UMEÅ UNIVERSITY
Dept. of Computing Science
Obligatory assignment

Linux I/O Scheduling

Obligatory assignment

Namn	John-John Markstedt
E-post	c14jmt@cs.umu.se
Datum	March 20, 2018

1 Introduction

In this report three Linux I/O schedulers are tested and evaluated. The test is implemented in C.

2 Background

The I/O schedulers divides resources of disk I/O between pending block I/O requests. The schedulers can increase performance be merging and sorting requests. Since disk seeks is relatively slow compared to CPU cycles notable performance can be gained by sorting/merging the request to minimize the physical spinning of the disk and movement of the needle.

To actually test the policies and not some other more dominant factor of the stack some precautionary details needs to be dealt with. Firstly an actual hard disk drive must be read and written to and not SSD, RAM or any other medium where the schedulers effect is negligible. Also any sort of caching should be removed for the same reason.

3 Hypothesis and benchmarks

I/O schedulers are a part of a bigger machine and it's not obvious what should be consider good performance. Because what constitutes performance differ by purpose. Purpose can vary greatly. For example between the casual user who doesn't want his email to be blocked because he downloads a movie in the background to that of a specialist in computational problems who plans the computations weeks in advance and only cares about total throughput.

The schedulers tested are *cfs*, *noop* and *deadline*. The produced test program creates a bunch of threads and each thread starts to either read or write to a file, three different read and write sizes are implemented. Of course the read and write speed are of interest to all of the schedulers but their priority by purpose and which speeds are of interest depends on the specific scheduler.

3.1 cfs

The *Completely Fair scheduling* is the default scheduler on current instances and as the name suggest is supposed to be fair. It's for the average user that doesn't want his/her gui to freeze while doing a small read/write just because he/her compiles the Linux kernel in the background. Therefor total throughput is not the most important metric. Variance might be of interest here and maybe worst case performances. This scheduler should be expected to read/write faster on small files while heavier work are going on at the same time.

3.2 noop

The *noop* scheduler does no more nor less than FIFO, constant overhead. This is perfect when a SSD, flash or other memory are used and the nothing is to be gained for merging and sorting the request queue[1]. For an actual HDD however it should be slower overall, but some work that got early in the queue might be fast. Since it doesn't optimize the HDD seeks it should be slowest overall.

3.3 deadline

The *deadline* scheduler guaranties a start service time[1]. It does so by imposing a limit requests to a expiration time. The expiration time for reads are 500ms and 5s writes. Total throughput should most likely suffer if the request queue becomes long.

4 Running tests

The source files can be found on github[2],
A README is attached in the repository with instructions on how to run it.

4.1 Basic structure of test

First the bash script disables cache.
For each scheduler run or benchmark program.

The program starts, sets up and starts up threads. Each thread either read or write a small 0,1 KB file, 100KB or 100MB file simultaneously. All these I/O operations are independently timed and an mean, median and deviations are calculated for each file size and each operation(r/w). Also total throughput is measured, basically all these threads together.

This is then repeated with different amount of threads.

The test repeats the file reads/writes proportionally to the thread amount, a minimum of six times. From those repeats the median, mean and deviation can be calculated.

4.2 Environment

The results are gathered from the test running on a machine with following operating system and HDD.

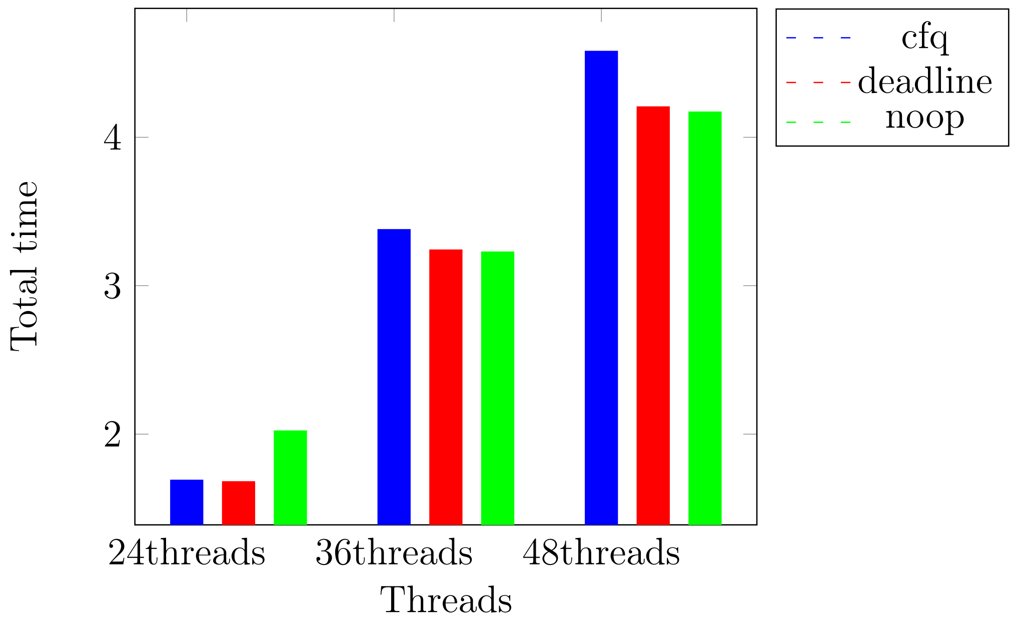
Operating System

```
Distributor ID: Ubuntu\\
Description: Ubuntu 16.04.1 LTS\\
Release: 16.04\\
Codename: xenial\\
```

Hard disk drive

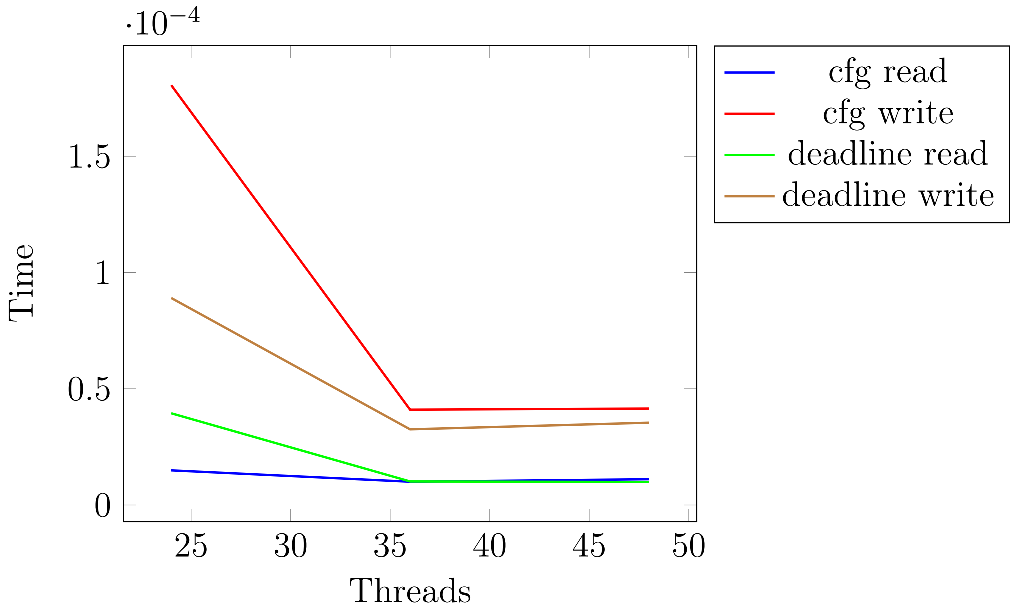
```
== START OF INFORMATION SECTION ===
Model Family:     Seagate Barracuda 7200.14 (AF) \\
Device Model:     ST1000DM003-1SB102
Serial Number:    Z9A5JTJ0
LU WWN Device Id: 5 000c50 0924c0316
Firmware Version: CC43
User Capacity:    1 000 204 886 016 bytes [1,00 TB]
Sector Sizes:     512 bytes logical, 4096 bytes physical
Rotation Rate:    7200 rpm
Form Factor:      3.5 inches
Device is:        In smartctl database [for details use: -P show]
ATA Version is:   ATA8-ACS T13/1699-D revision 4
SATA Version is:  SATA 3.0, 6.0 Gb/s (current: 3.0 Gb/s)
Local Time is:    Mon Mar 19 14:29:36 2018 CET
SMART support is: Available - device has SMART capability.
SMART support is: Enabled
```

5 Results



Figur 1: Total throughput, in seconds

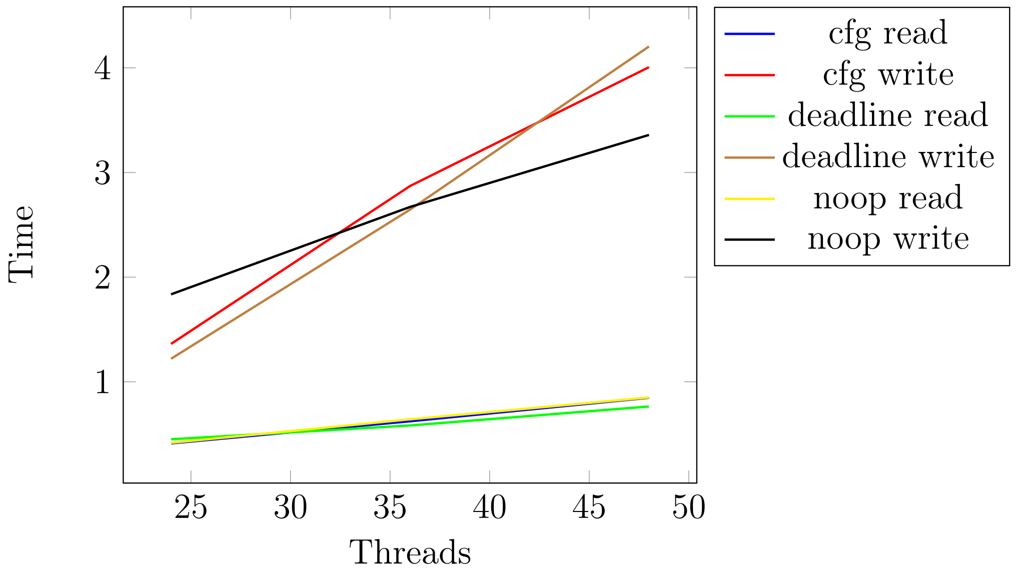
5.1 Median for small files



Figur 2: Median for small files, time to read/write 0.1kb

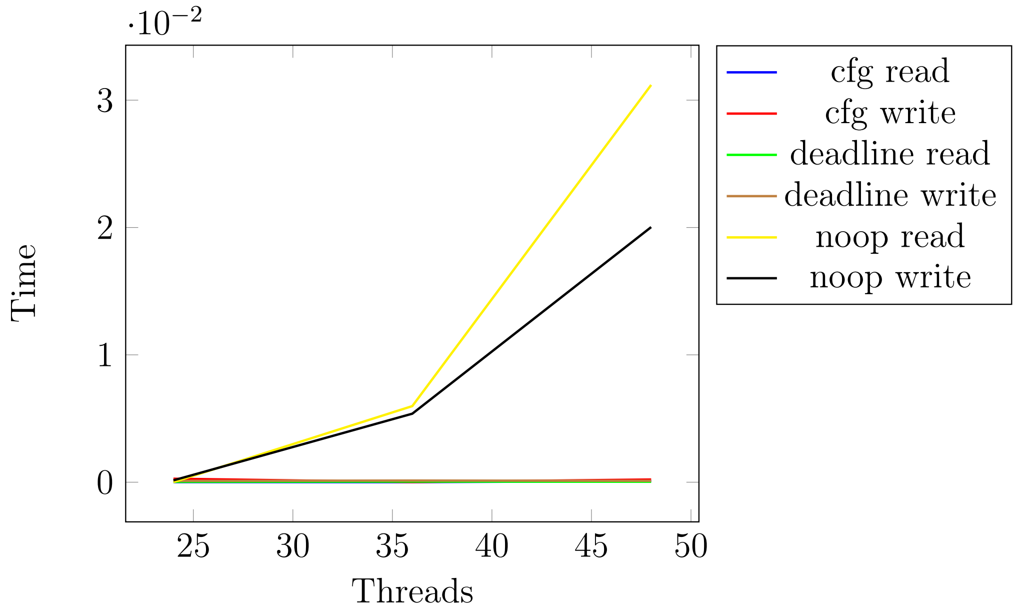
The noop datapoints where all over the place and were removed to be readable.

5.2 Median for large files



Figur 3: Median for small files, time to read/write 100MB

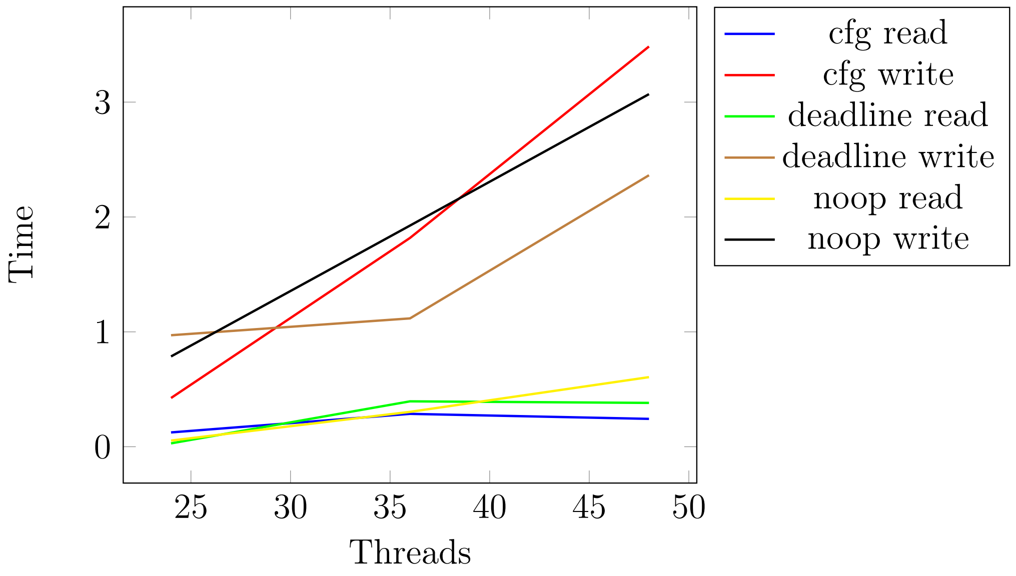
5.3 Standard deviation for small files



Figur 4: Standard deviation for small files, time deviated from mean to read/write 0.1KB

Only thing to note is that noops variance is garbage.

5.4 Standard deviation for large files



Figur 5: Standard deviation for large files, time deviated from mean to read/write 100MB

6 Summary and Conclusions

The hypotheses seems to hold true for fewer threads. Over 30 it seems to go in complete opposite direction. I wonder if thats because the overhead of switching between reads/writes to keep it fair. It stands however on the Standard deviation when threads are increased. The result doesn't show that deadline priorities deadline, most likely due to the switching of I/O request from deadlines.

We also see that deadline prioritize reads, since it's deviation is much lower compared to the writes.

However it's quite interesting that, with increasing work size(amount of threads) the total throughput goes up fastest for noop but it's deviation also goes up fastest of the schedulers. This suggests that noops total speed comes at the cost of being unfair. This cost becomes significant for smaller files where the priority, switching between requests takes longer time relative to the actual read/write time.

7 Tuning

Each of the schedulers can be tuned to fit your needs, if you do something with a very specific read/write pattern it's probable that some time can be cut.

For example if there is no necessity for any particular request to finish, one could tune the read and write expires in deadline so some request doesn't need to be forced to the front of the queue and neglect the optimal order.

If fairness is of concern, the low latency flag of CFQ can be set to 1, for example if an application requires something like real-time media streaming[3].

If using noop with an SSD or any medium which doesn't rotate, the rotational flag should be set to 0. This skips unnecessary logic[3]. Also if you want to boost performance of an read/write heavy application, the *optimal_io_size* could be used to set the requests optimal for performance. The requests should be a multiple of the *optimal_io_size*[3].

8 Appendix

8.1 deadline

time\threads	24	36	48
Median Large Read	4.508281e-01	5.827010e-01	7.640525e-01
Median Large Write	1.220076e+00	2.645316e+00	4.203992e+00
Median Medium Read	5.040169e-04	2.589226e-04	6.183982e-03
Median Medium Write	2.565503e-03	3.809929e-04	9.125471e-03
Median Small Read	3.945827e-05	1.013279e-05	9.894371e-06
Median Small Write	8.904934e-05	3.254414e-05	3.540516e-05
Mean Large Read	4.441810e-01	5.423790e-01	7.385803e-01
Mean Large Write	1.149797e+00	2.606468e+00	3.414538e+00
Mean Medium Read	2.309799e-03	9.280046e-04	9.148002e-03
Mean Medium Write	2.521038e-03	1.049161e-03	1.154286e-02
Mean Small Read	3.248453e-05	1.891454e-05	1.433492e-05
Mean Small Write	9.047985e-05	5.586942e-05	4.974008e-05
Deviation Large Read	2.939829e-02	3.952315e-01	3.817024e-01
Deviation Large Write	9.707248e-01	1.117286e+00	2.363344e+00
Deviation Median Medium Read	6.685795e-03	3.741321e-03	2.752705e-02
Deviation Median Medium Write	4.039926e-03	3.559908e-03	2.998894e-02
Deviation Median Small Read	2.599173e-05	4.759383e-05	3.396513e-05
Deviation Median Small Write	1.014423e-04	1.276315e-04	1.086958e-04
Total Throughput time	1.678868e+00	3.239830e+00	4.203992e+00

Figur 5: Showing run data for cfq

8.2 cfq

time\threads	24	36	48
Median Large Read	4.117070e-01	6.226795e-01	8.473586e-01
Median Large Write	1.361951e+00	2.870458e+00	4.006412e+00
Median Medium Read	2.491474e-04	6.134510e-03	1.062012e-02
Median Medium Write	4.109144e-04	3.399849e-04	5.861521e-04
Median Small Read	1.490116e-05	1.001358e-05	1.108646e-05
Median Small Write	1.806021e-04	4.100800e-05	4.148483e-05
Mean Large Read	4.022875e-01	5.913817e-01	8.152933e-01
Mean Large Write	1.363843e+00	2.579763e+00	3.344527e+00
Mean Medium Read	2.495646e-04	6.951292e-03	1.080018e-02
Mean Medium Write	2.523005e-03	3.646016e-03	8.231550e-03
Mean Small Read	1.472235e-05	1.033147e-05	2.691150e-05
Mean Small Write	1.870394e-04	4.152457e-05	9.182096e-05
Deviation Large Read	1.247088e-01	2.862271e-01	2.426294e-01
Deviation Large Write	4.242320e-01	1.817729e+00	3.484045e+00
Deviation Median Medium Read	5.124322e-05	1.681176e-02	2.748194e-02
Deviation Median Medium Write	7.433659e-03	1.812884e-02	2.867138e-02
Deviation Median Small Read	7.676745e-06	3.156984e-06	8.785615e-05
Deviation Median Small Write	2.721254e-04	1.050891e-05	2.102680e-04
Total Throughput time	1.688821e+00	3.377146e+00	4.578295e+00

Figur 5: Showing run data for cfq

8.3 noop

time\threads	24	36	48
Median Large Read	4.168215e-01	6.417174e-01	8.503779e-01
Median Large Write	1.835837e+00	2.670707e+00	3.359053e+00
Median Medium Read	9.906292e-04	1.192355e-02	8.616924e-03
Median Medium Write	2.156973e-03	1.297498e-02	1.362741e-02
Median Small Read	1.096725e-05	1.170194e-02	8.554578e-03
Median Small Write	1.094341e-04	1.179945e-02	8.554578e-03
Mean Large Read	4.173437e-01	5.833800e-01	7.555637e-01
Mean Large Write	1.675934e+00	2.389424e+00	3.049520e+00
Mean Medium Read	1.096308e-03	1.038818e-02	1.928723e-02
Mean Medium Write	2.149463e-03	1.305763e-02	2.289876e-02
Mean Small Read	1.126528e-05	1.051696e-02	9.569794e-03
Mean Small Write	1.119375e-04	1.026710e-02	8.305043e-03
Deviation Large Read	5.379346e-02	3.037622e-01	6.059237e-01
Deviation Large Write	7.859759e-01	1.925276e+00	3.069310e+00
Deviation Median Medium Read	1.201016e-03	7.973069e-03	4.986949e-02
Deviation Median Medium Write	7.654056e-04	1.360010e-03	4.708369e-02
Deviation Median Small Read	4.925234e-06	5.974571e-03	3.119514e-02
Deviation Median Small Write	1.442692e-04	5.378539e-03	2.003178e-02
Total Throughput time	2.020777e+00	3.226226e+00	4.169057e+00

Figur 5: Showing run data for deadline

References

[1] https://www.techrepublic.com/article/how-to-change-the-linux-io-scheduler-to-fit-your-needs/

[2] *https://github.com/Johnstedt/Linux_IO_Scheduling_Comparison.*

[3] https://cromwell-intl.com/open-source/performance-tuning/disks.html