Operating Systems Project: Topic 9

Implement and Evaluate a Custom Disk Scheduler

Liangsen Wang

224040364@link.cuhk.edu.cn

2026.1.27

Outline

Outline

The Mission: Topic 9 Requirements

Objective: Implement a new I/O scheduling algorithm in the Linux `blk-mq` layer.

Requirement 1: Implementation

- **Base:** Study existing schedulers like `mq-deadline` or `kyber`.
- **Core Logic:** Implement a **Priority-Aware** scheduler.
- **Goal:** High-priority tasks (e.g., DB) must preempt low-priority tasks (e.g., Backup).

Requirement 2: Evaluation

- **Tool:** Use `fio` to generate mixed workloads.
- **Metrics:** Throughput (IOPS), Latency (Completion Latency), and CPU Overhead.

Advanced Options

Option A: Adaptive Strategy

- **Concept:** Dynamically switch logic based on workload patterns.
- **Scenario:** If sequential I/O detected $\rightarrow$ Merge aggressively. If random I/O $\rightarrow$ Dispatch immediately.

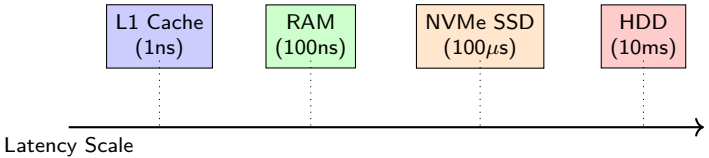
Option B: SSD vs HDD Optimization

- **HDD:** Physics dictates performance (Head Movement). Sorting is crucial.
- **SSD:** Internal Parallelism dictates performance. Queue depth is crucial.
- **Task:** Compare your scheduler's behavior on both device types.

Outline

Theory 1: The CPU-Disk Gap

Why do we need a scheduler at all? Why not just FIFO?



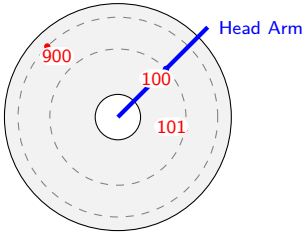
The Block Layer's Job:
Hide this massive latency via
Batching, Reordering, and Merging.

Theory 2: HDD Physics (The Elevator Algorithm)

For Hard Drives, **Seek Time** dominates. Moving the mechanical arm is expensive.
Scenario: Head at 100. Requests: 900, 101.

- **FIFO:** $100 \rightarrow 900 \rightarrow 101$. Total Seek: $800 + 799 = 1599$ tracks.
- **Elevator (Sorting):** $100 \rightarrow 101 \rightarrow 900$. Total Seek: $1 + 799 = 800$ tracks.

Conclusion: Reordering doubles performance on HDD.

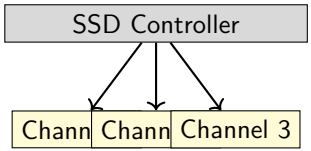


Theory 3: SSD Physics (Internal Parallelism)

SSDs have no moving parts. Seek time is zero. However, they have multiple internal NAND Channels.
Implication:

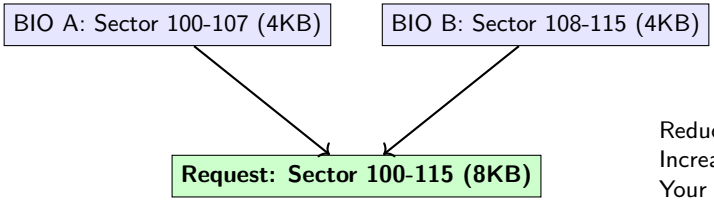
- Sorting by address matters less.
- **Queue Depth** matters more. You must flood the device with requests to keep all channels busy.
- Priority Scheduling is more effective here because the device is fast enough to switch contexts instantly.

Requests must be parallel!



Theory 4: Merging (The Silent Optimization)

Before scheduling, the kernel tries to **Merge**.



Reduces overhead.
Increases throughput.
Your scheduler receives the **Merged Request**.

Outline

Mechanics 1: Evolution of the Block Layer

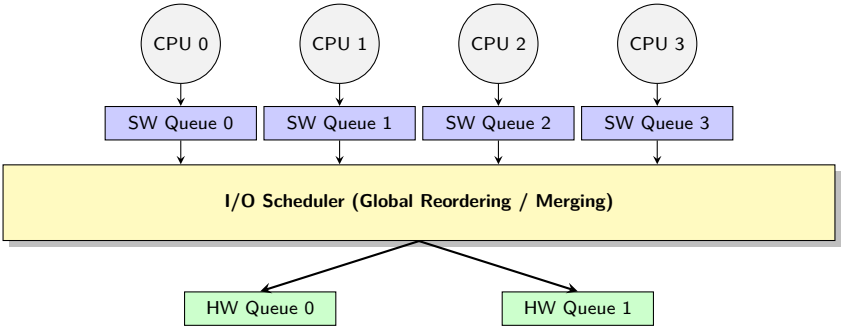
Legacy (Single Queue)

- Single Request Queue per device.
- Protected by one Global Lock.
- **Bottleneck:** CPU contention on multi-core systems.

blk-mq (Multi-Queue)

- **Software Queues:** One per CPU (No locking!).
- **Hardware Queues:** Mapped to device channels.
- **Your Scheduler:** Sits between SW and HW queues.

Mechanics 2: blk-mq Architecture



Task: You implement the logic inside the Yellow Box.

Mechanics 3: The Request Structure

You interact with `struct request`.

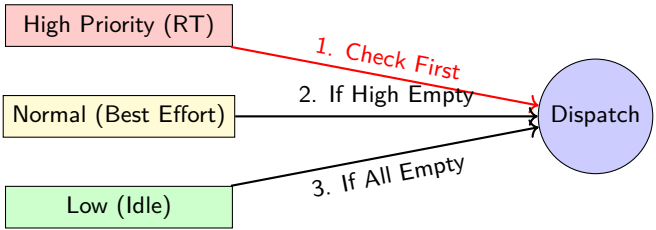
```
// include/linux/blk-mq.h
struct request {
    struct request_queue *q;
    struct list_head queuelist; // Used to chain requests in your scheduler

    unsigned short ioprio;      // <--- YOUR FOCUS
    // IOPRIO_PRIO_CLASS(ioprio) -> RT, BE, IDLE
    // IOPRIO_PRIO_DATA(ioprio)  -> Level 0-7

    sector_t __sector;          // Target address
    unsigned int __data_len;    // Size
    ...
};
```

Key Idea: Extract `ioprio` and put the request into the correct linked list.

Outline

Step 1: The Design (Multi-Level Queue)

We will implement a simple **Strict Priority Scheduler**.



The Risk: Starvation. If High Priority is always full, Low Priority never runs.

Step 2: Solving Starvation (Aging)

To make it robust (and get better grades), implement **Aging**.

The Logic

1. Add a timestamp `arrival_time` to the request when inserting.
2. During dispatch, check the Low Priority list.
3. If `(Current Time - arrival_time) > STARVATION_LIMIT`:
4. **Promote** the request to High Priority temporarily.

This ensures that even low-priority backups eventually complete.

Step 3: The Skeleton Code

Copy `block/mq-deadline.c` as a template.

```
struct my_data {
    struct list_head high_prio_list;
    struct list_head low_prio_list;
};

static bool my_dispatch_request(struct blk_mq_hw_ctx *hctx) {
    struct my_data *d = hctx->queue->elevator->elevator_data;
    struct request *rq;

    // 1. Try High Prio
    if (!list_empty(&d->high_prio_list)) {
        rq = list_first_entry(&d->high_prio_list, ...);
        list_del_init(&rq->queuelist);
        blk_mq_dispatch_rq_list(hctx, &list, ...);
        return true;
    }
    // 2. Try Low Prio ...
    return false;
}

```

Step 4: Registering the Scheduler

```
static struct elevator_type my_sched = {
    .ops = {
        .insert_requests = my_insert_requests,
        .dispatch_request = my_dispatch_request,
        .next_request = my_next_request,
        .init_sched = my_init_sched,
        .exit_sched = my_exit_sched,
    },
    .elevator_name = "my_prio_sched",
    .elevator_owner = THIS_MODULE,
};

// In module_init:
elv_register(&my_sched);
```

Outline

Evaluation 1: FIO Workload Design

You need to simulate contending processes. Use `fio`.

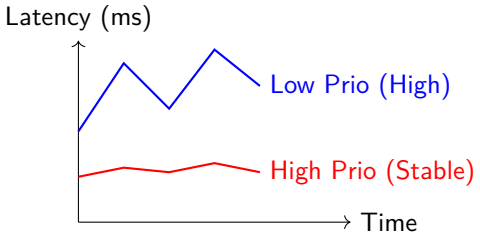
```
[global]
ioengine=libaio
direct=1
size=1G
time_based
runtime=30

[database-sim]
prio=1  # High
rw=randread

[backup-sim]
prio=7  # Low
rw=read

```

Expected Result:

- **High Prio:** High IOPS, Low Latency.
- **Low Prio:** Low IOPS, High Latency (but non-zero!).

Evaluation 2: Visualization Requirement

Generate charts to prove your scheduler works.



Example of expected outcome

Comparison: Run the same test on `none` or `mq-deadline` to show the difference.

Outline

Summary & Roadmap

1. **Compile:** Build the kernel and ensure you can switch schedulers.
2. **Clone:** Copy ‘mq-deadline.c‘ to start.
3. **Implement:**
   - Modify `struct elevator_queue` to have 3 lists.
   - Modify `insert_requests` to parse `ioprio`.
   - Modify `dispatch_request` to check lists in order.
4. **Refine:** Add Aging to prevent starvation.
5. **Test:** Run FIO script and plot graphs.

Resources: `block/blk-mq.c`, `include/linux/ioprio.h`