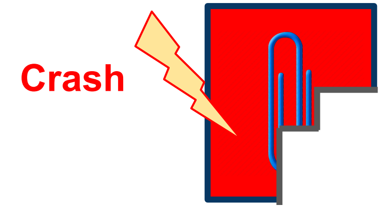
TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions

Yige Hu, Zhiting Zhu, Ian Neal,
Youngjin Kwon, Tianyu Chen,
Vijay Chidambaram,
Emmett Witchel
The University of Texas at Austin

Applications need crash consistency



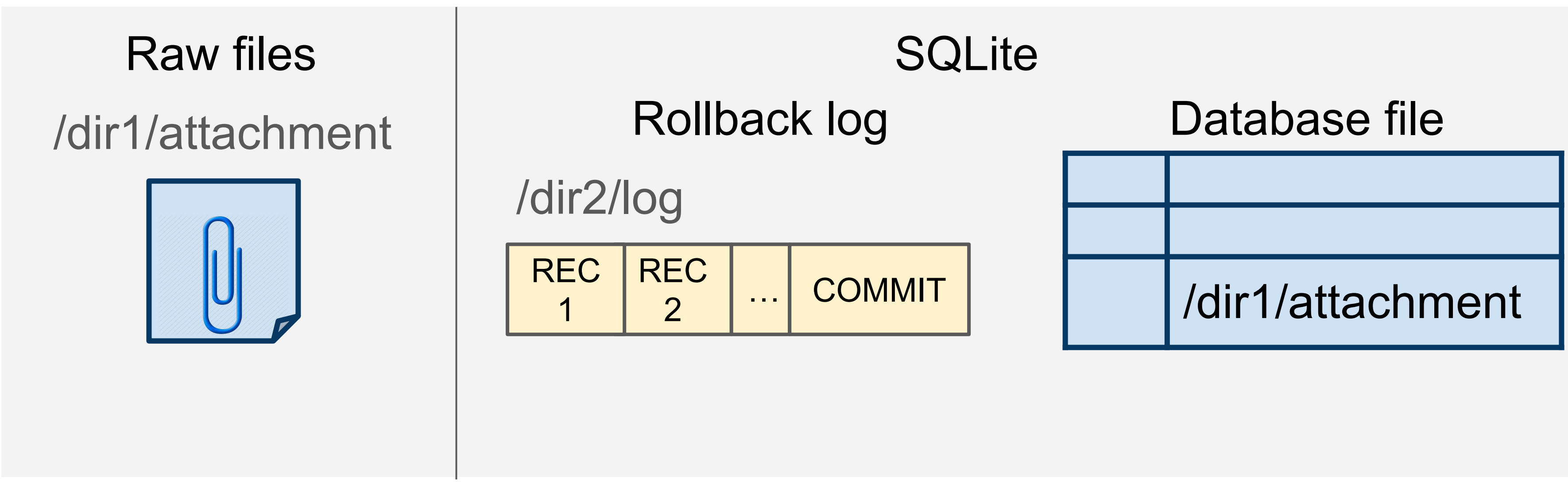
- Systems may fail in the middle of operations due to power loss or kernel bugs
- Crash consistency ensures that the application can recover to a correct state after a crash
- Applications store persistent state across multiple files and abstractions
  - Example: email attachment file and its path name stored in a SQLite database file become inconsistent on a crash
  - No POSIX mechanism to atomically update multiple files

Efficient crash consistency is hard

- Applications build on file-system primitives to ensure crash consistency
- Unfortunately, POSIX only provides the sync-family system calls, e.g., fsync()
  - fsync() forces dirty data associated with the file to become durable before the call returns
- fsync() is an expensive call
  - As a result, applications don’t use it as much as they should
- This results in **complex, error-prone applications** [OSDI 14]

Example: Android mail client

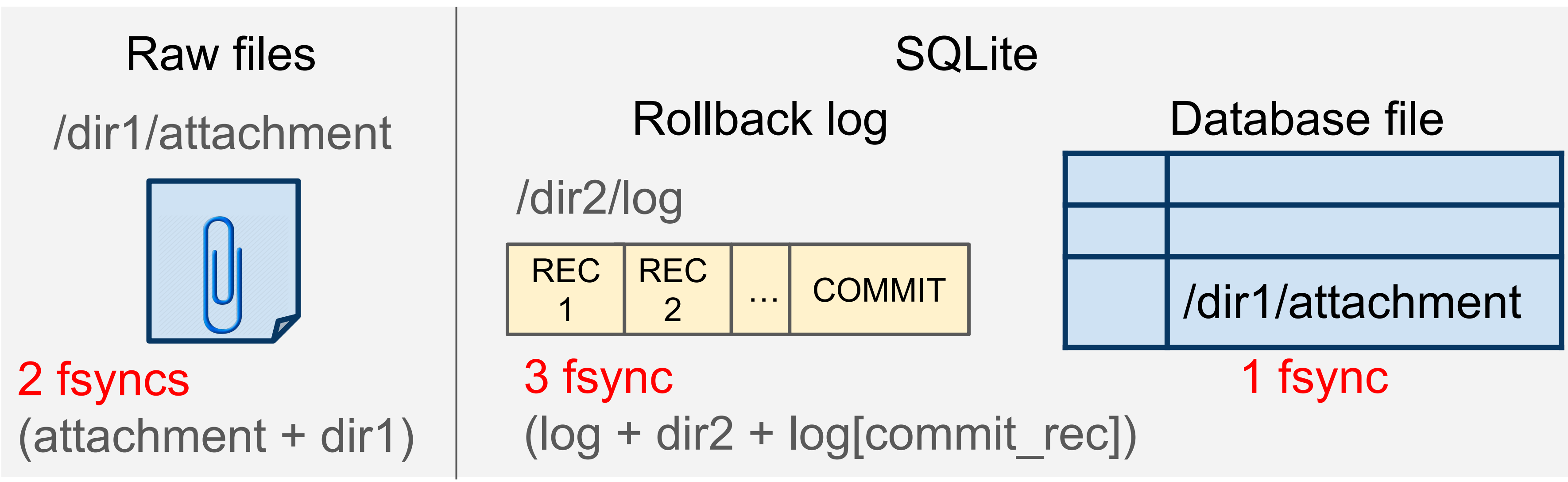
- The Android mail client receives an email with attachment
  - Stores attachment as a regular file
  - File name of attachment stored in SQLite
  - Stores email text in SQLite



Example: Android mail client

- The Android mail client receives an email with attachment
  - Stores attachment as a regular file
  - File name of attachment stored in SQLite
  - Stores email text in SQLite

Doing this safely requires 6 fsyncs!



File creation/deletion needs fsync on parent directory

System support for transactions

- POSIX lacks an efficient atomic update to multiple files
  - E.g., the attachment file and the two database-related files
- Sync and redundant writes lead to poor performance.

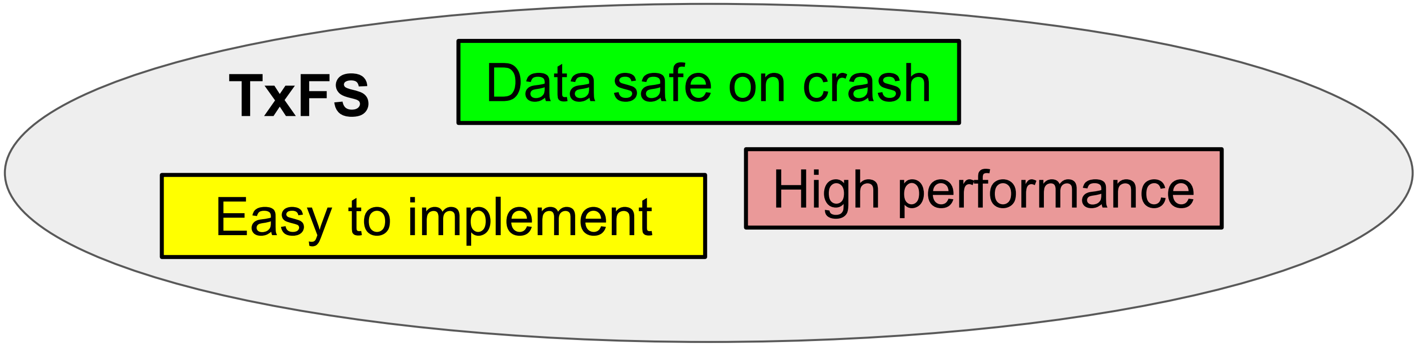
The file system should provide transactional services!

Didn’t transactional file systems fail?

- Complex implementation
  - Transactional OS: QuickSilver [TOCS 88], TxOS [SOSP 09] (**10k LOC**)
  - In-kernel transactional file systems: Valor [FAST 09]
- Hardware dependency
  - CFS [ATC 15], MARS [SOSP 13], TxFLash [OSDI 08], Isotope [FAST 16]
- Performance overhead
  - Valor [FAST 09] (**35% overhead**).
- Hard to use
  - Windows NTFS (TxF), released 2006 (deprecated 2012)

TxFS: Texas Transactional File System

- Reuse file-system journal for atomicity, consistency, durability
  - Well-tested code, reduces implementation complexity
- Develop techniques to isolate transactions
  - Customize techniques to kernel-level data structures
- Simple API - one syscall to **begin/end/abort** a transaction
  - Once TX begins, all file-system operations included in transaction

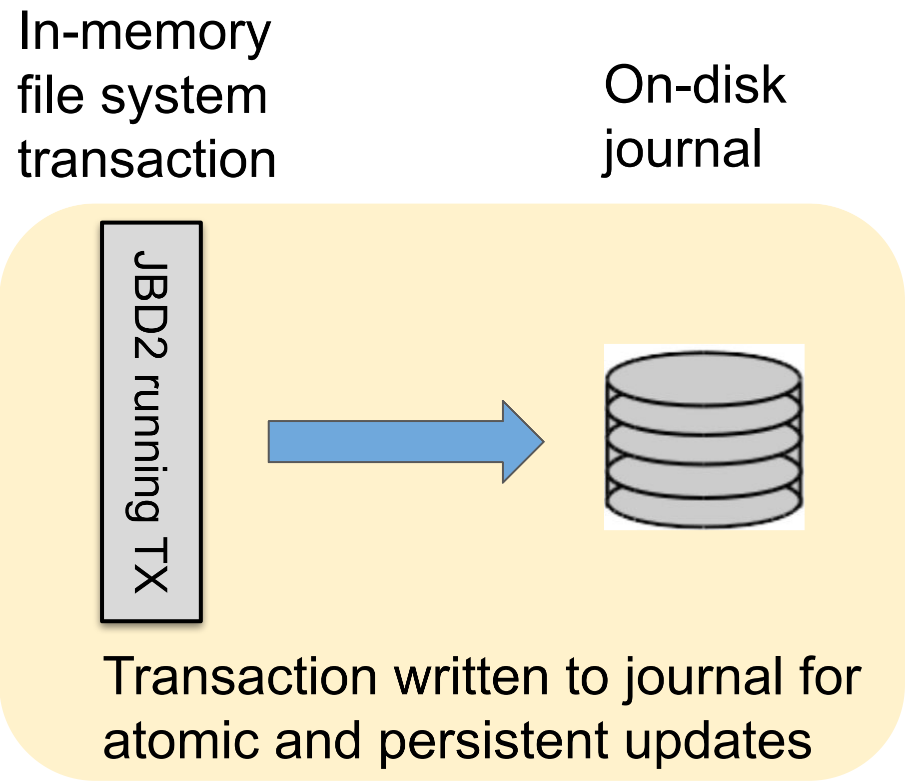


Outline

- Using the file-system journal for A, C, and D
- Implementing isolation
  - Avoid false conflicts on global data structures
  - Customize conflict detection for kernel data structures
- Using transactions to implement file-system optimizations
- Evaluating TxFS

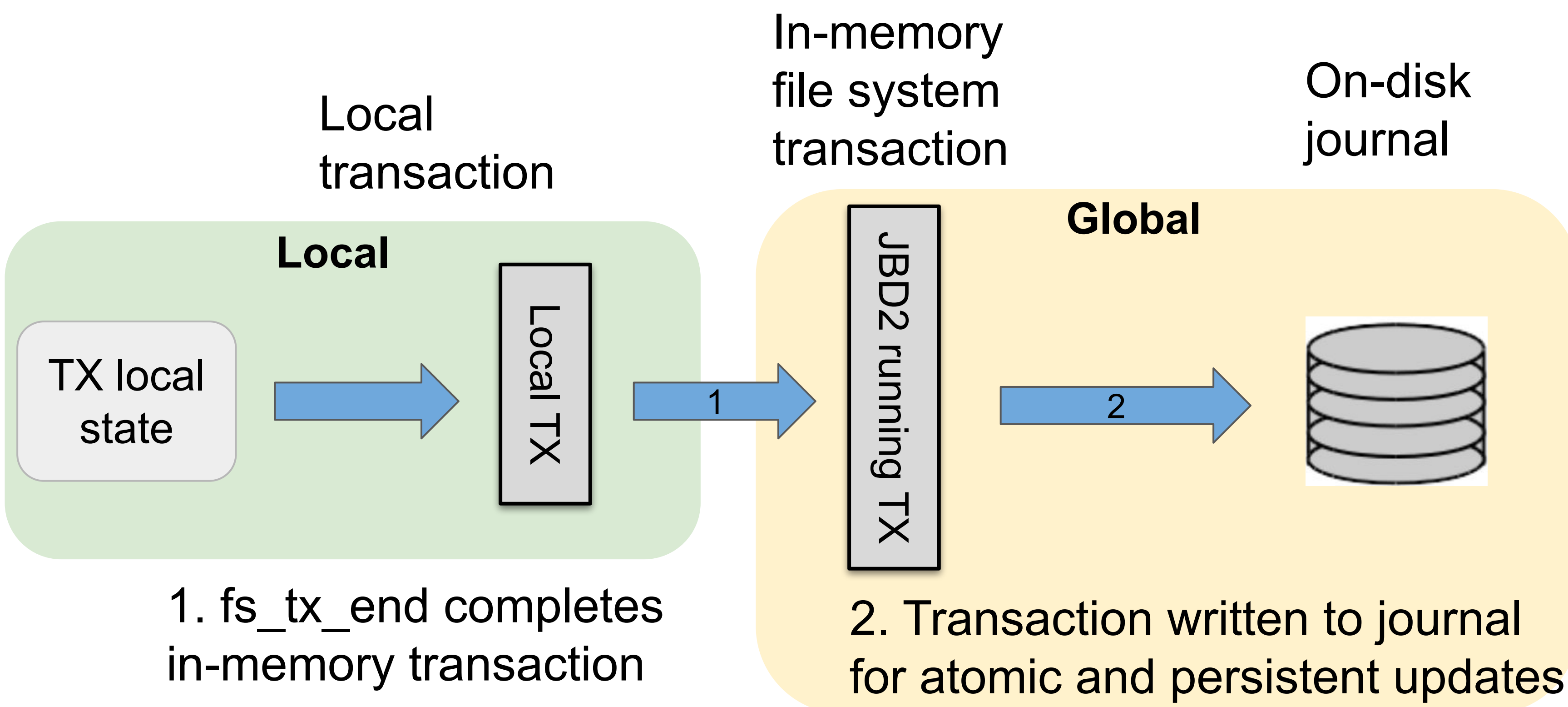
Atomicity, consistency and durability

- File systems already have a log that TxFS can reuse
  - E.g., ext4 journal is a write-ahead log (JBD2 layer)



Atomicity, consistency and durability

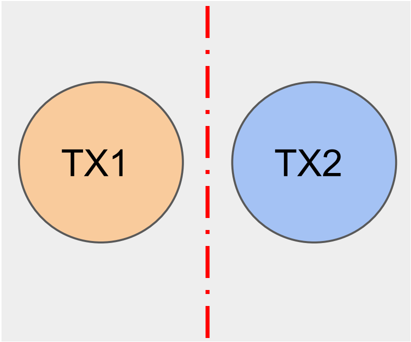
- Decreased complexity: use the file system's crash consistency mechanism to create transactions



Outline

- Using the file-system journal for A, C and D
- Implementing isolation
  - Avoid false conflicts on global data structures
  - Customize conflict detection for kernel data structures
- Using transactions to implement file-system optimizations
- Evaluating TxFS

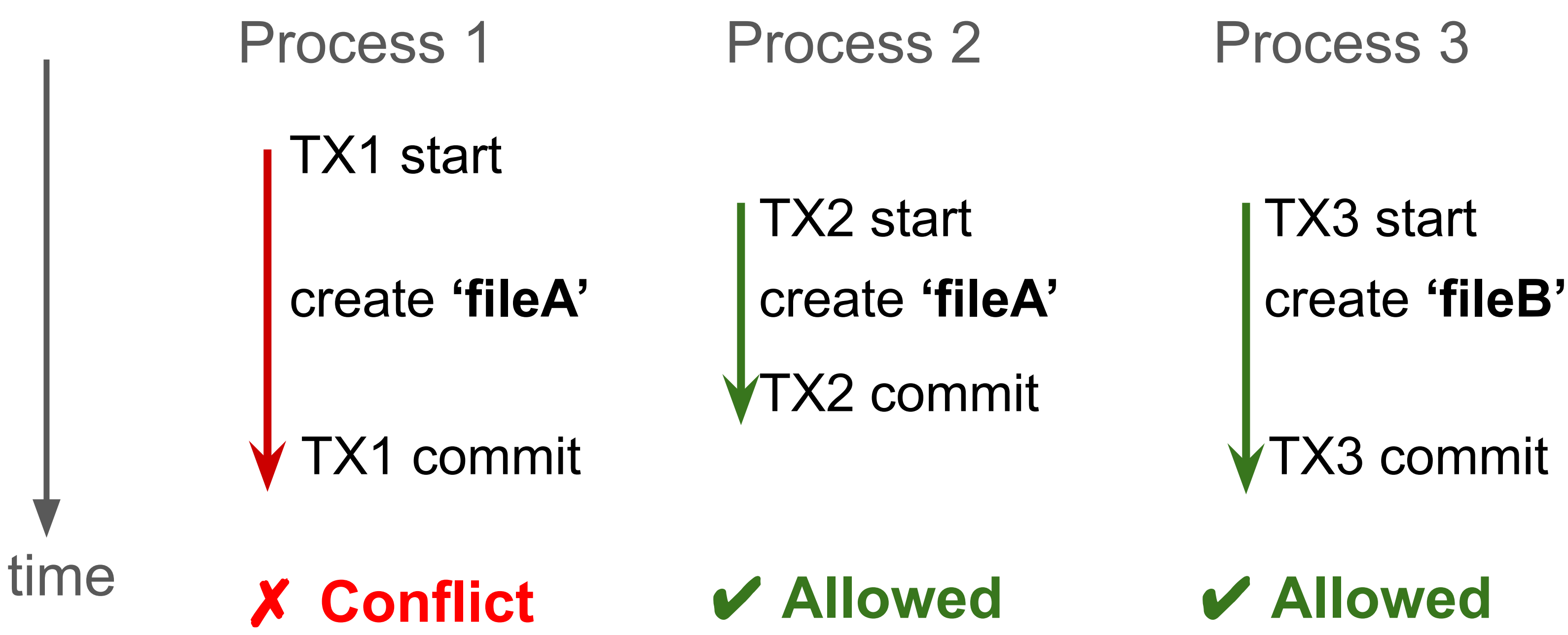
Isolation with performance



- Isolation - concurrent transactions act as if serially executed
  - At the level of repeatable reads
- Transaction-private copies
  - In-progress writes are local to a kernel thread
- Detect conflicts
  - Efficiently specialized to kernel data structure
- Maintain high performance
  - Fine-grained page locks
  - Avoid false conflicts

Challenge of isolation: Concurrency and performance

- Concurrent creation of the same file name is a conflict
- Writes to global data structures (e.g. bitmaps) should proceed

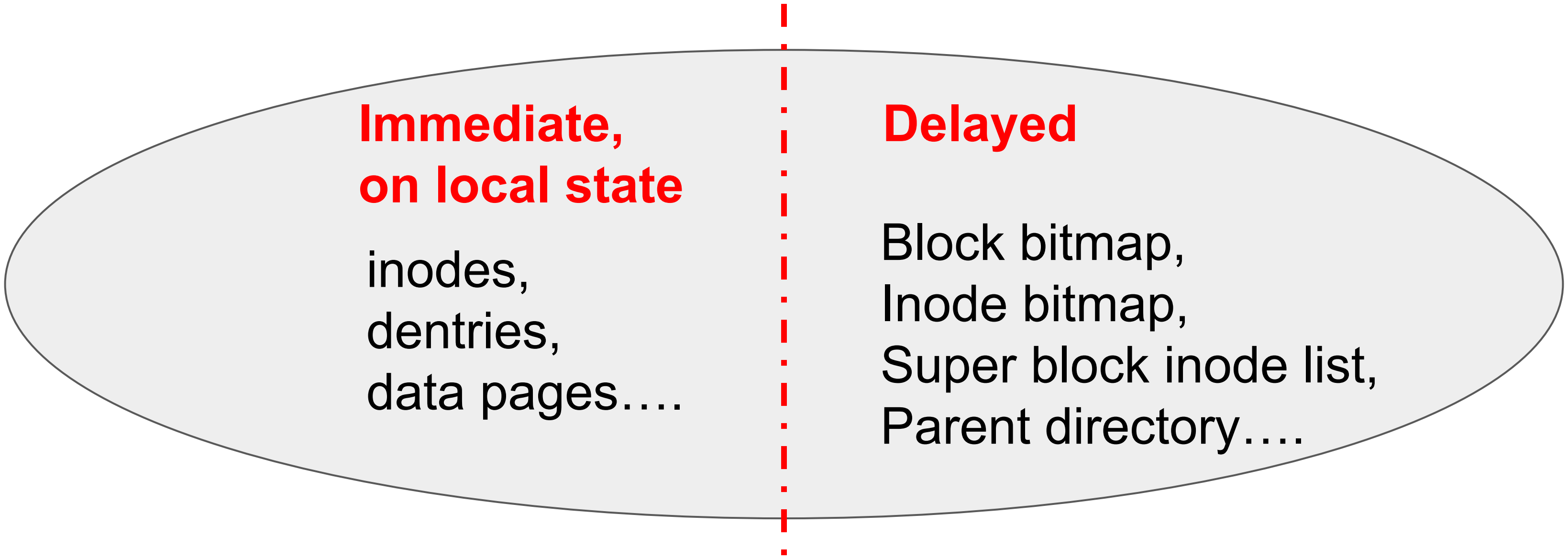


Avoid false conflicts on global data structures

- Two classes of file system functions
  - Operations that modify locally visible state

    - Executed immediately on private data structure copies
  - Operations that modify global state

    - Delayed until commit point

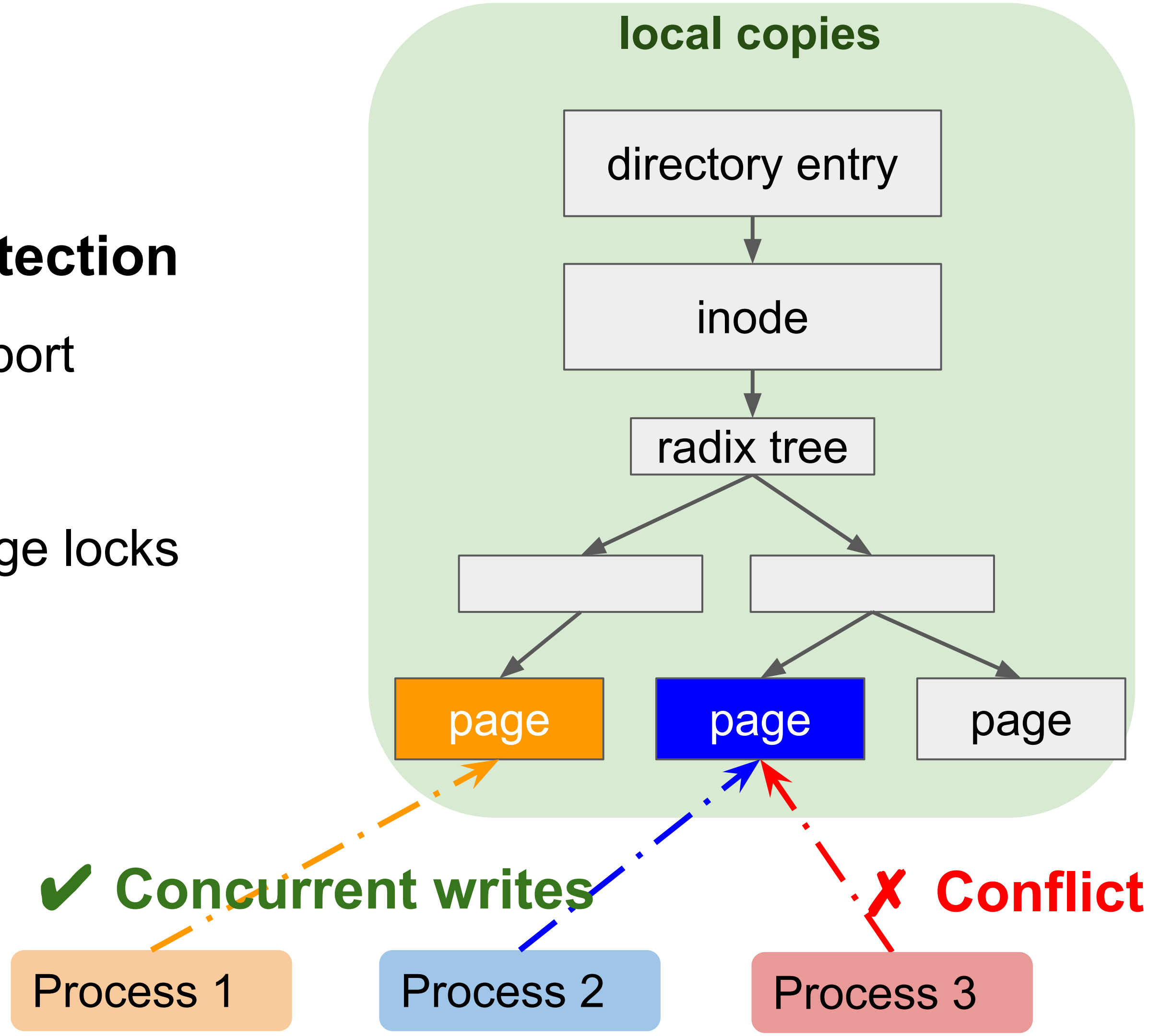


Customize isolation to each data structure

- Data pages
  - Unified API within file system code
  - Easy to differentiate read/write access
  - **Copy-on-write & eager conflict detection**
- inodes and directory entries (dentries)
  - Accessed haphazardly within file system code
  - Hard to differentiate read/write access
  - **Copy-on-read & lazy conflict detection (at commit time)**

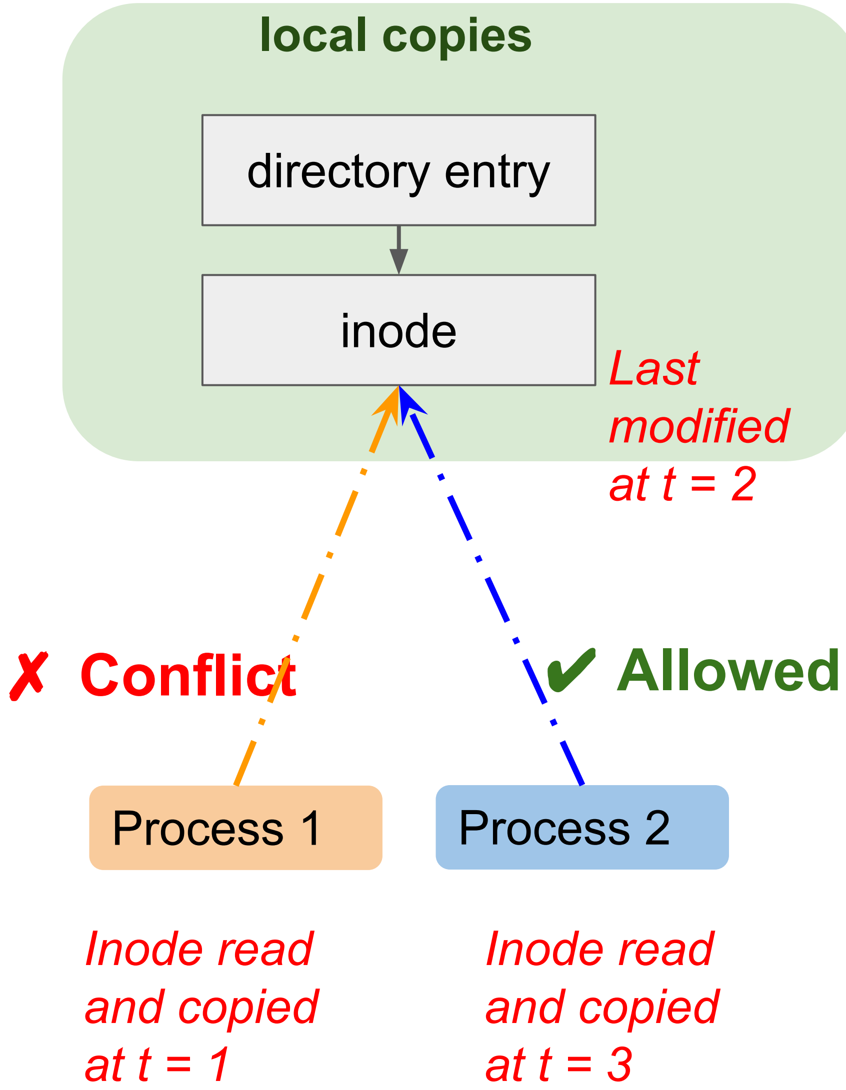
Page isolation

- **Copy-on-write**
- **Eager conflict detection**
  - Enables early abort
- Higher scalability
  - Fine-grained page locks

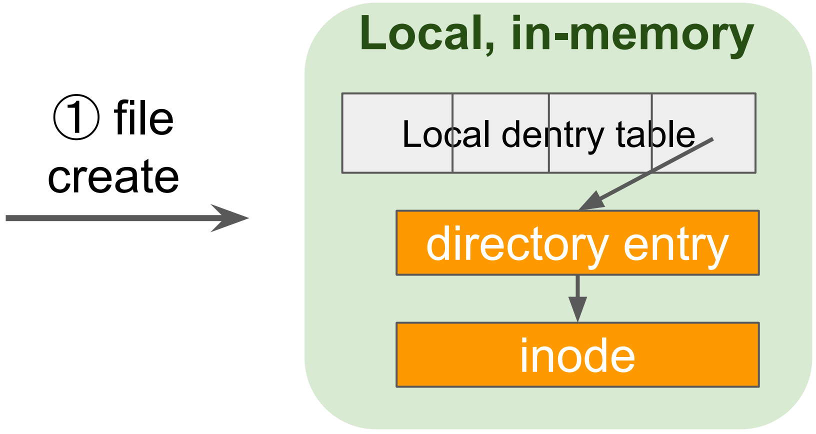


Inode & dentry isolation

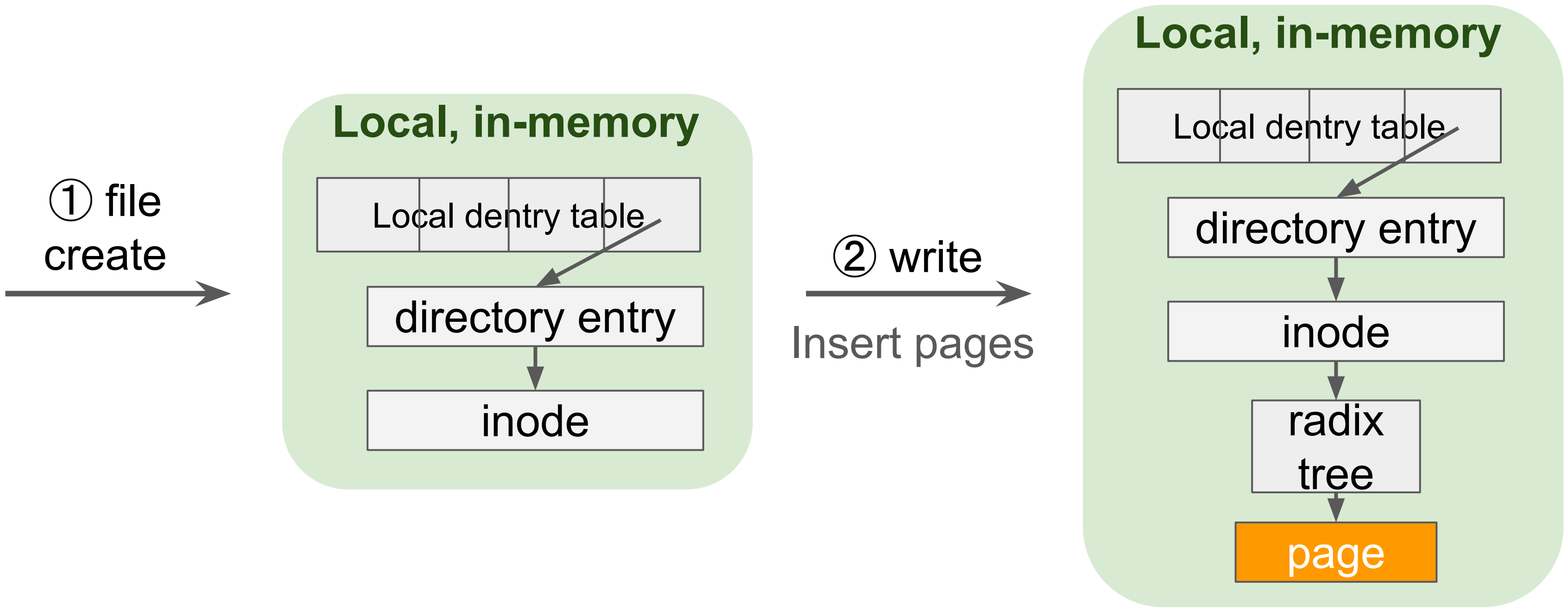
- **Copy-on-read**
- **Lazy conflict detection**
  - Timestamp-based conflict resolution
  - Necessary due to kernel's haphazard updates



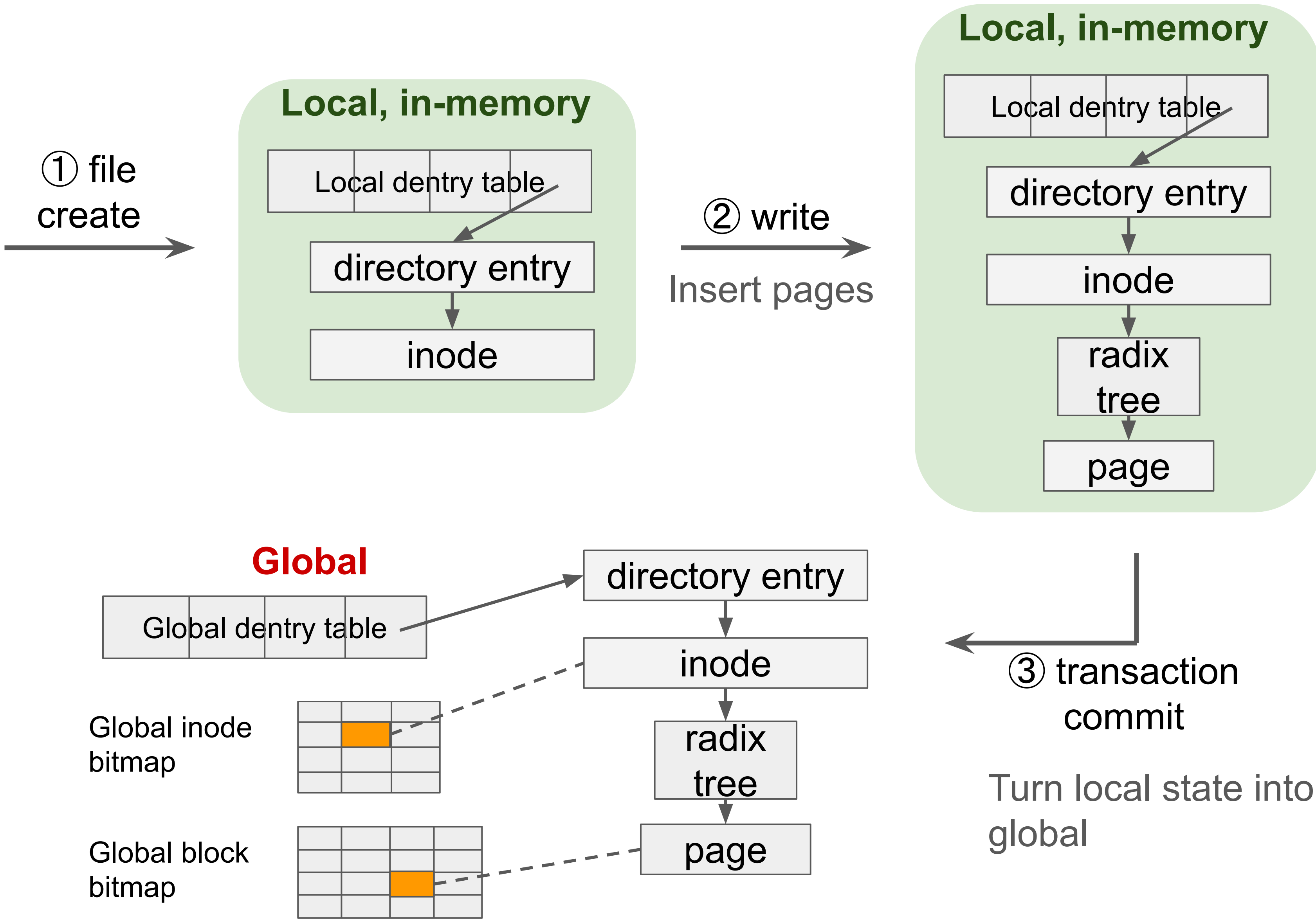
Example: file creation



Example: file creation

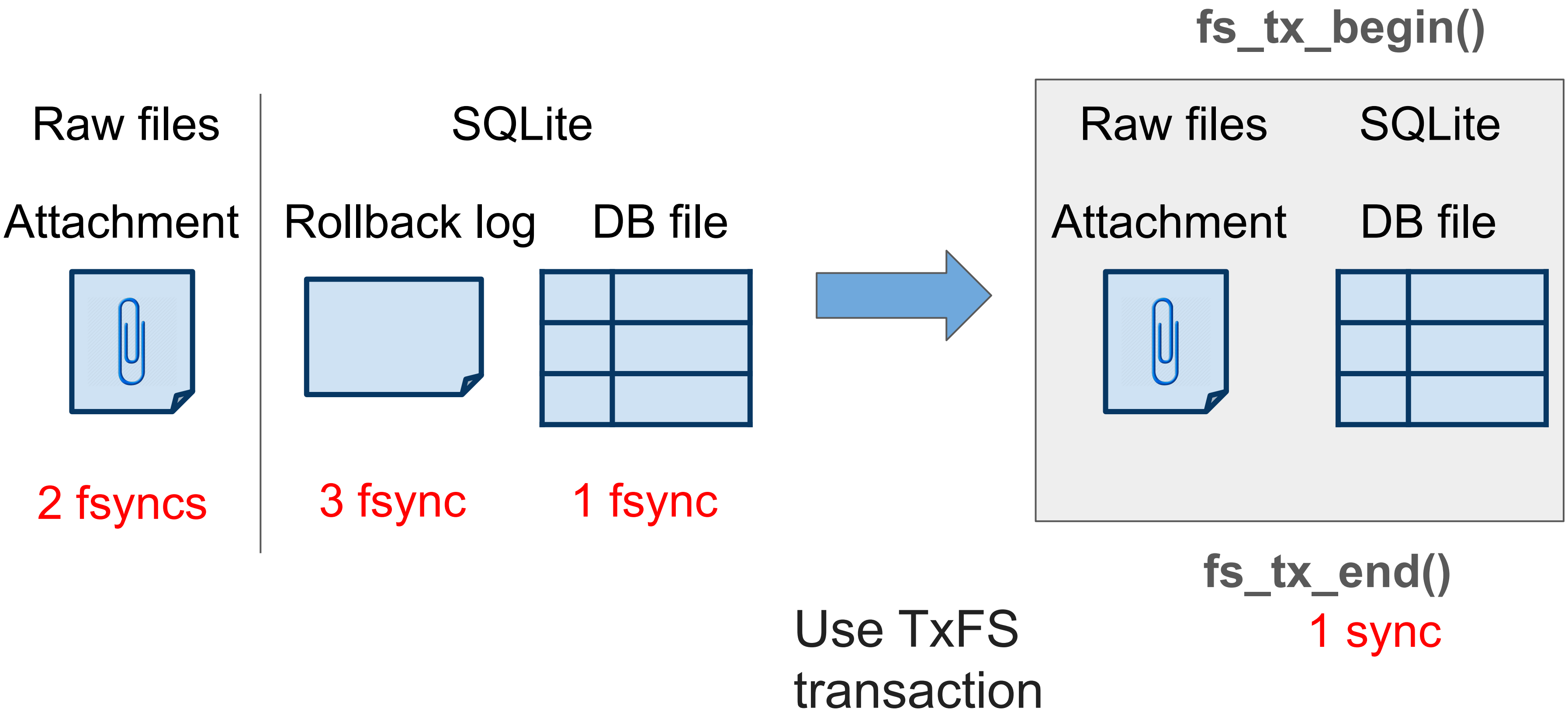


Example: file creation



TxFS API: Cross-abstraction transactions

- Modify the Android mail application to use TxFS transactions.

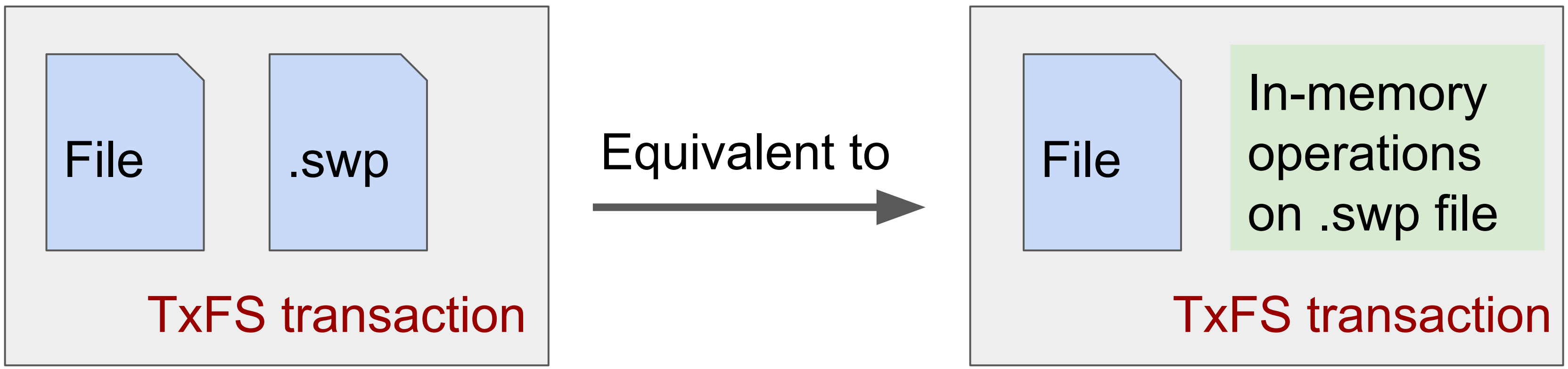


Outline

- Using the file-system journal for A, C and D
- Implementing isolation
  - Avoid false conflicts on global data structures
  - Customize conflict detection for kernel data structures
- Using transactions to implement file-system optimizations
- Evaluating TxFS

Transactions as a foundation for other optimizations

- Transactions present batched work to file system
  - Group commit
  - Eliminate temporary durable files
- Transactions allow fine-grained control of durability
  - Separate ordering from durability (osync [SOSP 13])



Example: Eliminate temporary durable files in Vim

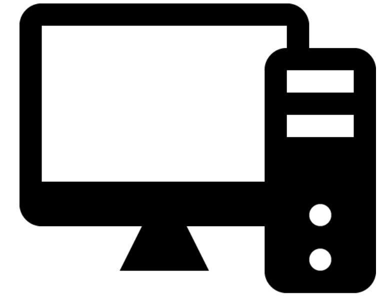
Implementation

- Linux kernel version 3.18.22
- Lines of code for implementation

Reusable code

Part	Lines of code
TxFS internal bookkeeping	1,300
Virtual file system (VFS)	1,600
Journal (JBD2)	900
Ext4	1,200
Total	5,200

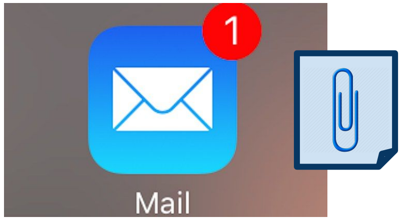
Evaluation: configuration



- Software
  - OS: Ubuntu 16.04 LTS (Linux kernel 3.18.22)
- Hardware
  - 4 core Intel Xeon E3-1220 CPU, 32 GB memory
  - Storage: Samsung 850 (250 GB) SSD

Experiment	**TxFS benefit**	**Speedup**
Single-threaded SQLite	Less IO & sync, batching	1.31x
TPC-C	Less IO & sync, batching	1.61x
Android Mail	Cross abstraction	2.31x
Git	Crash consistency	1.00x

Microbenchmark: Android mail client



- Eliminating logging IO

```
/* Write attachment */
open(/dir/attachment)
write(/dir/attachment)
fsync(/dir/attachment)
fsync(/dir/)
/* Update database */
open(/dir/journal)
write(/dir/journal)
fsync(/dir/journal)
fsync(/dir/)
write(/dir/db)
fsync(/dir/db)
unlink(/dir/journal)
fsync(/dir/)
```

```
fs_tx_begin()
/* Write attachment */
open(/dir/attachment)
write(/dir/attachment)
fsync(/dir/attachment)
fsync(/dir/)
/* Update database */
open(/dir/journal)
write(/dir/journal)
fsync(/dir/journal)
fsync(/dir/)
write(/dir/db)
fsync(/dir/db)
unlink(/dir/journal)
fsync(/dir/)
fs_tx_end()
```

```
fs_tx_begin()

/* Write attachment */
open(/dir/attachment)
write(/dir/attachment)

/* Update database */
write(/dir/db)

fs_tx_end()
```

Wrap with transaction:
20% throughput increase

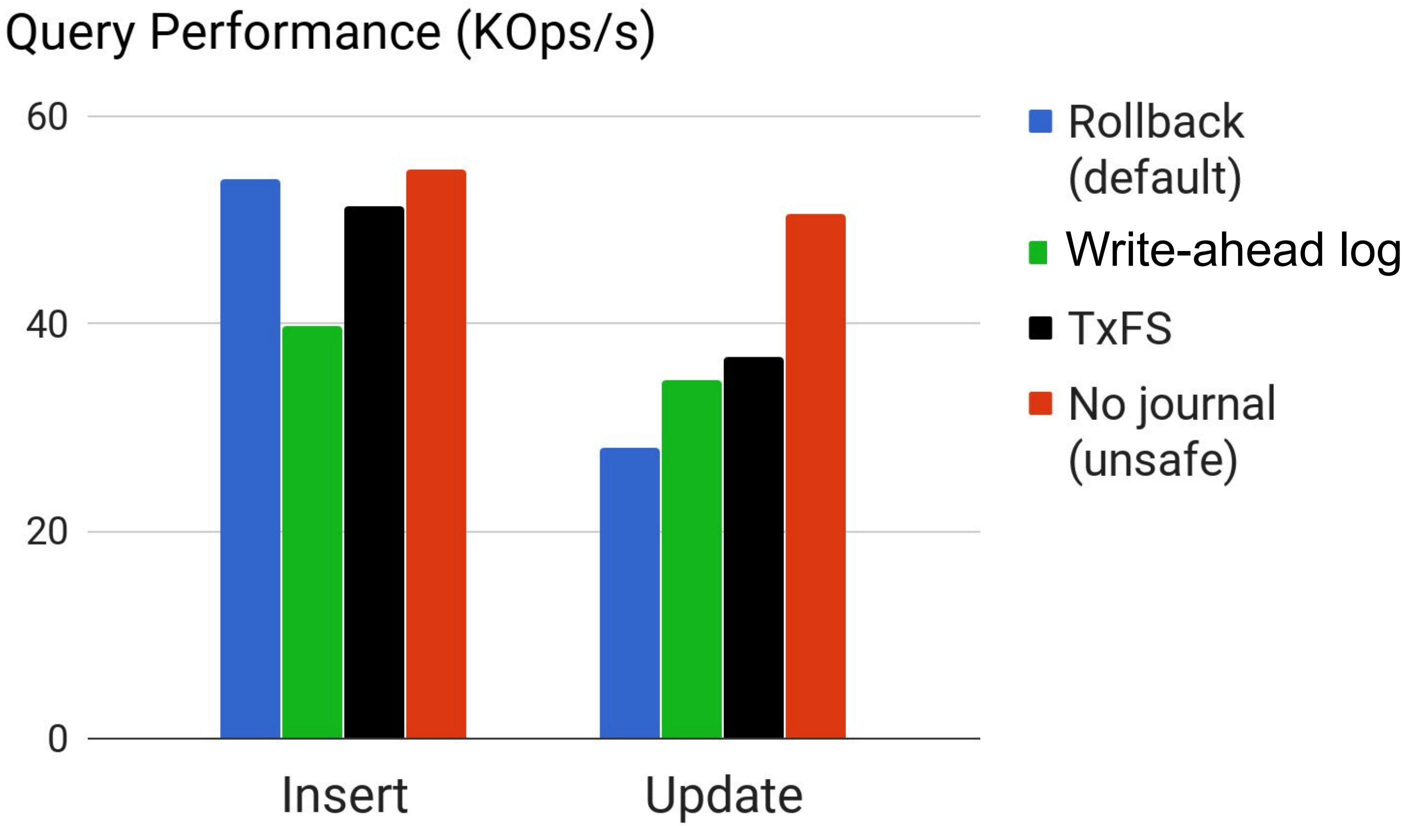
Manual rewrite:
55% throughput increase

Git - consistency w/o overhead

- On a crash, git is vulnerable to garbage files and corruption
  - Currently, no fsync() to order operations (for high performance)
  - Possible loss of working tree, not recoverable with git-fsck
- TxFS transactions make Git fast and safe
  - No garbage files nor data corruption on crash
  - No observable performance overhead

Workload running in a VM: initialize a Git repository; git-add 20,000 empty files; crash at different vulnerable points

Evaluation: single-threaded SQLite



1.5M 1KB operations. 10K operations grouped in a transaction. Database prepopulated with 15M rows.

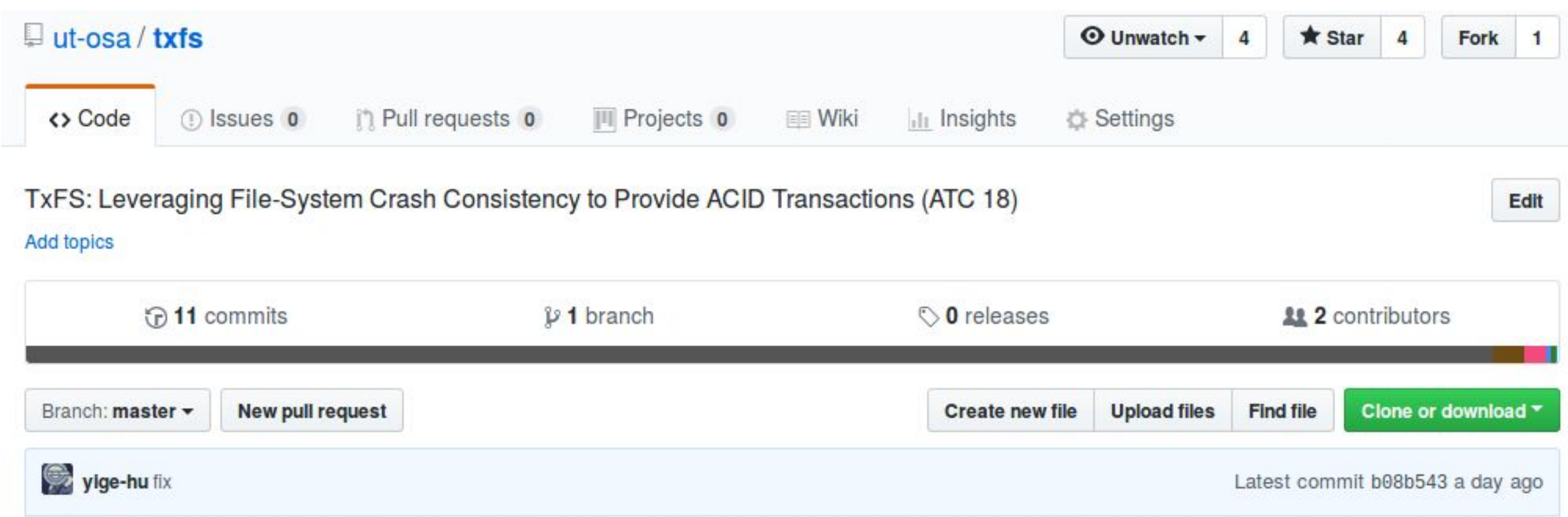
TxFS Summary

Data safe on crash

Easy to implement

High performance

- Persistent data is structured; tough to make crash consistent
- Transactions make applications simpler, more efficient
  - They enable optimizations that reduce IO and system calls
- File-system journal makes implementing transactions easier
- Source code: https://github.com/ut-osa/txfs



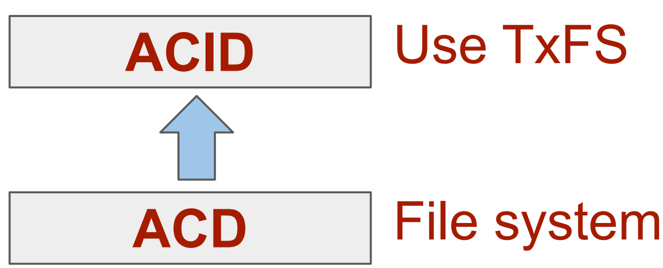
Thank you!

Limitations

- Do not support directory operations
- Do not support transactions across file systems
- Memory copy overhead in read-only transactions
- Transaction size limited by memory and on-disk journal size

Evaluation: correctness

- Stress tests
- Crash consistency
  - Boot a virtual machine and creates many types of transactions in multiple threads with random amounts of contained work and conflict probabilities
  - Crash the VM at a random time
  - Check if the file system journal is recoverable, and the file system passes all fsck checks



Prior works

Category	System	Isolation	Easy-to-use APIs	Hardware independence	Performance	Complexity
In-kernel transactional FS	**TxFS**	✔	✔	✔	H	L
	Valor	✔	❌	✔	H	L
	TxF	✔	❌	✔	H	H
Transactional OS	TxOS	✔	✔	✔	H	H
FS over userspace databases	OdeFS	Relying on DBs	❌	✔	L	L
	Inversion					
	DBFS					
	Amino					
Transactional storage	CFS	❌	✔	❌	H	L
	MARS	✔	❌	❌	H	H
	Isotope	✔	✔	✔	H	H
Failure atomicity	msync	❌	✔	✔	H	L
	AdvFS	❌	✔	✔	H	L

The table compares prior work providing ACID transactions or failure atomicity in a local file system. Legend: ✔- supported, ❌- unsupported, L - Low, H - High. Note that only TxFS provides isolation and durability with high performance and low implementation complexity without restrictions or hardware modifications.