SNIA ■ SANTA CLARA, 2017

Application Crash Consistency and Performance with CCFS

Thanu Pillai

Google

(This talk covers work done entirely at University of Wisconsin-Madison)

Application-Level Crash Consistency

Storage must be robust even with system crashes

- Power loss (2016 Github outage, Internet outage across UK) [source:www.datacenterknowledge.com]
- Kernel bugs [Lu et al., OSDI 2014, Palix et al., ASPLOS 2011, Chou et al., SOSP 2001]

Applications need to implement crash consistency

- E.g., Database applications ensure transactions are atomic

Applications implement crash consistency wrongly

- Pillai et al., OSDI '14 (11 applications) and Zhou et al., OSDI '14 (8 databases)
- Conclusion: All applications had some form of incorrectness

Ordering and Application Consistency

App crash consistency depends on FS behavior [Pillai et al., OSDI 2014]

- E.g., Bad FS behavior: 60 vulnerabilities in 11 applications
- Good FS behavior: 10 vulnerabilities in 11 applications

FS-level ordering is important for applications

- All writes should (logically) be persisted in their issued order
- Major factor affecting application crash consistency

Few FS configurations provide FS-level ordering

- Ordering is considered bad for performance

This talk covers ...

Problem: Current FS and Application Behavior

- How much do file systems differ in crash consistency?
- How much does it affect real-world applications?

Solution: Stream abstraction

- Allows FS-level ordering with little performance overhead
- Allows much higher performance with application code changes

Crash-Consistent File System (CCFS)

- Efficient implementation of stream abstraction on ext4

Outline

- Introduction
- Example
- Problem: FS and Application Behavior
- Stream API
- Crash-Consistent File System
- Conclusion

Toy Example: Overview

A file initially contains the string “a foo”

- Assume each character in “a foo” is a block of data

Task: Atomically change the contents to “a bar”

- On a power loss, we must retrieve either “a foo” or “a bar”

Toy Example: Simple Overwrite

Intermediate states possible on crash

Initial state	Modification	Final state
/x/f1 "a foo"	pwrite(/x/f1, 2, "bar")	/x/f1 "a bar"

Intermediate state 1	Intermediate state 2	Intermediate states 3, 4, 5 ....
/x/f1 "a boo"	/x/f1 "a far"	

SDC 17

Toy Example: Maintaining Consistency

What if crash atomicity is needed?

Use application-level logging (a.k.a. *undo logging/rollback journaling*)

1. Make a copy of old data in “log” file
2. Modify actual file
3. Delete log file
4. On a crash, data can be recovered from the log

Toy Example: Protocol #1

Works in ext3(*data-journal*)!

Some possible intermediate states

Update Protocol

```
creat(/x/log1);
write(/x/log1, "2, 3, foo");

pwrite(/x/f1, 2, "bar");

unlink(/x/log1);
```

1.

/x/f1	"a foo"
/x/log1	""

2.

/x/f1	"a foo"
/x/log1	"2, 3, f"

3.

/x/f1	"a boo"
/x/log1	"2, 3, foo"

→ Recover from log file during recovery

Toy Example: Protocol #1

Works in ext3(*data-journal*)!

Doesn't work in ext3(*data-ordered*)

Update Protocol

```
creat(/x/log1);
write(/x/log1, "2, 3, foo");


pwrite(/x/f1, 2, "bar");

unlink(/x/log1);
```

A possible intermediate state

/x/f1	"a boo"
/x/log1	""

Recovery not possible!

Toy Example: Protocol #2

Works in ext3(*data-journal*), (*data-ordered*)!

Doesn't work in ext3(*writeback*)

Update Protocol

Crash here →

```
creat(/x/log1);
write(/x/log1, "2, 3, foo");
fsync(/x/log1);

pwrite(/x/f1, 2, "bar");
fsync(/x/f1);
unlink(/x/log1);
```

A possible intermediate states

```
/x/f1          "a foo"
/x/log1        "2, 3,
#!@"
```

File size alone increases for log1, and garbage occurs. Recovery cannot differentiate between garbage and data!

Toy Example: Protocol #3

Works in ext3(*data-journal*), (*data-ordered*), (*writeback*)

Not enough, according to Linux manpages

Update Protocol

```
creat(/x/log1);
write(/x/log1, "2, 3, checksum, foo");
fsync(/x/log1);

pwrite(/x/f1, 2, "bar");
fsync(/x/f1);
unlink(/x/log1);
```

A possible intermediate states

The log file’s directory entry might never be created

Toy Example: Protocol #4

Works in all file systems

Update Protocol

```
creat(/x/log1);
write(/x/log1, "2, 3, checksum, foo");
fsync(/x/log1);
fsync(/x);
pwrite(/x/f1, 2, "bar");
fsync(/x/f1);
unlink(/x/log1);
```

Toy Example: Protocol #5

Works in all file systems

(Additional fsync() required for durability in all FS)

Update Protocol

```
creat(/x/log1);
write(/x/log1, "2, 3, checksum, foo");
fsync(/x/log1);
fsync(/x);
pwrite(/x/f1, 2, "bar");
fsync(/x/f1);
unlink(/x/log1);
fsync(/x);
```

Example: Summary

File systems vary in crash-related behavior

- ext3(*ordered*) re-orders, while ext3(*journaled*) does not

Applications usually *depend* on some behavior

- Depend on ordering: Some fsync() calls can be omitted

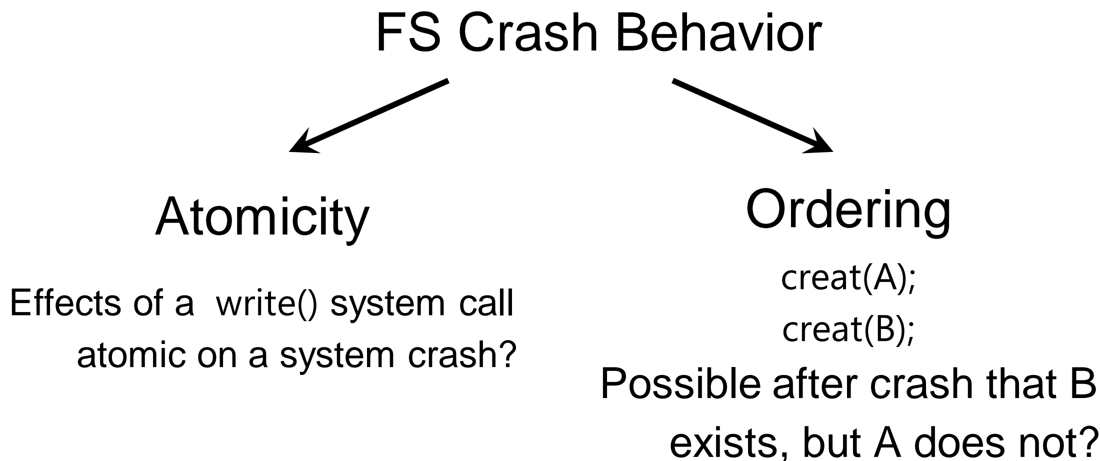
Outline

- Introduction
- Example
- Problem: FS and Application Behavior
- Stream API
- Crash-Consistent File System
- Conclusion

File-System Behavior

Each file system behaves differently across a crash

- Little standardization of behavior across crashes



File-System Study: Results

Main result: File systems vary in their persistence properties

File system configuration		Atomicity				Ordering			
		One sector overwrite	*Append content*	*Many sector overwrite*	*Directory operation*	*Overwrite → Any op*	*Append → Any op*	*Dir-op → Any op*	*Append → Rename*
ext2	*Async*		✗	✗	✗	✗	✗	✗	✗
ext2	*Sync*		✗	✗	✗				
ext3	*writeback*		✗	✗		✗	✗		✗
ext3	*ordered*			✗		✗			
ext3	*data-journal*			✗					
ext4	*writeback*		✗	✗		✗	✗		✗
ext4	*ordered*			✗		✗	✗		
ext4	*no-delalloc*			✗		✗			
ext4	*data-journal*			✗					
btrfs				✗			✗	✗	
Xfs	*default*			✗		✗	✗		
Xfs	*wsync*			✗		✗			

Vulnerabilities Study

Previous work: App crash consistency vs FS behavior

[Pillai et al., OSDI 2014]

"Vulnerability": Place in application source code that can lead to inconsistency, depending on FS behavior

Vulnerabilities Study: Results

	Ext2-like FS	Btrfs	Ext3-DJ
Ordering	✗	✗	✓
Atomicity	✗	✓	✓
LevelDB-1.10	10	4	1
LevelDB-1.15	6	3	1
LMDB	1		
GDBM	5	4	2
HSQLDB	10	4	
SQLite-Roll	1	1	1
SQLite-WAL	0		
PostgreSQL	1		
Git	9	5	2
Mercurial	10	8	3
VMWare	1		
HDFS	2	1	
ZooKeeper	4	1	
Total	**60**	**31**	**10**

Under data-journaled ext3, with both atomicity and ordering, 10 vulnerabilities

- Minor consequences

SQLite-Roll (1) → Documentation error

Git (2) → Dirstate corruption

Problem: Real-world vs Ideal FS behavior

Ideal behavior: Ordering, “weak atomicity”

- *All* file system updates should be persisted in-order
- Writes can split at sector boundary; everything else atomic

Modern file systems already provide weak atomicity

- E.g.: Default modes of ext4, btrfs, xfs

Only rarely used FS configurations provide ordering

- E.g.: Data-journaling mode of ext4, ext3

Outline

- Introduction
- Example
- Problem: FS and Application Behavior
- Stream API
- Crash-Consistent File System
- Conclusion

Why not use an order-preserving FS?

Some existing file systems preserve order

- Example: ext3 and ext4 under data-journaling mode
- Performance overhead?

New techniques are efficient in maintaining order

- CoW, optimized forms of journaling
- Ordering doesn't require disk-level seeks

Reason: False ordering dependencies

- Inherent overhead of ordering, irrespective of technique used

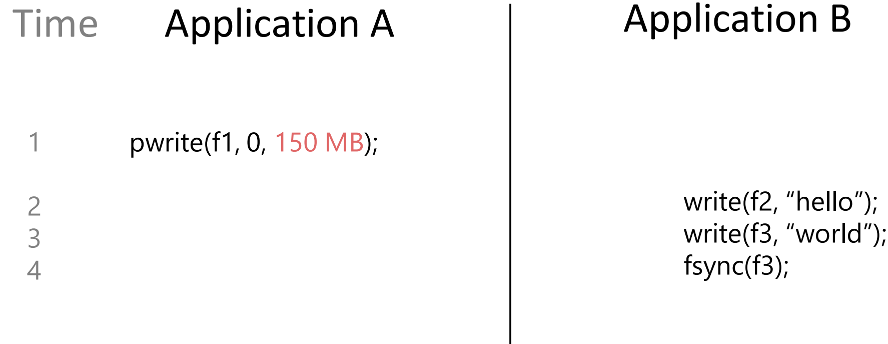
SDC 17

False Ordering Dependencies

Problem: Ordering between independent applications

Solution: Order only within each application

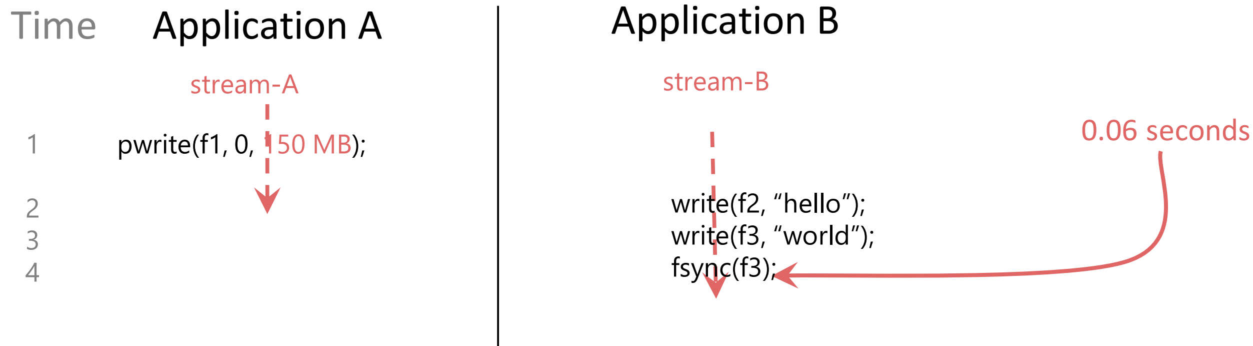
- Avoids performance overhead, provides app consistency



Stream Abstraction

New abstraction: Order only within a *"stream"*

- Each application is usually put into a separate stream



Stream API: Normal Usage

New set_stream() call

- All updates after set_stream(X) associated with stream X
- When process forks, previous stream is adopted

Using streams is easy

- Add a single set_stream() call in beginning of application
- Backward-compatible: set_stream() is no-op in older FSes

Stream API: Extended Usage

set_stream() is versatile

- Many applications can be assigned the same stream
- Threads within an application can use different streams
- Single thread can keep switching between streams

Ordering vs Durability: stream_sync(), IGNORE_FSYNC flag

- Applications use fsync() for both ordering and durability
- IGNORE_FSYNC ignores fsync(), respects stream_sync()

[Chidambaram et al., SOSP2013]

Streams: Summary

In an ordered FS, false dependencies cause overhead

- Inherent overhead, independent of technique used

Streams provide order only within application

- Writes across applications can be re-ordered for performance
- For consistency, ordering required only within application

Easy to use!

Outline

Introduction

Example

Problem: FS and Application Behavior

Stream API

Crash-Consistent File System

Conclusion

CCFS: Design

"Crash consistent file system"

- Efficient implementation of stream abstraction

Basic design: Based on ext4 with data-journaling

- Ext4 data-journaling guarantees global ordering
- Ordering across all applications: false dependencies
- CCFS uses separate transactions for each stream

Multiple challenges

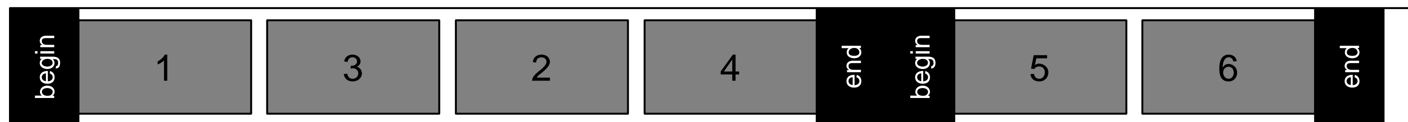
Ext4 Journaling: Global Order

On system crash, on-disk journal transactions recovered atomically, in sequential order

Global ordering is maintained!

Main memory

Running transaction



On-disk journal

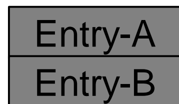
CCFS: Stream Order

Ordering maintained within stream,
re-order across streams!

CCFS: Multiple Challenges

Example: Two streams updating adjoining dir-entries

Block-1 (belonging to directory X)



Application A	Application B
set_stream(A)	set_stream(B)
creat(/X/A)	
	creat(/X/B)

Challenge #1: Block-Level Journaling

CCFS solution:

Record running transactions at byte granularity

Commit at block granularity

Application A

```
set_stream(A)
creat(/X/A)
```

Application B

```
set_stream(B)

creat(/X/B)
```

stream-A transaction

stream-B transaction

Main memory

Entry-A

Entry-B

Old version of entry-A

On-disk journal

begin

Entry-A

Entry-B

end

Entire block-1 committed

More Challenges ...

1. Both streams update directory's modification date
    - Solution: Delta journaling
2. Directory entries contain pointers to adjoining entry
    - Solution: Pointer-less data structures
3. Directory entry freed by stream A can be reused by stream B
    - Solution: Order-less space reuse
4. Ordering technique: Data journaling cost
    - Solution: Selective data journaling [Chidambaram et al., SOSP 2013]
5. Ordering technique: Delayed allocation requires re-ordering
    - Solution: Order-preserving delayed allocation

Details in the paper!

Evaluation

1. Does CCFS solve application vulnerabilities?

	Vulnerabilities	
Application	**ext4**	**ccfs**
LevelDB	1	0
SQLite-Roll	0	0
Git	2	0
Mercurial	5	2
ZooKeeper	1	0

Ext4: 9 Vulnerabilities

- Consistency lost in LevelDB
- Repository corrupted in Git, Mercurial
- ZooKeeper becomes unavailable

CCFS: 2 vulnerabilities in Mercurial

- Dirstate corruption

Evaluation

2. Performance within an application

Throughput: normalized to ext4 (Higher is better)

	ext4	ccfs	ccfs+
varmail			
randwrite			
createfiles			
seqwrite			
fileserver			
webserver			
Git			
SQLite			
LevelDB			

But, performance can be improved with IGNORE_FSYNC and stream_sync()!

Evaluation: Summary

Crash consistency: Better than ext4

- 9 vulnerabilities in ext4, 2 minor in CCFS

Performance: Like ext4 with little programmer overhead

- Much better with additional programmer effort

Conclusion

- FS crash behavior is currently not standardized
- Current applications have bad crash consistency
- Ideal FS behavior is considered bad for performance
- Stream abstraction and CCFS solve this dilemma