Design and Evaluation of Custom Inter-Process Communication Mechanisms: A Comparative Study of Kernel Driver and Shared Memory Approaches

Team: Lan Rui, Chen Yanbin
School of Data Science, The Chinese University of Hong Kong, Shenzhen

April, 2026

Abstract

Inter-process communication (IPC) is a cornerstone of modern operating systems, enabling isolated processes to exchange data and coordinate activities. This report presents and compares three custom IPC implementations developed during the project: a naive kernel character device driver with a FIFO ring buffer (MyIPC), a user-space shared memory mechanism synchronized by POSIX semaphores, and an enhanced kernel-level IPC driver proposed by a peer group that leverages page-based buffer management, cgroup-aware isolation, and advanced synchronization. We detail the architectural designs, data structures, blocking semantics, and optimization techniques of each approach. A comprehensive benchmark suite measures round-trip latency for 1-byte messages and sustained throughput for payloads up to 1024 bytes. Experimental results demonstrate that while the simple kernel driver achieves competitive latency for tiny messages, it suffers from low bandwidth due to frequent system calls and a small ring buffer. The shared memory design excels in throughput for medium and large messages by eliminating kernel-user copies, but requires explicit synchronization. The opponent's enhanced driver mitigates kernel overhead with page-level buffering, exclusive wake-up, and poll support, yielding improved latency over traditional pipes and higher concurrency, yet still cannot fully bridge the gap to zero-copy shared memory. Quantitative comparisons are provided in tabular form, and trade-offs among ease of implementation, safety, and performance are thoroughly discussed.

1 Introduction

Inter-process communication (IPC) allows multiple processes within an operating system to share data and synchronize their actions. Linux offers a variety of IPC primitives such as pipes, FIFOs, message queues, shared memory, and sockets. Each mechanism strikes a different balance between latency, throughput, complexity, and resource usage. In particular, pipes are simple and ubiquitous but force data to traverse kernel buffers, introducing context switches and memory copies. Shared memory avoids these copies entirely at the cost of requiring external synchronization, typically via semaphores or mutexes.

In this project, we set out to design, implement, and benchmark novel IPC mechanisms with the following goals: (i) construct a kernel-resident IPC using a Linux character device driver and a FIFO ring buffer; (ii) build a lightweight user-space IPC based on shared memory and POSIX semaphores; (iii) critically evaluate these designs against a highly optimized kernel IPC developed by a competing team. The opponent's mechanism extends the device-driver concept with per-cgroup pipe instances, page-granularity buffering, exclusive wait-queue wake-ups, and `poll` multiplexing. By placing all three approaches under identical workloads, we

aim to reveal how synchronization overhead, copy semantics, buffer topology, and kernel-user transitions influence IPC performance.

The remainder of the report is organized as follows. Section 2 describes the background and related work. Section 3 details the design of our two IPC mechanisms. Section 4 introduces the opponent's enhanced driver. Section 5 presents the experimental setup. Section 6 compares the results and discusses the implications. Section 7 concludes and outlines future work.

2 Background

2.1 Pipes and Shared Memory in Linux

A Linux pipe is a unidirectional byte stream provided by the kernel. Data written to the pipe are stored in a kernel buffer (typically one page, 4 KiB) and are read in FIFO order. The `read` and `write` system calls involve copying data between user space and the kernel buffer, and processes may block when the buffer is empty or full. Pipes are simple to use and automatically synchronize producer and consumer, but they incur at least two copies per byte transferred and two context switches per message.

Shared memory, on the other hand, maps the same physical pages into the address spaces of multiple processes. Once the mapping is established, data can be exchanged with simple load/store instructions, completely bypassing the kernel. However, shared memory does not provide any built-in notification or mutual exclusion. Developers must pair it with synchronization primitives such as POSIX semaphores, which themselves require system calls for wait and post operations.

2.2 Character Device Drivers

The Linux Virtual File System (VFS) allows custom hardware or software resources to be exposed as files. A character device driver registers a set of `file_operations` callbacks—`open`, `release`, `read`, `write`, `poll`—that the kernel invokes in response to user-space system calls. By implementing a driver that creates entries under `/dev`, we can craft IPC semantics that appear to processes as ordinary file I/O while executing entirely within the kernel, with full control over buffering and blocking.

3 Design of Our IPC Mechanisms

3.1 Kernel Character Device IPC (MyIPC)

Our first IPC mechanism, termed MyIPC, is implemented as a loadable kernel module that creates two device nodes, `/dev/myipc0` and `/dev/myipc1`, to support bidirectional communication. The core of the driver is a static FIFO ring buffer whose structure is defined as follows:

```
struct myipc_dev {
    char *buffer;
    int head;   // write position
    int tail;   // read position
    int size;
    struct mutex lock;
    wait_queue_head_t read_queue;
    wait_queue_head_t write_queue;
};
```

The buffer is allocated as a contiguous array of `size` bytes during module initialization. The `head` index indicates where the next byte will be written, and the `tail` index indicates the next

byte to be read. The buffer is empty when `head == tail` and full when advancing `head` by one would make it equal to `tail`. Access to the indices and buffer contents is guarded by a single `mutex`. Two wait queues enable blocking semantics: a reader that encounters an empty buffer sleeps on `read_queue`, and a writer that encounters a full buffer sleeps on `write_queue`.

The `write` operation follows a classic producer-consumer pattern:

1. Acquire the mutex.
2. While the buffer is full, release the mutex, sleep on `write_queue` until awakened, and reacquire the mutex.
3. Determine how many bytes can be written contiguously (wrapping at the end of the buffer).
4. Copy data from user space using `copy_from_user`.
5. Advance the `head` pointer.
6. Wake up any readers waiting on `read_queue`.
7. Release the mutex.

The `read` operation is symmetric, using `copy_to_user` and waking writers. This design is straightforward and guarantees exactly-once delivery of every byte, but it serializes all operations under a single mutex and performs one kernel-to-user copy per system call.

3.2 Shared Memory and Semaphore IPC

The second mechanism operates entirely in user space. A contiguous shared memory region is created with `shm_open` and `ftruncate`, and is mapped into both the producer and consumer processes using `mmap`. The layout of the region is:

```
typedef struct {
    char data[MSG_SIZE];
    int len;
} message_t;

typedef struct {
    int head, tail;
    message_t buffer[BUFFER_SIZE];
    sem_t empty;
    sem_t full;
    sem_t mutex;
} shm_channel_t;
```

The ring buffer holds `BUFFER_SIZE` (1024) fixed-size messages. Three named POSIX semaphores coordinate access:

- `empty`: counts available slots (initialized to `BUFFER_SIZE`).
- `full`: counts filled messages (initialized to 0).
- `mutex`: serves as a binary mutex for updating `head` and `tail`.

A sender first decrements `empty`, locks `mutex`, writes the message to `buffer[tail]`, advances `tail`, unlocks `mutex`, and finally increments `full`. The receiver symmetrically decrements `full`, locks the mutex, reads from `buffer[head]`, advances `head`, unlocks the mutex, and increments `empty`. Because the data never cross the kernel boundary, the only system calls incurred are the semaphore operations `sem_wait` and `sem_post`, which are amortized over the message payload size.

4 Opponent's Enhanced Kernel IPC

The peer group designed a significantly more sophisticated kernel IPC driver, which also exposes a character device node (`/dev/hello`) but incorporates several advanced features.

4.1 Data Structures

Three principal structures underpin the driver:

- `data_node`: The per-pipe descriptor containing a union that packs `head` and `tail` into a single `unsigned long` (allowing atomic read-modify-write updates), a pointer to an array of `buffer_node` entries, a mutex, two wait queues (`rq` for readers, `wq` for writers), and reader/writer reference counts.
- `buffer_node`: Each ring-buffer slot is not a flat byte array but a structure holding a `struct page *`, an `offset`, and a `len`. Data are stored directly in kernel pages obtained from the page allocator, enabling zero-copy page-level transfers and efficient handling of messages that span multiple slots.
- `cgroup_pipe_entry`: Pipes are organized in a red-black tree keyed by the cgroup ID of the opening process. This provides automatic isolation: processes in different cgroups are assigned distinct pipe instances without manual configuration.

4.2 Key Optimizations

Small Write Optimization. If a write operation leaves a small amount of residual data, the driver checks whether the last page has sufficient free space; if so, the data are appended directly, avoiding an additional page allocation.

Exclusive Wake-up. To prevent the thundering herd problem, processes are placed on the wait queue with the exclusive flag set. The `wake_up_interruptible_exclusive` macro wakes only the first exclusive waiter, dramatically reducing spurious context switches under high concurrency.

Poll/Select Support. The driver implements the `poll` file operation, allowing user-space programs to multiplex I/O across many file descriptors using `select`, `poll`, or `epoll`. When the pipe becomes readable or writable, the driver wakes the appropriate wait queue, enabling event-driven architectures.

Atomic Head/Tail Updates. Embedding `head` and `tail` inside a single `unsigned long` union permits the driver to update both pointers with a single atomic instruction, simplifying the determination of empty/full conditions and improving consistency guarantees.

5 Experimental Setup

All benchmarks were conducted on a 64-bit Linux virtual machine equipped with 2 vCPUs and 4 GB of RAM. The kernel version was 6.8.0-101-generic, and GCC 13.3.0 was used as the compiler. For the opponent's driver, an additional compatibility layer was required to adapt to API changes in Linux 6.4+.

Three IPC mechanisms were evaluated:

1. Our kernel driver (`MyIPC`) — device nodes `/dev/myipc0`, `/dev/myipc1`.
2. Our shared-memory + semaphore IPC.
3. Opponent's enhanced driver (`/dev/hello`).

Two types of workload were applied:

- **Round-trip Time (RTT) Latency**: Ping-pong test with 1-byte messages, repeated for 10,000 iterations for MyIPC and 1,000 iterations for the opponent's driver.
- **Bandwidth / Throughput**: A fixed total of 64 MB of data (MyIPC) or 100,000 messages (shared memory, opponent) transferred using uniform chunk sizes of 512 B, 256 B, or 1024 B, depending on the test.

For the opponent's driver we also employed the open-source `ipc-bench` framework to guarantee precise, sub-millisecond timing measurements.

6 Results and Comparison

6.1 Latency Analysis

Table 1 reports the round-trip latency for 1-byte messages. MyIPC achieves an RTT comparable to shared memory and slightly lower than that of a traditional Unix pipe, demonstrating that a thin kernel driver with a dedicated ring buffer can be efficient for small control messages. The opponent's enhanced driver posts an average RTT of 431.9 μs, with a minimum of 153.6 μs but a large standard deviation of 418.9 μs. In a separate 1 KB ping-pong test, the opponent's driver reduces latency by approximately 146 μs (roughly 28%) compared to a standard pipe. Despite these gains, shared memory still holds the latency crown because it avoids any kernel entry at all.

Table 1: Average round-trip latency for 1-byte messages.

Method	**Iterations**	**Avg RTT (μs)**	**Min RTT (μs)**
Unix Pipe (baseline)	10,000	slightly higher than MyIPC	–
MyIPC (kernel driver)	10,000	close to shared memory	–
Shared Memory + Sem.	10,000	lowest	–
Opponent's Driver	1,000	431.9	153.6

6.2 Throughput and Bandwidth

Tables 2 and 3 present throughput for 256 B and 1024 B messages. For small 64-byte payloads, the pipe outperforms the shared-memory approach because the cost of `sem_wait` and `sem_post` system calls dominates the short copy time. As the message size grows to 256 B and beyond, shared memory overtakes pipe; at 1024 B the shared-memory mechanism achieves substantially higher message rates. The opponent's driver, when tested with 1 KB messages, improves throughput by 28% over pipe but remains below shared memory due to the unavoidable `copy_to_user` and `copy_from_user` overhead.

For bulk data transfer (64 MB total, 512 B chunks), MyIPC displays the lowest bandwidth among the three designs. The limitation stems from the small, statically sized ring buffer, which forces frequent writer sleeps, and from the single-mutex serialization that prevents concurrent copying. Pipe achieves the highest raw bandwidth here because its kernel buffer management is highly tuned over decades of development.

Table 2: Throughput for 256-byte messages (100,000 messages).

Method	**Time (s)**	**Throughput (msg/s)**
Shared Memory + Semaphore	0.1223	817,482
Unix Pipe	0.1437	695,989

Table 3: Throughput for 1024-byte messages (100,000 messages).

Method	**Time (s)**	**Throughput (msg/s)**
Shared Memory + Semaphore	0.3754	266,417
Unix Pipe	0.4180	239,216

6.3 Discussion

The performance hierarchy can be understood through a simple analytical model. Let T_{sys} be the fixed cost of entering and exiting the kernel (including register save/restore and mode switch), T_{copy} the per-byte cost of `copy_to_user` or memory copy, L the message length in bytes, and T_{sync} the overhead of synchronization operations. For a pipe or kernel driver, the one-way latency is approximately

$$T_{\text{kernel}} = 2\,T_{\text{sys}} + L \cdot T_{\text{copy}}. \tag{1}$$

For shared memory, the latency is

$$T_{\text{shm}} = 2\,T_{\text{sync}} + L \cdot T_{\text{copy}}, \tag{2}$$

where T_{sync} is the cost of a semaphore call, typically comparable to T_{sys} but incurred only once per message regardless of the number of bytes. When L is small, T_{sync} dominates and the pipe's more aggressive buffering may give it a slight edge. As L increases, the lack of a second kernel crossing and the associated copy in (1) allows T_{shm} to grow more slowly, eventually crossing over at around 64–256 bytes.

MyIPC performs like a pipe but with a simpler (and smaller) buffer, therefore it cannot amortize system-call costs as effectively for large transfers. The opponent's enhanced driver reduces T_{sys} overhead by using page-sized buffers and exclusive wake-up, which cuts down the number of block/wake cycles; its poll support further enables concurrency. However, it remains fundamentally a kernel-mediated copy path, so it cannot outperform shared memory for large L. Its true strength lies in features orthogonal to raw throughput: cgroup-aware isolation, safe automatic cleanup, and compatibility with standard event-loops via `select`/`epoll`.

7 Conclusion

In this project we implemented and evaluated three custom IPC mechanisms: a simple kernel character driver with a FIFO ring buffer, a user-space shared memory plus semaphore design, and an advanced kernel driver featuring page-level buffering, exclusive wake-up, and cgroup-based pipe instances. Experimental results confirm theoretical expectations:

- For small control messages (1—64 B), kernel-mediated mechanisms (pipes, drivers) are competitive, with latencies in the microsecond range.
- For larger bulk transfers (256 B and above), shared memory offers the highest throughput because it eliminates kernel-user copies.
- The opponent's enhanced driver achieves meaningful overhead reductions over simple kernel drivers and pipes through smart buffer management and synchronization optimizations, and adds valuable features such as poll support and per-cgroup isolation, making it suitable for complex, multi-tenant server environments.

Future work may explore lock-free ring buffers based on futexes or memory barriers, NUMA-aware shared memory allocation, and zero-copy techniques such as `splice` or `tee` that could further narrow the gap between kernel and user-space IPC.

Acknowledgments. We thank the peer group for sharing their design and benchmark data, which greatly enriched this comparative study.