Split-level I/O Scheduling

Suli Yang, Tyler Harter, Anand Krishnamurphy, Salini Kowsalya, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
What is Scheduling?

Scheduling Involves:

- Specifying
- Accounting
- Reordering
What is (disk) Scheduling?

Scheduling Involves:

- Specifying
- Accounting
- Reordering
What is (disk) Scheduling?

Scheduling Involves:

- Specifying
- Accounting
- Reordering

Client 1 (75%)

Client 2 (25%)
What is (disk) Scheduling?

Scheduling Involves:
- Specifying
- Accounting
- Reordering

Client 1 (75%)

Client 2 (25%)

50%

50%
What is (disk) Scheduling?

Scheduling Involves:

- Specifying
- Accounting
- Reordering

Client 1 (75%)
Clint 2 (25%)

Queues
75%
25%

disk
Problem Preview: FS

Client 1 (75%) → File System → Client 2 (25%)

disk
Outline

Intro: disk scheduling basics

CFQ isn’t fair!

FS Scheduling Challenges (ext4 case study)

Naive approach (not our idea)

Split-Level Scheduling

Conclusions
CFQ Eval (Linux Default)

“Completely” Fair Queue
Maintains per-task queues
Time-share across queues
Higher priority => bigger time slice
Prios are 0-7, with 0 highest (fastest)
Eval Workloads

8 tasks, priorities from 0-7
Each task accesses its own file
Sequential I/O only
4KB requests
Does CFQ respect priorities for basic reads and writes?
(a) Read

Priority MBs/second

Priority

0 1 2 3 4 5 6 7

MBs/second
Conclusion: CFQ respects read priorities -- good!
(b) Write

Priority MBs/second
Conclusion: write priorities not respected
Why? >99% of I/O blamed on writeback task
What if we force each process does its own writing?
(with O_DIRECT)
(c) Direct

MBs/second vs Priority

0.0
0.3
0.6
0.9
1.2
1.5

0 1 2 3 4 5 6 7
Conclusion: yes, but performance suffers
Does 0_DIRECT trick work if metadata is flushed often?
Conclusion: no, priorities not respected
Why? Fsync enforces global ordering which CFQ cannot help with.
CFQ Eval Conclusion

Rename CFQ => SFQ (sometimes fair queueing)

Is CFQ just a bad implementation?

No, the whole scheduling framework and architecture is bad

FS/block interface gives schedulers little/no knowledge of or control over FS features important to scheduling
Outline

Intro: disk scheduling basics

CFQ isn’t fair!

FS Scheduling Challenges (ext4 case study)

Naive approach (not our idea)

Split-Level Scheduling

Conclusions
What makes CFQ’s life hard?

• …Writes!

• Write delegation prevents correct accounting.

• Ordering requirement prevents priority-based re-ordering
An ext4 Case Study
# Problematic FS Features

<table>
<thead>
<tr>
<th></th>
<th>Accounting</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Journaling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared Metadata</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Buffering</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delayed Allocation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Problematic FS Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Accounting</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Journaling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared Metadata</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Buffering</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delayed Allocation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conflict of interest!

*Journal* has ordering requirement for *consistency*

*Scheduler* wants to re-order for *fairness*
Review (ordered mode)

- FS/Journal
- Scheduler
- Disk
Review (ordered mode)

high-prio `write()`

FS/Journal

Scheduler

Disk
Review (ordered mode)

low-prio write()

data transaction transaction data

FS/Journal
Scheduler
Disk
Review (ordered mode)
batching combines two small transactions into one big one for performance
Review (ordered mode)

high-prio `fsync()` blocks til transaction on disk

FS/Journal

Scheduler

Disk
Review (ordered mode)

Consistency imposes requirement that transaction hits disk *after all* data blocks.

It doesn’t matter which block the scheduler flushes first. Scheduler can’t unbatch the transaction to help the fsync().
Review (ordered mode)

high-prio `fsync()` blocks til transaction on disk

Priority inversion!  High-prio `fsync` depends on low-prio block
file system journal writes transaction \textit{on behalf of} the actual writers

Also, who to blame for the transaction write?
# Problematic FS Features

<table>
<thead>
<tr>
<th></th>
<th>Accounting</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Journaling</td>
<td>bad</td>
<td>bad</td>
</tr>
<tr>
<td>Shared Metadata</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Buffering</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delayed Allocation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Problematic FS Features

<table>
<thead>
<tr>
<th></th>
<th>Accounting</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Journaling</td>
<td><strong>bad</strong></td>
<td><strong>bad</strong></td>
</tr>
<tr>
<td>Shared Metadata</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Buffering</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delayed Allocation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Problematic FS Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Accounting</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Journaling</td>
<td>bad</td>
<td>bad</td>
</tr>
<tr>
<td>Shared Metadata</td>
<td>bad</td>
<td>bad</td>
</tr>
<tr>
<td>Write Buffering</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delayed Allocation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Problematic FS Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Accounting</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Journaling</td>
<td>bad</td>
<td>bad</td>
</tr>
<tr>
<td>Shared Metadata</td>
<td>bad</td>
<td>bad</td>
</tr>
<tr>
<td>Write Buffering</td>
<td>bad</td>
<td>neutral</td>
</tr>
<tr>
<td>Delayed Allocation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problematic FS Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Accounting</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Journaling</td>
<td>bad</td>
<td>bad</td>
</tr>
<tr>
<td>Shared Metadata</td>
<td>bad</td>
<td>bad</td>
</tr>
<tr>
<td>Write Buffering</td>
<td>bad</td>
<td>neutral</td>
</tr>
<tr>
<td>Delayed Allocation</td>
<td>bad</td>
<td>good</td>
</tr>
</tbody>
</table>
Just ext4?

- Almost all file systems use ordering requirements to ensure crash consistency (Soft updates: FFS, Journaling: CFS, Copy-on-Write: ZFS)

- Write delegation everywhere (Write-back built in kernel, delaying work for performance)
Just ext4?

• Write delegation and ordering requirements are universal file system properties

• Makes block level write scheduling inherently hard (if not impossible)
Outline

Intro: disk scheduling basics

CFQ isn’t fair!

FS Scheduling Challenges (ext4 case study)

Naive approach (not our idea)

Split-Level Scheduling

Conclusions
System Call Scheduling

Idea: hold back read and write system calls instead of holding back block I/O

Craciunas et al., SIGOPS OSR ‘08

Advantages:
  Simple
  Does scheduling above the messy FS level
Traditional Scheduling

Client → Read() → FS → Queues → disk
Client → Write() → FS → Queues → disk
Client → Read() → FS → Queues → disk
Client → Write() → FS → Queues → disk
Client → Read() → FS → Queues → disk
Client → Write() → FS → Queues → disk
Client → Read() → FS → Queues → disk
Client → Write() → FS → Queues → disk

Block
System Call Scheduling

- Client
- Client
- Client

Queues
- Read()
- Write()

FS

Block

- Read
- Write

disk
System Call Scheduling

Problem 1: what if we read/write can be absorbed by cache?
System Call Scheduling

Client

Client

Client

Queues

Read()

Write()

Cache

Read

Write

FS

Block

Problem 2: writes now block (previously asynchronous)
Problem 3: not all FS I/O has the same cost (e.g., random I/O), or that involving metadata
Outline

Intro: disk scheduling basics
CFQ isn’t fair!
FS Scheduling Challenges (ext4 case study)
Naive approach (not our idea)
Split-Level Scheduling
Conclusions
New Cross Layer Scheduler Framework

- New notification to scheduler: *file system events* (write/fsync called/completed, write back happened)
- New action available: queue *system calls* in addition to block level requests, *flush* file cache
- New info of accounting: *io-tag* for client identification
New Cross Layer Scheduler Framework

- File system view *and* block level view: both high level ordering and low level optimization
- Ability to control important *file system* behavior and memory state.
- io-tag enables *correct and accurate* (low level) accounting.
Things We Enable

- Correct priority-based I/O scheduling.
- I/O isolation based on cache partitioning.
- Real end-to-end latency control.
- and others…
Split Level Actual Fair Queuing

Client

Queues

Read()

fsync()

Write()

FS

Block

Queues

Read

Write

Accounting
Outline

Intro: disk scheduling basics
CFQ isn’t fair!
FS Scheduling Challenges (ext4 case study)
Naive approach (not our idea)
Split-Level Scheduling (Preliminary Results)
Conclusions
Asynchronous Writes now work!
Write+Fsync works too!
Outline

Intro: disk scheduling basics

CFQ isn’t fair!

FS Scheduling Challenges (ext4 case study)

Naive approach (not our idea)

Split-Level Scheduling (Implementation)

Conclusions
Conclusions

Life’s not fair, but file systems should be

Reads are easy, writes are hard

Simple layer stacking makes some problems impossible to solve - have to work cross-layer
New Cross Layer Scheduler Framework

- New notification to scheduler: `add_block_req`, `add_write_call`, `add_fsync_call`, `req_complete`, `write_complete`, `fsync_complete`, `writeback_happened`, `disk_need_work`

- New action available: `issue_block_req`, `issue_write_call`, `issue_fsync_call`, `flush_file_cache`

- New info of accounting: `io-tag` for client identification