Topic 9 Final Report

Implementing and Evaluating a Priority-Aware blk-mq Disk Scheduler

Team 17

Sengnam Chen Student ID: 225040187
225040187@link.cuhk.edu.cn

Yuxuan Fan Student ID: 225040159
225040159@link.cuhk.edu.cn

May 2, 2026

Contents

Abstract 4

1 Introduction 5
- 1.1 Background . . . 5
- 1.2 Problem statement . . . 5

2 Background and Key Concepts 5
- 2.1 blk-mq and elevator scheduling . . . 5
- 2.2 BIO versus request . . . 5
- 2.3 I/O priority classification . . . 6
- 2.4 Why strict priority needs aging . . . 6
- 2.5 HDD versus SSD scheduling intuition . . . 6
- 2.6 Request merging . . . 6

3 Implementation 6
- 3.1 Basic scheduler overview . . . 6
- 3.2 Priority classification . . . 7
- 3.3 Insertion path in the basic scheduler . . . 7
- 3.4 Aging and dispatch in the basic scheduler . . . 8
- 3.5 Advanced scheduler: adaptive mode detection . . . 8
- 3.6 Advanced scheduler: media-aware request selection . . . 9
- 3.7 Advanced scheduler: merge-aware behavior . . . 9
- 3.8 Advanced scheduler: maintaining multiple data structures . . . 10

4 Experimental Design 10
- 4.1 Workload design . . . 10
- 4.2 Compared configurations . . . 11
- 4.3 Metrics and repeated runs . . . 11

5 Experimental Results 11
- 5.1 IOPS Results . . . 11
  - 5.1.1 Basic Scheduler Group . . . 11
  - 5.1.2 Advanced Scheduler Group . . . 12
  - 5.1.3 IOPS Interpretation . . . 13
- 5.2 P99 Latency Results . . . 13
  - 5.2.1 Basic Scheduler Group . . . 13
  - 5.2.2 Advanced Scheduler Group . . . 14
  - 5.2.3 P99 Latency Interpretation . . . 15
- 5.3 CPU Overhead Results . . . 15
- 5.4 Latency Time-Series Results . . . 16
  - 5.4.1 Basic Scheduler Group . . . 16
  - 5.4.2 Advanced Scheduler Group . . . 17
- 5.5 Immediate Observations . . . 18

6 In-Depth Analysis 18
- 6.1 Overall Assessment . . . 18
- 6.2 Why the Basic Scheduler Does Not Show Strong Gains . . . 19
- 6.3 Why the Advanced Scheduler Improves Relative Performance . . . 19
  - 6.3.1 Adaptive Strategy . . . 19
  - 6.3.2 Media-Aware Behavior . . . 19
  - 6.3.3 Merge-Aware Behavior . . . 20
- 6.4 Why High-Priority Latency Is Not Always Absolutely Lower . . . 20
- 6.5 Why CPU Overhead Does Not Increase Much . . . 20
- 6.6 Why Error Bars Matter . . . 20
- 6.7 Comparison with `mq-deadline` . . . 21
- 6.8 Final Evaluation of the Experimental Results . . . 21

7 Discussion 21
- 7.1 Main value of the project . . . 21

8 Limitations and Future Work 22
- 8.1 Limitations . . . 22

8.2 Future work 22
9 Conclusion **22**

List of Figures

1 IOPS results for the basic scheduler group. Each configuration was repeated five times, and the bars show mean IOPS with error bars. 12
2 IOPS results for the advanced scheduler group. The advanced scheduler becomes more competitive after adaptive and media-aware optimizations. 13
3 P99 latency results for the basic scheduler group. The basic scheduler shows limited tail-latency control. 14
4 P99 latency results for the advanced scheduler group. The advanced scheduler improves tail latency compared with the basic version. 15
5 CPU overhead results for the basic scheduler group. CPU time is similar across all schedulers. 16
6 CPU overhead results for the advanced scheduler group. The advanced logic does not introduce a large measurable CPU overhead. 16
7 Latency time series for the basic scheduler group. The basic scheduler shows noticeable run-time fluctuation. 17
8 Latency time series for the advanced scheduler group. The advanced scheduler shows more stable behavior after adaptive and media-aware optimizations. 18

List of Tables

1 Two-job fio workload for priority evaluation 11

Abstract

This project implements and evaluates a custom Linux blk-mq I/O scheduler for Topic 9. The basic scheduler uses three priority queues, `HIGH`, `NORMAL`, and `LOW`, and dispatches requests in strict priority order while applying aging to prevent starvation. The scheduler classifies requests from Linux I/O priority metadata, allowing latency-sensitive database-like requests to be separated from background backup-like requests. We then develop an advanced version with two extensions: an adaptive strategy that detects sequential versus random access patterns from request sectors, and a media-aware policy that treats rotational and non-rotational devices differently. The evaluation uses fio mixed workloads and compares `none`, `mq-deadline`, and `my_prio_sched`. Each configuration is repeated five times and averaged to reduce run-to-run noise. The results show that the basic scheduler is correct and interpretable but limited, while the advanced scheduler is more competitive in both IOPS and P99 latency. However, mature Linux schedulers remain strong baselines, and the observed results must be interpreted as a systems trade-off rather than a simple algorithmic win.

Keywords: Linux kernel; blk-mq; I/O scheduler; priority scheduling; aging; fio; mq-deadline; request merging; SSD; HDD

1 Introduction

1.1 Background

Storage I/O remains one of the major bottlenecks in modern operating systems. Even when CPU and memory are fast, block devices still introduce substantially higher latency. The Linux block layer reduces this gap through request batching, reordering, merging, and scheduling. In modern kernels, this work is handled by the blk-mq multi-queue block layer, which replaces the older single-queue design with software and hardware queues designed for multi-core and high-IOPS devices [1].

In real workloads, not all I/O requests have the same importance. A database query or latency-sensitive service request often requires lower latency, while a backup task can usually tolerate delay as long as it continues to make progress. A scheduler that only optimizes aggregate throughput may fail to express this service distinction. This motivates a custom priority-aware scheduler that explicitly separates high-priority and low-priority requests.

1.2 Problem statement

This project addresses three concrete questions:

1. Can we implement a new I/O scheduler in the Linux blk-mq elevator framework?
2. Can a simple priority scheduler provide differentiated service between a database-like workload and a backup-like workload?
3. Can adaptive and media-aware extensions improve the basic design by considering access pattern and storage-device behavior?

To answer these questions, we implemented two scheduler versions:

- **Basic**: three priority FIFO queues with aging-based starvation prevention;
- **Advanced**: the basic scheduler plus adaptive sequential/random detection, media-aware dispatch, batching, red-black-tree ordering, and merge-aware behavior.

2 Background and Key Concepts

2.1 blk-mq and elevator scheduling

The blk-mq layer organizes block I/O using per-CPU software queues and hardware dispatch queues. A scheduler, also called an elevator, can be selected at runtime through sysfs. It can decide how requests are inserted, merged, and dispatched. Linux provides mature schedulers such as `mq-deadline`, while `none` represents minimal scheduler intervention. The multiqueue design was introduced to reduce queue lock contention and improve scalability on SSDs and multi-core systems [1].

2.2 BIO versus request

A BIO, or Block I/O descriptor, describes a lower-level block operation, including operation type, target sector, and memory pages. A `struct request` is the scheduling object used by the block layer. One request may contain one or more BIOs. Our scheduler primarily inserts and dispatches requests, but the advanced version also implements BIO-related merge callbacks.

2.3 I/O priority classification

Linux I/O priority contains a priority class and a data field. The scheduler maps this information into three internal classes:

- `RT` → `HIGH`;
- `IDLE` → `LOW`;
- `NONE` → `NORMAL`;
- `BE` level 0–2 → `HIGH`;
- `BE` level 3–5 → `NORMAL`;
- `BE` level 6–7 → `LOW`.

This mapping allows fio workloads using `prio=1` and `prio=7` to be separated even when both are best-effort requests.

2.4 Why strict priority needs aging

A pure three-level priority queue may cause starvation: if high-priority requests continuously arrive, low-priority requests may never be dispatched. Therefore, before normal strict-priority dispatch, the scheduler checks whether lower-priority requests have waited too long. If so, they are dispatched earlier through aging.

2.5 HDD versus SSD scheduling intuition

HDDs are rotational devices, so request order affects seek distance and rotational delay. Sector-ordered dispatch can improve locality [5]. SSDs have no mechanical seek, but they benefit from internal parallelism and sufficient queue depth. Therefore, address sorting is less important than avoiding unnecessary software overhead and keeping requests flowing to the device.

2.6 Request merging

The Linux block layer already supports BIO and request merging. Our advanced scheduler does not create merging from scratch; it participates in existing merge mechanisms and adds scheduler-specific policy. In particular, it restricts merging to same-priority objects and enables stronger merge participation when the workload appears sequential. This is similar in spirit to how block schedulers provide merge callbacks while still relying on common block-layer helpers.

3 Implementation

3.1 Basic scheduler overview

The basic scheduler attaches a private data structure to `elevator_queue.elevator_data`. It maintains a special `dispatch` queue, three priority FIFO lists, and one aging threshold. The scheduler only performs insertion and dispatch based on system-provided priority order; it does not optimize by request address locality or hardware-specific behavior.

Listing 1: Private state of the basic scheduler

```
1 struct my_prio_data {
2     spinlock_t lock;
3     struct list_head dispatch;
4     struct list_head fifo[MY_PRIO_COUNT];
```

```
    unsigned long aging_expire;
};
```

3.2 Priority classification

The classification function converts Linux I/O priority into the scheduler's three internal queues. This is the first step that connects fio priority values with scheduler behavior.

Listing 2: Priority classification in the basic scheduler

```
static enum my_prio_class my_prio_classify_ioprio(unsigned short ioprio)
{
    const u8 class = IOPRIO_PRIO_CLASS(ioprio);
    const u8 level = IOPRIO_PRIO_DATA(ioprio);

    switch (class) {
    case IOPRIO_CLASS_RT:
        return MY_PRIO_HIGH;
    case IOPRIO_CLASS_IDLE:
        return MY_PRIO_LOW;
    case IOPRIO_CLASS_NONE:
        return MY_PRIO_NORMAL;
    case IOPRIO_CLASS_BE:
    default:
        if (level <= my_be_high_max)
            return MY_PRIO_HIGH;
        if (level >= my_be_low_min)
            return MY_PRIO_LOW;
        return MY_PRIO_NORMAL;
    }
}
```

3.3 Insertion path in the basic scheduler

During insertion, each request is classified, timestamped, and appended to the corresponding FIFO list. A special `BLK_MQ_INSERT_AT_HEAD` request is placed into the `dispatch` queue, which is checked before normal priority queues.

Listing 3: Basic request insertion path

```
static void my_insert_request(struct blk_mq_hw_ctx *hctx,
                              struct request *rq,
                              blk_insert_t flags)
{
    struct request_queue *q = hctx->queue;
    struct my_prio_data *d = q->elevator->elevator_data;
    const enum my_prio_class prio = my_prio_classify_rq(rq);

    rq->fifo_time = jiffies;

    if (flags & BLK_MQ_INSERT_AT_HEAD) {
        list_add(&rq->queuelist, &d->dispatch);
        return;
    }

    list_add_tail(&rq->queuelist, &d->fifo[prio]);
}
```

3.4 Aging and dispatch in the basic scheduler

The basic dispatch path first handles the special `dispatch` queue, then checks aging, and finally dispatches in strict priority order. This makes the scheduler simple and explainable while still avoiding starvation.

Listing 4: Basic dispatch order

```
if (!list_empty(&d->dispatch))
    return pop(d->dispatch);

rq = my_dispatch_aged_request(d, now);
if (rq)
    return rq;

if (!list_empty(&d->fifo[MY_PRIO_HIGH]))
    return pop(d->fifo[MY_PRIO_HIGH]);
if (!list_empty(&d->fifo[MY_PRIO_NORMAL]))
    return pop(d->fifo[MY_PRIO_NORMAL]);
return pop(d->fifo[MY_PRIO_LOW]);
```

3.5 Advanced scheduler: adaptive mode detection

The advanced version adds a lightweight sequential/random detector. It compares the current request's starting sector with the previous request's end sector. Small gaps increase `seq_score`; large gaps increase `rand_score`. The mode then controls batching and merge participation.

Listing 5: Sequential/random detector used in Option A

```
static void my_update_mode(struct my_prio_data *d, struct request *rq)
{
    sector_t pos = blk_rq_pos(rq);
    sector_t end = pos + blk_rq_sectors(rq);
    sector_t gap;

    if (!d->detect_last_end) {
        d->detect_last_end = end;
        return;
    }

    gap = (pos >= d->detect_last_end) ?
        pos - d->detect_last_end : d->detect_last_end - pos;

    if (gap <= d->seq_gap_sectors) {
        d->seq_score++;
        if (d->rand_score > 0)
            d->rand_score--;
    } else {
        d->rand_score++;
        if (d->seq_score > 0)
            d->seq_score--;
    }

    d->mode = (d->seq_score >= d->rand_score) ?
        MY_MODE_SEQUENTIAL : MY_MODE_RANDOM;
    if (d->mode == MY_MODE_RANDOM)
        my_reset_batch(d);

    d->detect_last_end = end;
}
```

3.6 Advanced scheduler: media-aware request selection

For SSD-like devices, the scheduler selects the FIFO head of the chosen priority class. For HDD-like devices, it uses a red-black tree ordered by sector and selects a request close to the last dispatch position. An expiration guard prevents old FIFO-head requests from waiting indefinitely.

Listing 6: SSD path: select the FIFO head

```
static struct request *my_take_fifo_head(struct request_queue *q,
                                         struct my_prio_data *d,
                                         enum my_prio_class prio)
{
    struct request *rq = my_peek_request(&d->fifo[prio]);
    if (!rq)
        return NULL;
    my_remove_request(q, d, prio, rq);
    return rq;
}
```

Listing 7: HDD path: sector-ordered selection with expiration guard

```
static struct request *my_take_sorted_request(struct request_queue *q,
                                              struct my_prio_data *d,
                                              enum my_prio_class prio,
                                              unsigned long now)
{
    struct request *rq, *fifo_head;
    struct rb_node *node;

    fifo_head = my_peek_request(&d->fifo[prio]);
    if (!fifo_head)
        return NULL;

    if (my_request_waited_expire(fifo_head, d->hdd_fifo_expire, now)) {
        my_remove_request(q, d, prio, fifo_head);
        return fifo_head;
    }

    rq = my_from_pos(&d->sort[prio], d->dispatch_last_sector);
    if (!rq) {
        node = rb_first(&d->sort[prio]);
        if (node)
            rq = rb_entry(node, struct request, rb_node);
    }
    if (!rq)
        rq = fifo_head;

    my_remove_request(q, d, prio, rq);
    return rq;
}
```

3.7 Advanced scheduler: merge-aware behavior

The advanced scheduler also adds merge-related callbacks. The key idea is to preserve priority semantics while enabling more merge opportunities under sequential mode. In other words, Linux already provides common merge mechanisms, and our scheduler adds policy conditions on top of them.

Listing 8: Same-priority merge constraint and sequential-mode merge attempt

```
static bool my_allow_merge(struct request_queue *q,
```

```
                          struct request *rq,
                          struct bio *bio)
{
    return my_prio_classify_rq(rq) == my_prio_classify_bio(bio);
}

static bool my_bio_merge(struct request_queue *q,
                         struct bio *bio,
                         unsigned int nr_segs)
{
    struct my_prio_data *d = q->elevator->elevator_data;
    struct request *free = NULL;
    bool ret;

    if (d->mode != MY_MODE_SEQUENTIAL)
        return false;

    spin_lock(&d->lock);
    ret = blk_mq_sched_try_merge(q, bio, nr_segs, &free);
    spin_unlock(&d->lock);

    if (free)
        blk_mq_free_request(free);
    return ret;
}
```

3.8 Advanced scheduler: maintaining multiple data structures

Unlike the basic version, the advanced version maintains a FIFO list, a red-black tree, and merge hash information. Therefore, a dispatched or merged request must be removed consistently from all relevant structures.

Listing 9: Consistent removal from scheduler structures

```
static void my_remove_request(struct request_queue *q,
                              struct my_prio_data *d,
                              enum my_prio_class prio,
                              struct request *rq)
{
    list_del_init(&rq->queuelist);

    if (!RB_EMPTY_NODE(&rq->rb_node))
        elv_rb_del(&d->sort[prio], rq);

    if (!hlist_unhashed(&rq->hash))
        elv_rqhash_del(q, rq);

    if (q->last_merge == rq)
        q->last_merge = NULL;
}
```

4 Experimental Design

4.1 Workload design

The evaluation uses fio to generate a mixed two-job workload. The high-priority job, `database_sim`, uses 4 KiB random reads and `prio=1`. The low-priority job, `backup_sim`, uses 128 KiB sequential reads and `prio=7`. This design intentionally models a latency-sensitive database workload competing with a lower-priority background backup workload.

Table 1: Two-job fio workload for priority evaluation

Job	Role	Pattern	Block size	Priority
`database_sim`	High-priority	random read	4 KiB	BE prio 1
`backup_sim`	Low-priority	sequential read	128 KiB	BE prio 7

4.2 Compared configurations

The experiments compare the following scheduler configurations:

- `none`: minimal scheduler intervention;
- `mq-deadline`: Linux's mature multiqueue deadline scheduler;
- `my_prio_sched-basic`: three priority queues plus aging;
- `my_prio_sched-optionab`: adaptive and media-aware upgrade.

4.3 Metrics and repeated runs

We evaluate IOPS, P99 latency, CPU overhead, and latency time series. Each group is repeated five times, and the reported bars show the mean across repeated runs. Error bars represent run-to-run variation. This is necessary because block-I/O experiments can fluctuate due to random sampling, SSD internal state, and virtualization effects.

5 Experimental Results

This section presents the experimental results of both the basic scheduler and the advanced scheduler under the mixed-priority fio workload. Each configuration was executed five times, and the reported results are averaged across repeated runs. Error bars are included to show run-to-run variability. This repeated-run design is important because the experiments were conducted in a VirtualBox virtual machine, where the measured performance can be affected by host-side CPU scheduling, virtual disk behavior, random address sampling, and transient device state.

The evaluation compares three schedulers: `none`, `mq-deadline`, and our custom `my_prio_sched`. The workload contains two jobs. The high-priority job, `database_sim`, models latency-sensitive database traffic using small random reads. The low-priority job, `backup_sim`, models background backup traffic using larger sequential reads. Therefore, the two workloads differ not only in priority but also in request size and access pattern. As a result, we should not directly interpret absolute latency differences between the two jobs as priority effects. Instead, the more meaningful comparison is to examine the same job under different schedulers and determine whether the custom scheduler improves high-priority service while preserving non-zero low-priority progress.

5.1 IOPS Results

5.1.1 Basic Scheduler Group

Figure 1 shows the IOPS results for the basic scheduler group. The basic version of `my_prio_sched` implements the core three-level priority queues and aging-based starvation prevention, but its throughput improvement is limited. For `backup_sim`, `mq-deadline` achieves the highest mean IOPS, while `my_prio_sched` performs below or close to the other baselines. For `database_sim`, `none` and `mq-deadline` also achieve higher average IOPS than the basic custom scheduler.

This result is reasonable because the basic scheduler only performs priority-based insertion and dispatch. It classifies requests into HIGH, NORMAL, and LOW FIFO queues, dispatches higher-priority requests first, and uses aging to prevent starvation. However, it does not consider request address locality, device-specific characteristics, or merge opportunities. In a mixed workload containing both random database traffic and sequential backup traffic, such a simple strict-priority policy may disrupt sequential locality and may not improve aggregate throughput.

The error bars in the basic group are also relatively large, especially for `my_prio_sched`. This indicates that the basic scheduler is sensitive to run-time conditions, such as random address distribution, virtualized storage behavior, and queue state. Therefore, the basic scheduler should be interpreted mainly as a functional baseline that verifies priority-aware queueing, rather than as a final optimized scheduler.

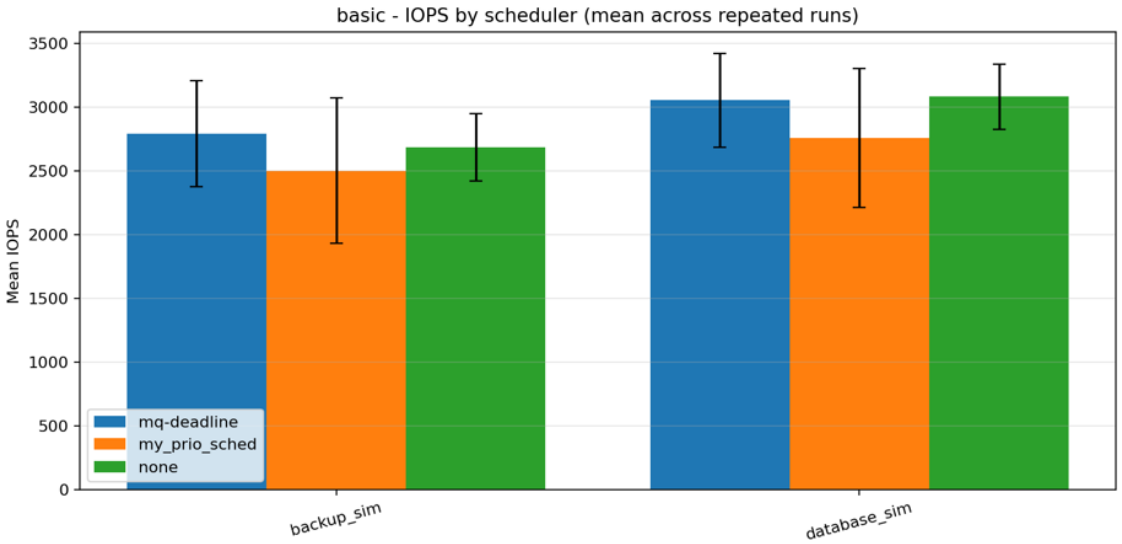

Figure 1: IOPS results for the basic scheduler group. Each configuration was repeated five times, and the bars show mean IOPS with error bars.

5.1.2 Advanced Scheduler Group

Figure 2 shows the IOPS results for the advanced scheduler group. Compared with the basic group, the relative performance of `my_prio_sched` improves noticeably. For `backup_sim`, the advanced custom scheduler achieves higher average IOPS than both `mq-deadline` and `none`. For `database_sim`, the advanced scheduler also reaches the highest or near-highest average IOPS among the compared schedulers.

This improvement shows that the two optimization directions are effective. Option A adds adaptive behavior by detecting whether the workload appears sequential or random from request sector gaps. Option B adds media-aware behavior by selecting different request-selection policies for HDD-like and SSD-like devices. In addition, the advanced scheduler introduces batching and merge-aware callbacks, allowing sequential workloads to benefit from more continuous request handling.

It is important to interpret this result carefully. The absolute IOPS values of the advanced group should not be compared too strongly against the basic group as if they were produced in an identical runtime state. The two groups may have been tested under different kernel builds or different experimental sessions, and the virtualized environment can introduce variability. The stronger conclusion is that, within the advanced group itself, `my_prio_sched` becomes more competitive against `mq-deadline` and `none`. This indicates that adaptive and media-aware optimizations make the custom scheduler more effective than the basic priority-only design.

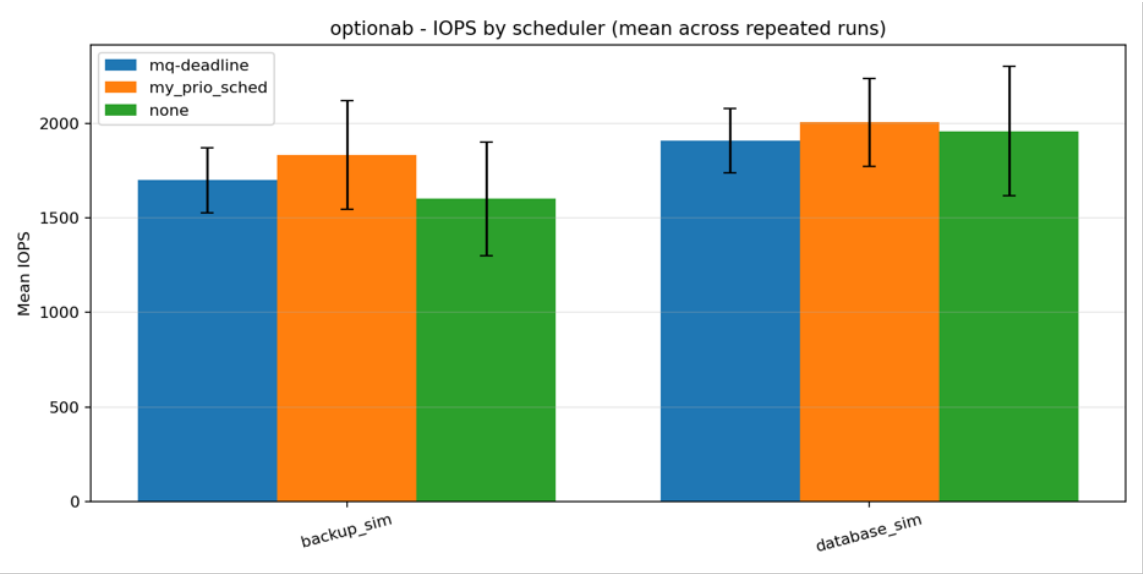

Figure 2: IOPS results for the advanced scheduler group. The advanced scheduler becomes more competitive after adaptive and media-aware optimizations.

5.1.3 IOPS Interpretation

The IOPS results show a clear contrast between the two implementations. The basic scheduler demonstrates that a simple strict-priority three-queue design is not sufficient to guarantee throughput gains under a mixed workload. The advanced scheduler, however, shows that combining priority-aware dispatch with adaptive batching, media-aware request selection, and merge-aware behavior can improve throughput.

Thus, the IOPS results support two conclusions. First, the basic scheduler is structurally correct but performance-limited. Second, the advanced scheduler better matches the characteristics of the workload and the storage stack, which makes it more competitive in throughput.

5.2 P99 Latency Results

5.2.1 Basic Scheduler Group

Figure 3 shows the P99 latency results for the basic scheduler group. P99 latency measures the 99th percentile of completion latency, which reflects the latency experienced by the slowest 1% of requests. Compared with mean latency, P99 latency is more useful for evaluating tail behavior in latency-sensitive workloads.

The basic `my_prio_sched` does not show strong tail-latency control. For `backup_sim`, its P99 latency is higher than or close to the baselines. For `database_sim`, its P99 latency is also higher than both `mq-deadline` and `none`. This means that although the basic scheduler can express priority ordering at the software queue level, it does not necessarily reduce the completion time of the slowest requests.

This result does not imply that the basic implementation is logically wrong. Rather, it shows the limitation of a simple priority policy. The basic scheduler decides which request leaves the scheduler queue first, but it does not control requests that are already in-flight below the scheduler. It also does not perform sector-aware ordering or merge-aware optimization. Therefore, once requests are submitted to the underlying virtual block device, the basic scheduler has limited control over their final completion order.

In addition, the aging mechanism in the basic version is designed primarily as a starvation guard. It ensures that low-priority requests do not wait forever, but it is not a fine-grained tail-

latency optimizer. Therefore, its limited P99 performance is consistent with the design scope of the basic version.

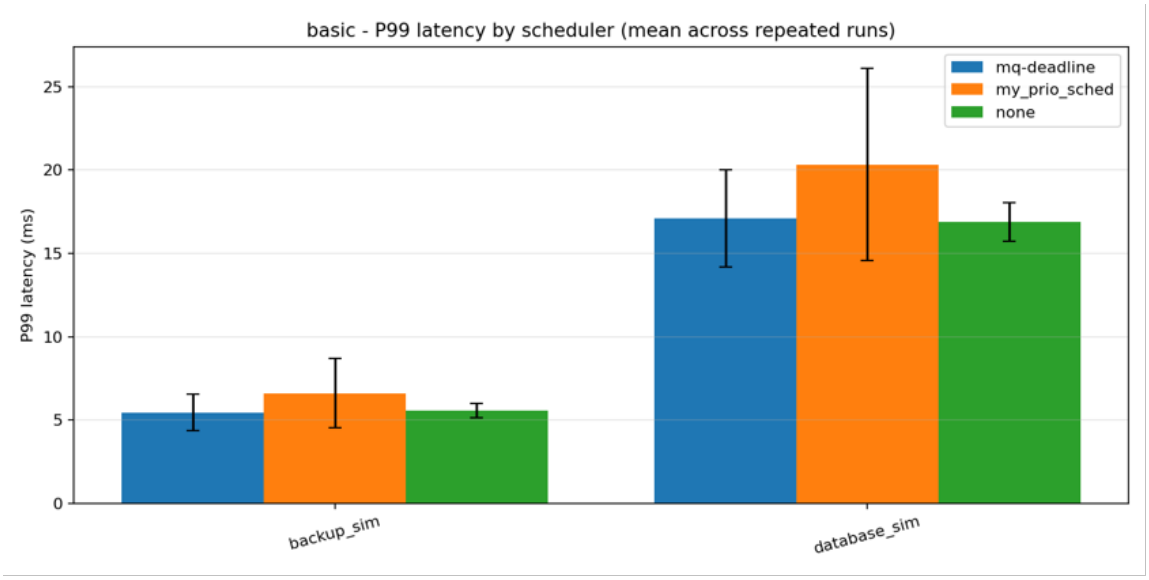

Figure 3: P99 latency results for the basic scheduler group. The basic scheduler shows limited tail-latency control.

5.2.2 Advanced Scheduler Group

Figure 4 shows the P99 latency results for the advanced scheduler group. Compared with the basic group, the advanced scheduler provides a much stronger tail-latency result. For `backup_sim`, `my_prio_sched` achieves a P99 latency lower than or close to the baselines. For `database_sim`, `my_prio_sched` achieves lower P99 latency than both `mq-deadline` and `none`.

This improvement is important because P99 latency captures the slowest portion of the request distribution. A scheduler may improve average throughput while still allowing occasional long delays. The advanced scheduler's P99 result suggests that the optimization does not only improve average throughput, but also helps control tail behavior.

The improvement can be explained by the advanced implementation. Sequential-mode batching reduces unnecessary repeated global arbitration and preserves more continuous service. The HDD path uses sector-sorted selection with an expiration guard, which balances locality and waiting-time control. The SSD path avoids unnecessary sorting and directly selects the FIFO head. Merge-aware callbacks also create additional opportunities to combine adjacent requests in sequential mode. Together, these mechanisms help reduce request fragmentation and stabilize tail latency.

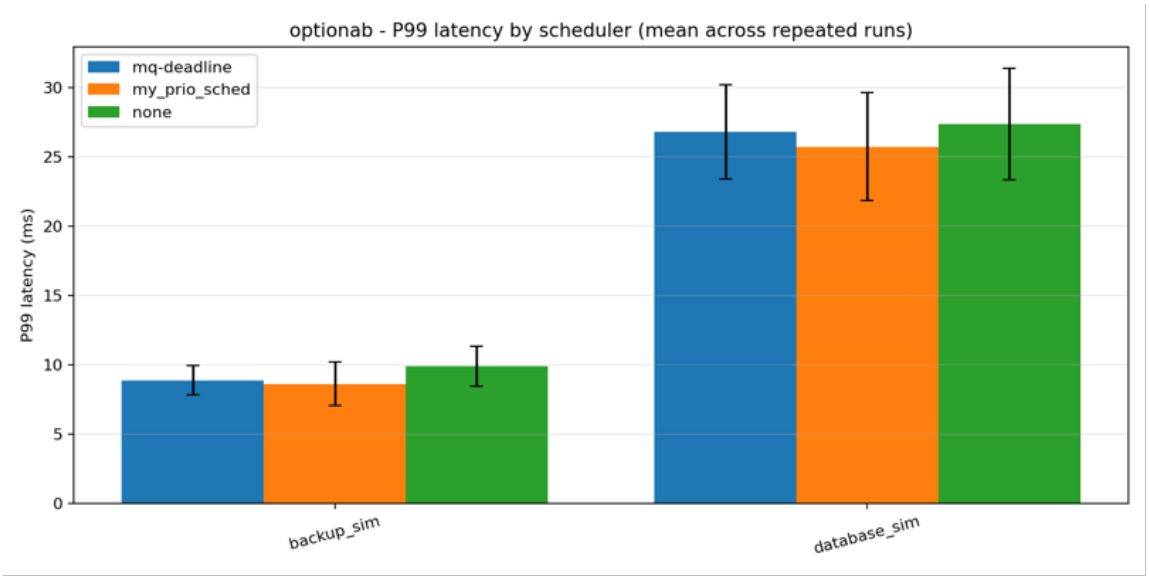

Figure 4: P99 latency results for the advanced scheduler group. The advanced scheduler improves tail latency compared with the basic version.

5.2.3 P99 Latency Interpretation

The P99 results are one of the strongest pieces of evidence distinguishing the basic and advanced versions. In the basic group, `my_prio_sched` does not control tail latency well, especially for the high-priority database workload. In the advanced group, however, `my_prio_sched` becomes competitive and even achieves the best P99 latency for `database_sim`.

However, it is important not to over-interpret absolute latency differences between `database_sim` and `backup_sim`. The database workload uses small random reads, while the backup workload uses larger sequential reads. Random small-block I/O is naturally more sensitive to queue state and virtualized device behavior. Therefore, the correct interpretation is not that high-priority latency must always be absolutely lower than low-priority latency. Instead, the key question is whether the same high-priority job improves under the custom scheduler compared with the baselines. Under this interpretation, the advanced scheduler shows a positive effect.

5.3 CPU Overhead Results

Figures 5 and 6 show the CPU overhead results for the basic and advanced groups. In both groups, the CPU time of `none`, `mq-deadline`, and `my_prio_sched` is very close. This indicates that the custom scheduler does not introduce a large measurable CPU overhead.

The basic scheduler adds priority classification, multiple FIFO queues, locking, and aging checks. The advanced scheduler further adds mode detection, batch-state maintenance, red-black tree insertion and deletion, merge-hash maintenance, merge callbacks, and media-aware dispatch paths. Even with these additional operations, the measured CPU time remains close to the baselines.

This suggests that the observed IOPS and latency differences are mainly due to I/O request organization rather than a major difference in CPU cost. However, this metric also has limitations. The measured CPU time mainly reflects fio-side CPU accounting and may not fully expose internal kernel lock contention or scheduler-path cost. It is best interpreted as a sanity check showing that the custom scheduler does not create an obviously excessive CPU burden.

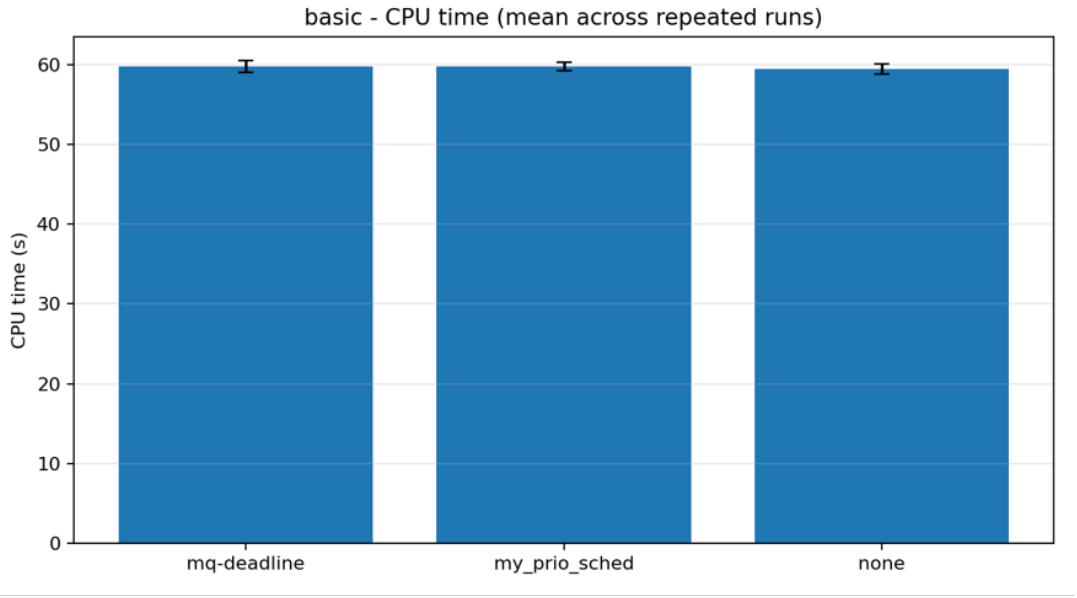

Figure 5: CPU overhead results for the basic scheduler group. CPU time is similar across all schedulers.

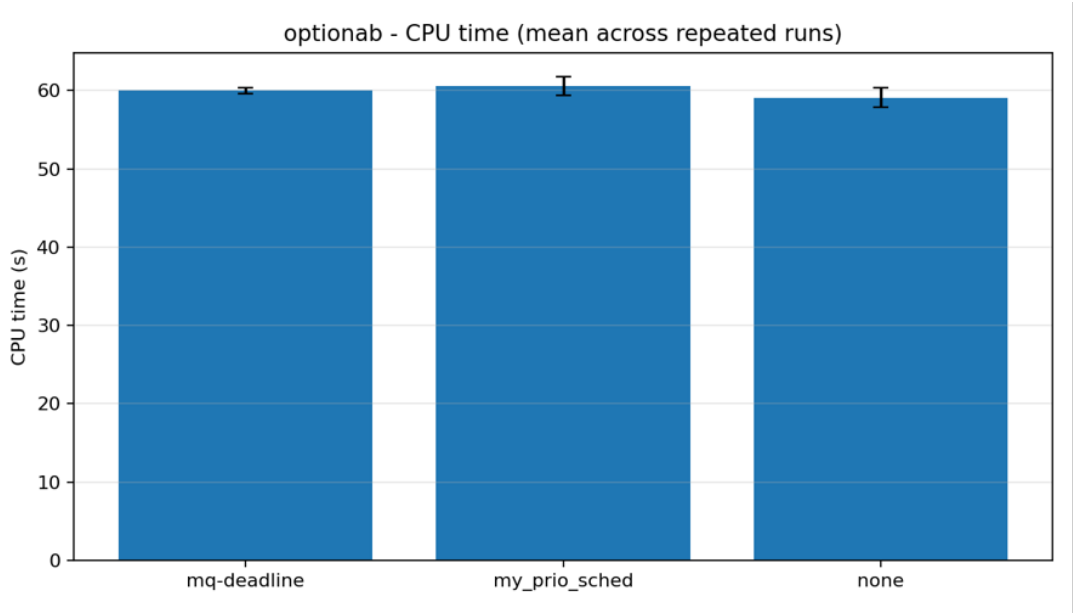

Figure 6: CPU overhead results for the advanced scheduler group. The advanced logic does not introduce a large measurable CPU overhead.

5.4 Latency Time-Series Results

5.4.1 Basic Scheduler Group

Figure 7 shows the completion-latency time series for the basic scheduler group. The curves show noticeable fluctuation across the 30-second run. The higher-latency curves generally correspond to `database_sim`, while the lower-latency curves generally correspond to `backup_sim`. This ordering should not be interpreted as a failure of priority scheduling. Instead, it mainly reflects the different workload patterns: `database_sim` uses small random reads, while `backup_sim` uses larger sequential reads.

The basic `my_prio_sched` curve still shows substantial variation, especially for the database workload. It does not consistently stay below the baseline curves. This is consistent with the P99 latency result: the basic scheduler expresses priority at the queue level, but does not provide strong time-series stability.

The `none` scheduler also shows spikes in some windows. This indicates that minimal scheduling intervention can reduce software overhead, but it does not necessarily prevent tail-latency spikes under a mixed workload. `mq-deadline` tends to behave more steadily because it is a mature scheduler with more balanced request organization and waiting-time control.

Overall, the basic time-series result shows that the basic scheduler can preserve low-priority progress but does not provide strong dynamic latency control.

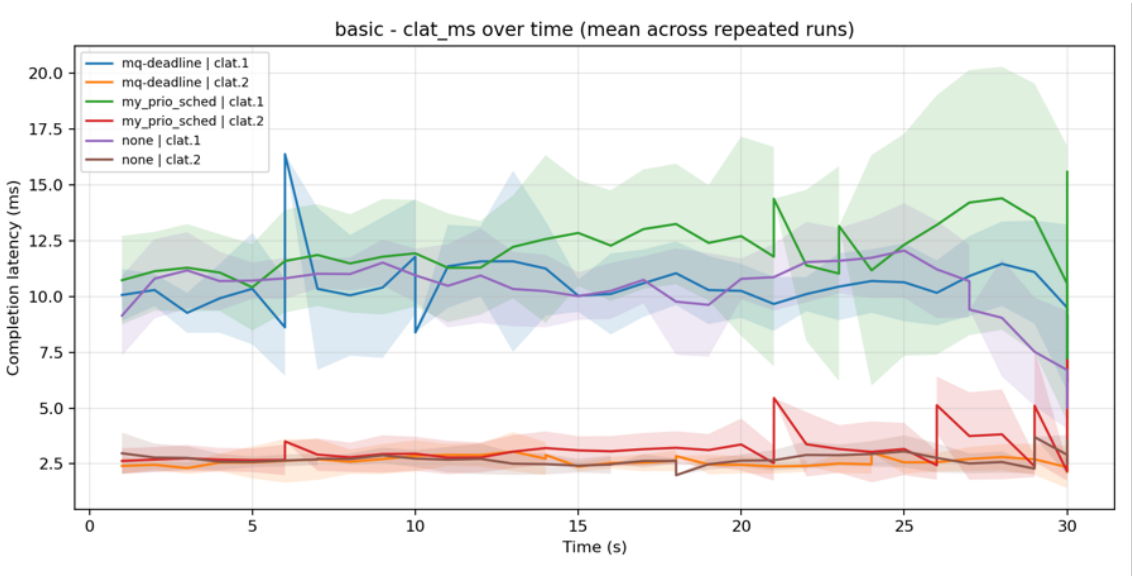

Figure 7: Latency time series for the basic scheduler group. The basic scheduler shows noticeable run-time fluctuation.

5.4.2 Advanced Scheduler Group

Figure 8 shows the latency time series for the advanced scheduler group. Compared with the basic group, the advanced scheduler shows more stable behavior. Although some fluctuations remain, the curves do not show prolonged uncontrolled latency growth. This is consistent with the improved P99 latency results.

For `backup_sim`, the advanced `my_prio_sched` maintains a relatively low latency range, showing that the low-priority job is not starved. For `database_sim`, the absolute latency remains higher than `backup_sim`, but this is expected because the database workload is random and uses smaller requests. The more meaningful observation is that, within the same job, the advanced scheduler improves P99 behavior compared with the baselines.

The advanced time-series result supports the design goal of the optimized scheduler. It does not merely chase the lowest instantaneous latency; instead, it tries to maintain more stable service over the full runtime by combining adaptive mode detection, media-aware request selection, batching, and merge-aware behavior.

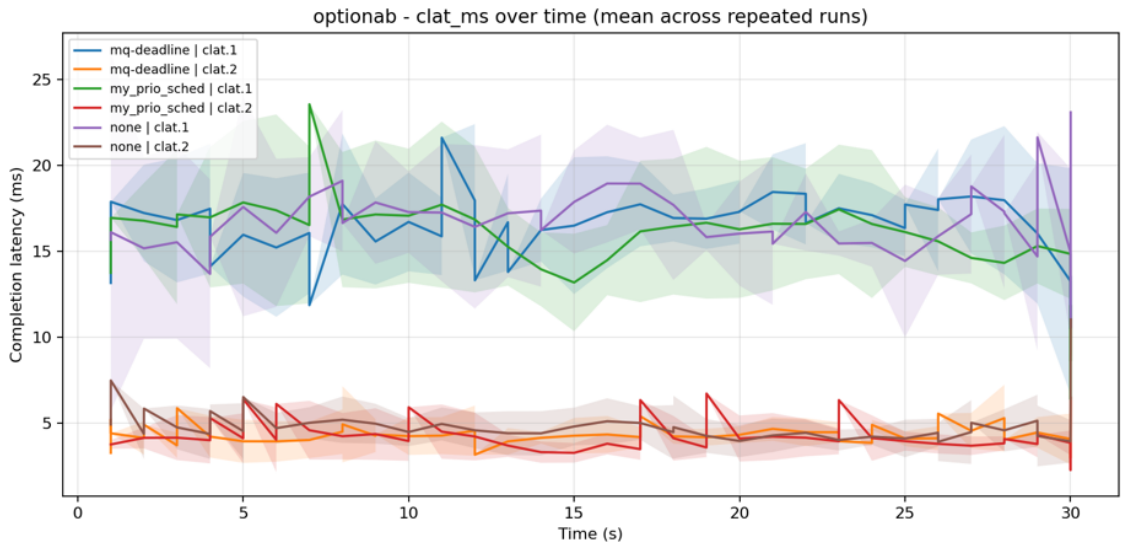

Figure 8: Latency time series for the advanced scheduler group. The advanced scheduler shows more stable behavior after adaptive and media-aware optimizations.

5.5 Immediate Observations

At a glance, four patterns stand out from the experimental results.

1. **The basic scheduler has limited performance benefit.** The basic `my_prio_sched` implements the required priority queues and starvation prevention, but it does not consistently outperform the baselines in either IOPS or P99 latency.

2. **The advanced scheduler improves relative performance.** After adding adaptive and media-aware mechanisms, `my_prio_sched` becomes much more competitive. In the advanced group, it achieves the best or near-best IOPS and P99 latency for the tested jobs.

3. **CPU overhead remains controlled.** CPU time is similar across `none`, `mq-deadline`, and `my_prio_sched`. This indicates that the additional scheduling logic does not introduce a large CPU penalty in the tested environment.

4. **Run-to-run variability is visible.** The error bars and time-series fluctuations show that single-run results are not reliable enough. Repeated runs and averaged results are necessary for this virtualized experimental environment.

These observations define the main interpretation of the experiment: the basic scheduler is a correct functional baseline, while the advanced scheduler better captures workload-aware and media-aware scheduling behavior.

6 In-Depth Analysis

6.1 Overall Assessment

The results appear reasonable and internally consistent. They do not look like a failed script run or a broken scheduler implementation. Instead, they show a typical systems trade-off: the basic scheduler is simple and interpretable, but its performance gains are limited; the advanced scheduler introduces additional system awareness and therefore becomes more effective.

The basic scheduler demonstrates the minimum mechanism required for priority-aware I/O scheduling. It classifies requests according to `ioprio`, inserts them into HIGH, NORMAL, and

LOW FIFO queues, dispatches higher-priority requests first, and uses aging to avoid starvation. This design is correct as a baseline, but it only controls software-queue dispatch order.

The advanced scheduler extends this design by asking two additional questions: whether the current workload looks sequential or random, and whether the device should be handled using an HDD-like or SSD-like dispatch policy. The experimental results show that these additional signals help the scheduler organize requests more effectively.

6.2 Why the Basic Scheduler Does Not Show Strong Gains

The basic scheduler does not show strong gains because it is intentionally simple. It expresses priority, but it does not use other information that is important for block I/O performance.

First, it does not consider address locality. The `backup_sim` workload is sequential and may benefit from continuous request organization. A strict-priority scheduler can interrupt this locality if high-priority random requests repeatedly preempt the sequential stream.

Second, it does not distinguish between HDD-like and SSD-like behavior. HDDs benefit from sector ordering because seek cost matters, while SSDs benefit more from low overhead and sufficient queue depth. The basic scheduler uses the same policy for both cases.

Third, it does not actively enhance merging. Linux already performs some generic bio/request merging, but the basic scheduler does not add scheduler-level merge callbacks. Therefore, it cannot selectively increase merge opportunities for sequential workloads.

Fourth, it does not control in-flight queue depth. Once requests are dispatched below the scheduler, the basic scheduler cannot preempt them or reorder their completion. As a result, strict priority at dispatch time does not always translate into lower end-to-end latency.

Finally, its aging mechanism is a starvation-prevention mechanism rather than a fine-grained tail-latency control mechanism. Aging ensures low-priority progress, but it does not optimize P99 latency. These limitations explain why the basic scheduler is functionally correct but performance-limited.

6.3 Why the Advanced Scheduler Improves Relative Performance

The advanced scheduler improves performance because it turns the scheduler from a pure priority queue into a workload-aware and media-aware policy.

6.3.1 Adaptive Strategy

The advanced scheduler uses request sector gaps to infer whether the workload is sequential or random. In sequential mode, it enables batching and more active merge behavior. In random mode, it resets batching and avoids extending sequential assumptions.

This adaptive behavior helps explain the improved IOPS and P99 latency in the advanced group. Instead of applying the same strict-priority decision repeatedly, the scheduler changes its behavior based on observed access patterns.

6.3.2 Media-Aware Behavior

The advanced scheduler also distinguishes HDD-like and SSD-like dispatch paths. On the HDD path, it uses a red-black tree to select requests according to sector order, while also applying an expiration guard to prevent FIFO head requests from waiting too long. On the SSD path, it selects the FIFO head directly to avoid unnecessary sorting overhead.

This design better matches storage-device behavior. HDDs benefit from locality-aware dispatch, while SSDs benefit from a simpler path that avoids extra software reordering when address locality is less important.

6.3.3 Merge-Aware Behavior

The advanced scheduler also adds merge-aware callbacks. It allows only same-priority request/bio merges, which preserves priority semantics. In sequential mode, it enables front-merge lookup and bio-level merge attempts. After merging, it updates the red-black tree and FIFO timestamp state so that the scheduler's private data structures remain consistent.

This merge-aware behavior is conservative rather than fully aggressive. The scheduler does not introduce a long waiting window purely for merge. However, even this conservative form of merge awareness helps improve request organization in sequential workloads.

6.4 Why High-Priority Latency Is Not Always Absolutely Lower

A simple expectation is that high-priority work should always show lower latency than low-priority work. In this experiment, however, `database_sim` often has higher absolute latency than `backup_sim`. This does not contradict the design goal because the two jobs have different I/O patterns.

`database_sim` uses 4KB random reads. Such requests are small, fragmented, and sensitive to transient queue state. `backup_sim` uses 128KB sequential reads, which are more continuous and can benefit from locality and merging. Therefore, the database workload can naturally have higher completion latency even though it has higher priority.

The correct way to evaluate priority effectiveness is to compare the same workload across different schedulers. Under this interpretation, the advanced scheduler shows a positive effect: it improves `database_sim` IOPS and P99 latency relative to the baselines while keeping `backup_sim` throughput non-zero.

6.5 Why CPU Overhead Does Not Increase Much

Although the advanced scheduler adds more logic, the measured CPU time does not increase significantly. This suggests that the additional mechanisms are lightweight enough for the tested workload.

There are several possible reasons. First, the fio workload is I/O-bound, so I/O waiting dominates execution time. Second, mode detection and batch-state updates are simple operations. Third, red-black tree and merge callbacks are only exercised when relevant requests enter the scheduler. Fourth, the VirtualBox environment may hide some small kernel-side CPU differences behind virtualization overhead.

Therefore, the CPU overhead results support the claim that the advanced scheduler improves request organization without imposing a large measurable CPU cost.

6.6 Why Error Bars Matter

The error bars are important because the experiment shows visible run-to-run variability. Earlier single-run experiments produced unstable scheduler rankings. This is expected in a virtualized environment with random I/O, mixed workloads, and host-side interference.

Therefore, this report avoids drawing strong conclusions from any single run. Instead, we focus on repeated-run averages and the overall trend. If two schedulers have close means and overlapping error bars, it is safer to describe them as comparable rather than declaring one to be universally better.

The advanced group is meaningful because `my_prio_sched` shows a consistent positive trend across multiple metrics: it improves relative IOPS and P99 latency without a large CPU penalty.

6.7 Comparison with `mq-deadline`

`mq-deadline` is a mature Linux blk-mq scheduler. It combines request ordering with deadline-like waiting control and is designed to balance throughput, latency, and fairness. Therefore, it is expected to perform well, especially in the basic group where our custom scheduler only applies simple priority queues.

The advanced scheduler becomes more competitive because it adds some of the missing system awareness: workload-mode detection, media-aware request selection, batching, and merge-aware callbacks. In the advanced group, `my_prio_sched` can match or outperform `mq-deadline` on some IOPS and P99 latency metrics.

However, this does not mean that the custom scheduler is universally better than `mq-deadline`. `mq-deadline` is a mature general-purpose scheduler that handles many edge cases. Our scheduler is a course project prototype focused on priority-aware behavior and two targeted optimizations. The fair conclusion is that the advanced version becomes competitive in the tested workload, not that it fully replaces `mq-deadline`.

6.8 Final Evaluation of the Experimental Results

Taken together, the experimental results support the following evaluation.

1. **The basic scheduler is functionally correct but performance-limited.** It implements priority queues and aging, but it does not consistently outperform the baselines.
2. **The advanced scheduler improves relative performance.** With adaptive and media-aware policies, `my_prio_sched` achieves stronger IOPS and P99 latency results.
3. **Low-priority progress is preserved.** `backup_sim` maintains non-zero IOPS in both groups, indicating that starvation is avoided.
4. **CPU overhead is acceptable.** The custom scheduler's CPU time remains close to `mq-deadline` and `none`.
5. **Results should be interpreted by relative trends.** Because the two jobs have different request sizes and access patterns, and because the experiment runs in a virtualized environment, the most meaningful comparisons are within the same job across different schedulers.

Overall, the experiment shows a clear design progression. The basic scheduler provides a correct priority-aware baseline. The advanced scheduler improves this baseline by adding adaptive strategy, media-aware dispatch, and merge-aware behavior. These optimizations make the custom scheduler more effective for the mixed workload while preserving low-priority progress and keeping CPU overhead controlled.

7 Discussion

7.1 Main value of the project

The main value of this project is not simply whether one bar is higher than another. The project connects Linux block-layer theory with an actual kernel implementation. It shows how to register a blk-mq scheduler, maintain scheduler-private state through `elevator_data`, classify requests by I/O priority, implement aging, extend dispatch logic, participate in request merging, and evaluate behavior with fio.

8 Limitations and Future Work

8.1 Limitations

1. The experiments were conducted in a virtualized setup, which increases noise and may affect scheduler ranking.
2. The advanced sequential/random detector is global rather than per-flow, so mixed workloads can pollute the mode decision.
3. The merge-aware logic is conservative. It participates in Linux's merge paths but does not introduce a strong explicit aggregation window.
4. The database and backup jobs have different access patterns, so absolute latency between the two jobs should not be compared directly.

8.2 Future work

Future extensions could include per-priority token buckets, explicit in-flight depth control, per-flow adaptive detection, stronger sequential aggregation windows, evaluation on bare-metal Linux, and comparison across real SSD and HDD devices. The scheduler could also adopt fair-queueing ideas such as deficit round robin [4] or QoS-oriented mechanisms such as reservation and limit controls [3].

9 Conclusion

This project implements and evaluates a custom blk-mq I/O scheduler for Topic 9. The basic scheduler demonstrates the core idea of priority-aware dispatch through three FIFO queues and aging. The advanced scheduler improves the design with adaptive workload detection, media-aware request selection, batching, and merge-aware callbacks. The final repeated-run results show that the basic scheduler is limited, while the advanced version improves IOPS and P99 latency and maintains CPU overhead close to existing schedulers. The results also show that storage scheduling is highly workload- and environment-dependent.

References

[1] Bjørling, M., Axboe, J., Nellans, D., and Bonnet, P. (2013). *Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems*. Proceedings of the 6th International Systems and Storage Conference (SYSTOR). https://kernel.dk/blk-mq.pdf

[2] Corbet, J. (2018). *I/O scheduling for single-queue devices*. LWN.net. https://lwn.net/Articles/767987/

[3] Gulati, A., Merchant, A., and Varman, P. J. (2010). *mClock: Handling Throughput Variability for Hypervisor IO Scheduling*. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI). https://www.usenix.org/conference/osdi10/mclock-handling-throughput-variability-hypervisor-io-scheduling

[4] Shreedhar, M. and Varghese, G. (1995). *Efficient Fair Queuing Using Deficit Round-Robin*. Proceedings of ACM SIGCOMM. https://dl.acm.org/doi/10.1145/217391.217453

[5] Worthington, B. L., Ganger, G. R., and Patt, Y. N. (1994). *Scheduling Algorithms for Modern Disk Drives*. Proceedings of ACM SIGMETRICS, pp. 241–251. https://dl.acm.org/doi/10.1145/183018.183045