JOURNAL OF SCIENCE
SAKARYA UNIVERSITY

Sakarya University Journal of Science

ISSN 1301-4048 | e-ISSN 2147-835X | Period Bimonthly | Founded: 1997 | Publisher Sakarya University |

http://www.saujs.sakarya.edu.tr/

Title: Performance comparison and analysis of Linux block I/O schedulers on SSD

Authors: Yunus Ozen, Abdullah Yildirim

Recieved: 2018-11-01 15:03:41

Revised: 2018-11-11 23:53:28

Accepted: 2018-12-13 09:51:56

Article Type: Research Article

Volume: 23

Issue: 1

Month: February

Year: 2019

Pages: 106-112

How to cite
Yunus Ozen, Abdullah Yildirim; (2019), Performance comparison and analysis of Linux block I/O schedulers on SSD. Sakarya University Journal of Science, 23(1), 106-112, DOI: 10.16984/saufenbilder.477446
Access link
http://www.saujs.sakarya.edu.tr/issue/38708/477446

Sakarya University Journal of Science 23(1), 106-112, 2019

Performance comparison and analysis of Linux block I/O schedulers on SSD

Yunus Ozen*[1], Abdullah Yildirim[1]

ABSTRACT

A computer system's one of the slowest operation is disk seek operation. Sending out read and write requests to the block devices such as disks as soon as the request arrives results in poor performance. After performing sorting and merging operations, the operating system kernel issues block I/O requests to a disk for improving the overall system performance. The kernel subsystem to perform scheduling the block I/O requests is named as the I/O scheduler. This paper introduces performance comparison and detailed analyses of Deadline, CFQ, Noop and BFQ block I/O schedulers that are contained in the Linux 4.1x kernel. The tests have been carried out on an SSD block device that is common in hardware combinations of both personal and professional use-case scenarios. The performance of the schedulers has been evaluated in terms of throughput. Each scheduler has advantages in different use-case scenarios and provides better throughput in a suitable environment.

Keywords: Block I/O Scheduler, Deadline, Noop, CFQ, BFQ.

1. INTRODUCTION

Block devices such as hard drives or flash memories run in a random access fashion to write or read fixed-size pieces of data. That data is named as a block. Whenever a piece of data is requested for a block device, the read/write head seeks from a position to another position. This seeking operation is a slow operation. Since block devices are performance-sensitive the kernel has a dedicated sub-system called block I/O layer to optimize seeking operations. The main motivation of the Linux kernel version 2.5 development was to optimize the block I / O layer. The bio struct proposed with version 2.5 in addition to the bufferhead struct is still an essential part of the modern Linux kernel [1]. The bio struct is the basic container for block I/O requests in the Linux kernel. It stores the active block I/O operations as a list of segments. A segment represents a bunch of buffers that are contiguous in memory. The bio struct provides flexibility to perform multiple block I/O operations with its segments based approach.

Block devices have request queues to schedule pending read or write requests. The kernel subsystem to perform scheduling the block I/O requests is called the I/O scheduler. Sending requests to the block device immediately ends up in poor performance. The scheduler organizes the request order in the queue and dispatching time to the block device. The main objective is reducing seeks to improve overall throughput of

* Corresponding Author: yunus@yunus.gen.tr

[1] Yalova University, Computer Engineering, Yalova, Turkey

[2] Yalova University, Computer Engineering, Yalova, Turkey

the system. The kernel issues the requests to the device after performing some merging and sorting operations in this purpose. Merging is the bundling of two or more requests into a single request. Merging the requests reduces overhead while decreases the seek operations. The whole request queue is kept sorted according to sector positions. The purpose in sorting is minimizing the number of seeks by keeping the disk head moving into the same direction [2].

Linus Elevator was the first I/O scheduler in the Linux. It performs both merging and sorting operations to optimize the number of seeking. The request is added to the tail of the request queue, if a suitable location is not found for a request to merge [3]. If an existing request is older than a threshold, the new request is added to the tail of the queue. That is not efficient but prevents several requests to be starved. This improves latency but causes to request starvation.

Several schedulers such as Deadline, CFQ, and Noop are introduced after version 2.6 to overcome this starvation problem. The main motivation of those earlier schedulers was reducing the number of seek operation on rotational magnetic block devices such as hard drives [1].

Rotational magnetic block devices have been replaced by solid state drives (SSDs) recently in areas ranging from smart devices to large data center implementations. SSDs have some advantages over traditional HDDs in terms of throughput, reliability, and energy consumption. They are free from the latency of the seeking time of rotational magnetic disks. However, existing I/O hardware and schedulers have been designed and optimized for rotational magnetic disk specifications [4].

The SSDs are getting research interest for their potential to make appropriate optimizations with a new motivation [5].

The literature already has some recent studies that modify queue structures of existing state of the art schedulers and exploit the internal parallelism of SSDs. FlashFQ [6] analyzes request size to estimate the response time and uses the start-time fair queueing to provide fairness among concurrent tasks on SSDs. Gao et al. [7] proposed a scheduler called PIQ for minimizing the access conflicts among the I/O requests in one batch. External mergesort is a common sorting algorithm to sort large amounts of data. FMsort focuses on enhancing the merge phase of external mergesort for SSDs [8]. Several studies have been proposed to take advantage of the internal parallelism of SSDs to improve performance [9]. Mao et al. designed a new I/O scheduler called Amphibian by utilizing internal parallelism of SSDs. Amphibian performs size-based request ordering to prioritize requests with small sizes [10]. Chen et al. performed experiments to show the results of optimization based on internal parallelism for the performance improvement [11]. Guo et al. [12] proposed a scheduler called SBIOS considering full use of read internal parallelism and avoiding the block cross penalty. Most of the studies in the literature focus on existing complex schedulers for adapting them to hardware opportunities of SSDs. Revisiting state of the art schedulers in terms of throughput and highlighting the potential for a simple scheduler is needed.

The focus of our study is to make a comparison between the state of the art I/O schedulers in the Linux kernel in terms of throughput.

The rest of the paper is organized as follows. The schedulers Deadline, CFQ, Noop, and BFQ block I/O are presented in Section 2. Benchmark setup, experimental platform details, performance metrics, and preferred workloads are described in Section 3. Performance evaluation is presented in Section 4. We finally conclude this paper in Section 5.

2. I/O SCHEDULERS

The I/O scheduler merges and sorts the pending block I/O requests into the request queues and sends them to the system. This section briefly describes the Deadline, CFQ, Noop, and BFQ block I/O schedulers that are compared in terms of throughput and analyzed in this paper. Deadline, CFQ, Noop and BFQ block I/O schedulers are chosen, because they are contained in most of the Linux distributions with the 4.1x kernel.

2.1. The Deadline I/O Scheduler

The Deadline scheduler is one of the earliest schedulers that focus on the starvation problem of the Linus Elevator. Linus Elevator targets merging I/O requests to a specific portion of the disk and this causes starvation of the requests to another portion of the disk. The read requests are generally dependent on each other because of data locality. The scheduler merges them to minimize the seek operation and this causes starvation. There is a tradeoff between minimizing seeks and preventing starvation. The Deadline scheduler contains a request queue that is sorted sectorwise on disk and merged like Linus Elevator. The Deadline scheduler inserts the request into another queue according to the type of request. Write requests are inserted into a write FIFO queue and read requests are inserted into a read FIFO queue. Deadline scheduler maintains a balance to make these operations fair with its multi-queue structure. It gives smaller expiration value to the read requests than write requests to prevent write requests starving read requests. The simplified diagram of deadline scheduler is shown in Figure 1.

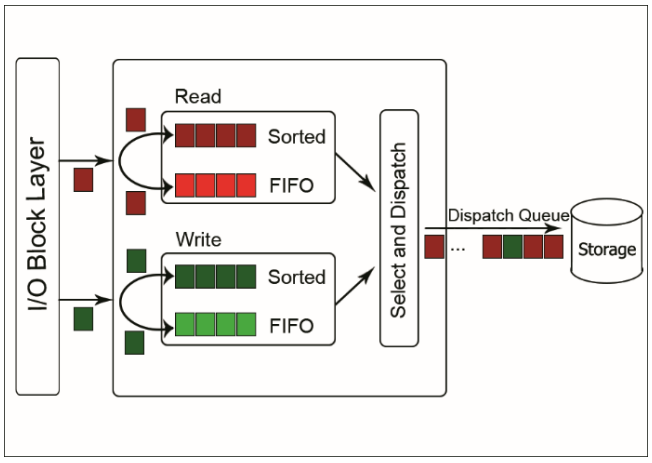

Figure 1. Deadline scheduler.

2.2. The Complete Fair Queuing I/O Scheduler (CFQ)

The CFQ scheduler organizes incoming I/O requests to the queues based on the processes. The newly submitted I/O request is combined with neighboring requests and insertion sorted sectorwise in every queue. The CFQ scheduler differs from other schedulers with its per-process queues. The round-robin structure lets a number of requests to be dispatched before continuing on to the next one. Each process gains an equal slice of disk bandwidth and this algorithm provides fairness at a per-process fashion. The simplified diagram of CFQ scheduler is shown in Figure 2.

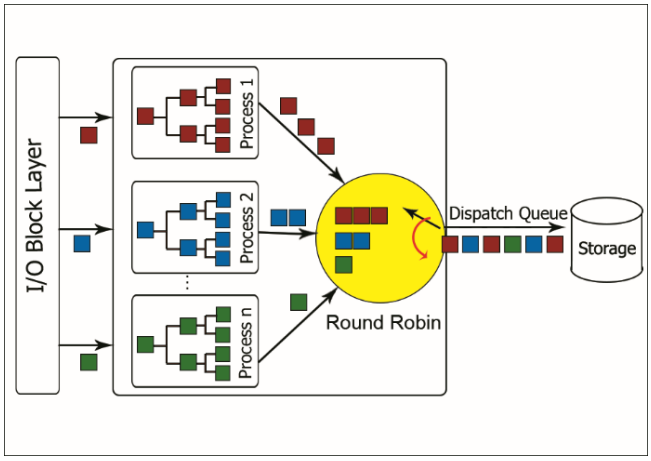

Figure 2. CFQ scheduler

2.3. The Noop I/O Scheduler

The Noop scheduler does not sort requests before inserting to the queue. It merges the new request to the adjacent request and maintains a single request queue in a near-FIFO order. It is said to be the first I/O scheduler that targets the block devices such as flash memories that run in a completely random-access fashion. The simplified diagram of Noop scheduler is shown in Figure 3.

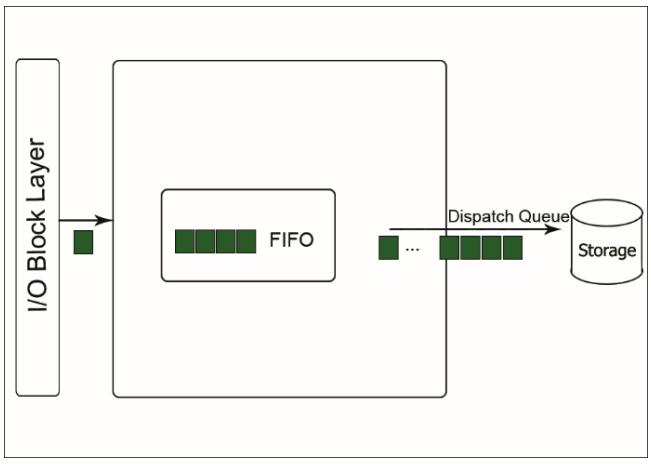

Figure 3. Noop scheduler.

2.4. The Budget Fair Queueing I/O Scheduler (BFQ)

The BFQ scheduler is an equal-share disk scheduling algorithm. It is based on CFQ that is default I/O scheduler in several Linux distributions. BFQ converts time intervals based round-robin to the number of sectors based round-robin. It assigns a sector budget to each request instead of a time slice. The simplified diagram of BFQ scheduler is shown in Figure 4.

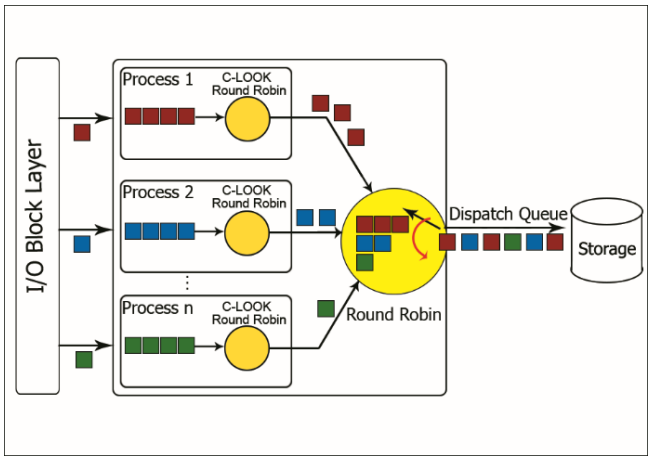

Figure 4. BFQ scheduler.

3. EXPERIMENTAL EVALUATION

The benchmark setup and the experimental platform with the different file sizes used for the performance analysis of the Deadline, CFQ, Noop, and BFQ schedulers are presented in this section. The results provided are read, reread, write and rewrite results of different schedulers as already explained in Section 2. For our analysis, we used IOzone to perform benchmarking of selected I/O schedulers.

3.1. Benchmark Setup

The preferred benchmark tool for the study presented in this paper is the IOzone benchmarking tool [13]. It generates workloads for several file operations. The IOzone provides a framework to run different scenarios to measure the performance of the system. It can measure the performance of file operations with different file sizes and different sized chunks of the file at a time. These chunks are particular spots within a file to read or write in one try. The size of these chunks affects the I/O performance. The experiments have been executed for each of the schedulers on the selected platform with constant 64 KB sized chunks and varying file sizes from 64 KB to 500 MB to measure the throughput performance against varying file sizes. The maximum throughput obtained for each of the file operations has been reported.

3.2. Experimental Platform

The experiments have been carried out on 4 cores 1.90 GHz Intel i7 351U processor system, with 4GB main memory, 256KB L2 cache, 4MB L3 cache running Manjaro Distribution (Linux 4.19.0-3). The SSD was a 240GB Sandisk U100 SATA 600. An EXT4 file-system has been used on the drive. The computer has been rebooted before each experiment to remove cache related effects.

3.3. Performance Metrics

The throughput of disks is finite and comparatively small while reading and writing. It causes bottlenecks that block I/O schedulers intend to improve overall system performance by changing the throughput performance. Total disk throughput (in KB/s) has been used as the metric to show the performance of the schedulers in the benchmark experiments. I/O intensive process execution time has been used to measure disk throughput. Any other processes have been killed except the daemons before benchmark execution. In these experiments, larger throughput (smallest execution time) means better scheduling for that file operations.

3.4. Workloads

The write, rewrite, read, and reread workloads have been used for the experiments using IOzone benchmark tool.

The read test is for measuring the reading performance of an existing file. The reread test is for measuring the reading performance of a file that was recently read. In this case, the performance tends to be higher as the data is

cached by the OS as it is recently accessed. The write test is for measuring the writing performance of a new file to the disk. Whenever a new file is written, the metadata is written on the block device in addition to the data itself. The rewrite performance becomes higher than the performance of writing a file because of this overhead. The rewrite test is for measuring the writing performance of a file that already exists on the disk. Whenever an existing file is written, the required effort is lower as the metadata is not written again.

4. ANALYSIS AND RESULTS

This section shows a comparative performance analysis using the workloads described in Section 3 for the Linux I/O schedulers Deadline, CFQ, Noop, and BFQ. The aim of the analysis is to understand how different schedulers perform under different workloads in terms of throughput.

For multiprocess throughput evaluation, we let IOzone run 3 processes for the initial write, rewrite, read, reread tests. These tests have been carried out 10 times and the results have been analyzed through the average of these tests.

Figure 5 shows the initial write test results. Noop gives the best performance with its SSD-ready structure. CFQ scheduler gives equal chance to every process and it has better write performance comparing to Deadline and BFQ. The schedulers without process priority have worse throughput results. Noop has an average 10% better write performance than its closest competitor CFQ.

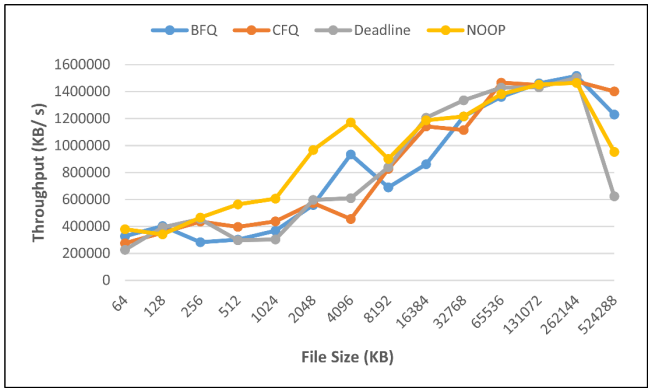

Figure 5. Write test results.

Figure 6 presents the rewrite test results. The throughput results have nearly the same ratio with the write test results. The write tests write the data and also the metadata for the file, but the rewrite test writes only the data to disk. All schedulers have higher rewriting performance than writing performance. The average rewrite performance of all schedulers is 38% better than the write performance. Noop has an average 4% better rewrite performance than its closest competitor CFQ.

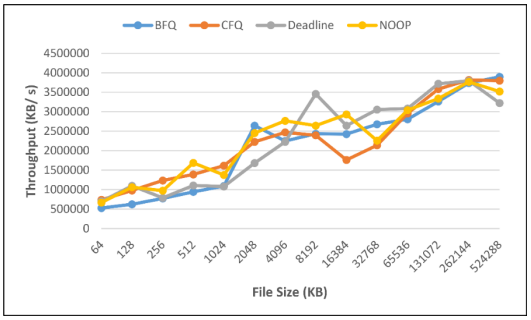

Figure 6. Rewrite test results.

The read and reread tests are shown in Figure 7 and Figure 8. Both read and reread throughput results are better then write and rewrite results for all schedulers. Noop has the best results. Deadline scheduler has better results than CFQ and BFQ in both read and reread tests because it prioritizes reads more than writes. Noop has an average 7.1% better read and 10.2% better reread performance than its closest competitor Deadline. Deadline has an average 2.9% and 6.7% better read performance than CFQ and BFQ respectively. It also has an average 1.5% and 7.9% better reread performance than CFQ and BFQ respectively.

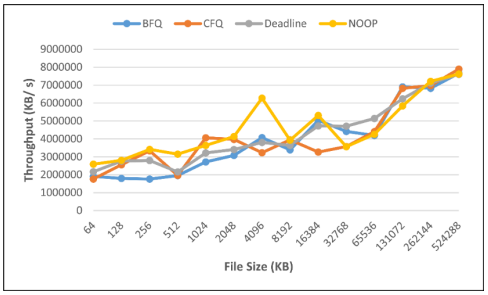

Figure 7. Read test results.

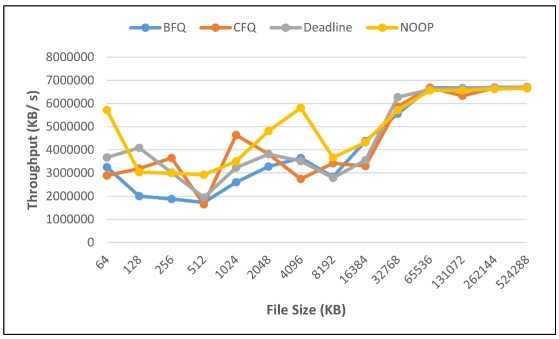

Figure 8. Reread test results.

The average of all tests is shown in Figure 9. CFQ scheduler provides an equal chance to each process in its round-robin structure. This makes it not suitable for environments that might need to prioritize request types for processes. Deadline scheduler is suitable for read-intensive works. Its default timeout values prioritize reads more than writes. These values are configurable according to the features of the work. Noop scheduler is optimized for systems that do not need a specific I/O scheduler. It has a modest structure with its single FIFO queue. It is suitable for the environments where the operating system is in a hypervisor. The underlying host operating system does scheduling itself in hypervisors or cloud environments that the operating systems inside virtual boxes do not need complex I/O schedulers. The BFQ scheduler is optimized for interactive tasks of personal-use scenarios instead of server scenarios. It focuses on delivering the lowest latency rather than reaching higher throughput. The operating system distributions focusing on personal usage do not perform the heavy read or write operations generally and lower latency is the prioritized to throughput.

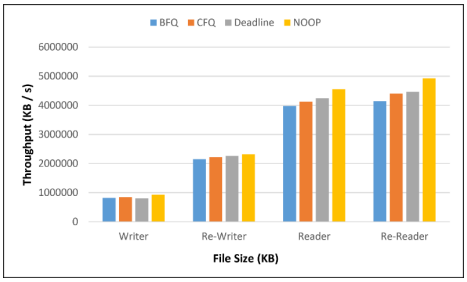

Figure 9. Avarage of all test results.

5. CONCLUSION

Block devices maintain request queues and approaches to schedule pending read or write requests. The kernel I/O scheduler subsystem is responsible for request optimization. The performance of Deadline, CFQ, Noop, and BFQ block I/O schedulers that are included in the latest Linux 4.1x kernel are compared in terms of throughput this paper. The tests have been carried out on an SSD block devices that are common in ranging from small handheld devices to large-scale data center configurations. According to the test results, each scheduler has different advantages over others. CFQ scheduler is suitable for the systems that require balanced I/O access and do not need process prioritization. Deadline scheduler has better performance on read-intensive works. Noop is for the systems on the cloud or hypervisors. BFQ performs better on interactive use-case scenarios. Noop is the simplest scheduler and it is considered to have the potential for optimized new implementations targeting SSD block devices.

REFERENCES

[1] R. Love, "The Block I/O Layer," in *Linux Kernel Development*, Crawfordsville, Indiana, Addison-Wesley, 2010, pp. 290-304.

[2] F. Chen, R. Lee, and X. Zhang. “Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing,” *In HPCA’11*, San Francisco, CA, 2011.

[3] J. Fusco, "The I/O Scheduler," in *The Linux programmer’s toolbox*, Upper Saddle River, NJ, Pearson Education, 2007, pp. 282-284.

[4] J. Kim, J. Kim, P. Park, J. Kim and J. Kim, "SSD Performance Modeling Using Bottleneck Analysis," in *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 80-83, 1 Jan.-June 2018.

[5] S. Mittal and J. S. Vetter, "A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems," in *IEEE Transactions*

on Parallel and Distributed Systems, vol. 27, no. 5, pp. 1537-1550, 1 May 2016.

[6] K. Shen and S. Park, “FlashFQ: A fair queueing I/O scheduler for flash-based SSDs,” in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, San Jose, CA, USA, Jun. 2013, pp. 67–78.

[7] C. Gao *et al*., "Exploiting Parallelism for Access Conflict Minimization in Flash-Based Solid State Drives," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 168-181, Jan. 2018.

[8] J. Lee, H. Roh and S. Park, "External Mergesort for Flash-Based Solid State Drives," in *IEEE Transactions on Computers*, vol. 65, no. 5, pp. 1518-1527, 1 May 2016.

[9] W. Wang and T. Xie, “PCFTL: A plane-centric flash translation layer utilizing copy-back operations,” *IEEE Trans. Parallel Distrib. Syst*., vol. 26, no. 12, pp. 3420–3432, Dec. 2015.

[10] B. Mao and S. Wu, “Exploiting request characteristics and internal parallelism to improve SSD performance,” in *Proc. 33rd IEEE Int. Conf. Comput. Design (ICCD)*, NY, USA, Oct. 2015, pp. 447–450.

[11] F. Chen, R. Lee, and X. Zhang, “Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing,” in *Proc. 17th Int. Conf. High-Perform. Comput. Archit. (HPCA)*, San Antonio, TX, USA, Feb. 2011, pp. 266–277.

[12] J. Guo, Y. Hu and Bo Mao, "SBIOS: An SSD-based Block I/O Scheduler with improved system performance," *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, Boston, MA, 2015, pp. 357-358.

[13] W. D. Norcott, D. Capps, “Iozone filesystem benchmark,” [Online]. Available: www.iozone.org. [Accessed 4 October 2018].