Operating Systems Project: Topic 11

Asynchronous I/O Benchmark and Enhancement (io_uring)

Liangsen Wang

224040364@link.cuhk.edu.cn

2026.1.29

Outline

Outline

The Mission: Topic 11 Requirements

Objective: Compare the performance of the modern `io_uring` interface against traditional Synchronous I/O.

Requirement 1: Benchmark (User Space)

- **Task:** Write a C program to perform file I/O using both `read`/`write` and `io_uring`.
- **Metrics:** Measure **Latency**, **Throughput (IOPS)**, and **Syscall Counts**.
- **Goal:** Prove that `io_uring` is faster for high-concurrency workloads.

Requirement 2: Kernel Enhancement (Advanced)

- **Target:** `fs/io_uring.c`.
- **Task:** Modify submission logic or optimize queue contention.

The Question:
Can we do I/O without paying for System Calls?

Advanced Options

Option A: New Submission Strategy

- **Current:** `io_uring` processes requests FIFO.
- **Idea:** Implement "Priority Submission". Add a flag to urgent requests so the kernel processes them before others in the batch.

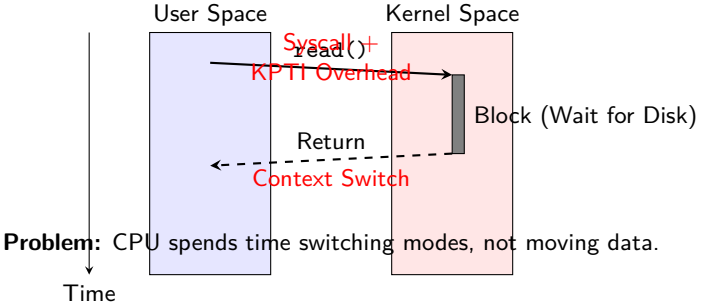
Option B: Lock Contention Analysis

- **Problem:** Multiple threads sharing one Ring Buffer fight for the `SQ` lock.
- **Task:** Analyze lock contention using `perf` and propose optimizations (e.g., per-CPU rings).

Outline

Theory 1: The Cost of Synchronous I/O

Why is `read()` slow? Because of **Context Switches** and **Spectre/Meltdown Mitigations**.



Theory 2: Why not Legacy AIO?

Linux already had `libaio` (`io_submit`). Why create `io_uring`?

Legacy AIO Problems:

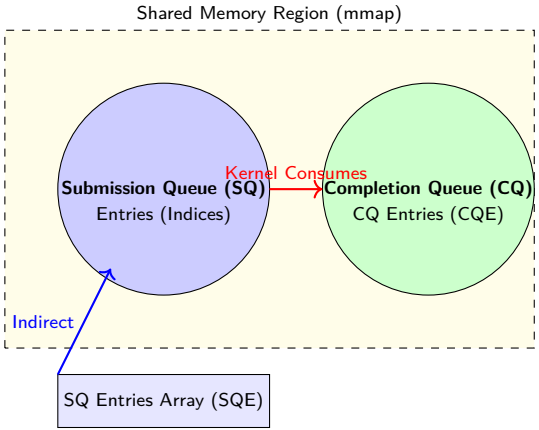
- **Direct I/O Only:** Only worked with `O_DIRECT` (bypassing cache). Buffered I/O would still block!
- **Complex API:** Hard to use correctly.
- **Data Copy:** Still required copying data structures for every submission.

io_uring Solutions:

- **Universal:** Works for Buffered I/O, Network Sockets, etc.
- **Zero Copy:** Shared memory ring buffers.
- **Extensible:** Designed to support any syscall asynchronously.

Theory 3: The io_uring Architecture

Core Concept: Two circular ring buffers mapped into user space.



User writes SQE.
Kernel writes CQE.
No copying!

Outline

Advanced 1: Lockless Coordination

How do User and Kernel share a ring without locks? **Memory Barriers**.

The Producer-Consumer Contract

- **User (Producer):** Writes SQE $\rightarrow$ `smp_store_release(tail)`
- **Kernel (Consumer):** `smp_load_acquire(tail)` $\rightarrow$ Reads SQE

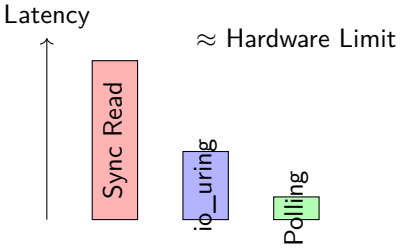
Why this matters: Explicit locking (‘mutex‘) puts threads to sleep. Memory barriers just ensure ordering, allowing simpler wait-free progress in the fast path.

Advanced 2: IOPOLL (Zero Syscalls)

Standard `io_uring` still needs 1 syscall (`io_uring_enter`) to wake the kernel. **Polling Mode (`IORING_SETUP_IOPOLL`)** eliminates even that.

Mechanism:

- Kernel creates a specific kernel thread (`io-wq`).
- This thread **spin-loops**, checking the SQ ring for new entries.
- User simply writes to memory. Kernel picks it up instantly.



Trade-off: High CPU usage (100% on one core) for ultra-low latency.

Advanced 3: Registered Buffers & Files

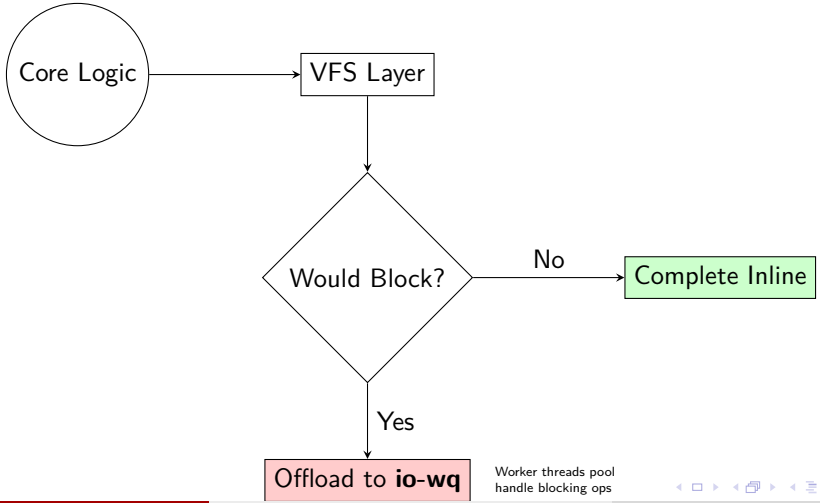
Problem: For every I/O, the kernel must: 1. Map user virtual address to physical pages (`get_user_pages`). 2. Lookup file descriptor references (`fget`).
Solution: Pre-Registration

- User maps buffers **once** at startup.
- Kernel pins pages in RAM permanently.
- During I/O, skip mapping and reference counting.

Result: Another 10-20% performance gain.

Advanced 4: What if it blocks?

If an operation cannot be non-blocking (e.g., buffered read not in cache), `io_uring` cannot just block the whole ring.



Outline

Implementation 1: liburing Setup

Don't use raw syscalls. Use **liburing**.

```c
#include <liburing.h>

#define QUEUE_DEPTH 4096
struct io_uring ring;

// Setup
int ret = io_uring_queue_init(QUEUE_DEPTH, &ring, 0);
if (ret < 0) {
    fprintf(stderr, "queue_init failed: %d\n", ret);
    return 1;
}

```

Implementation 2: The Submission Loop

```c
struct io_uring_sqe *sqe;
struct iovec iov;

// 1. Get an entry from the ring
sqe = io_uring_get_sqe(&ring);

// 2. Prepare the operation (PREP macro)
io_uring_prep_readv(sqe, fd, &iov, 1, offset);

// Optional: Set User Data (to identify this request later)
io_uring_sqe_set_data(sqe, my_data_ptr);

// 3. Submit (Batched!)
// This only does the syscall if the ring is full or you force it
io_uring_submit(&ring);

```

Implementation 3: The Completion Loop

```c
struct io_uring_cqe *cqe;
unsigned head;
int count = 0;

// Efficiently iterate over completed events
io_uring_for_each_cqe(&ring, head, cqe) {
    count++;

    // Check result
    if (cqe->res < 0) {
        // Handle Error
    }

    // Process data...
}

// Mark events as seen (Advance the Ring)
io_uring_cq_advance(&ring, count);
```

Kernel 1: The Battlefield (fs/io_uring.c)

For the advanced requirement, you must edit the kernel.

Key Functions

- `io_uring_enter()`: The system call entry point.
- `io_submit_sqes()`: Iterates the user's SQ ring.
- `io_issue_sqe()`: Attempts to issue a single request.

Task: Insert a `printk` or logic inside `io_submit_sqes` to count how many requests are processed in one batch.

Kernel 2: Implementing Priority

```
// In fs/io_uring.c :: io_submit_sqes(...)

// Loop through SQEs
io_for_each_link(req, head) {

    // YOUR MODIFICATION: Check a flag
    // Assuming you added a flag IOSQE_URGENT to include/uapi/linux/io_uring.h

    if (req->flags & IOSQE_URGENT) {
        // Handle High Priority: Maybe dispatch to a special workqueue?
        trace_printk("High Prio Req detected!\n");
    }

    io_submit_sqe(req, ...);
}

```

Outline

Evaluation 1: Designing the Test

You need to stress the system to see the difference.

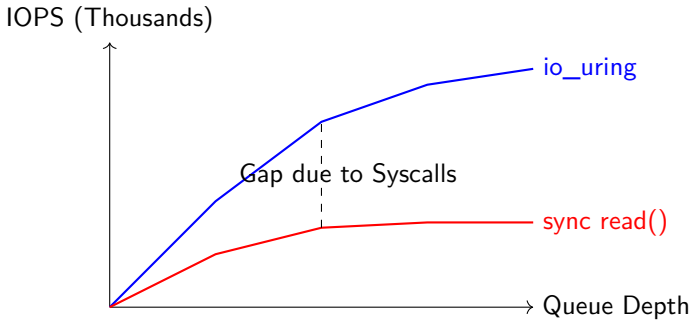
Test A: Throughput (IOPS)

- 4KB Random Reads.
- Queue Depth = 128.
- **Target:** 500k+ IOPS on NVMe.

Test B: Latency

- Queue Depth = 1.
- Measure time per operation.
- **Target:** Compare Polling mode vs. Interrupt mode.

Evaluation 2: What success looks like



If the lines overlap, you are likely limited by the Disk, not the Software. Use a RAM-disk or faster NVMe for testing.

Outline

Summary

1. **Problem:** Synchronous I/O wastes CPU cycles on system calls and context switches.
2. **Solution:** `io_uring` uses shared memory rings to batch submission and completion.
3. **Implementation:** Use `liburing` for the user-space benchmark. Modify `fs/io_uring.c` for kernel enhancements.
4. **Result:** Order-of-magnitude improvement in IOPS for high-speed devices.

Roadmap & Resources

Action Plan:

1. **Read:** `man io_uring_enter`.
2. **Baseline:** Write the `read()` benchmark loop.
3. **Experiment:** Write the `io_uring` version. Enable Polling.
4. **Hack:** Compile kernel and inject counters into the submission loop.

Resources:

- *"Efficient IO with io_uring"* (Jens Axboe, Kernel Maintainer).
- `liburing` GitHub repository (Examples folder is gold).