Combining automated and interactive proofs to verify the **DaisyNFS** concurrent and crash-safe NFS server

Tej Chajed
MIT CSAIL

Joseph Tassarotti
Boston College

Mark Theng
MIT CSAIL

Ralf Jung
MPI-SWS

Frans Kaashoek
MIT CSAIL

Nickolai Zeldovich
MIT CSAIL

File systems are essential to many applications

Almost every application stores its data in a file system

Bugs in the file system can permanently lose data

Performance of the file system affects many programs

Suppose we want to write a correct file system

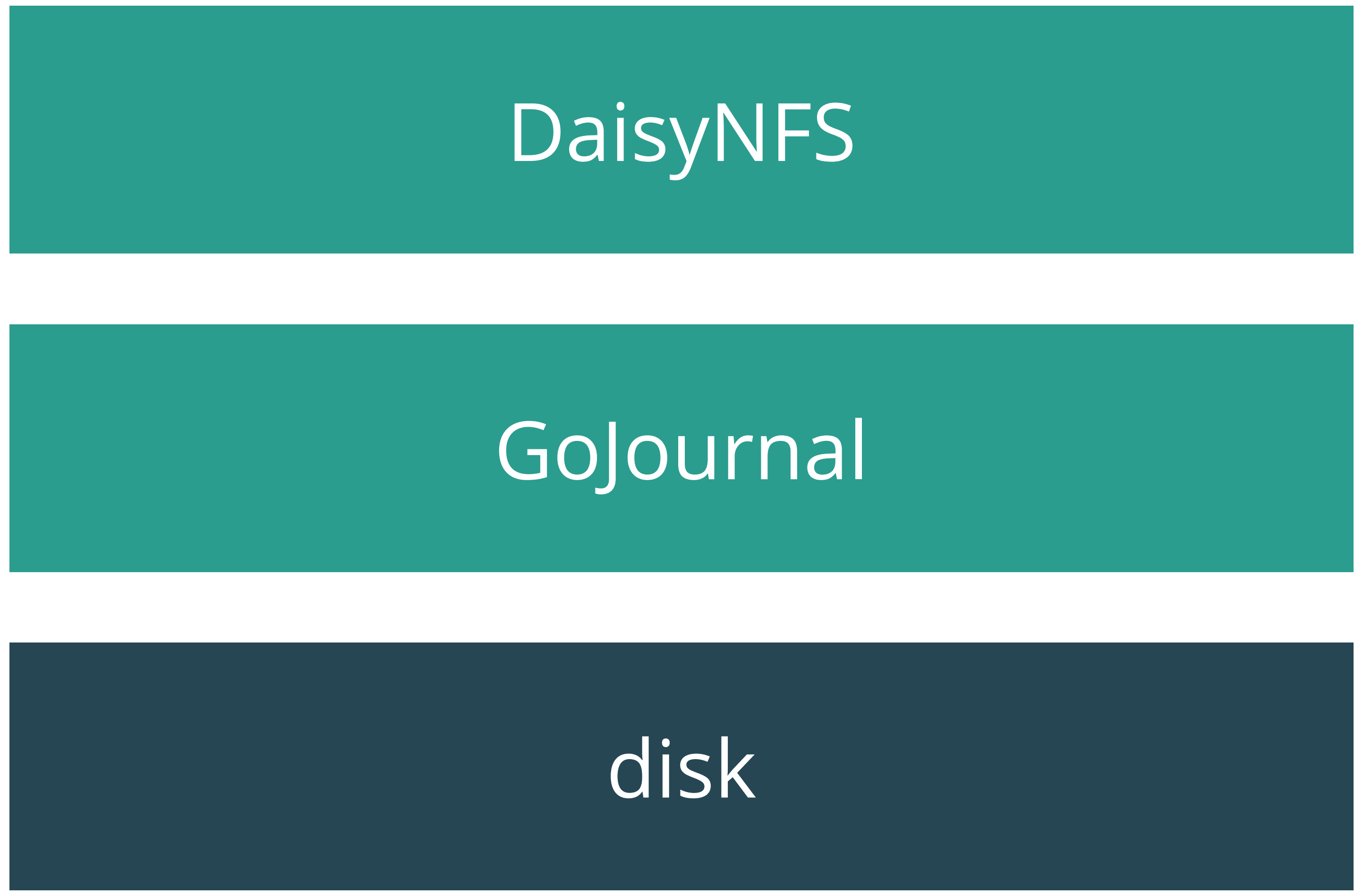
Correct: file-system operations atomically follow specification, even on crash

Performant: take advantage of concurrent operations to efficiently use CPU and I/O

GoJournal is a verified transaction system

atomic, crash-safe transactions
comes with a machine-checked **proof**

GoJournal is a verified transaction system



performant NFS server

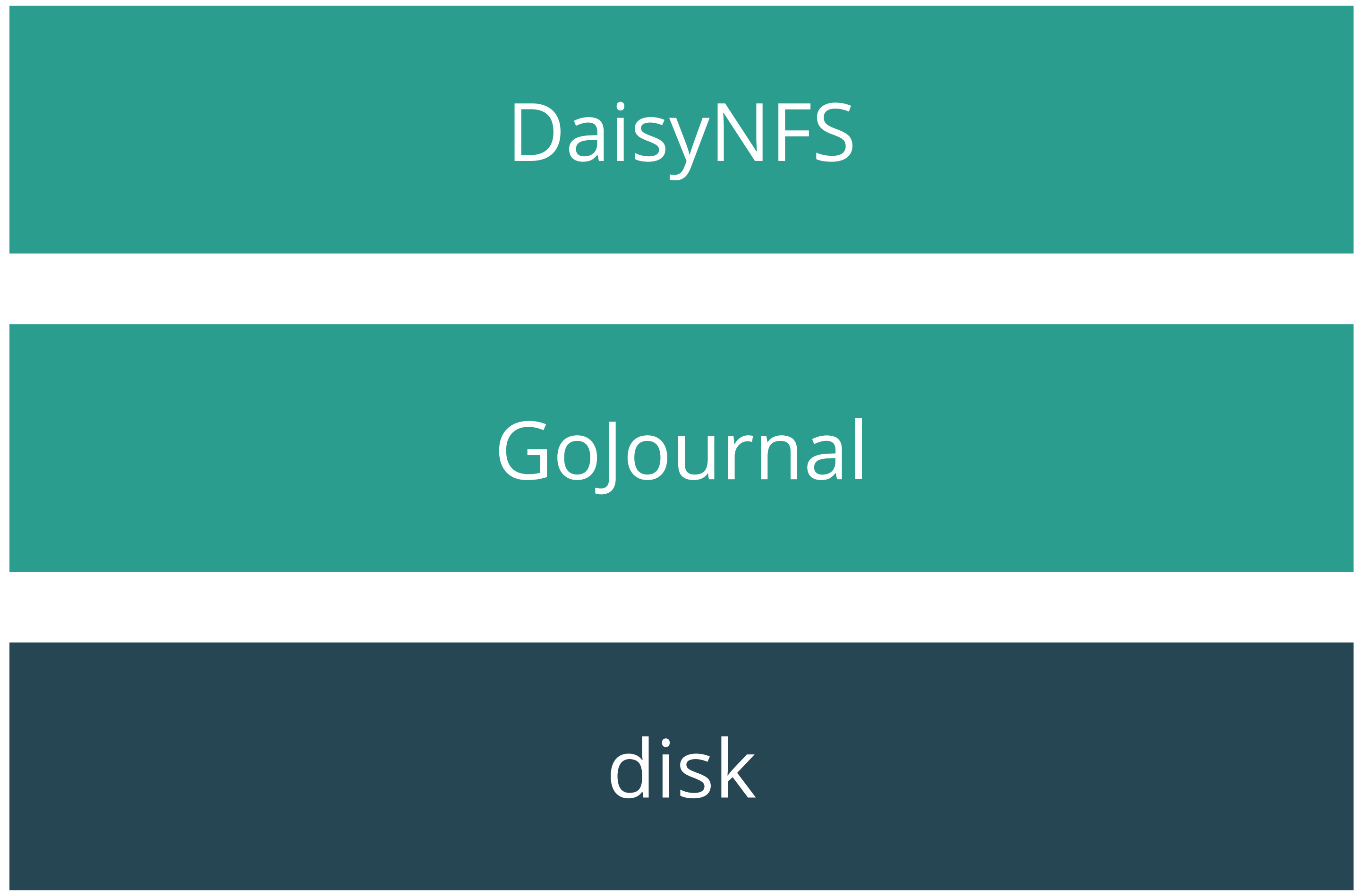
atomic, crash-safe transactions
comes with a machine-checked **proof**

GoJournal is a verified transaction system

performant NFS server

atomic, crash-safe transactions
comes with a machine-checked **proof**

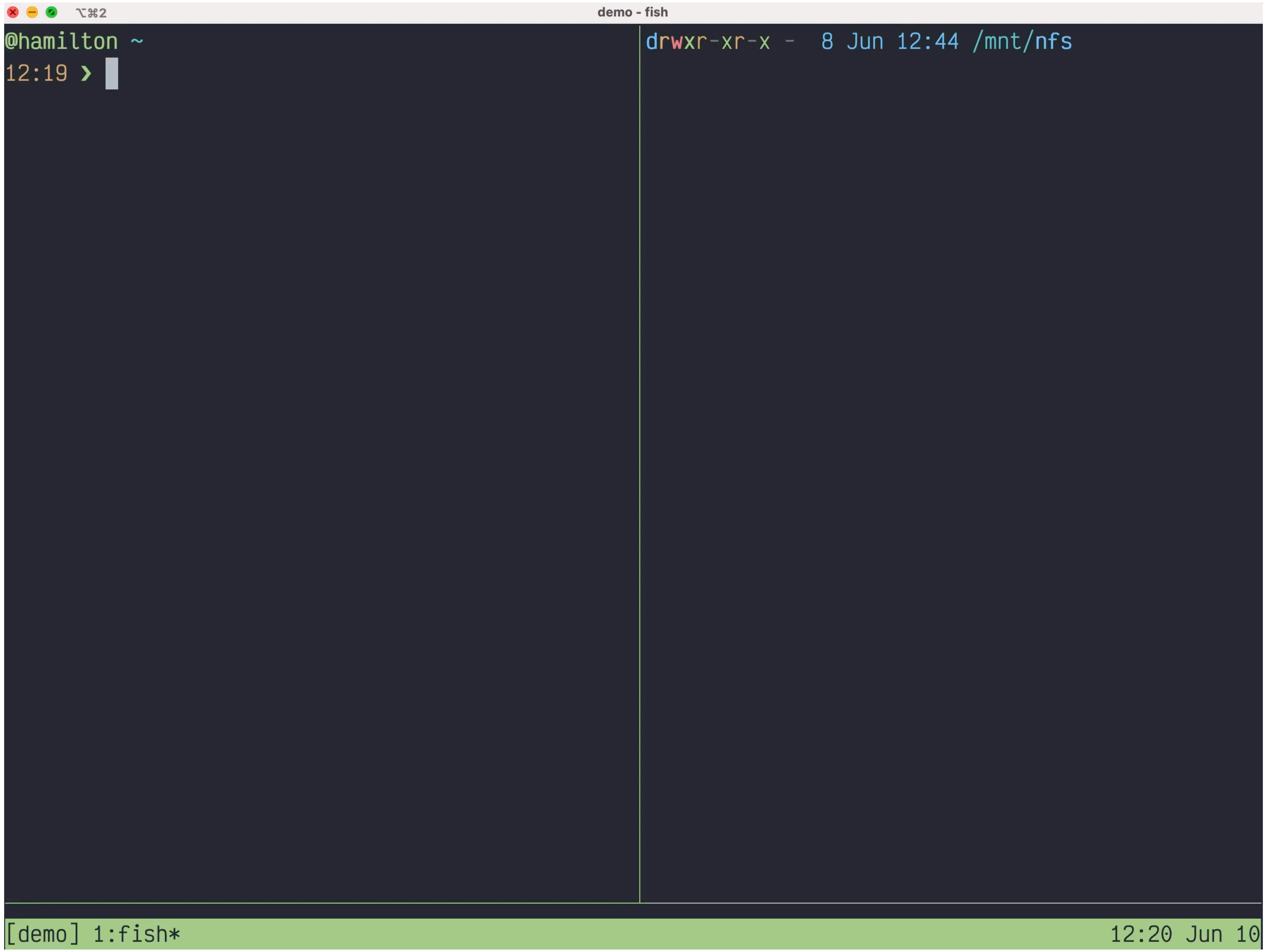
…and DaisyNFS is a verified file system



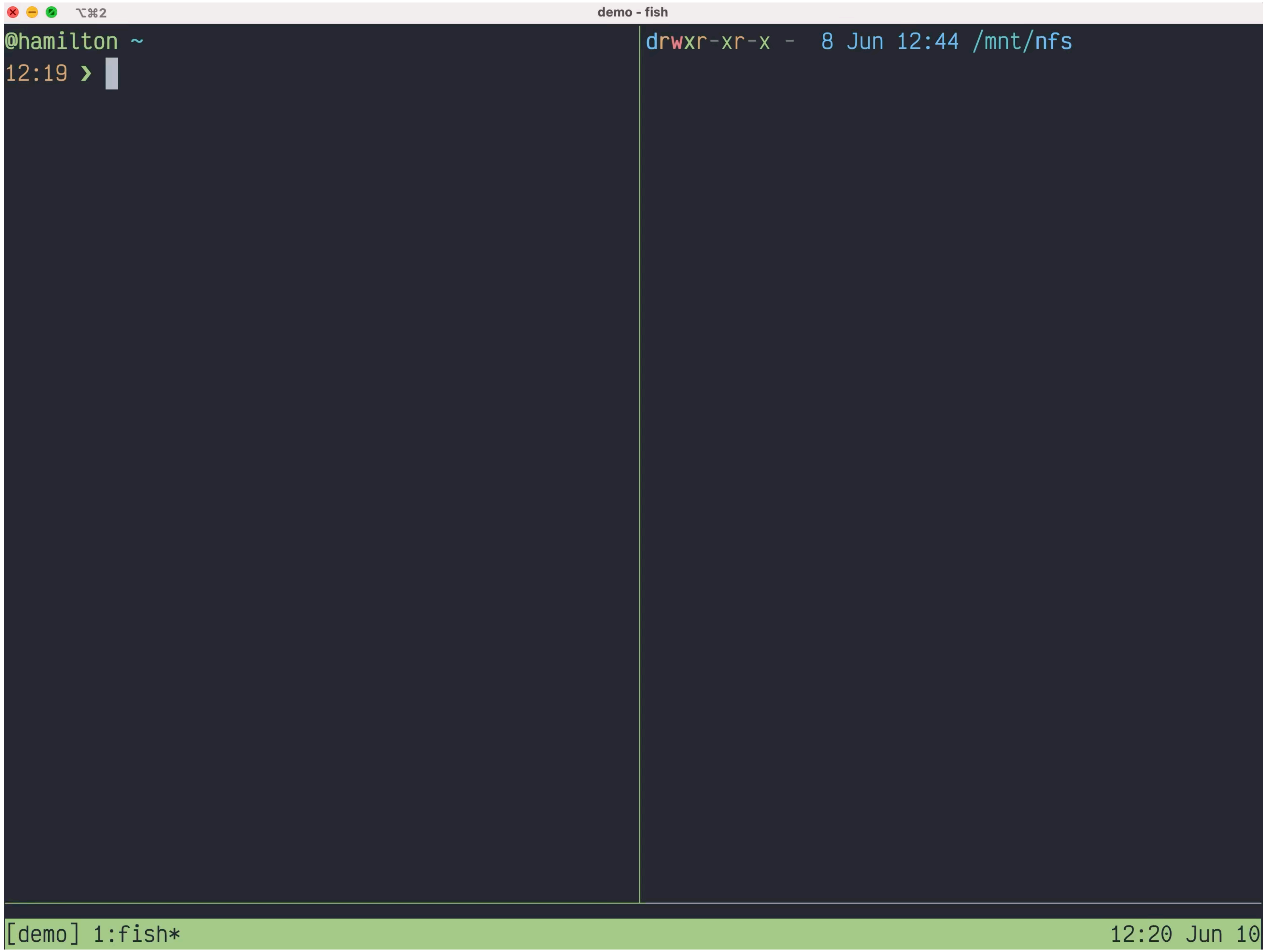
performant NFS server
comes with a machine-checked **proof**

atomic, crash-safe transactions
comes with a machine-checked **proof**

DaisyNFS is a real file system



DaisyNFS is a real file system



Current approaches cannot handle a system of with these features or complexity

crash safe but sequential file systems
FSCQ, Yggdrasil, VeriBetrKV

concurrent systems
CertiKOS, Armada, CIVL

crash safe and concurrent
Perennial 1.0

Contributions

GoJournal, the first verified transaction system

Perennial 2.0, a new verification framework

DaisyNFS, a verified concurrent file system

Contributions

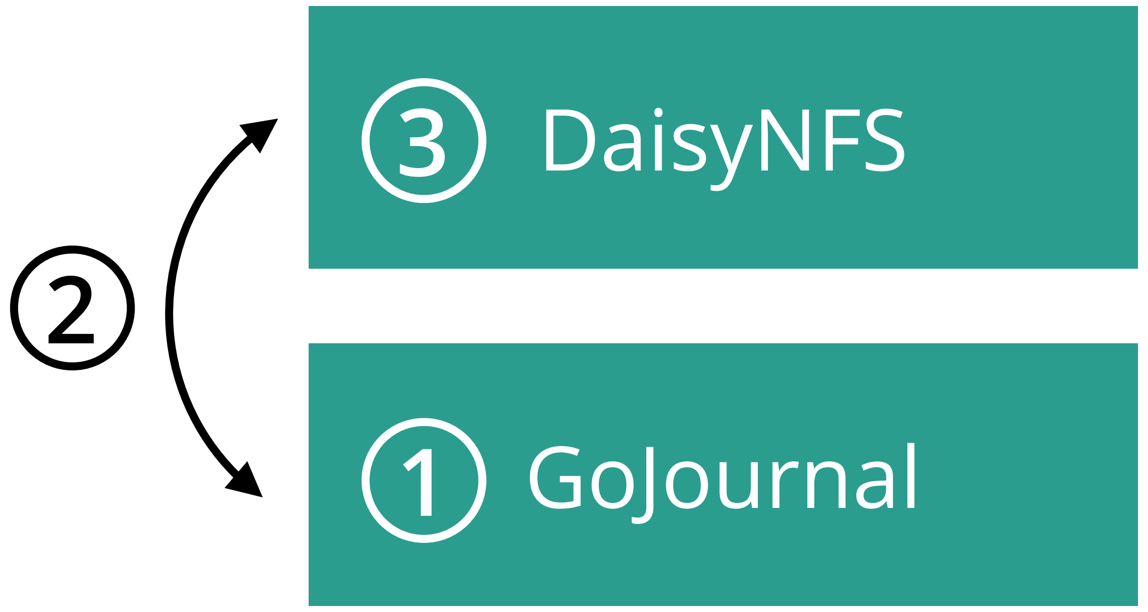
interactive proofs — GoJournal, the first verified transaction system

Perennial 2.0, a new verification framework

automated proofs — DaisyNFS, a verified concurrent file system

Strategy for verifying DaisyNFS with Perennial and Dafny

Outline for this talk



1. Why is GoJournal challenging to verify?

2. How do we connect GoJournal to Dafny?

3. How do we prove DaisyNFS correct?

DaisyNFS proof in a nutshell

mechanized proofs

Theorem 1 (Coq): any program using GoJournal transactions might as well have atomic transactions

Theorem 2 (Dafny): *if every operation runs atomically*, DaisyNFS implements NFS spec

DaisyNFS proof in a nutshell

mechanized proofs

Theorem 1 (Coq): any program using GoJournal transactions might as well have atomic transactions

Theorem 2 (Dafny): *if every operation runs atomically*, DaisyNFS implements NFS spec

Theorem 3: DaisyNFS implements NFS spec atomically.

Proof: apply theorem 1, then theorem 2

GoJournal specification is not so simple

Theorem 1 (Coq): **any program*** using GoJournal transactions **might as well**† have atomic transactions

*restrictions may apply

†in some sense

GoJournal

Connecting GoJournal to Dafny

DaisyNFS

Note on publications

"GoJournal: a verified, concurrent, crash-safe journaling system"
from OSDI 2021

DaisyNFS
under submission to SOSP 2021

GoJournal provides atomic transactions

```
// one-time init
var d Disk
jrnl := OpenJrnl(d)
```

GoJournal provides atomic transactions

```
// one-time init
var d Disk
jrnl := OpenJrnl(d)
```

```
// copy block at 0 to 1 and 2
tx := jrnl.Begin()
buf := tx.ReadBuf(0, blockSz)
tx.OverWrite(1, buf.Data)
tx.OverWrite(2, buf.Data)
tx.Commit()
```

GoJournal provides atomic transactions

```
// one-time init
var d Disk
jrnl := OpenJrnl(d)
```

```
// copy block at 0 to 1 and 2
tx := jrnl.Begin()
buf := tx.ReadBuf(0, blockSz)
tx.OverWrite(1, buf.Data)
tx.OverWrite(2, buf.Data)
tx.Commit()
```

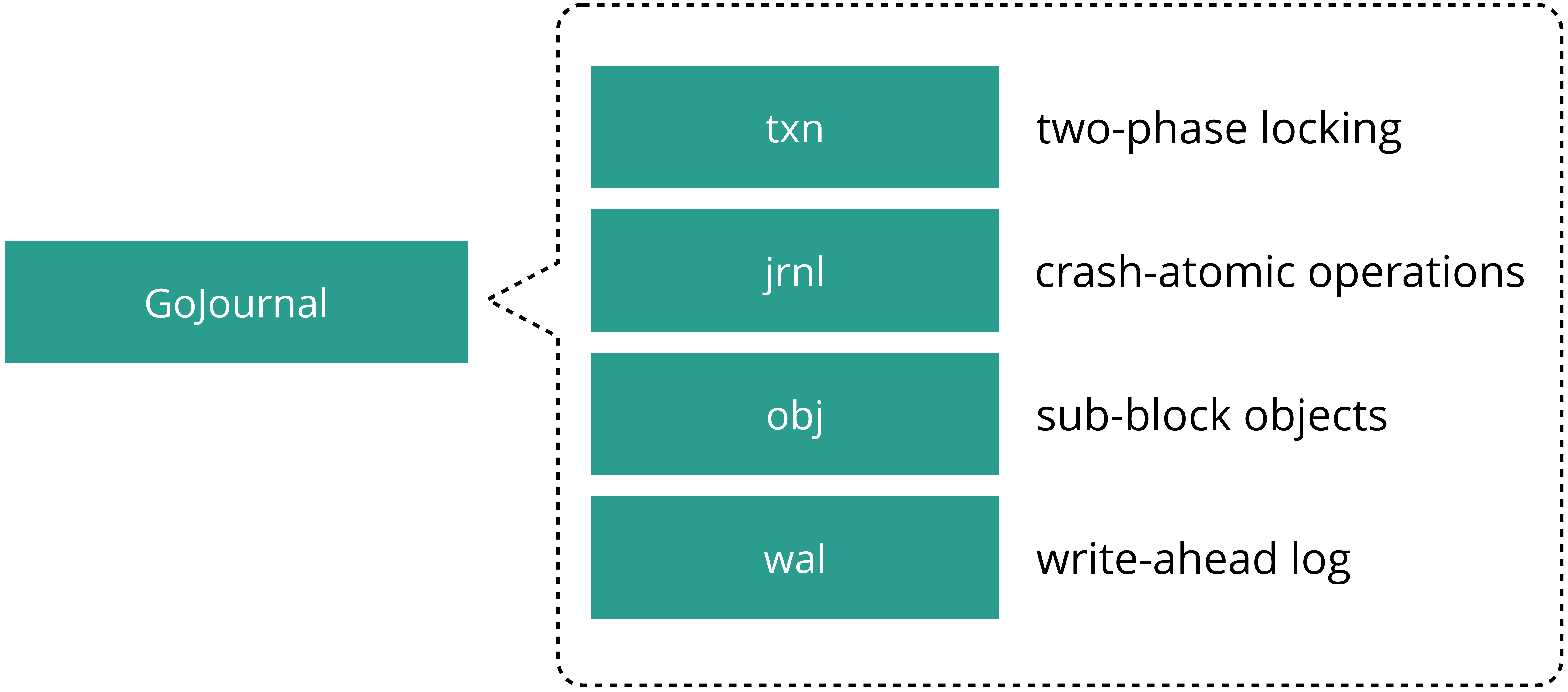
transactions proceed concurrently if they access different objects

Transactions can read and write objects within a block

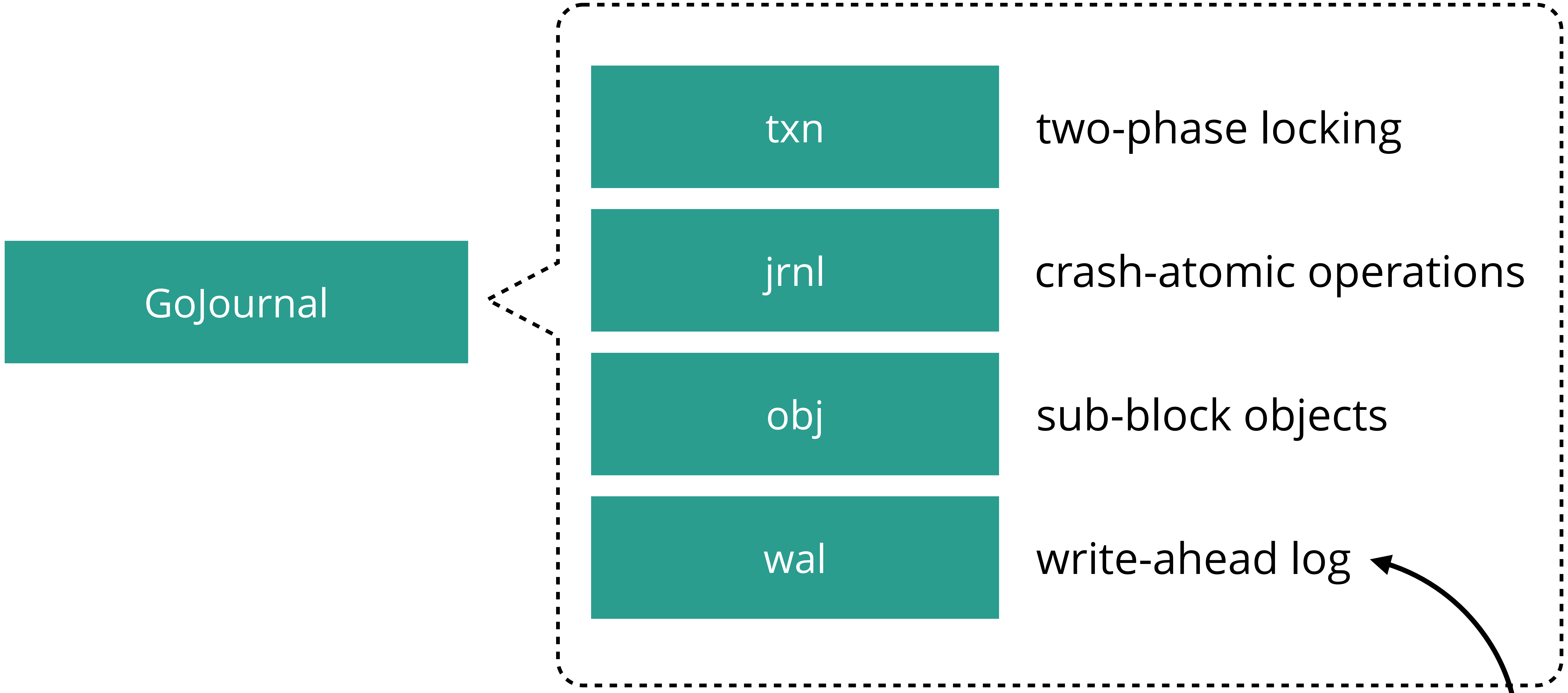
File system has 128-byte inodes

Sub-block access means locking is more fine-grained

GoJournal has a modular implementation and proof

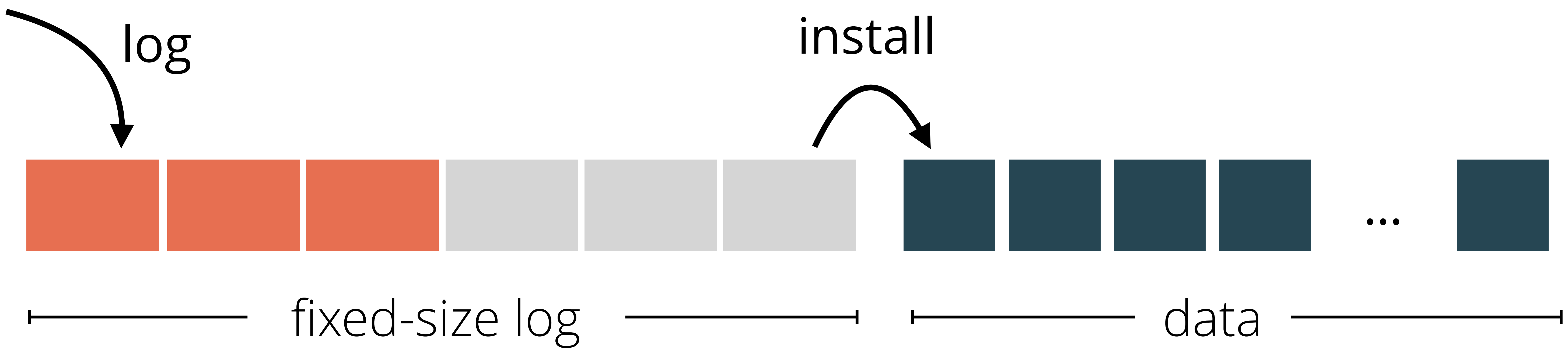


GoJournal has a modular implementation and proof



many interesting challenges, this talk will focus on just one in the write-ahead log

Write-ahead logging is the core atomicity primitive



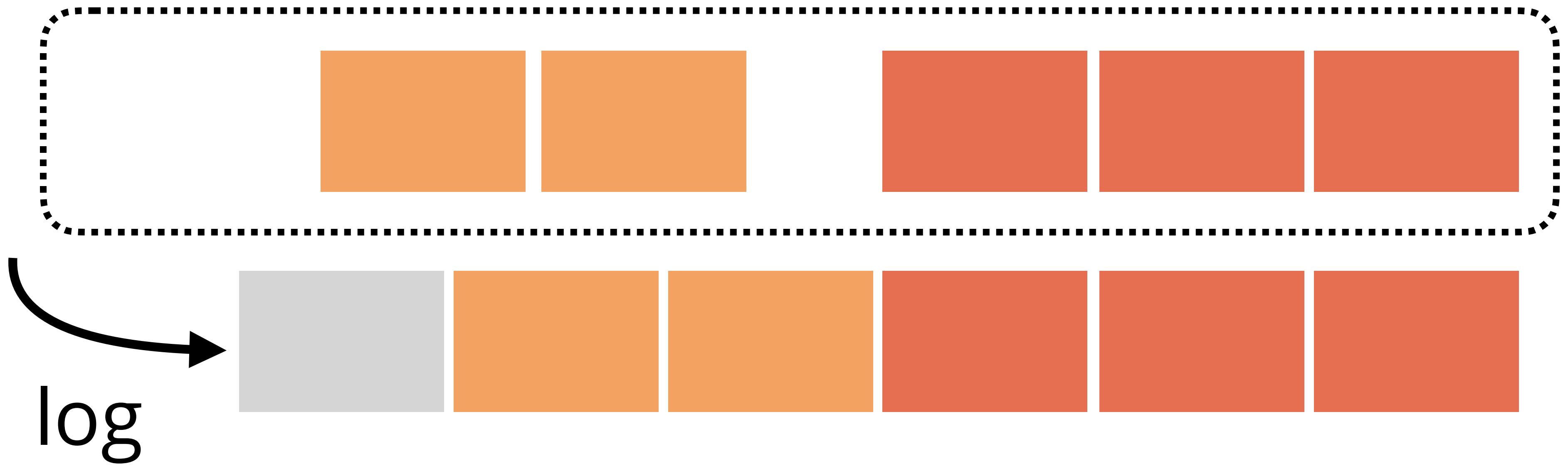
Writes are buffered before being logged

1 write gets buffered

Writes are buffered before being logged

1 write gets buffered

2 write gets logged



Writes are not atomic with crashes

1 write gets buffered

2 read returns new data

system crashes here

Writes are not atomic with crashes

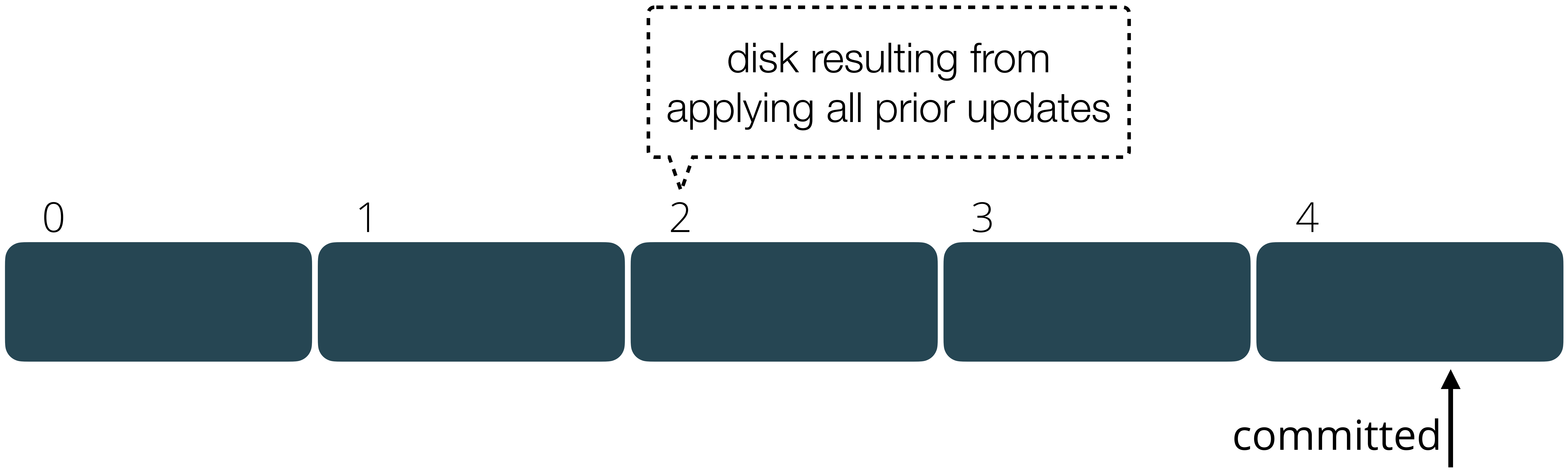
1. write gets buffered

2. read returns new data

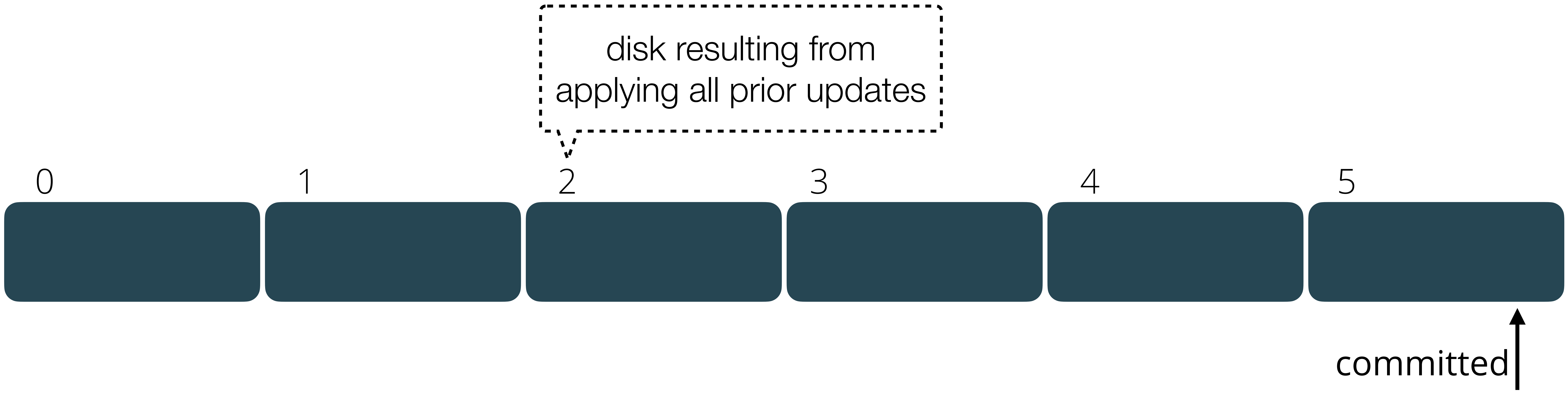
system crashes here

3. **read returns old data**

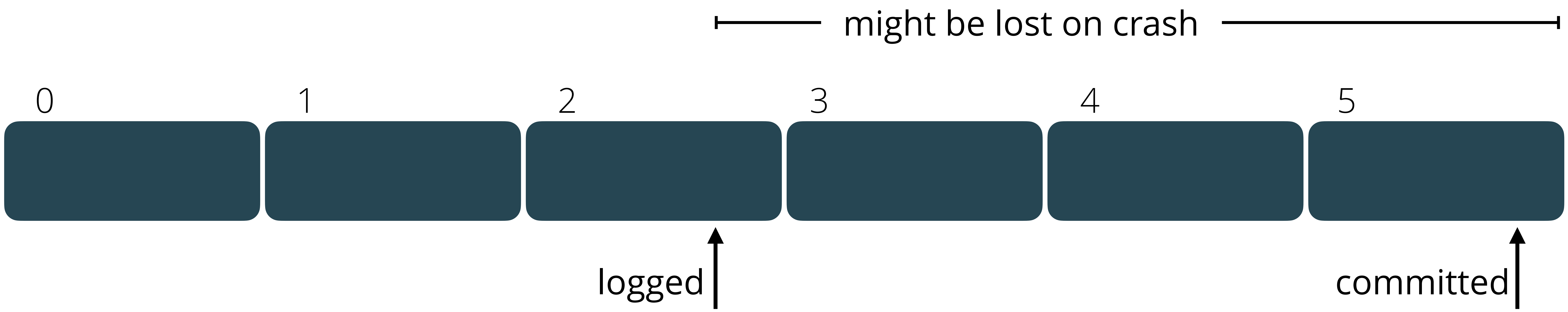
Write-ahead log state is a sequence of disks



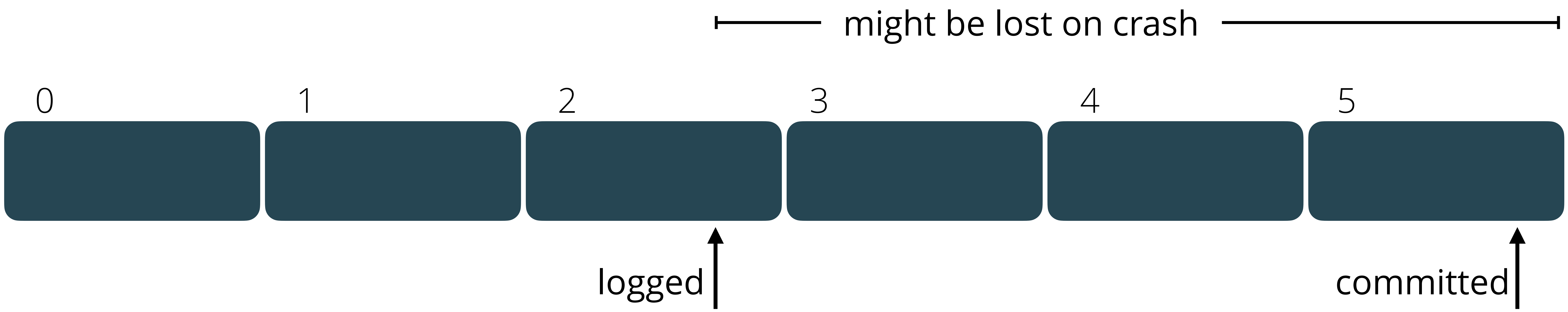
Write-ahead log state is a sequence of disks



Caller must reason about durable point

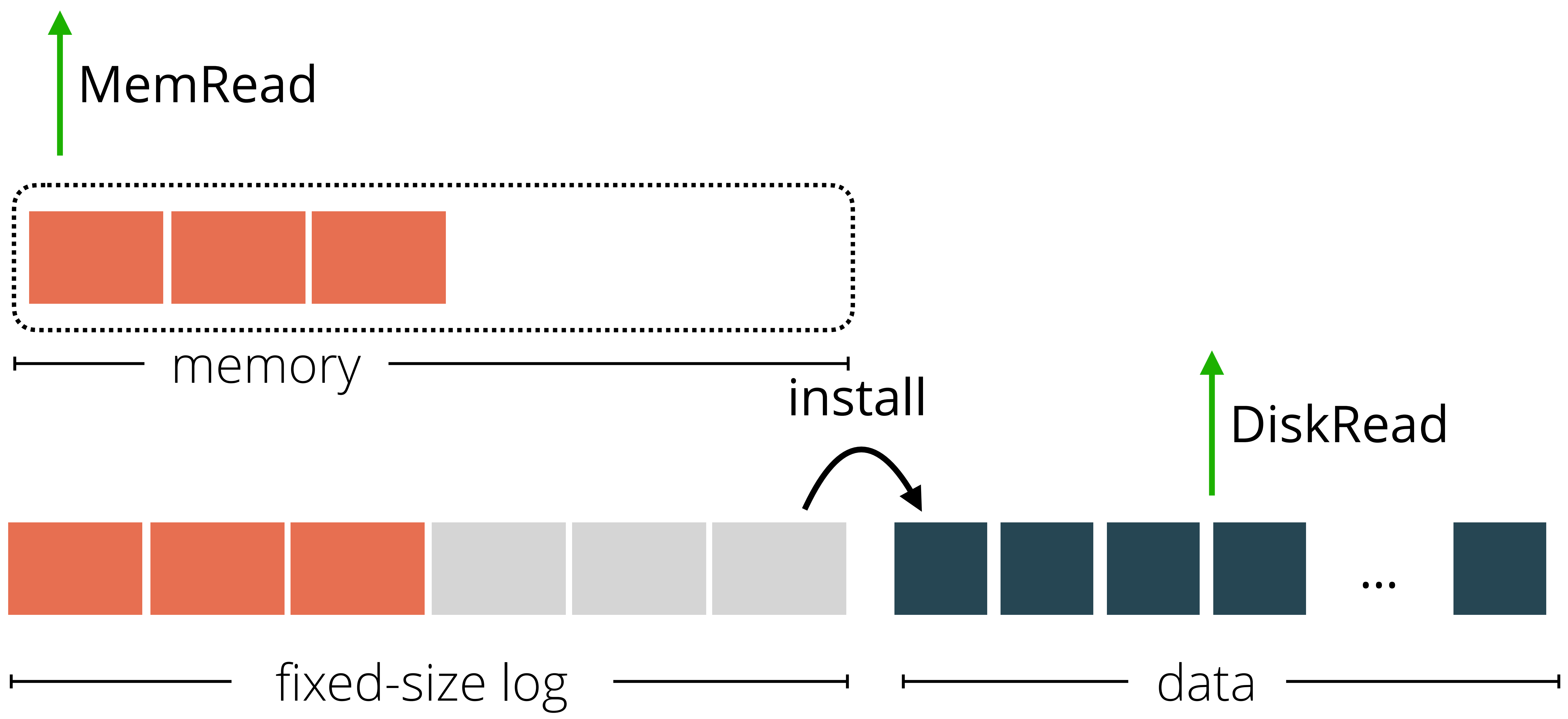


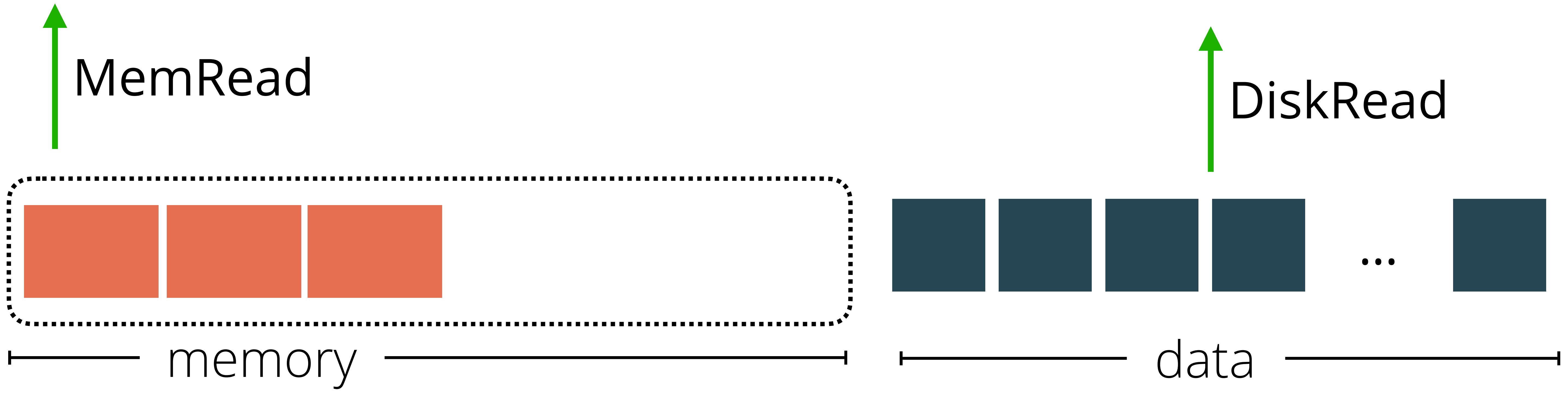
Caller must reason about durable point



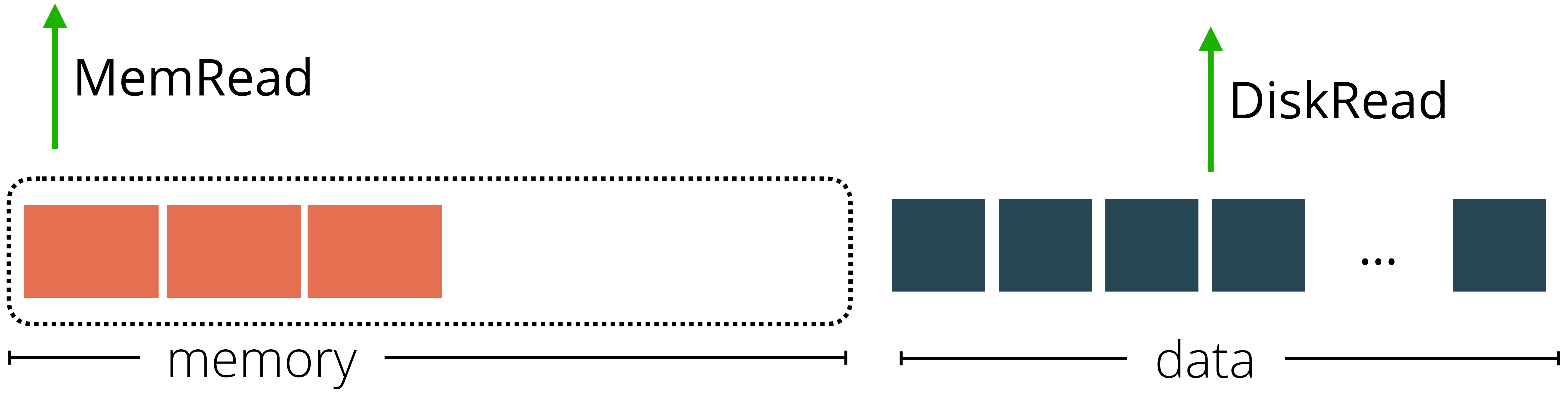
logger can advance the real durable point at any time, so caller actually only knows **lower bound**

Log is cached for the purposes of reading



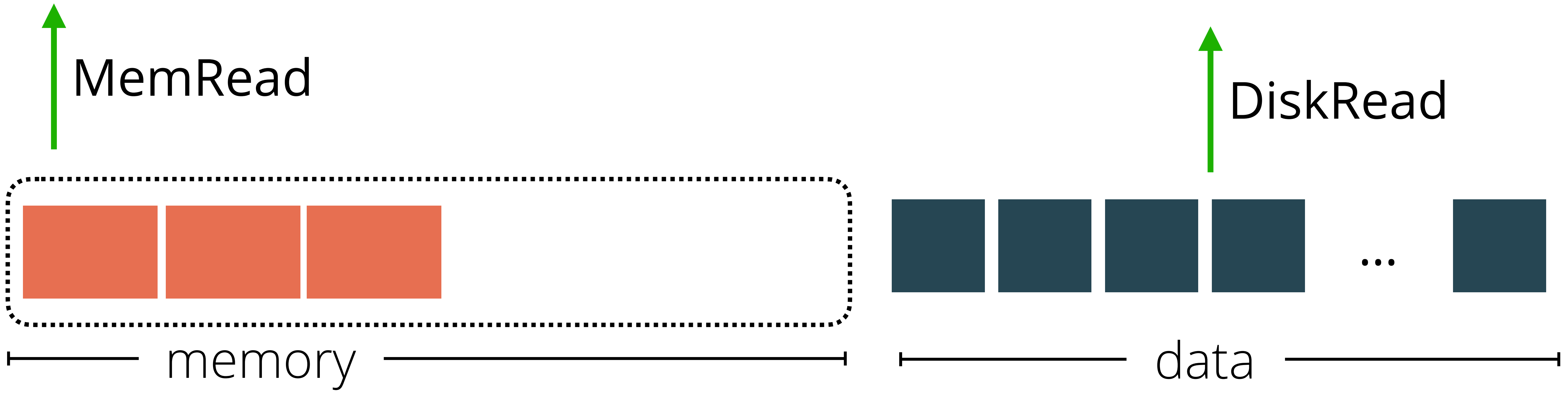

MemRead
DiskRead
...
memory
data

Reading is deceptively simple



Reading is deceptively simple

```
func Read(a) Block {
  v, ok := MemRead(a)
  if ok {
    return v
  }
  return DiskRead(a)
}
```



Is Read atomic?

```
v, ok := MemRead(a)
if ok {
  return v
}
return DiskRead(a)
```

Is Read atomic?

```
v, ok := MemRead(a)
if ok {
  return v
}
return DiskRead(a)
```

MemRead(a) → miss

Write(a, v’)

Is Read atomic?

```
v, ok := MemRead(a)
if ok {
  return v
}
return DiskRead(a)
```

MemRead(a) → miss

Write(a, v’)

A install v’

Is Read atomic?

```
v, ok := MemRead(a)
if ok {
  return v
}
return DiskRead(a)
```

MemRead(a) → miss

Write(a, v')

A install v'

B DiskRead(a) → v'

Is Read atomic?

```
v, ok := MemRead(a)
if ok {
  return v
}
return DiskRead(a)
```

MemRead(a) → miss

Write(a, v')

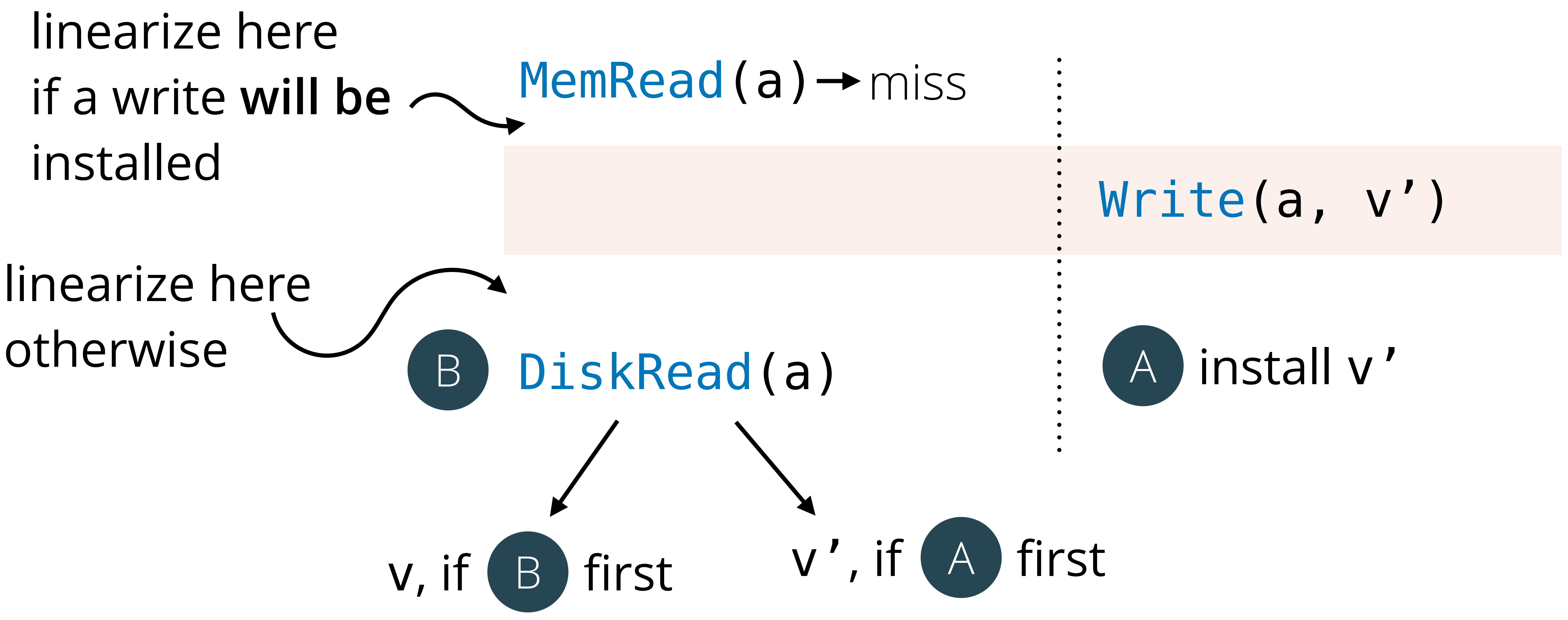
B DiskRead(a) → v

incorrect if Write has happened!

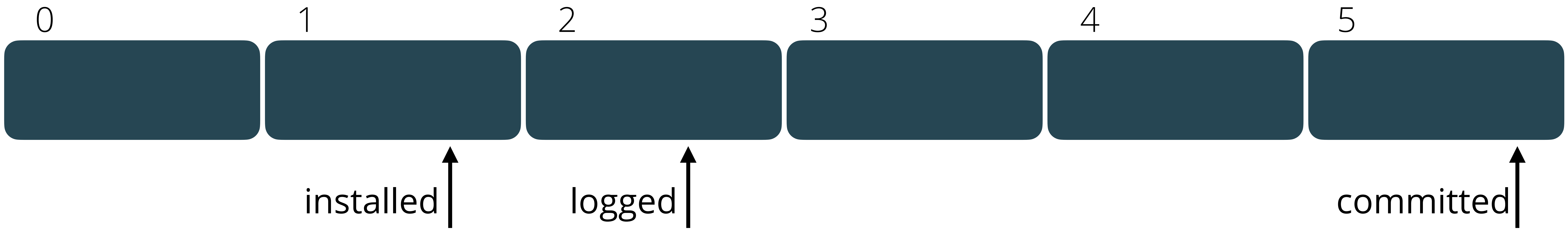
A install v'

Write-ahead log Read is atomic in a future-dependent way

```
v, ok := MemRead(a)
if ok {
  return v
}
return DiskRead(a)
```



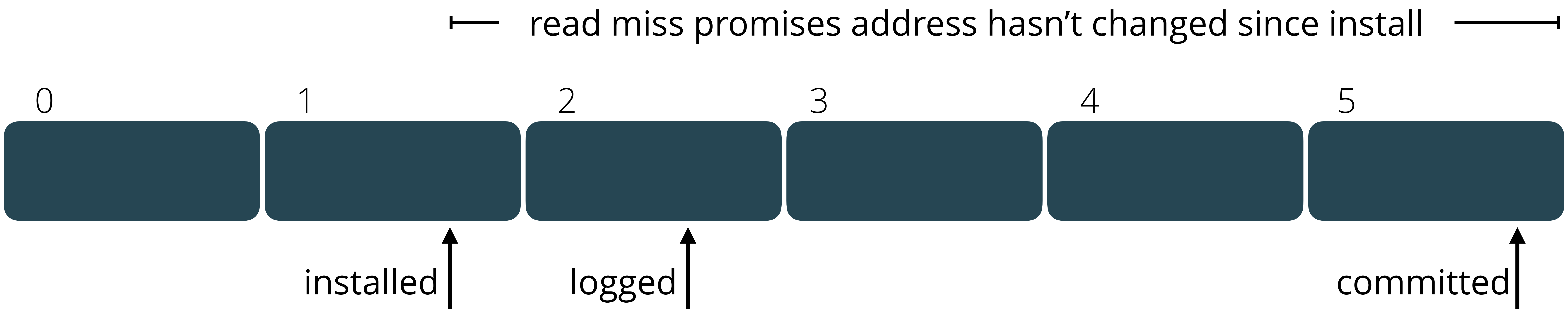
Reads are specified as two atomic operations



`DiskRead` reads installed disk

`MemRead` checks latest disk

Reads are specified as two atomic operations



`DiskRead` reads installed disk

`MemRead` checks latest disk

Perennial 2.0 introduces new verification techniques to prove GoJournal correct

see OSDI 2021 paper for details

Logically atomic crash specifications

Lifting for internal GoJournal specification

Crash-aware lock specification

One new idea in Perennial 2.0: ownership of durable state

Lock specification says nothing during a critical section

One new idea in Perennial 2.0: ownership of durable state

Lock specification says nothing during a critical section

New **crash-aware lock spec** adds a new crash invariant that must hold even during critical section

Limitations

Assume synchronous disk

Don’t support unstable transactions

Can’t incorporate other data structures or storage

GoJournal

Connecting GoJournal to Dafny

DaisyNFS

Intuitively, transactions are atomic

```
tx := jrnl.Begin()
buf := tx.ReadBuf(0, blockSz)
tx.OverWrite(1, buf.Data)
tx.OverWrite(2, buf.Data)
tx.Commit()
```

||

```
tx := jrnl.Begin()
tx.OverWrite(7, data)
tx.Commit()
```

GoJournal specification relates code programs to simpler spec programs

$p_c : \mathtt{Go}\langle\text{Disk}\rangle$

```
tx := jrnl.Begin()
buf := tx.ReadBuf(0, blockSz)
tx.OverWrite(1, buf.Data)
tx.OverWrite(2, buf.Data)
tx.Commit()
```

GoJournal specification relates code programs to simpler spec programs

expand to (large) implementations

$$p_c : \mathtt{Go}\langle\text{Disk}\rangle$$

```
tx := jrnl.Begin()
buf := tx.ReadBuf(0, blockSz)
tx.OverWrite(1, buf.Data)
tx.OverWrite(2, buf.Data)
tx.Commit()
```

GoJournal specification relates code programs to simpler spec programs

$p_c : \texttt{Go}\langle\text{Disk}\rangle$

```
tx := jrnl.Begin()
buf := tx.ReadBuf(0, blockSz)
tx.OverWrite(1, buf.Data)
tx.OverWrite(2, buf.Data)
tx.Commit()
```

$p_s : \texttt{Go}\langle\text{Txn}\rangle$

```
Atomically (
  buf ← ReadBuf(0, blockSz);
  OverWrite(1, buf.Data);
  OverWrite(2, buf.Data)
)
```

GoJournal specification relates code programs to simpler spec programs

$p_c : \mathtt{Go}\langle\text{Disk}\rangle$

```
tx := jrnl.Begin()
buf := tx.ReadBuf(0, blockSz)
tx.OverWrite(1, buf.Data)
tx.OverWrite(2, buf.Data)
tx.Commit()
```

$p_s : \mathtt{Go}\langle\text{Txn}\rangle$

Atomically (
  buf ← ReadBuf(0, blockSz);
  OverWrite(1, *buf*.Data);
  OverWrite(2, *buf*.Data)
)

$p_c \subseteq p_s$ code *implements* spec if every code behavior is allowed by spec

Specification only holds for well-behaved programs

```
var x *int
```

```
tx := jrnl.Begin()
*x = *x + 1
tx.Commit()
```

Specification only holds for well-behaved programs

```
var x *int
```

```
tx := jrnl.Begin()
*x = *x + 1
tx.Commit()
```

‖

```
tx := jrnl.Begin()
*x = *x + 1
tx.Commit()
```

✗

Specification only holds for well-behaved programs

```
var x *int
```

```
tx := jrnl.Begin()
*x = *x + 1
tx.Commit()
```

‖

```
tx := jrnl.Begin()
*x = *x + 1
tx.Commit()
```

✗

Caller can use only transaction system for shared state, but can “think” between operations

Proof uses type system as a way to specify well-behaved transactions

$$\Gamma \vdash p_c \sim p_s : \tau$$

relational type system that relates two programs at a common type

Proof uses type system as a way to specify well-behaved transactions

$$\Gamma \vdash p_c \sim p_s : \tau$$

relational type system that relates two programs at a common type

informally, think of this as a partial function from p_s to p_c that "links" with the GoJournal implementation

Full GoJournal specification: program refinement

$$\forall p_c, p_s,$$

$$\vdash p_c \sim p_s : \tau \quad \rightarrow \quad p_c \subseteq p_s$$

GoJournal

Connecting GoJournal to Dafny

DaisyNFS

Coming back to DaisyNFS

$$\vdash \texttt{s}_{\text{code}} \sim \texttt{s}_{\text{dfy}} : \tau$$

DaisyNFS is our code program, so needs to be well-typed for the theorem to apply

Remainder of proof is an on-paper argument

Need to imagine DaisyNFS code modeled in $\mathtt{Go}\langle \mathrm{Txn} \rangle$

Need to convince ourselves that this model meets preconditions in GoJournal specification

DaisyNFS's overall correctness theorem

$$s_{\text{NFS}} \supseteq s_{\text{dfy}} \supseteq s_{\text{code}}$$

handler that responds
following NFS spec

DaisyNFS's overall correctness theorem

$$s_{\mathrm{NFS}} \supseteq s_{\mathrm{dfy}} \supseteq s_{\mathrm{code}}$$

handler that responds
following NFS spec

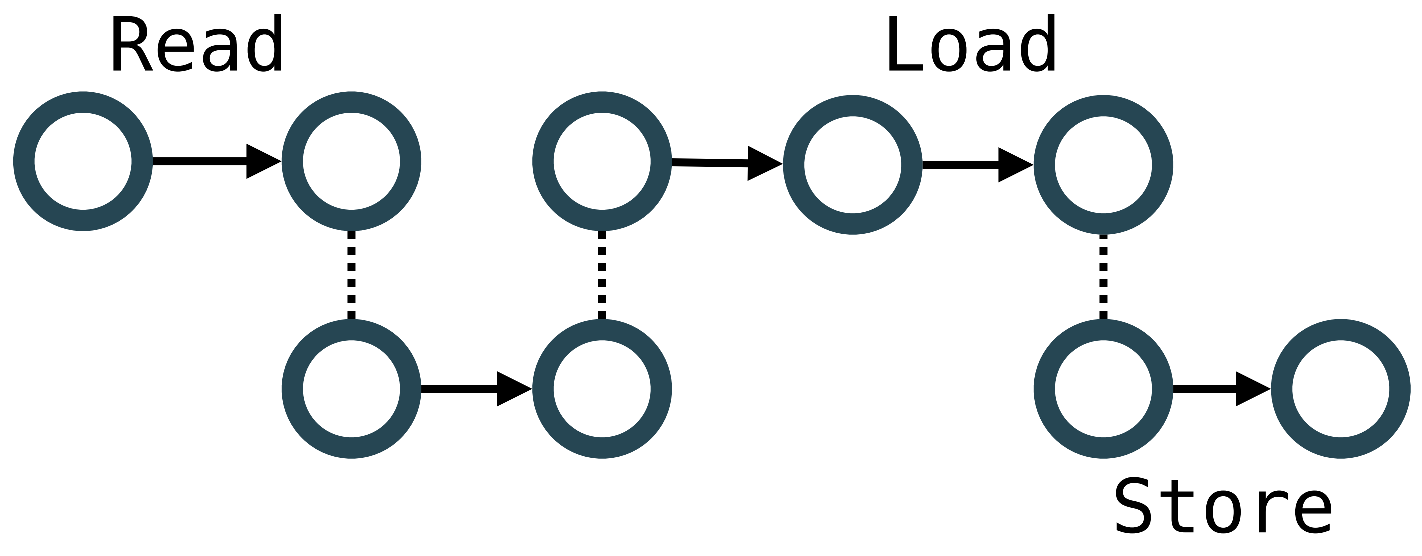
Dafny implementation

running code

An example of following this specification

$$s_{\text{NFS}} \supseteq s_{\text{dfy}} \supseteq s_{\text{code}}$$

$s_{\text{code}} : \texttt{Go}\langle \text{Disk} \rangle$

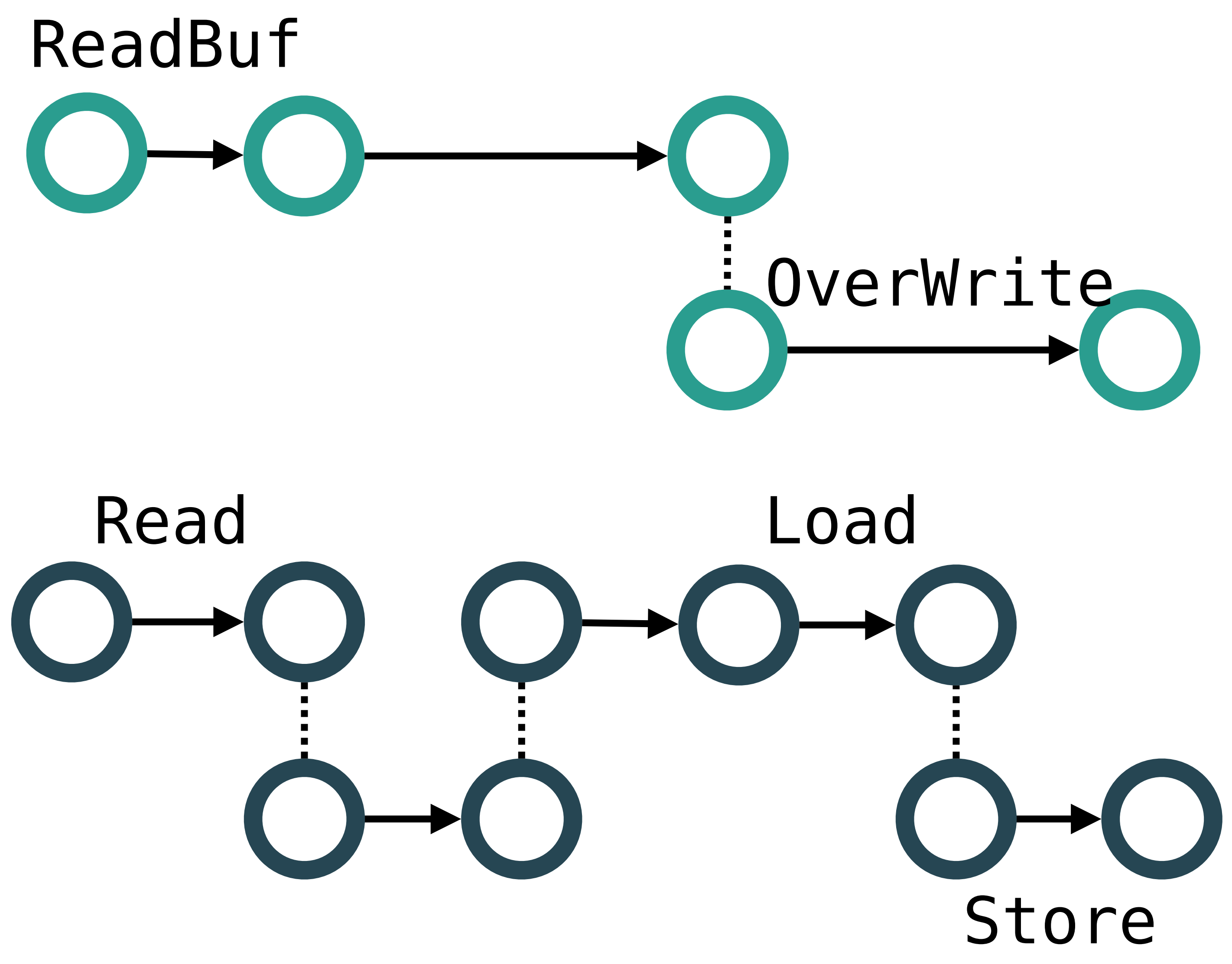


An example of following this specification

$$s_{NFS} \supseteq s_{dfy} \supseteq s_{code}$$

$s_{dfy} : \mathtt{Go}\langle \mathrm{Txn} \rangle$

$s_{code} : \mathtt{Go}\langle \mathrm{Disk} \rangle$



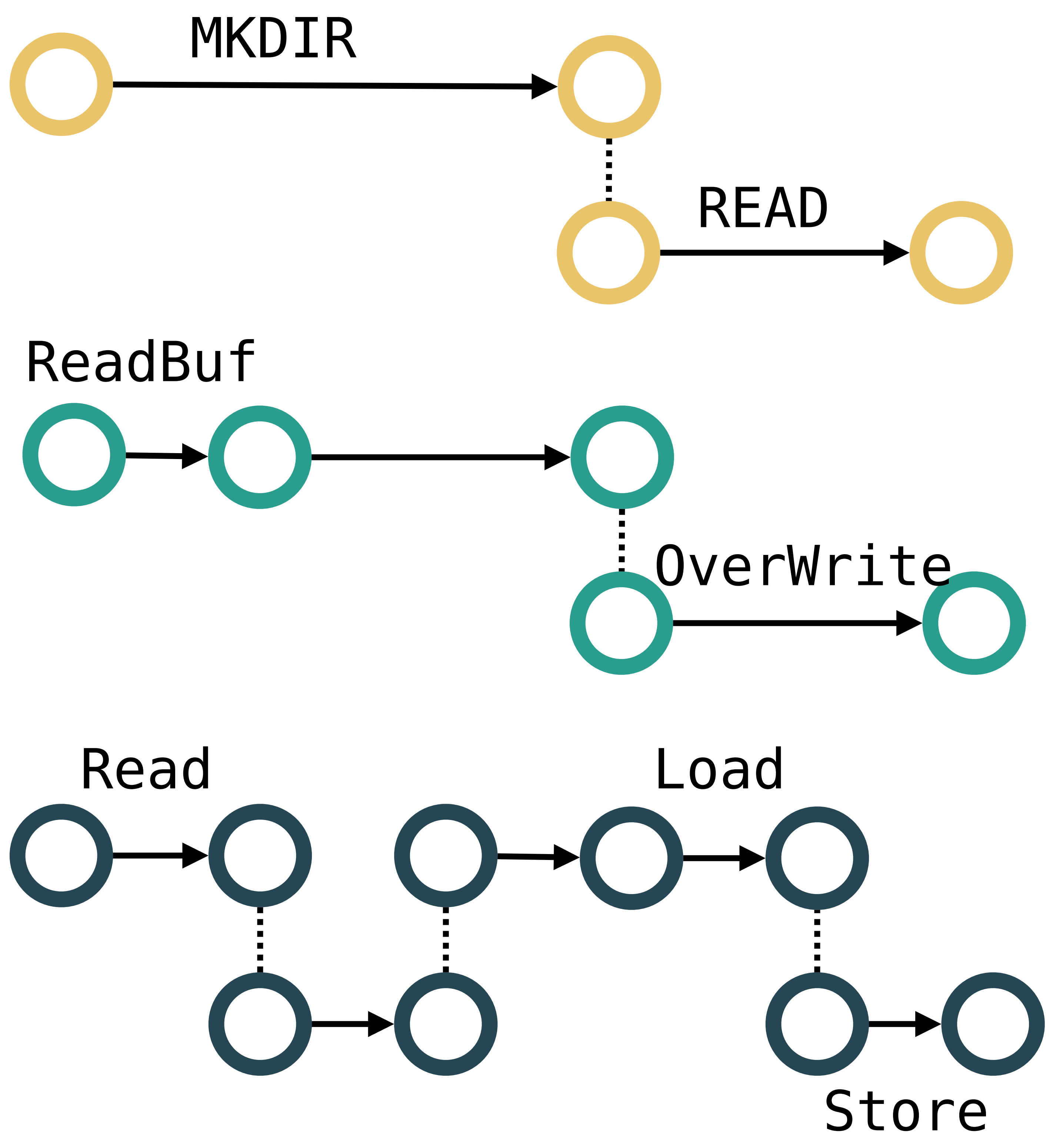
An example of following this specification

$$s_{\text{NFS}} \supseteq s_{\text{dfy}} \supseteq s_{\text{code}}$$

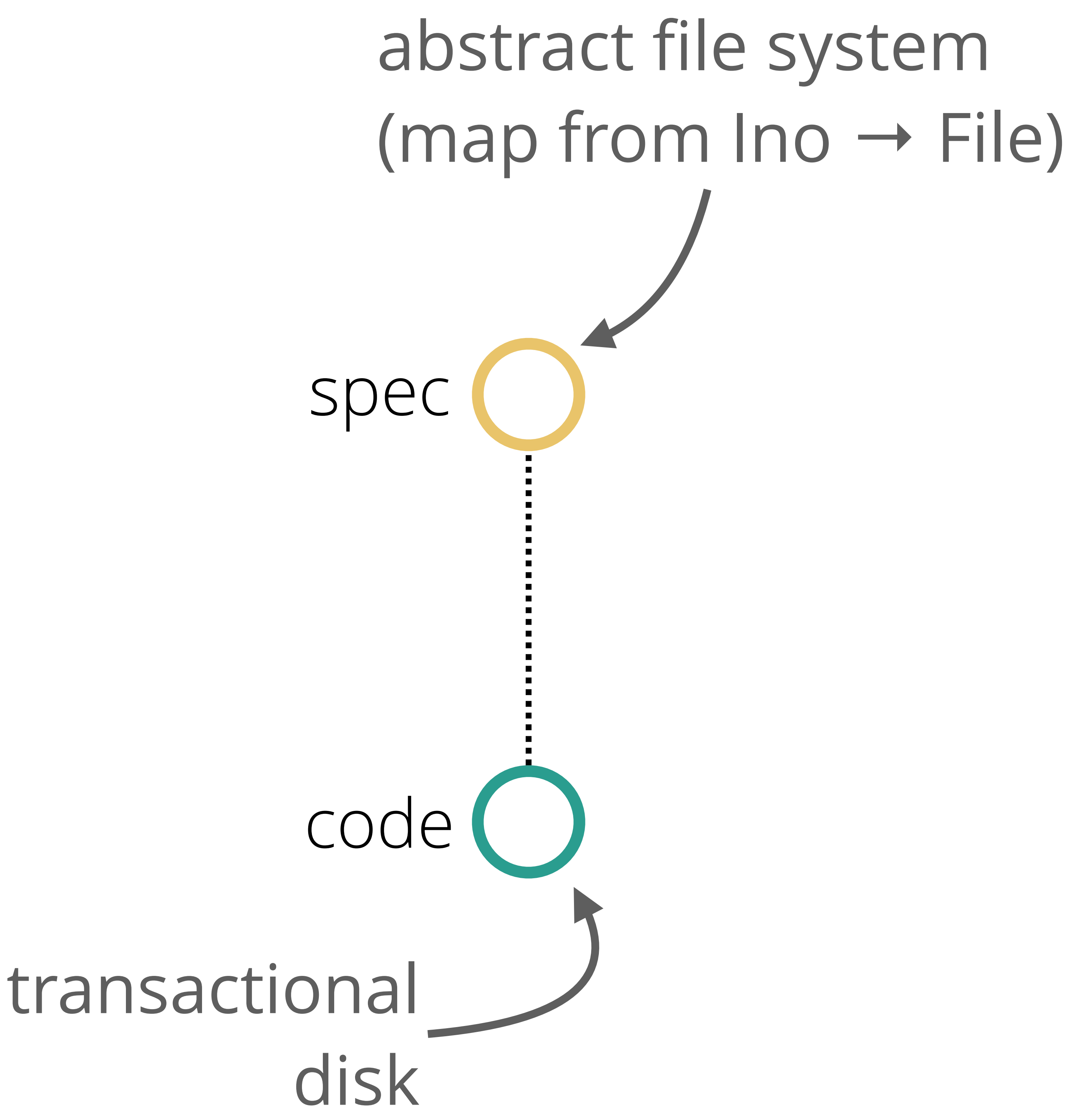
$s_{\text{NFS}} : \texttt{Go}\langle \text{NFS} \rangle$

$s_{\text{dfy}} : \texttt{Go}\langle \text{Txn} \rangle$

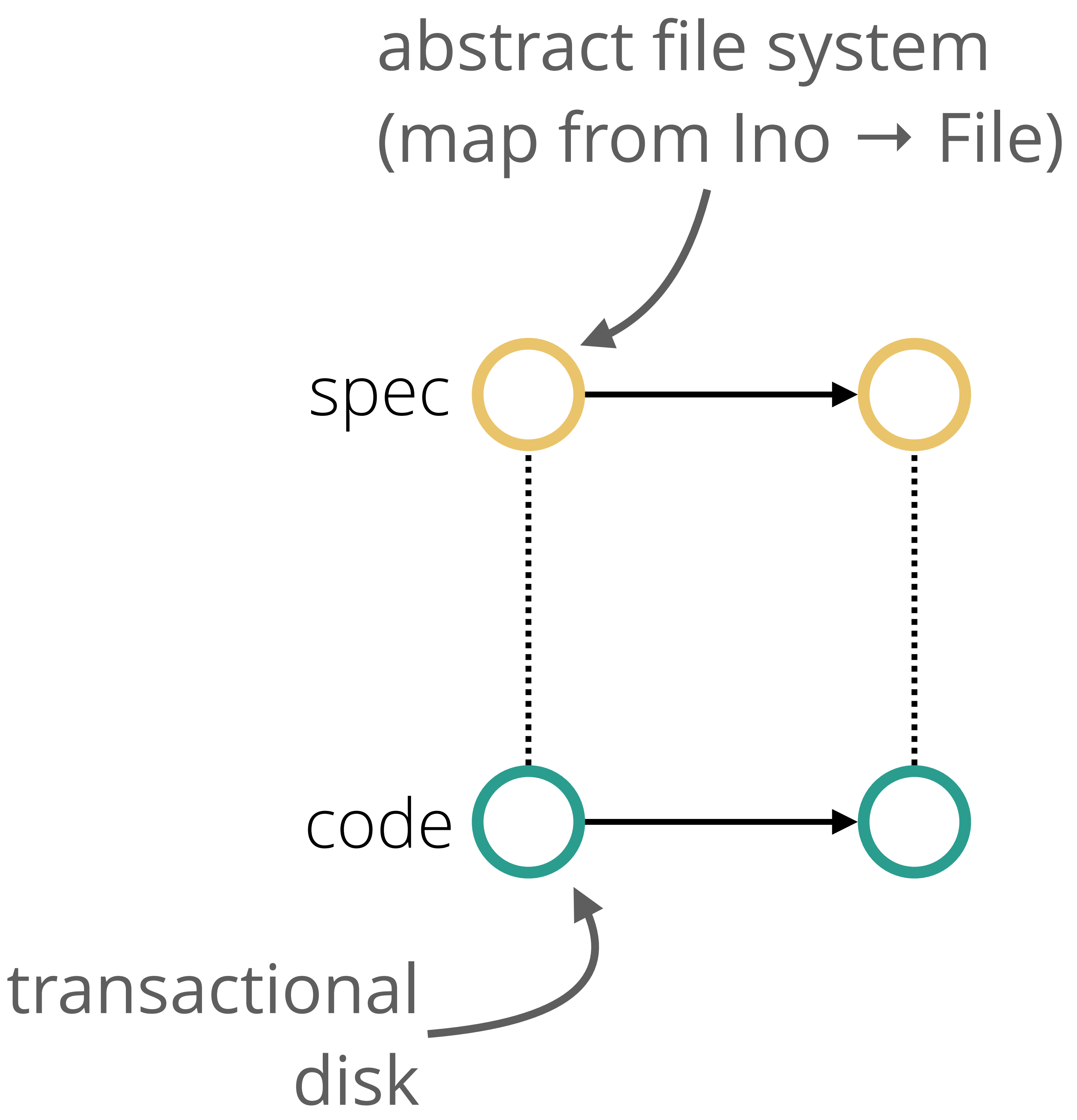
$s_{\text{code}} : \texttt{Go}\langle \text{Disk} \rangle$



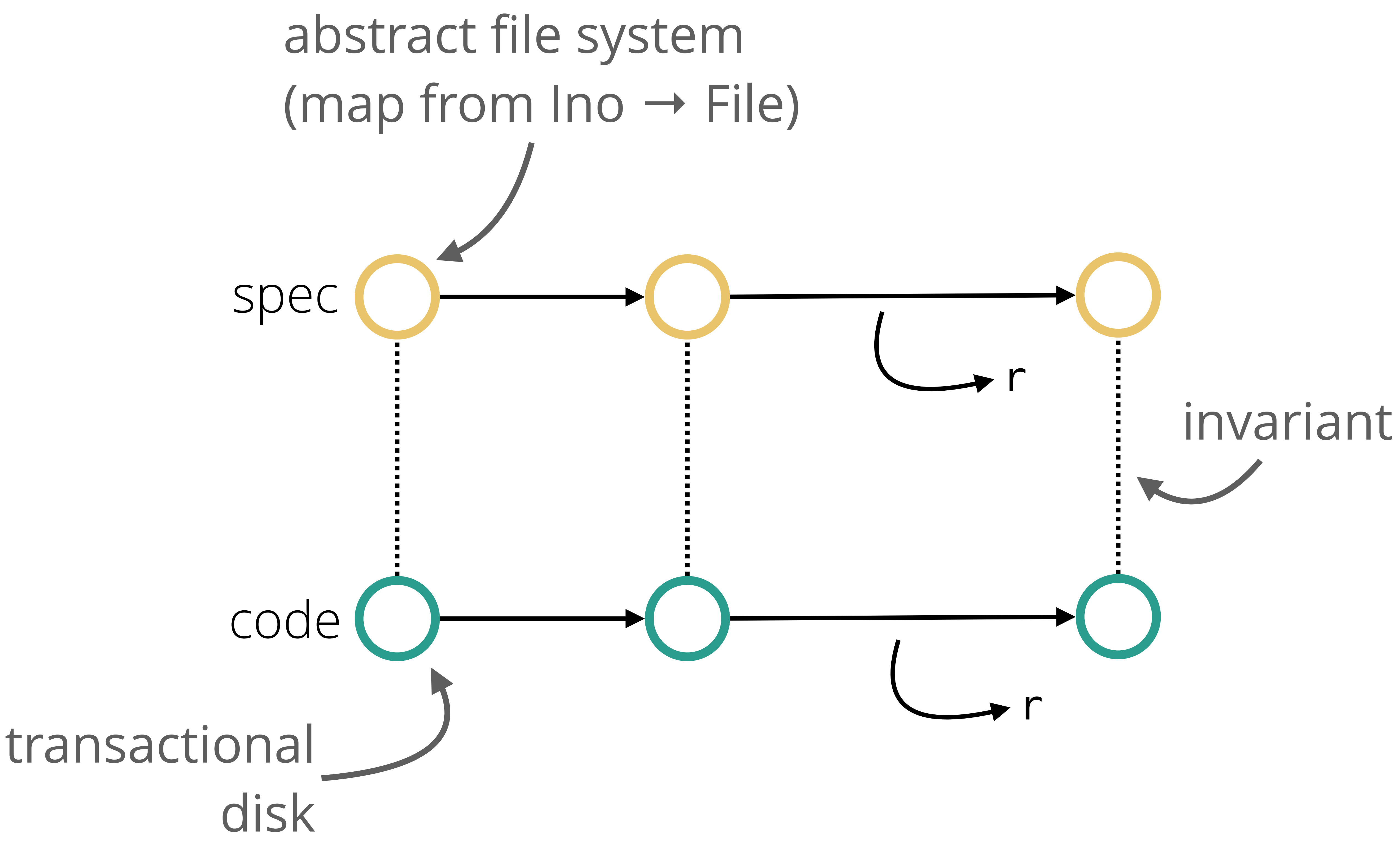
Thanks to transactions, Dafny proof is standard simulation



Thanks to transactions, Dafny proof is standard simulation



Thanks to transactions, Dafny proof is standard simulation

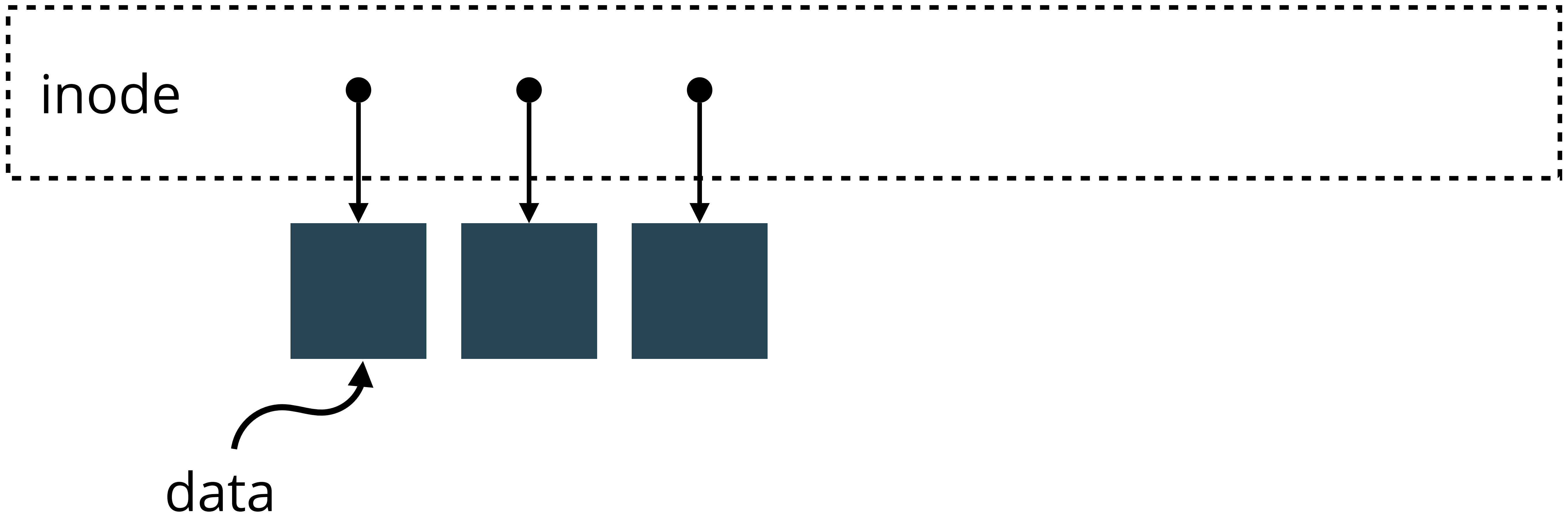


Two interesting challenges in DaisyNFS proof

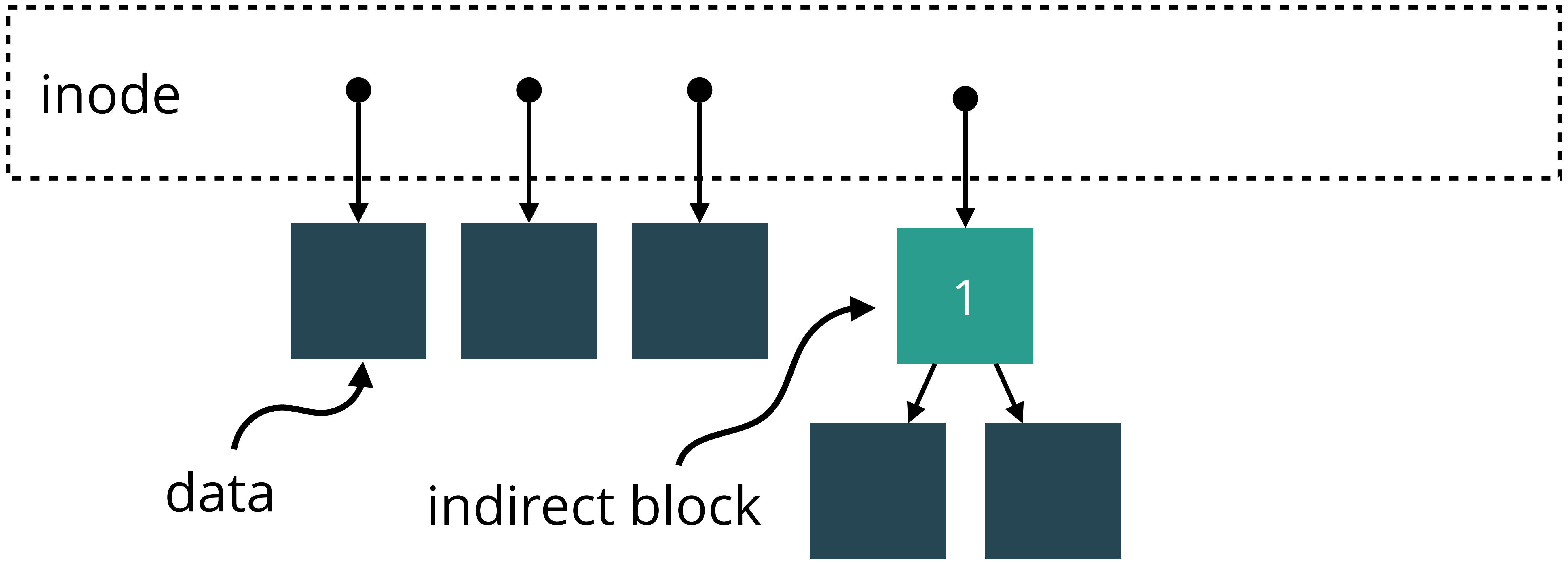
Supporting large files

Reclaiming free space

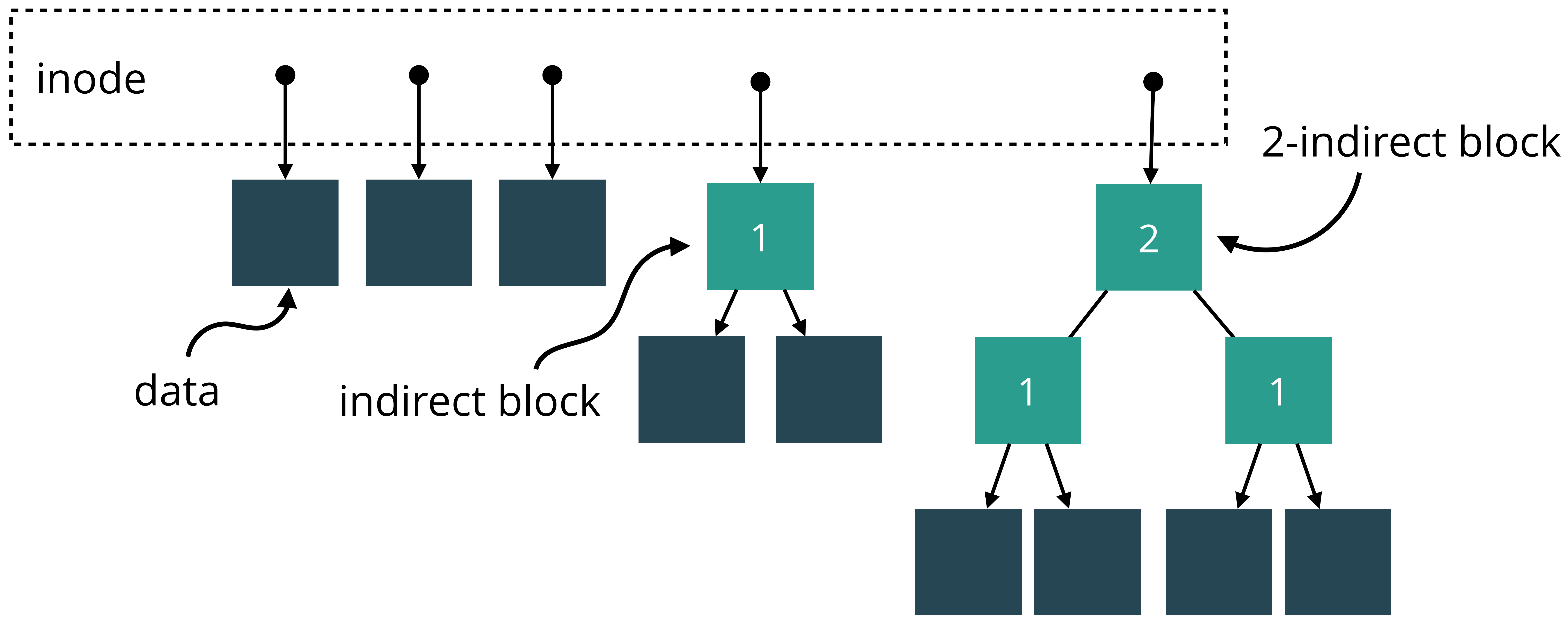
Large files require indirect blocks



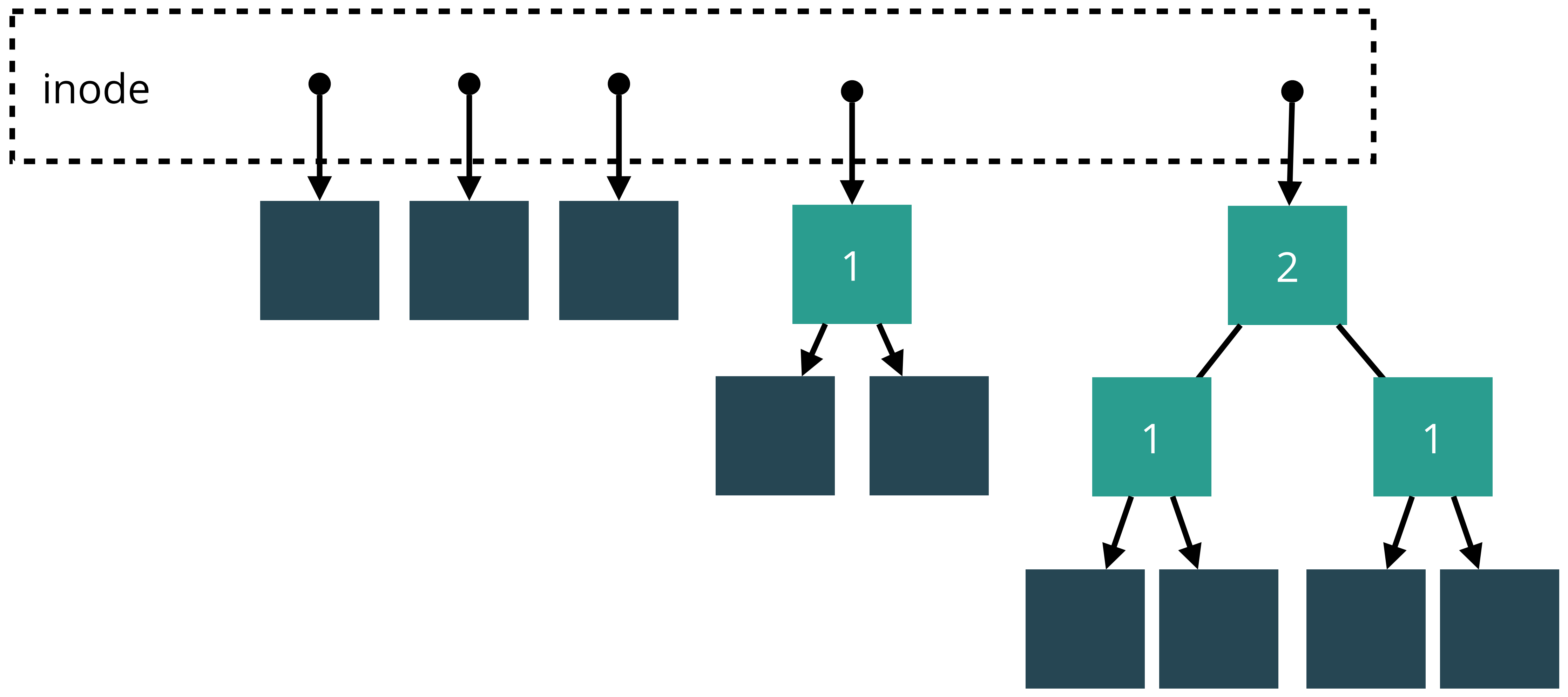
Large files require indirect blocks



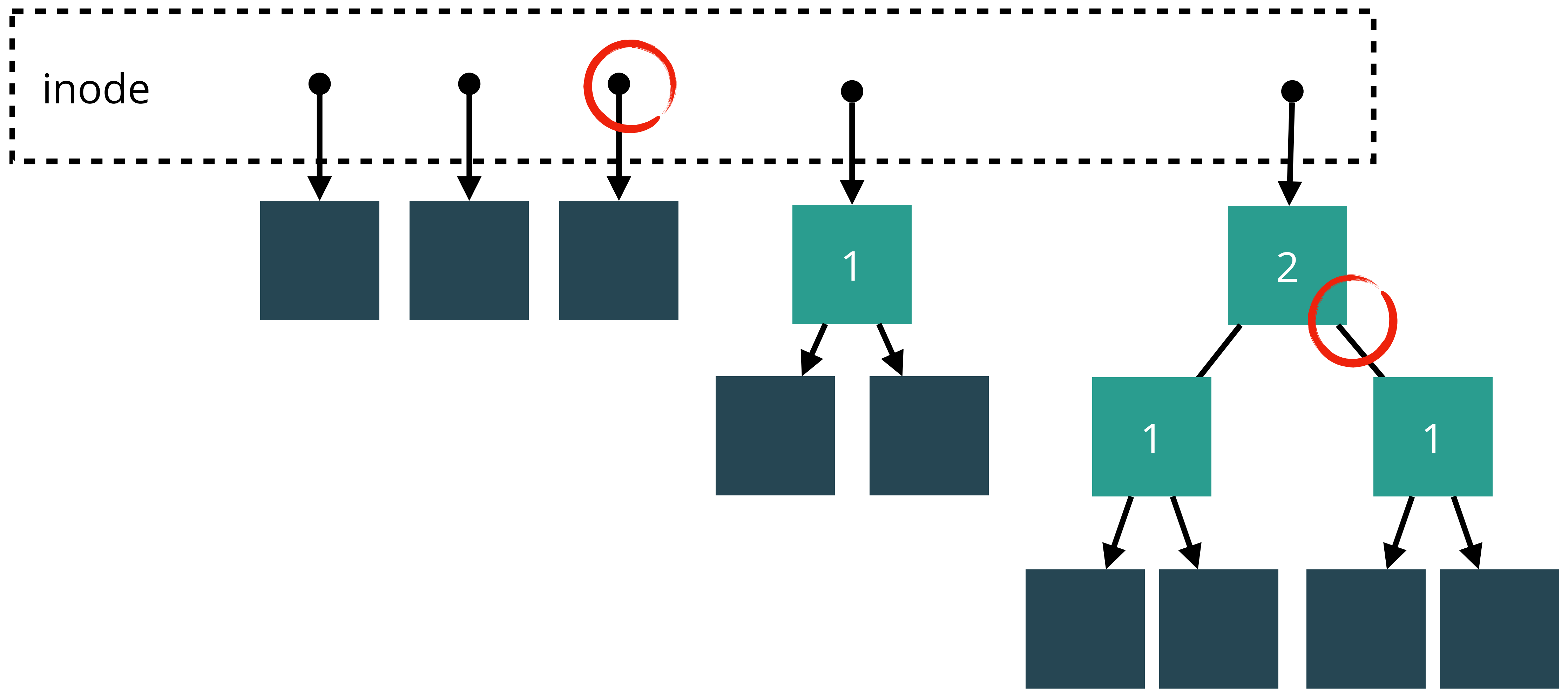
Large files require indirect blocks



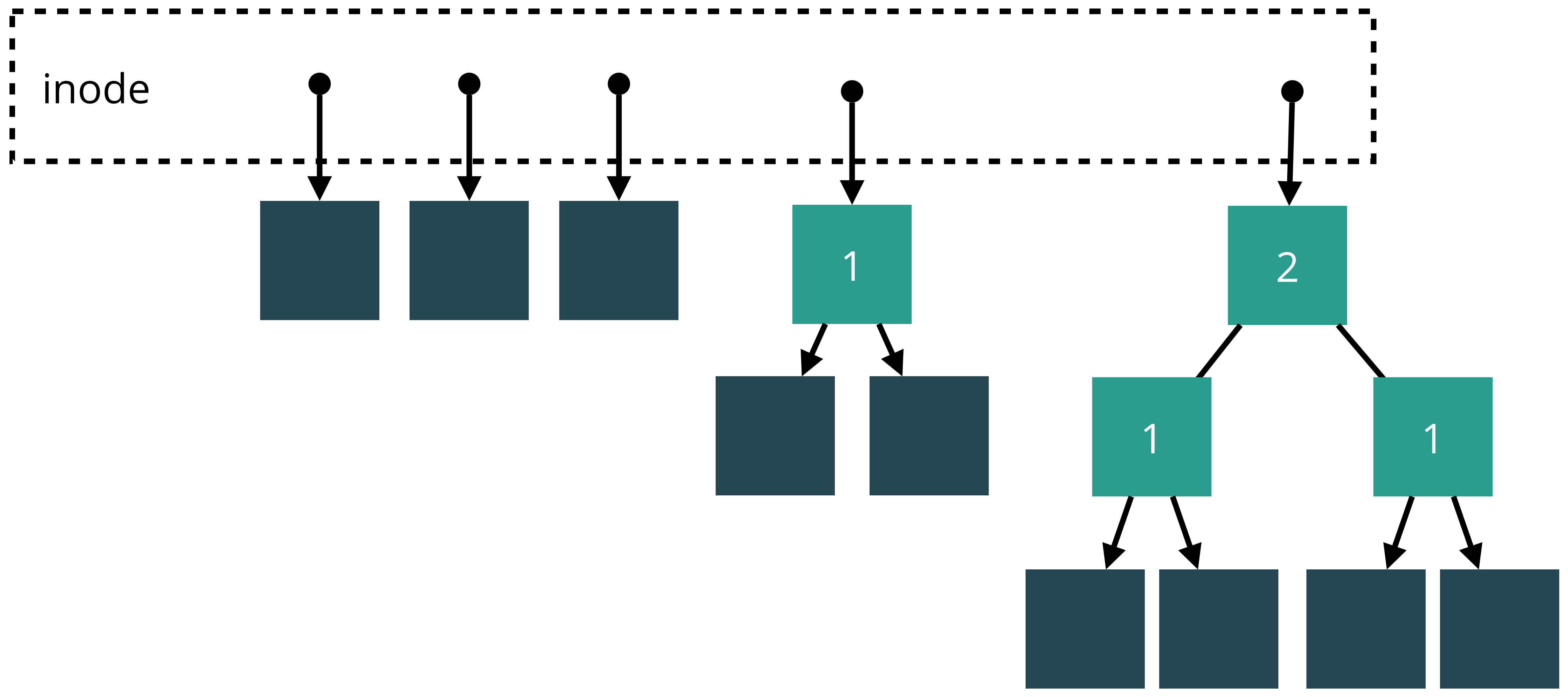
Invariant must ensures all pointers are distinct



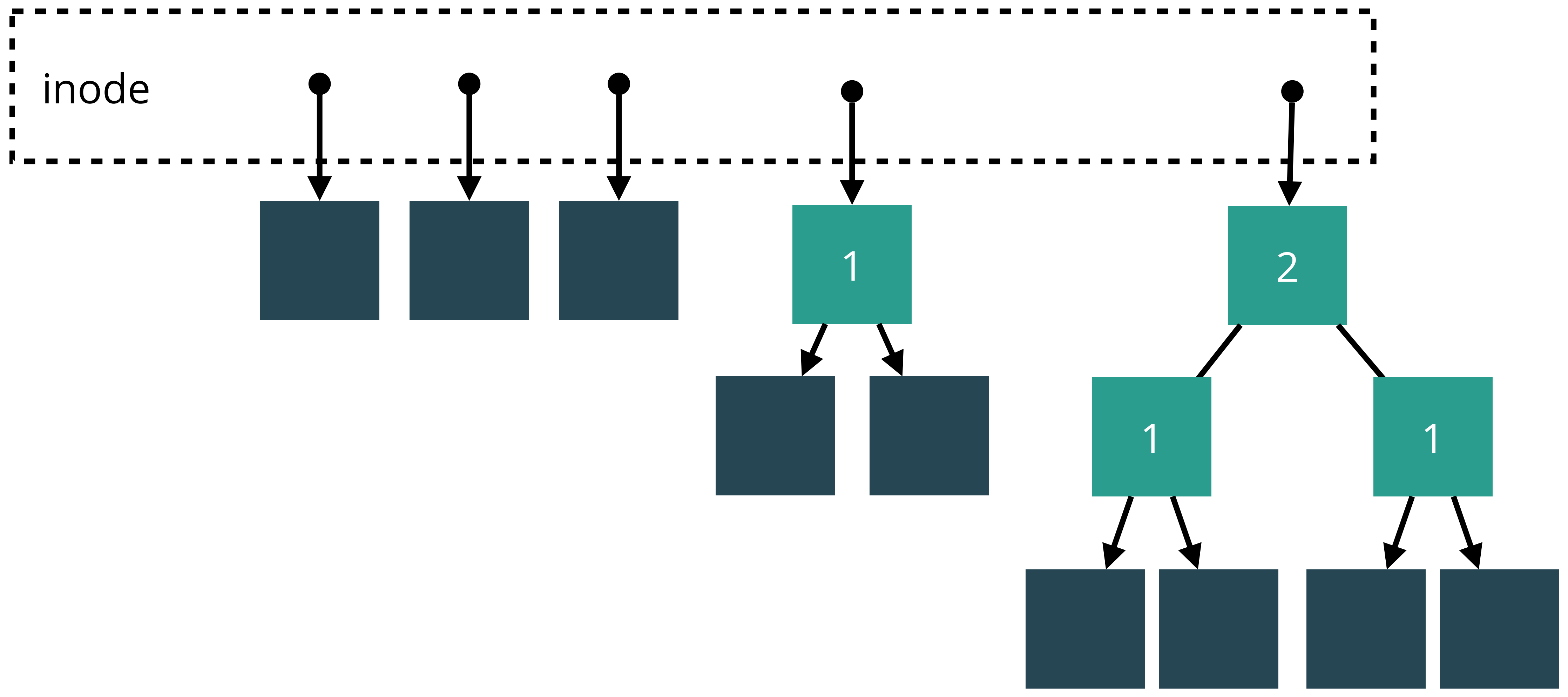
Invariant must ensures all pointers are distinct



Invariant must ensures all pointers are distinct

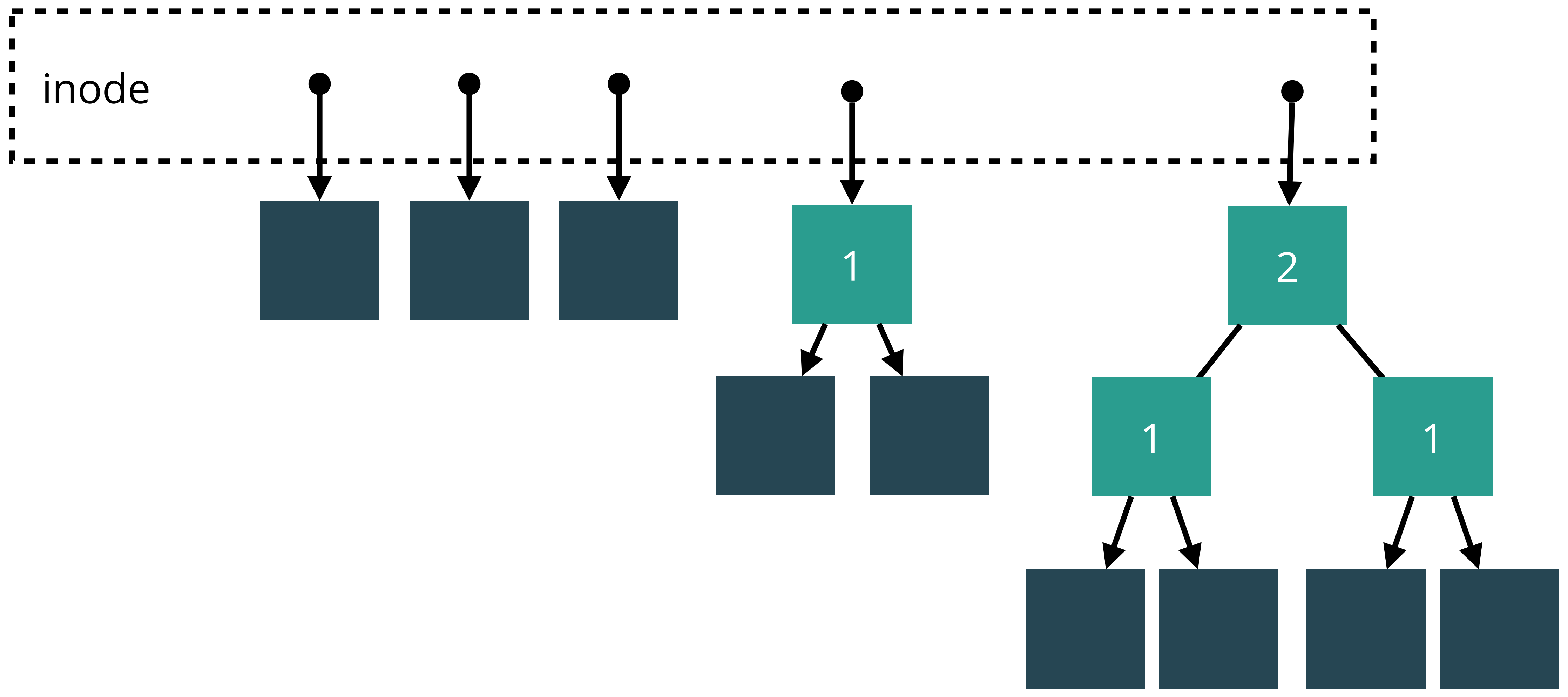


Invariant must ensures all pointers are distinct



DFSCQ: separation logic ensures inodes are separate, metadata and data separate, and everything in between

Invariant must ensures all pointers are distinct



DFSCQ: separation logic ensures inodes are separate, metadata and data separate, and everything in between

DaisyNFS: maintain **partial bijection** between positions in tree and block numbers

Freeing space is another interesting challenge

Problem: freeing 100GB file requires a lot of writes, might not fit in one transaction

Solution: freeing is a background task we prove is **(logically) a no-op**

```
method zeroFreeSpace(txn: Txn, ino: Ino, sz_hint: uint64)
  returns (done: bool)
  modifies Repr
  requires fs.has_jrnl(txn)
  requires Valid() ensures Valid()
  ensures data == old(data)
```

Dafny makes verifying DaisyNFS reasonable

We're verifying a sequential, crash-free file system

Dafny is not perfect but the process was efficient

Limitations in DaisyNFS

Missing some NFS features

Adding in-memory caching is difficult

Use unverified Linux client

Implementation overview

1,600 lines GoJournal

github.com/mit-pdos/go-journal

Implementation overview

1,600 lines GoJournal

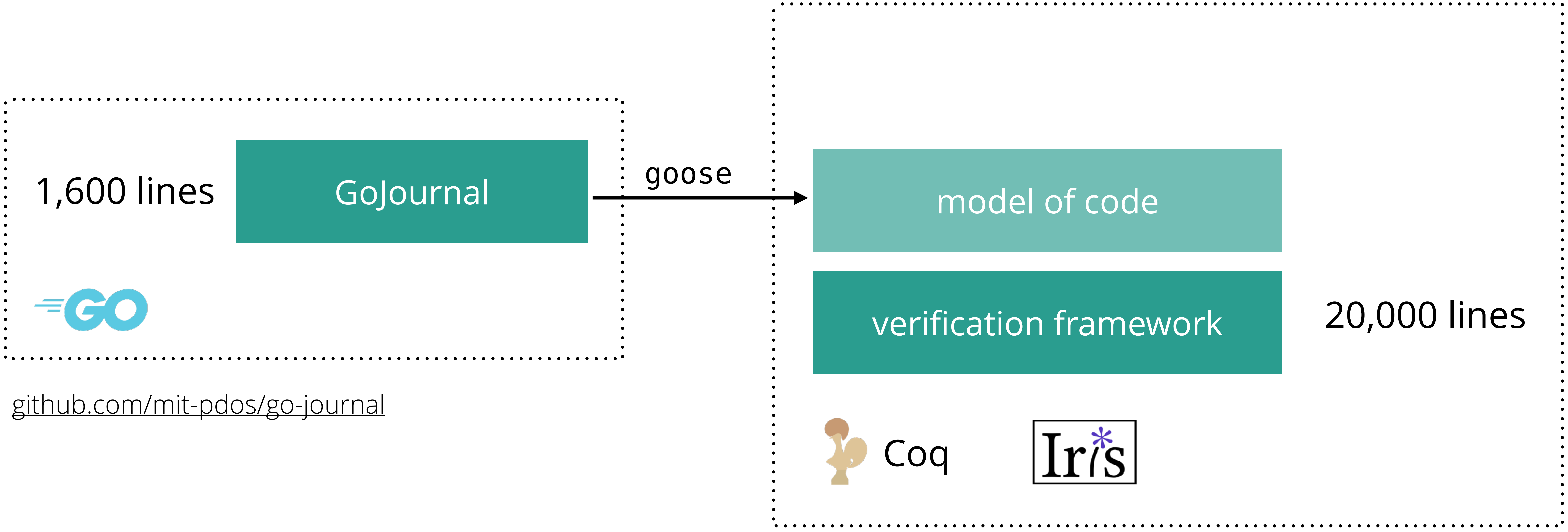
GO

github.com/mit-pdos/go-journal

verification framework 20,000 lines

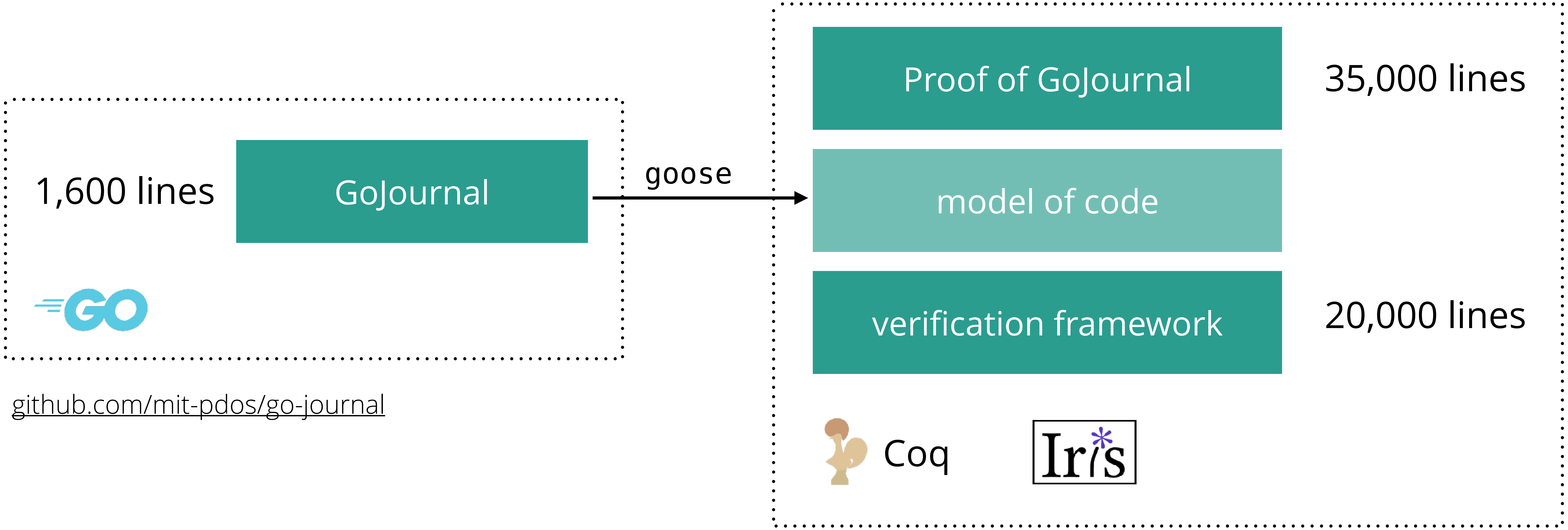
Coq Iris

Implementation overview



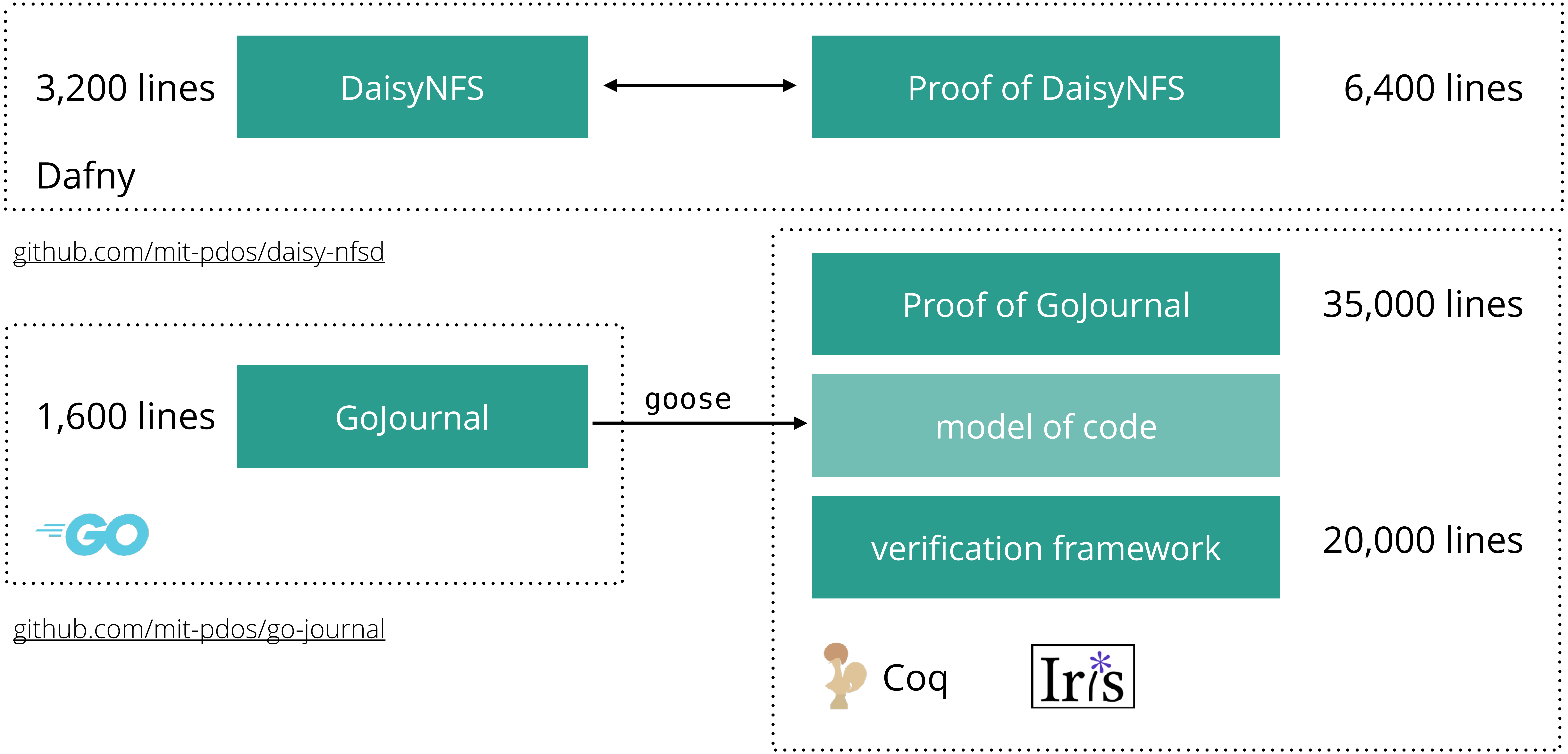
github.com/mit-pdos/go-journal

Implementation overview



github.com/mit-pdos/go-journal

Implementation overview



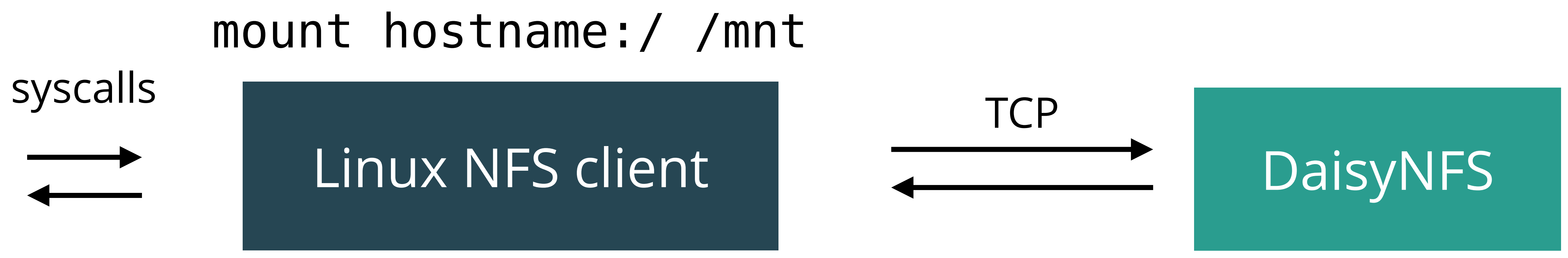
Evaluating DaisyNFS's performance

Compare against Linux kernel NFS server exporting ext4 (with `data=journal` mode for fair comparison)

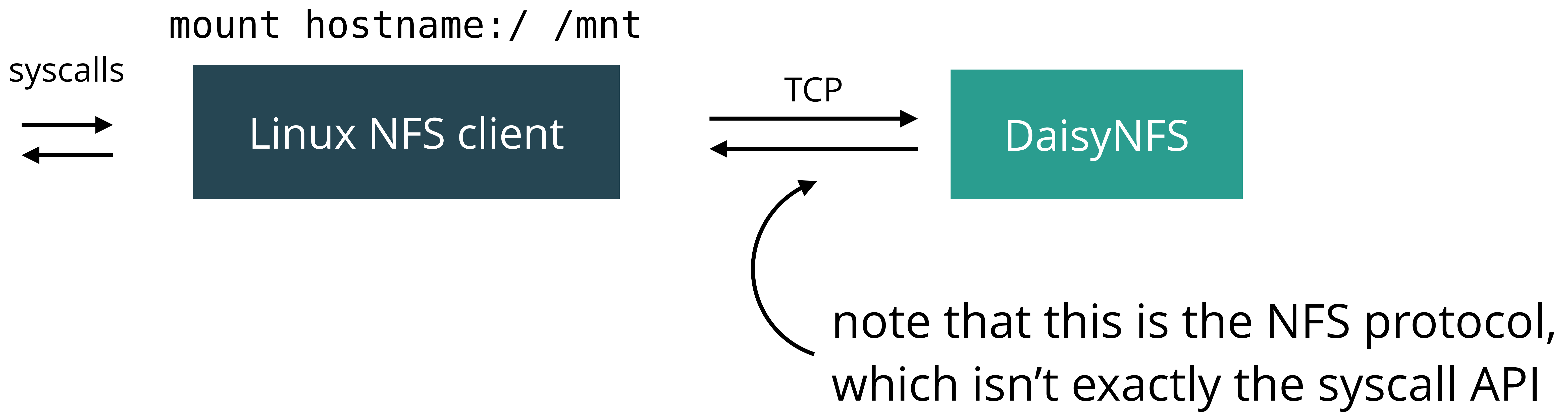
Mount NFS server using Linux NFS client

Using an NFS server

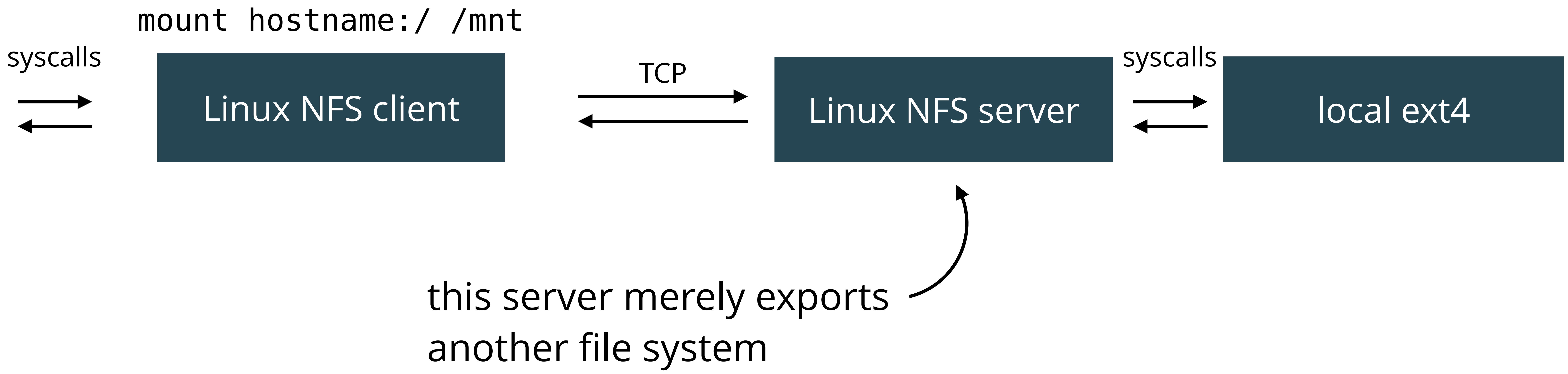
Using an NFS server



Using an NFS server



Using the Linux NFS server



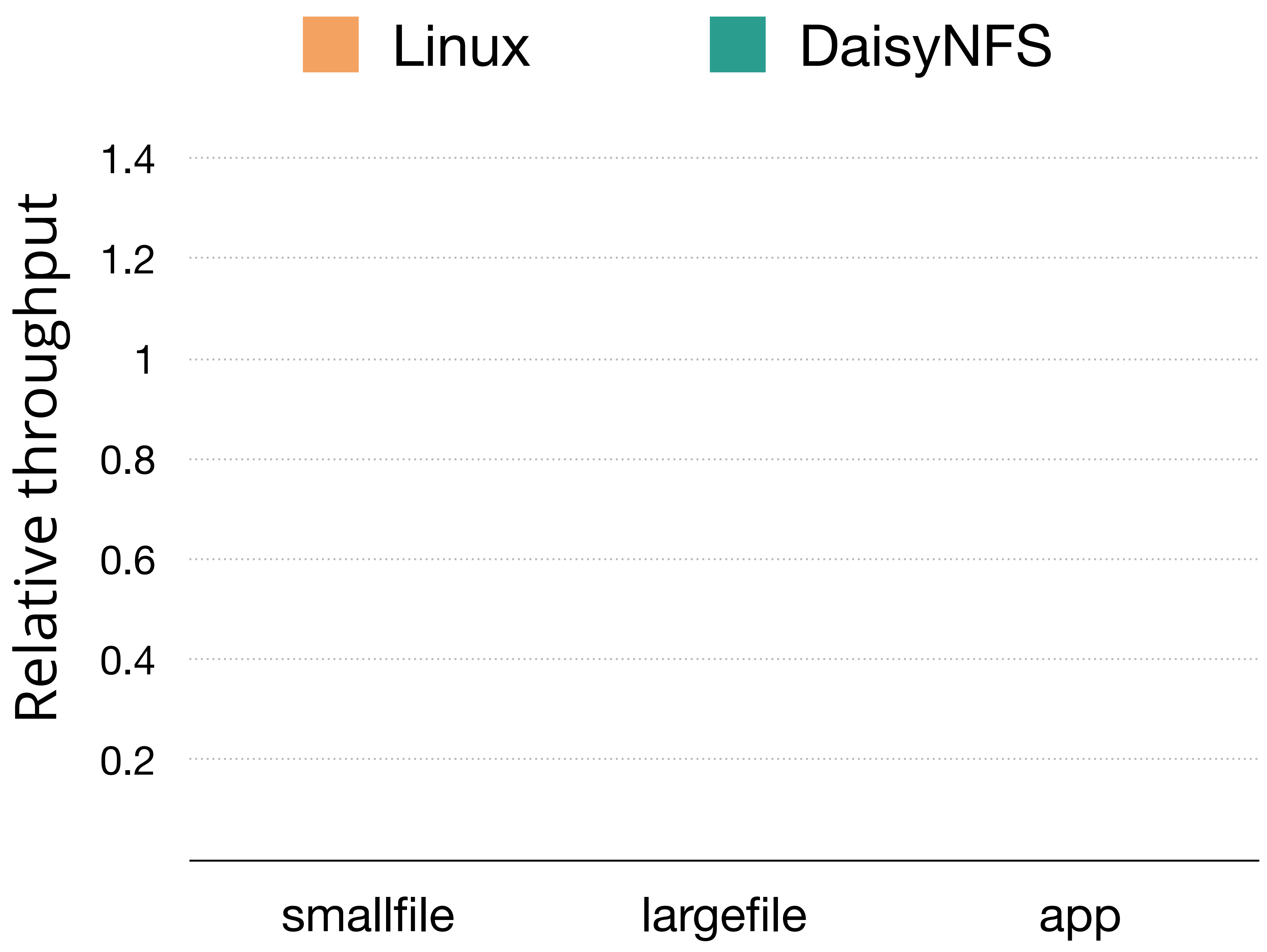
Experimental setup

Hardware: i3.metal instance

36 cores at 2.3GHz, NVMe SSD

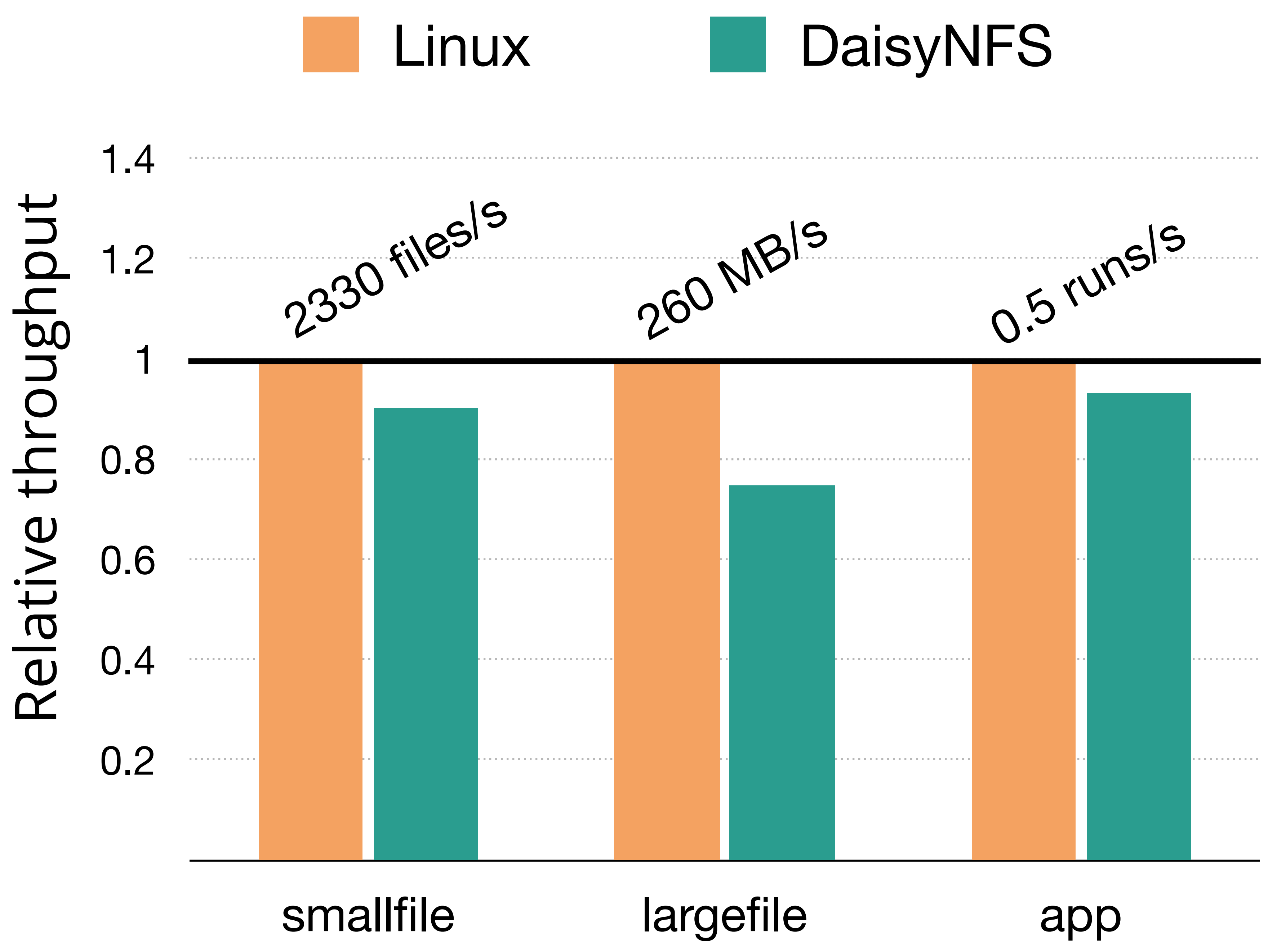
Benchmarks:

- smallfile: metadata heavy
- largefile: lots of data
- app: `git clone` + `make`

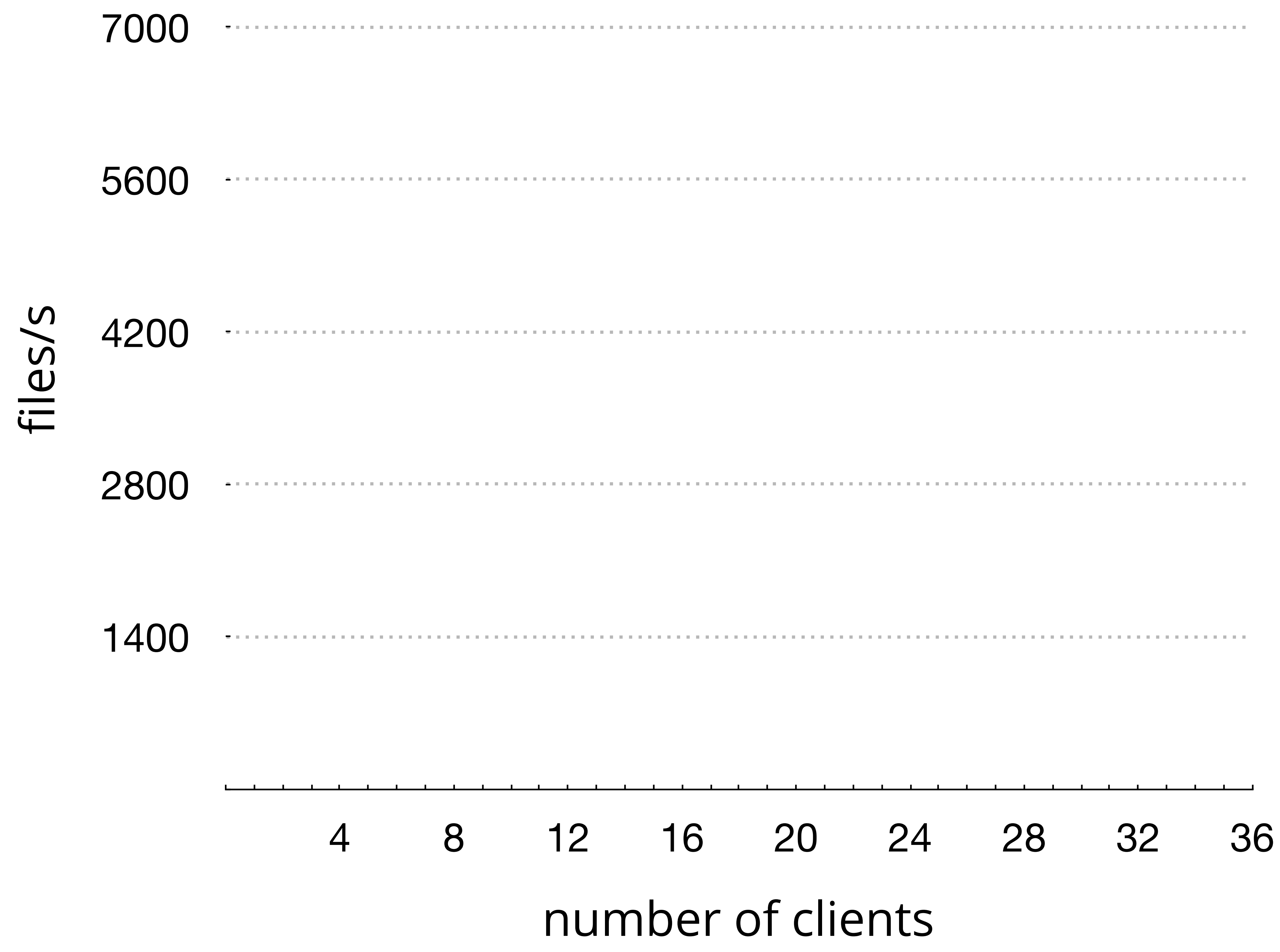


Compare DaisyNFS throughput to Linux, running on an in-memory disk

DaisyNFS gets comparable performance even on a single thread

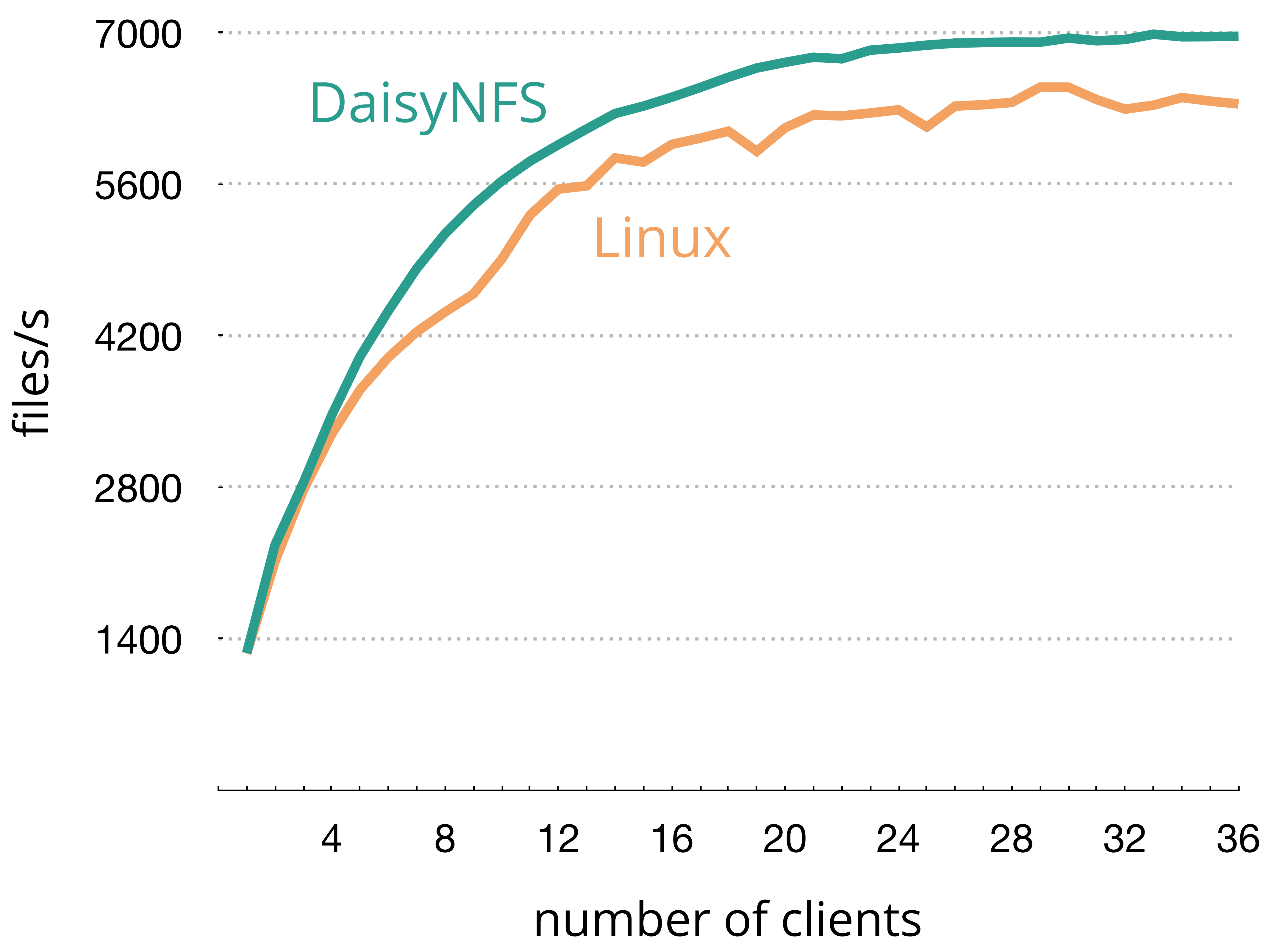


Compare DaisyNFS throughput to Linux, running on an in-memory disk



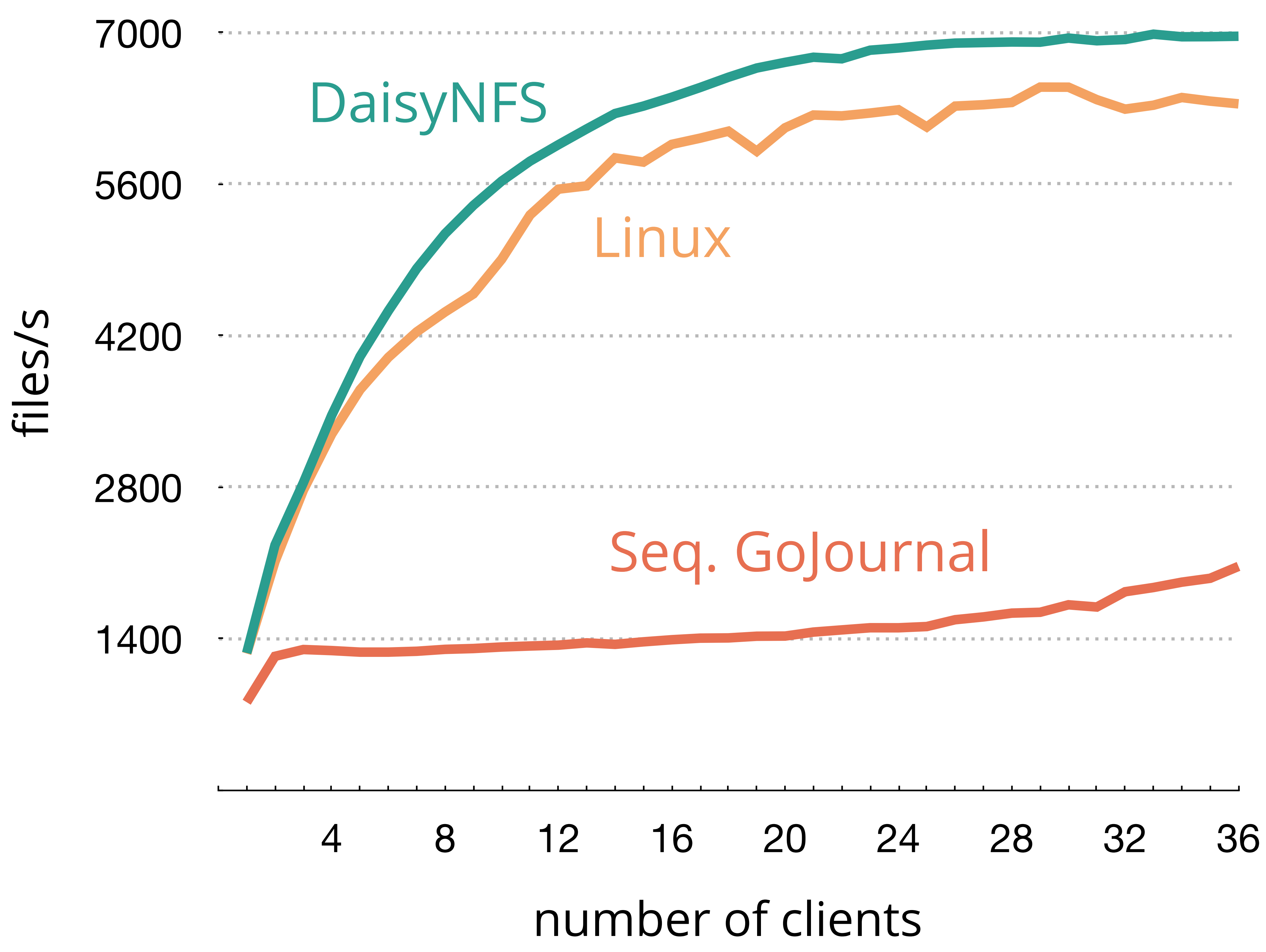
Run smallfile with many clients on an NVMe SSD

GoJournal can take advantage of multiple clients



Run smallfile with many clients on an NVMe SSD

Concurrency in the journal matters



Seq. GoJournal is DaisyNFS but with locks around tricky concurrent parts of WAL

Summary

DaisyNFS is a verified, concurrent, crash-safe file system

Built on top of GoJournal, a verified transaction system

Verification strategy combines Perennial and Dafny

DaisyNFS gets good performance