Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System

Mo Zou[1], Haoran Ding[1], Dong Du[1], Ming Fu[2], Ronghui Gu[3], Haibo Chen[12]

1 IPADS, Shanghai Jiao Tong University

2 Huawei Technologies Co. Ltd

3 Columbia University

File systems are buggy and underspecified

- 40% of FS patches fix bugs [Lu et al., FAST'13]
  - 20% of the bugs are concurrency bugs
    - Hard to eliminate due to many possible interleavings

- POSIX is **vague** about concurrent behavior
  - E.g., unclear whether an operation should be atomic
  - Hard to reason about higher-level applications

Approach: formal verification

- Concurrent implementation meets specification
  - Under arbitrary interleavings
  - Proof checked by proof assistant (Coq)
- Avoid large classes of bugs
- Specification serves as a precise interface

Verification efforts

- File system verification
  - FSCQ project [SOSP’15,SOSP’17,Tej M.S. thesis]
  - Yggdrasil [OSDI’16]
  - Cogent [ASPOLOS’16]

No fine-grained concurrency

- Concurrent system verification
  - CertiKOS [OSDI’16]
  - CSPEC [OSDI’18]

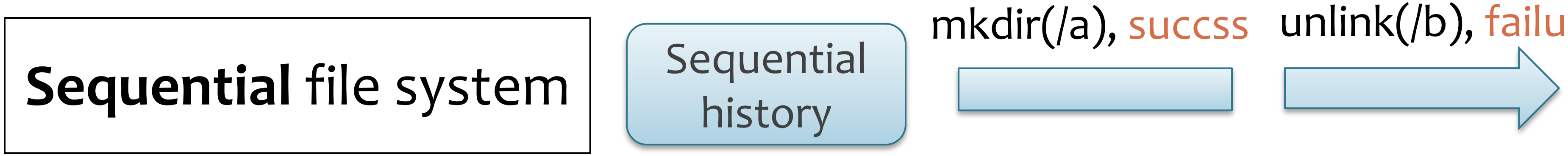
Not applicable to FS

Goal: verify a fine-grained, concurrent file system

Contributions

- **CRL-H: C**oncurrent **R**elation **L**ogic with **H**elpers for concurrent file systems
  - **Helper mechanism**
  - Proofs mechanically checked by Coq
- **AtomFS:** the first verified concurrent FS with fine-grained locking
  - **Fine-grained:** per-inode lock (no crash-safety)
  - **Atomic** interfaces
  - Verified directly in **C language**

How to specify “correct”?



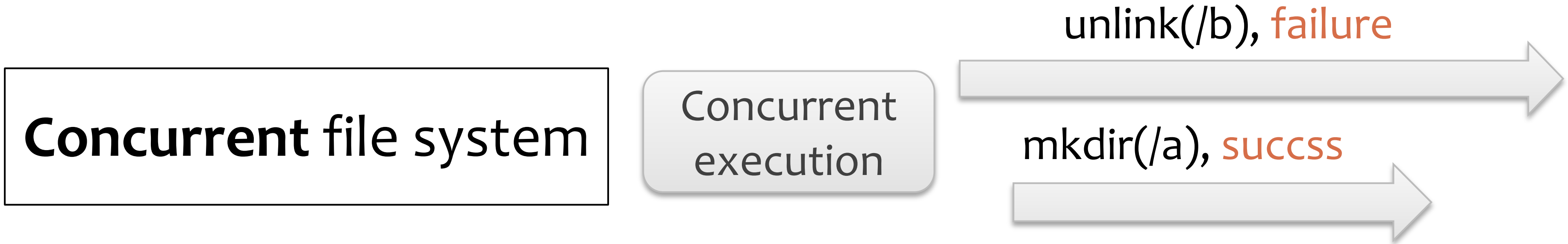
Sequential file system

Sequential history

mkdir(/a), succss

unlink(/b), failure

For a sequential file system, **correct** if sequential history is legal ✓



Concurrent file system

Concurrent execution

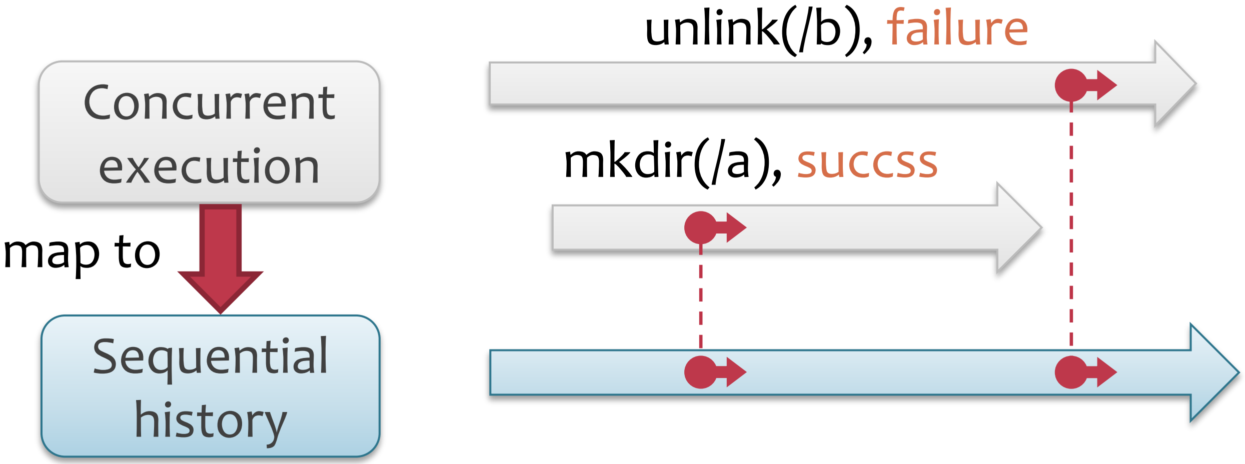
unlink(/b), failure

mkdir(/a), succss

How to describe concurrent via sequential?

This work: “correct” means linearizability

Linearizability: describe concurrent via sequential



Concurrent execution

map to

Sequential history

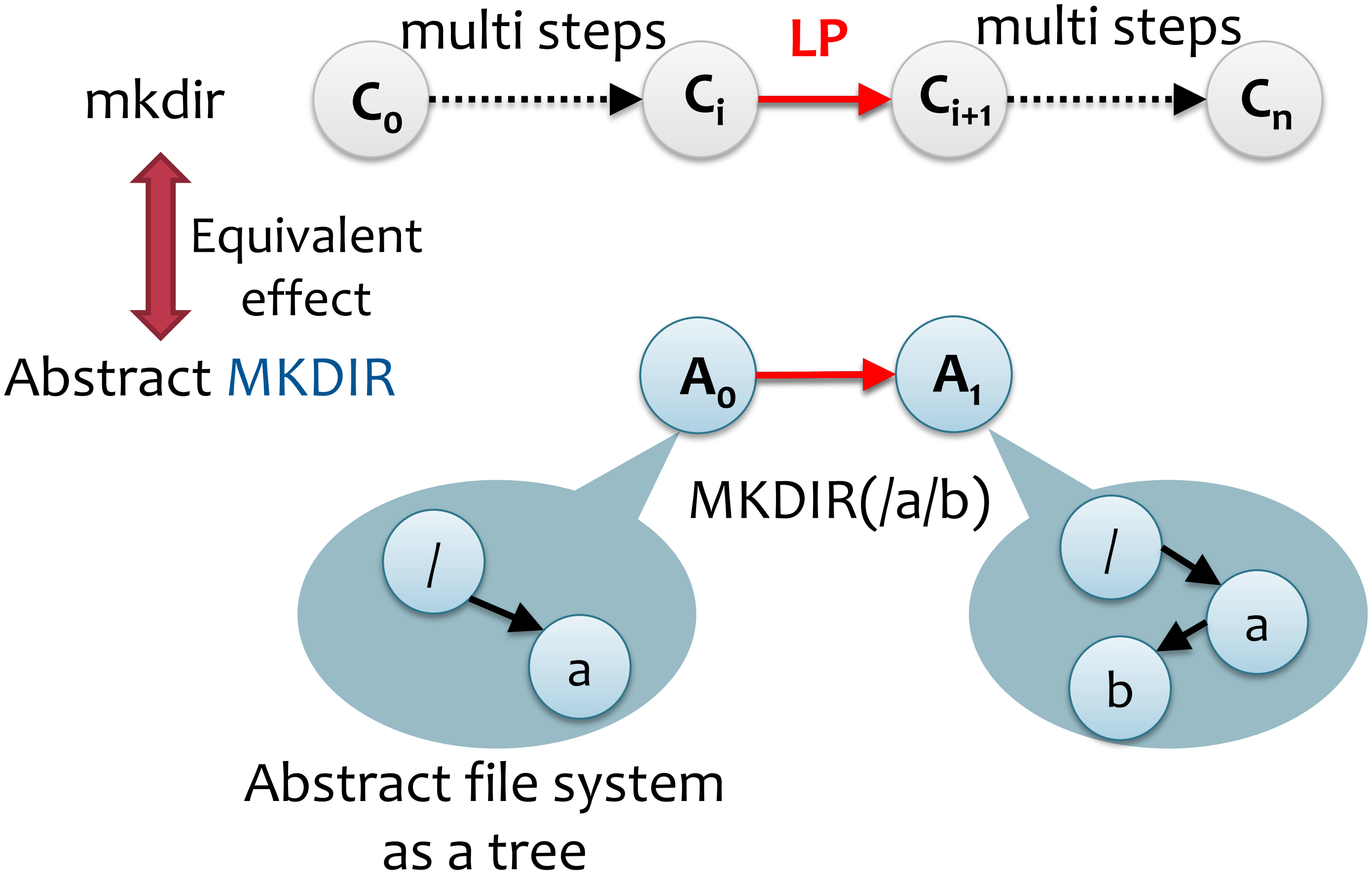
unlink(/b), failure

mkdir(/a), succss

●→ linearization point (LP)---effect happens **atomically**

Correct if equivalent “sequential” history is legal

Prove linearizability via forward simulation



mkdir

C_0 multi steps C_i LP C_{i+1} multi steps C_n

Equivalent effect

Abstract MKDIR

A_0 MKDIR(/a/b) A_1

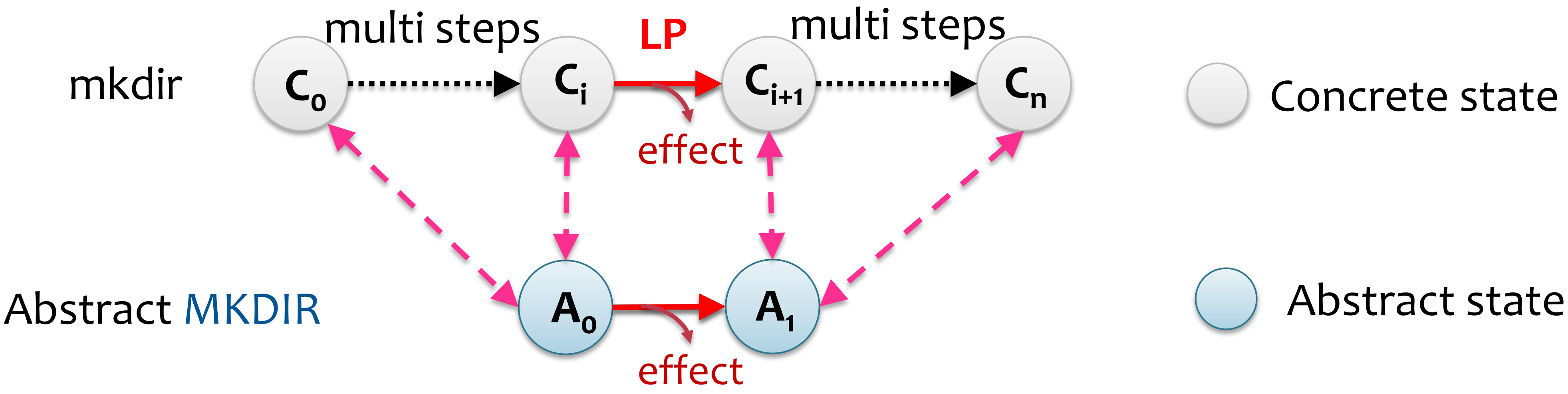
Abstract file system as a tree

○ Concrete state

● Abstract state

Define abstract state and abstract operation

Prove linearizability via forward simulation

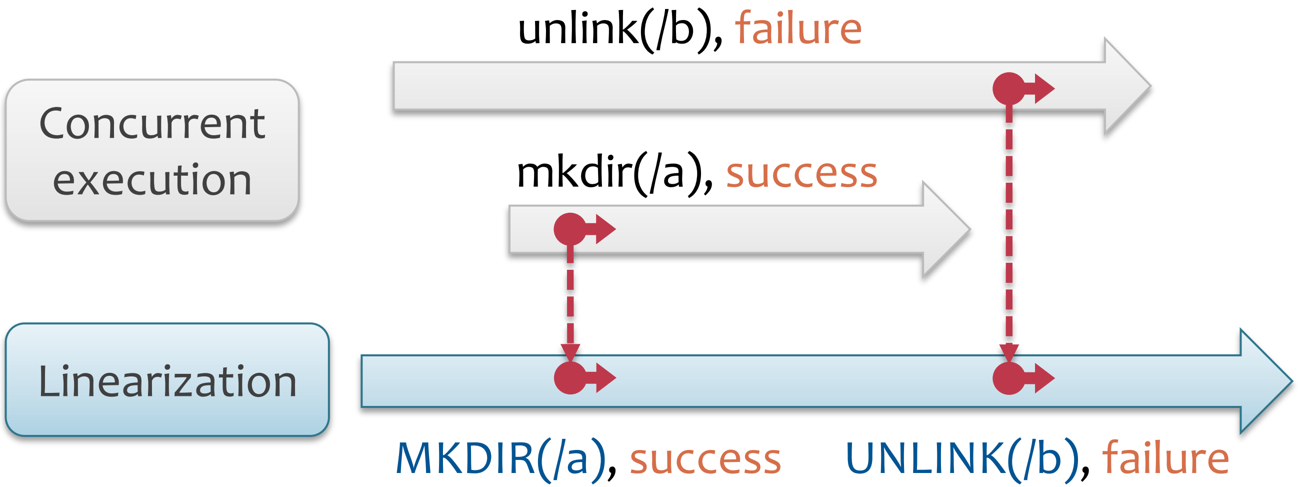


Decide linearization point

Define abstraction relation

mkdir ⟺ MKDIR

Prove linearizability via forward simulation



Concurrent execution

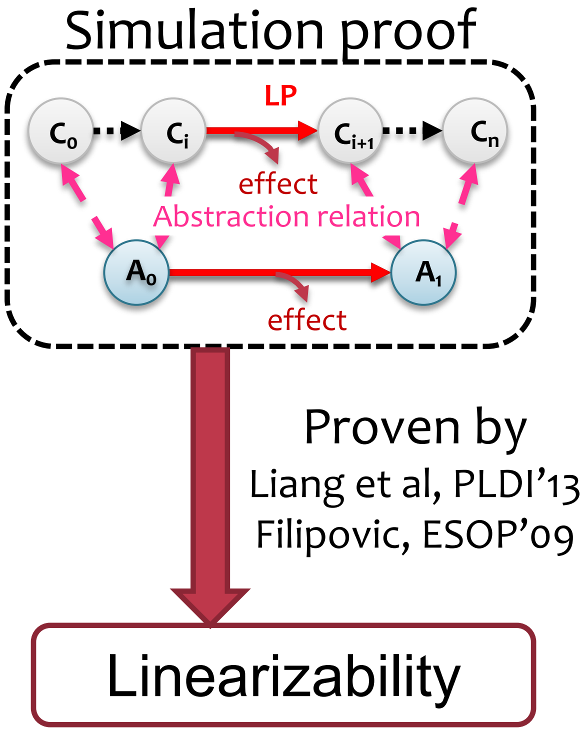
unlink(/b), failure

mkdir(/a), success

Linearization

MKDIR(/a), success

UNLINK(/b), failure



Simulation proof

C_0 C_i C_{i+1} C_n

LP

effect

Abstraction relation

A_0 A_1

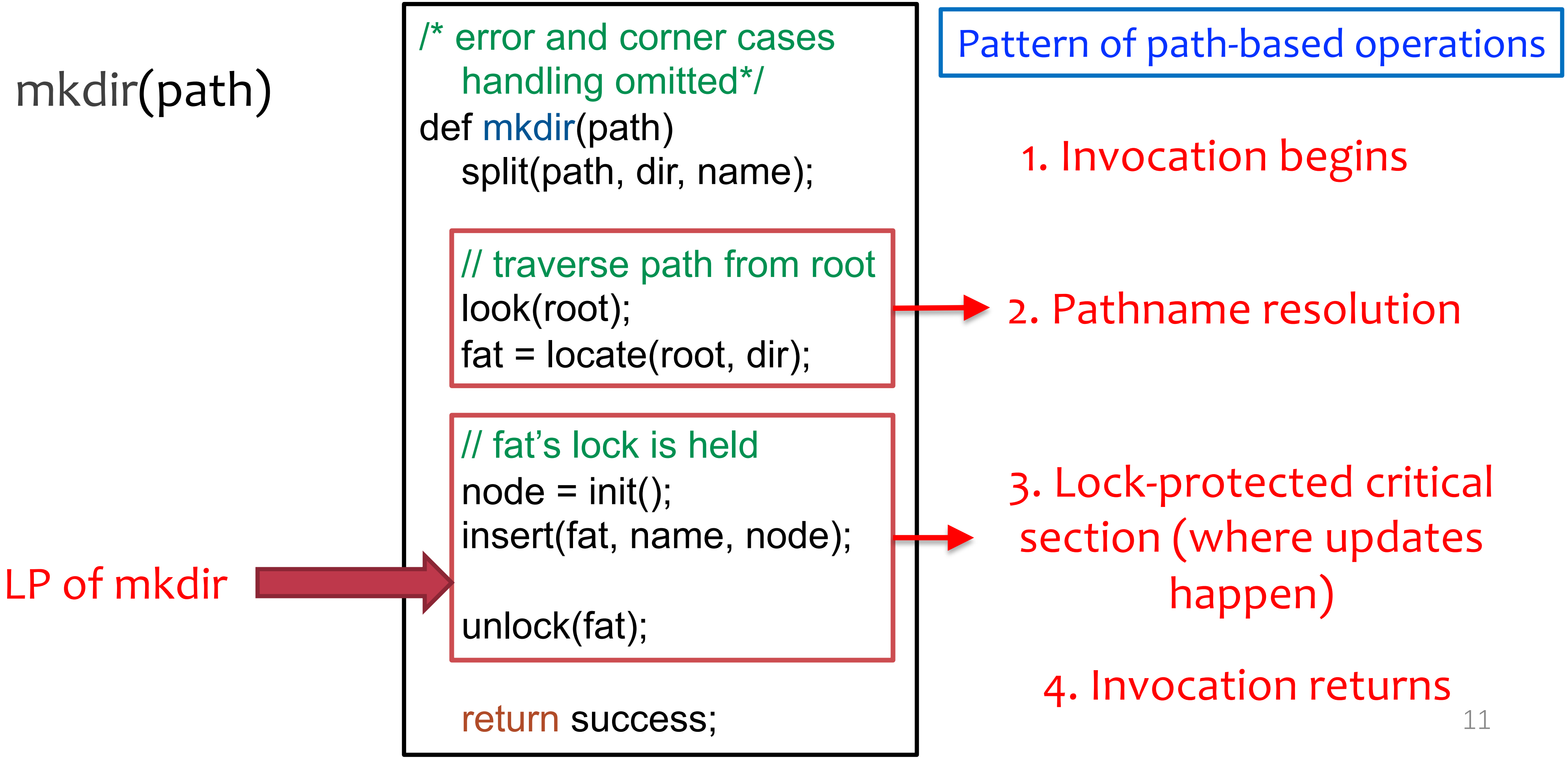
effect

Proven by
Liang et al, PLDI’13
Filipovic, ESOP’09

Linearizability

Construct linearization of abstract operation following temporal order of LPs

Strawman: fixed LP in critical section



mkdir(path)

```
/* error and corner cases
   handling omitted*/
def mkdir(path)
   split(path, dir, name);

   // traverse path from root
   look(root);
   fat = locate(root, dir);

   // fat's lock is held
   node = init();
   insert(fat, name, node);

   unlock(fat);

   return success;
```

LP of mkdir

Pattern of path-based operations

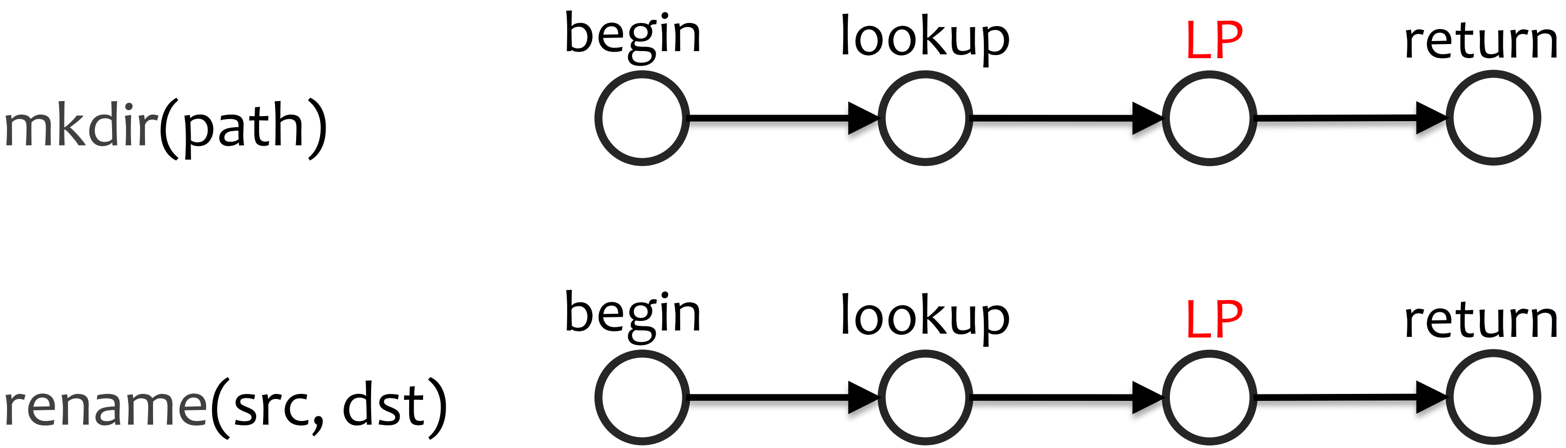
1. Invocation begins

2. Pathname resolution

3. Lock-protected critical section (where updates happen)

4. Invocation returns

Strawman: fixed LP in critical section



mkdir(path)

begin

lookup

LP

return

rename(src, dst)

begin

lookup

LP

return

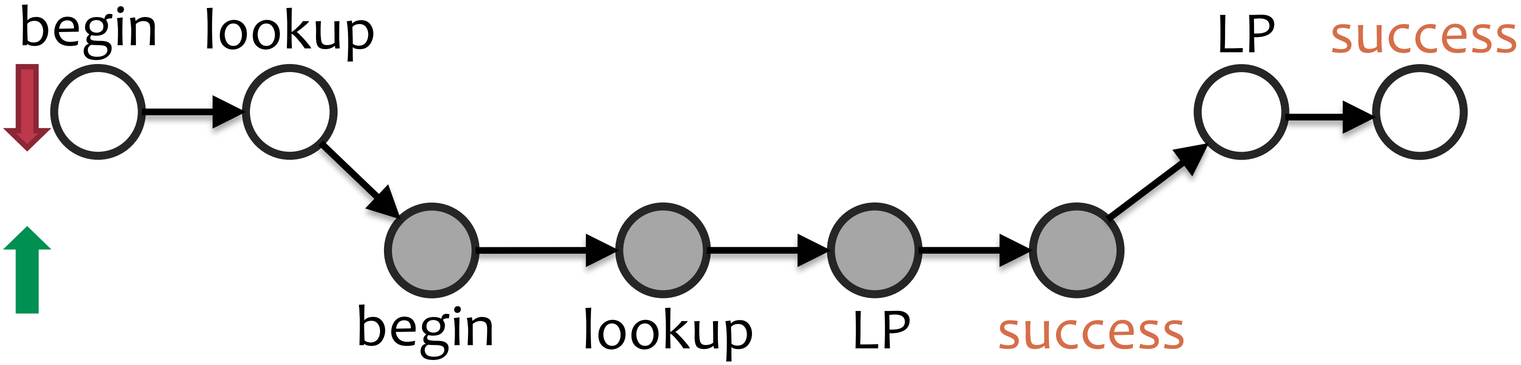
If fixed LP is correct and implementation is linearizable → We can construct linearization for any concurrent execution

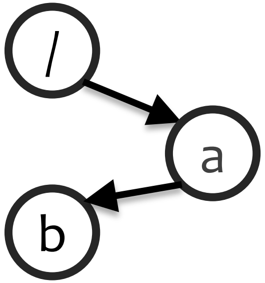
Challenge: fixed LP could fail in linearization

Two threads

○ **mkdir(/a/b/c)** ↓

● **rename(/a,/d)** ↑





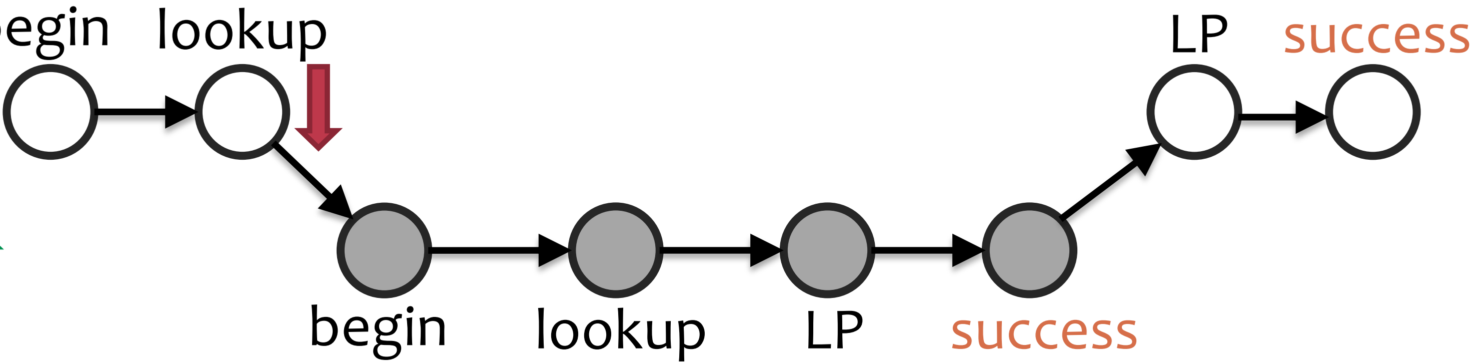
Initially

Challenge: fixed LP could fail in linearization

Two threads

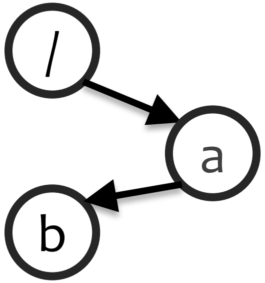
○ mkdir(/a/b/c)

● rename(/a,/d)⬆

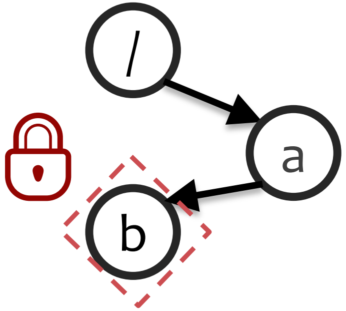


begin lookup LP success

begin lookup LP success



Initially



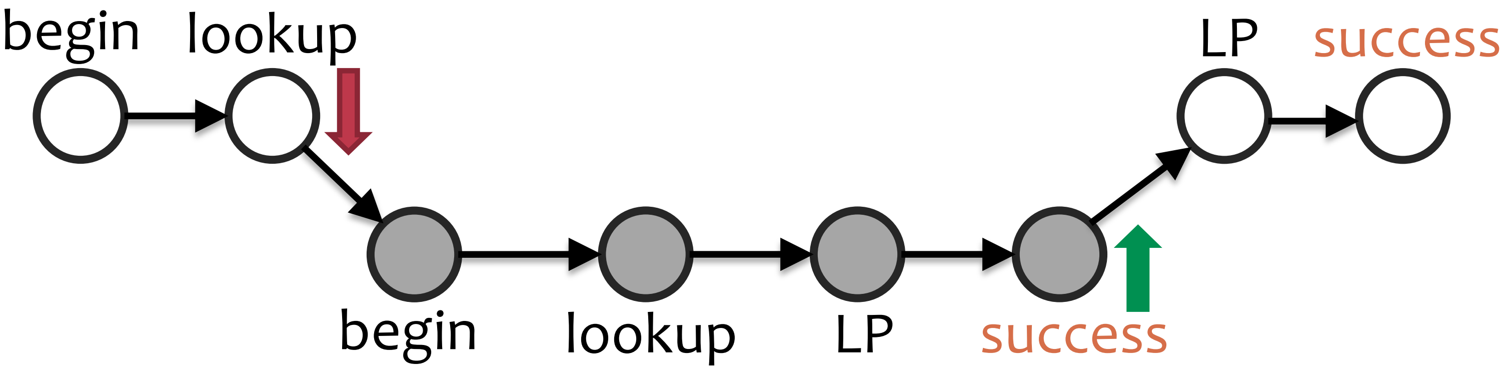
mkdir traverses
to b

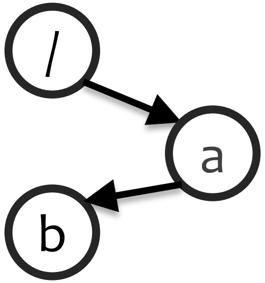
Challenge: fixed LP could fail in linearization

Two threads

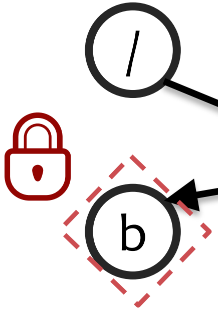
○ mkdir(/a/b/c)

● rename(/a,/d)

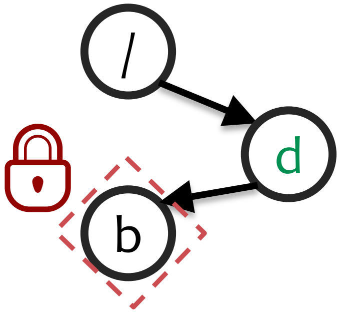




Initially



mkdir traverses to b



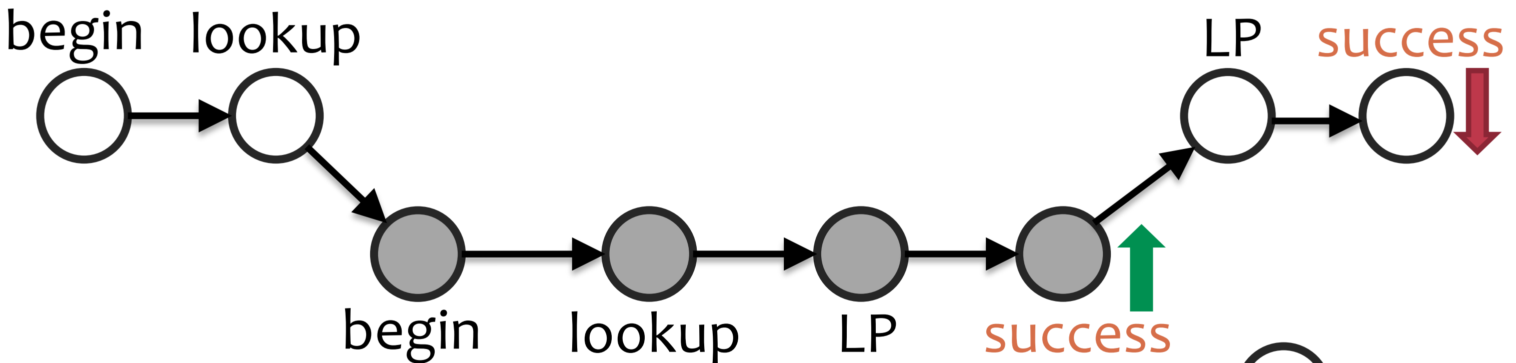
rename finishes

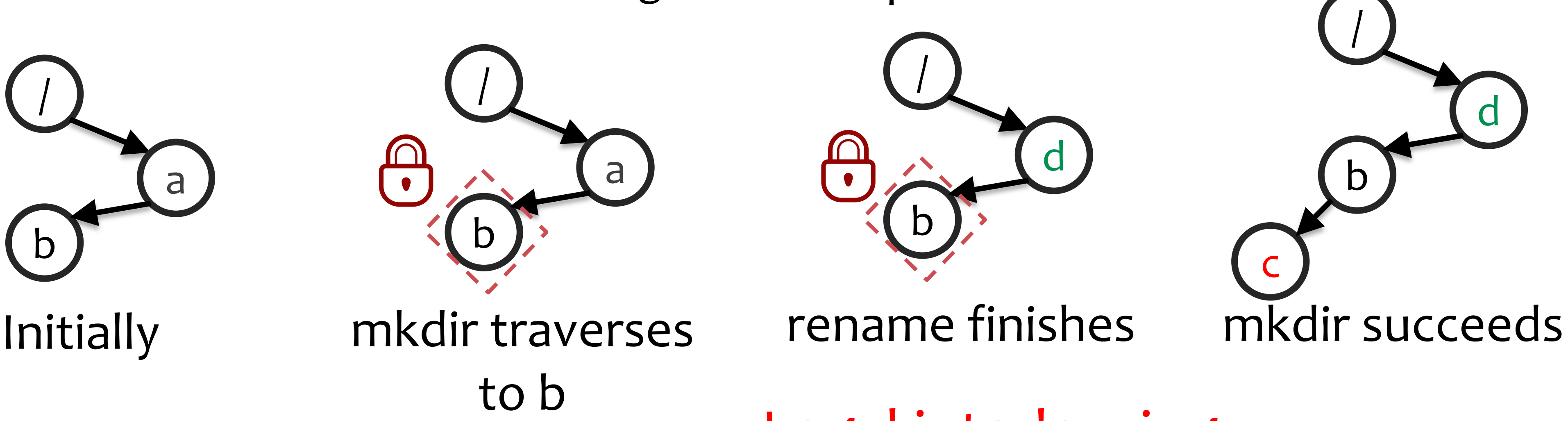
Challenge: fixed LP could fail in linearization

Two threads

○ **mkdir(/a/b/c)**

● **rename(/a,/d)**





Initially

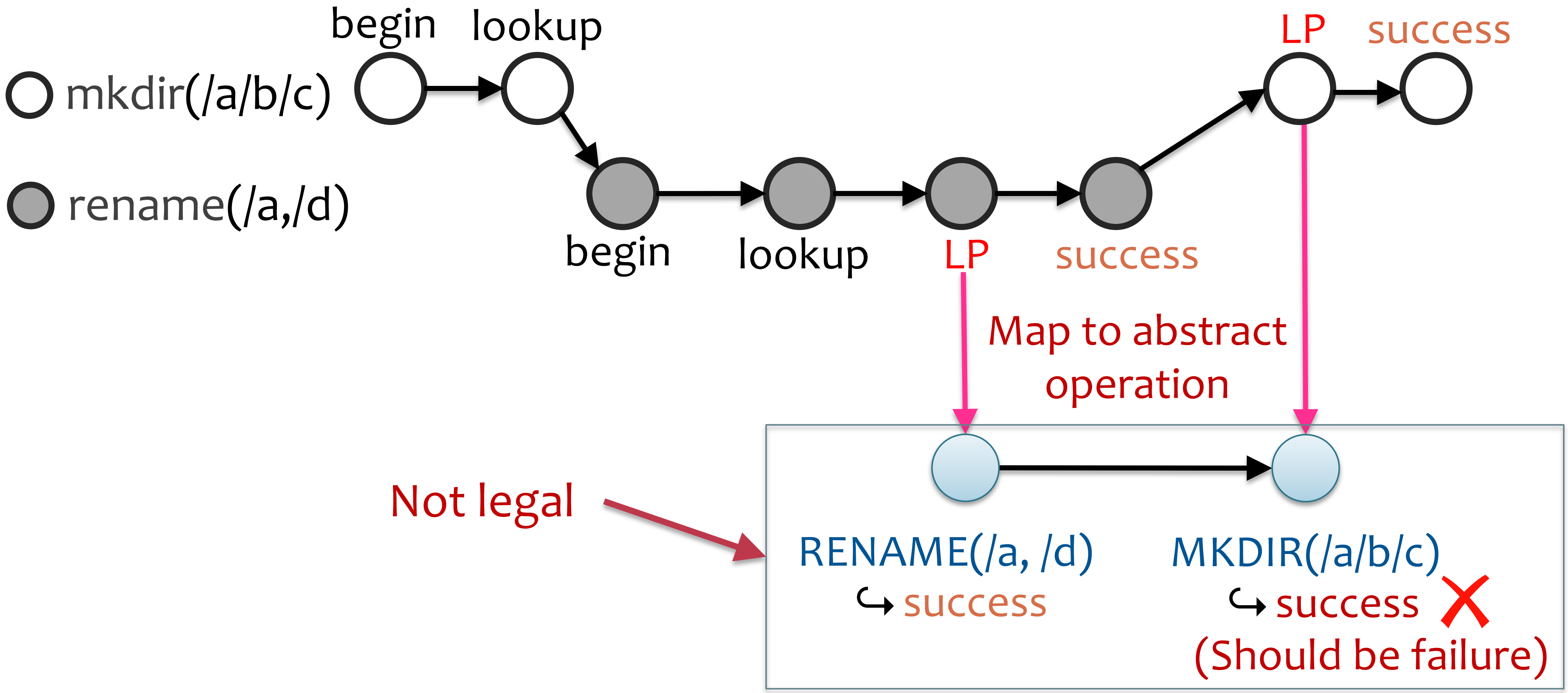
mkdir traverses to b

rename finishes

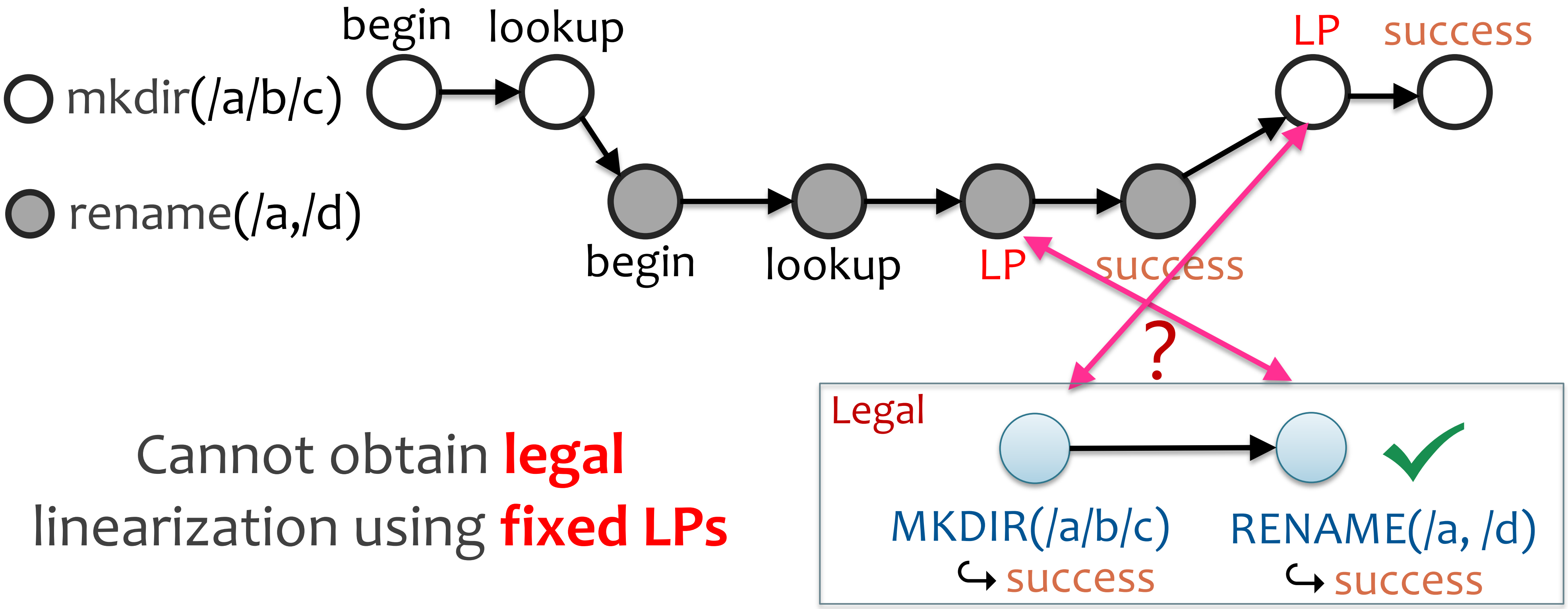
mkdir succeeds

Legal interleaving

Challenge: fixed LP could fail in linearization



Challenge: fixed LP could fail in linearization

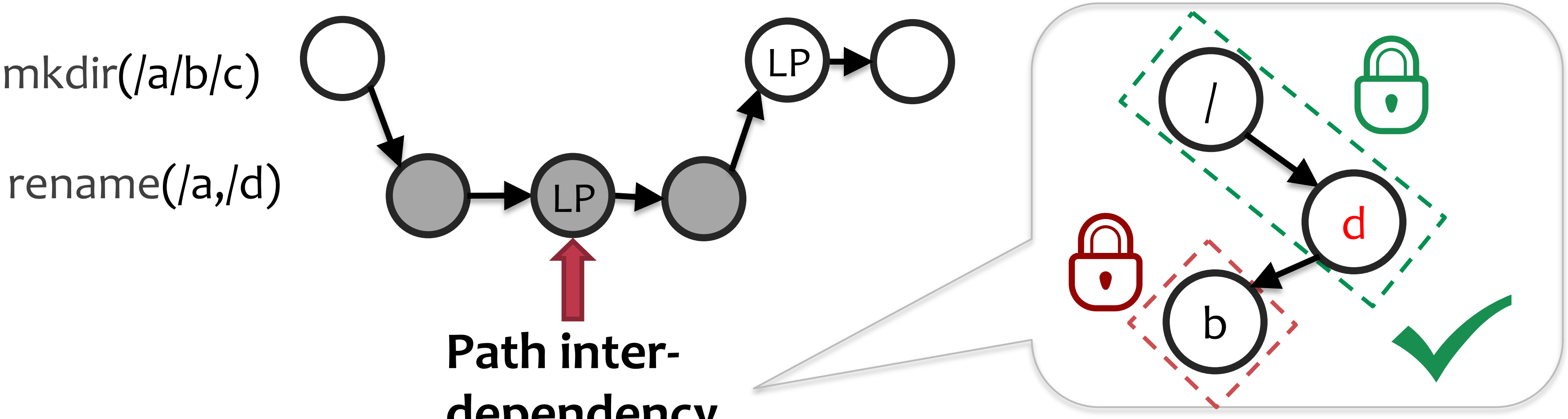


Cannot obtain **legal** linearization using **fixed LPs**

Check other cases ⇒ All failed cases involve **rename**

Observation: rename modifies other Op’s traversed path

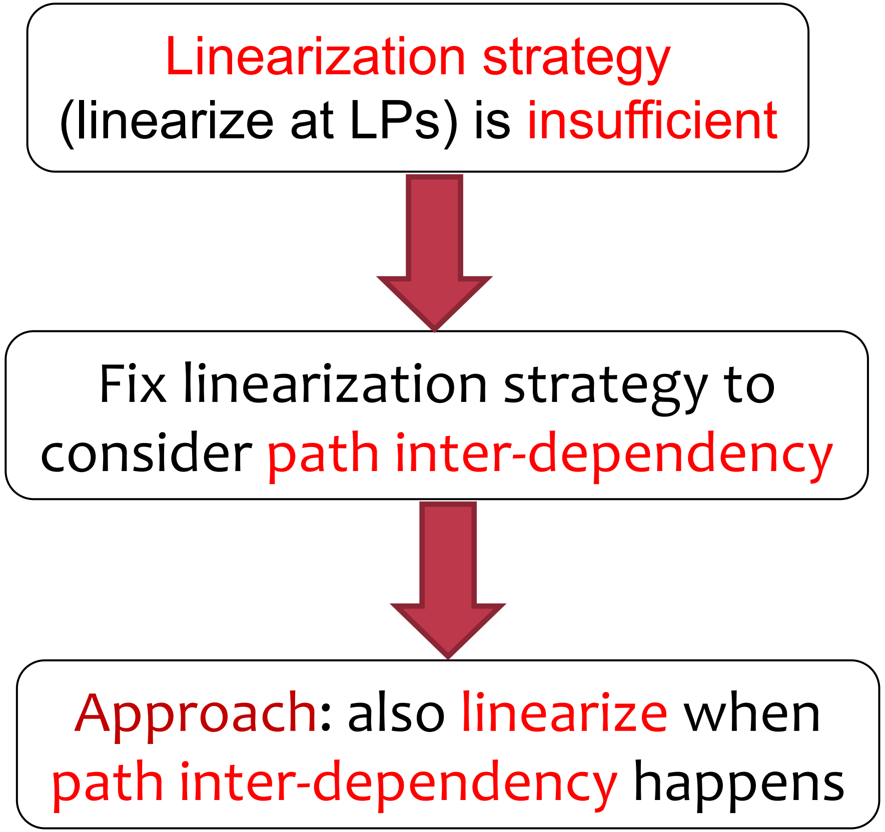
- We call this phenomenon **path inter-dependency**
  - Rename, only operation that can modify an internal inode



Path inter-dependency

Allowed by fine-grained implementation

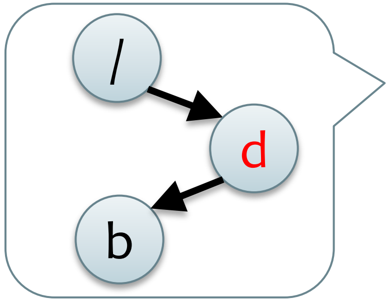
Should consider path inter-dependency in linearization



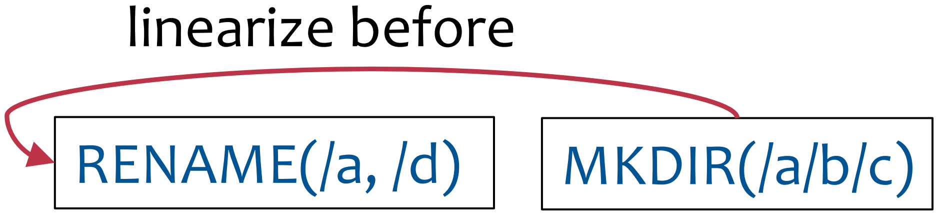
Linearization strategy (linearize at LPs) is insufficient

Fix linearization strategy to consider path inter-dependency

Approach: also linearize when path inter-dependency happens



RENAME(/a,/d) break MKDIR(/a/b/c)’s path integrity

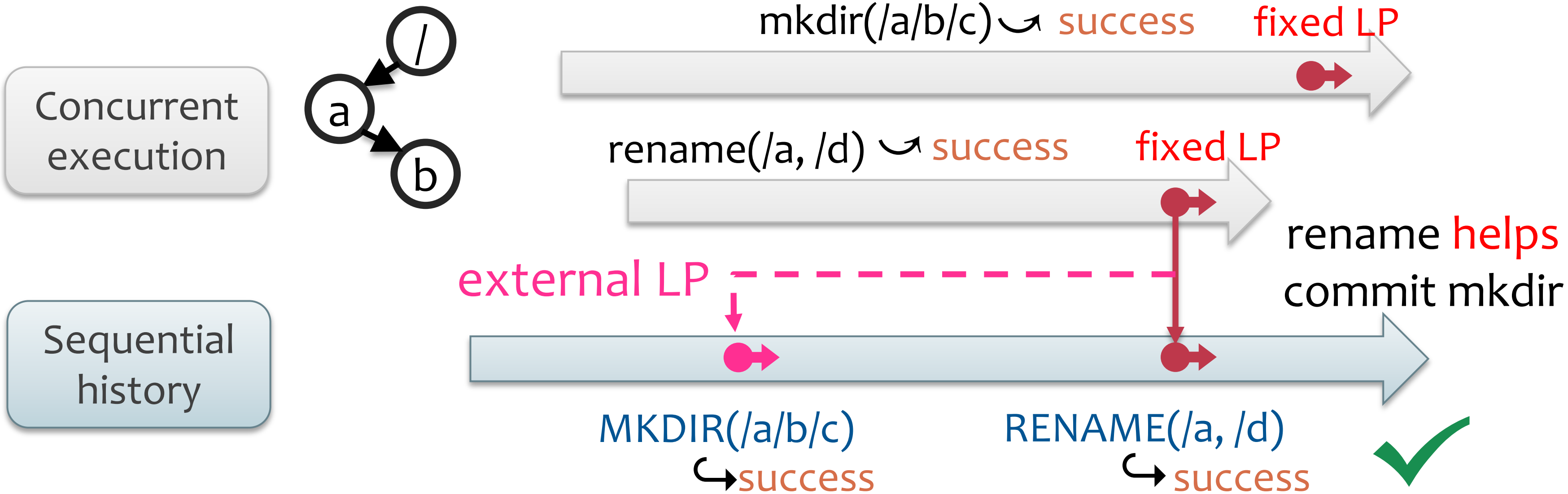


linearize before

RENAME(/a, /d)

MKDIR(/a/b/c)

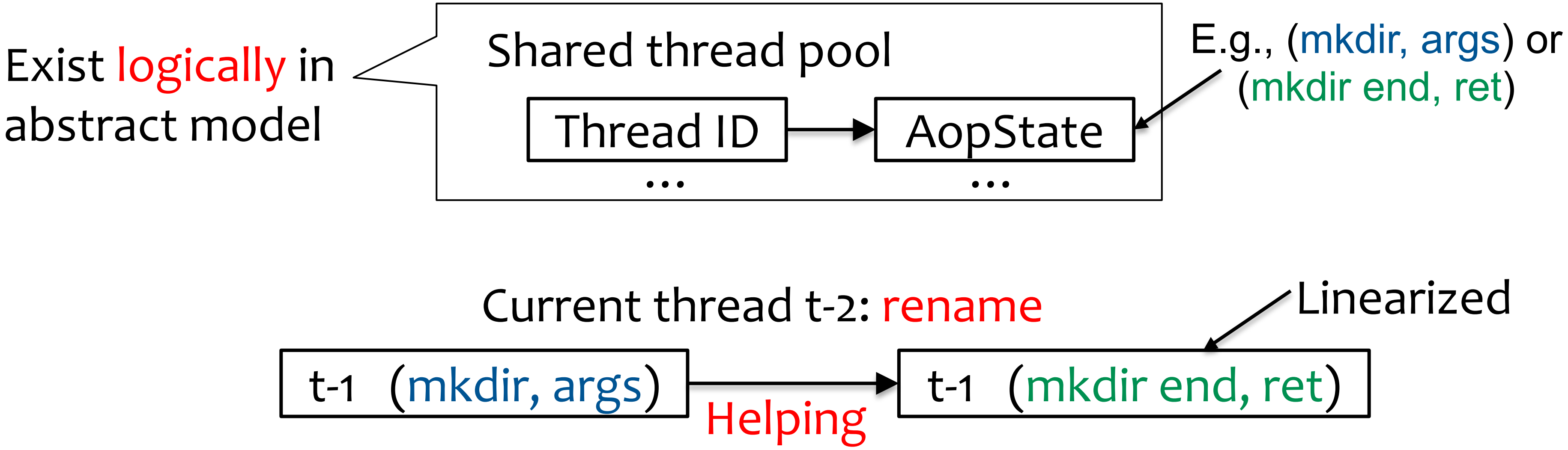
Approach: linearize when path inter-dependency happens



- The LP of Op_1 (e.g., mkdir) resides in another Op_2 (e.g., rename)
  - This kind of LP is called external linearization point
- For path-based Op, LP could be internal (“fixed LP”) or external (triggered by rename)

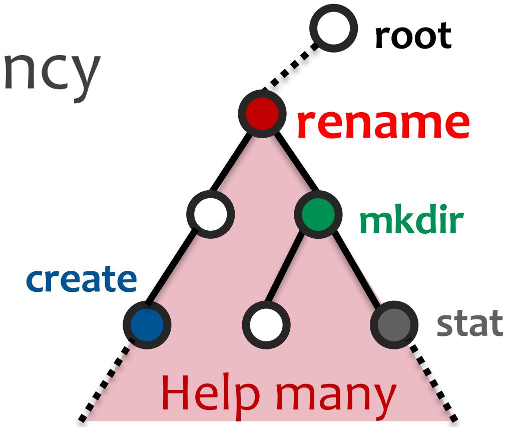
Helping: linearize abstract operations of other threads

- Helping: linearize abstract operations of other threads [Liang et al, PLDI'13]


Exist logically in abstract model
Shared thread pool
Thread ID
AopState
…
…
E.g., (mkdir, args) or (mkdir end, ret)
Current thread t-2: rename
Linearized
t-1 (mkdir, args)
Helping
t-1 (mkdir end, ret)

File system-specific challenges

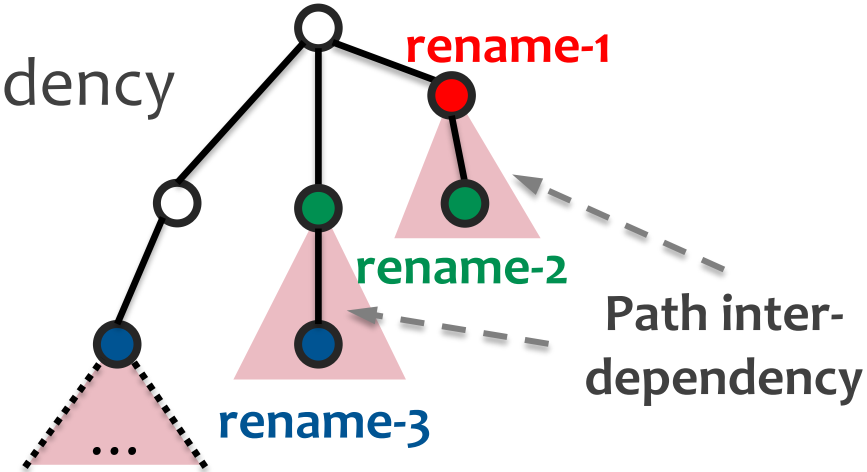
- Which threads to help (for a rename)?
- Helping order?
- Handle recursive path inter-dependency



Decide helping set and order

File system-specific challenges

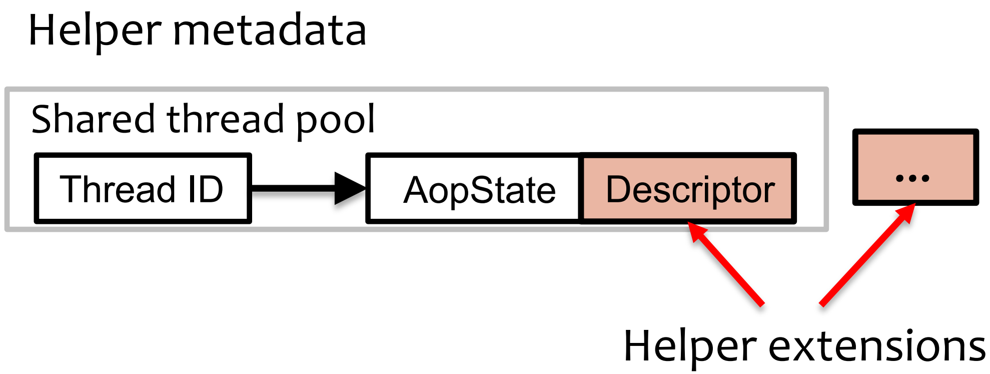
- Which threads to help (for a rename)?
- Helping order?
- Handle recursive path inter-dependency

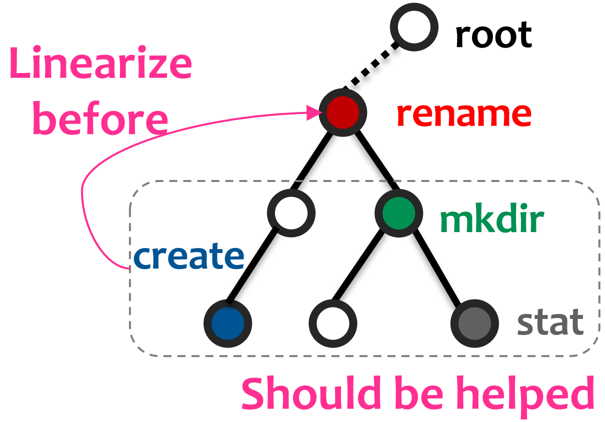


Recursive path inter-dependency

Helpers: extend helping with file system-specific notions

- **Helper metadata** provides global information
  - E.g., add “lock path” in Descriptor to record traversed path
- Decide whether Op$_1$ should be **linearized before** Op$_2$
  - E.g., rename can use “lock path” to decide which threads to help



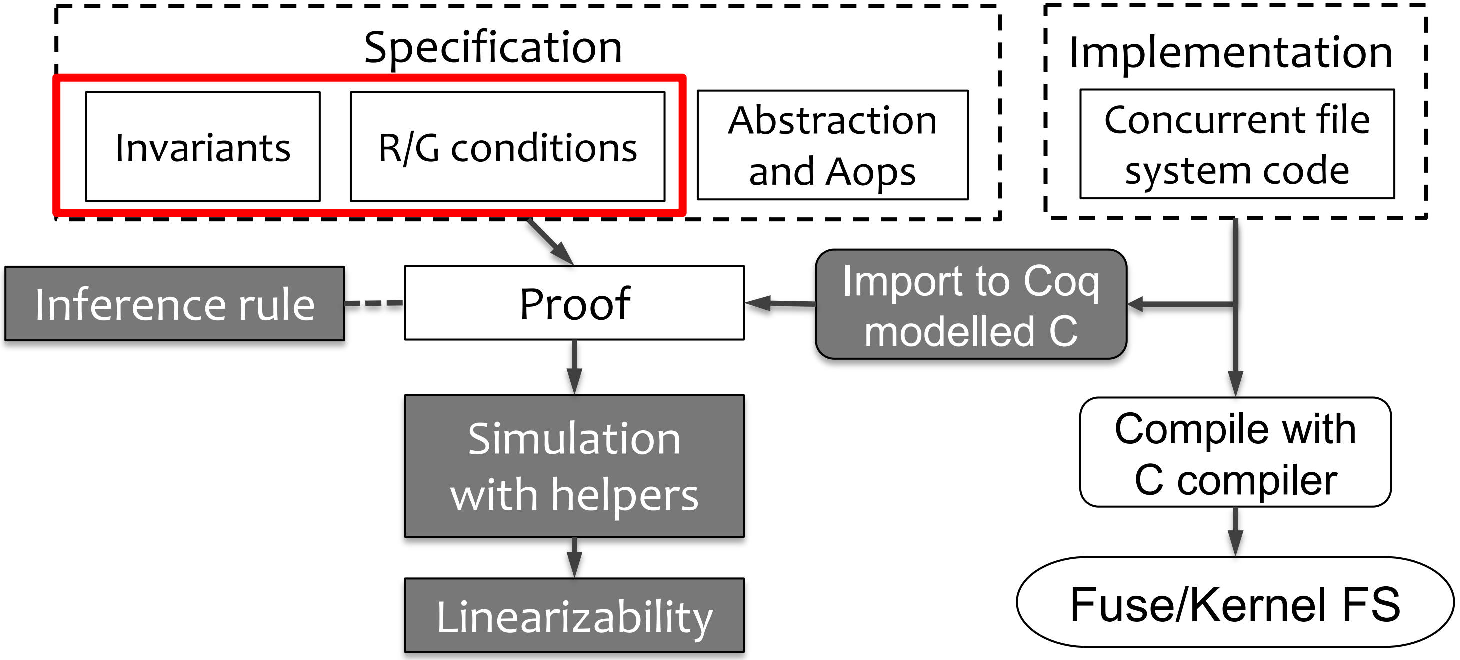


Specifying and proving with CRL-H

Read paper for details

CRL-H framework: **C**oncurrent **R**elation **L**ogic with **H**elpers

$$\text{Rely; Guarantee; Invariant} \vdash \{\text{Pre} * (\text{aop}, \text{args})\}\ \text{Code}\ \{\text{Post} * (\text{aop end}, \text{ret})\}$$



Local rely guarantee for fine-grained **concurrency**

Specifying and proving with CRL-H

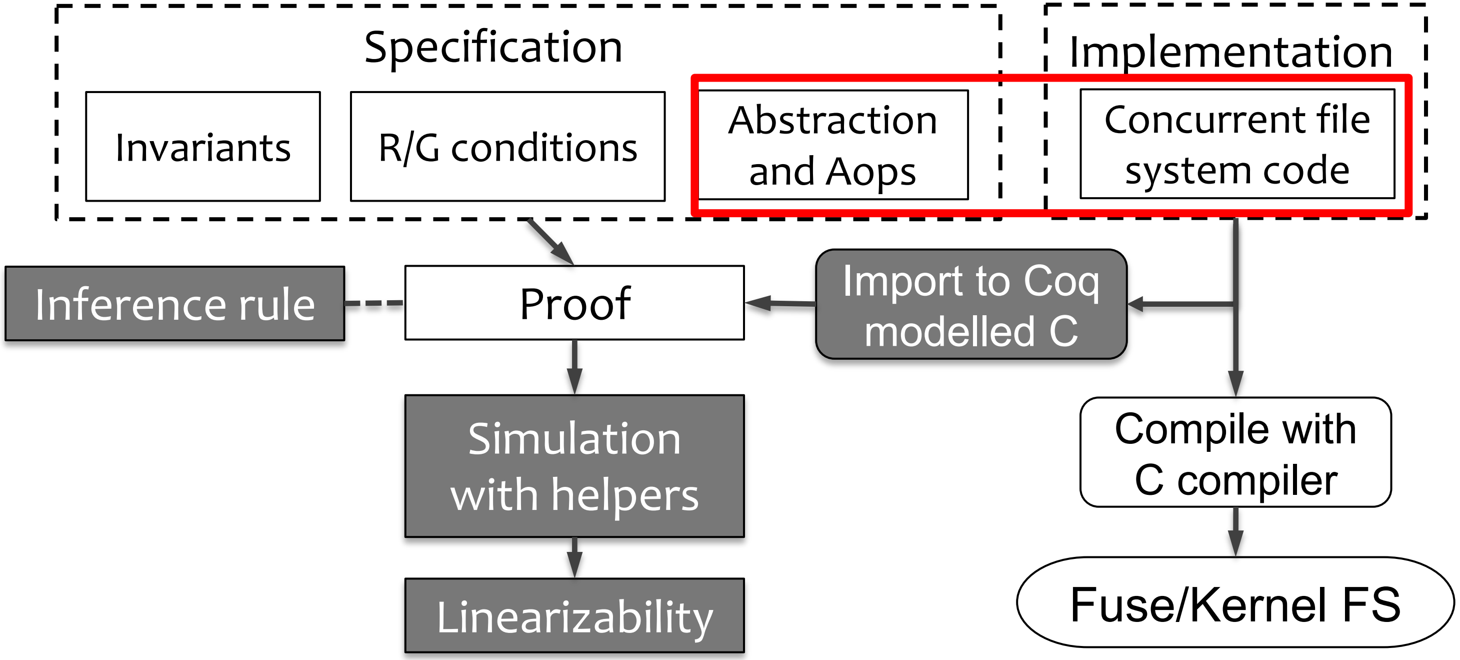
Read paper for details

CRL-H framework: **C**oncurrent **R**elation **L**ogic with **H**elpers

$\text{Rely; Guarantee; Invariant} \vdash \{\text{Pre} * (\text{aop}, \text{args})\}\ \text{Code}\ \{\text{Post} * (\text{aop end}, \text{ret})\}$

RENAME

rename(args)



Local rely guarantee for fine-grained **concurrency**

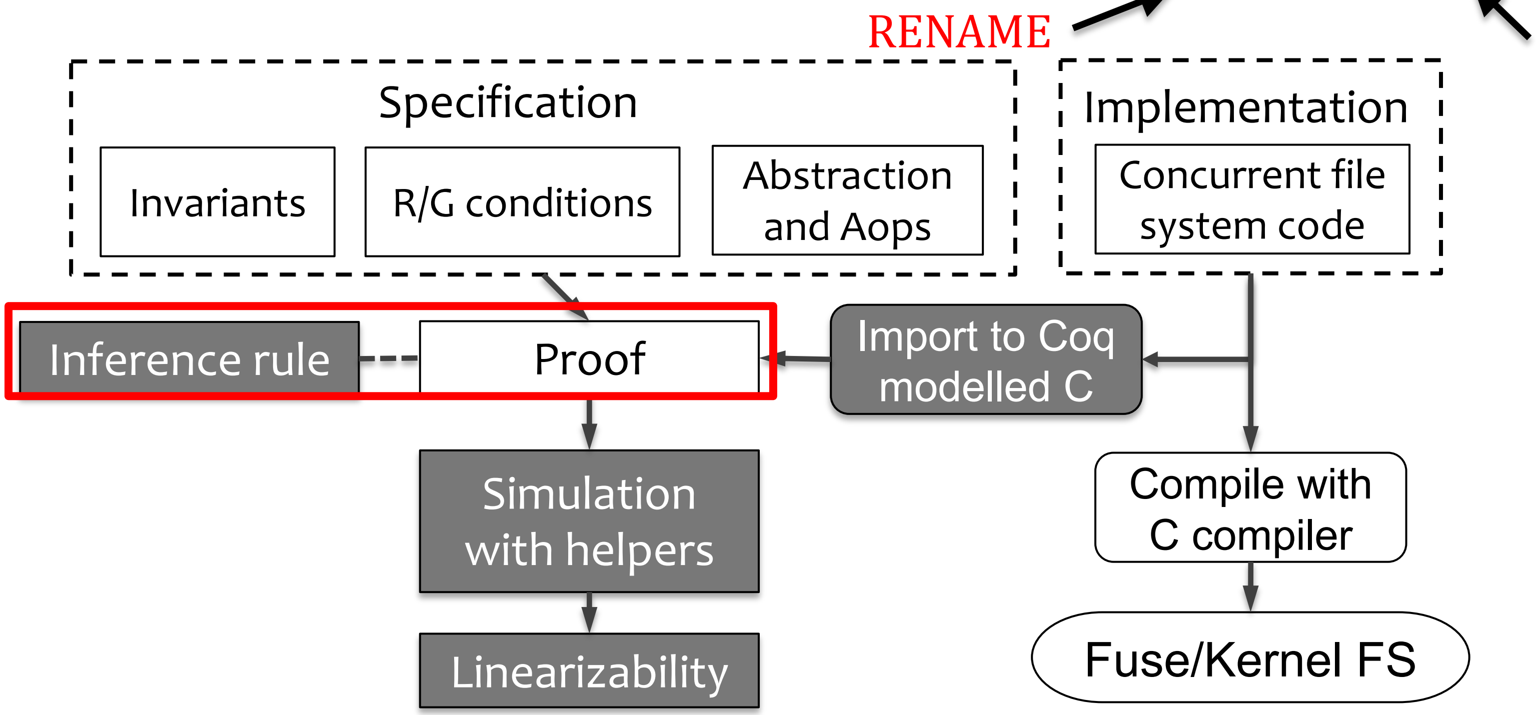
Relational reasoning

Specifying and proving with CRL-H

Read paper for details

CRL-H framework: **C**oncurrent **R**elation **L**ogic with **H**elpers

$\text{Rely; Guarantee; Invariant} \vdash \{\text{Pre} * (\text{aop}, \text{args})\}\ \text{Code}\ \{\text{Post} * (\text{aop end}, \text{ret})\}$



Local rely guarantee for fine-grained **concurrency**

Relational reasoning

Inference rule

Specifying and proving with CRL-H

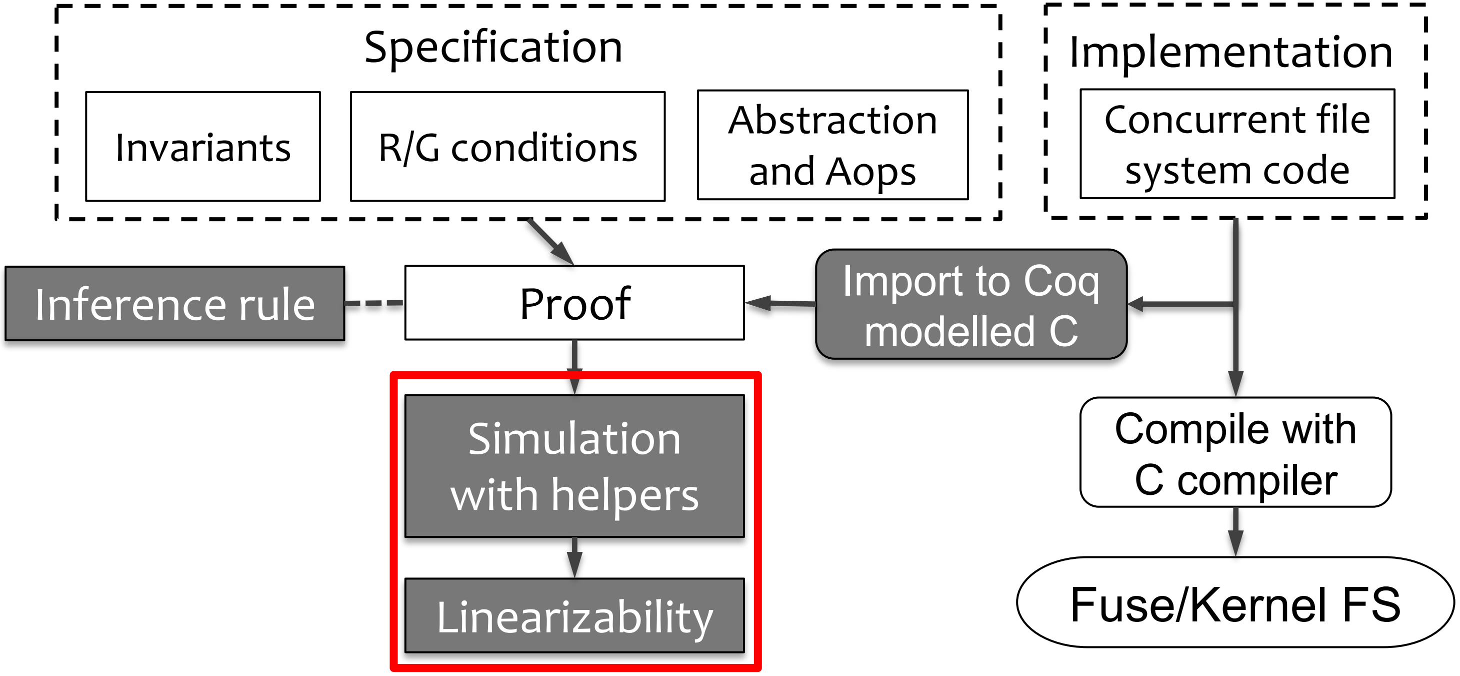
Read paper for details

CRL-H framework: **C**oncurrent **R**elation **L**ogic with **H**elpers

$$\text{Rely; Guarantee; Invariant} \ \vdash \ \{\text{Pre} * (\text{aop}, \text{args})\}\ \text{Code}\ \{\text{Post} * (\text{aop end}, \text{ret})\}$$

RENAME

rename(args)



Specification

Invariants

R/G conditions

Abstraction and Aops

Implementation

Concurrent file system code

Inference rule

Proof

Import to Coq modelled C

Simulation with helpers

Linearizability

Compile with C compiler

Fuse/Kernel FS

Local rely guarantee for fine-grained **concurrency**

Relational reasoning

Inference rule

Correctness theorem

Specifying and proving with CRL-H

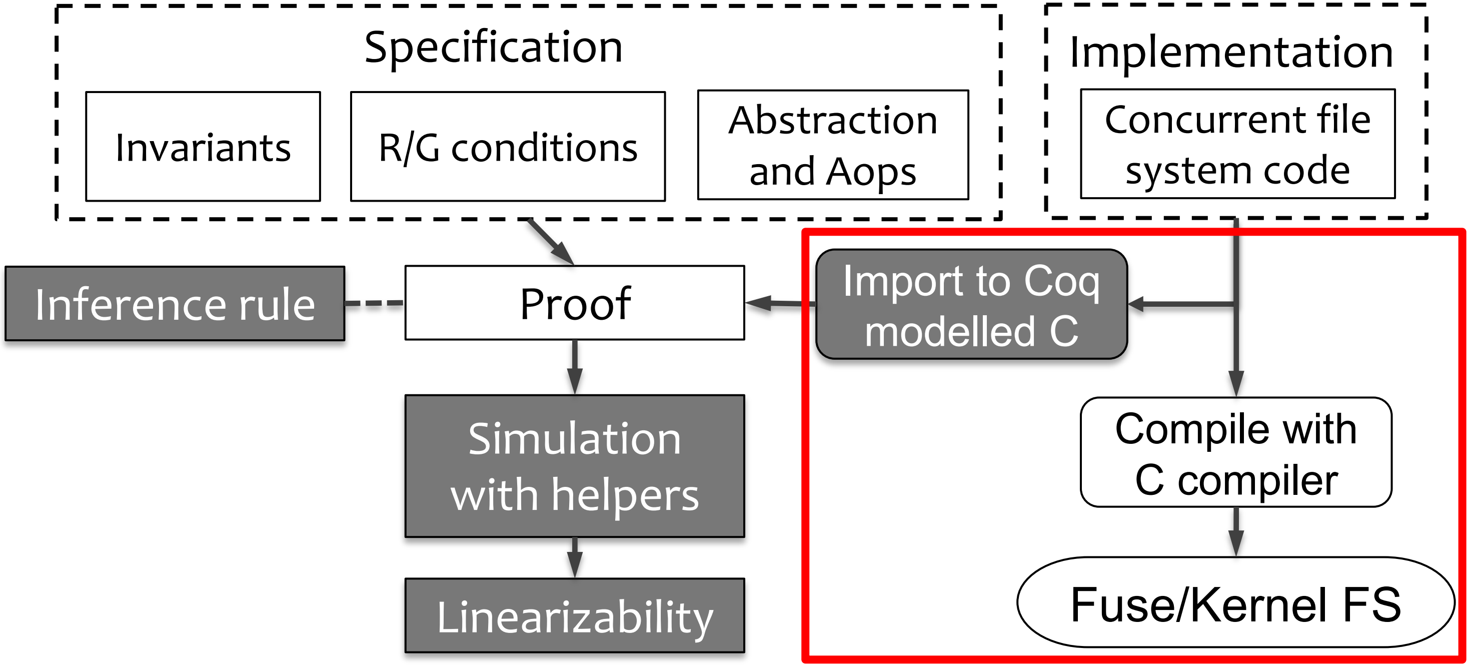
Read paper for details

CRL-H framework: **C**oncurrent **R**elation **L**ogic with **H**elpers

$$\text{Rely; Guarantee; Invariant} \vdash \{\text{Pre} * (\text{aop}, \text{args})\}\ \text{Code}\ \{\text{Post} * (\text{aop end}, \text{ret})\}$$

RENAME

rename(args)



Local rely guarantee for fine-grained **concurrency**

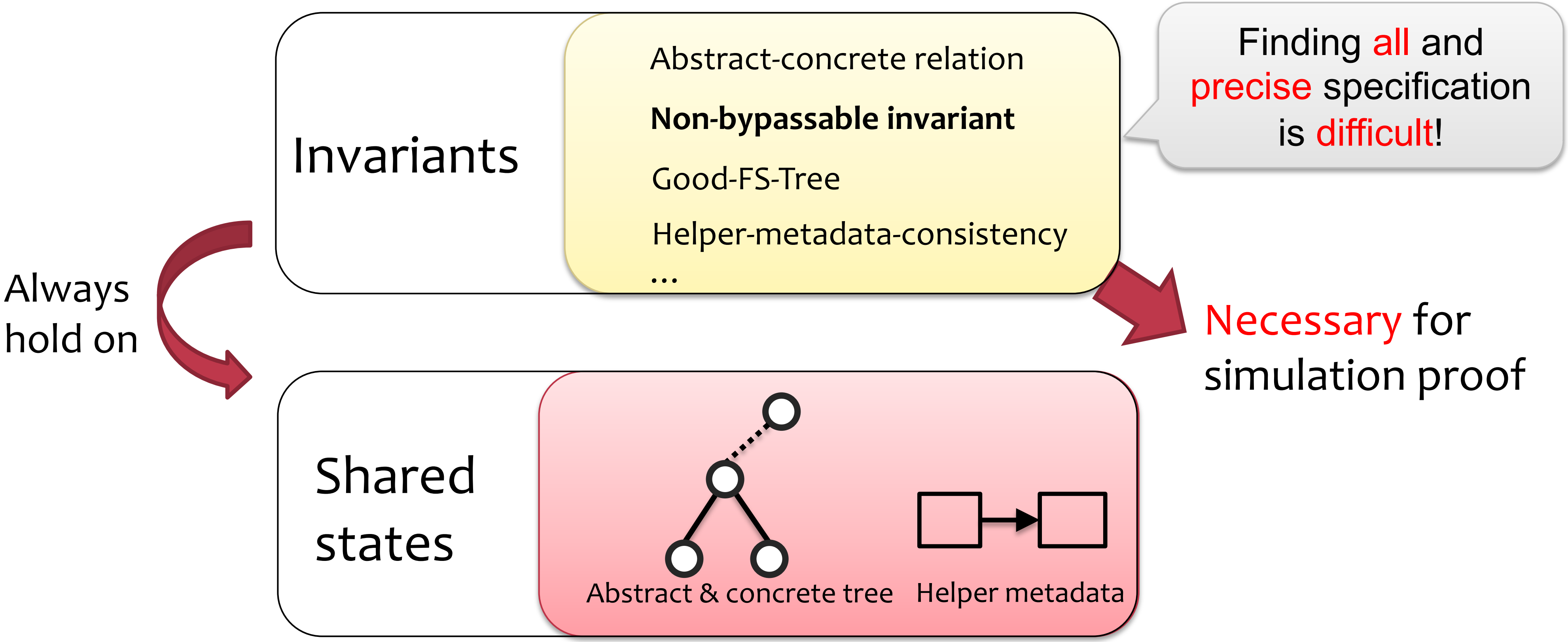
Relational reasoning

Inference rule

Correctness theorem

C language modeling

Invariants in proving AtomFS



Invariants

Abstract-concrete relation

Non-bypassable invariant

Good-FS-Tree

Helper-metadata-consistency

…

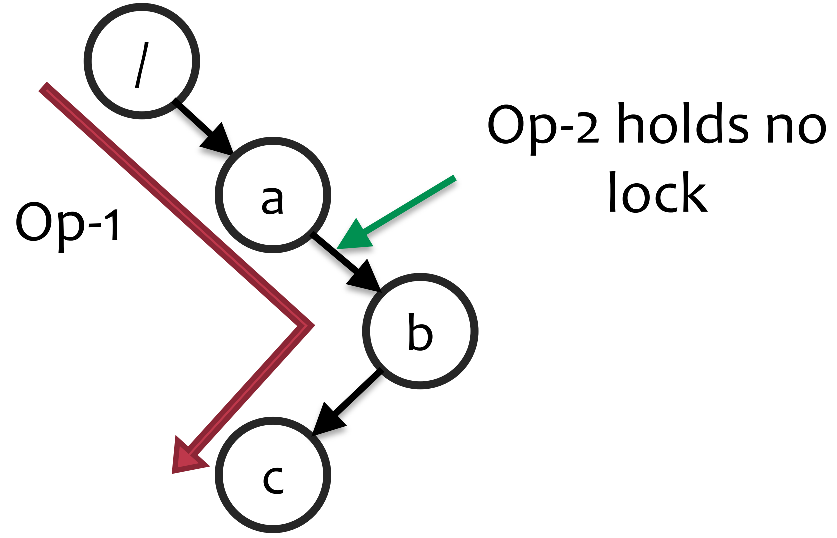
Finding all and precise specification is difficult!

Always hold on

Necessary for simulation proof

Shared states

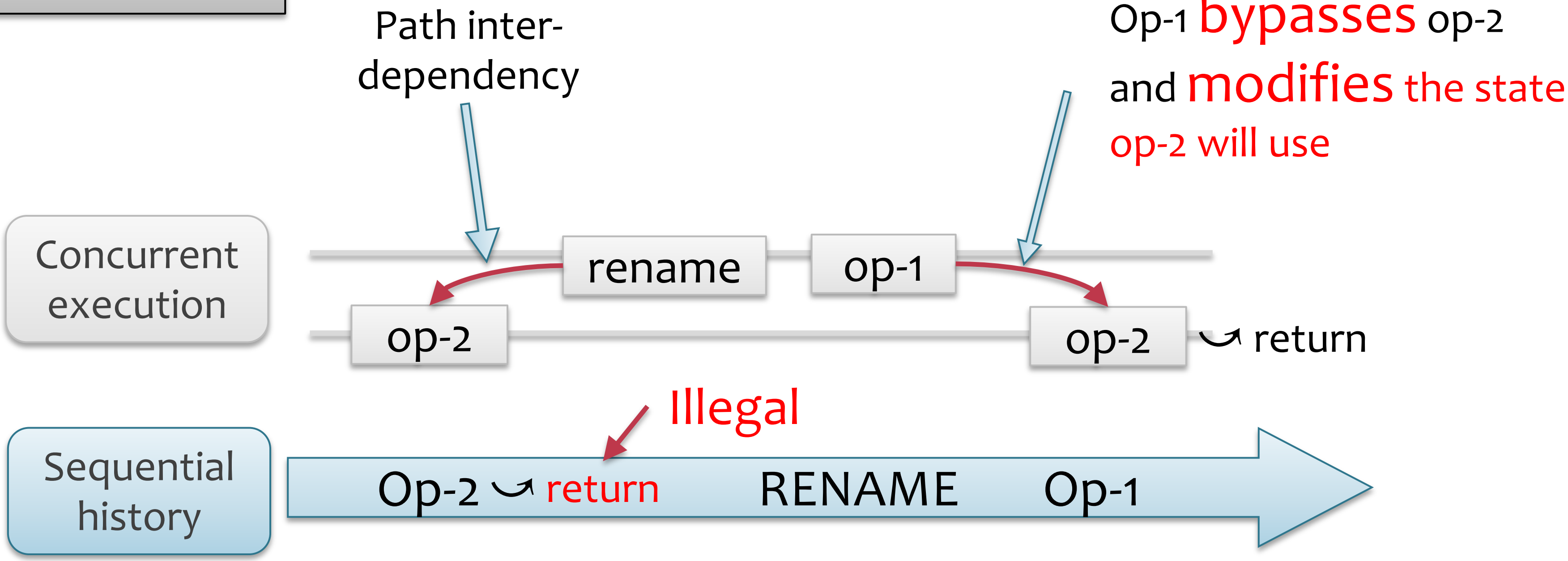
Operation bypassing leads to non-linearizability



Op-1 bypasses Op-2

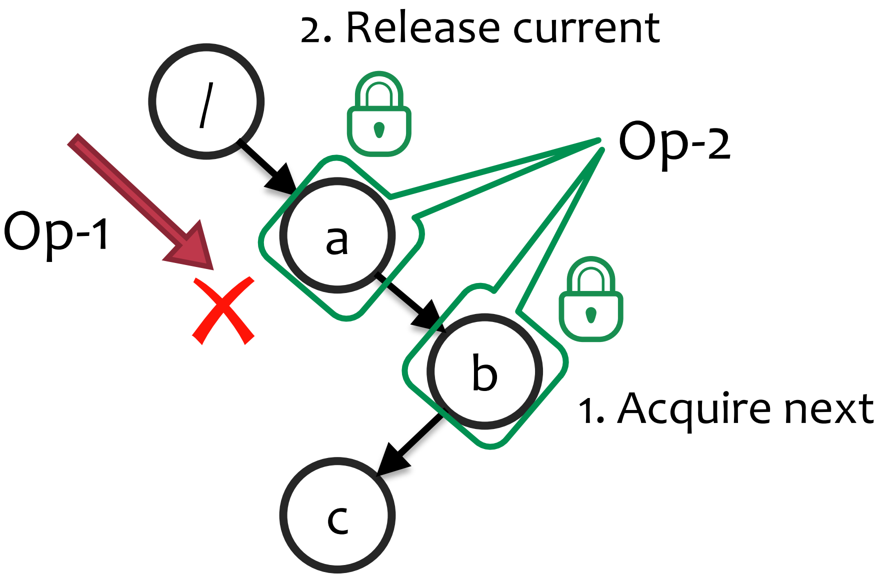
Operation bypassing leads to non-linearizability

Read paper for a concrete case



Construct a non-linearizable interleaving

Lock coupling forbids operation bypassing



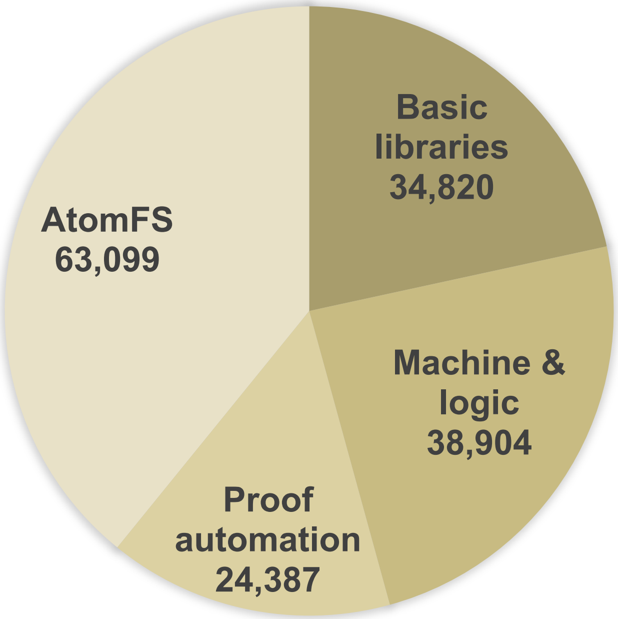
Forbid bypassing by always holding a lock

- Non-bypassable invariant to capture the property
- Cons: reduce parallelism
- Pros: ensure linearizability
  - Easier to reason about for users

Tradeoff between performance and reasoning!

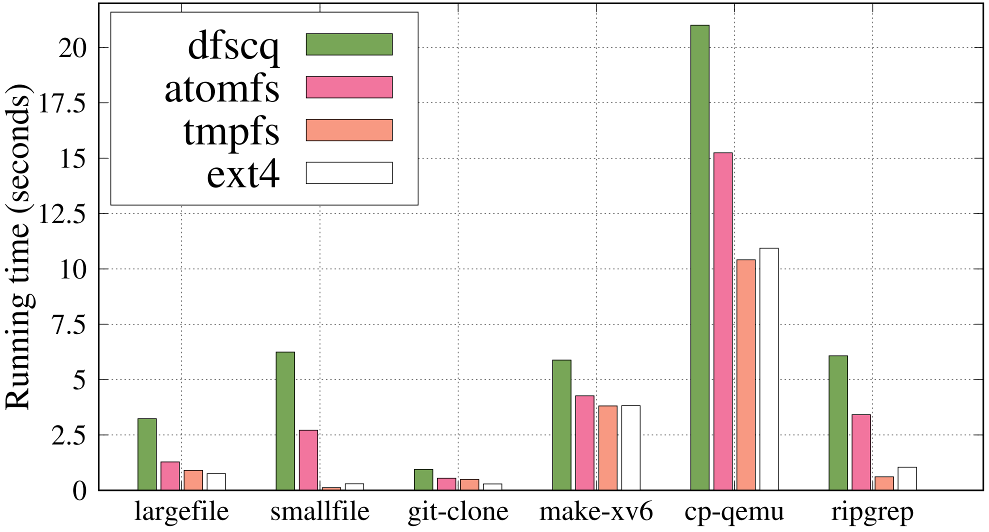
Implementing CRL-H and AtomFS in Coq

- 1.5 years of effort, including building the framework and proving AtomFS
- CRL-H, ~100k LOC
  - Most can be reused
- AtomFS
  - 673 lines of C code
  - 2k lines of specification
  - 60k lines of proof



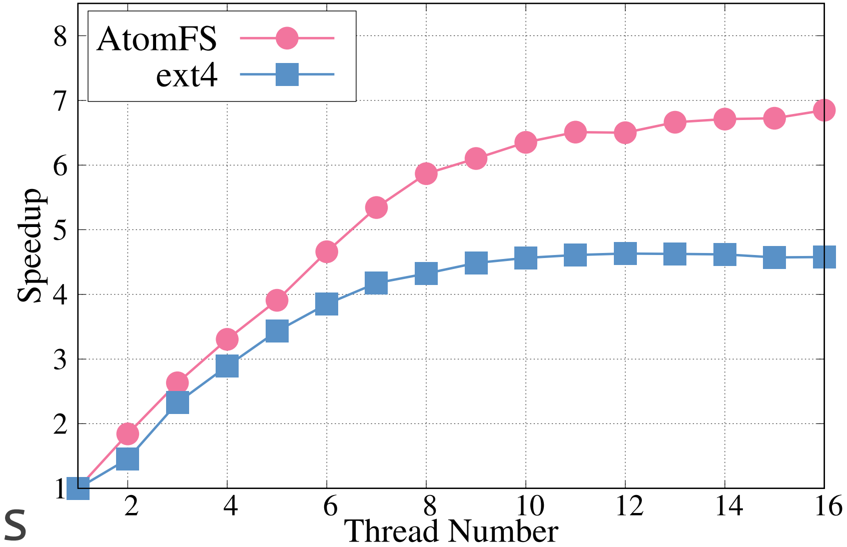
Evaluation: AtomFS achieves reasonable performance

- Single core performance
- **Faster than DFSCQ** (1.38x-2.52x)
  - Avoid Haskell overhead
- **Slower than ext4 and tmpfs**
  - FUSE overhead
  - Simplified data strucutre



Evaluation: AtomFS achieves reasonable performance

- Multicore scalability
- **Better scalability than ext4**
  - Not bypass VFS-level path lookup
  - Bottleneck: lock coupling traverse
- **Worse performance than ext4**
  - 6.39x lower throughput with 16cores
  - Not implement optimizations



Speedup on Fileserver

(compared to single core)

Conclusion

- CRL-H: specify and prove concurrent file systems
  - Path inter-dependency and external LP challenge
  - Helper mechanism
- AtomFS: first verified concurrent FS with fine-grained locking
  - Atomic interfaces
  - Reasonable performance

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

Thanks!

IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

SHANGHAI JIAO TONG UNIVERSITY 1896

HUAWEI

IN LUMINE TUO VIDEBIMUS LUMEN