—FIFO semantics, blocking I/O, and a benchmark comparison against existing primitives—the driver introduces a per-cgroup logical isolation layer that turns a single physical device node into an arbitrary number of mutually invisible logical channels, conceptually mirroring how Docker virtualises shared kernel resources for containers. A series of micro-optimisations on the data path—page-based ring with recycling, small-write merging, exclusive wait with chained wake-up, and a 64-bit atomic snapshot for the lock-free fast path—together let the driver outperform the stock Linux pipe by roughly twenty-five percent on 1-byte round-trip latency and by twenty-eight percent on 1-KB sustained throughput, while shared memory remains an order of magnitude faster, as expected, because it bypasses the kernel altogether. The remaining gap to shared memory, the lazy reclamation of per-cgroup instances, and the inherent variance of virtualised measurements together delineate the natural avenues for future work: a zero-copy `mmap`-based data path, integration with the cgroup-destruction hook, and a single-producer single-consumer lock-free variant for specialised pipelines. Taken together, the prototype demonstrates that a modest amount of architectural care—most notably a deliberate choice of isolation primitive—can turn a textbook character device into a container-aware IPC mechanism that is both faster than the standard pipe and easier to reason about than shared memory.

References

[1] P. Goldsborough, "IPC-Bench: Benchmarks for various inter-process communication mechanisms on Linux and OS X," https://github.com/goldsborough/ipc-bench, accessed 2025.

[2] Linux Kernel Project, "`fs/pipe.c`: Linux pipe implementation (kernel 6.8 source tree)," https://elixir.bootlin.com/linux/v6.8/source/fs/pipe.c, accessed 2025.

[3] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, 3rd ed. Sebastopol, CA: O'Reilly Media, 2005.

[4] R. Love, *Linux Kernel Development*, 3rd ed. Upper Saddle River, NJ: Addison-Wesley, 2010.

[5] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. Sebastopol, CA: O'Reilly Media, 2005.

[6] Linux Kernel Project, "Control Group v2 (cgroup-v2) documentation," https://docs.kernel.org/admin-guide/cgroup-v2.html, accessed 2025.

[7] Linux Kernel Project, "Overview of the Linux Virtual File System," https://docs.kernel.org/filesystems/vfs.html, accessed 2025.

[8] Linux man-pages project, "pipe(7) – overview of pipes and FIFOs," https://man7.org/linux/man-pages/man7/pipe.7.html, accessed 2025.

[9] Linux man-pages project, "shm_overview(7) – overview of POSIX shared memory," https://man7.org/linux/man-pages/man7/shm_overview.7.html, accessed 2025.

[10] Linux man-pages project, "epoll(7) – I/O event notification facility," https://man7.org/linux/man-pages/man7/epoll.7.html, accessed 2025.

[11] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, art. no. 2, Mar. 2014.