SquirrelFS: using the Rust compiler to check file-system crash consistency

Hayley LeBlanc, Nathan Taylor,
James Bornholt, Vijay Chidambaram

TEXAS
The University of Texas at Austin

Current approaches to ensuring crash consistency

Current approaches to ensuring crash consistency

Testing

eXplode (OSDI ‘06), Yat (ATC ‘14),
CrashMonkey (OSDI ‘18), Hydra (SOSP ‘19),
Vinter (ATC ‘22), Chipmunk (EuroSys ‘23) . . .

Current approaches to ensuring crash consistency

Testing

eXplode (OSDI ‘06), Yat (ATC ‘14),
CrashMonkey (OSDI ‘18), Hydra (SOSP ‘19),
Vinter (ATC ‘22), Chipmunk (EuroSys ‘23) . . .

- Incomplete
- Requires specialized tools

Current approaches to ensuring crash consistency

Testing

Verification

eXplode (OSDI ‘06), Yat (ATC ‘14),
CrashMonkey (OSDI ‘18), Hydra (SOSP ‘19),
Vinter (ATC ‘22), Chipmunk (EuroSys ‘23) . . .

FSCQ (SOSP ‘15), DFSCQ (SOSP ‘17),
Perennial (SOSP ‘19), VeriBetrKV (OSDI ‘20) . . .

- Incomplete
- Requires specialized tools

Current approaches to ensuring crash consistency

Testing

Verification

eXplode (OSDI ‘06), Yat (ATC ‘14),
CrashMonkey (OSDI ‘18), Hydra (SOSP ‘19),
Vinter (ATC ‘22), Chipmunk (EuroSys ‘23) . . .

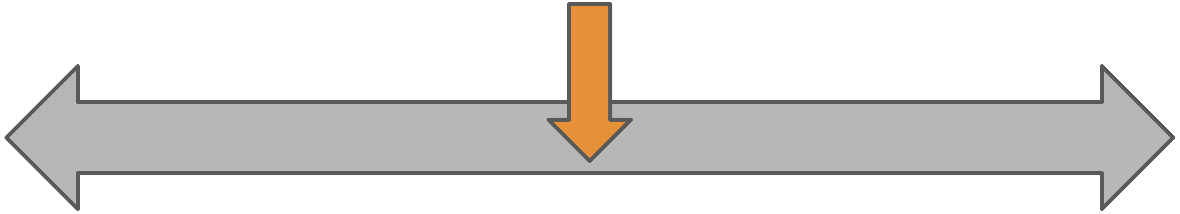
- Incomplete
- Requires specialized tools

FSCQ (SOSP ‘15), DFSCQ (SOSP ‘17),
Perennial (SOSP ‘19), VeriBetrKV (OSDI ‘20) . . .

- Requires specialized expertise
- Development takes longer
- Often impacts performance

Current approaches to ensuring crash consistency

Testing



Verification

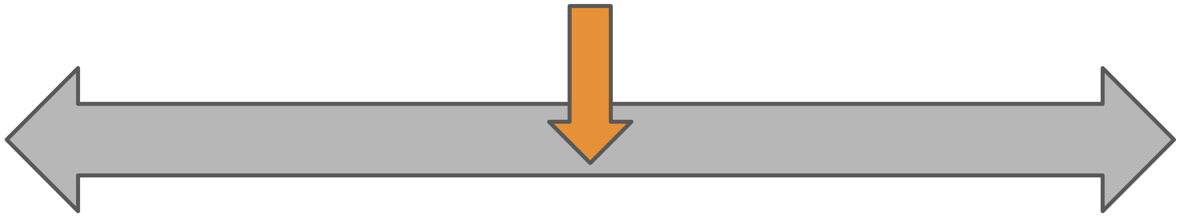
eXplode (OSDI '06), Yat (ATC '14),
CrashMonkey (OSDI '18), Hydra (SOSP '19),
Vinter (ATC '22), Chipmunk (EuroSys '23) . . .

FSCQ (SOSP '15), DFSCQ (SOSP '17),
Perennial (SOSP '19), VeriBetrKV (OSDI '20) . . .

- Incomplete
- Requires specialized tools

- Requires specialized expertise
- Development takes longer
- Often impacts performance

Current approaches to ensuring crash consistency



Testing

Verification

eXplode (OSDI ‘06), Yat (ATC ‘14),
CrashMonkey (OSDI ‘18), Hydra (SOSP ‘19),
Vinter (ATC ‘22), Chipmunk (EuroSys ‘23) . . .

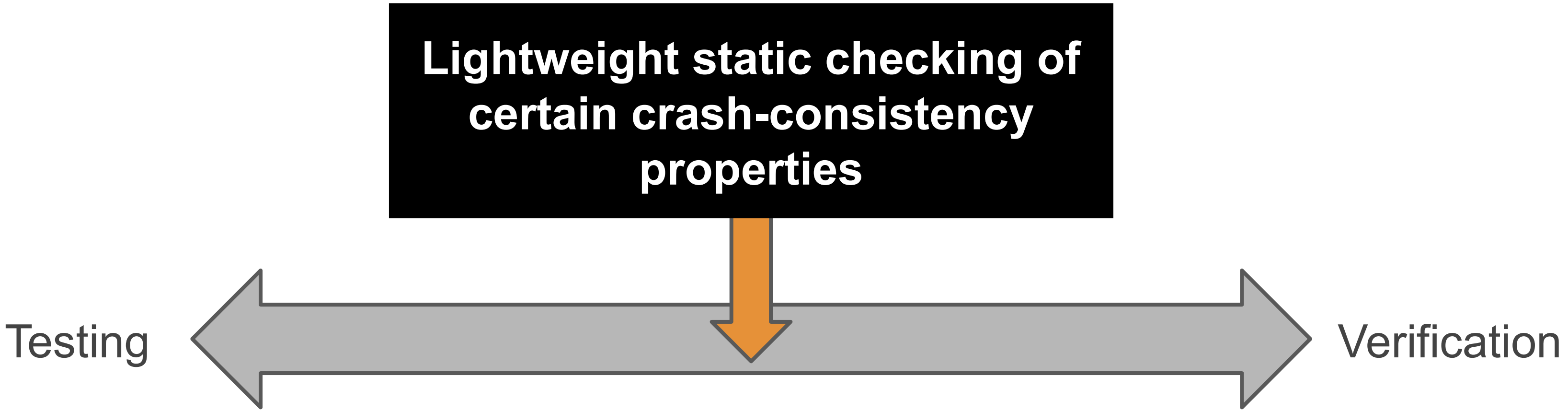
FSCQ (SOSP ‘15), DFSCQ (SOSP ‘17),
Perennial (SOSP ‘19), VeriBetrKV (OSDI ‘20) . . .

- Incomplete
- Requires specialized tools

- No specialized expertise or tools required
- Statically check critical properties

- Requires specialized expertise
- Development takes longer
- Often impacts performance

Current approaches to ensuring crash consistency


Lightweight static checking of certain crash-consistency properties
Testing
Verification

eXplode (OSDI ‘06), Yat (ATC ‘14), CrashMonkey (OSDI ‘18), Hydra (SOSP ‘19), Vinter (ATC ‘22), Chipmunk (EuroSys ‘23) . . .

FSCQ (SOSP ‘15), DFSCQ (SOSP ‘17), Perennial (SOSP ‘19), VeriBetrKV (OSDI ‘20) . . .

- Incomplete
- Requires specialized tools

- No specialized expertise or tools required
- Statically check critical properties

- Requires specialized expertise
- Development takes longer
- Often impacts performance

Rust programming language

Rust programming language

High-performance, low-level systems programming language

Rust programming language

High-performance, low-level systems programming language

Strong type system that can statically prevent:

Rust programming language

High-performance, low-level systems programming language

Strong type system that can statically prevent:

- Data races

Rust programming language

High-performance, low-level systems programming language

Strong type system that can statically prevent:

- Data races
- Memory safety issues

Rust programming language

High-performance, low-level systems programming language

Strong type system that can statically prevent:

- Data races
- Memory safety issues
- **Some crash-consistency bugs!** (Corundum ASPLOS ‘21)

Rust programming language

High-performance, low-level systems programming language

Strong type system that can stati

- Data races
- Memory safety issues
- **Some crash-consistency bugs!** (Corundum ASPLOS '21)

Rust crate for
persistent memory
storage systems

Rust programming language

High-performance, low-level systems programming language

Strong type system that can stati

- Data races
- Memory safety issues
- **Some crash-consistency bugs!** (Corundum ASPLOS ‘21)

Rust crate for **persistent memory** storage systems

Statically checks **low-level** crash-consistency properties

Rust programming language

High-performance, low-level systems programming language

Strong type system that can stati

- Data races
- Memory safety issues
- **Some crash-consistency bugs!** (Corundum ASPLOS ‘21)

Rust crate for **persistent memory** storage systems

Statically checks **low-level** crash-consistency properties

This work: use Rust to statically check **higher-level** crash-consistency properties in a persistent memory file system

Rust programming language

High-performance, low-level systems programming language

Strong type system that can stati

- Data races
- Memory safety issues
- **Some crash-consistency bugs!** (Corundum ASPLOS ‘21)

Rust crate for **persistent memory** storage systems

Statically checks **low-level** crash-consistency properties

This work: use Rust to statically check **higher-level** crash-consistency properties in a persistent memory file system

Atomicity of system calls

SquirrelFS

SquirrelFS

Persistent memory file system with **statically-checked ordering-related crash-consistency properties**

SquirrelFS

Persistent memory file system with **statically-checked ordering-related crash-consistency properties**

Static checks rely only on existing Rust features

SquirrelFS

Persistent memory file system with **statically-checked ordering-related crash-consistency properties**

Static checks rely only on existing Rust features

Introduces **Synchronous Soft Updates** crash-consistency mechanism

SquirrelFS

Persistent memory file system with **statically-checked ordering-related crash-consistency properties**

Static checks rely only on existing Rust features

Introduces **Synchronous Soft Updates** crash-consistency mechanism

Uses the **typestate pattern** to statically check ordering of durable updates

SquirrelFS

Persistent memory file system with **statically-checked ordering-related crash-consistency properties**

Static checks rely only on existing Rust features

Introduces **Synchronous Soft Updates** crash-consistency mechanism

Uses the **typestate pattern** to statically check ordering of durable updates

Achieves similar or better performance to other PM file systems

SquirrelFS

Persistent memory file system with **statically-checked ordering-related crash-consistency properties**

Static checks rely only on existing Rust features

Introduces **Synchronous Soft Updates** crash-consistency mechanism

Uses the **typestate pattern** to statically check ordering of durable updates

Achieves similar or better performance to other PM file systems

https://github.com/utsaslab/squirrelfs

Roadmap

1. Introduction
2. Ordering for crash consistency
3. Typestate pattern
4. SquirrelFS implementation
5. Evaluation

Roadmap

1. Introduction
2. **Ordering for crash consistency**
3. Typestate pattern
4. SquirrelFS implementation
5. Evaluation

Ordering for crash consistency

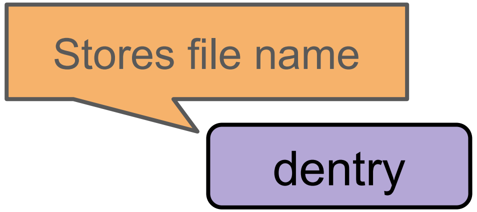
Crash consistency depends on the **order of durable updates** (Ganger & Patt ‘94, Frost et al. ‘07, Chidambaram et al. ‘12)

Simple example: creating a new file

Ordering for crash consistency

Crash consistency depends on the **order of durable updates** (Ganger & Patt '94, Frost et al. '07, Chidambaram et al. '12)

Simple example: creating a new file



Ordering for crash consistency

Crash consistency depends on the **order of durable updates** (Ganger & Patt ‘94, Frost et al. ‘07, Chidambaram et al. ‘12)

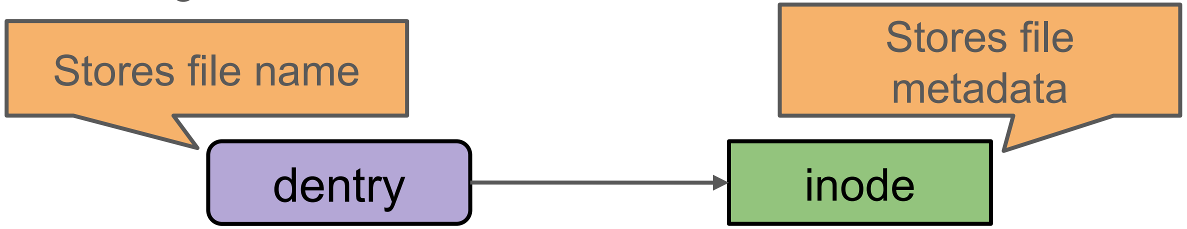
Simple example: creating a new file

Stores file name
dentry
Stores file metadata
inode

Ordering for crash consistency

Crash consistency depends on the **order of durable updates** (Ganger & Patt '94, Frost et al. '07, Chidambaram et al. '12)

Simple example: creating a new file



Ordering for crash consistency

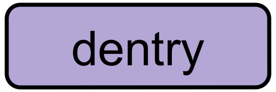
Crash consistency depends on the **order of durable updates** (Ganger & Patt ‘94, Frost et al. ‘07, Chidambaram et al. ‘12)

Simple example: creating a new file

Ordering for crash consistency

Crash consistency depends on the **order of durable updates** (Ganger & Patt ‘94, Frost et al. ‘07, Chidambaram et al. ‘12)

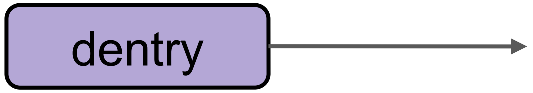
Simple example: creating a new file



Ordering for crash consistency

Crash consistency depends on the **order of durable updates** (Ganger & Patt ‘94, Frost et al. ‘07, Chidambaram et al. ‘12)

Simple example: creating a new file



Ordering for crash consistency

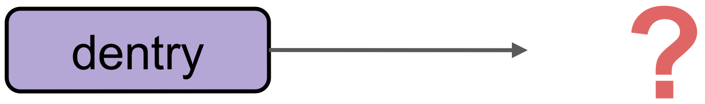
Crash consistency depends on the **order of durable updates** (Ganger & Patt ‘94, Frost et al. ‘07, Chidambaram et al. ‘12)

Simple example: creating a new file

Ordering for crash consistency

Crash consistency depends on the **order of durable updates** (Ganger & Patt ‘94, Frost et al. ‘07, Chidambaram et al. ‘12)

Simple example: creating a new file

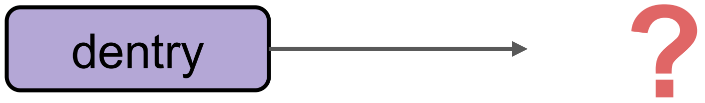


Setting dentry pointer depends on inode initialization

Ordering for crash consistency

Crash consistency depends on the **order of durable updates** (Ganger & Patt '94, Frost et al. '07, Chidambaram et al. '12)

Simple example: creating a new file



Setting dentry pointer depends on inode initialization

Statically enforcing durable update ordering can prevent many crash-consistency bugs

Soft updates

Track dependencies between durable in-place updates to enforce crash-consistent ordering (Ganger & Patt OSDI ‘94)

Soft updates

Track dependencies between durable in-place updates to enforce crash-consistent ordering (Ganger & Patt OSDI ‘94)

…[Soft updates] is a significant impediment to progressing in the vfs layer so we plan to get it out of the way. It is too clever for us to continue maintaining as it is.

Bob Beck, OpenBSD commit message, 2023

Soft updates

Track dependencies between durable in-place updates to enforce crash-consistent ordering (Ganger & Patt OSDI ‘94)

…[Soft updates] is a significant impediment to progressing in the vfs layer so we plan to get it out of the way. It is too clever for us to continue maintaining as it is.

Bob Beck, OpenBSD commit message, 2023

Tracking asynchronous dependencies

Soft updates

Track dependencies between durable in-place updates to enforce crash-consistent ordering (Ganger & Patt OSDI ‘94)

…[Soft updates] is a significant impediment to progressing in the vfs layer so we plan to get it out of the way. It is too clever for us to continue maintaining as it is.

Bob Beck, OpenBSD commit message, 2023

Tracking asynchronous dependencies

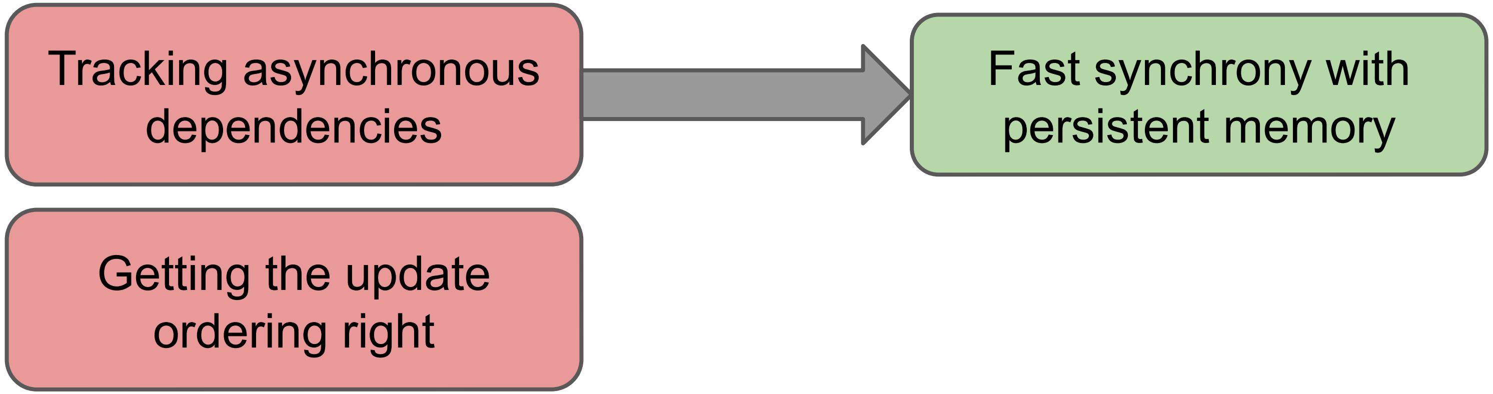
Getting the update ordering right

Soft updates

Track dependencies between durable in-place updates to enforce crash-consistent ordering (Ganger & Patt OSDI ‘94)

…[Soft updates] is a significant impediment to progressing in the vfs layer so we plan to get it out of the way. It is too clever for us to continue maintaining as it is.

Bob Beck, OpenBSD commit message, 2023


Tracking asynchronous dependencies
Getting the update ordering right
Fast synchrony with persistent memory

Soft updates

Track dependencies between durable in-place upda

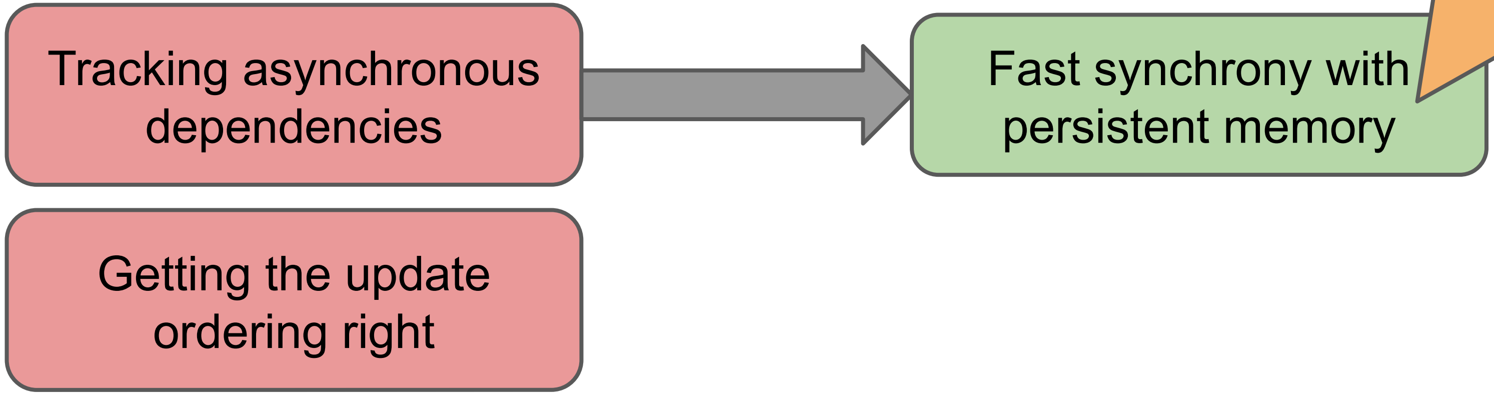
ordering (Ganger & Patt OSDI ‘94)

…[Soft updates] is a significant impediment
layer so we plan to get it out of the way. It is to
maintaining as it is.

Bob Beck, Ope

Low-latency durable storage devices with DRAM-like interface

- Intel Optane DC PM
- Battery-backed DRAM
- Future CXL-attached mem


Tracking asynchronous dependencies
Getting the update ordering right
Fast synchrony with persistent memory

Soft updates

Track dependencies between durable in-place upda

ordering (Ganger & Patt OSDI '94)

…[Soft updates] is a significant impediment

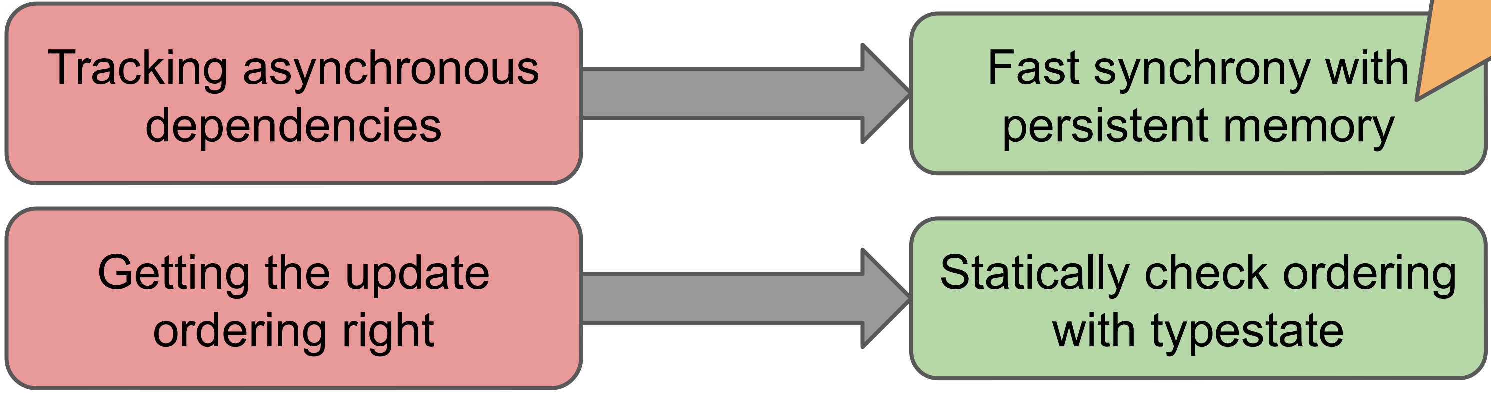
layer so we plan to get it out of the way. It is to

maintaining as it is.

Bob Beck, Ope

Low-latency durable storage devices with DRAM-like interface

- Intel Optane DC PM
- Battery-backed DRAM
- Future CXL-attached mem


Tracking asynchronous dependencies
Fast synchrony with persistent memory
Getting the update ordering right
Statically check ordering with typestate

Roadmap

The typestate pattern

Encode runtime state in an object’s type with no runtime overhead

The typestate pattern

Encode runtime state in an object’s type with no runtime overhead

```
fn init_inode(i: &mut Inode)
```

The typestate pattern

Encode runtime state in an object’s type with no runtime overhead

```
fn init_inode(i: Inode<Free>)
    -> Inode<Init>
```

The typestate pattern

Encode runtime state in an object’s type with no runtime overhead

```
fn init_inode(i: Inode<Free>)
     -> Inode<Init>
```

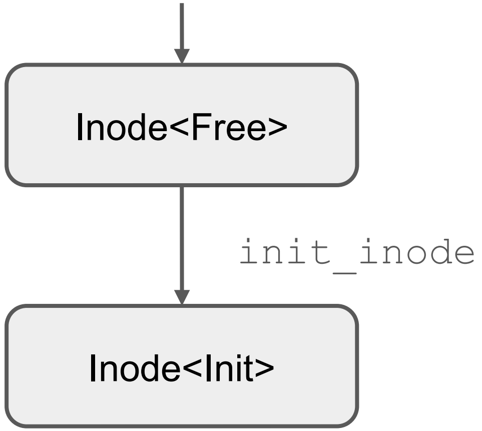
Consumes input state and returns new state

The typestate pattern

Encode runtime state in an object’s type with no runtime overhead

```
fn init_inode(i: Inode<Free>)
    -> Inode<Init>
```

Consumes input state and returns new state



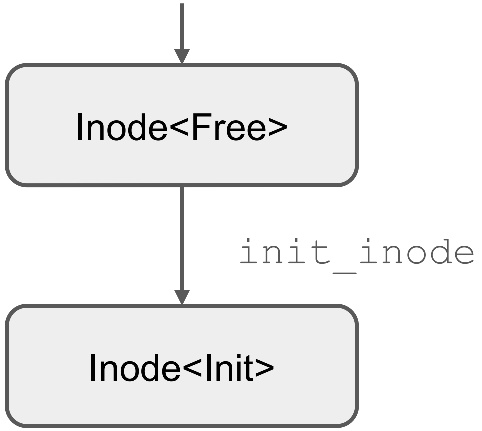
The typestate pattern

Encode runtime state in an object’s type with no runtime overhead

```
fn init_inode(i: Inode<Free>)
    -> Inode<Init>
```

Consumes input state and returns new state

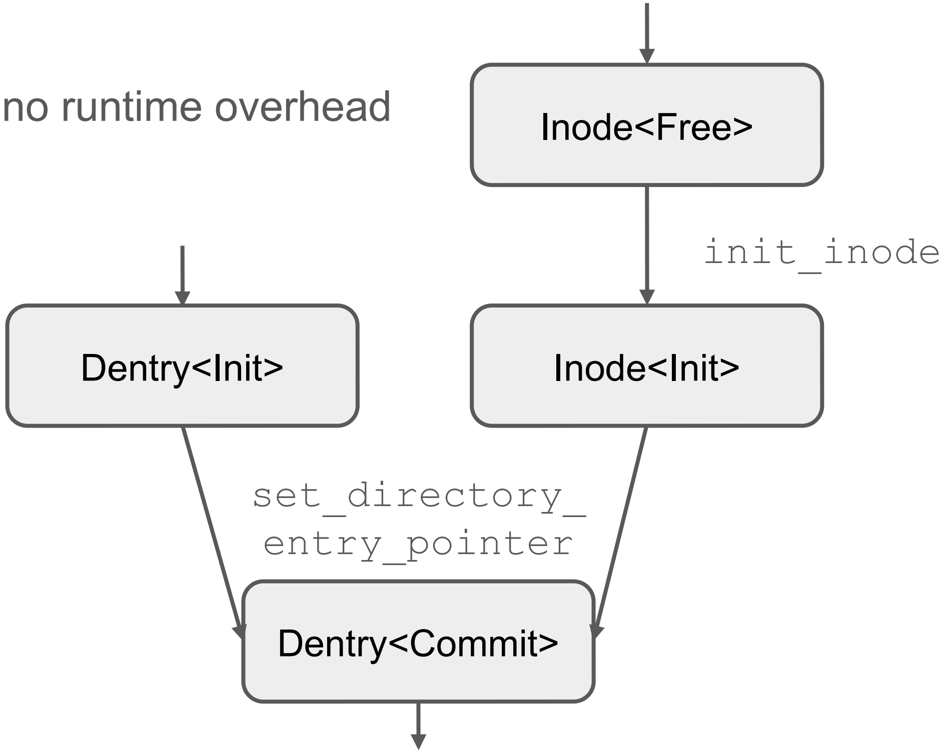
```
 fn set_directory_entry_ptr(
      d: Dentry<Init>,
      i: Inode<Init>
 ) -> Dentry<Commit>
```



The typestate pattern

Encode runtime state in an object’s type with no runtime overhead

```
fn init_inode(i: Inode<Free>)
    -> Inode<Init>

 fn set_directory_entry_ptr(
      d: Dentry<Init>,
      i: Inode<Init>
 ) -> Dentry<Commit>
```

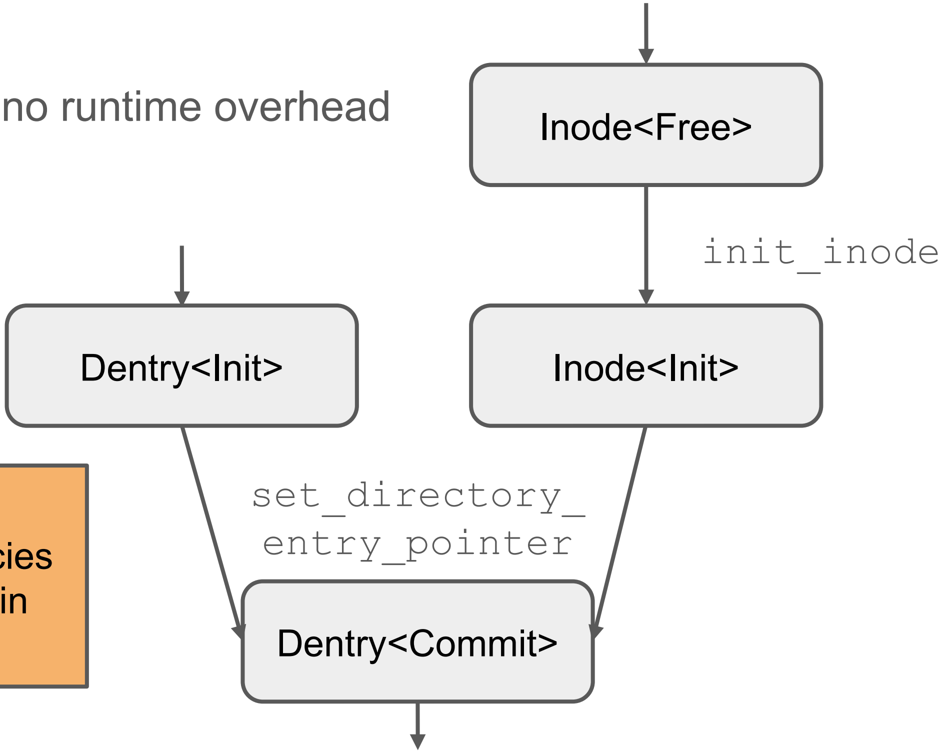


The typestate pattern

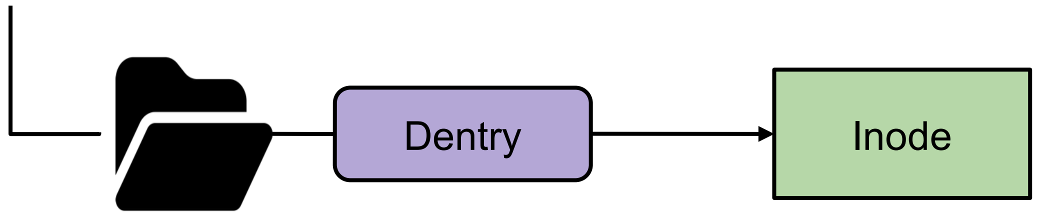
Encode runtime state in an object's type with no runtime overhead

```
fn init_inode(i: Inode<Free>)
    -> Inode<Init>

 fn set_directory_entry_ptr(
      d: Dentry<Init>,
      i: Inode<Init>
 ) -> Dentry<Commit>
```

Update dependencies encoded in types

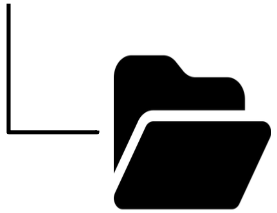


Typestate for crash consistency



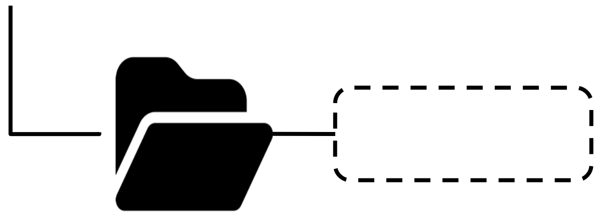
```
fn create_file(name: String) {
    let d = Dentry::get_free_dentry(); // obtain Dentry<Free>
    let i = Inode::get_free_ino(); // obtain Inode<Free>
    let d = d.set_name(name); // Dentry<Free> -> Dentry<Init>
    let d = d.set_directory_entry_ptr(i); // BUG!!
}
```

Typestate for crash consistency



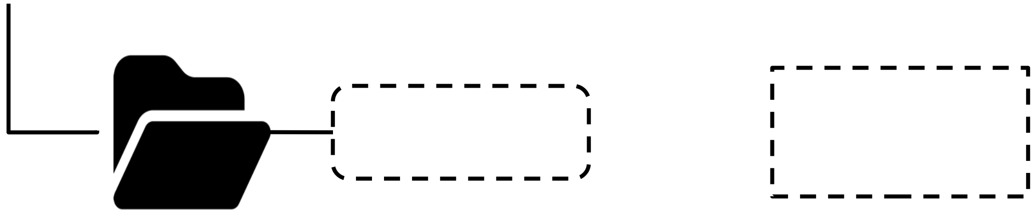
```
fn create_file(name: String) {
    let d = Dentry::get_free_dentry(); // obtain Dentry<Free>
    let i = Inode::get_free_ino(); // obtain Inode<Free>
    let d = d.set_name(name); // Dentry<Free> -> Dentry<Init>
    let d = d.set_directory_entry_ptr(i); // BUG!!
}
```

Typestate for crash consistency



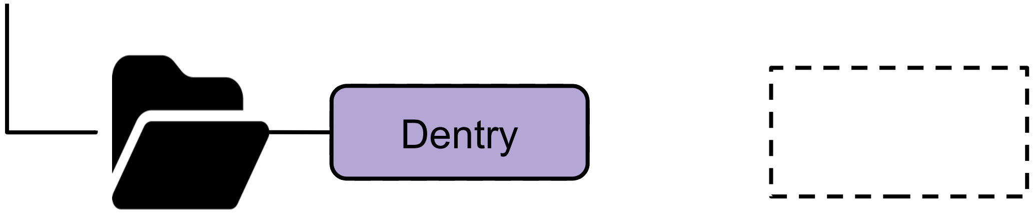
```
fn create_file(name: String) {
    let d = Dentry::get_free_dentry(); // obtain Dentry<Free>
    let i = Inode::get_free_ino(); // obtain Inode<Free>
    let d = d.set_name(name); // Dentry<Free> -> Dentry<Init>
    let d = d.set_directory_entry_ptr(i); // BUG!!
}
```

Typestate for crash consistency



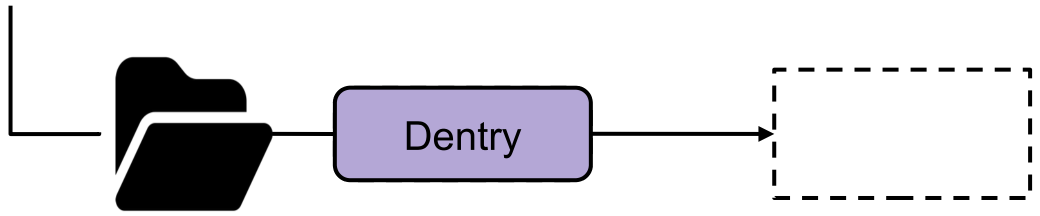
```
fn create_file(name: String) {
    let d = Dentry::get_free_dentry(); // obtain Dentry<Free>
    let i = Inode::get_free_ino();  // obtain Inode<Free>
    let d = d.set_name(name); // Dentry<Free> -> Dentry<Init>
    let d = d.set_directory_entry_ptr(i); // BUG!!
}
```

Typestate for crash consistency



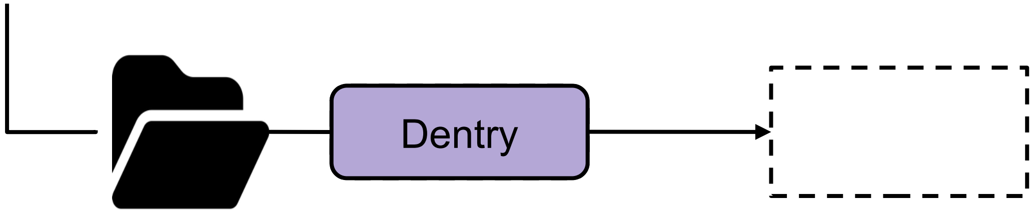
```
fn create_file(name: String) {
    let d = Dentry::get_free_dentry(); // obtain Dentry<Free>
    let i = Inode::get_free_ino(); // obtain Inode<Free>
    let d = d.set_name(name); // Dentry<Free> -> Dentry<Init>
    let d = d.set_directory_entry_ptr(i); // BUG!!
}
```

Typestate for crash consistency



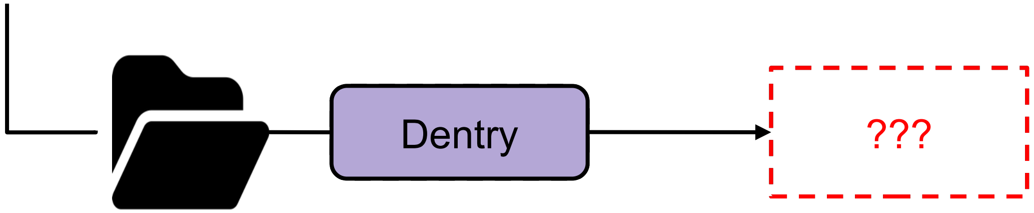
```
fn create_file(name: String) {
    let d = Dentry::get_free_dentry(); // obtain Dentry<Free>
    let i = Inode::get_free_ino(); // obtain Inode<Free>
    let d = d.set_name(name); // Dentry<Free> -> Dentry<Init>
    let d = d.set_directory_entry_ptr(i); // BUG!!
}
```

Typestate for crash consistency



```
fn create_file(name: String) {
    let d = Dentry::get_free_dentry(); // obtain Dentry<Free>
    let i = Inode::get_free_ino(); // obtain Inode<Free>
    let d = d.set_name(name); // Dentry<Free> -> Dentry<Init>
    let d = d.set_directory_entry_ptr(i); // BUG!!
}
```

Typestate for crash consistency



```
fn create_file(name: String) {
    let d = Dentry::get_free_dentry(); // obtain Dentry<Free>
    let i = Inode::get_free_ino(); // obtain Inode<Free>
    let d = d.set_name(name); // Dentry<Free> -> Dentry<Init>
    let d = d.set_directory_entry_ptr(i); // BUG!!
}
```

```
error[E0308]: mismatched types
  --> src/main.rs:46:39
   |
46 |     let d = d.set_directory_entry_ptr(i);
   |               ----------------------- ^ expected `Inode<Init>`, found `Inode<Free>`
   |               |
   |               arguments to this method are incorrect
   |
   = note: expected struct `Inode<Init>`
              found struct `Inode<Free>`
```

Roadmap

SquirrelFS implementation

Typestate-checked Synchronous Soft Updates for crash consistency

7500 LOC of Rust

Simple durable layout with volatile indexes and allocators

Atomic metadata-related system calls (including `rename`)

Modeled as a transition system and model checked in Alloy

Roadmap

1. Introduction
2. Ordering for crash consistency
3. Typestate pattern
4. SquirrelFS implementation
5. **Evaluation**

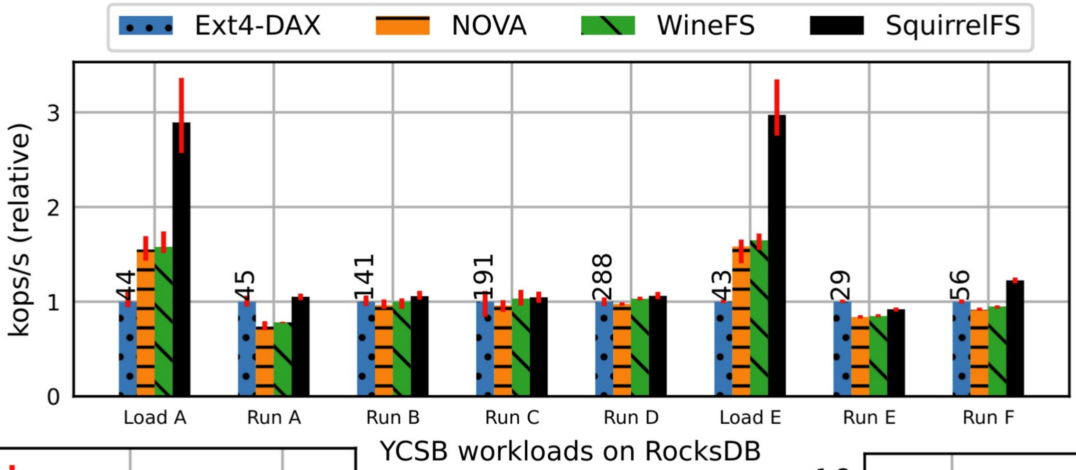
Evaluation

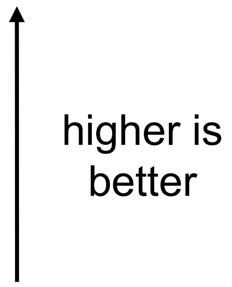
Evaluated on 128GB Intel Optane DC Persistent Memory Module

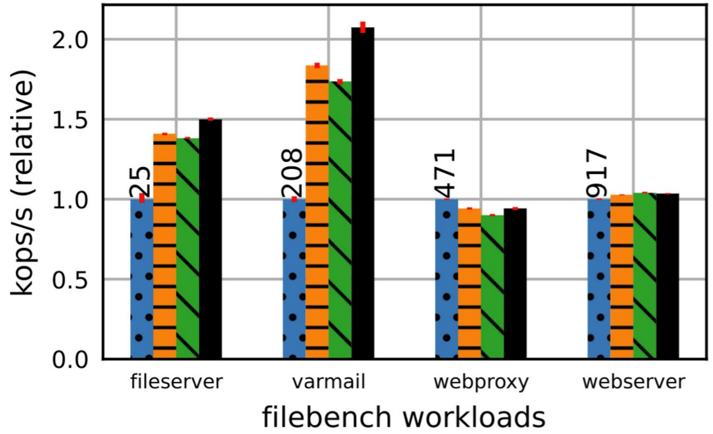
Compared against **Ext4-DAX**, **NOVA**, and **WineFS**

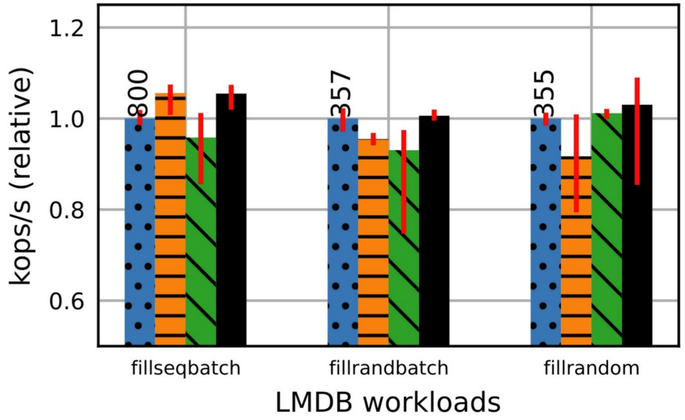
1. How does SquirrelFS compare to other PM file systems?
2. How long does it take to statically check SquirrelFS's crash-consistency properties?

SquirrelFS performance









Compilation and verification times

Compilation and verification times

System (verified)	Lines of code	Verification time (s)
FSCQ	31K	39600
VeriBetrKV	45K	6480

Compilation and verification times

System (verified)	Lines of code	Verification time (s)
FSCQ	31K	39600
VeriBetrKV	45K	6480

System (unverified)	Lines of code	Compilation time (s)
Ext4	45K	38
NOVA	16K	20
WineFS	9K	13

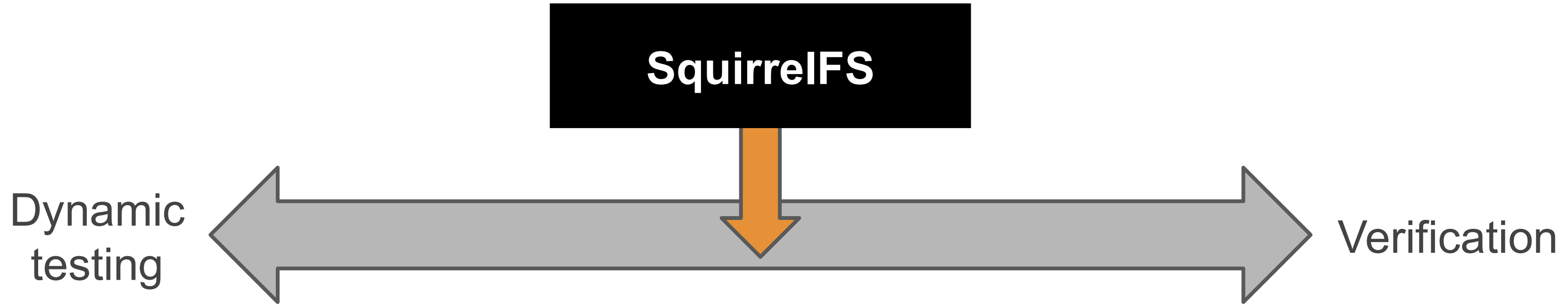
Compilation and verification times

System (verified)	Lines of code	Verification time (s)
FSCQ	31K	39600
VeriBetrKV	45K	6480

System (unverified)	Lines of code	Compilation time (s)
Ext4	45K	38
NOVA	16K	20
WineFS	9K	13

System (typestate-checked)	Lines of code	Compile+check time (s)
SquirrelFS	**7.5K**	**10**

Conclusion



Typestate pattern statically checks ordering for crash consistency

Synchronous Soft Updates crash-consistency mechanism

Comparable performance to existing PM file systems

https://github.com/utsaslab/squirrelfs

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

Extra slides

Background: persistent memory

Low latency on the order of DRAM

Byte-addressable via memory loads and stores

Cache-line flushes and memory fences for durability and ordering

Examples:

- Intel Optane DC Persistent Memory Module
- Battery-backed DRAM
- Future devices: Micron, startups, CXL.mem, …

Background: soft updates

Crash-consistency mechanism based on ordering in-place updates

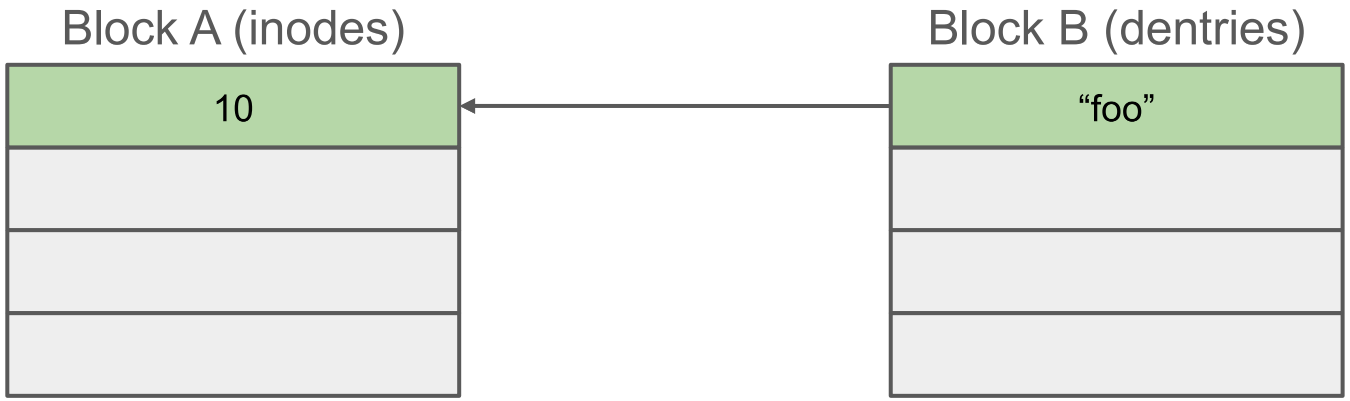
Rules:

1. Never point to a structure before it is initialized
2. Never reuse a resource before nullifying existing references to it
3. Never reset the old pointer to a resource before setting the new one

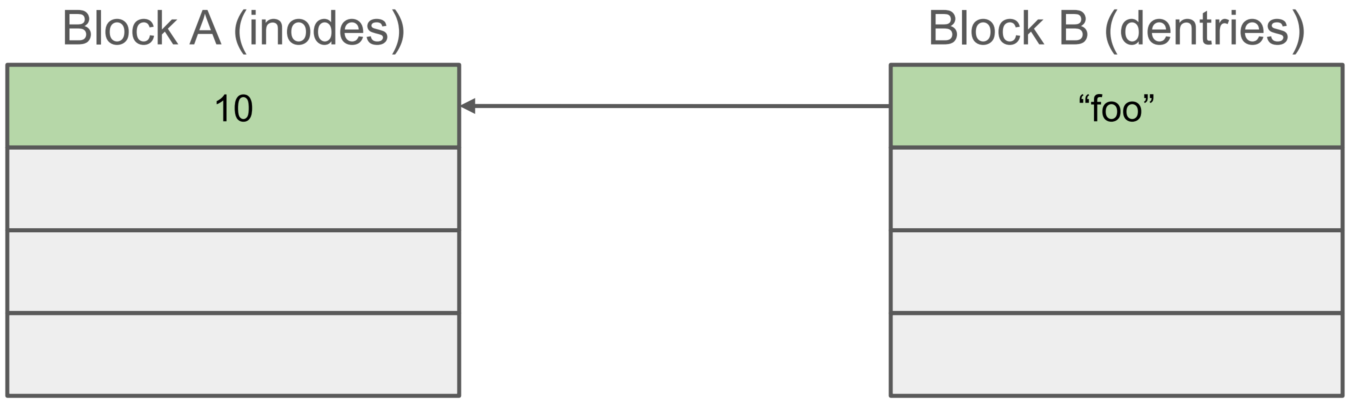
Enforced by tracking update dependencies and ordering durable updates

Reduces write amplification, but increases complexity

Soft updates cyclic dependencies example

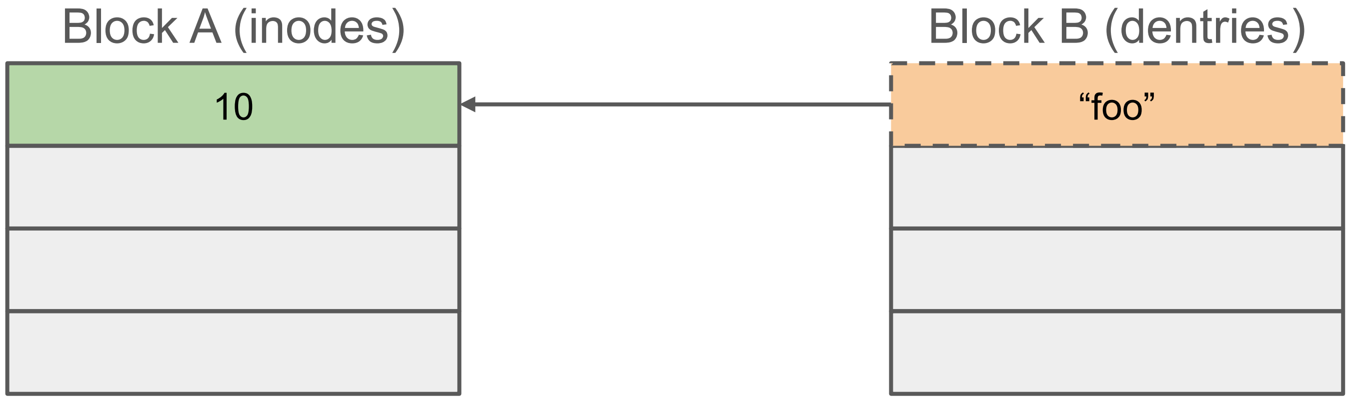


Soft updates cyclic dependencies example



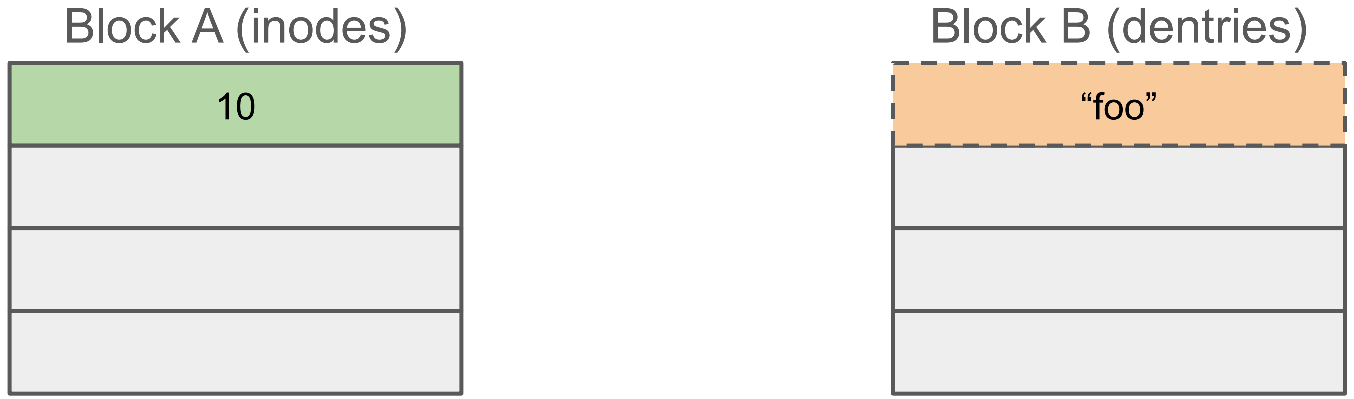
1. unlink foo

Soft updates cyclic dependencies example



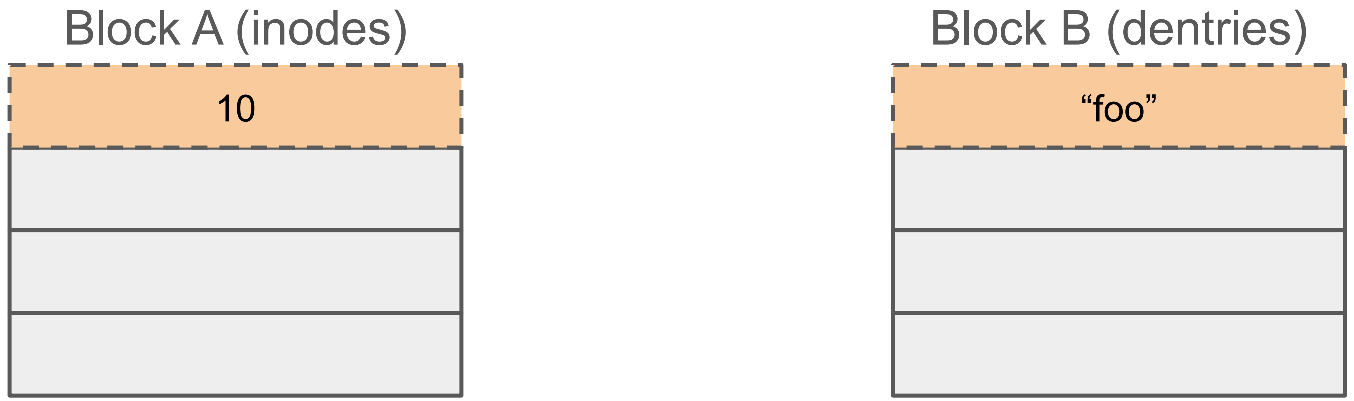
1. unlink foo

Soft updates cyclic dependencies example



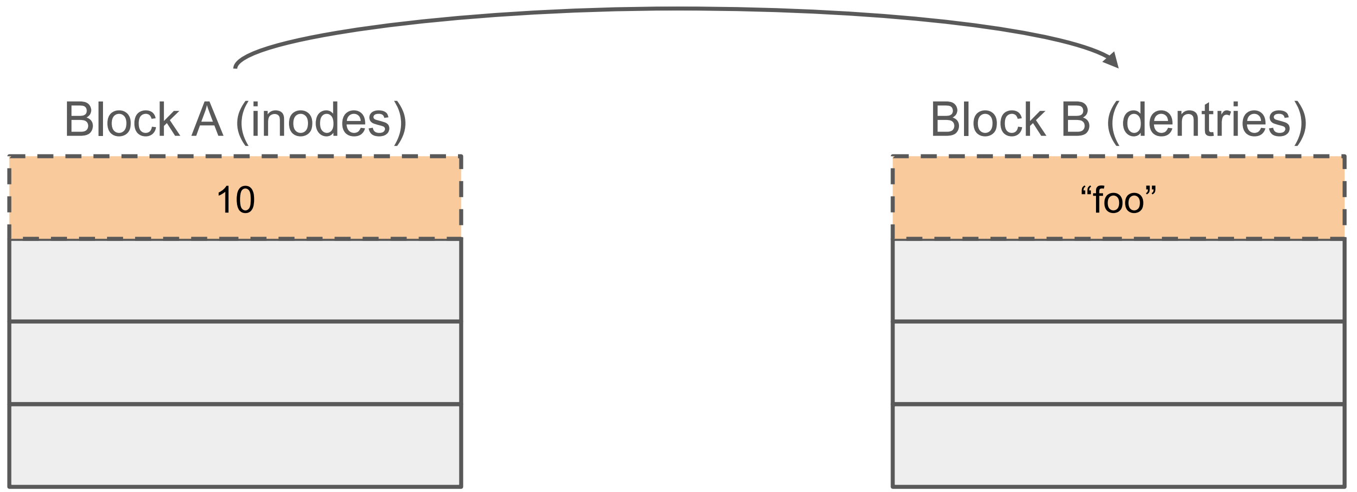
1. unlink foo

Soft updates cyclic dependencies example



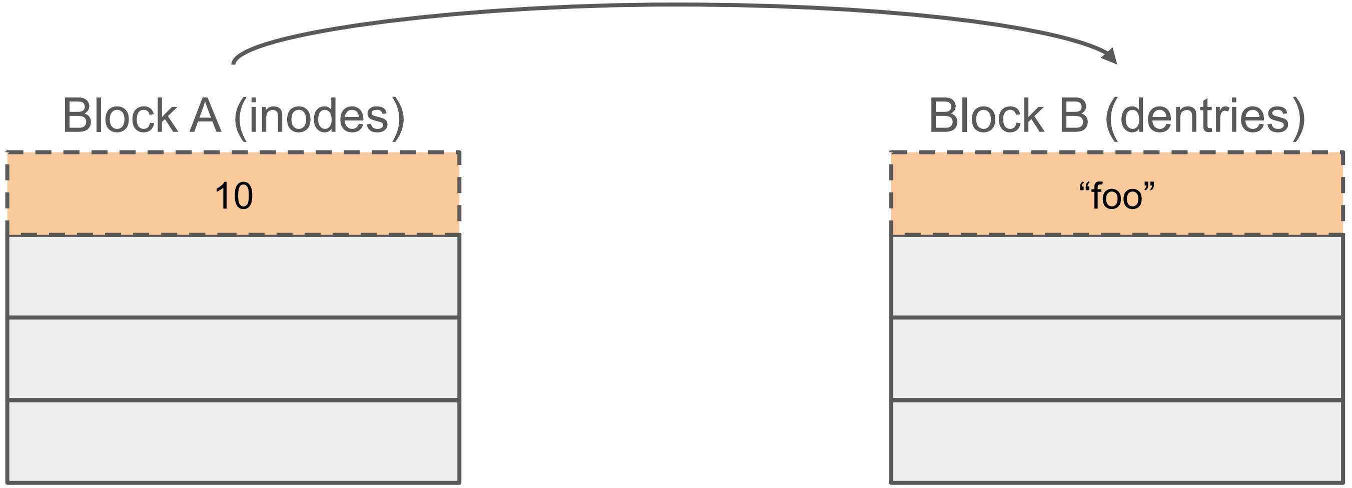
1. unlink foo

Soft updates cyclic dependencies example



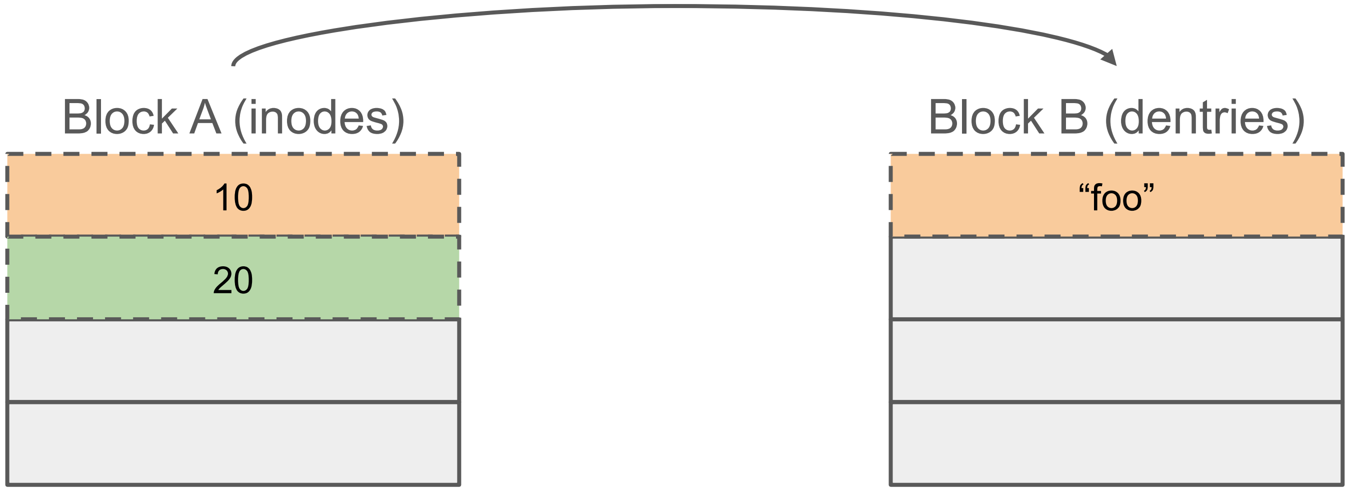
1. unlink foo

Soft updates cyclic dependencies example



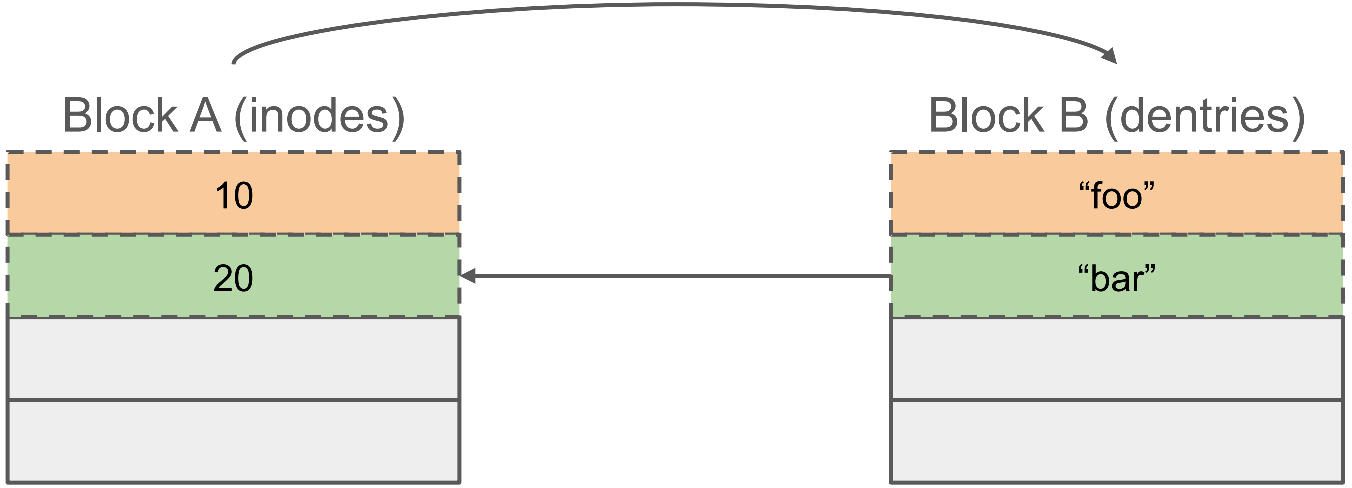
1. unlink foo
2. create bar

Soft updates cyclic dependencies example



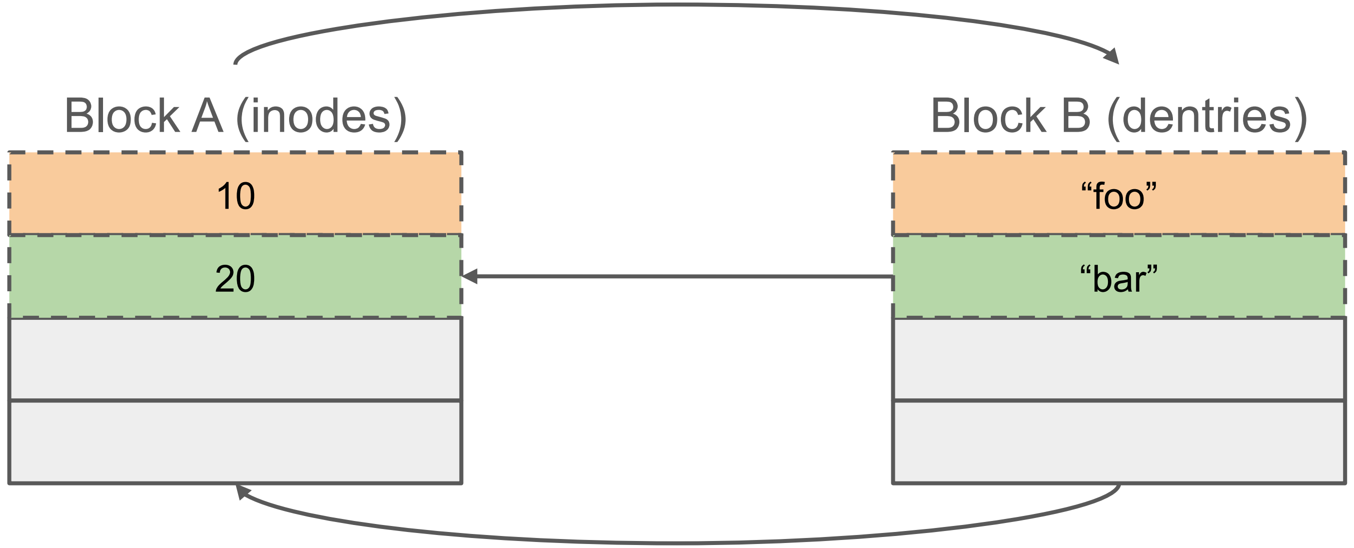
1. unlink foo
2. create bar

Soft updates cyclic dependencies example



1. unlink foo
2. create bar

Soft updates cyclic dependencies example



1. unlink foo
2. create bar

Background: verification

Mathematical proof that a program is correct

Prove that the complex implementation matches a simpler specification of correctness

Developer writes a proof, computer checks it

Uses verification-aware programming languages or interactive theorem provers

E.g.: Verus verification framework for Rust

Typestate in Rust: update operations

```
impl Inode<Clean,Free> {
    fn init(self,...) -> Inode<Dirty,Init> {...}
}
impl Dentry<Clean,Free> {
    fn set_name(self, name: String) -> Dentry<Dirty,Init> {...}
}
impl Dentry<Clean,Init> {
    fn set_ino(self, ino: Inode<Clean,Init>) -> Dentry<Dirty,Committed> {...}
}
```

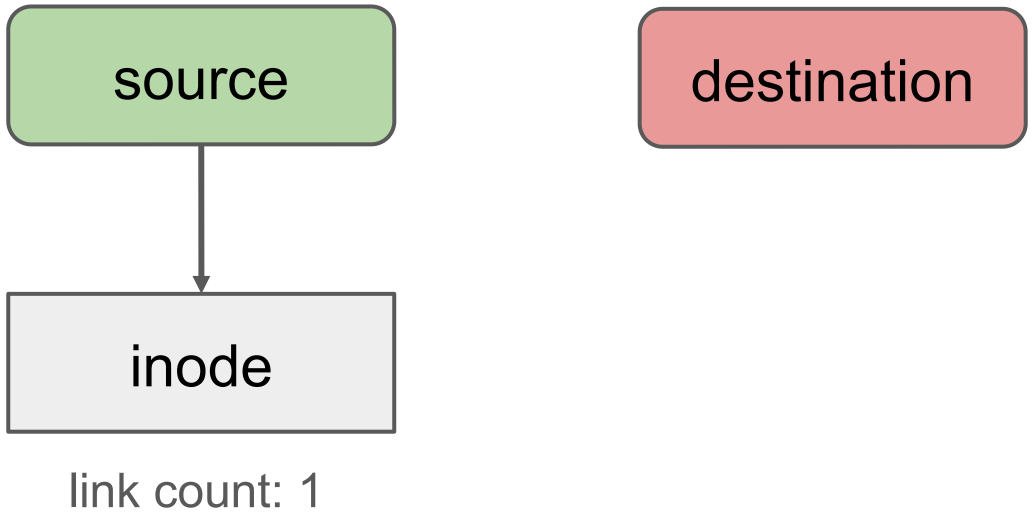
Typestate: ensuring persistence

```
impl<S> Inode<Dirty,S> {

     fn flush(self) -> Inode<InFlight,S> {...}

}

impl<S> Inode<InFlight,S> {

     fn fence(self) -> Inode<Clean,S> {...}

}
```

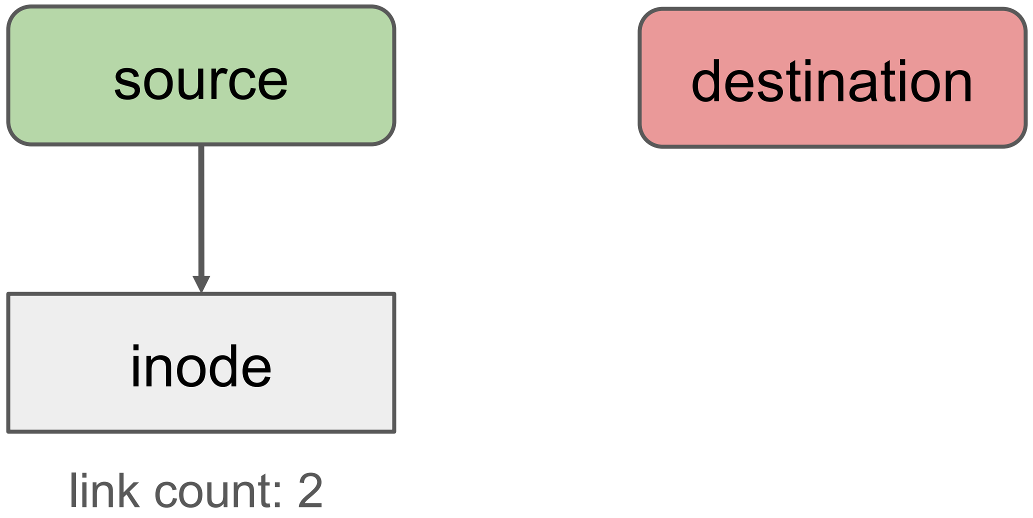
Directory entry validity rules

1. If a dentry’s inode number is 0, the dentry is invalid.
2. If dst’s rename pointer points to src, then:
    a. If dst.inode != src.inode, both dentries are valid
    b. If dst.inode == src.inode, src is invalid

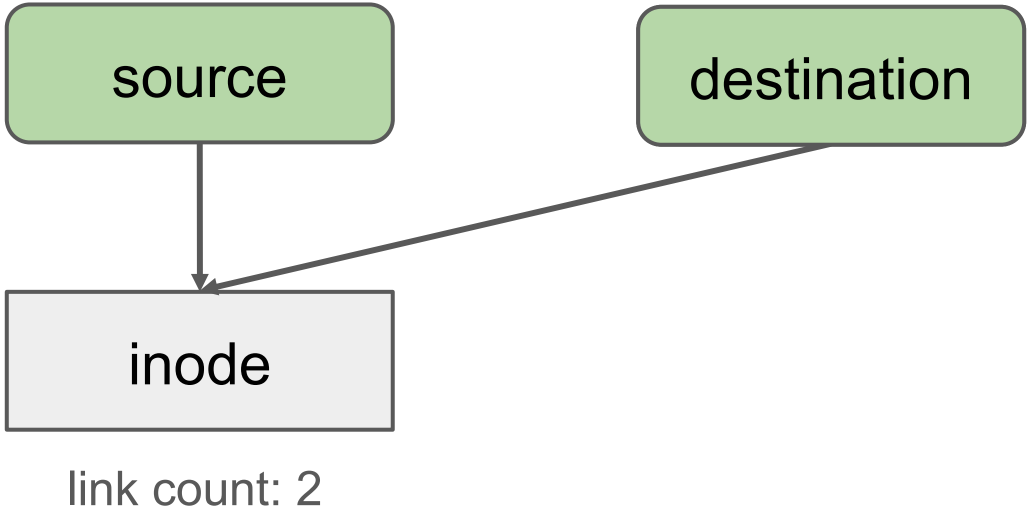
Traditional soft updates rename



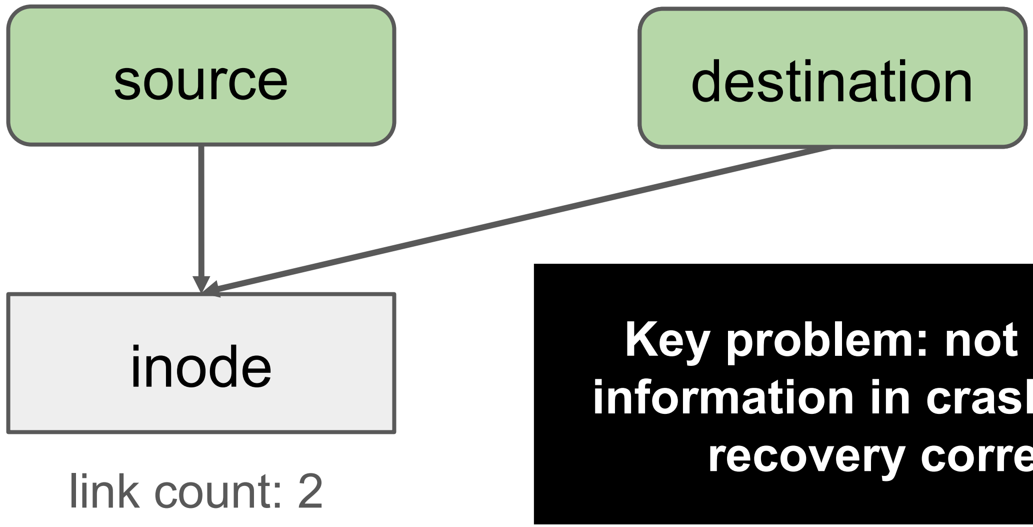
Traditional soft updates rename



Traditional soft updates rename

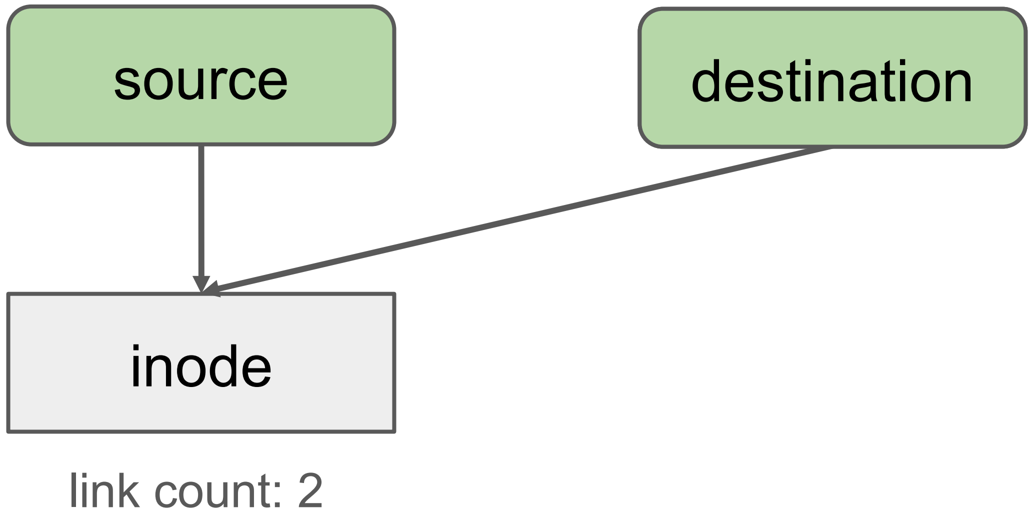


Traditional soft updates rename

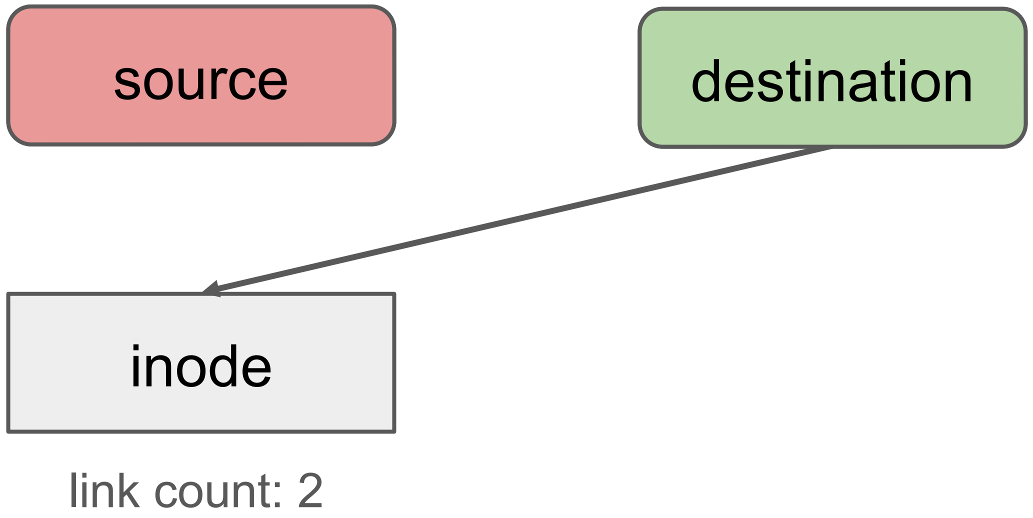


Key problem: not enough information in crash state to recovery correctly

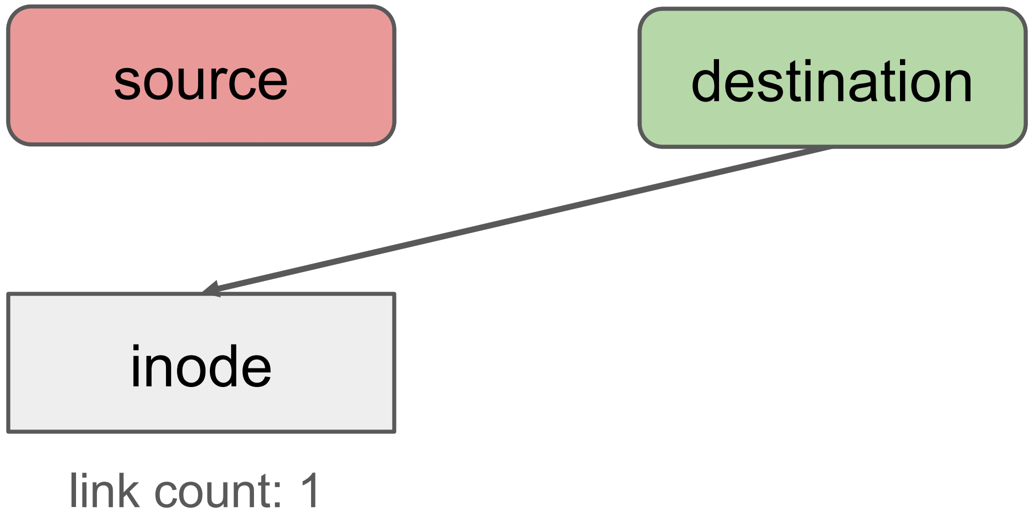
Traditional soft updates rename



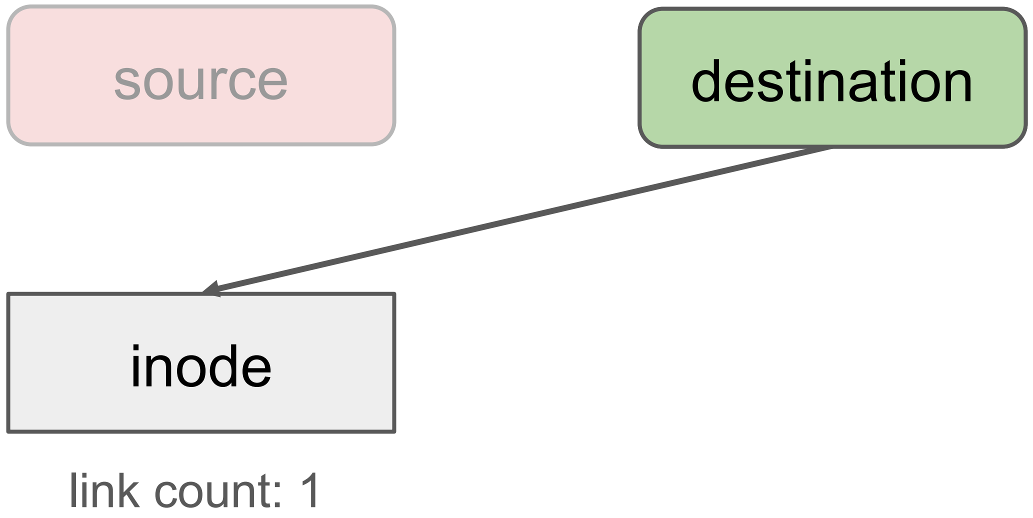
Traditional soft updates rename



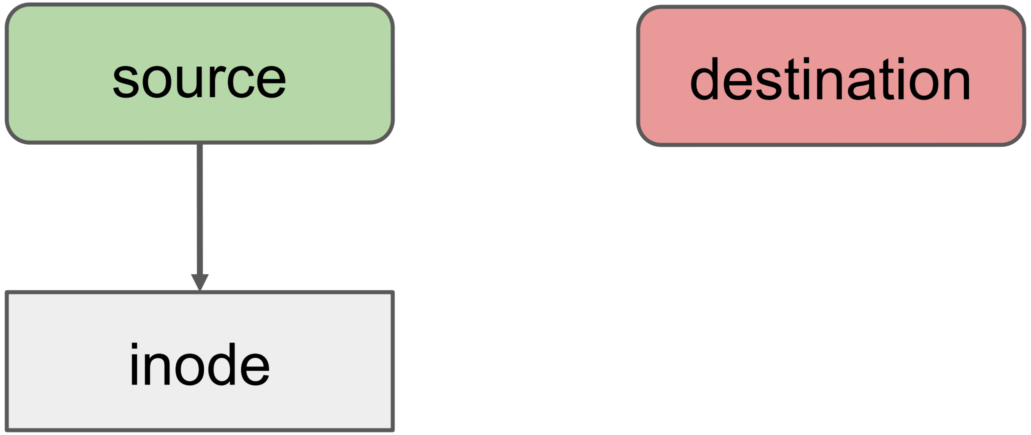
Traditional soft updates rename



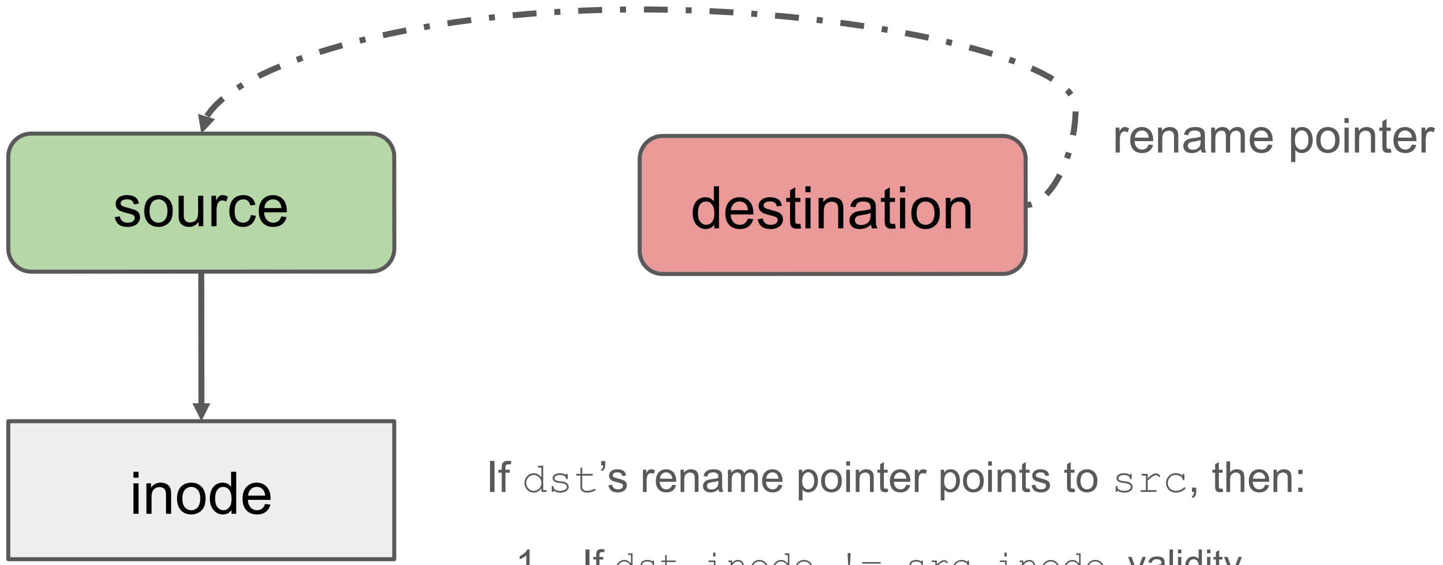
Traditional soft updates rename



Atomic rename with SSU



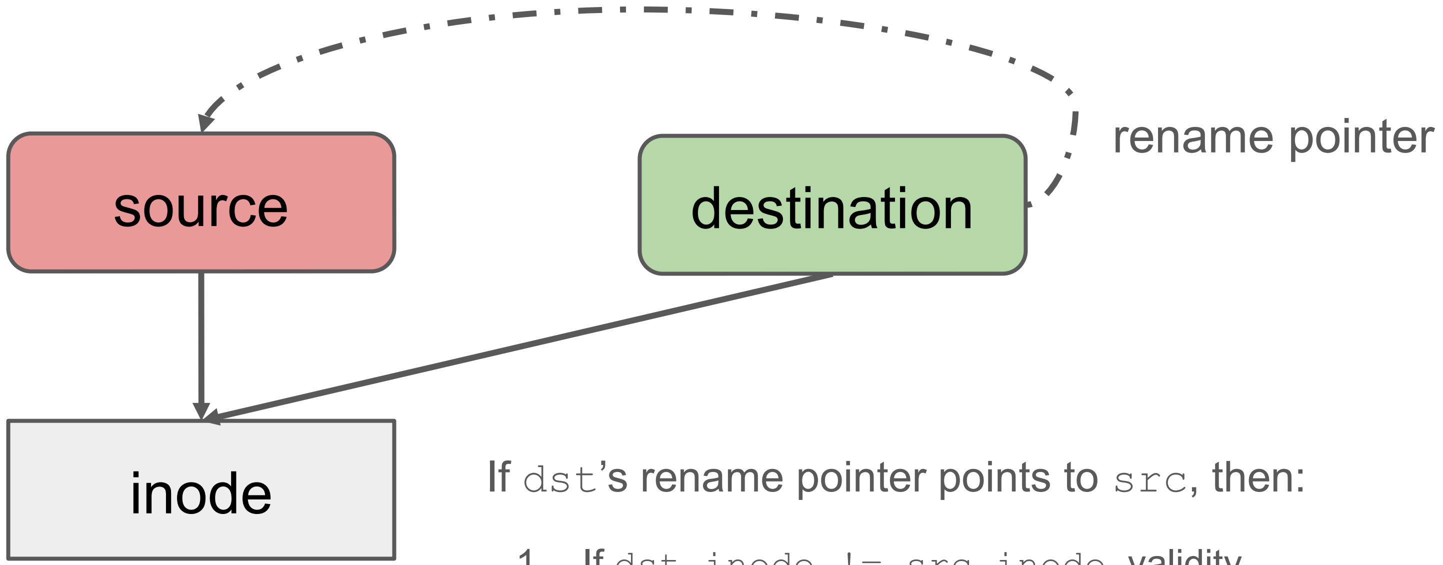
Atomic rename with SSU



If `dst`'s rename pointer points to `src`, then:

1. If `dst.inode != src.inode`, validity determined by inode number
2. If `dst.inode == src.inode`, src is invalid

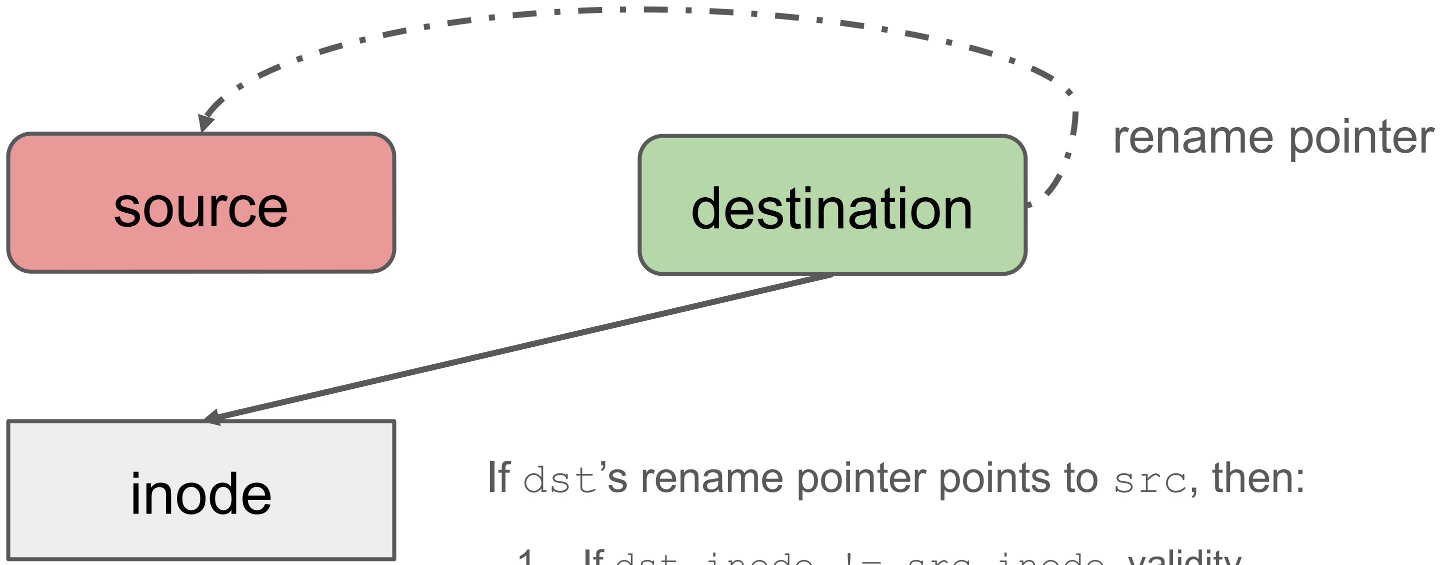
Atomic rename with SSU



If `dst`’s rename pointer points to `src`, then:

1. If `dst.inode != src.inode`, validity determined by inode number
2. If `dst.inode == src.inode`, src is invalid

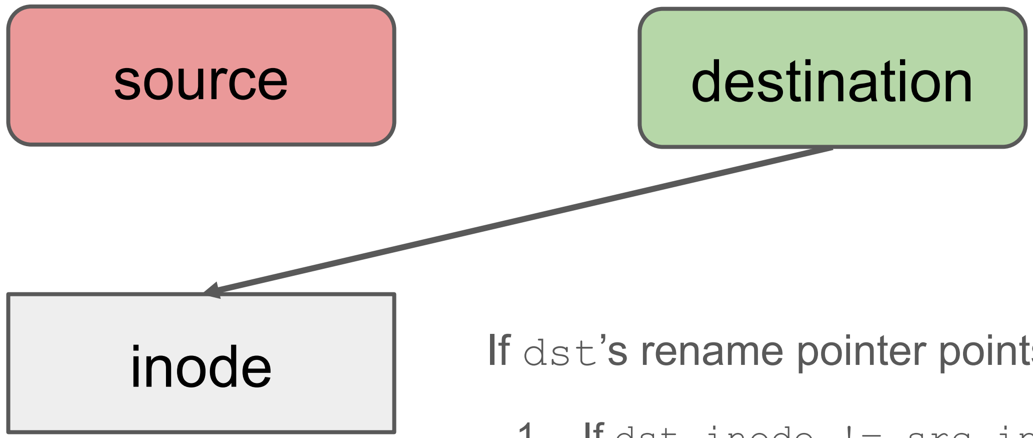
Atomic rename with SSU



If `dst`'s rename pointer points to `src`, then:

1. If `dst.inode != src.inode`, validity determined by inode number
2. If `dst.inode == src.inode`, src is invalid

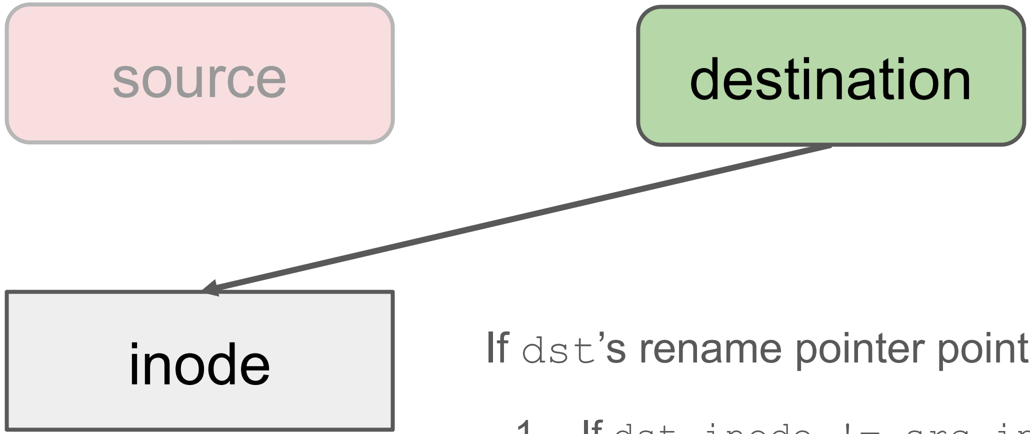
Atomic rename with SSU



If `dst`’s rename pointer points to `src`, then:

1. If `dst.inode != src.inode`, validity determined by inode number
2. If `dst.inode == src.inode`, src is invalid

Atomic rename with SSU



If `dst`'s rename pointer points to `src`, then:

1. If `dst.inode != src.inode`, validity determined by inode number
2. If `dst.inode == src.inode`, src is invalid

Typestate in SquirrelFS

Operational typestate

- What operations have been performed on this object?
- Is it free? Initialized? Allocated but not initialized?

Persistence typestate

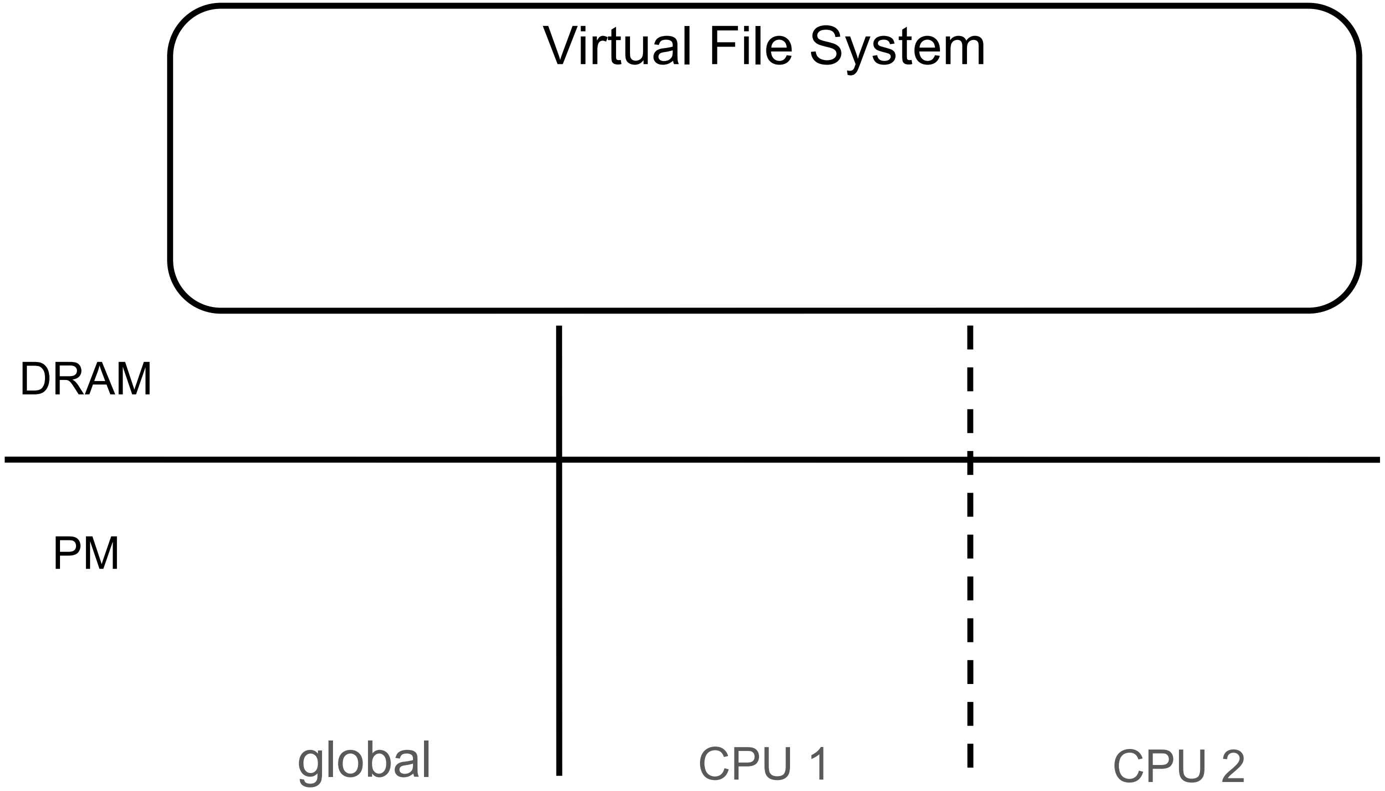
- Have the most recent updates been made durable?

Typestate transition functions make persistent updates and return the new typestate

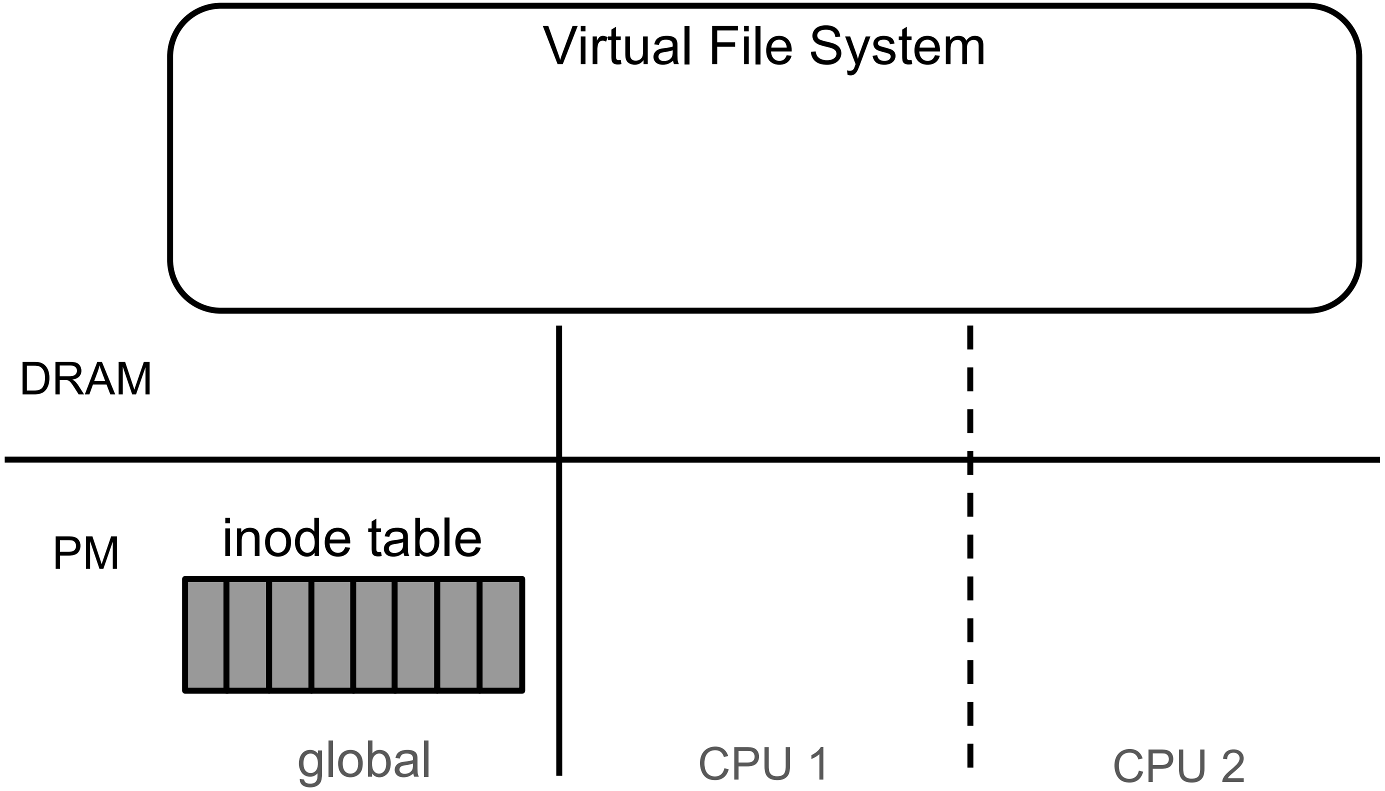
SquirrelFS crash consistency bugs

1. A cache line flush persistence function was passed a reference to a page pointer, rather than the page pointer itself (typestate transition body)
2. Missing case to free orphaned dir pages (recovery code)
3. Allocated but orphaned directory entries towards parent link count (recovery code)
4. Used persistent inode number, rather than inode table index, in inode table scan (recovery code)

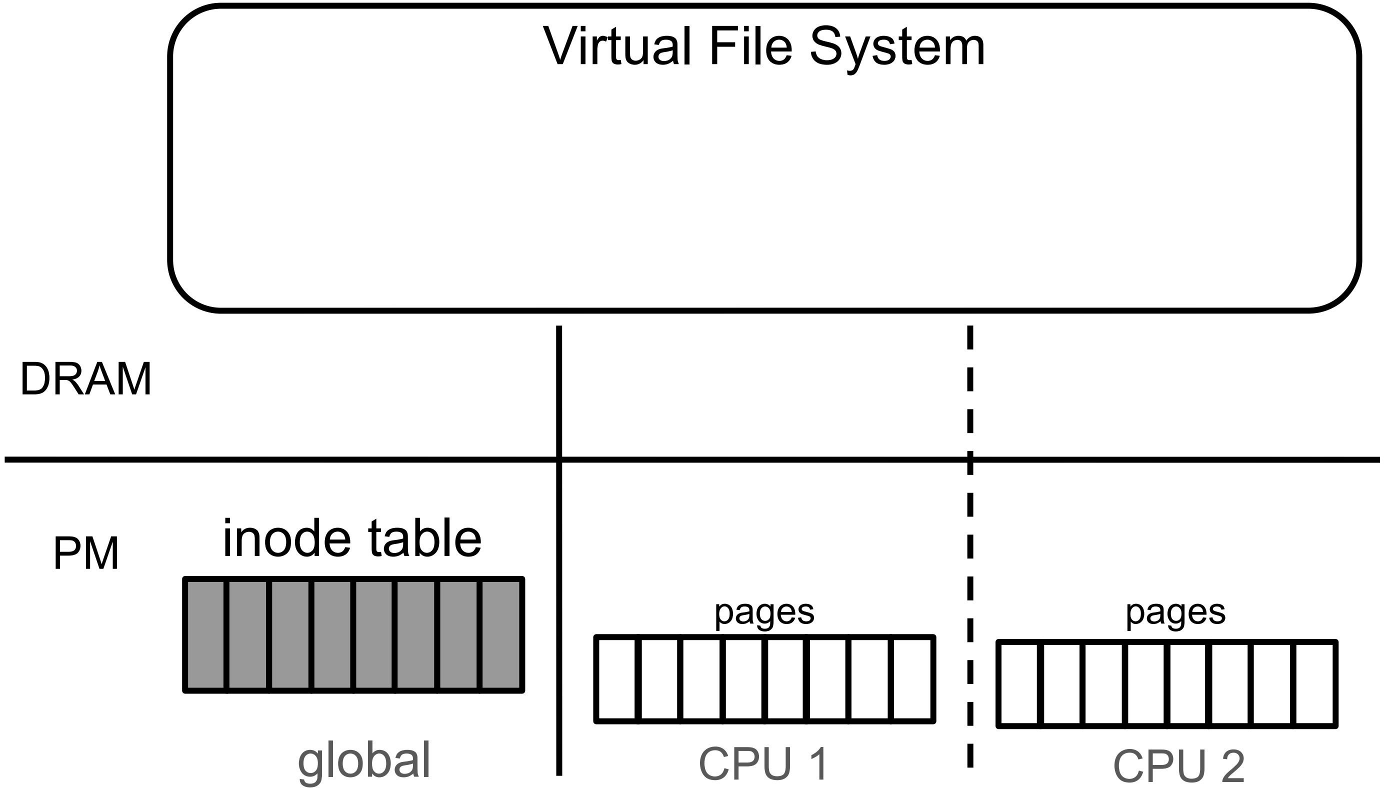
SquirrelFS architecture



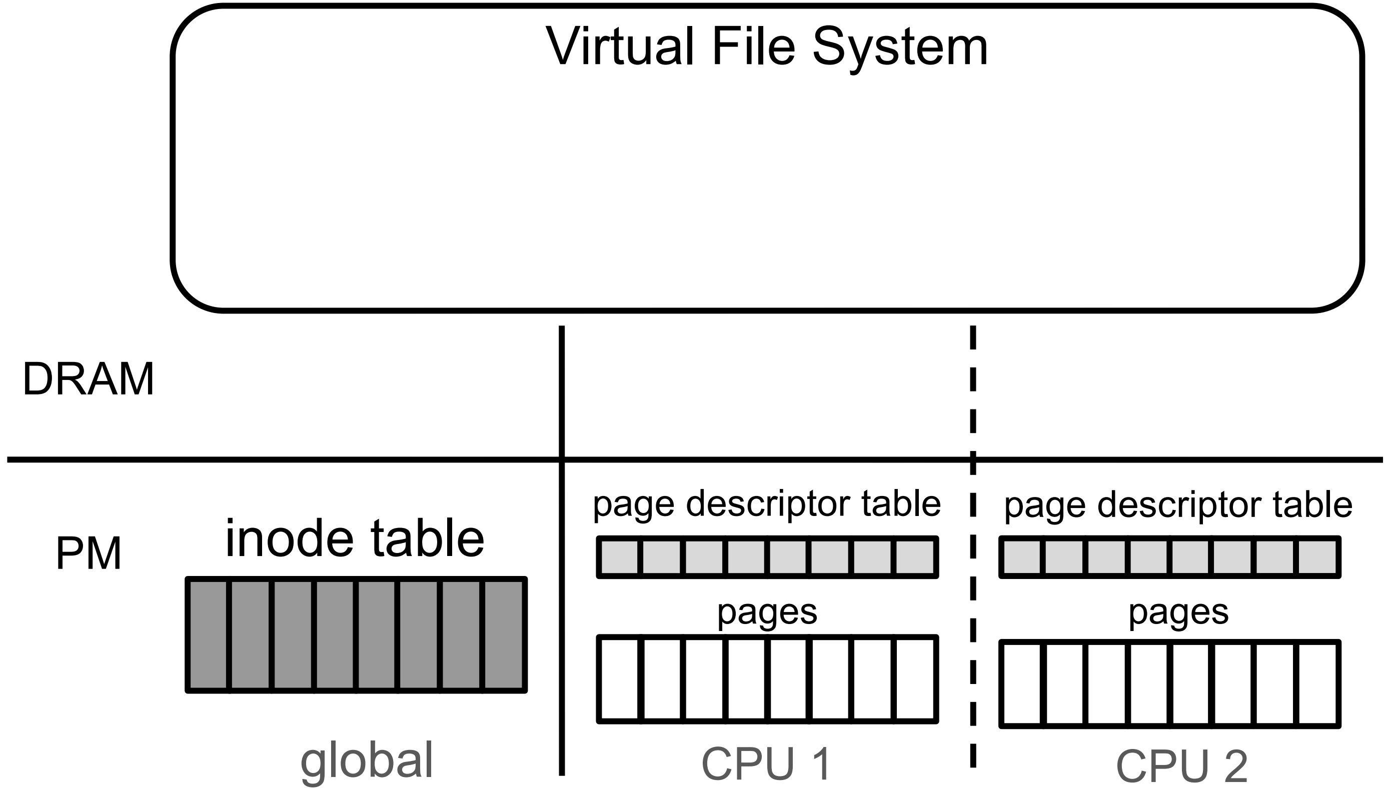
SquirrelFS architecture



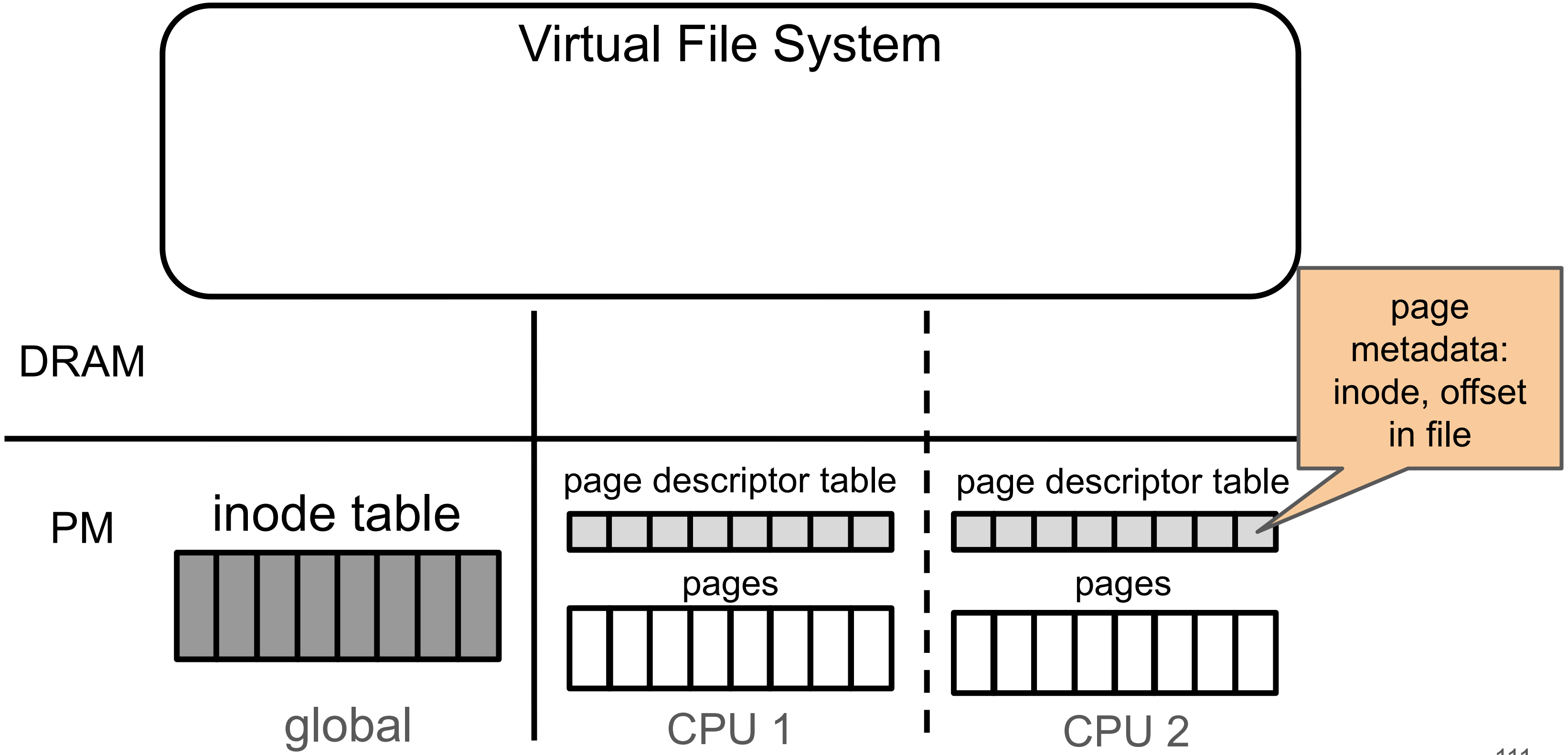
SquirrelFS architecture



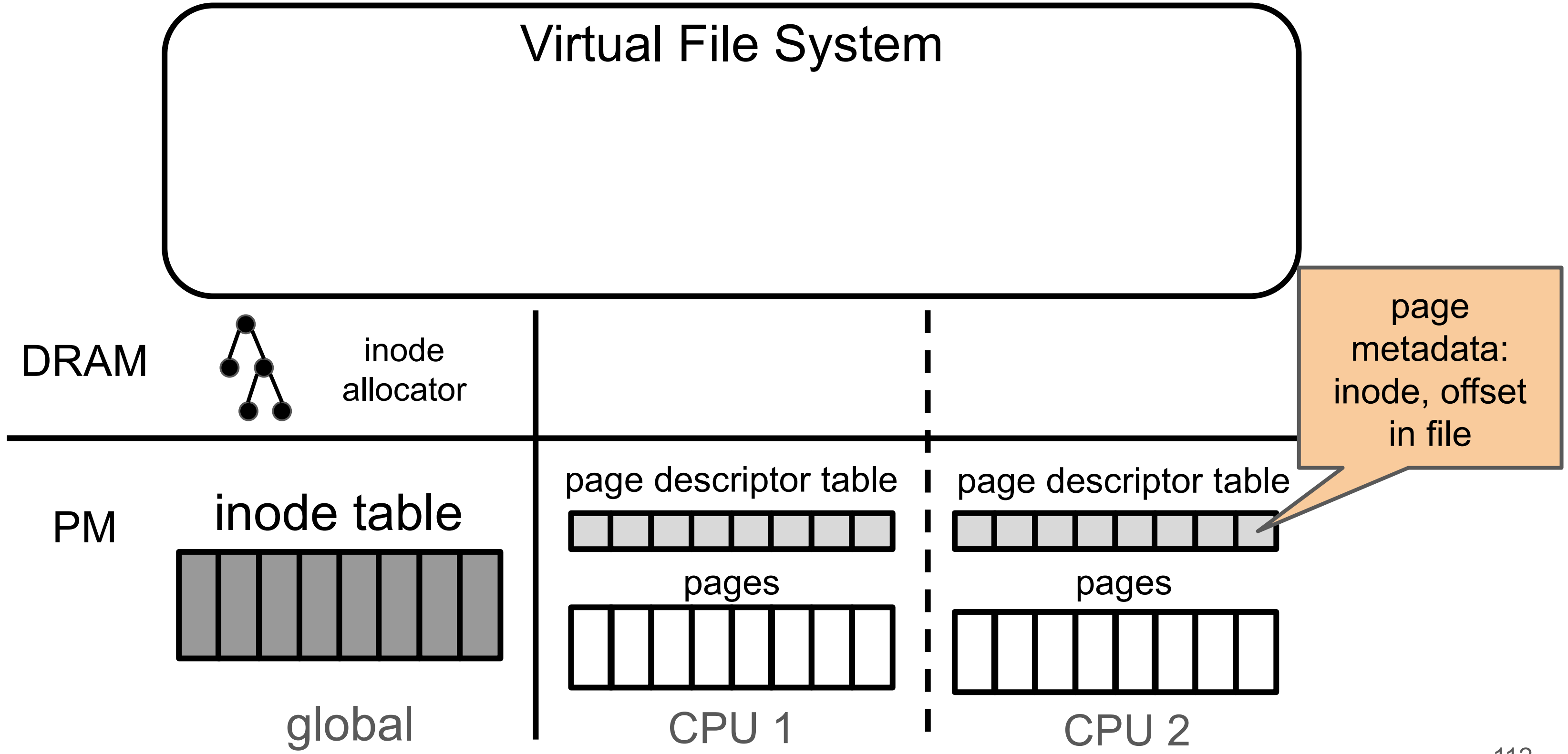
SquirrelFS architecture



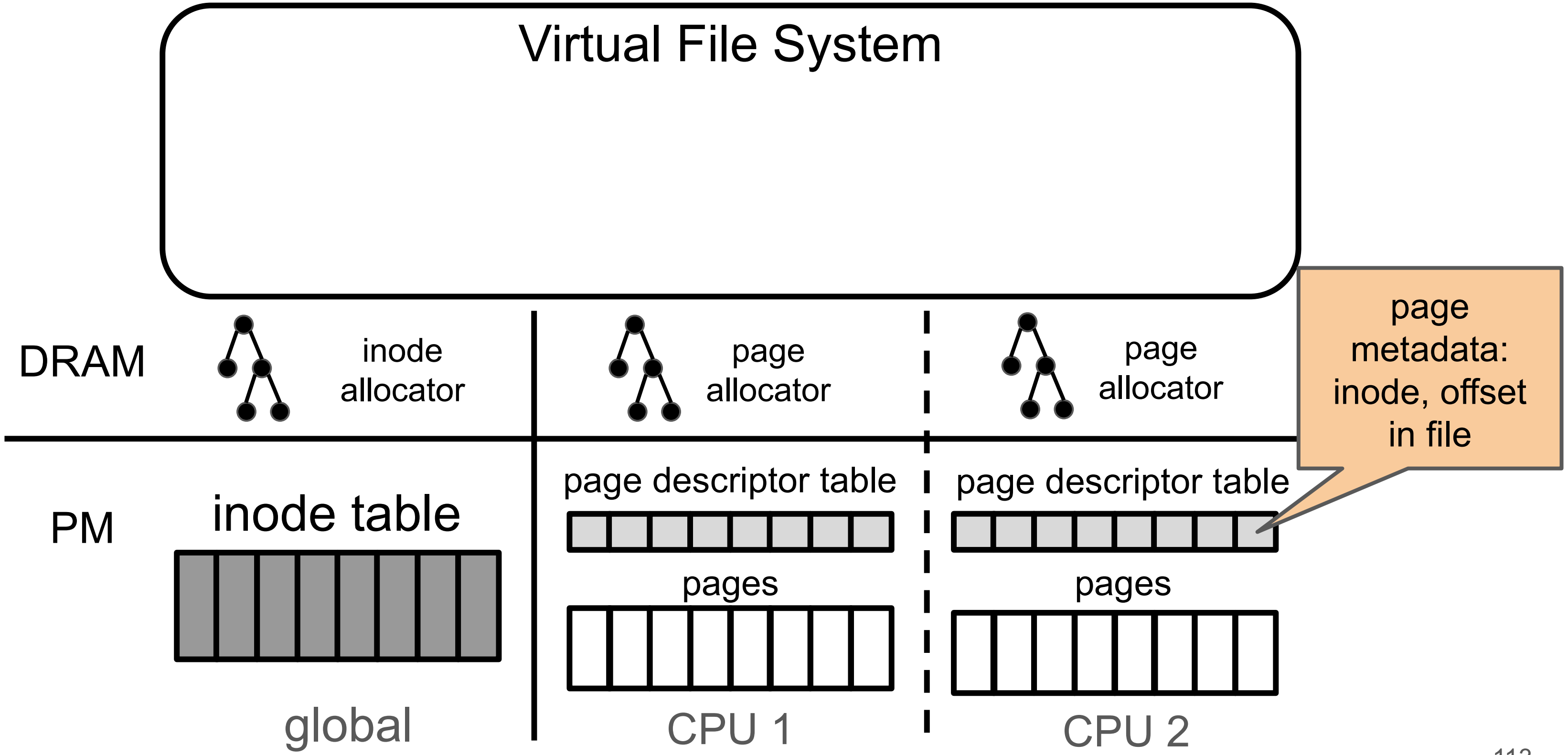
SquirrelFS architecture



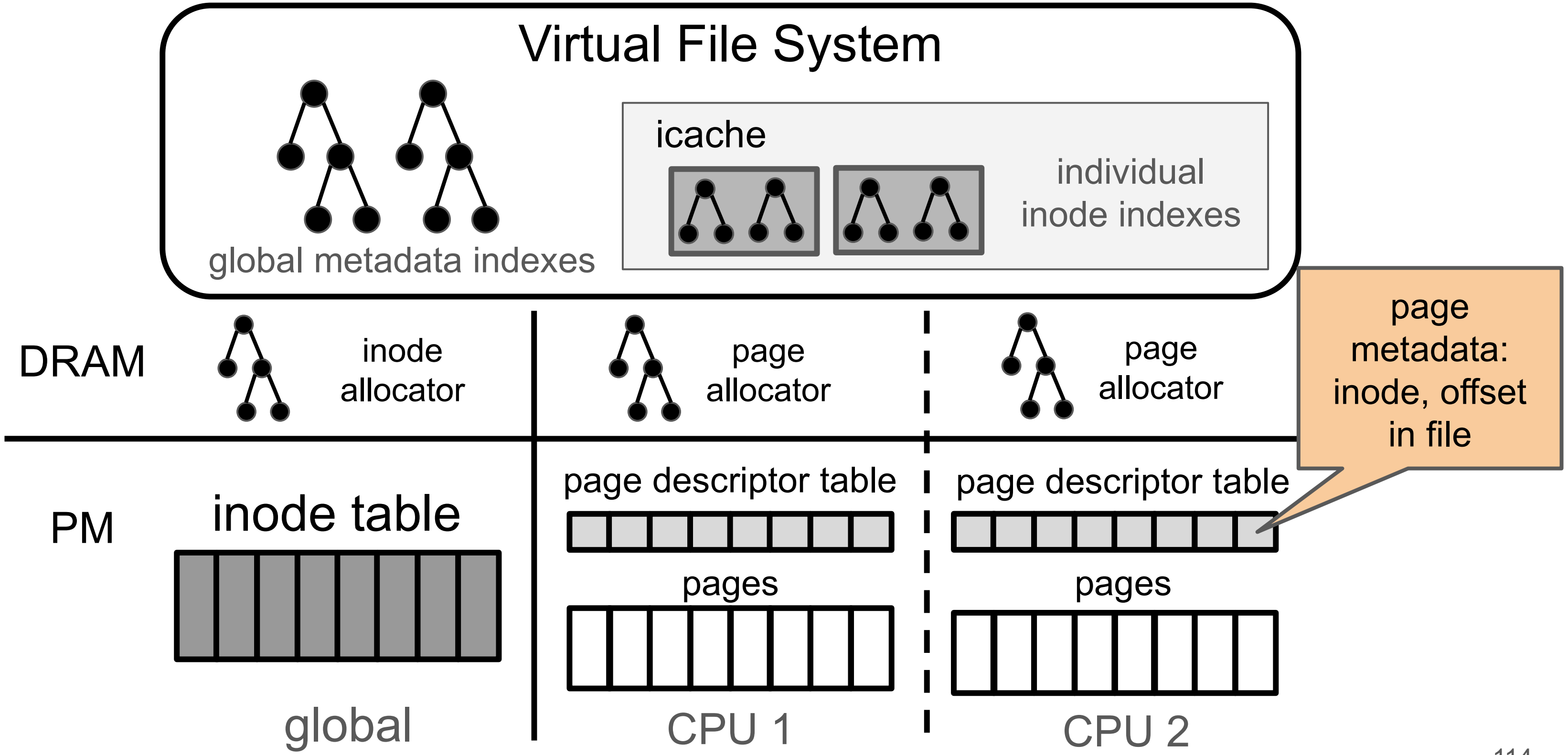
SquirrelFS architecture



SquirrelFS architecture



SquirrelFS architecture



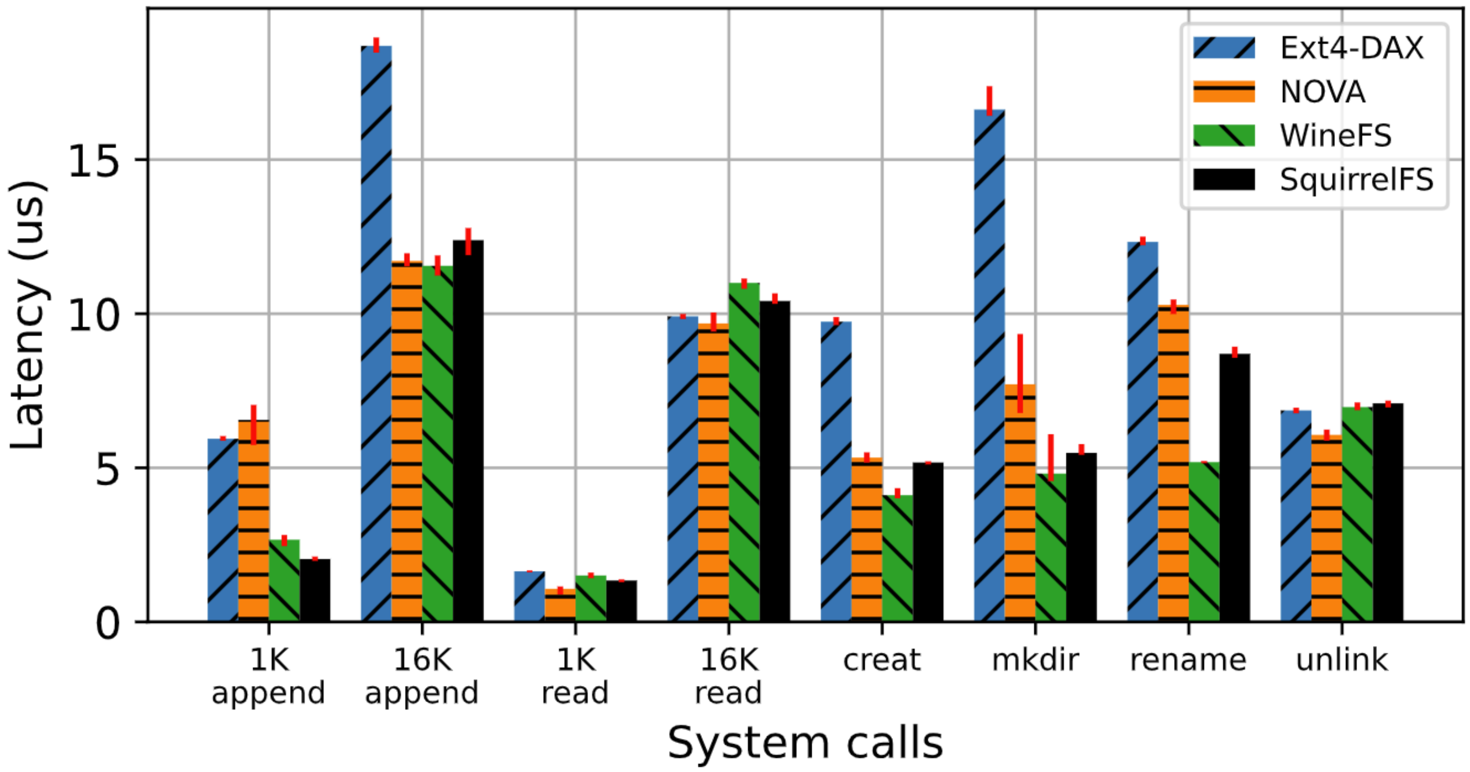
Bugs found with typestate

- Missing persistence primitives
  - E.g.: initial implementation of write was missing flush/fence calls after setting new page backpointer
- Incorrect ordering
  - E.g.: initial rename implementation incorrectly updated link count before clearing a directory entry, which could result in a dangling link later on

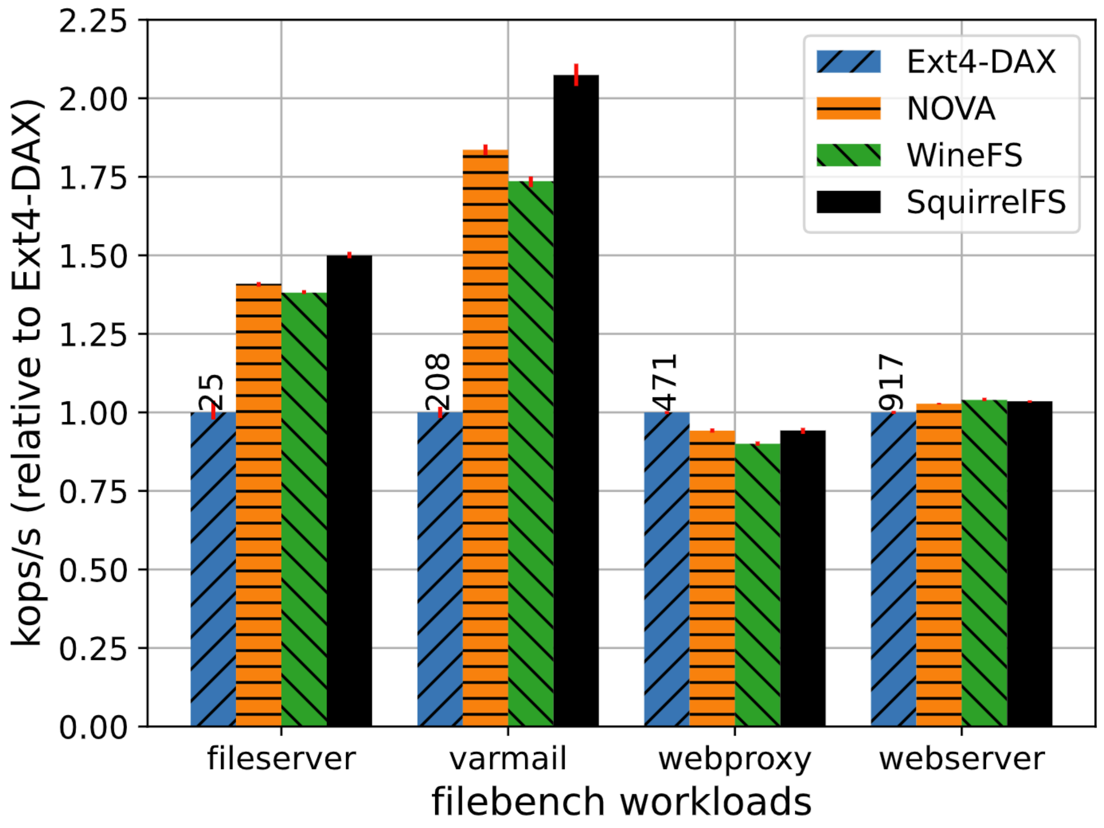
Bugs found with Alloy model

- Recovering from renames
  - Initial model did not include any crash recovery logic; we believed it was not necessary
  - Model found a counterexample where invalid directory entries could reappear after a crash during rename
  - Fixed by adding mandatory post-crash cleanup of rename pointers
- Handling . and .. dentries
  - Originally stored durably and included in update ordering rules
  - Alloy model repeatedly found issues with these rules, particularly during rename
  - Now stored only in volatile memory

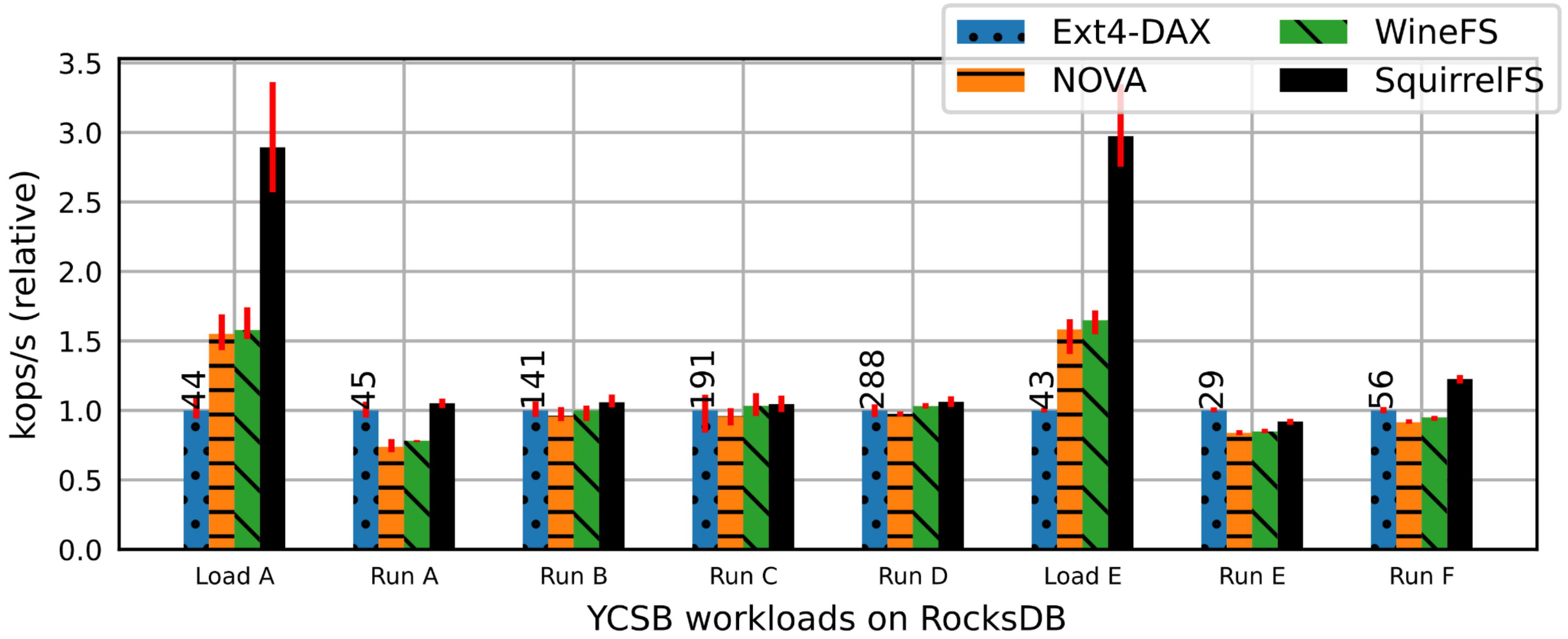
Microbenchmark: system call latency



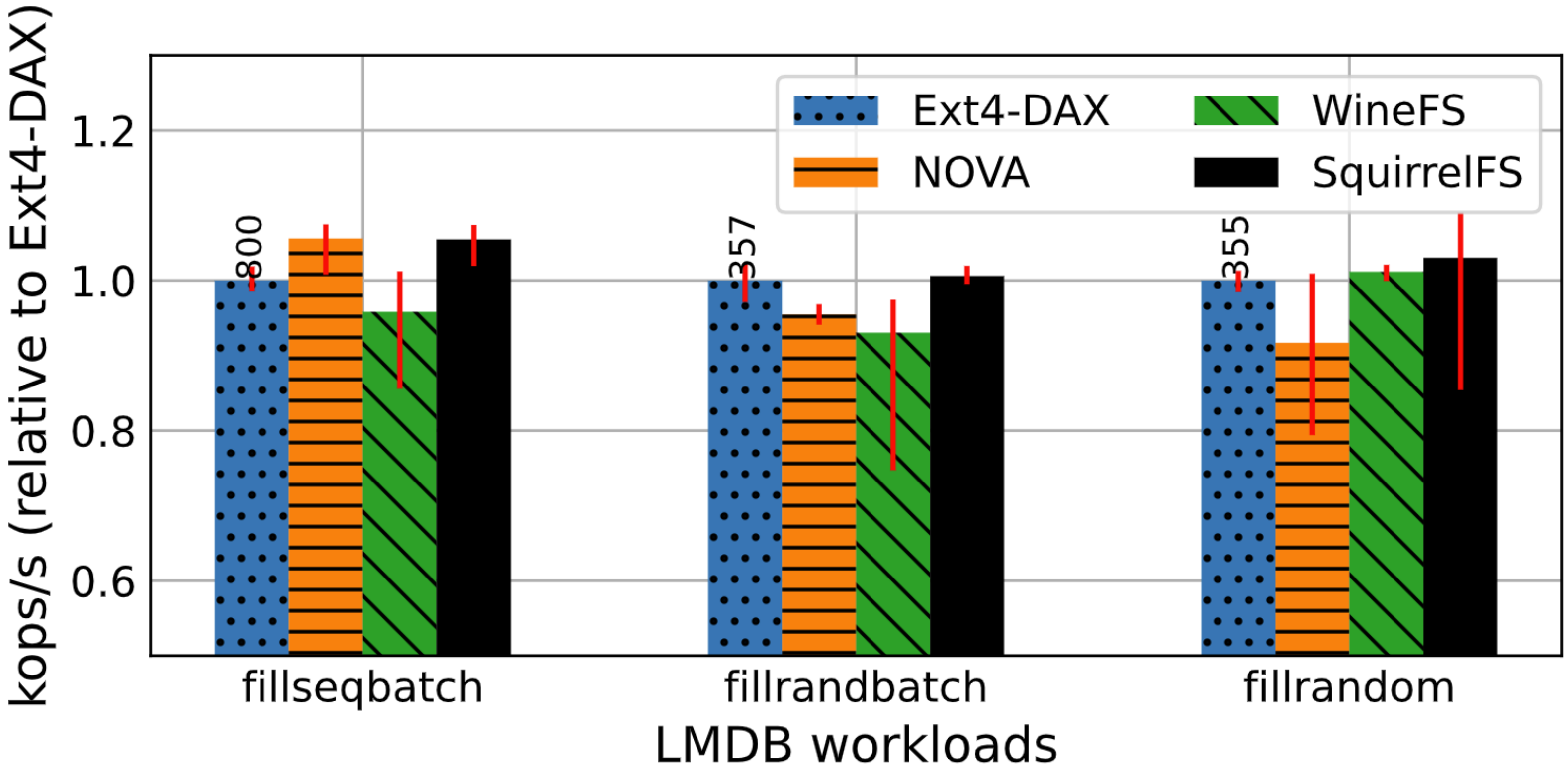
Macrobenchmark: filebench



Application benchmark: RocksDB



Application benchmark: LMDB



SquirrelFS mount times

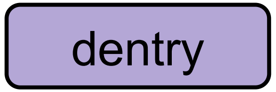
	System state	Mount time (s)
Normal mount	mkfs	5.80
	Empty	5.51
	Full	30.50
Recovery mount	Empty	5.76
	Full	55.50

Ordering for crash consistency

Running example: creating a new file

Ordering for crash consistency

Running example: creating a new file

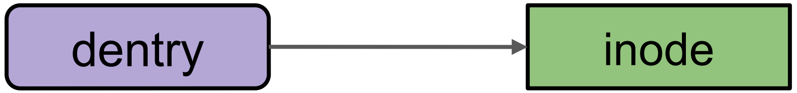


Ordering for crash consistency

Running example: creating a new file

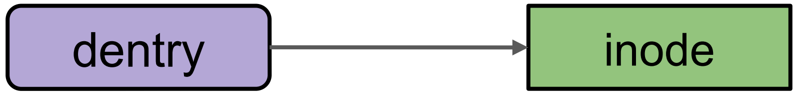
Ordering for crash consistency

Running example: creating a new file



Ordering for crash consistency

Running example: creating a new file



Invariant: a directory entry never points to an uninitialized inode

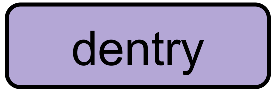
Ordering for crash consistency

Running example: creating a new file

Invariant: a directory entry never points to an uninitialized inode

Ordering for crash consistency

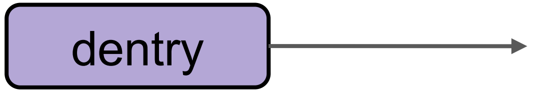
Running example: creating a new file



Invariant: a directory entry never points to an uninitialized inode

Ordering for crash consistency

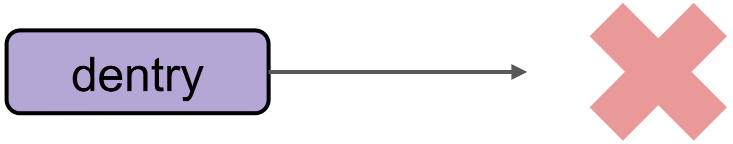
Running example: creating a new file



Invariant: a directory entry never points to an uninitialized inode

Ordering for crash consistency

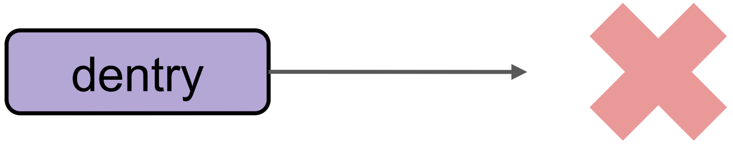
Running example: creating a new file



Invariant: a directory entry never points to an uninitialized inode

Ordering for crash consistency

Running example: creating a new file

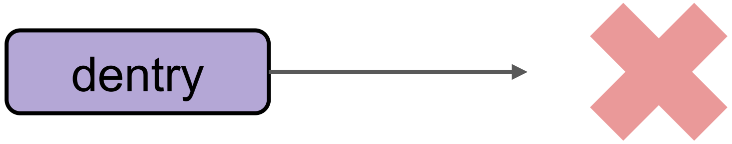


Invariant: a directory entry never points to an uninitialized inode

Ordering rule: inode must be initialized at the same time or before directory entry pointer is set

Ordering for crash consistency

Running example: creating a new file



Invariant: a directory entry never points to an uninitialized inode

Ordering rule: inode must be initialized at the same time or before directory entry pointer is set

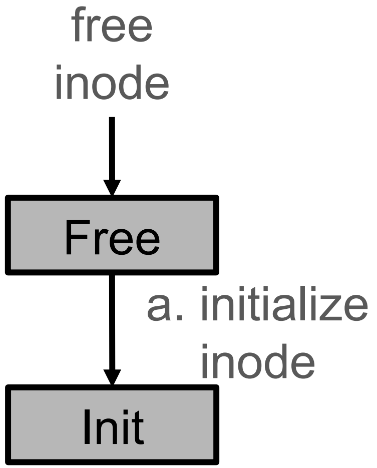
Statically enforcing durable update ordering can prevent crash-consistency bugs

Synchronous soft updates (SSU)

Soft updates: crash consistency from ordered in-place durable updates

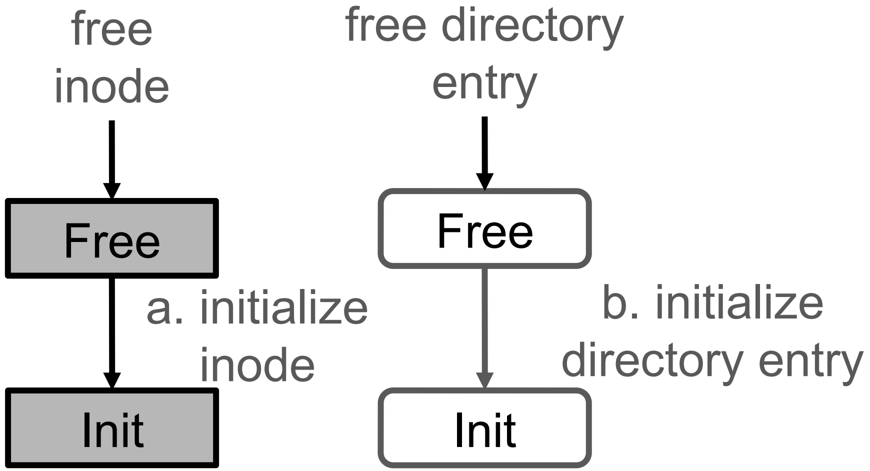
Synchronous soft updates (SSU)

Soft updates: crash consistency from ordered in-place durable updates



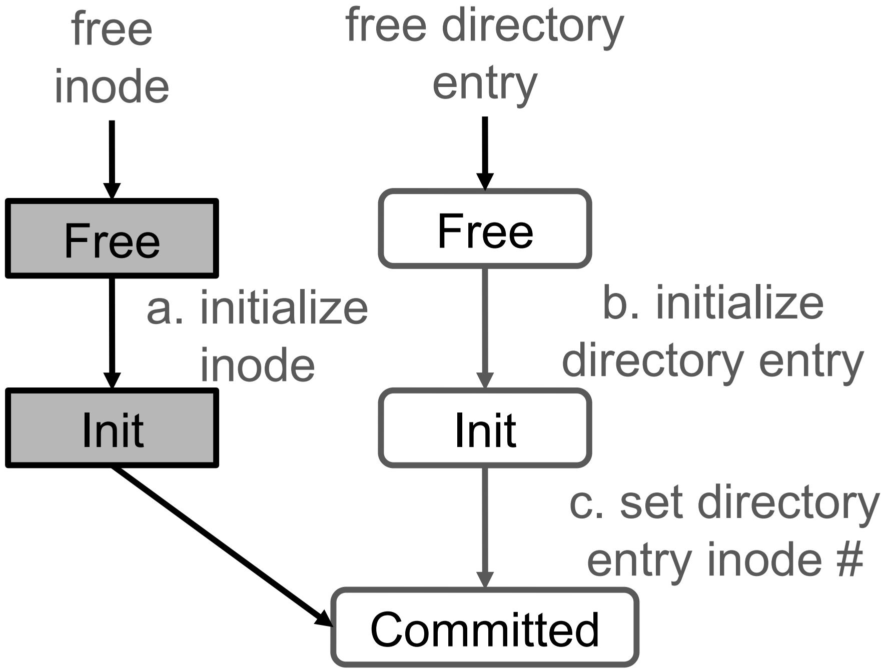
Synchronous soft updates (SSU)

Soft updates: crash consistency from ordered in-place durable updates



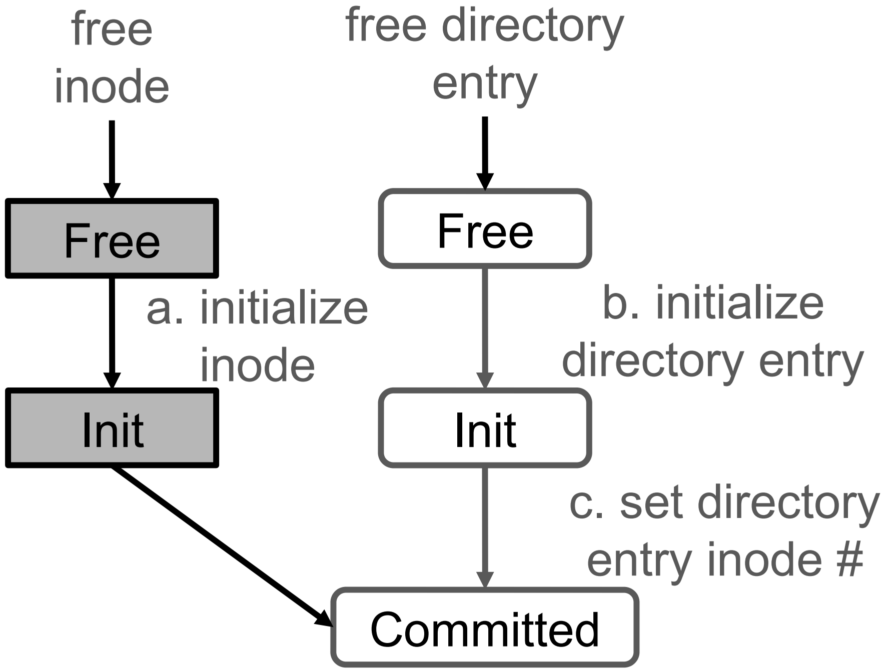
Synchronous soft updates (SSU)

Soft updates: crash consistency from ordered in-place durable updates



Synchronous soft updates (SSU)

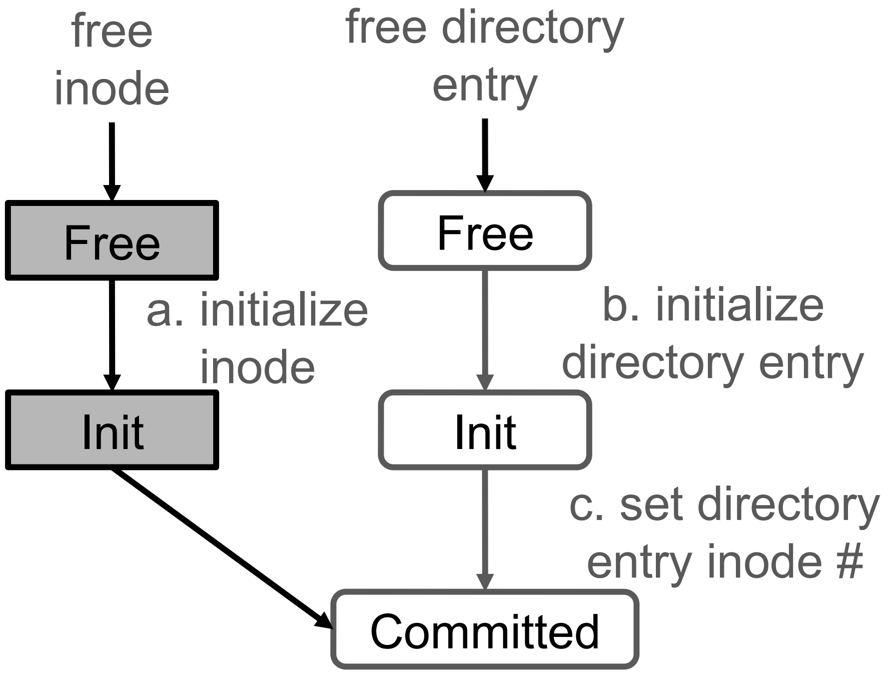
Soft updates: crash consistency from ordered in-place durable updates



Managing update dependencies in **asynchronous** soft updates is notoriously difficult

Synchronous soft updates (SSU)

Soft updates: crash consistency from ordered in-place durable updates

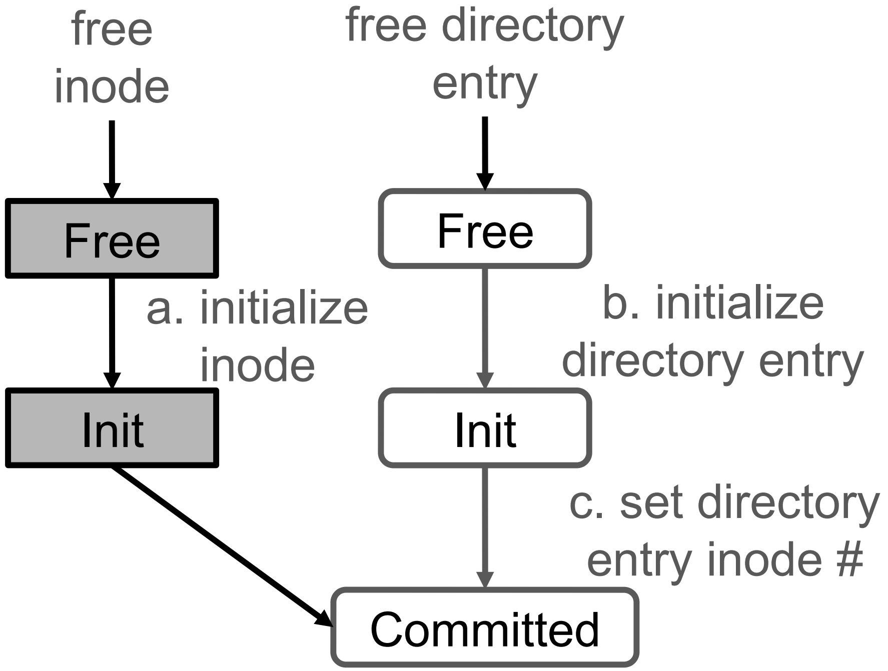


Managing update dependencies in **asynchronous** soft updates is notoriously difficult

Synchronous soft updates eliminates most complexity!

Synchronous soft updates (SSU)

Soft updates: crash consistency from ordered in-place durable updates

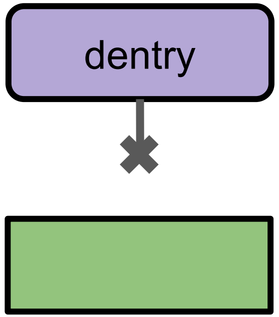


Managing update dependencies in **asynchronous** soft updates is notoriously difficult

Synchronous soft updates eliminates most complexity!

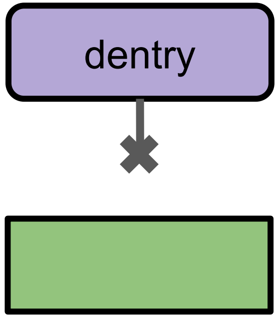
Fast **persistent memory** storage enables performant synchrony

The typestate pattern



without typestate	struct inode
with typestate	Inode<Clean, Free>

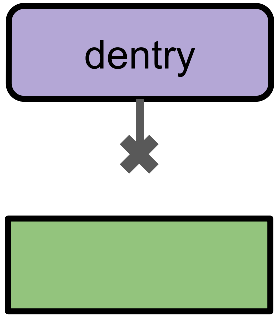
The typestate pattern



without typestate	struct inode
with typestate	Inode<Clean, Free>

Are there unflushed updates?

The typestate pattern

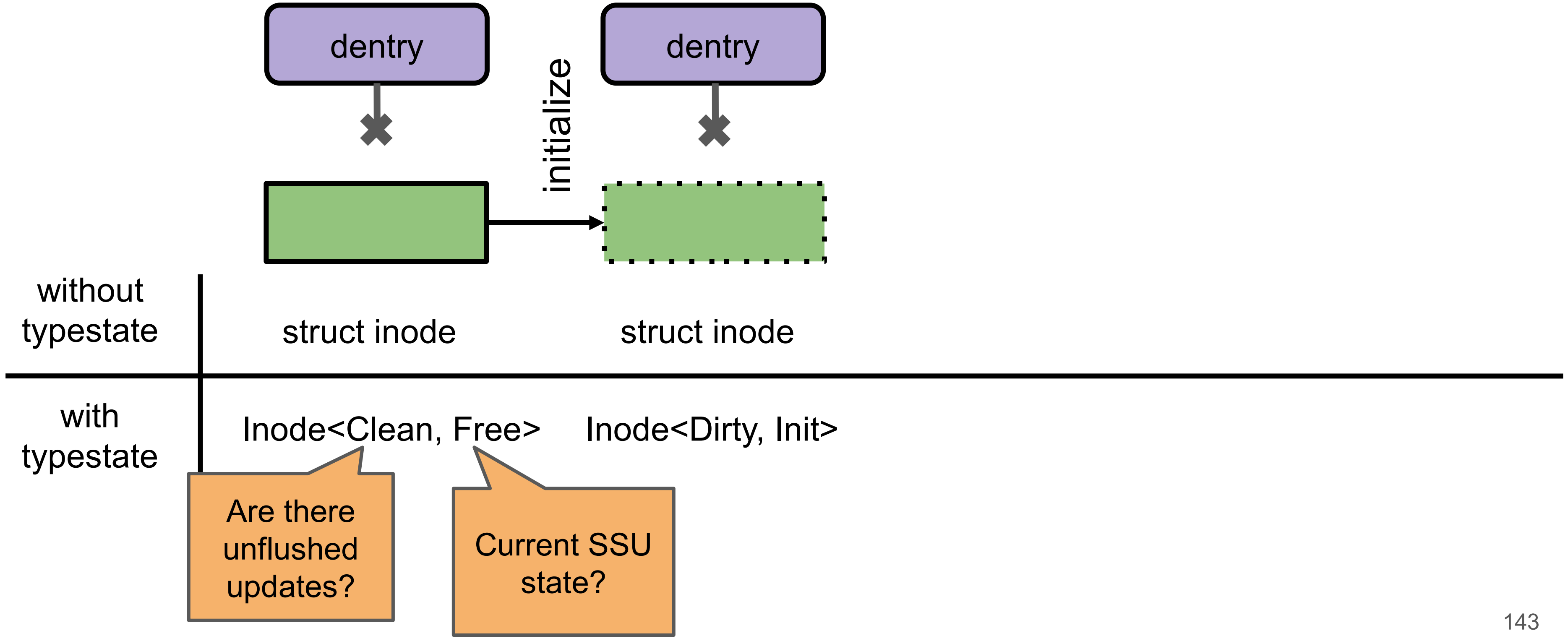


without typestate	struct inode
with typestate	Inode<Clean, Free>

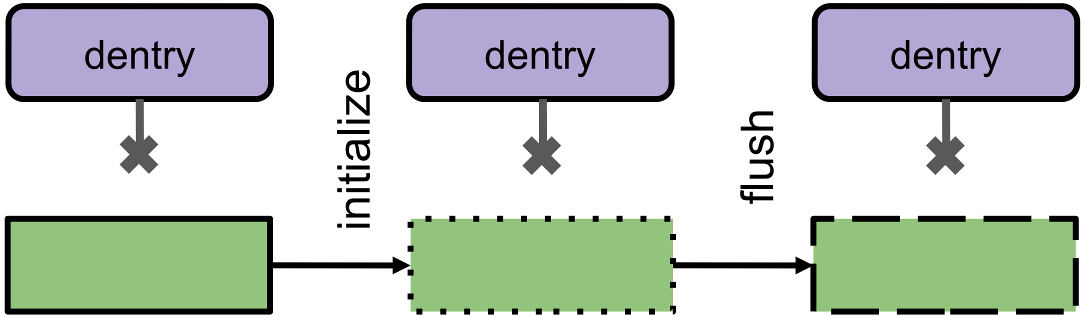
Are there unflushed updates?

Current SSU state?

The typestate pattern



The typestate pattern

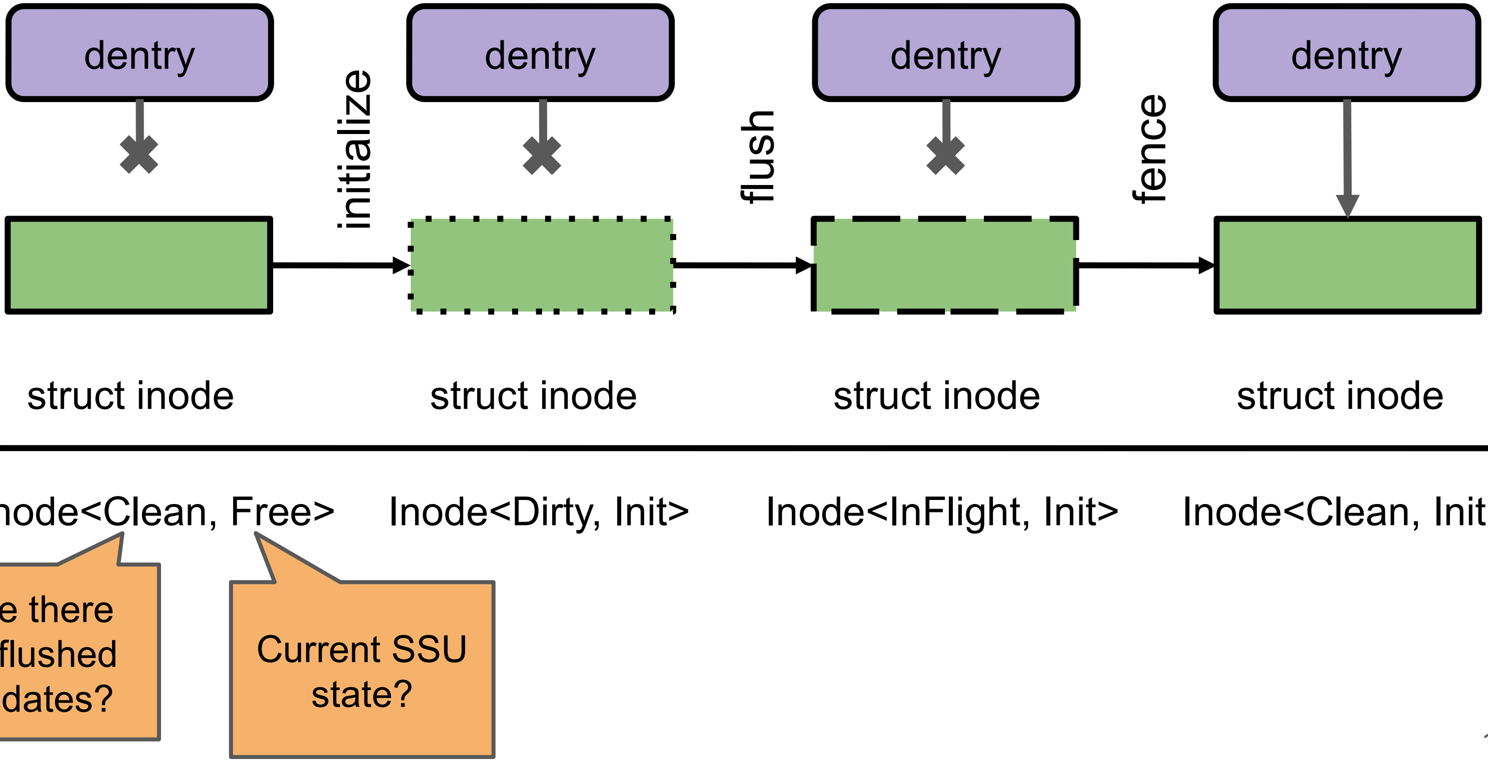


without typestate	struct inode	struct inode	struct inode
with typestate	Inode<Clean, Free>	Inode<Dirty, Init>	Inode<InFlight, Init>

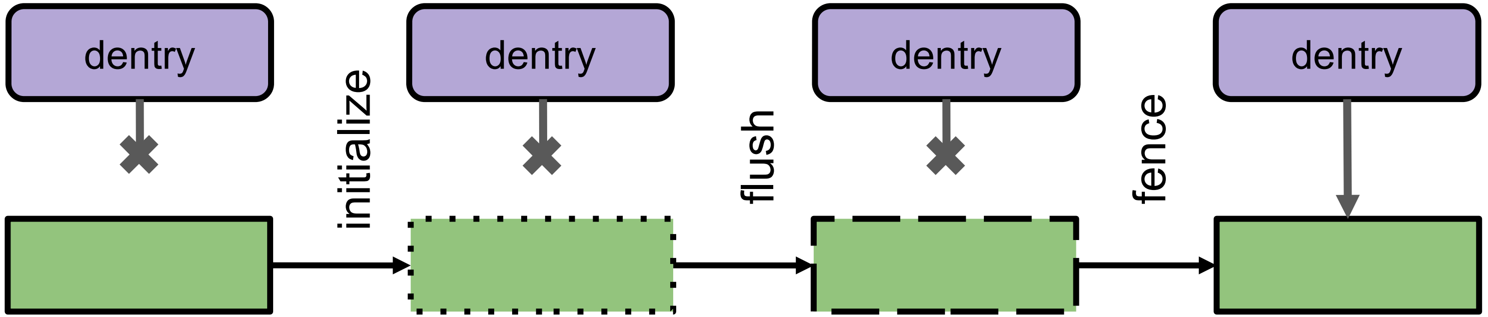
Are there unflushed updates?

Current SSU state?

The typestate pattern



The typestate pattern



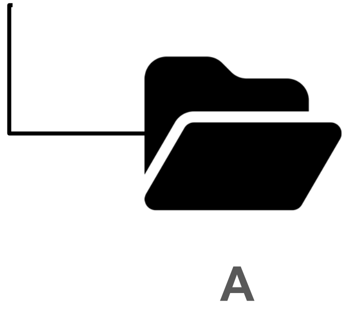
without typestate	struct inode	struct inode	struct inode	struct inode
with typestate	Inode<Clean, Free>	Inode<Dirty, Init>	Inode<InFlight, Init>	Inode<Clean, Init>

Ordering encoded in function signatures:

```
impl Inode<Clean,Free> {fn init(self) -> Inode<Dirty, Init> {...}}
```

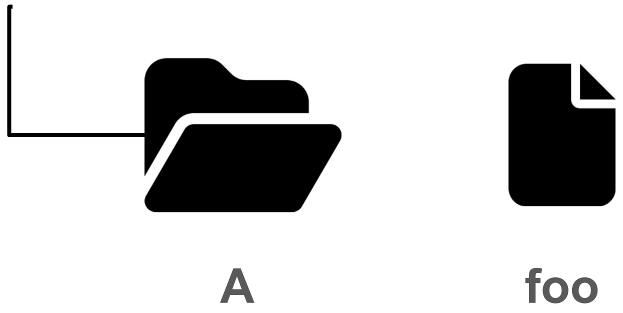
Ordering for crash consistency

Running example: creating a file



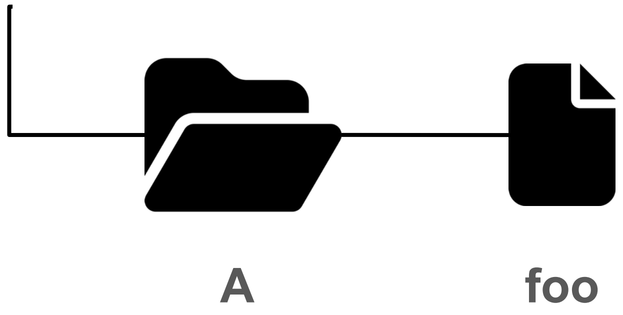
Ordering for crash consistency

Running example: creating a file



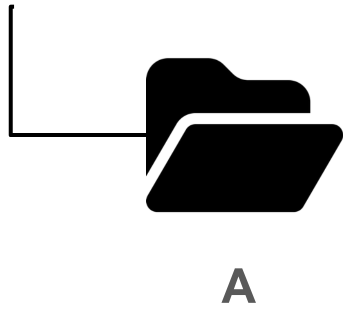
Ordering for crash consistency

Running example: creating a file



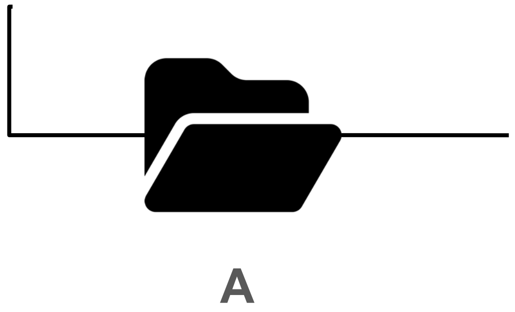
Ordering for crash consistency

Running example: creating a file



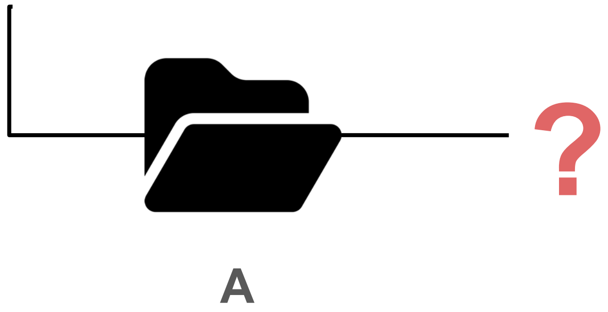
Ordering for crash consistency

Running example: creating a file



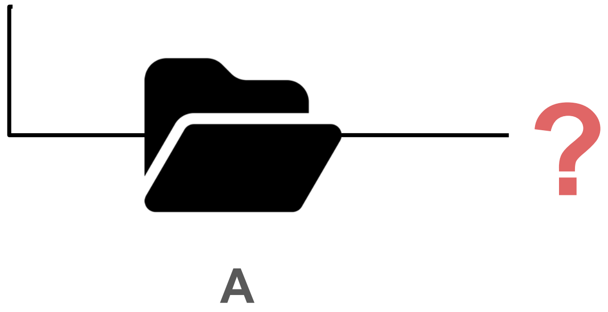
Ordering for crash consistency

Running example: creating a file



Ordering for crash consistency

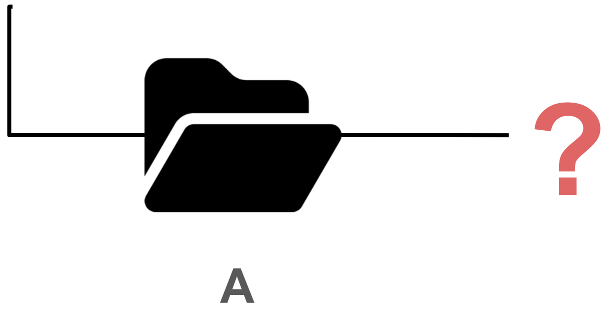
Running example: creating a file



Creating link from A → foo is dependent on initialization of foo

Ordering for crash consistency

Running example: creating a file

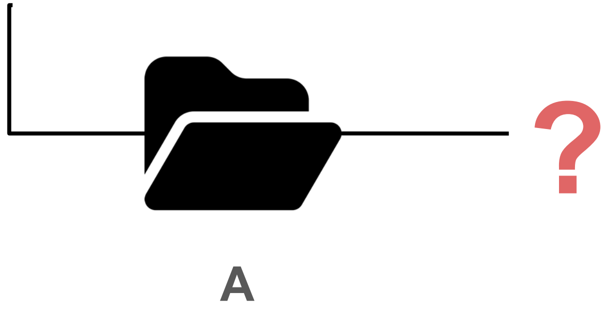


Creating link from A → foo is dependent on initialization of foo

Just two swapped operations can cause serious bugs!

Ordering for crash consistency

Running example: creating a file



Creating link from A → foo is dependent on initialization of foo

Just two swapped operations can cause serious bugs!

Statically enforcing durable update ordering can prevent crash-consistency bugs

Compilation times

System (unverified)	Lines of code	Compilation time (s)
Ext4	45K	38
NOVA	16K	20
WineFS	9K	13
SquirrelFS	**7.5K**	**10**

System (verified)	Lines of code	Verification time (**hours**)
FSCQ	31K	11
VeriBetrKV	45K	1.8