Verifying concurrent, crash-safe systems with **Perennial**

Tej Chajed, Joseph Tassarotti*, Frans Kaashoek, Nickolai Zeldovich

MIT and *Boston College

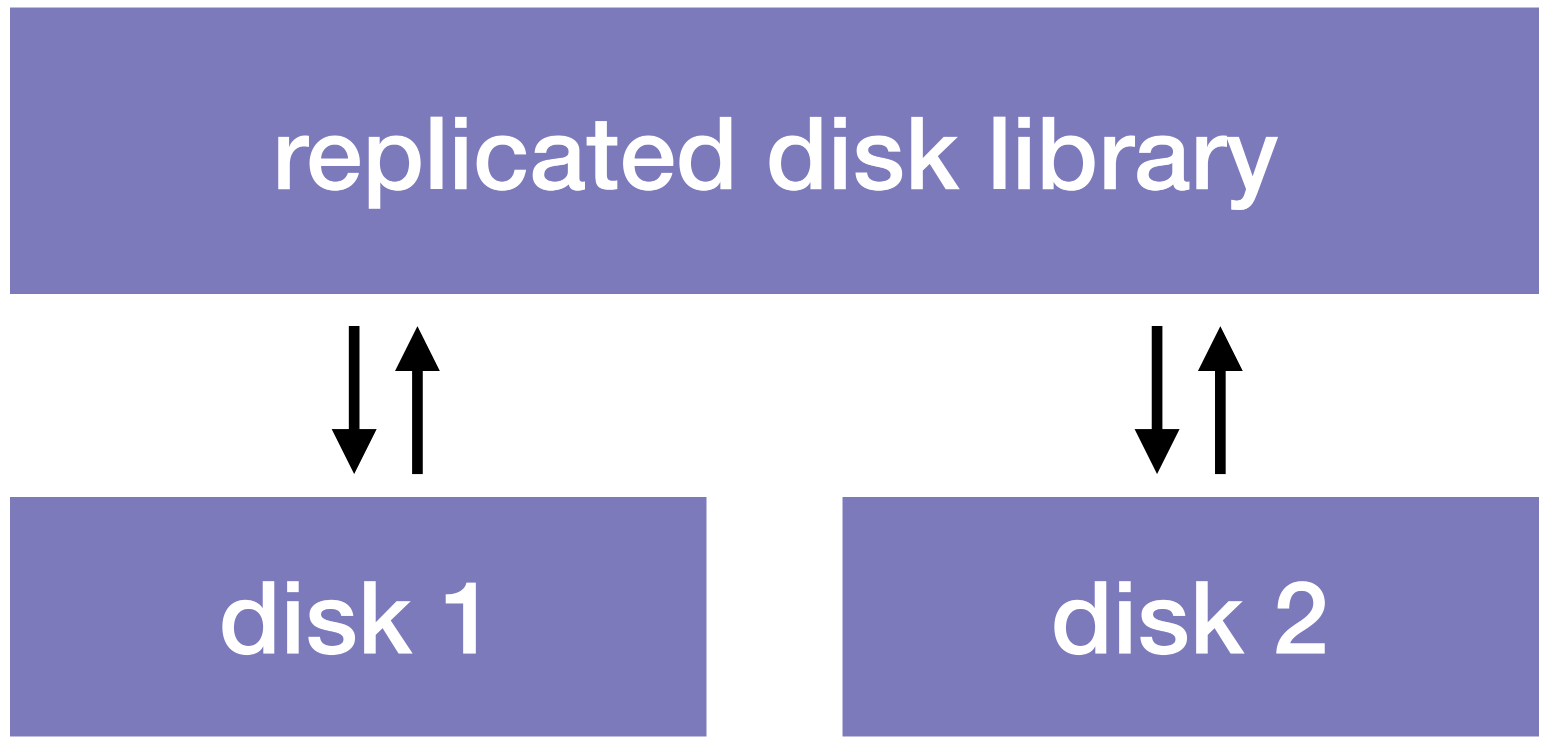
Many systems need concurrency and crash safety

Examples: file systems, databases, and key-value stores

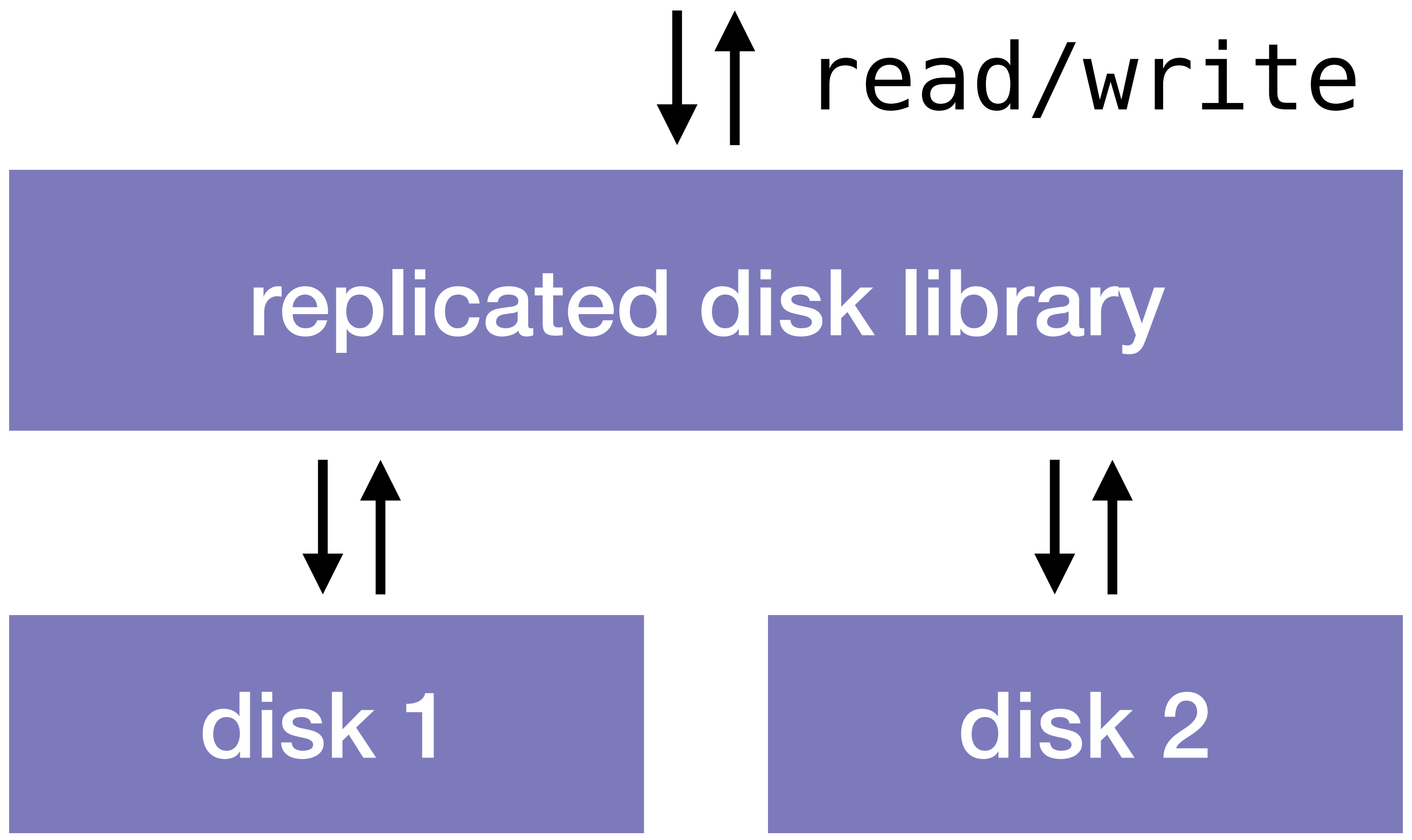
Make strong guarantees about keeping your data safe

Achieve high performance with concurrency

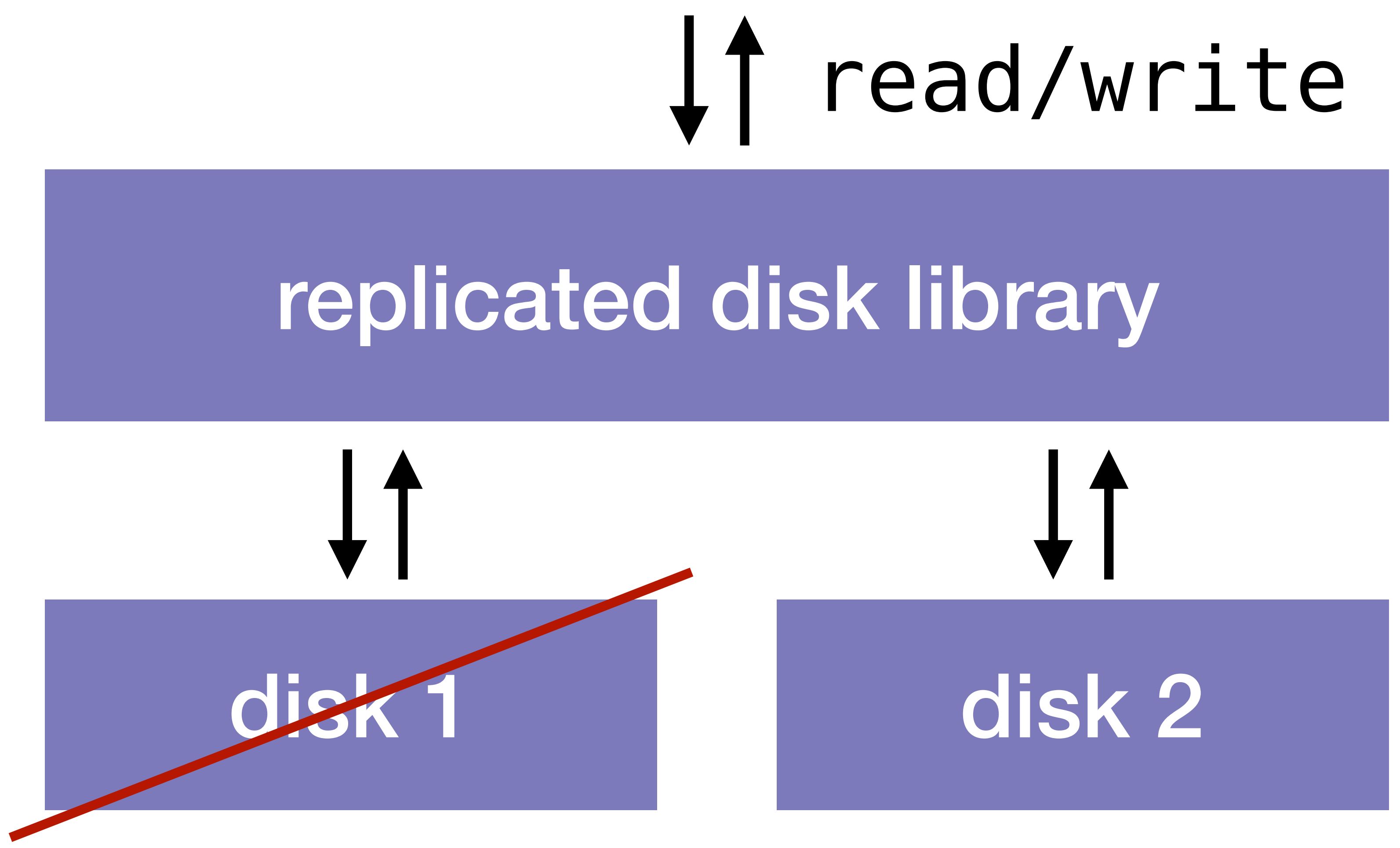
Simple example: replicated disk



Simple example: replicated disk



Simple example: replicated disk



Replicated disk is subtle

```
func write(a: addr, v: block) {
  lock_address(a)
  d1.write(a, v)
  d2.write(a, v)
  unlock_address(a)
}
```

Replicated disk is subtle

```
func write(a: addr, v: block) {
  lock_address(a)
  d1.write(a, v)
  d2.write(a, v)
  unlock_address(a)
}
```

what if system crashes here?

what if disk 1 fails?

Replicated disk is subtle

```
func write(a: addr, v: block) {
  lock_address(a)
  d1.write(a, v)
  d2.write(a, v)
  unlock_address(a)
}
```

what if system crashes here?
what if disk 1 fails?

```
// runs on reboot
func recover() {
  for a in … {
    // copy from d1 to d2
  }
}
```

Replicated disk is subtle

```
func write(a: addr, v: block) {
  lock_address(a)
  d1.write(a, v)
  d2.write(a, v)
  unlock_address(a)
}
```

what if system crashes here?

what if disk 1 fails?

```
// runs on reboot
func recover() {
  for a in … {
    // copy from d1 to d2
  }
}
```

```
func read(a: addr): block {
  lock_address(a)
  v, ok := d1.read(a)
  if !ok {
    v, _ = d2.read(a)
  }
  unlock_address(a)
  return v
}
```

Goal: systematically reason about all executions with formal verification

Existing verification frameworks do not support concurrency and crash safety

verified crash safety
FSCQ [SOSP ’15]
Yggdrasil [OSDI ’16]
DFSCQ [SOSP ’17]
…

verified concurrency
CertiKOS [OSDI ’16]
CSPEC [OSDI ’18]
AtomFS [SOSP ’19]
…

no system can do both

Combining verified crash safety and concurrency is challenging

Crash and recovery can interrupt a critical section

Crash wipes in-memory state

Recovery logically completes crashed threads’ operations

Perennial's techniques address challenges integrating crash safety into concurrency reasoning

Crash and recovery can interrupt a critical section

➡ **leases**

Crash wipes in-memory state

➡ **memory versioning**

Recovery logically completes crashed threads' operations

➡ **recovery helping**

Perennial's techniques address challenges integrating crash safety into concurrency reasoning

see paper

Crash and recovery can interrupt a critical section

➡ **leases**

this talk

Crash wipes in-memory state

➡ **memory versioning**

Recovery logically completes crashed threads' operations

➡ **recovery helping**

Contributions

Perennial: framework for reasoning about crashes and concurrency

see paper Goose: reasoning about Go implementations

Evaluation: verified mail server written in Go with Perennial

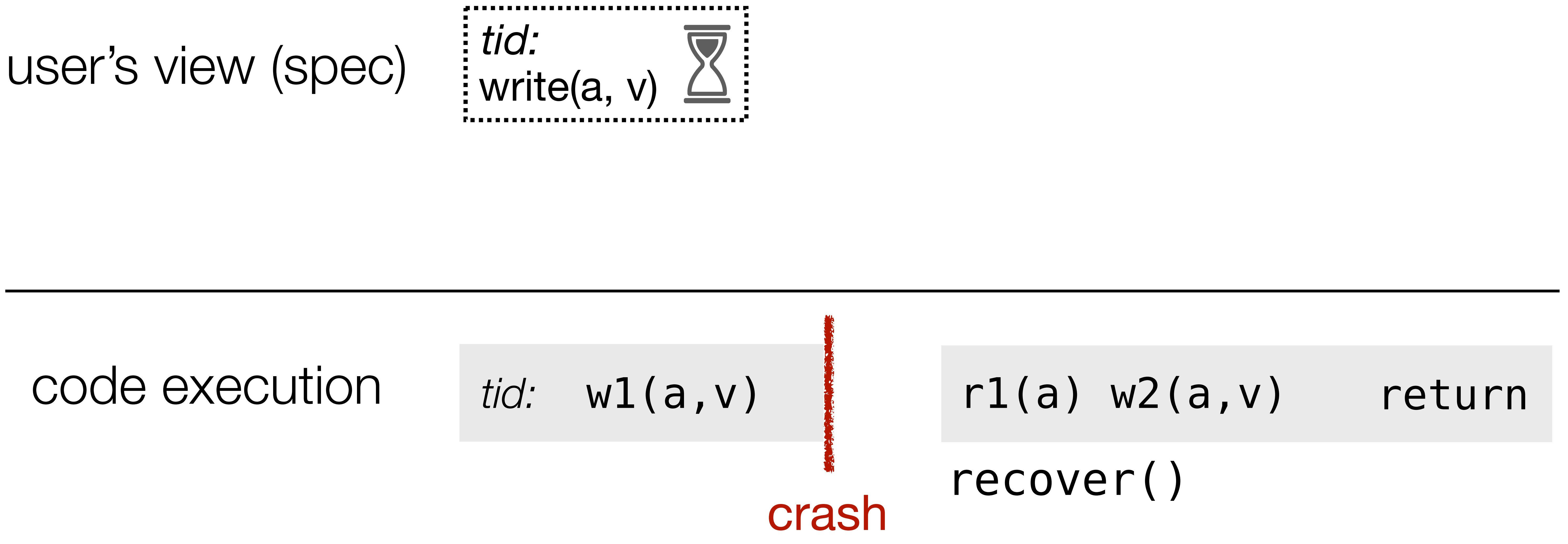
Specifying correctness: concurrent recovery refinement

All operations are **correct** and **atomic wrt concurrency and crashes**

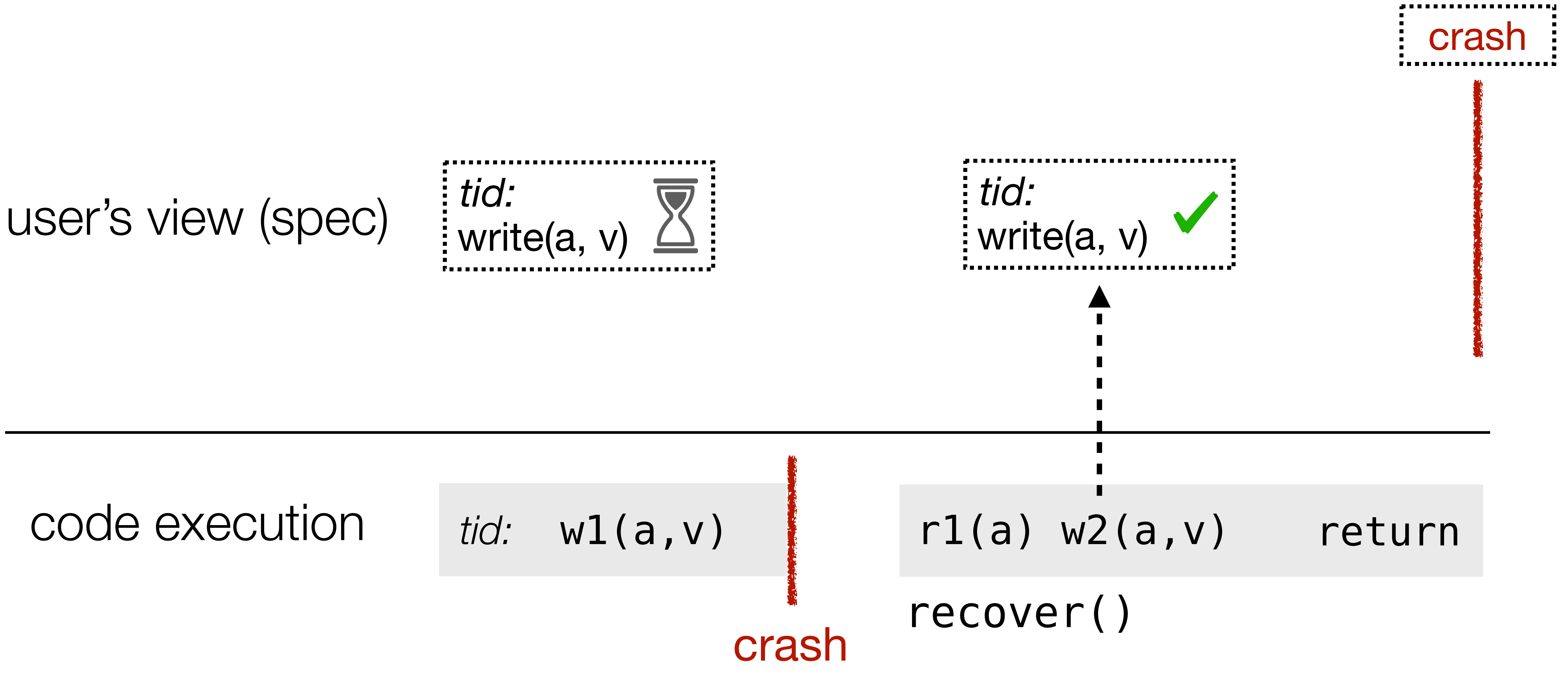
Recovery repairs system after reboot

Proving the replicated disk correct

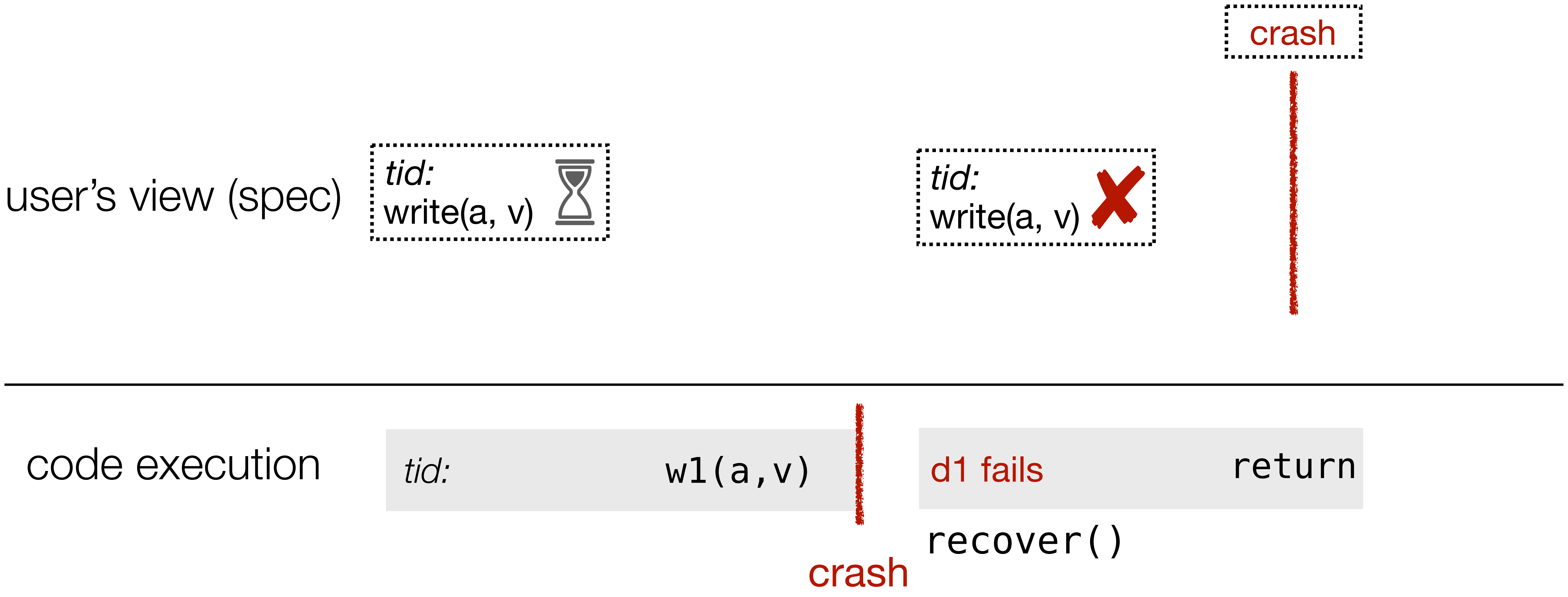
Recovery follows the specification when it completes



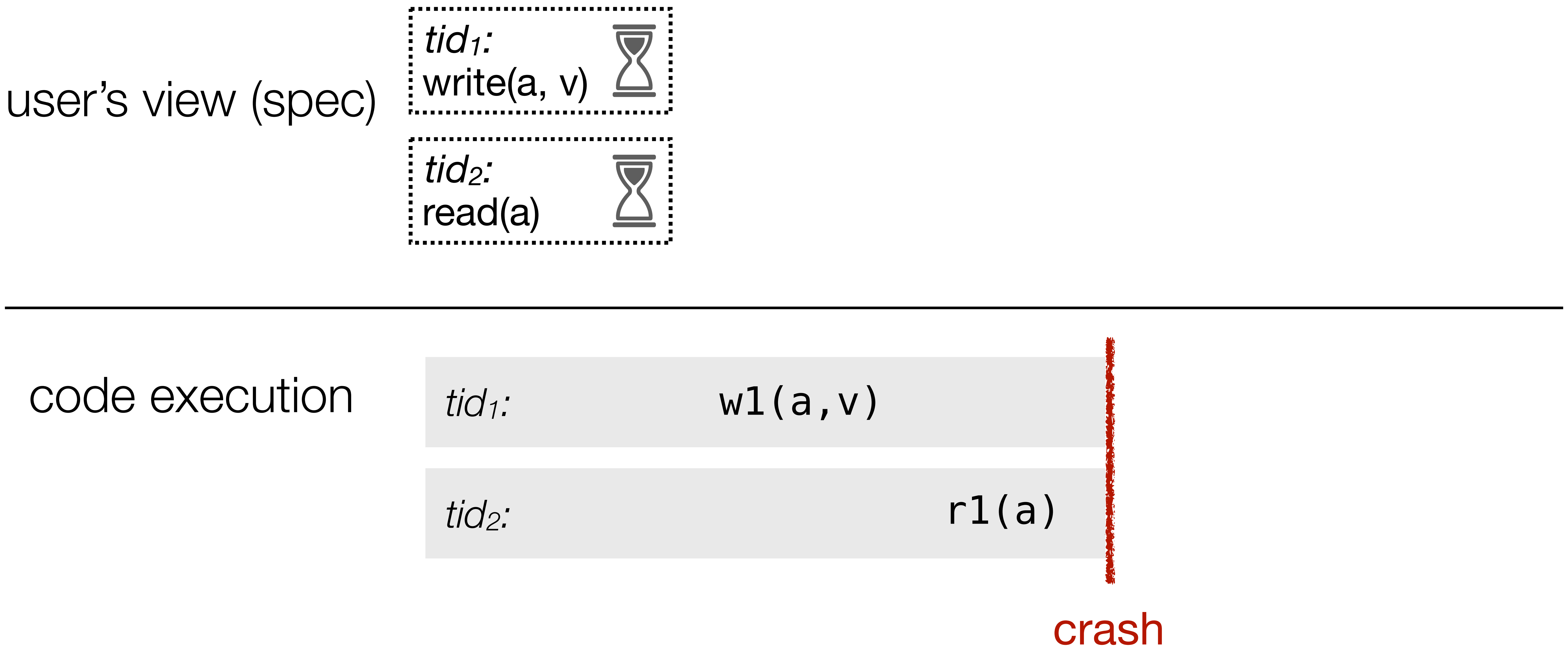
Recovery follows the specification when it completes



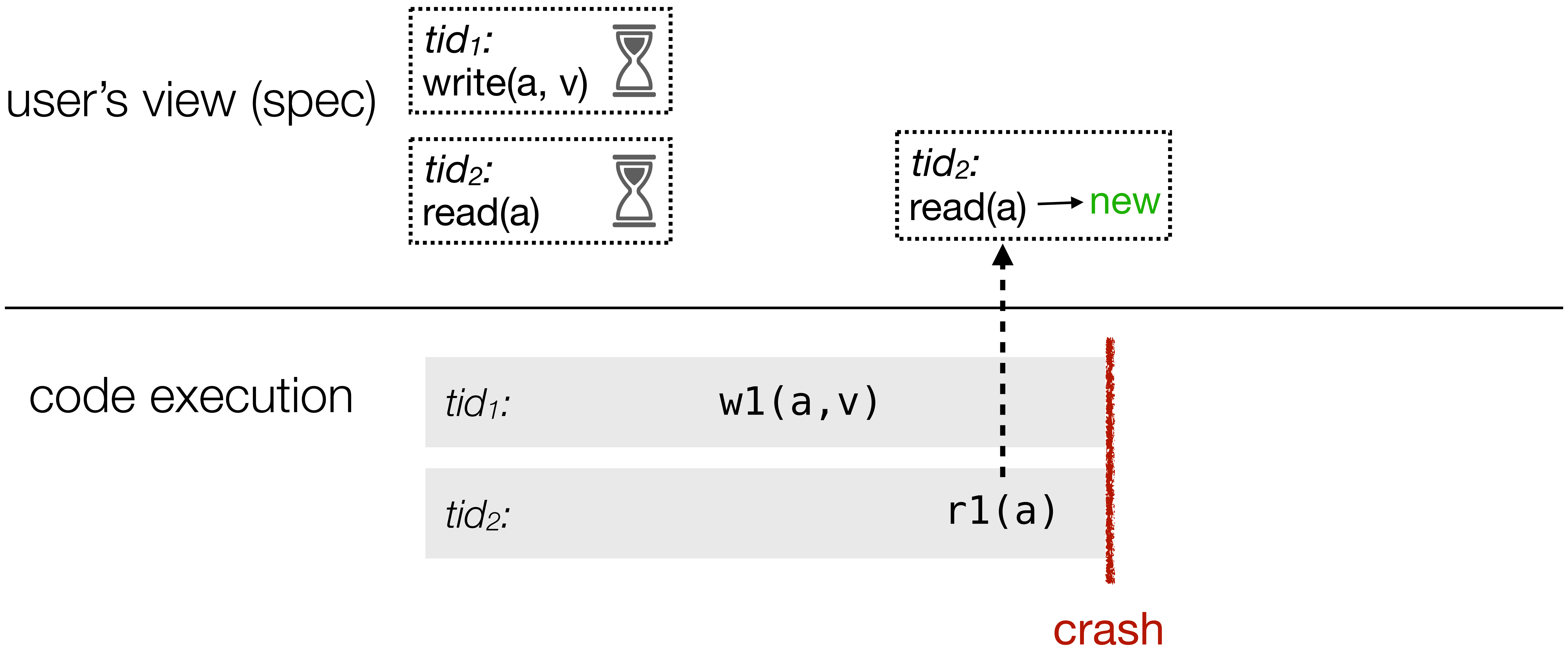
Crash can lose an interrupted write



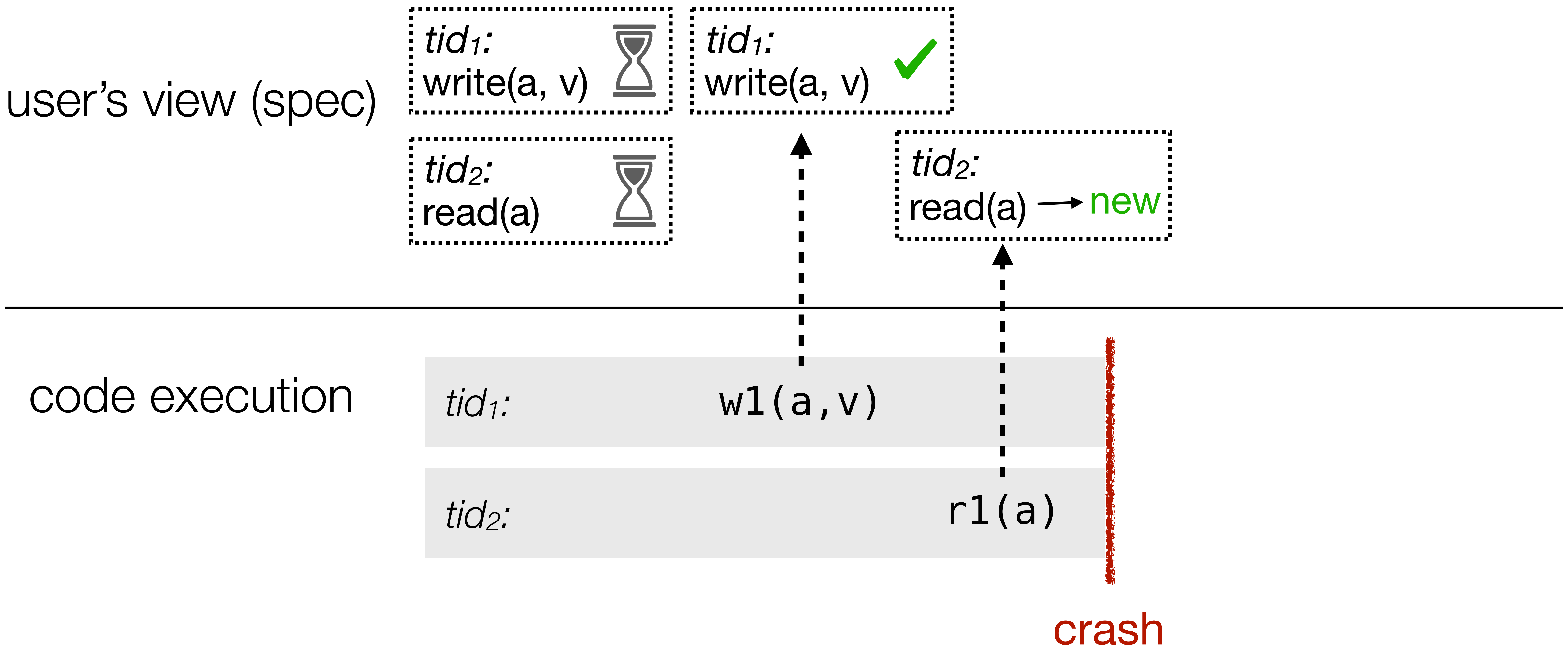
Not using locks leads to an illegal execution: read returns a still-volatile write



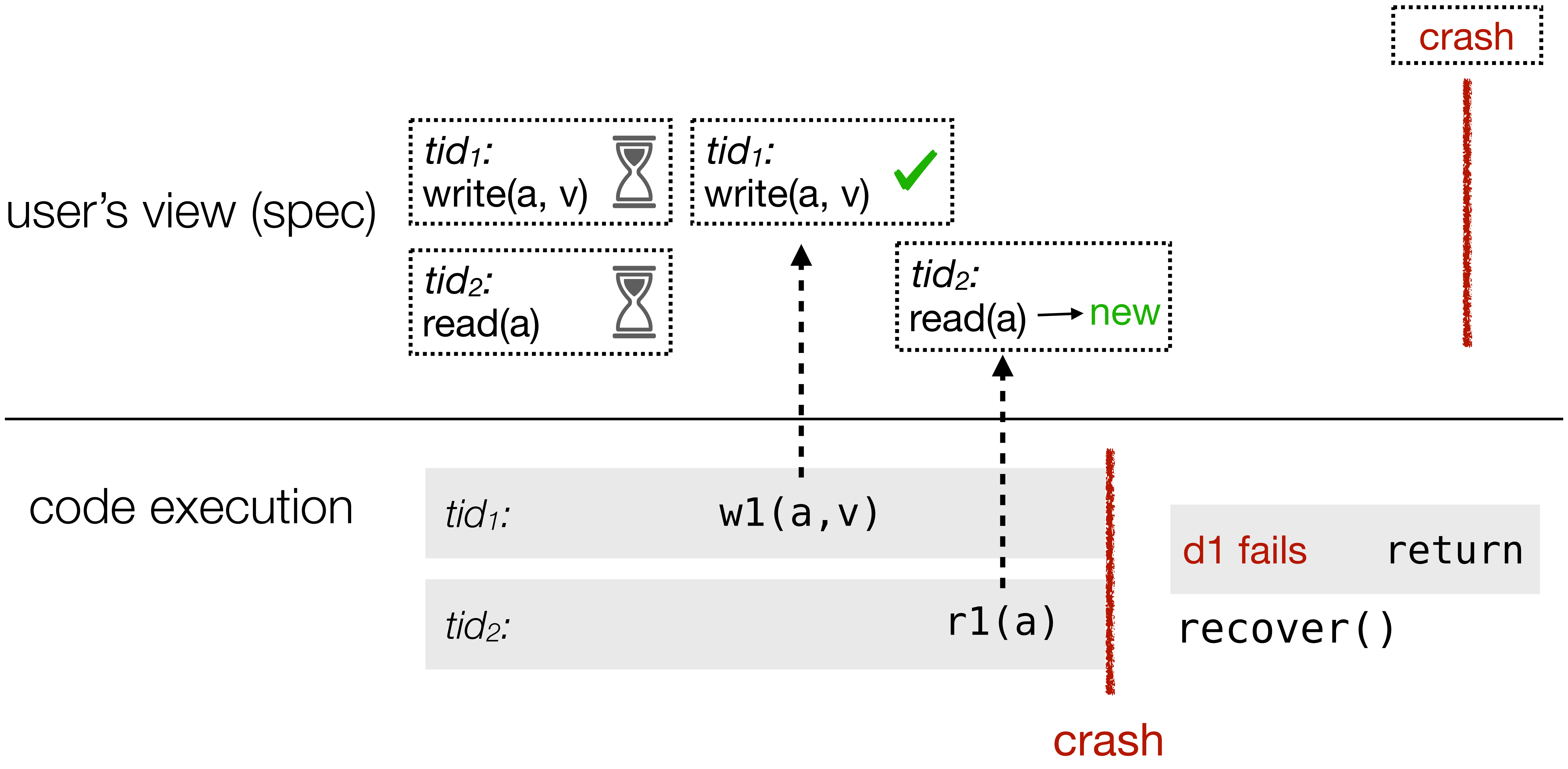
Not using locks leads to an illegal execution: read returns a still-volatile write



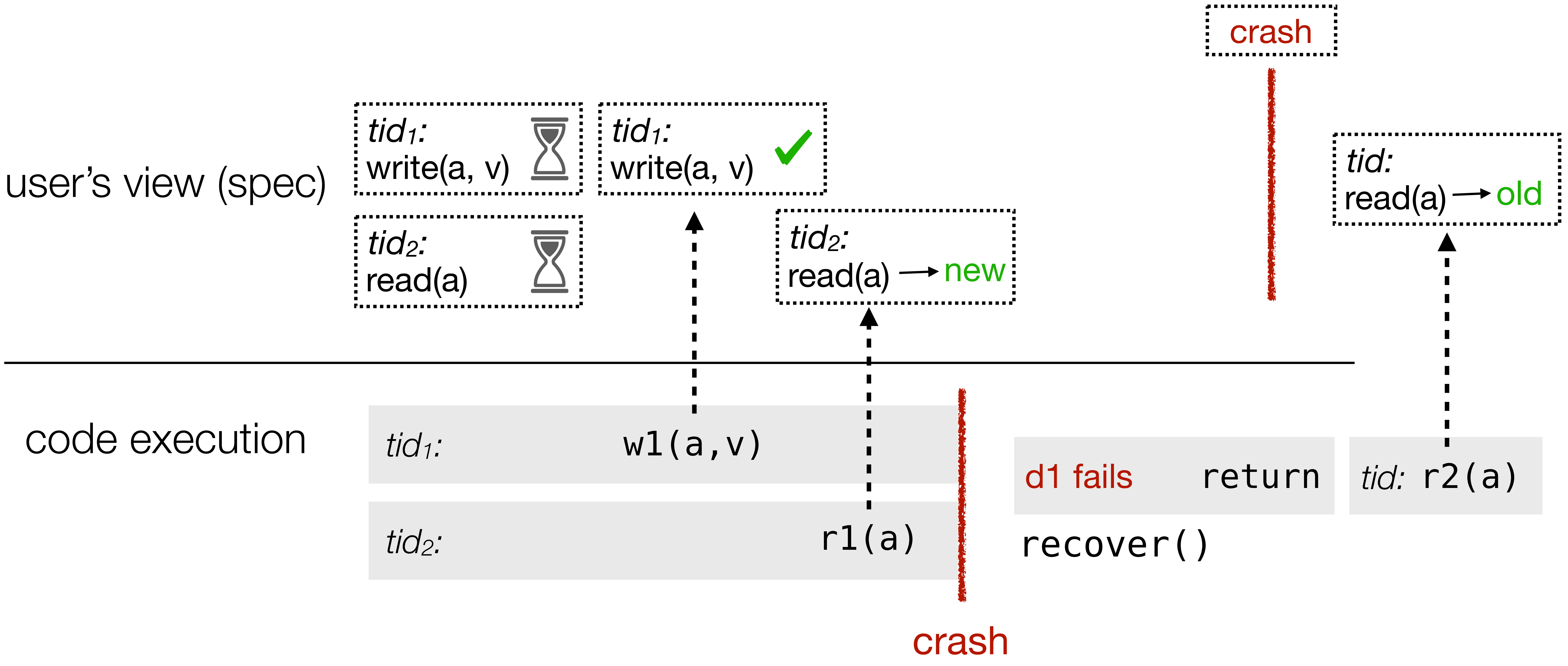
Not using locks leads to an illegal execution: read returns a still-volatile write



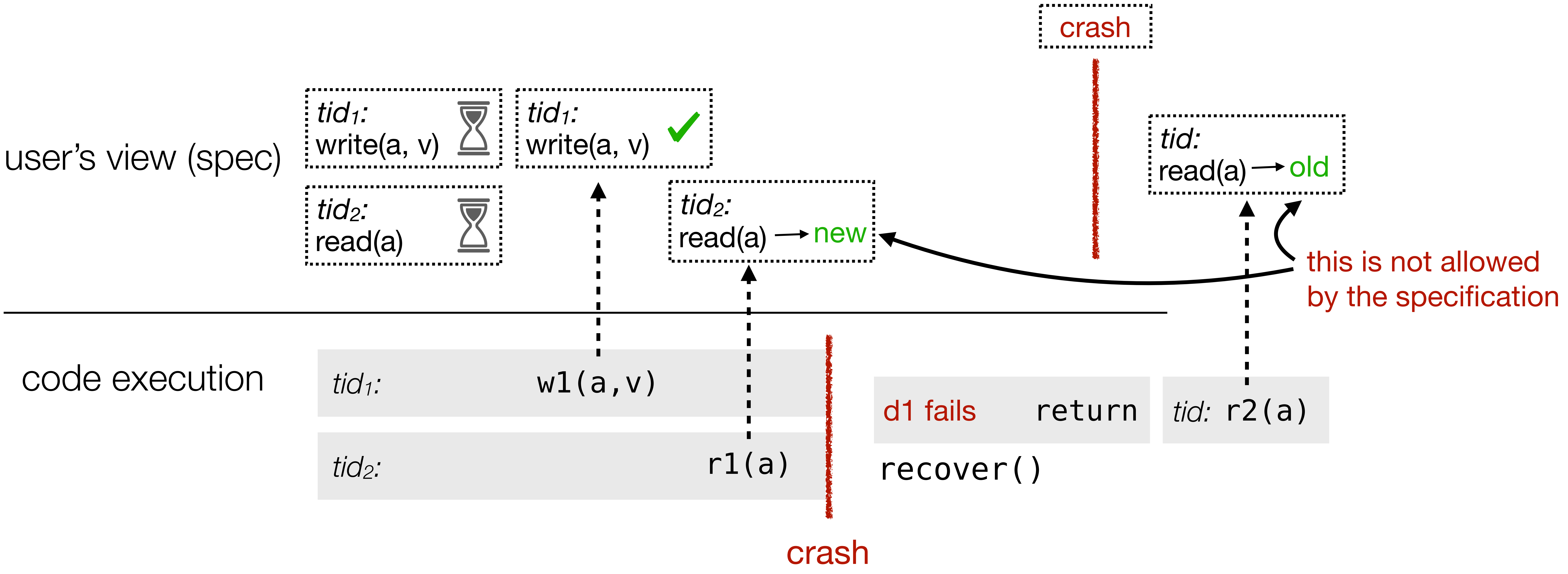
Not using locks leads to an illegal execution: read returns a still-volatile write



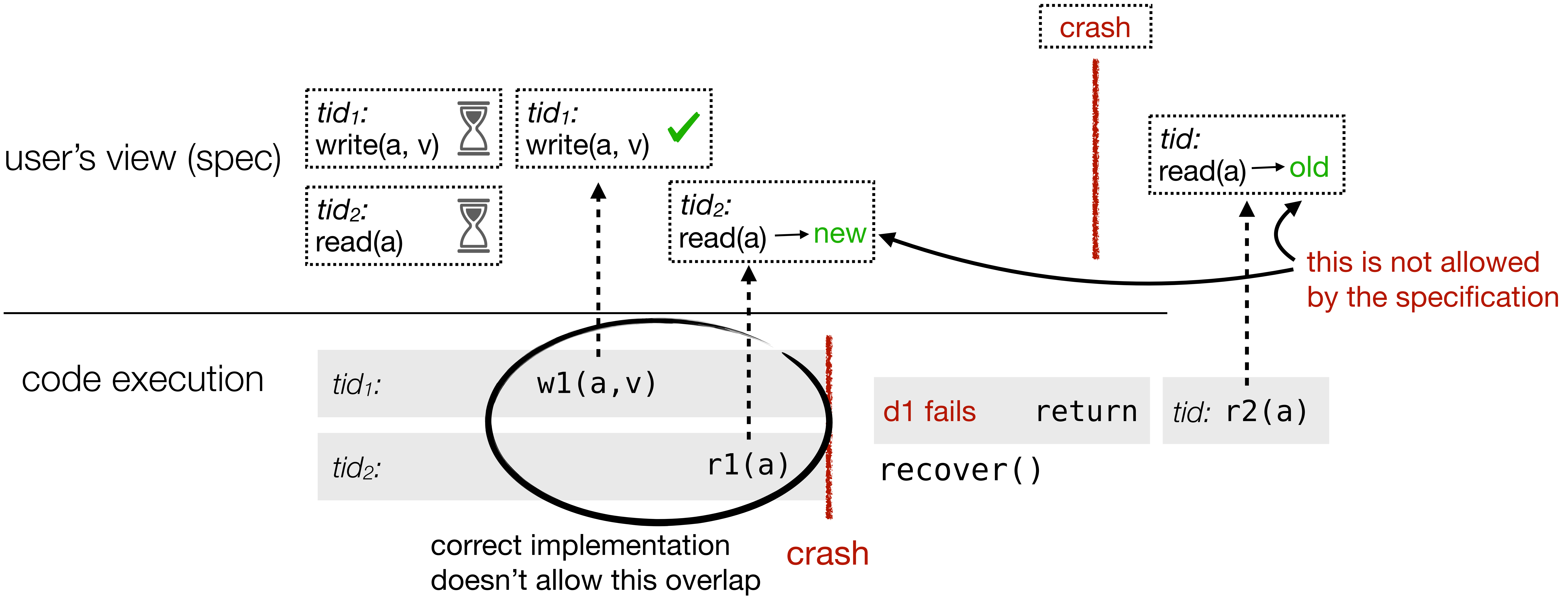
Not using locks leads to an illegal execution: read returns a still-volatile write



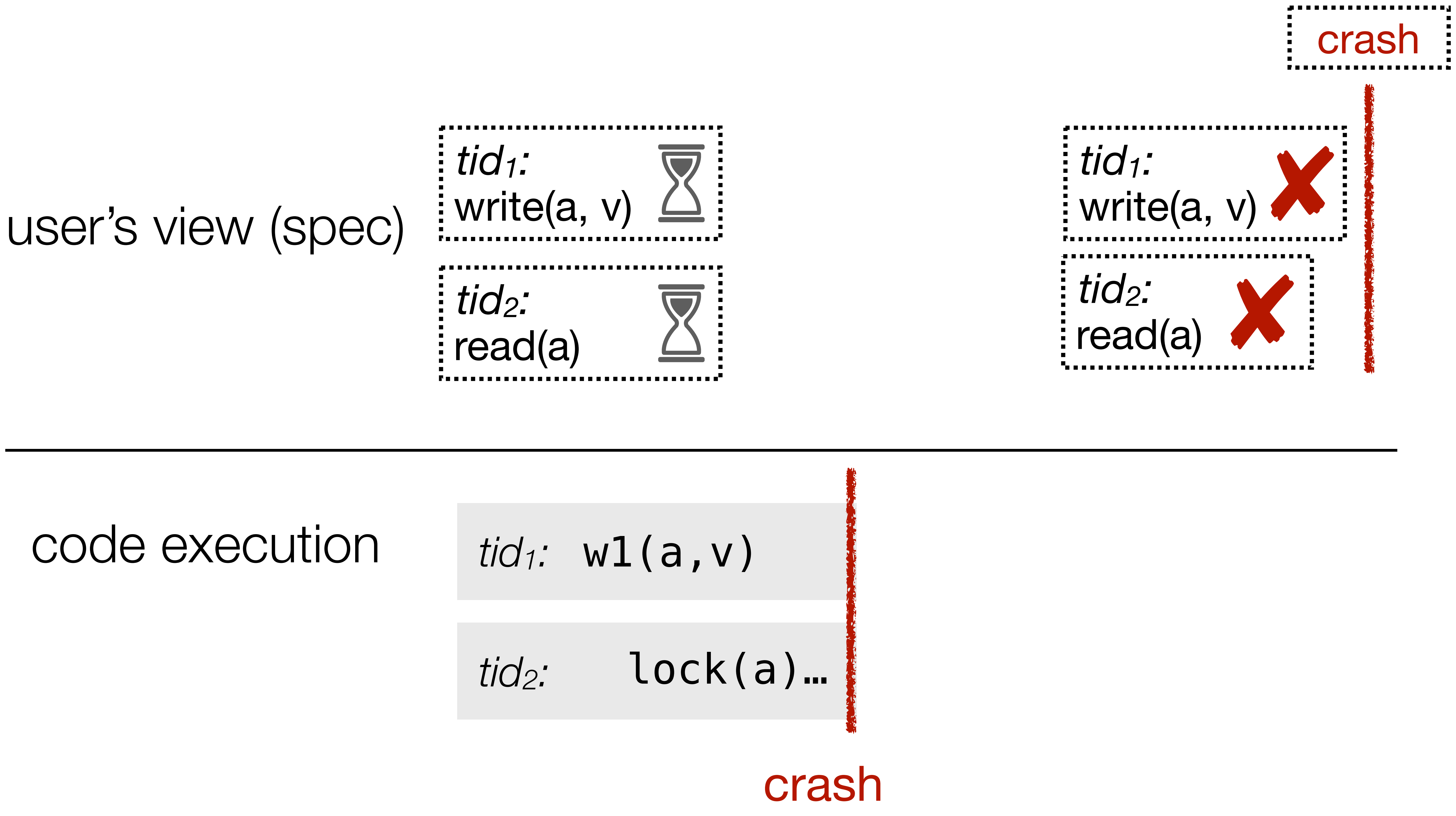
Not using locks leads to an illegal execution: read returns a still-volatile write



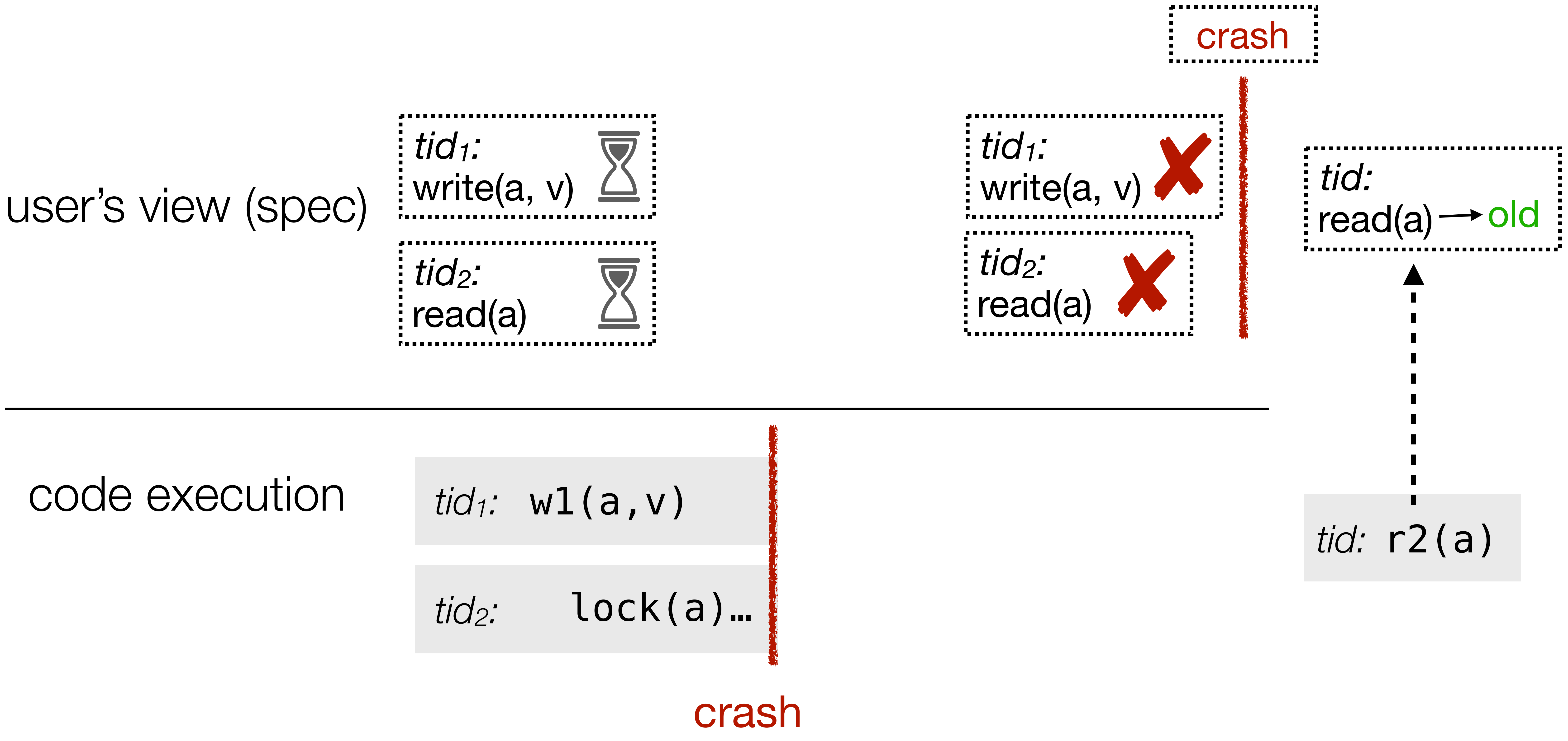
Not using locks leads to an illegal execution: read returns a still-volatile write



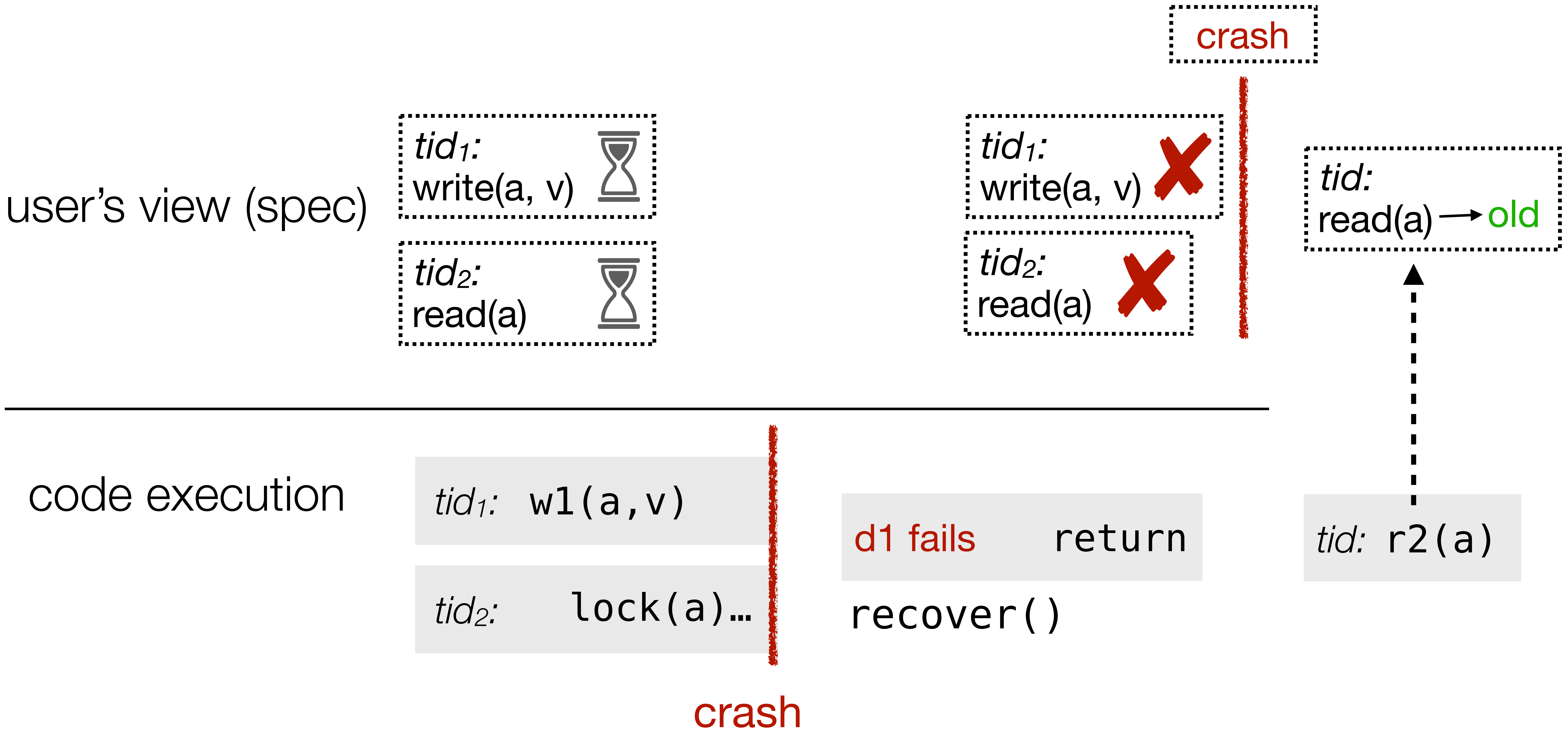
Reads use locks to only observe durable writes



Reads use locks to only observe durable writes

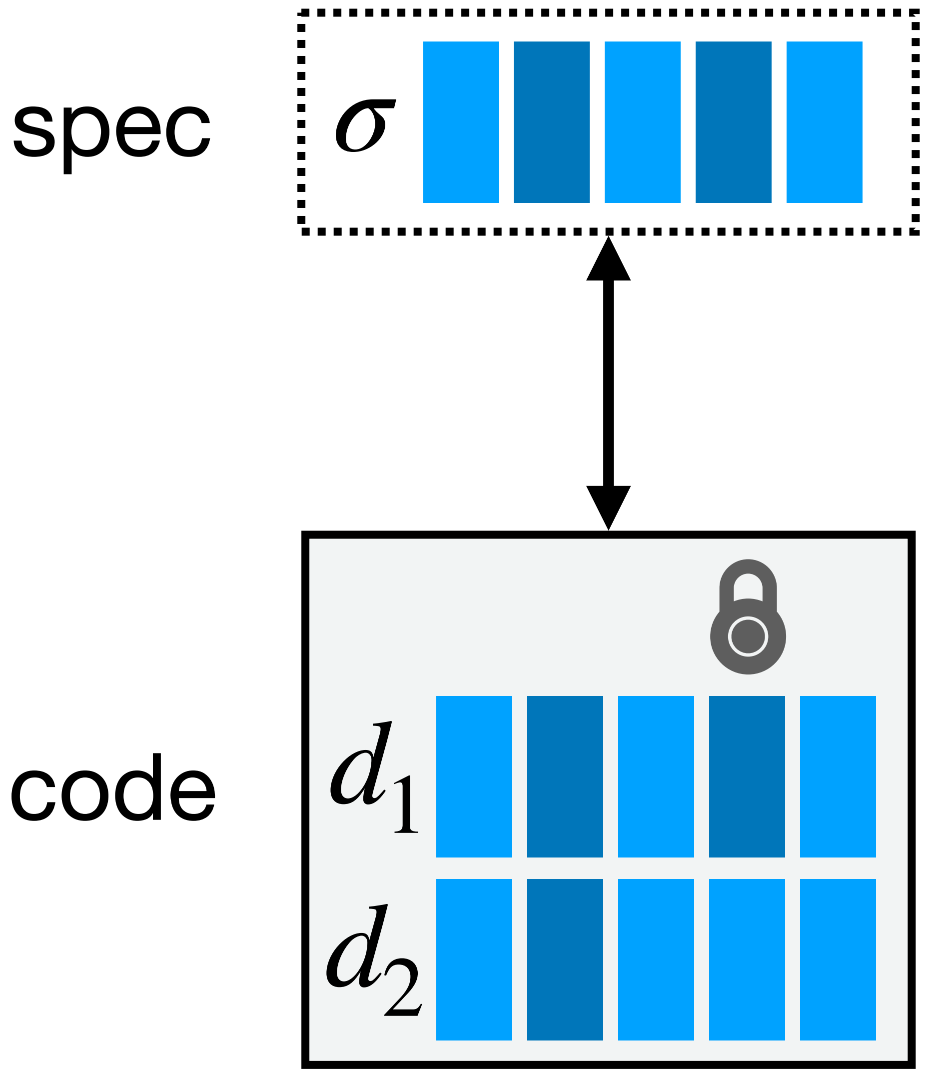


Reads use locks to only observe durable writes



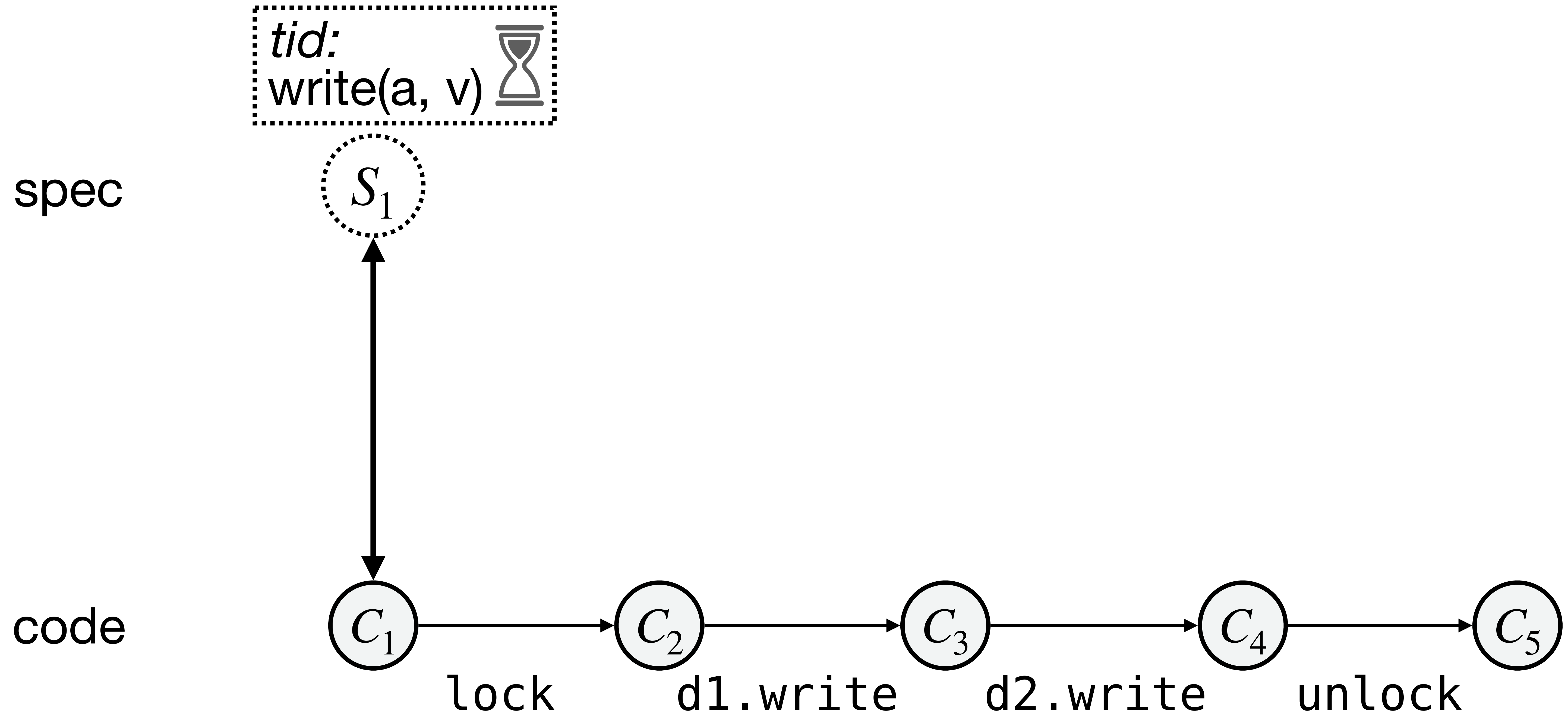
Proving refinement with forward simulation: relate code and spec states

Background



Proving refinement with forward simulation: prove every operation has a commit point

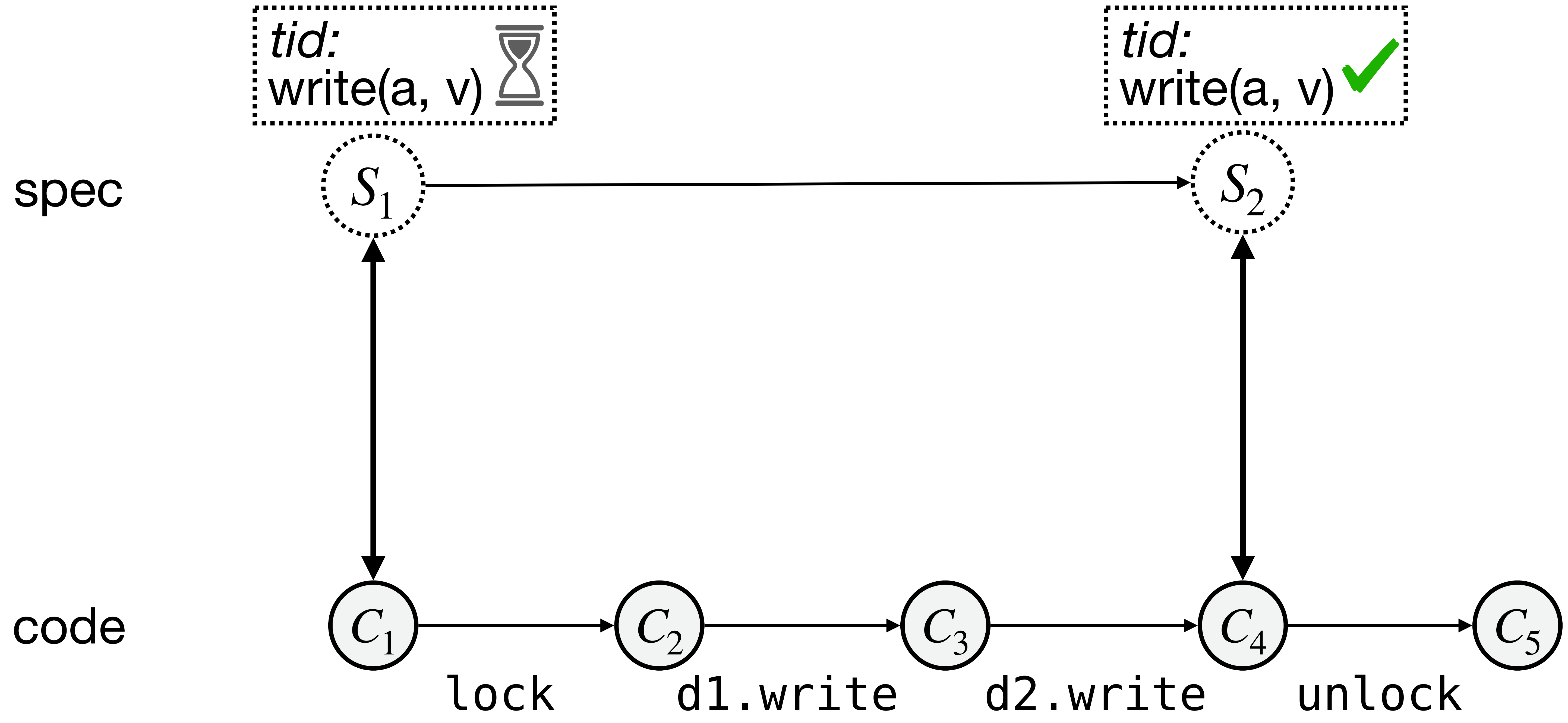
Background



1. Write down abstraction relation between code and spec states

Proving refinement with forward simulation: prove every operation has a commit point

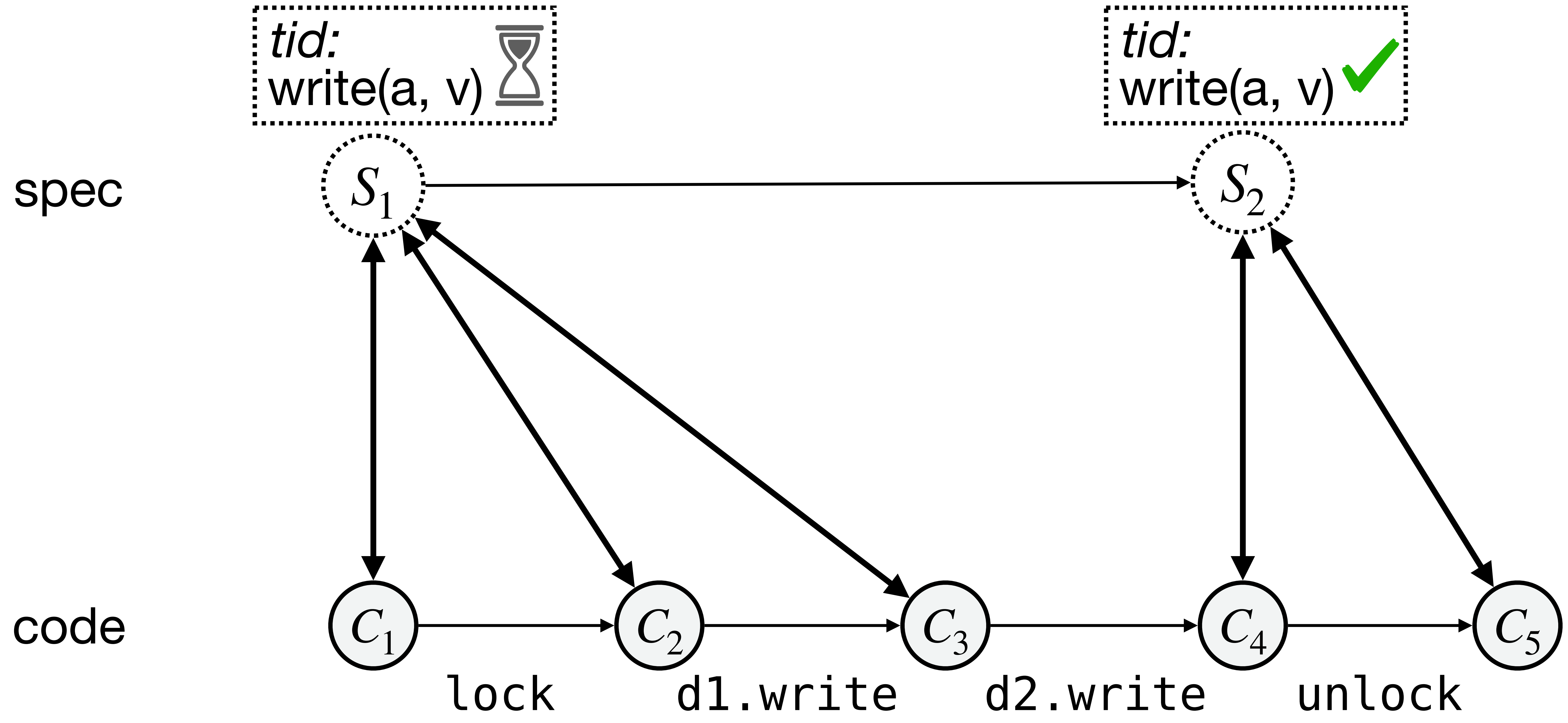
Background



1. Write down abstraction relation between code and spec states
2. Prove every operation commits

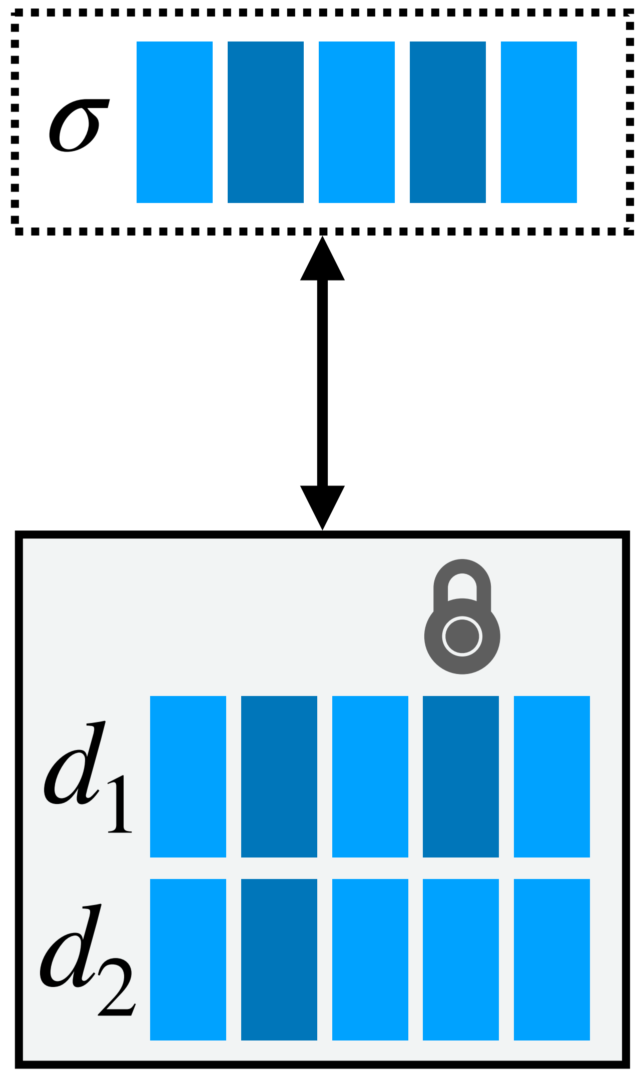
Proving refinement with forward simulation: prove every operation has a commit point

Background



1. Write down abstraction relation between code and spec states
2. Prove every operation commits
3. Prove abstraction relation is preserved

Abstraction relation for the replicated disk



abstraction relation:

$$\textbf{!locked}(a) \implies \begin{array}{r} \sigma[a] = d_1[a] \\ \wedge\ \sigma[a] = d_2[a] \end{array}$$

(if the disk has not failed)

Abstraction relation is preserved by writes and used by reads

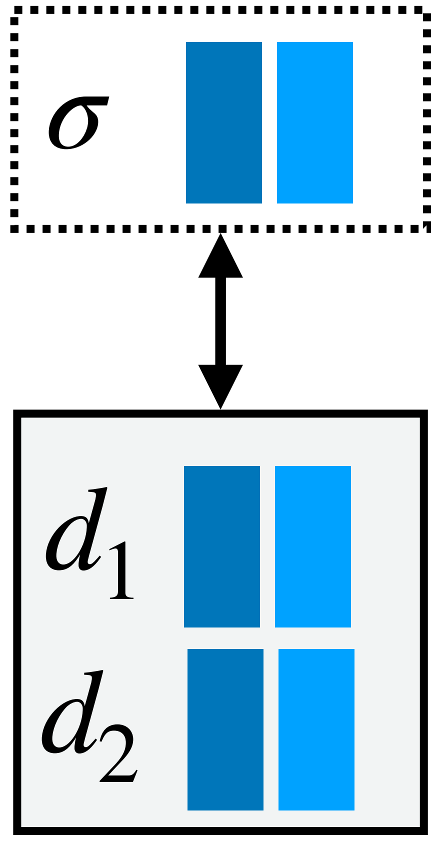
```
func write(a: addr, v: block) {
  lock_address(a)
  d1.write(a, v)
  d2.write(a, v)
  unlock_address(a)
}
```

1. establish lock invariant

```
func read(a: addr): block {
  lock_address(a)
  v, ok := d1.read(a)
  if !ok {
    v, _ = d2.read(a)
  }
  unlock_address(a)
  return v
}
```

2. obtain lock invariant

lock invariant
when $!\mathbf{locked}(a)$:



$$\sigma[a] = d_1[a] \\ \wedge\ \sigma[a] = d_2[a]$$

Crashing breaks the abstraction relation

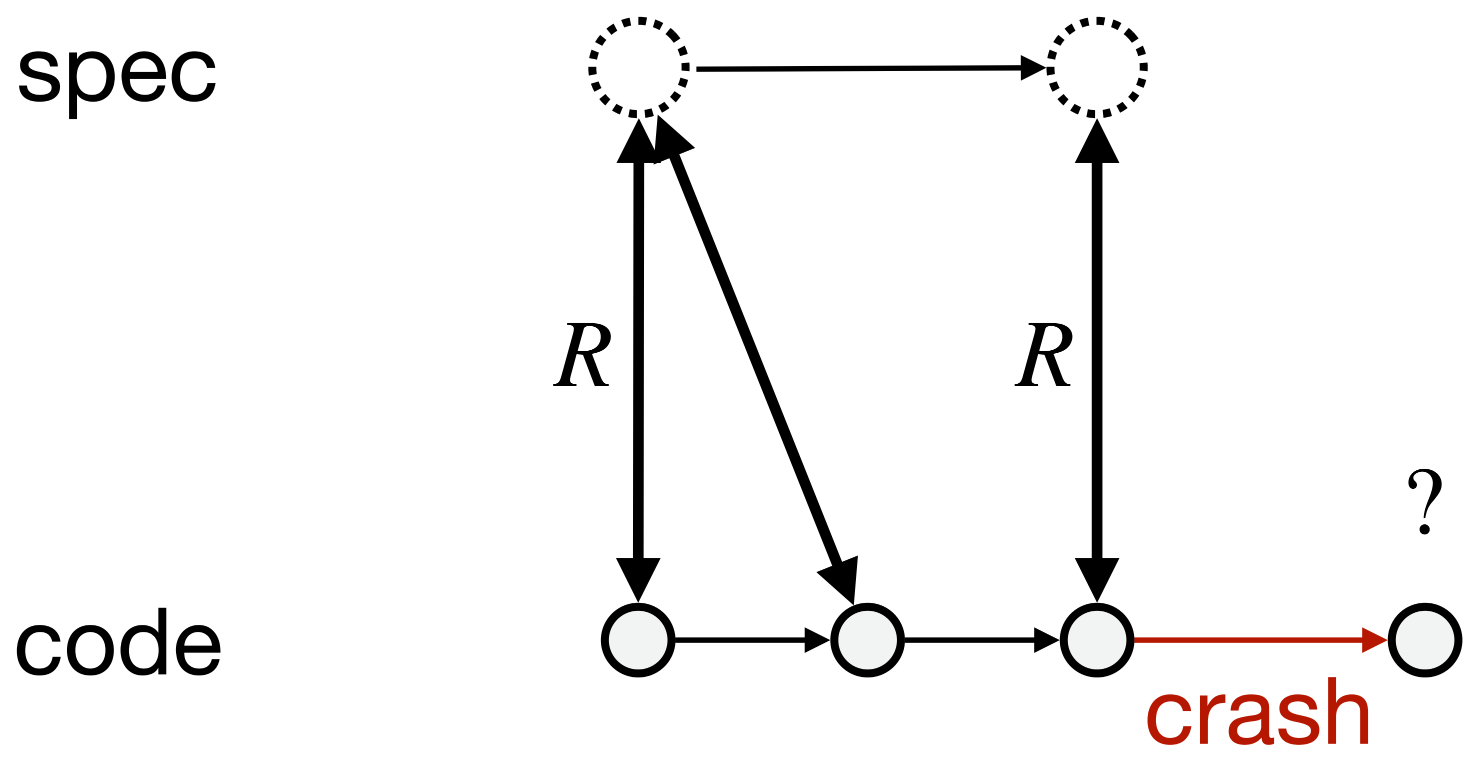
```
func write(a: addr,
           v: block) {

  lock_address(a)
  d1.write(a, v)
```

lock reverts to being free,
but disks are not in-sync

abstraction relation:

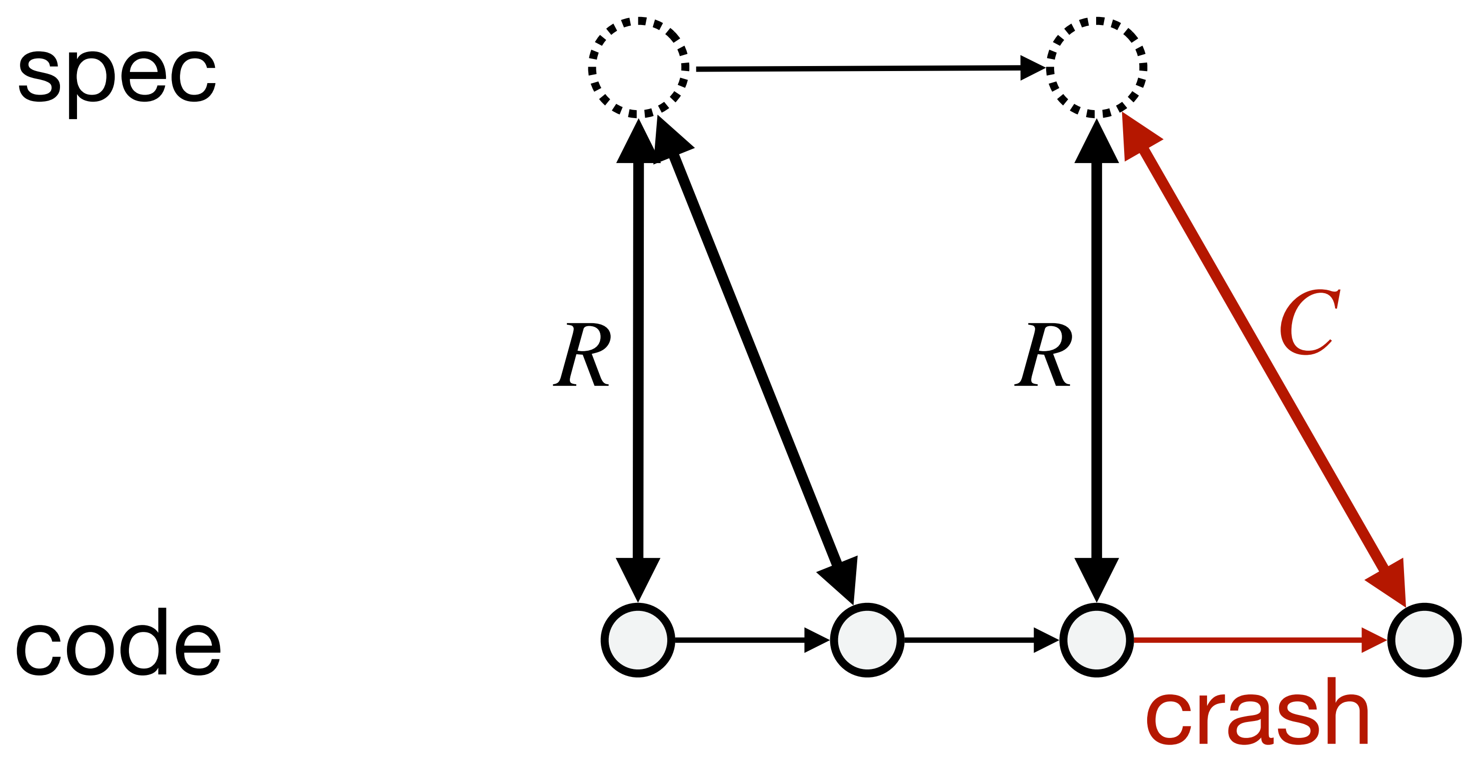
$$\textbf{!locked}(a) \implies \begin{aligned} &\sigma[a] = d_1[a] \\ \wedge\ &\sigma[a] = d_2[a] \end{aligned}$$

So far: abstraction relation always holds



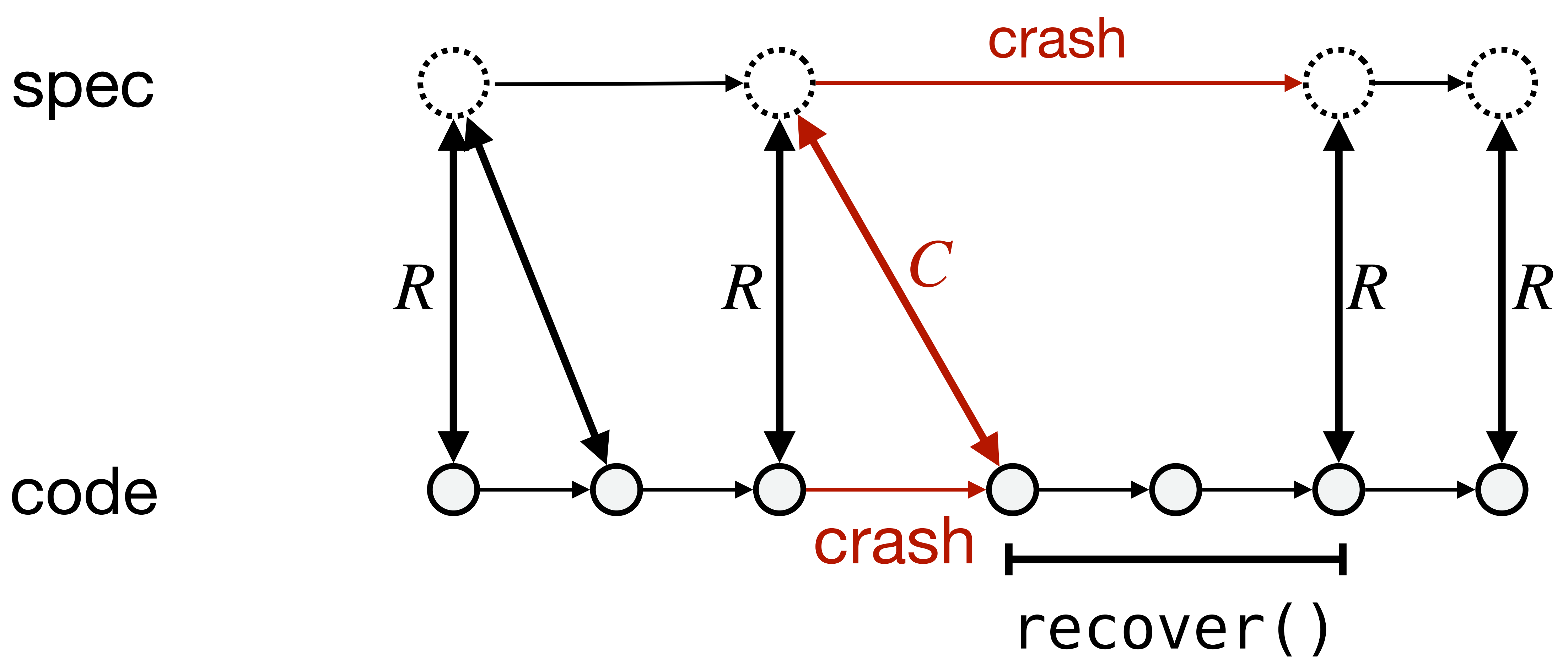
R abstraction relation

Separate a crash invariant from the abstraction relation



R	abstraction relation
C	crash invariant

Recovery proof uses the crash invariant to restore the abstraction relation



R abstraction relation

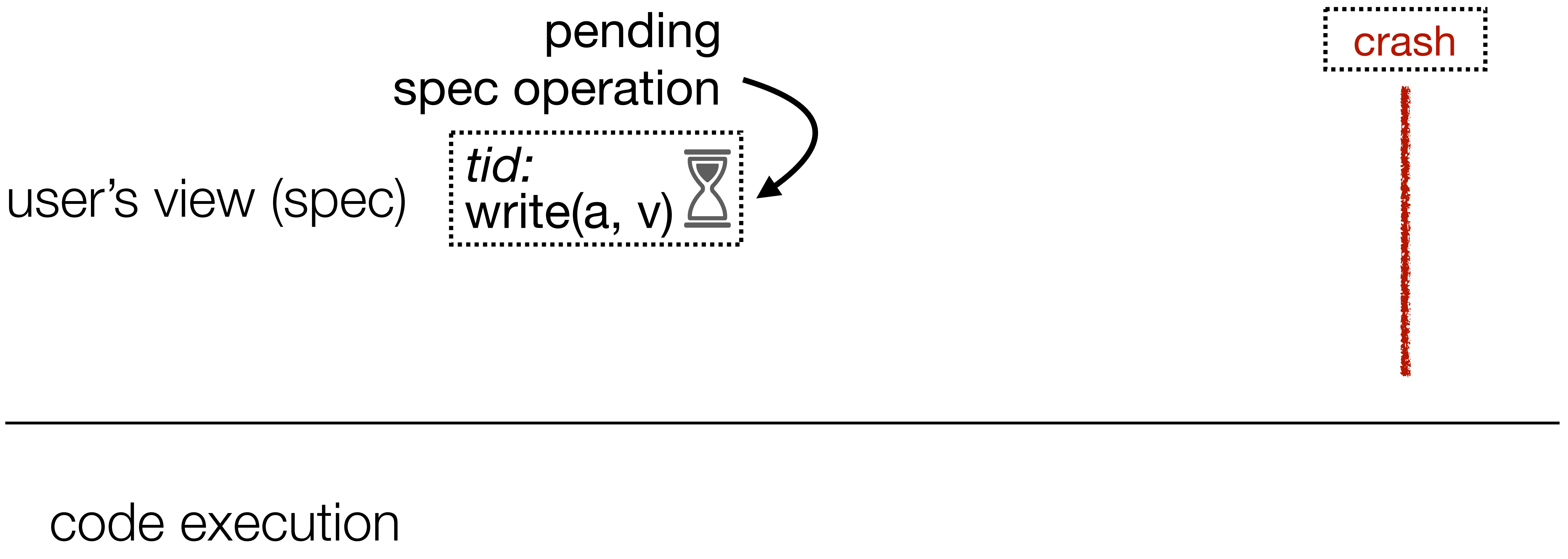
C crash invariant

Proving recovery correct: makes writes atomic

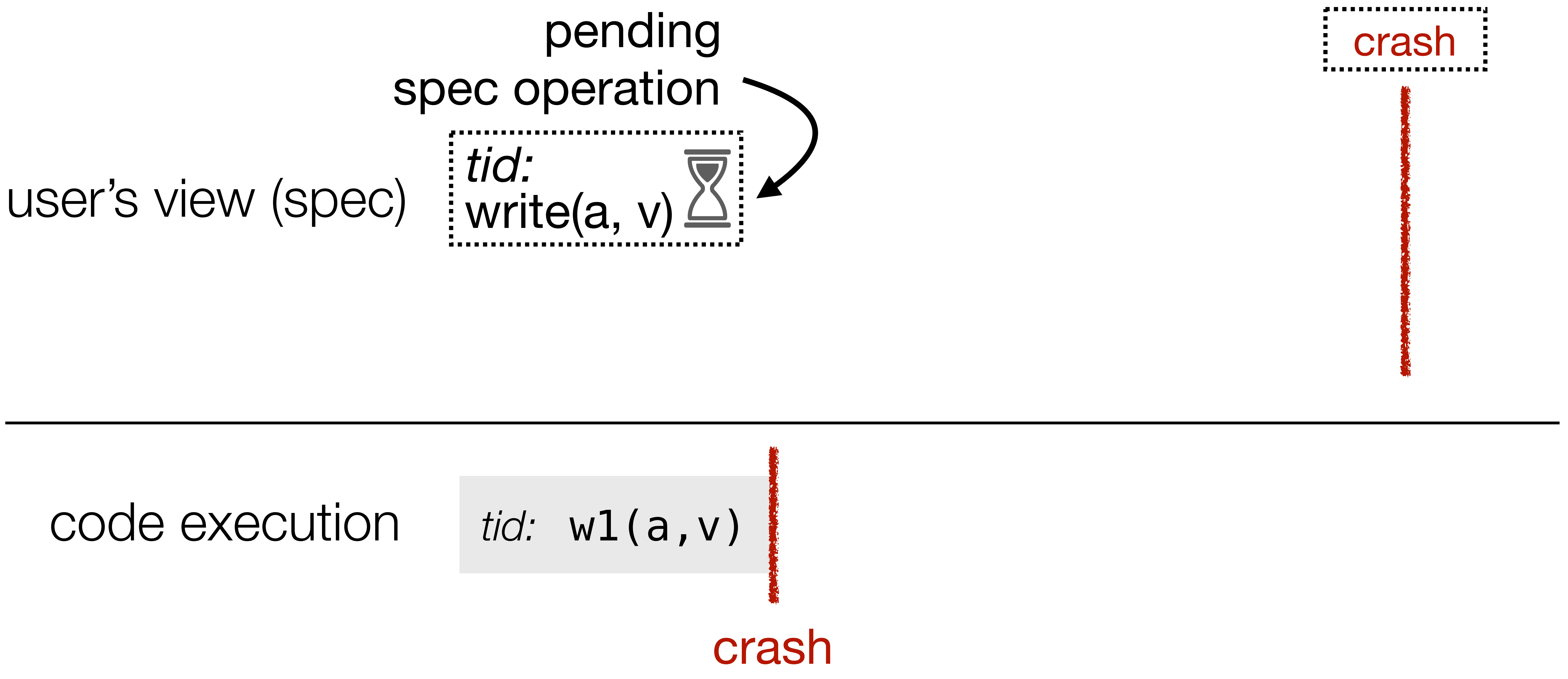
```
func write(a: addr,
           v: block) {

  lock_address(a)
  d1.write(a, v)
```

```
func recover() {
  for a in … {
    v, ok := d1.read(a)
    if !ok { … }
    d2.write(a, v)
  }

}
```

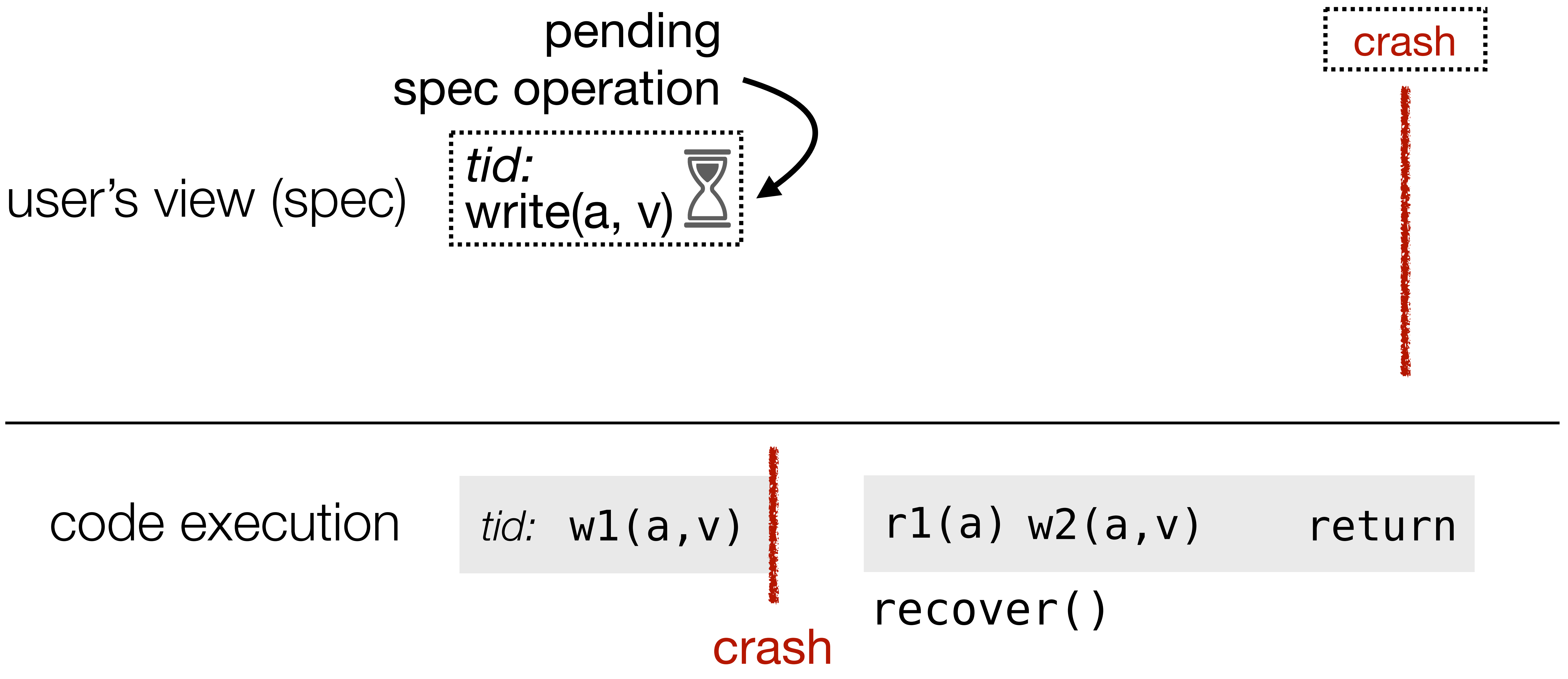
User sees an atomic write due to recovery



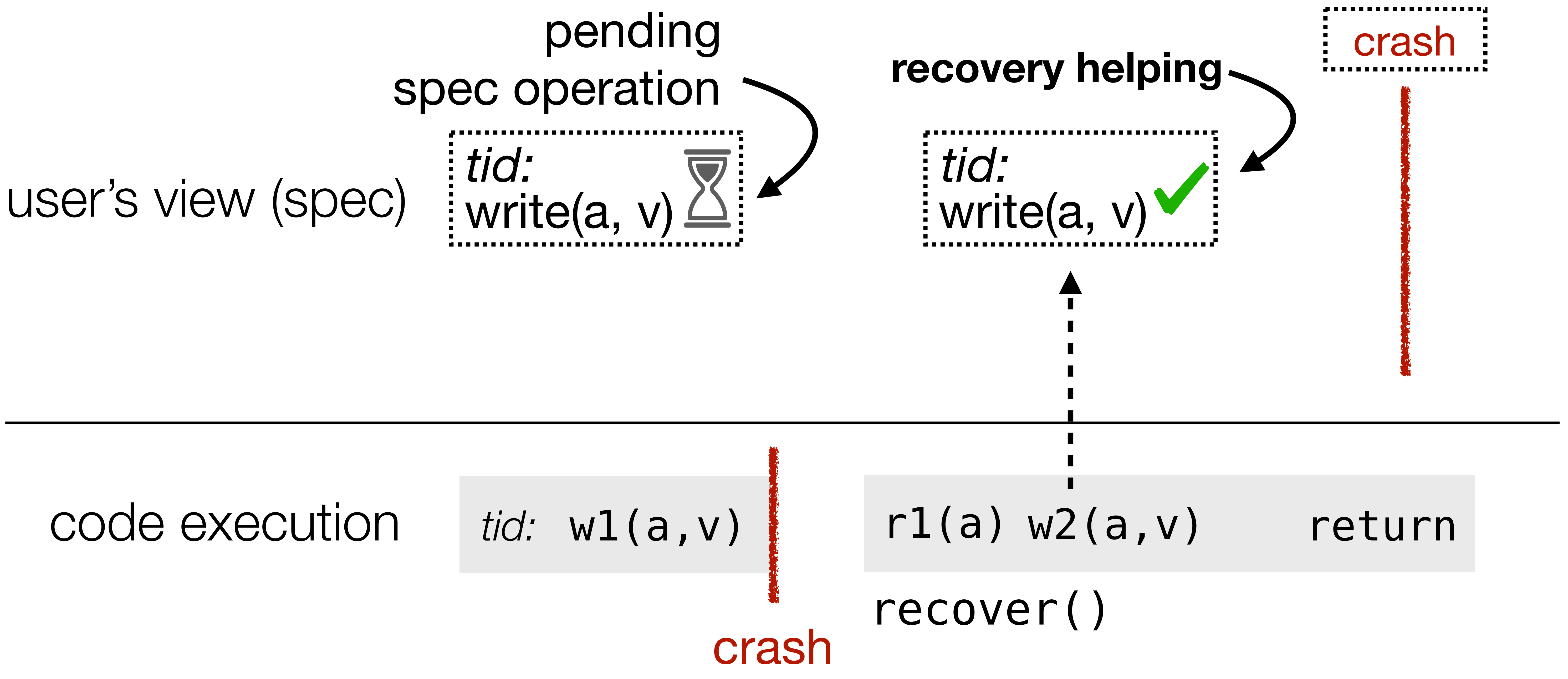
User sees an atomic write due to recovery



User sees an atomic write due to recovery



User sees an atomic write due to recovery



Recovery helping: recovery can commit writes from before the crash

```
func write(a: addr,
           v: block) {

  lock_address(a)
  d1.write(a, v)
```

tid:
write(a, v)

```
func recover() {
  for a in … {
    v, ok := d1.read(a)
    if !ok { … }
    d2.write(a, v)
  }

}
```

tid:
write(a, v) ✓

Crash invariant says "if disks disagree, some thread was writing the value on the first disk"

```
func write(a: addr,
           v: block) {

  lock_address(a)
  d1.write(a, v)
```

tid:
write(a, v)

```
func recover() {
  for a in … {
    v, ok := d1.read(a)
    if !ok { … }
    d2.write(a, v)
  }

}
```

tid:
write(a, v)

crash invariant:

$$d_1[a] \neq d_2[a] \implies \exists \mathrm{tid}.\ \textit{tid: } \text{write}(a, d_1[a])$$

Crash invariant says “if disks disagree, some thread was writing the value on the first disk”

```
func write(a: addr,
           v: block) {

  lock_address(a)
  d1.write(a, v)
```

tid:
write(a, v)

```
func recover() {
  for a in … {
    v, ok := d1.read(a)
    if !ok { … }
    d2.write(a, v)
  }

}
```

tid:
write(a, v)

crash invariant:

$d_1[a] \neq d_2[a] \implies$
$\exists \text{tid}.$ *tid:* write(a, $d_1[a]$)

Key idea: crash invariant can refer to interrupted spec operations

```
func write(a: addr,
           v: block) {

  lock_address(a)
  d1.write(a, v)
```

tid:
write(a, v)

```
func recover() {
  for a in … {
    v, ok := d1.read(a)
    if !ok { … }
    d2.write(a, v)
  }

}
```

tid:
write(a, v)

crash invariant:

$d_1[a] \neq d_2[a] \implies$
$\exists \mathrm{tid}.$ *tid:* write(a, $d_1[a]$)

Recovery proof shows code restores the abstraction relation by completing all interrupted writes

```
func write(a: addr,
           v: block) {

  lock_address(a)
  d1.write(a, v)
```

tid:
write(a, v)

```
func recover() {
  for a in … {
    v, ok := d1.read(a)
    if !ok { … }
    d2.write(a, v)
  }

}
```

tid:
write(a, v)

crash

abstraction relation:

$$\textbf{!locked}(a) \implies \begin{array}{r} \sigma[a] = d_1[a] \\ \wedge\ \sigma[a] = d_2[a] \end{array}$$

Proving concurrent recovery refinement

Recovery proof uses **crash invariant** to restore abstraction relation

Proof can refer to interrupted operations, enabling **recovery helping** reasoning

Users get **correct behavior** and **atomicity**

Perennial is implemented on top of Iris

Perennial (9k lines of Coq)
- leases
- memory versioning
- recovery helping

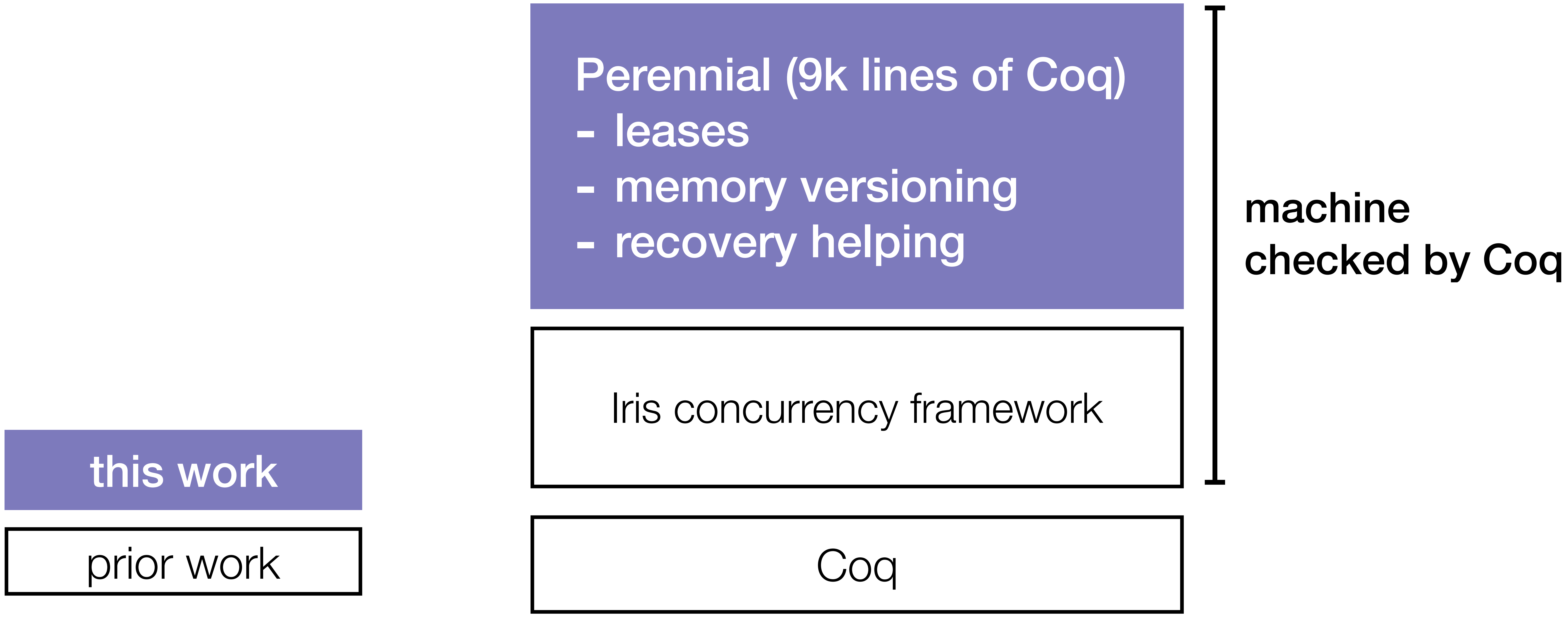
Iris concurrency framework

Coq

this work

prior work

Perennial is implemented on top of Iris



Encoding Perennial into Iris

What is Iris?

modern

extensible

concurrent

separation

logic $\{P\}\ e\ \{Q\}$

What is Iris?

modern

extensible

concurrent

separation $\{P * F\}\ e\ \{Q * F\}$

logic $\{P\}\ e\ \{Q\}$

What is Iris?

modern

extensible

concurrent $\{P * \boxed{I}\}\ e\ \{Q\}$ $\boxed{I}$ “invariant I”

separation $\{P * F\}\ e\ \{Q * F\}$

logic $\{P\}\ e\ \{Q\}$

What is Iris?

modern

r_1, r_2 can be from
user-defined ghost state

extensible $\mathrm{Own}(r_1) * \mathrm{Own}(r_2) \vdash \mathrm{Own}(r_1 \cdot r_2)$

concurrent $\{P * \boxed{I}\}\ e\ \{Q\}$ $\boxed{I}$ "invariant I"

separation $\{P * F\}\ e\ \{Q * F\}$

logic $\{P\}\ e\ \{Q\}$

What is Iris?

modern

r_1, r_2 can be from
user-defined ghost state

user defines how ghost state
can combine

extensible $\quad \mathrm{Own}(r_1) * \mathrm{Own}(r_2) \vdash \mathrm{Own}(r_1 \cdot r_2)$

concurrent $\quad \{P * \boxed{I}\}\ e\ \{Q\}$ $\quad \boxed{I}$ “invariant I”

separation $\quad \{P * F\}\ e\ \{Q * F\}$

logic $\quad \{P\}\ e\ \{Q\}$

What is Iris?

modern gitlab.mpi-sws.org/iris/iris

r_1, r_2 can be from user-defined ghost state

user defines how ghost state can combine

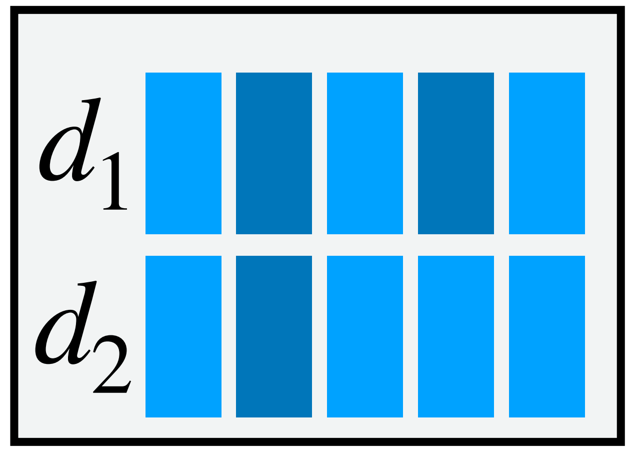
extensible $\mathrm{Own}(r_1) * \mathrm{Own}(r_2) \vdash \mathrm{Own}(r_1 \cdot r_2)$

concurrent $\{P * \boxed{I}\}\ e\ \{Q\}$ $\boxed{I}$ “invariant I”

separation $\{P * F\}\ e\ \{Q * F\}$

logic $\{P\}\ e\ \{Q\}$

Disk resources



"Points-to assertion" for memory: $m[a] \mapsto v$

Similarly, for disk addresses: $d_1[a] \mapsto v$

Represents *exclusive ownership* of one address

First encoding problem: modeling crashes

$d_1[a] \mapsto v$ still holds after a crash

$m[a] \mapsto v$ does not

Perennial uses *memory versioning* to model global crashes

Write $m[a] \mapsto_n v$, only applies to memory version n

Volatile resources are invalidated on crash by incrementing version number

Perennial uses *memory versioning* to model global crashes

Write $m[a] \mapsto_n v$, only applies to memory version n

Volatile resources are invalidated on crash by incrementing version number

Write $d_1[a] \mapsto v$, does not depend on version number

Use Iris invariants to encode abstraction relation and crash invariant

The resource $\boxed{I}$ holds at all intermediate points

Use an invariant to encode abstraction relation and crash invariant

An invariant can be used for an atomic step

tid_1: invariants

$\boxed{I}$ I

$(\boxed{I} * I)$

$\downarrow$ atomic step

$(\boxed{I} * I)$

$\boxed{I}$ I

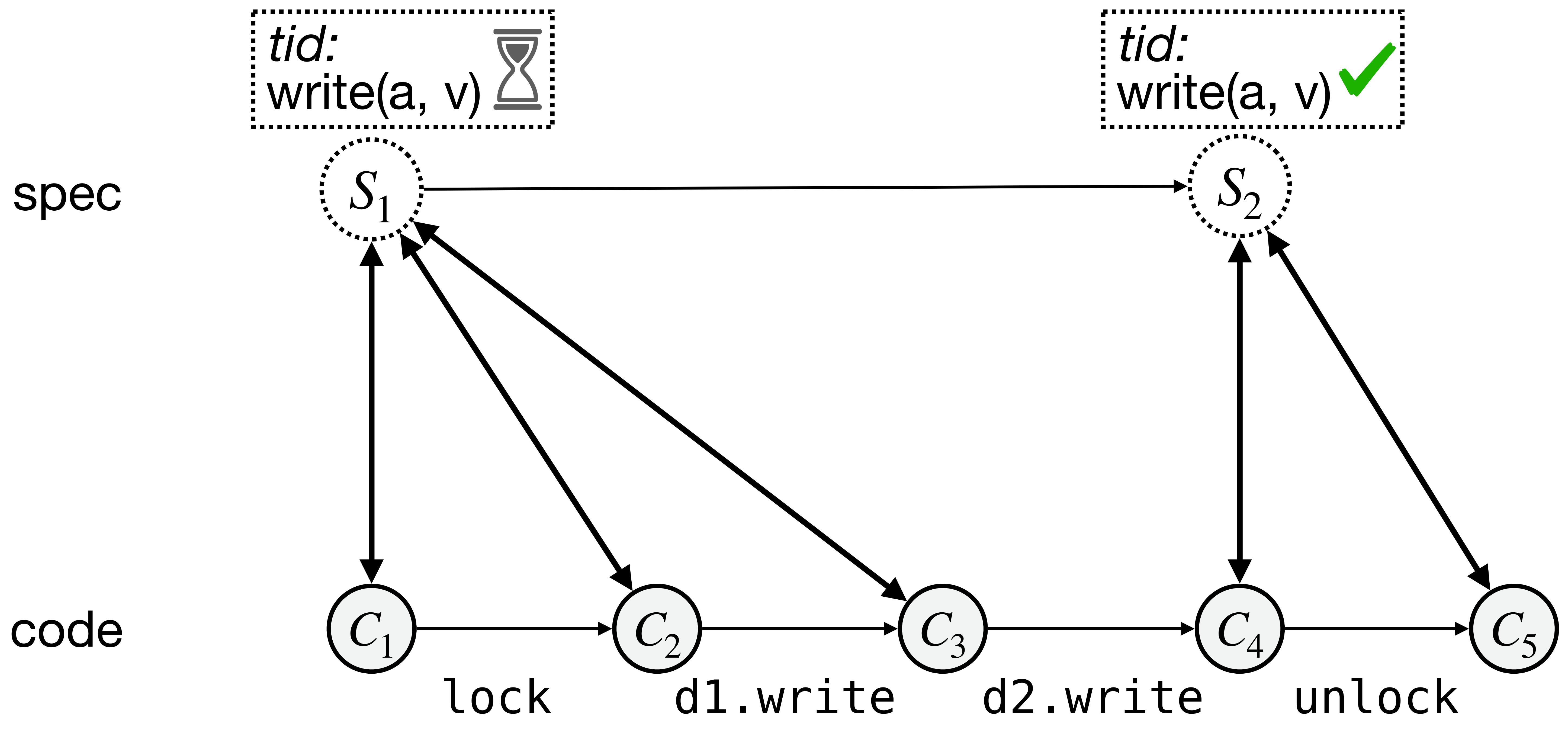
Invariants can be shared among threads

tid$_1$: invariants

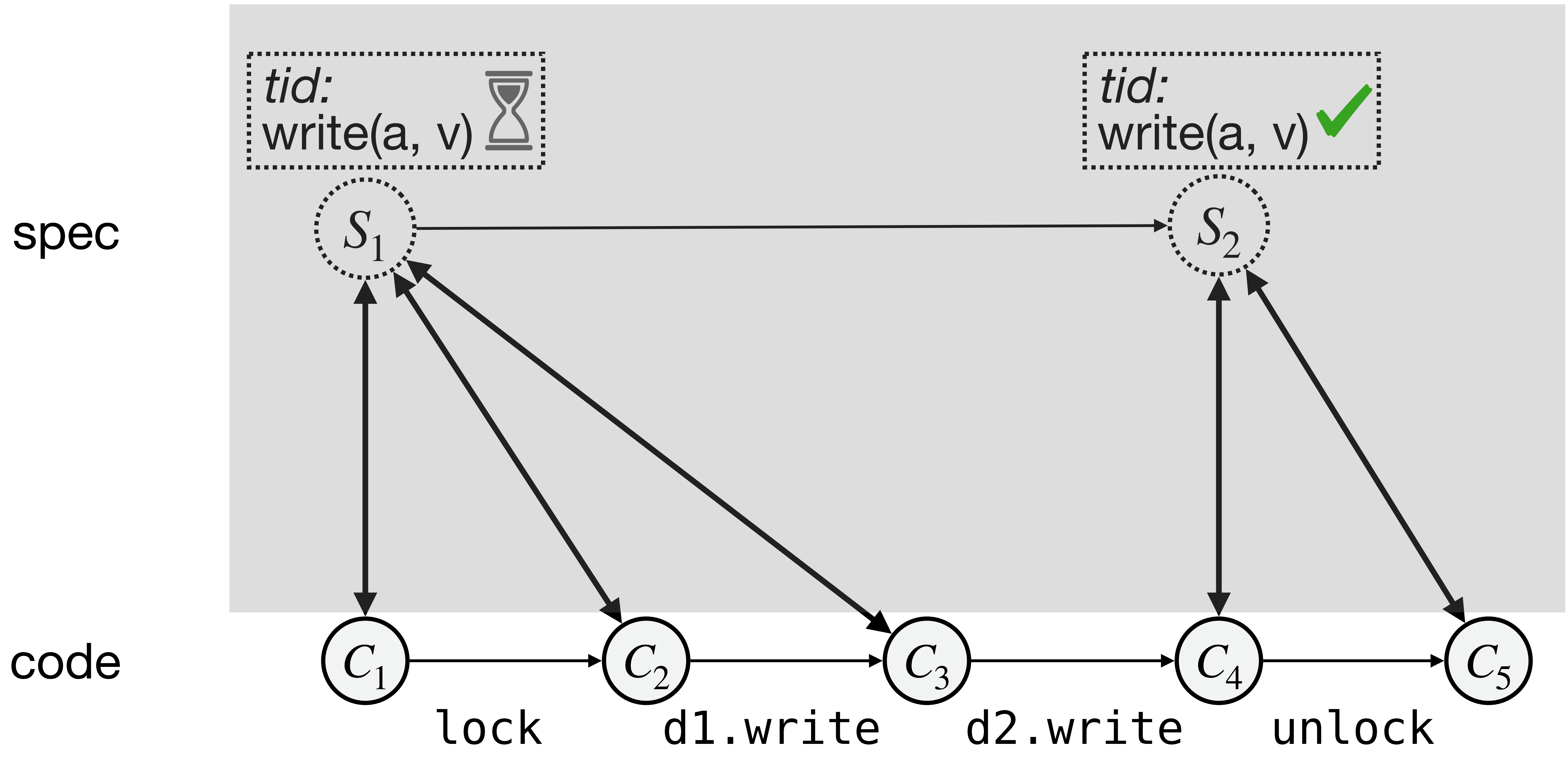
tid$_2$:

$$\begin{tikzcd}
(P * I) \arrow[d, dashed] \arrow[drr, dashed] & & \\
(P * \boxed{I}) \arrow[d] \arrow[dr, "fork"] & & I \\
(P * \boxed{I}) & \boxed{I} & I
\end{tikzcd}$$

Refinement in Iris



Refinement in Iris



Idea: model the entire specification part of refinement using ghost resources…

…and enforce it using an invariant.

Refinement in Iris: specification resources

specification resource
refers to user threads

also track abstract
state using ghost state

tid:
write(a, v) $*$ source(σ)

tid:
write(a, v)

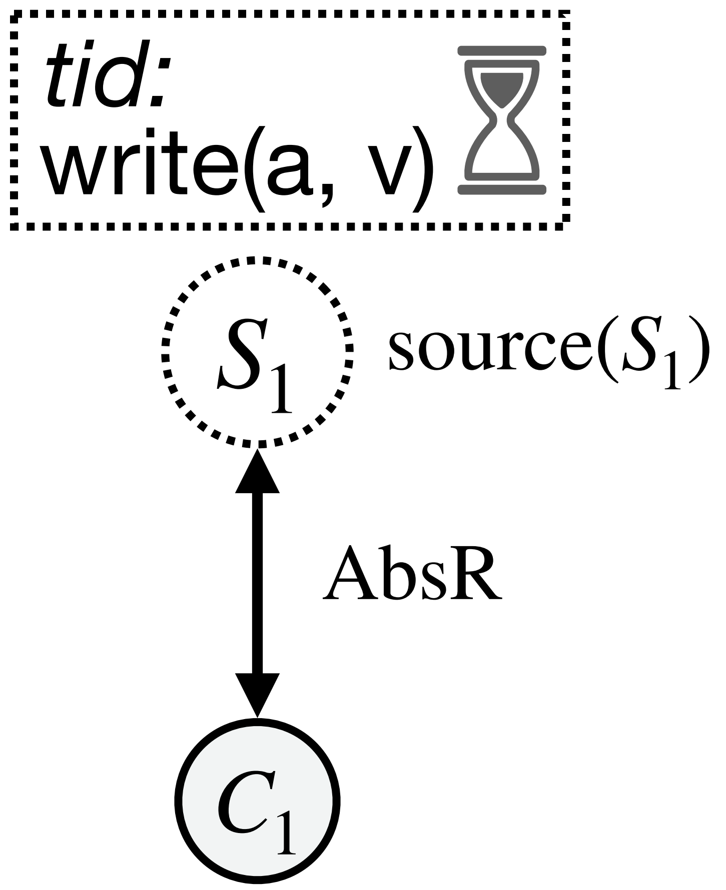
S_1 source(S_1)

Refinement in Iris: specification resources

specification resource
refers to user threads

also track abstract
state using ghost state

$$\left\{ \textit{tid:}\ \text{write(a, v)} * \boxed{\text{source}(\sigma) * \text{AbsR}} \right\}$$



Refinement in Iris: specification resources

specification resource
refers to user threads

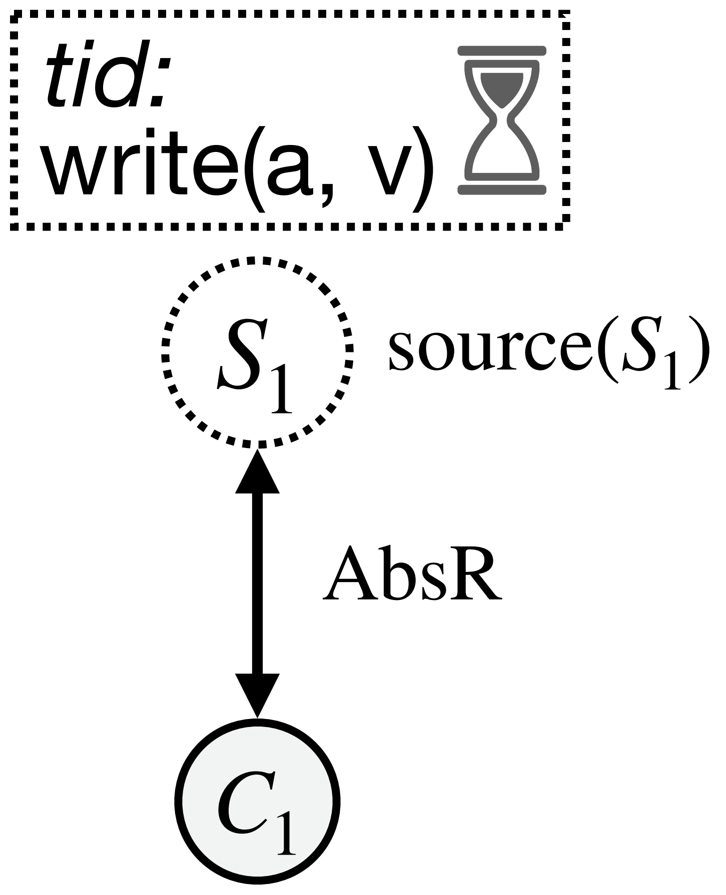
also track abstract
state using ghost state

{ *tid:* write(a, v) ⌛ } $*$ $\boxed{\mathrm{source}(\sigma) * \mathrm{AbsR}}$ }

```
write(a, v)
```

{ *tid:* write(a, v) ✓ }

Specification resources can only be updated
by following the spec



Perennial extends Iris refinement to incorporate crashing and recovery

$\{$ *tid:* write(a, v) $* \boxed{\mathrm{source}(\sigma) * \mathrm{AbsR}} \}$ $\{$ crashing $* \boxed{\mathrm{source}(\sigma) * \mathrm{CrashInv}} \}$

$\mathrm{AbsR}_n \implies \mathrm{CrashInv}_{n+1}$ crash invariant holds after a crash

$\mathrm{CrashInv}_{n+1} \implies \mathrm{CrashInv}_{n+2}$ for crashes during recovery

Perennial extends Iris refinement to incorporate crashing and recovery

$\left\{ \begin{array}{l} \textit{tid:} \\ \text{write(a, v)} ⌛ \end{array} * \boxed{\text{source}(\sigma) * \text{AbsR}} \right\}$

```
write(a, v)
```

$\left\{ \begin{array}{l} \textit{tid:} \\ \text{write(a, v)} ✓ \end{array} \right\}$

$\left\{ \text{crashing} * \boxed{\text{source}(\sigma) * \text{CrashInv}} \right\}$

$\text{AbsR}_n \implies \text{CrashInv}_{n+1}$ crash invariant holds after a crash

$\text{CrashInv}_{n+1} \implies \text{CrashInv}_{n+2}$ for crashes during recovery

Perennial extends Iris refinement to incorporate crashing and recovery

$\left\{ \boxed{\textit{tid:}\ \text{write(a, v)}\ ⌛} * \boxed{\text{source}(\sigma) * \text{AbsR}} \right\}$

```
write(a, v)
```

$\left\{ \boxed{\textit{tid:}\ \text{write(a, v)}\ ✔} \right\}$

$\left\{ \boxed{\text{crashing}} * \boxed{\text{source}(\sigma) * \text{CrashInv}} \right\}$

```
recover()
```

$\left\{ \boxed{\text{crash}} \right\}$

$\text{AbsR}_n \implies \text{CrashInv}_{n+1}$ crash invariant holds after a crash

$\text{CrashInv}_{n+1} \implies \text{CrashInv}_{n+2}$ for crashes during recovery

Perennial required relatively few changes to Iris

Introduced new disk and volatile resources with versioning

Recovery leases are also a new ghost resource

Hard part: prove that refinement triples imply concurrent recovery refinement

Reasoning about runnable systems

Perennial (9k lines of Coq)
- leases
- memory versioning
- recovery helping

Iris concurrency framework

Coq

developer-written

this paper

prior work

Reasoning about runnable systems

Go source

`go build`

`exe`

developer-written

this paper

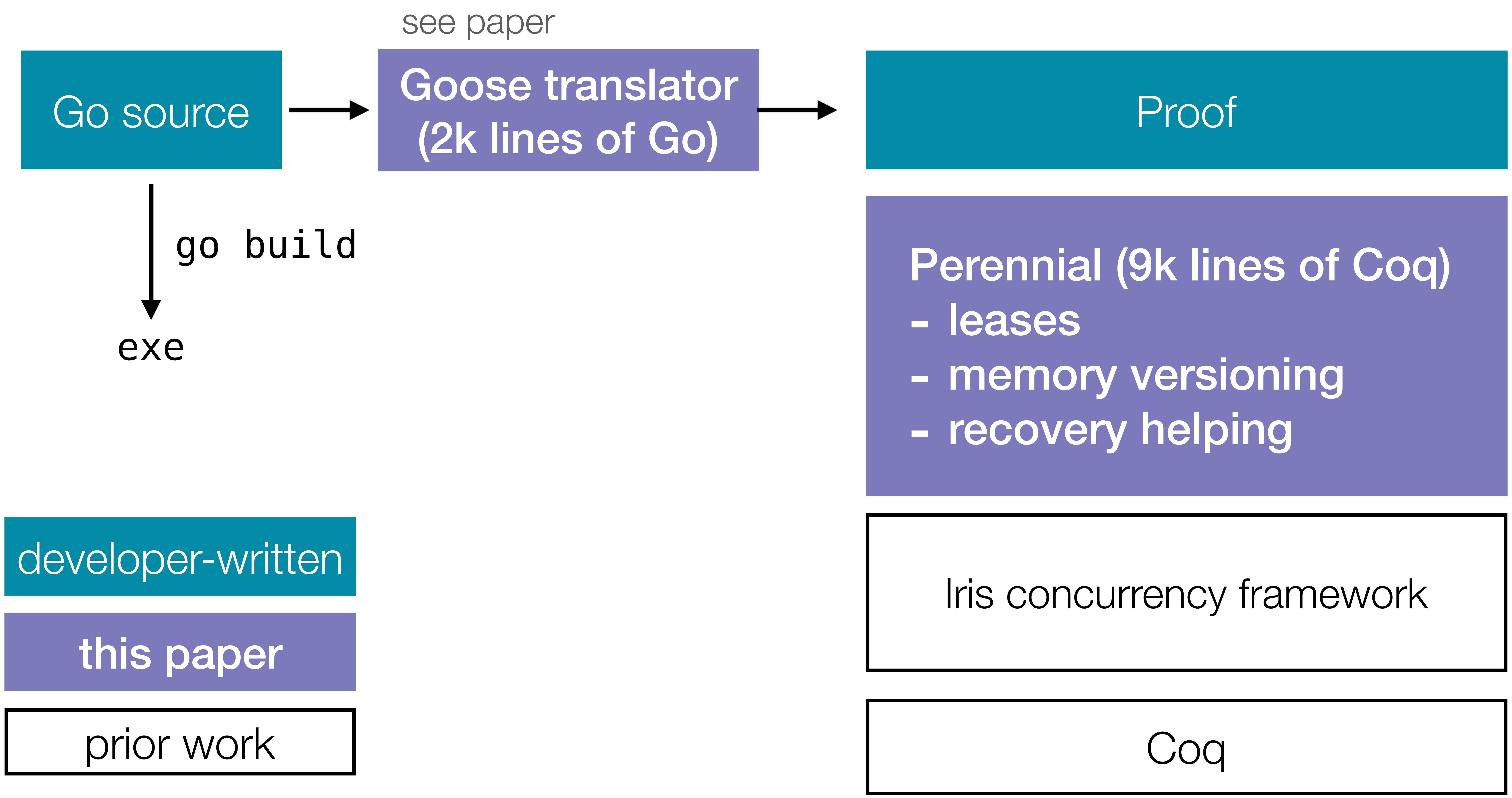
prior work

Perennial (9k lines of Coq)
- **leases**
- **memory versioning**
- **recovery helping**

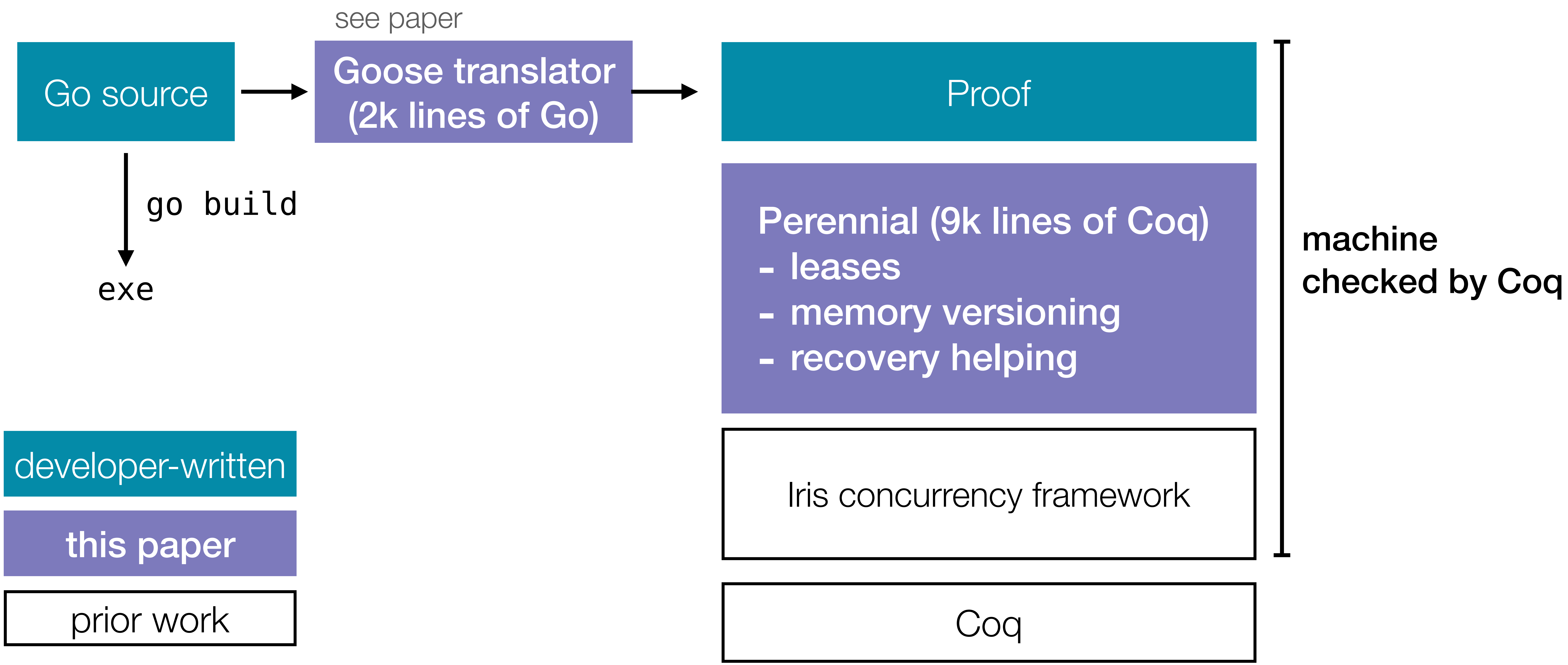
Iris concurrency framework

Coq

Reasoning about runnable systems



Reasoning about runnable systems



Translation example

db.go

```go
// A Table provides access to an immutable copy of da
// along with an index for fast random access.
type Table struct {
    Index map[uint64]uint64
    File  filesys.File
}

// CreateTable creates a new, empty table.
func CreateTable(p string) Table {
    index := make(map[uint64]uint64)
    f, _ := filesys.Create("db", p)
    filesys.Close(f)
    f2 := filesys.Open("db", p)
    return Table{Index: index, File: f2}
}
```

goose →

db.v

```coq
Module Table.
  (* A Table provides access to an immutable copy of data on the filesystem,
     along with an index for fast random access. *)
  Record t {model:GoModel} := mk {
    Index: Map uint64;
    File: File;
  }.
  Arguments mk {model}.
  Global Instance t_zero {model:GoModel} : HasGoZero t := mk (zeroValue _) (z
End Table.

(* CreateTable creates a new, empty table. *)
Definition CreateTable {model:GoModel} (p:string) : proc Table.t :=
  index ← Data.newMap uint64;
  let! (f, _) ← FS.create "db" p;
  _ ← FS.close f;
  f2 ← FS.open "db" p;
  Ret {| Table.Index := index;
         Table.File := f2; |}.
```

Goose translates to a simple Coq model

Target a shallow embedding

Disallow reassignments, so variables are translated to pure bindings

Only need a semantics for pointers, slices, and maps

Coq model correctly accounts for non-atomic loads and stores

Go emits ordinary u64 loads and stores: not sequentially consistent on x86

Model makes racy access undefined behavior

Reasoning is still pleasant since $m[a] \mapsto v$ is exclusive, so guarantees reads and writes will not race

Goose translator is carefully written to model Go faithfully

Use official `go/ast` and `go/types` packages

Avoid subtle parts of the language

Easy to hand-audit translation

Coq model type checks

Evaluation

This talk:

- proof-effort comparison

See paper:

- verified examples
- TCB
- bug discussion

Methodology:
Verify the same mail server as previous work, CSPEC [OSDI ’18]

Users can read, deliver, and delete mail

Implemented on top of a file system

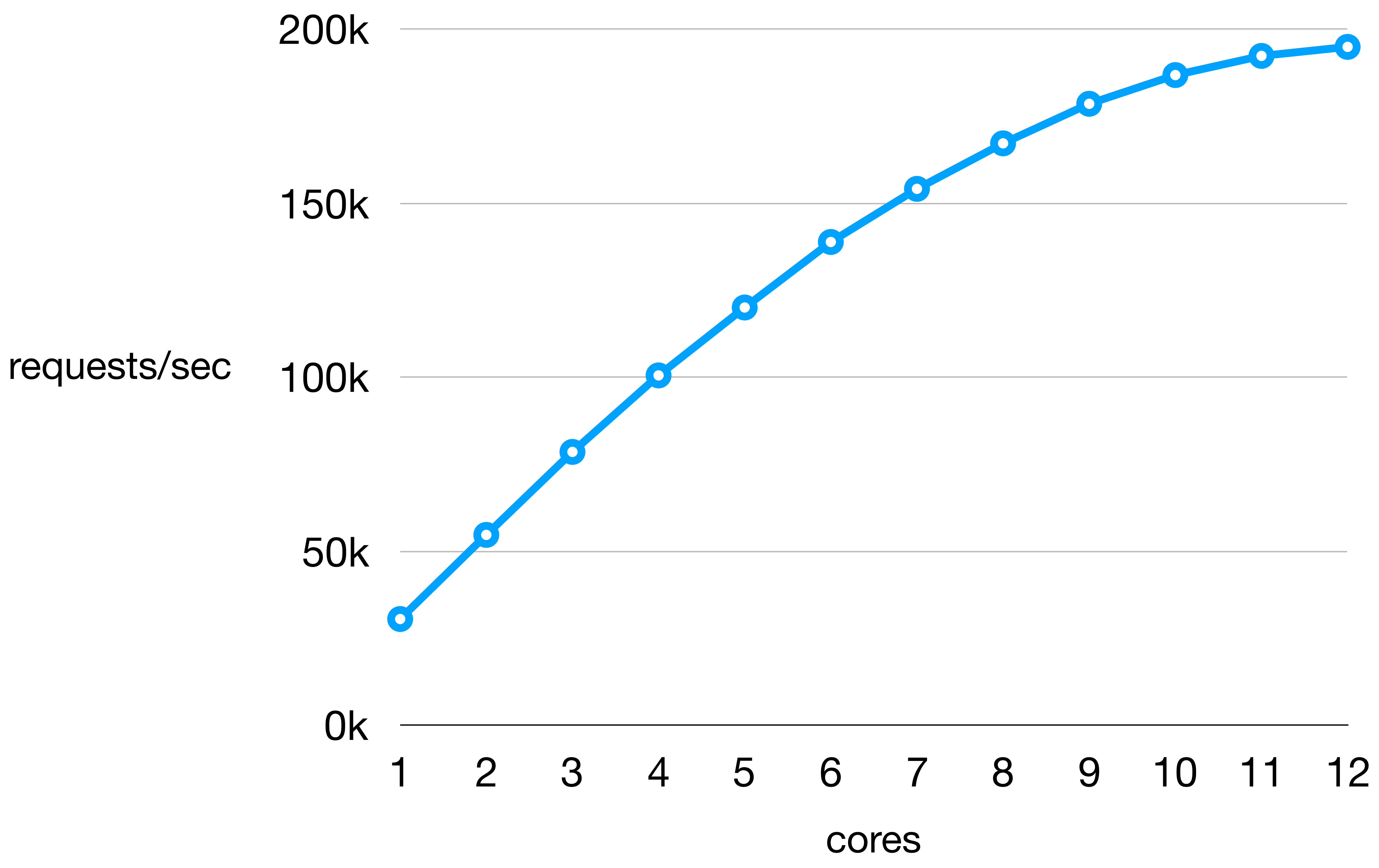
Operations are **atomic** (and **crash safe** in Perennial)

Perennial mail server was easier to verify and proves crash safety

	Perennial	**CSPEC** [OSDI ’18]
mail server proof	3,200	4,000
time	2 weeks (**after** framework)	6 months (**with** framework)
code	159 (Go)	215 (Coq)

Perennial mail server really is concurrent

(see the paper for details)



What’s next?

Verifying NFS: concurrent file system

Still using Iris, but improving metatheory and Goose

Conclusion

Perennial introduces **crash-safety techniques** that extend concurrent verification in Iris

Goose lets us reason about **Go implementations**

Verified a Go mail server with **less effort** than previous work and proved crash safety

github.com/mit-pdos/perennial