Implement and Evaluate a Custom Disk Scheduler

Jiang Xinhang
Team 18, Student ID: 225040179

Yang Kefan
Team 18, Student ID: 225040525

***Abstract*—This report presents `mq-prio`, a custom priority-aware I/O scheduler implemented in the Linux blk-mq layer. The required scheduler should preferentially serve high-priority workloads, such as foreground database requests, over low-priority workloads, such as background backup traffic, while still preserving low-priority progress. `mq-prio` uses multi-level FIFO queues, Linux request priority metadata, strict priority dispatch, and aging-based starvation control. fio mixed-workload experiments compare `mq-prio` with `mq-deadline` and `none` using IOPS, completion latency, and CPU overhead. The results show stronger priority separation and better high-priority mean service performance, with a remaining limitation in high-priority p99 latency.**

***Index Terms*—Linux kernel, blk-mq, I/O scheduler, priority scheduling, fio**

I. Introduction

The project requires a new I/O scheduling algorithm inside the Linux blk-mq layer. The scheduler must implement priority-aware behavior, allow high-priority tasks to preempt low-priority tasks under contention, and be evaluated using fio mixed workloads. The required metrics are throughput, completion latency, and CPU overhead.

The implemented scheduler is named `mq-prio`. Its design follows the required implementation path: existing blk-mq [2] schedulers, especially `mq-deadline`, are used as engineering references; request priority is extracted from kernel I/O priority metadata; dispatch order favors urgent traffic; and aging prevents starvation. The evaluation compares `mq-prio` with `mq-deadline` and the `none` scheduler to separate policy effects from baseline block-layer behavior.

II. Task Context and Design Rationale

A. Position in the I/O Stack

The scheduler is implemented only in the Linux block layer. Application behavior, filesystem logic, hardware queue mapping, and storage driver code are not changed. This boundary matches the blk-mq model, where the I/O scheduler operates between software submission queues and hardware dispatch queues.

B. Storage Considerations

Disk scheduling is necessary because storage latency is much larger than CPU and memory latency. A block scheduler can reduce this gap through batching, reordering, and merging. The value of each policy depends on the device. On HDDs, seek reduction is central because mechanical movement dominates latency. On SSDs, address sorting is less important, while queue depth and internal parallelism become more relevant. The present implementation focuses on priority control rather than device-specific seek optimization.

C. blk-mq Request Metadata

The blk-mq request structure exposes the fields needed by a scheduler [4], including `queuelist`, `ioprio`, request sector, and data length. In this implementation, `ioprio` is the key field because it encodes Linux I/O priority information. `mq-prio` uses this metadata to place requests into priority-specific queues instead of relying on process names or workload-specific identifiers.

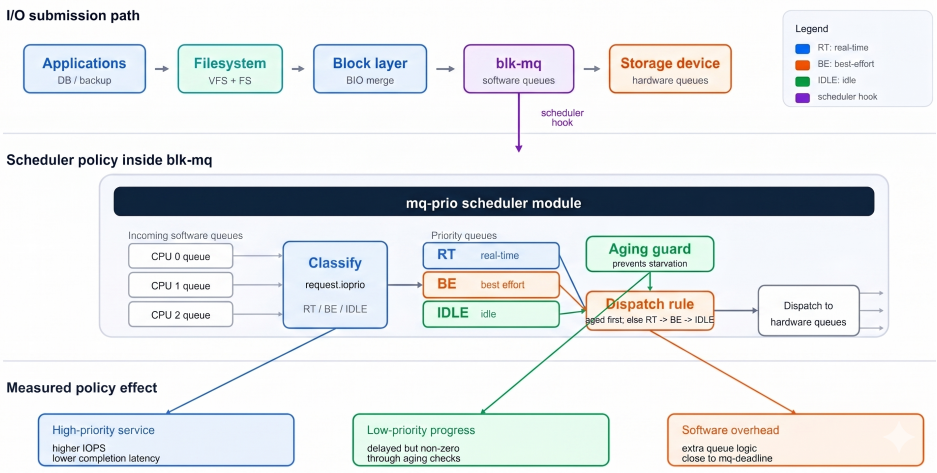


Fig. 1: Placement of `mq-prio` in the blk-mq I/O path. Requests are classified by priority before dispatch to hardware queues.

III. Implementation

A. Multi-Level Queue Design

`mq-prio` implements a multi-level queue scheduler. The internal policy separates requests into real-time, best-effort, and idle classes. Each class is stored as a FIFO list, preserving arrival order within the same priority level. This structure directly supports the required behavior that high-priority traffic should be served before low-priority traffic under contention.

B. Priority Classification

During insertion, each request is removed from the incoming list, mapped from Linux I/O priority information to an internal priority class, timestamped with `jiffies`, and appended to the corresponding FIFO queue. The timestamp is used later by the aging path. This design keeps priority handling inside the scheduler and avoids changes to upper layers.

Algorithm 1 `mq-prio` Insert Policy

1: **for** each request rq in input list **do**
2: remove rq from input list
3: $prio \leftarrow$ map `rq.ioprio` to {RT, BE, IDLE}
4: $rq.fifo\_time \leftarrow jiffies$
5: append rq to queue[$prio$]
6: **end for**

C. Dispatch and Aging

The dispatch policy is organized as a two-stage procedure for balancing priority preference and starvation control. When a request is inserted, it records an enqueue timestamp. During dispatch, the scheduler first checks whether any lower-priority request has waited longer than the configured starvation threshold. If such an expired request exists, it is selected before the normal strict-priority pass. This aging step acts as a bounded exception to priority ordering and ensures that low-priority traffic can still make forward progress under sustained high-priority load.

If no aged request is found, `mq-prio` applies strict priority dispatch. The scheduler scans the priority queues in RT, BE, and IDLE order, and dispatches the first request from the highest non-empty queue. Requests within the same priority class remain FIFO-ordered, so the scheduler preserves arrival order among requests with equal priority. This policy makes high-priority service the default behavior while using aging to avoid starvation of background work. As a result, `mq-prio` provides stronger priority differentiation than a general-purpose scheduler, but does not completely block low-priority requests during long periods of contention [1].

D. Scheduler Lifecycle

`mq-prio` is registered as a complete blk-mq elevator scheduler. Its operation table binds request insertion, request dispatch, request preparation, request finishing, scheduler initialization, scheduler exit, and module registration callbacks.

TABLE I: Dispatch Policy Summary

Stage	Action
1	Check lower-priority queues for aged requests and dispatch an expired request if present.
2	If no aged request exists, dispatch the first request from the highest non-empty queue.
3	Return no request only when all priority queues are empty.

The initialization path allocates the elevator object, allocates scheduler-private data, binds `elevator_data`, sets the scheduler flag, and attaches the elevator to the request queue. This lifecycle work is necessary for stable scheduler switching and repeatable fio evaluation.

IV. Evaluation Methodology

The evaluation follows a mixed-workload fio strategy. The high-priority job models latency-sensitive foreground database traffic with 4 KiB random reads and four jobs. The low-priority job models background backup activity with 128 KiB sequential reads and two jobs. This setup intentionally emulates foreground and background contention rather than isolating priority as the only variable. The workload uses Linux I/O priority values to distinguish the two jobs, and the same workload is executed under all compared schedulers.

TABLE II: fio Mixed-Workload Configuration

Job	Priority	Pattern	Jobs
`database-sim`	High, 1	4 KiB random read	4
`backup-sim`	Low, 7	128 KiB sequential read	2

The test environment is a Linux 6.5.0-18 virtual machine using fio 3.28. Each collected run lasts 15 seconds. The baselines are `mq-deadline`, representing a mature blk-mq scheduler, and `none`, representing direct dispatch without a scheduler policy. The measured metrics are read IOPS, mean completion latency, p99 completion latency, and CPU overhead. CPU overhead is reported as user CPU percentage plus system CPU percentage.

V. Results

A. Throughput

Fig. 2 shows that the high-priority workload achieves higher IOPS than the low-priority workload under all three schedulers. This pattern is expected because the high-priority job uses smaller 4 KiB requests and higher parallelism, but the degree of separation differs across schedulers. `mq-prio` gives the high-priority workload the highest IOPS among the tested schedulers: `database-sim` reaches 59,568.4 IOPS, which is 11.7% higher than `mq-deadline` and 12.5% higher than `none`.

The low-priority workload receives less throughput under `mq-prio` than under the two baselines. This is a direct

consequence of the policy: high-priority requests are more likely to be dispatched earlier, while low-priority requests are delayed unless the aging condition is reached. The important result is that low-priority throughput remains non-zero at 42,221.2 IOPS, indicating that the scheduler prefers urgent requests without completely starving background work.

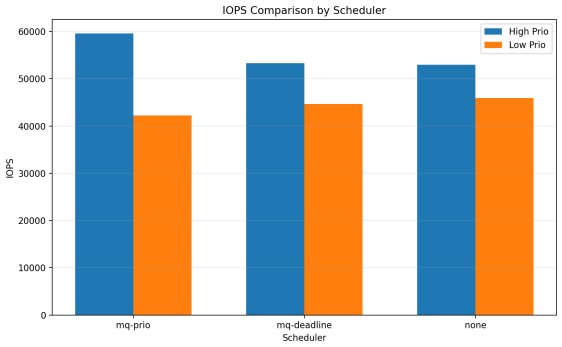


Fig. 2: Read IOPS under the mixed-priority fio workload.

B. Completion Latency

Fig. 3 first compares the aggregated completion latency of the high-priority and low-priority workloads under the three schedulers. Across all schedulers, the high-priority workload has lower completion latency than the low-priority workload. This is broadly consistent with the IOPS result: higher IOPS usually corresponds to lower completion latency, although the relationship is not strictly proportional because request size, access pattern, queue state, and device behavior also affect latency.

Among the three schedulers, `mq-prio` gives the lowest high-priority completion latency and the largest gap between high- and low-priority workloads. This indicates the strongest priority differentiation. `mq-deadline` also separates the two priority classes, but the gap is smaller. The `none` scheduler shows the weakest separation, which is expected because it adds little scheduling logic and relies more directly on the default blk-mq path and the device.

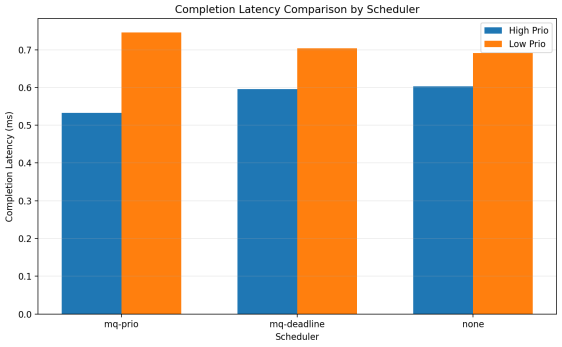


Fig. 3: Completion latency comparison by scheduler.

Fig. 4 further shows how completion latency changes over time. Under `mq-prio`, the high-priority curve is consistently below the low-priority curve for most of the run. The gap is persistent rather than limited to a few isolated points, which shows that `mq-prio` sustains priority differentiation over time.

The `mq-prio` high-priority curve fluctuates more strongly than the low-priority curve. This behavior is reasonable because high-priority requests are preferentially dispatched; once such a request receives service, it may complete quickly and create low-latency points. At the same time, the high-priority workload uses 4 KiB random reads, which are fragmented and sensitive to transient queue conditions, so its latency is naturally more variable. The sudden jump at the final point should be treated as boundary sampling noise, likely caused by an incomplete final `log_avg_msec` window, rather than as a representative trend.

The time-series plots for `mq-deadline` and `none` show the same general ordering: high-priority latency is usually lower than low-priority latency. However, their separations are less policy-driven than `mq-prio`. `mq-deadline` shows a more conservative and balanced priority effect, while `none` shows the weakest scheduling-policy distinction because it performs little additional I/O scheduling. The combined plot confirms this pattern across all schedulers.

The required behavior is that high-priority work should obtain high IOPS and low latency, while low-priority work should still complete. `mq-prio` provides the strongest separation on both derived measures. Its high-to-low IOPS ratio is 1.41, compared with 1.19 for `mq-deadline` and 1.15 for `none`. Its low-to-high mean-latency ratio is 1.40, compared with 1.18 and 1.15 for the baselines. These ratios show that `mq-prio` makes the priority policy more visible in measured behavior, whereas `mq-deadline` and `none` treat the two workloads more uniformly under contention.

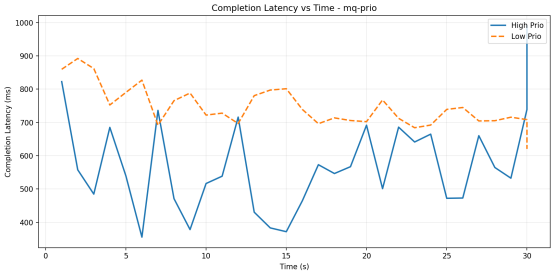
TABLE III: Derived Priority-Separation Metrics

Scheduler	IOPS Ratio	Latency Ratio	Avg. CPU
`mq-prio`	1.41	1.40	15.33%
`mq-deadline`	1.19	1.18	15.49%
`none`	1.15	1.15	11.66%

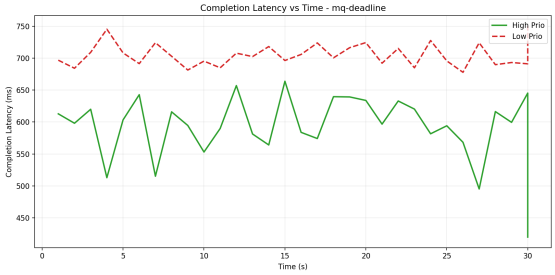
C. CPU Overhead

Fig. 5 reports the software-side cost of each scheduler. Across all three schedulers, the low-priority job has higher CPU overhead than the corresponding high-priority job in this experiment. This should be interpreted as a property of the selected mixed workload, not as a general claim that low-priority workloads are inherently more CPU-intensive.

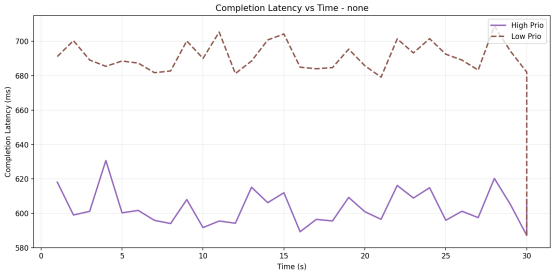
The `none` scheduler has the lowest average CPU overhead at 11.66%, which is expected because it adds almost no scheduling policy. `mq-prio` has an average CPU overhead of 15.33%, close to `mq-deadline` at 15.49%. This indicates that priority classification, multiple FIFO queues, locking, and aging checks introduce measurable work, but the cost remains comparable to an existing Linux scheduler. The result supports



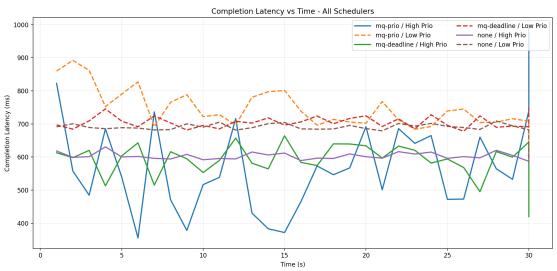
(a) mq-prio



(b) mq-deadline



(c) none



(d) all schedulers

Fig. 4: Completion latency over time under mixed-priority workloads.

a practical trade-off: `mq-prio` spends additional CPU cycles to obtain clearer priority control.

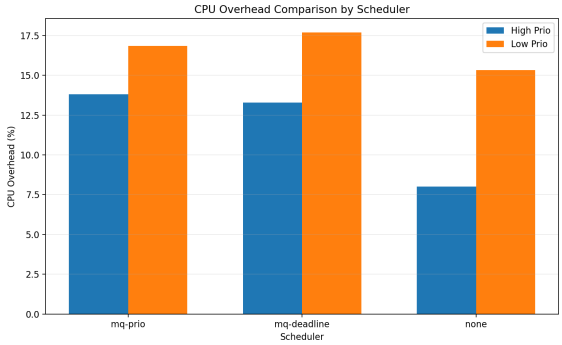


Fig. 5: CPU overhead reported by fio.

TABLE IV: Raw fio Results

Sched.	P	IOPS	Mean	P99	CPU
`mq-prio`	H	59,568.4	0.532	2.343	13.81
`mq-prio`	L	42,221.2	0.746	1.892	16.85
`mq-deadline`	H	53,309.2	0.595	1.843	13.29
`mq-deadline`	L	44,693.6	0.703	1.499	17.69
`none`	H	52,950.1	0.603	1.319	8.00
`none`	L	45,893.9	0.691	1.286	15.33

* H and L denote high and low priority. Mean and p99 are in milliseconds; CPU is in percent.

VI. Discussion

The results match the required objective of priority-aware disk scheduling. `mq-prio` favors the database-like workload under contention and keeps backup-like traffic non-zero through aging. Compared with `mq-deadline`, it provides clearer priority semantics. Compared with `none`, it adds policy control at the cost of additional scheduling work.

The current evaluation is limited by the virtualized storage environment and by a single mixed-workload configuration. Since the experiments were conducted inside a virtual machine, the underlying physical storage characteristics were abstracted by the virtualization layer. Therefore, the evaluation could not reliably compare HDD- and SSD-specific behavior, such as seek-sensitive scheduling on HDDs or queue-depth effects on SSDs.

A direct SSD/HDD comparison was not completed because the experiments were conducted in a virtualized environment. The virtual block device hides many physical storage characteristics, such as HDD seek time, rotational delay, SSD internal parallelism, controller-level scheduling, and real queue-depth behavior. As a result, the measured results mainly reflect the guest kernel scheduler, the virtual block layer, and the host-side storage abstraction, rather than the true behavior of a specific physical HDD or SSD.

A physical SSD/HDD comparison was not completed because the experiments were conducted in a virtualized environment [3]. The virtual block device hides key device characteristics, such as HDD seek and rotational delay, SSD internal parallelism, and real queue-depth behavior. On HDDs, priority dispatch may interact with seek-oriented reordering, while on SSDs it may mainly affect latency under parallel queueing. Future work should evaluate `mq-prio` on physical HDD and SSD devices with longer fio runs and per-window latency traces.

VII. Conclusion

`mq-prio` implements a complete priority-aware I/O scheduler inside the Linux blk-mq layer. The design combines multi-level FIFO queues, Linux request priority metadata, strict priority-based dispatch, and aging-based starvation control. This structure allows high-priority requests to receive earlier service while still preserving non-zero progress for low-priority traffic under contention.

The fio evaluation shows that `mq-prio` provides the clearest priority differentiation among the tested schedulers. Compared with `mq-deadline` and `none`, it achieves the highest IOPS and the lowest mean completion latency for the high-priority workload. The measured latency gap between high- and low-priority traffic also confirms that the scheduler policy is visible in end-to-end workload behavior, rather than only in the internal dispatch logic. At the same time, its CPU overhead remains close to `mq-deadline`, indicating that the added priority queues and aging checks introduce only moderate scheduling cost.

The main limitation is tail-latency stability. Although `mq-prio` improves high-priority throughput and mean latency, its high-priority p99 latency is higher than the baselines in the current experiment. This suggests that strict priority dispatch may interact with transient queue states, small random reads, and virtualized storage behavior. Further work should tune the aging threshold, evaluate longer runs, and validate the scheduler on physical SSD and HDD devices to better characterize device-dependent behavior and tail latency.

Acknowledgment

Jiang Xinhang and Yang Kefan contributed equally to the implementation, evaluation, and result analysis. The authors thank the course instructor, teaching assistants, and classmates for their guidance, feedback, and helpful discussions throughout the project.

References

[1] K. Doekemeijer, Z. Ren, T. De Matteis, B. Chandrasekaran, and A. Trivedi, "Does Linux provide performance isolation for NVMe SSDs? Configuring cgroups for I/O control in the NVMe era," in *Proc. IEEE International Symposium on Workload Characterization (IISWC)*, Irvine, CA, USA, 2025, pp. 353–367. [Online]. Available: https://doi.org/10.1109/IISWC66894.2025.00037

[2] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block IO: introducing multi-queue SSD access on multi-core systems," in *Proc. 6th International Systems and Storage Conference (SYSTOR)*, Haifa, Israel, 2013, Art. no. 22. [Online]. Available: https://doi.org/10.1145/2485732.2485740

[3] T. Heo, D. Schatzberg, A. Newell, S. Liu, S. Dhakshinamurthy, I. Narayanan, J. Bacik, C. Mason, C. Tang, and D. Skarlatos, "IOCost: block IO control for containers in datacenters," in *Proc. 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, 2022, pp. 595–608. [Online]. Available: https://doi.org/10.1145/3503222.3507727

[4] Z. Ren, K. Doekemeijer, N. Tehrany, and A. Trivedi, "BFQ, Multiqueue-Deadline, or Kyber? Performance characterization of Linux storage schedulers in the NVMe era," in *Proc. 15th ACM/SPEC International Conference on Performance Engineering (ICPE)*, London, United Kingdom, 2024, pp. 154–165. [Online]. Available: https://doi.org/10.1145/3629526.3645053