Using Dynamically Layered Definite Releases for Verifying the RefFS File System

Mo Zou[1,2], Dong Du[1,2], Mingkai Dong[1,2], Haibo Chen[1,2,3]

1 IPADS, Shanghai Jiao Tong University

2 Engineering Research Center for Domain-specific Operating Systems

3 Huawei Technologies Co. Ltd

File Systems are Important but Complex

Widely deployed yet hard to be bug-free

Verification is a promising approach

- AtomFS [SOSP'19]
- DaisyNFS [OSDI'22]

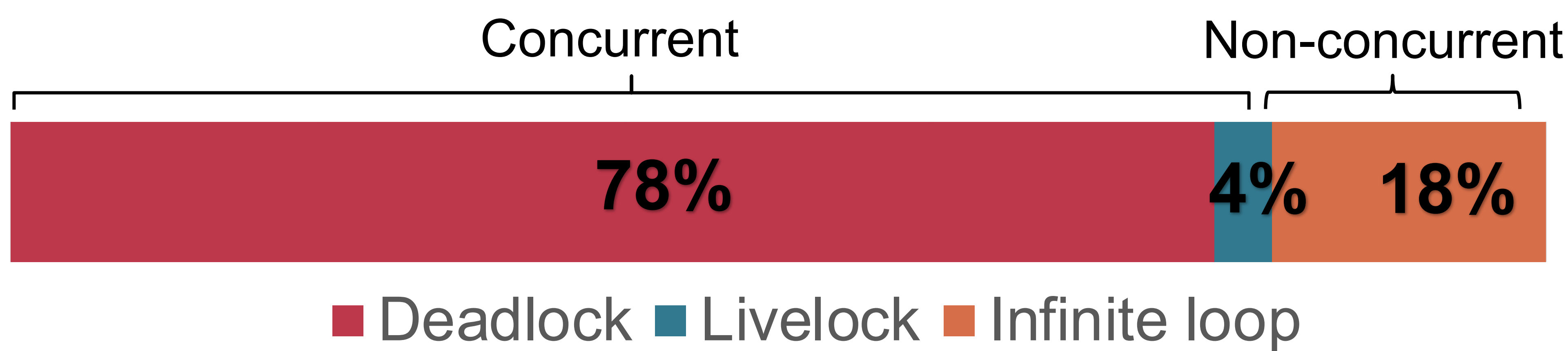
} Only safety property

Goal: verify the liveness of a concurrent file system

- Each operation terminates under fair scheduling

A Study of Termination Bugs

213 termination bugs in Linux FSs (2020-2023)



Deadlocks are dominant

A thread becomes blocked, waiting for an action that never happens; none of involved threads can make progress

A Study of Termination Bugs

This talk: focuses on deadlocks

Observations

Observation1: ad-hoc synchronization

Observation2: nested waiting

Observation3: dynamic waiting order

Observation1: Ad-hoc Synchronization

46% of deadlocks involve ad-hoc synchronization

```
while (1) {
    …
    if (cond)
        break;
}
```

Transaction completion, flushing of dirty inodes or others

Unlike lock/unlock, no specific pattern; hard to analyze

Observation2: Nested Waiting

79% of deadlocks involve nested waiting

Task A waits for Task B; Task B also waits for some task

```
// unlink (or rmdir)
inode_lock(parent);
...
inode_lock(child);
...
inode_unlock(parent);
```

Task A requests

Wait for

unlink

Wait for

Task B

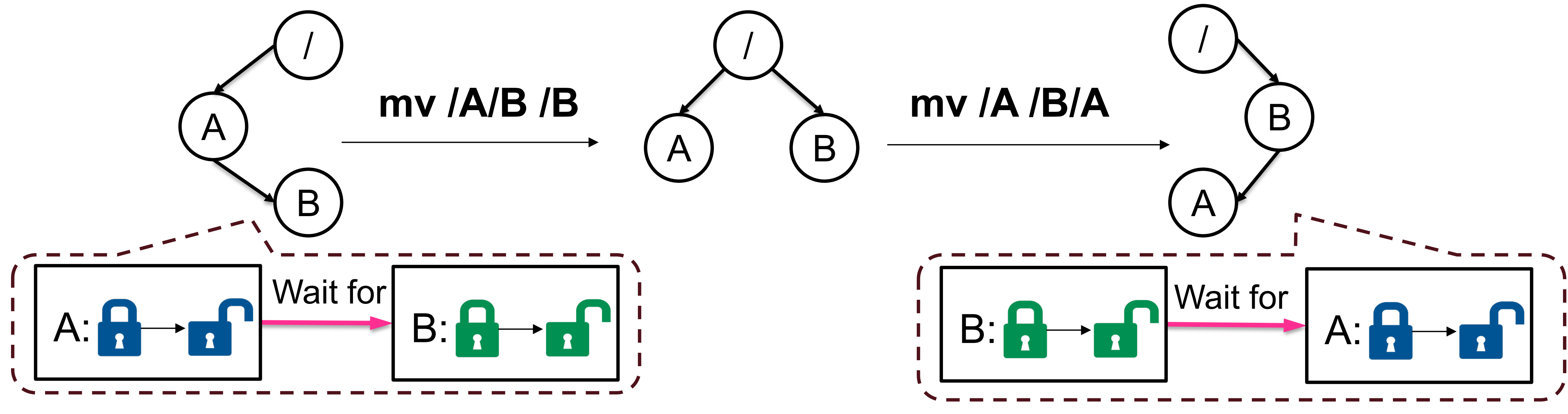
✗ Circular wait

✓ A global order of waits-for dependencies (but still absent)

Observation 3: Dynamic Waiting Order

In 8% of deadlocks, the waiting order is not statically known

For instance, the parent-child waiting order is dynamic



Challenging to formalize and reason about

Limitations of Previous Work

	Ad-hoc sync	Nested waiting (modularly)	Dynamic waiting order
Deadlock-freedom verification work [ESOP'19, POPL'22]	✗	✓	Limited support
LiLi [POPL'16, POPL'18]	✓	✗	
TaDA Live [TOPLAS'22]	✓	✓	

Only small examples

No executable or mechanized proof

Modular liveness verification of FS remains an open problem

Contributions

MoLi*: a framework for verifying concurrent FSs

- Acyclic waits-for graph

RefFS: the first to guarantee both safety and liveness

A protocol-level proof of Linux VFS's directory locking rules

- Found a bug; confirmed and fixed

* Modular Liveness verification

Outline

MoLi*: a framework for verifying concurrent FSs

- Acyclic waits-for graph

RefFS: the first to guarantee both safety and liveness

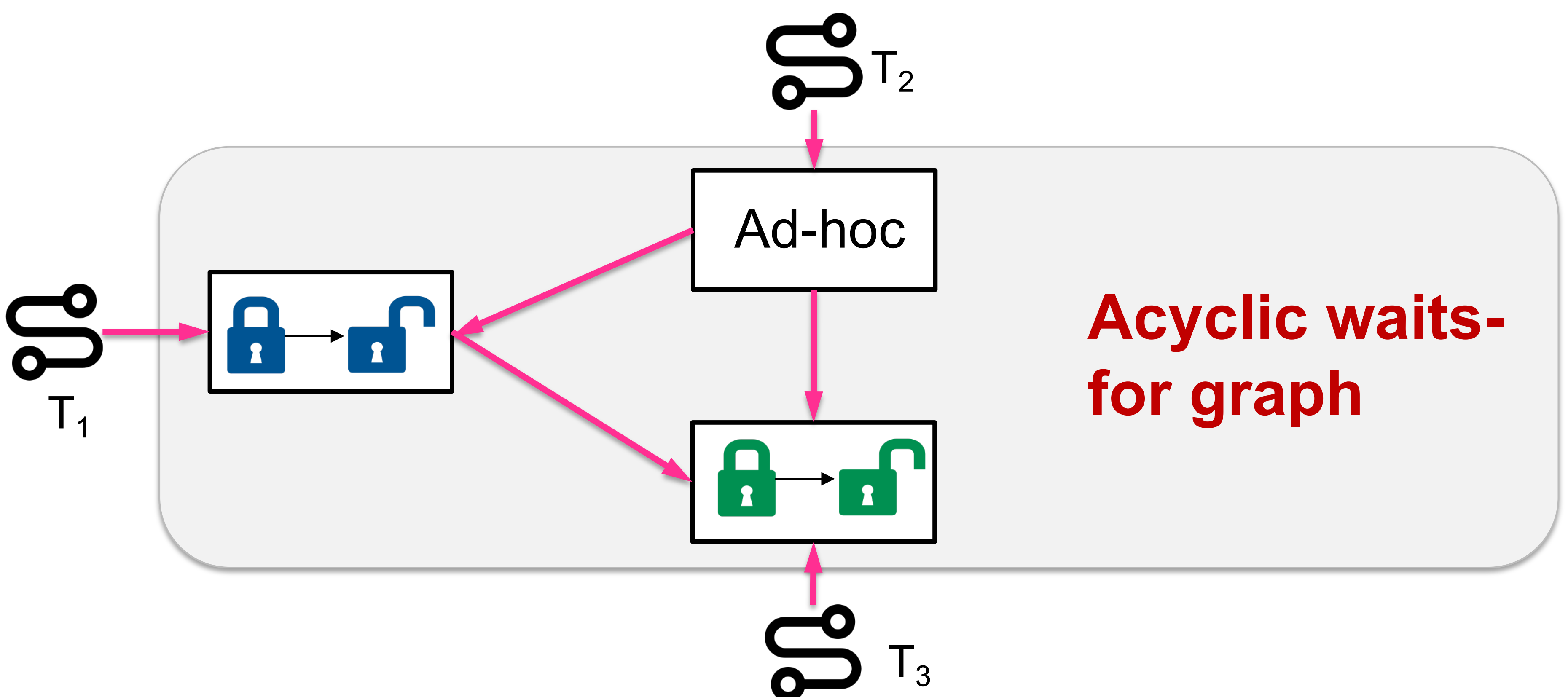
A protocol-level proof of Linux VFS’s directory locking rules

- Found a bug; confirmed and fixed

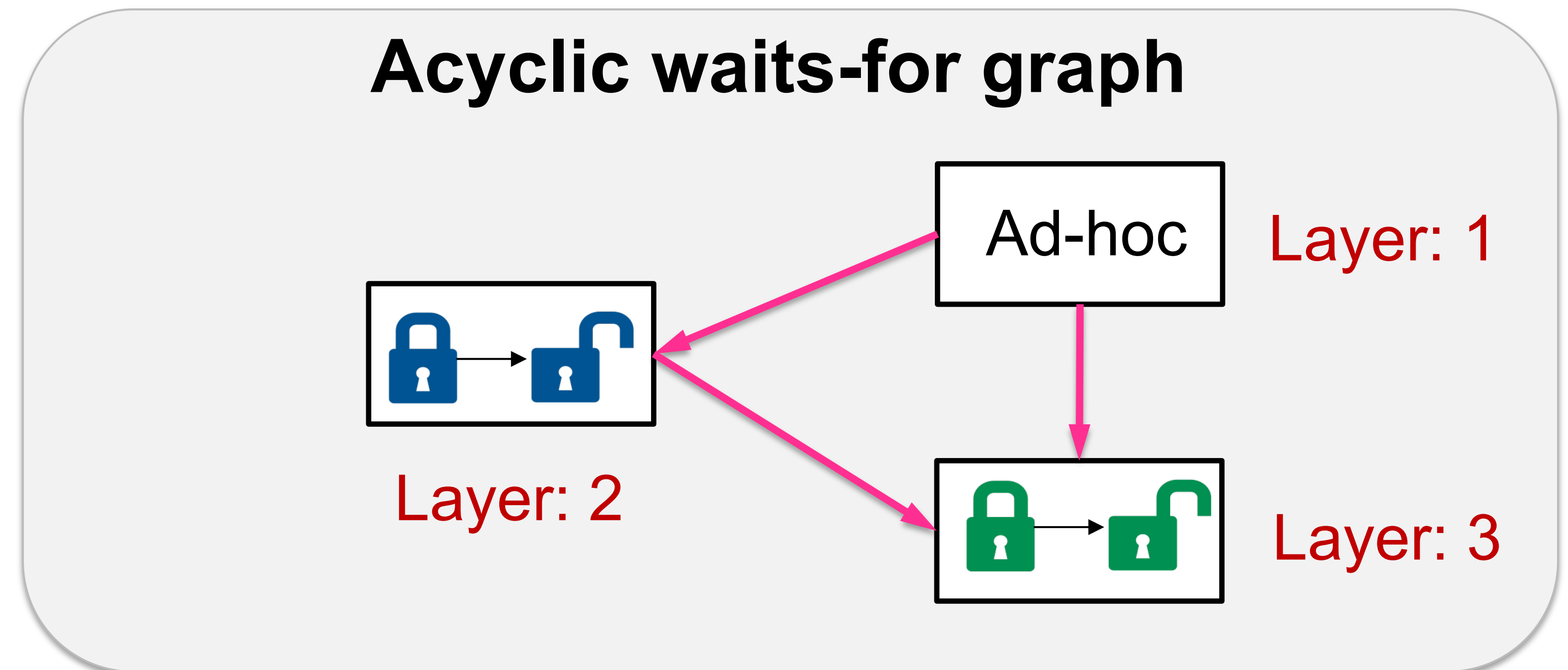
* Modular Liveness verification

Acyclic Waits-for Graph

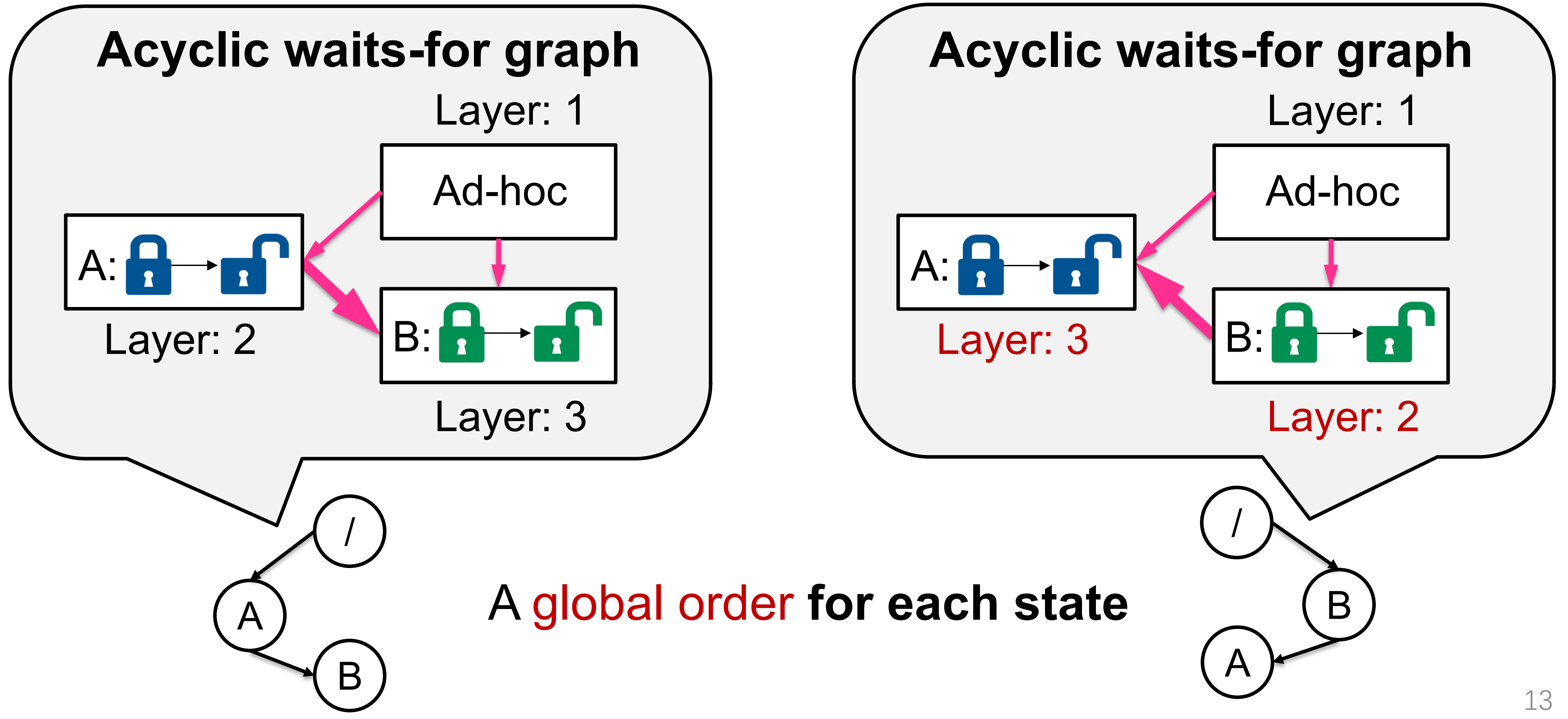
Blocked thread   Wait for   Unblocking action



Layering of Unblocking Actions

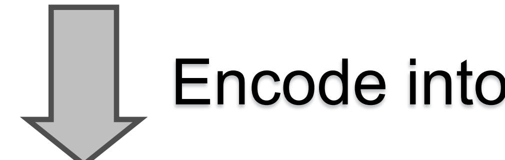


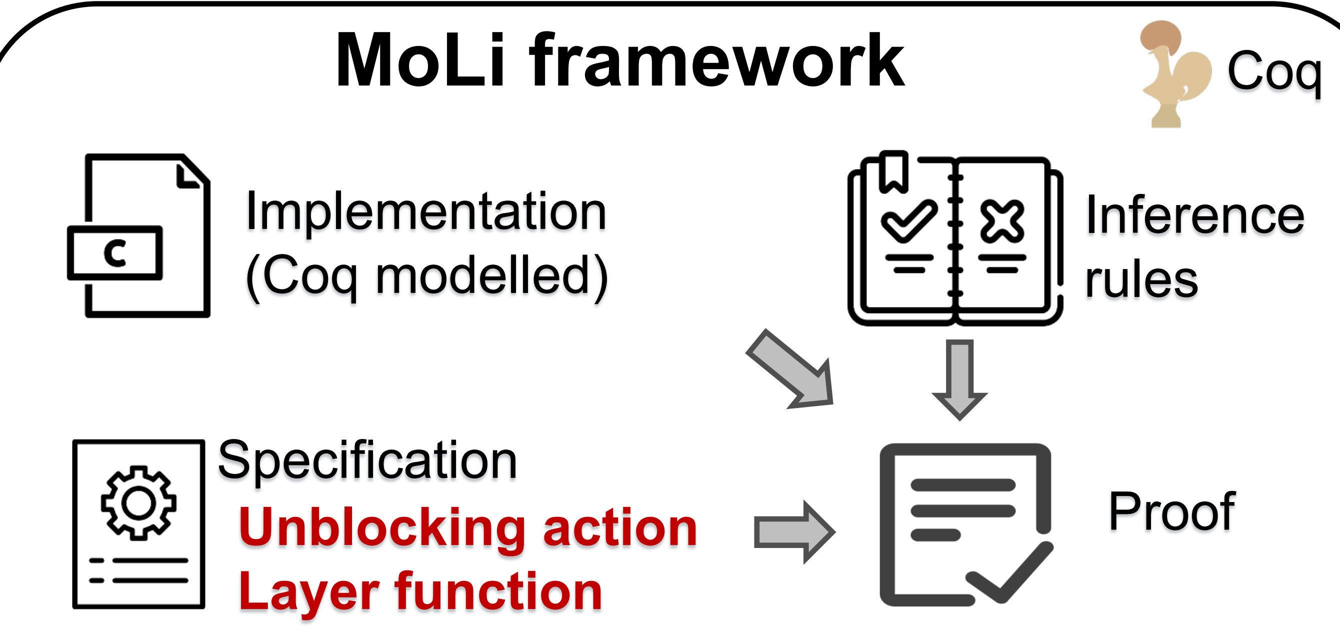
Dynamic Layering based on State

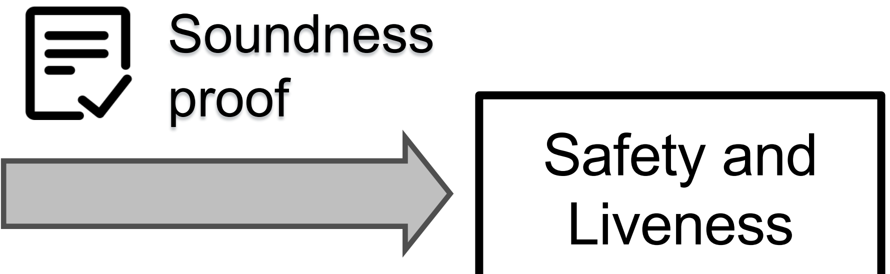


The MoLi Framework

Acyclic waits-for graph methodology







Outline

MoLi*: a framework for verifying concurrent FSs

- Acyclic waits-for graph

RefFS: the first to guarantee both safety and liveness

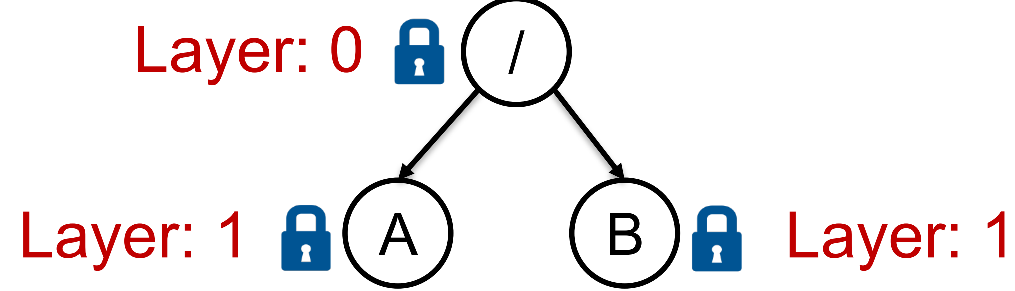
A protocol-level proof of Linux VFS's directory locking rules

- Found a bug; confirmed and fixed

* Modular Liveness verification

Application to a Concurrent File System—Specifying Parent-Child Order

Define a waits-for order between inode locks

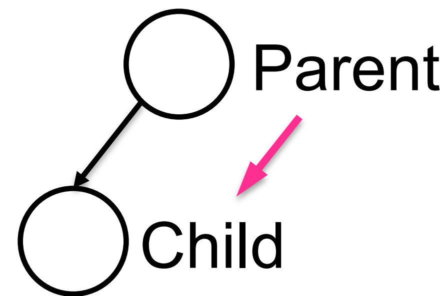


Layer = distance from root

Layers dynamically defined on state

Acyclic by definition

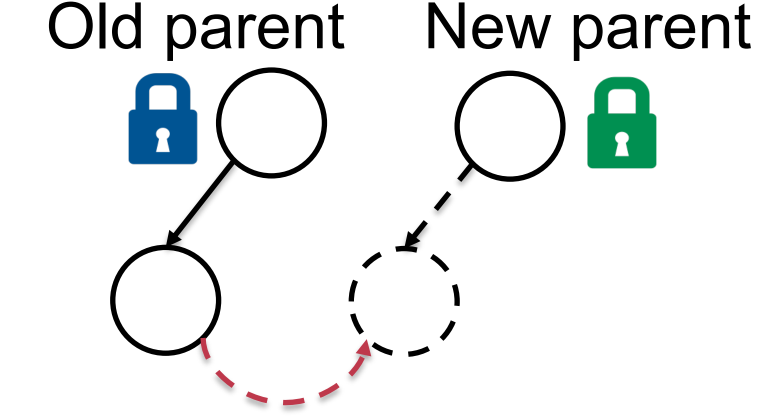
Order1: parent-child order

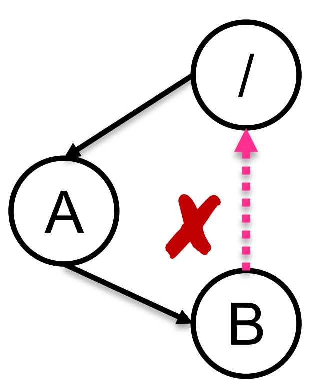


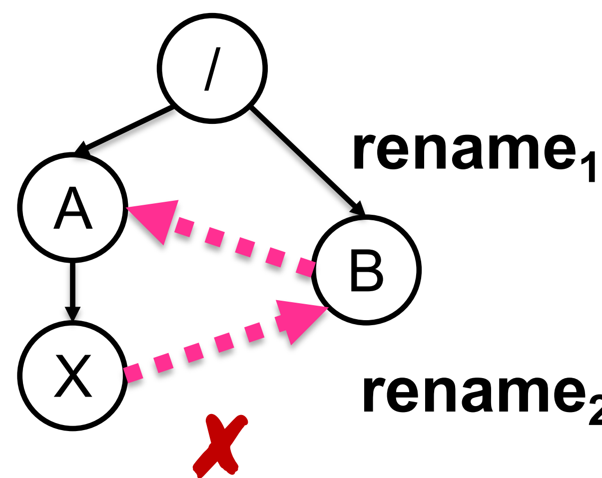
Locking Order for Rename

Order2: old and new parent order

- An order between any two directories
- Transitive with parent-child order
- Concurrent renames



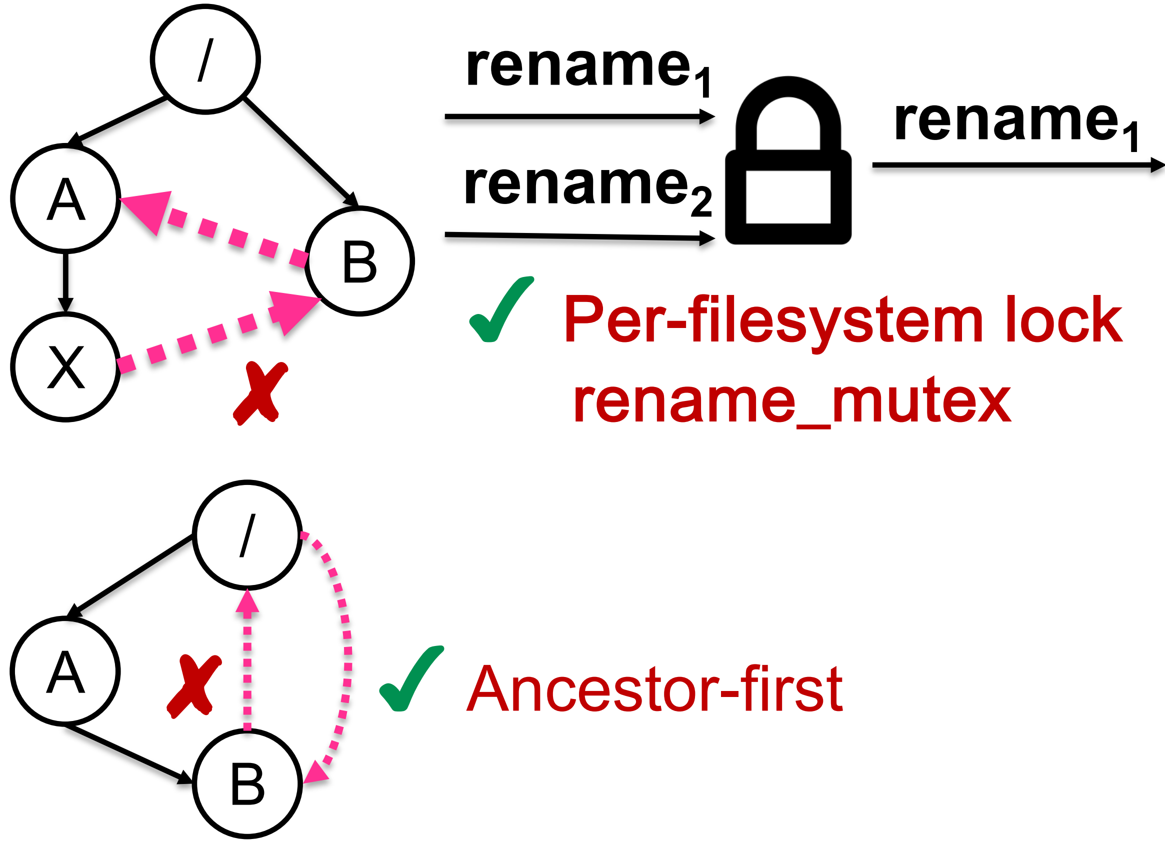




Code for Acquiring the Two Parents

```
1 def lock_rename(old, new){
2     if(old == new) {
3         inode_lock(old);
4         return;
5     }
6     mutex_lock(rename_mutex);
7     if (ancestor(new, old)) {
8         inode_lock(new);
9         inode_lock(old);
10        return;
11    }
12    inode_lock(old);
13    inode_lock(new);
14    return;
15 }
```

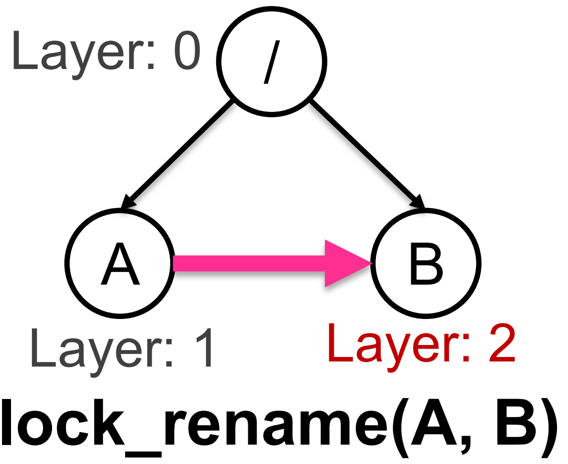
Two parents are the same : acquire one



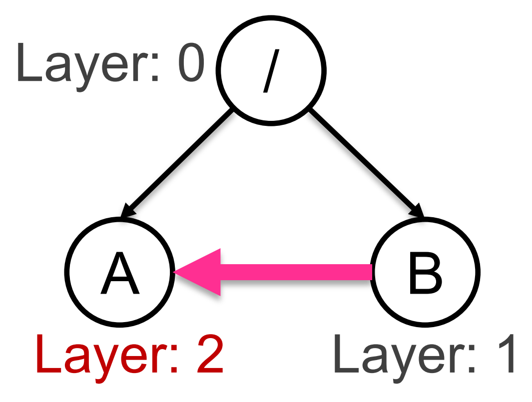
Default order: old parent first

Specifying Rename Order with Ghost State

Problem: order cannot be defined only with FS state



lock_rename(A, B)



lock_rename(B, A)

```
1 def lock_rename(old, new){
…
6     mutex_lock(rename_mutex);
7     if (ancestor(new, old)) {
8         inode_lock(new);
9         inode_lock(old);
10        return;
11  }
```

Ghost state = old → new

```
12    inode_lock(old);
13    inode_lock(new);
```

Clear ghost state

```
14    return;
15 }
```

Approach: use ghost state

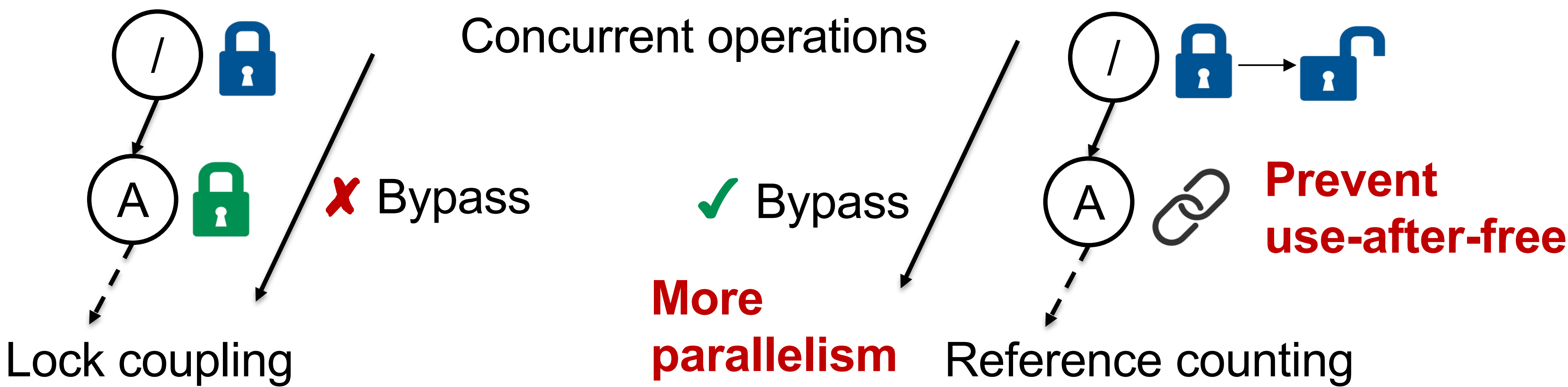
Layer = longest distance from root

Acyclic by construction

The RefFS File System

RefFS: a concurrent, in-memory FS running on fuse

Reference counting for highly concurrent traversals



Outline

MoLi*: a framework for verifying concurrent FSs

- Acyclic waits-for graph

RefFS: the first to guarantee both safety and liveness

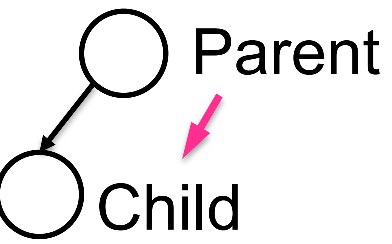
A protocol-level proof of Linux VFS's directory locking rules

- Found a bug; confirmed and fixed

* Modular Liveness verification

A Directory Order Bug in Linux VFS

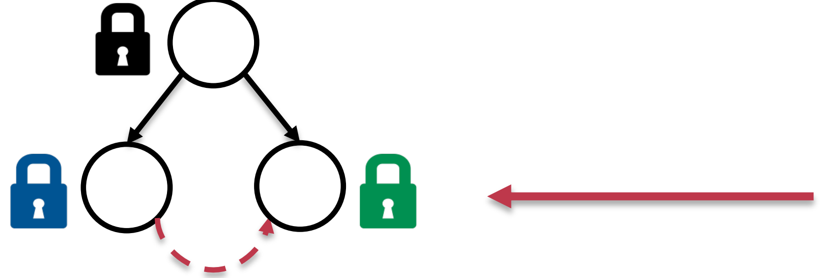
Order1: parent-child order



Order2: unrelated directory order under rename_mutex

Commit 28ecee: rename additionally lock source subdirectory

New order: source/target subdirectories

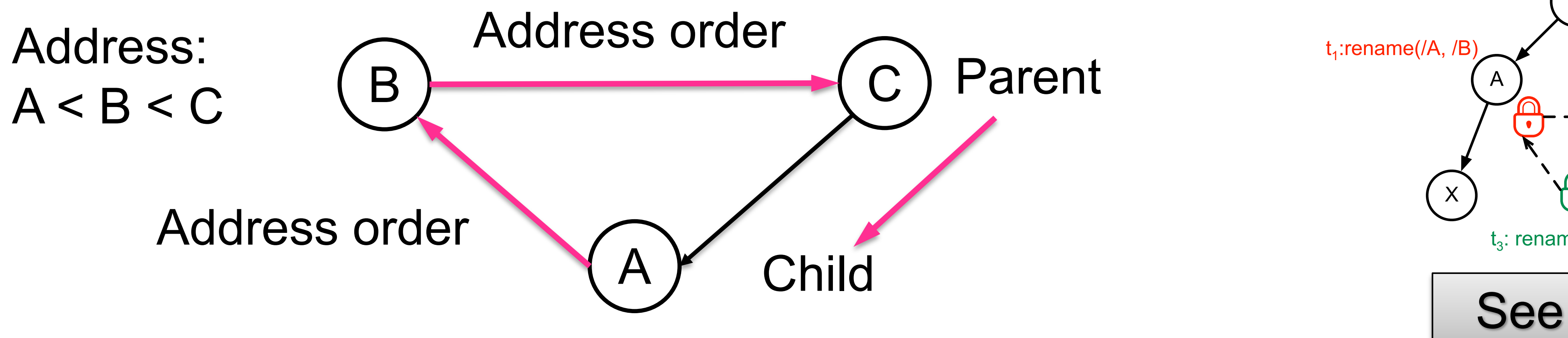


Non-cross-directory rename

Increasing address order for

A Directory Order Bug in Linux VFS

Problem: address order not transitive with parent-child order



Bug confirmed and fixed (we prove the fix correct)

Proof Patch for the Linux VFS

Linux Doc has a proof, but still misses the bug

- Proof by contradiction

> Suppose deadlocks are possible. Consider the minimal deadlocked set of threads. […] we have a cross-directory rename that locked Dn and blocked on attempt to lock D1 [..] Dn and D1 would have to be among those. Which pair could it be?
> It can't be the parents – indeed, since […]
> It can't be a parent and its child; otherwise we would've had a loop, since […]
> …
> That concludes the proof, since the set of operations with the properties required for a minimal deadlock can not exist.

- Detailed but lacks intuition

We submit a proof patch* to the Linux Doc

- Define the locking order; effective in preventing bugs

* https://lore.kernel.org/linux-fsdevel/20240412161000.33148-1-lostzoumo@gmail.com/

Evaluation

How much is the proof effort?

How well does RefFS perform?

Proof Effort

MoLi: reuse a prior framework (for AtomFS); add liveness

RefFS: reuse AtomFS; prove reference counting and liveness

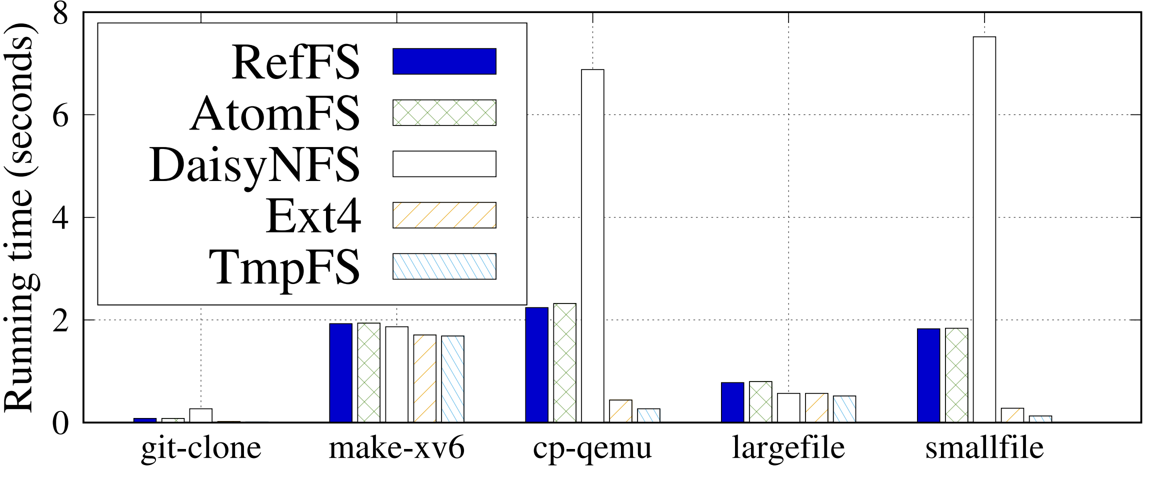
- 0.4K lines of code, 32K lines of proof
- Proof ratio 80:1 (AtomFS is 100:1)

Performance

RefFS achieves overall better performance than AtomFS

- Reference counting instead of lock coupling

Slower than ext4/tmpfs due to lacked optimizations



Application performance

More in the Paper

Program logic of MoLi

- Rely-guarantee style liveness reasoning
- Modular reasoning about nested waiting
- Support for infinite loop and livelock

Proof of RefFS

- Reference counting
- Non-atomic abstraction

Summary

MoLi: verifying liveness based on an acyclic waits-for graph

RefFS: the first concurrent FS to guarantee liveness

- Dynamic layering of lock release actions

Application to the Linux VFS

- We believe the methodology is applicable beyond FS

https://ipads.se.sjtu.edu.cn/projects/reffs

Thanks!