# Certifying a Crash-safe File System

Haogang Chen

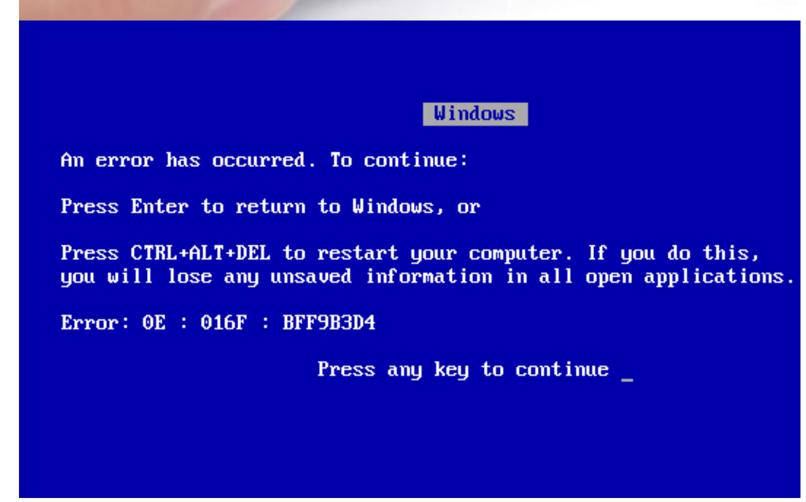
Thesis Advisors
Frans Kaashoek and Nickolai Zeldovich

# File systems should not lose data

- People use file systems to store permanent data

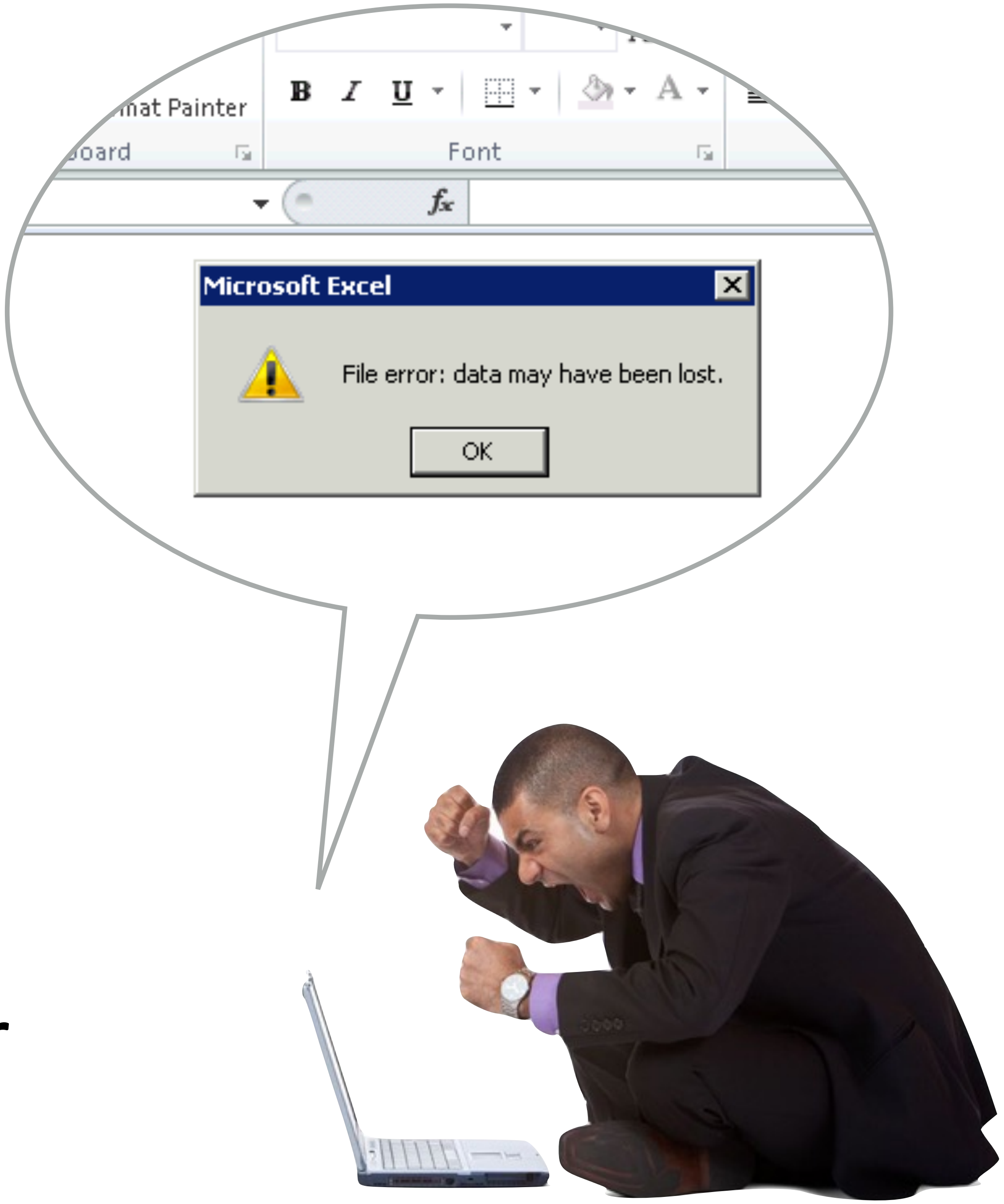
# File systems should not lose data

- People use file systems to store permanent data
- Computers can crash anytime
  - power failures
  - hardware failures (unplug USB drive)
  - software bugs



# File systems should not lose data

- People use file systems to store permanent data
- Computers can crash anytime
  - power failures
  - hardware failures (unplug USB drive)
  - software bugs
- File systems should not lose or corrupt data in case of crashes



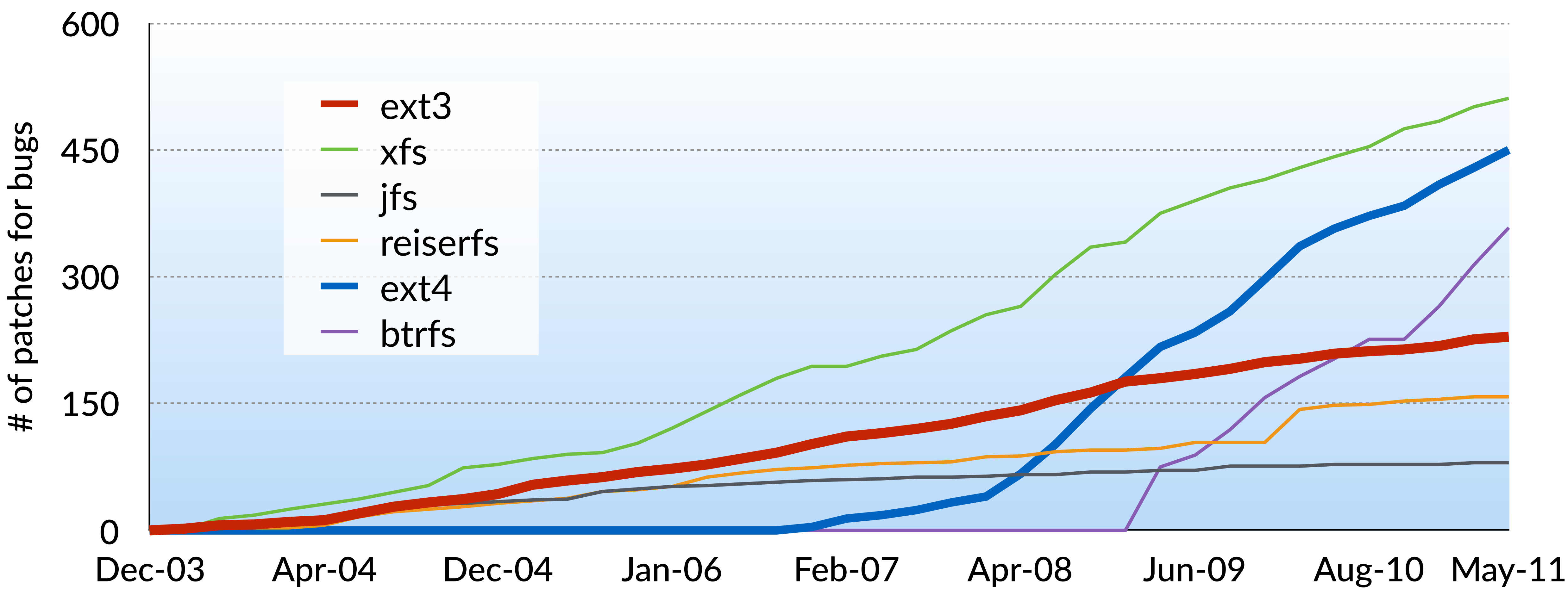
# File systems are complex and have bugs

- Linux ext4: ~60,000 lines of code

# File systems are complex and have bugs

- Linux ext4: ~60,000 lines of code

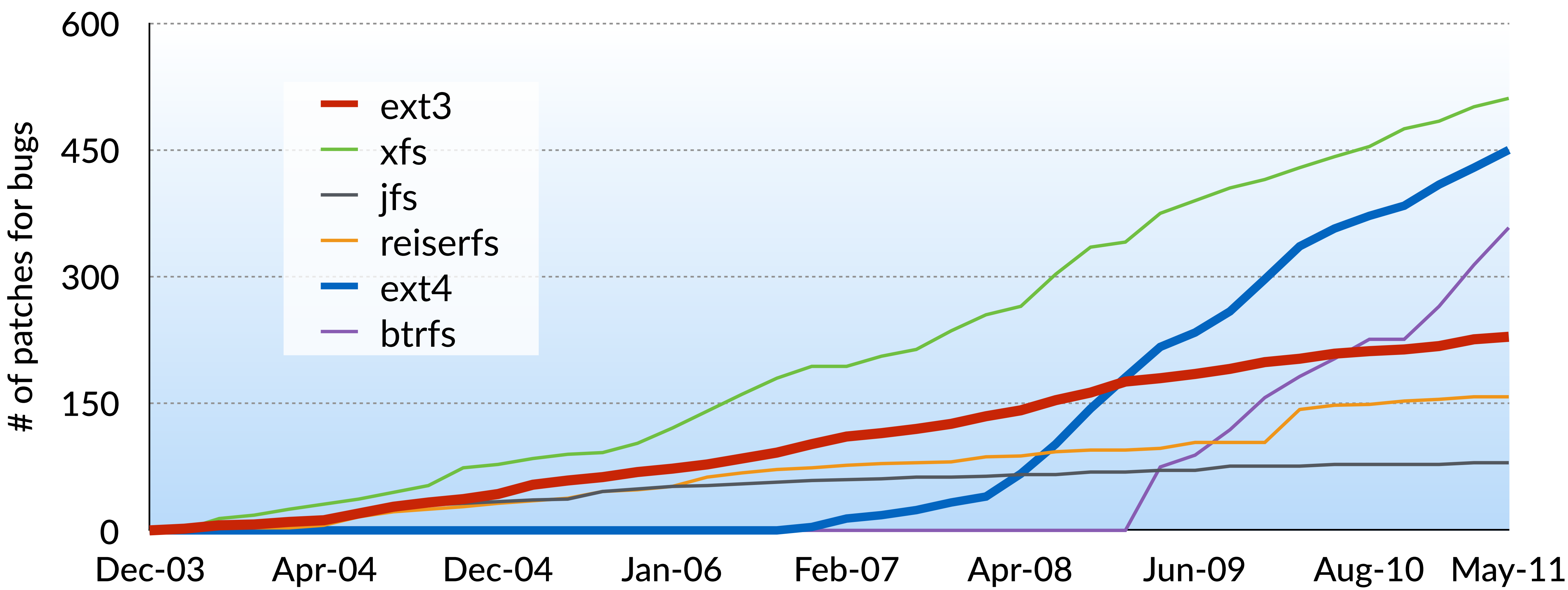
Cumulative number of bug patches in Linux file systems [Lu et al., FAST'13]



# File systems are complex and have bugs

- Linux ext4: ~60,000 lines of code
- Some bugs are serious: **data loss, security exploits**, etc.

Cumulative number of bug patches in Linux file systems [Lu et al., FAST'13]



# Researches in avoiding bugs in file systems

- Most research is on finding bugs
  - Crash injection (e.g., EXPLODE [OSDI'06])
  - Symbolic execution (e.g., EXE [Oakland'06])
  - Design modeling (e.g., in Alloy [ABZ'08])
- Some elimination of bugs by proving:
  - FS without directories [Arkoudas et al. 2004]
  - BilbyFS [Keller 2014]
  - UBIFS [Ernst et al. 2013]

# Researches in avoiding bugs in file systems

- Most research is on finding bugs
  - Crash injection (e.g., EXPLODE [OSDI'06])
  - Symbolic execution (e.g., EXE [Oakland'06])
  - Design modeling (e.g., in Alloy [ABZ'08])

**reduce # of bugs**

- Some elimination of bugs by proving:
  - FS without directories [Arkoudas et al. 2004]
  - BilbyFS [Keller 2014]
  - UBIFS [Ernst et al. 2013]

# Researches in avoiding bugs in file systems

- Most research is on finding bugs
  - Crash injection (e.g., EXPLODE [OSDI'06])
  - Symbolic execution (e.g., EXE [Oakland'06])
  - Design modeling (e.g., in Alloy [ABZ'08])

**reduce # of bugs**

- Some elimination of bugs by proving:
  - FS without directories [Arkoudas et al. 2004]
  - BilbyFS [Keller 2014]
  - UBIFS [Ernst et al. 2013]

**incomplete + no crashes**

# Dealing with crashes is hard

- Crashes expose many partially-updated states
  - Reasoning about all failure cases is hard

# Dealing with crashes is hard

- Crashes expose many partially-updated states
  - Reasoning about all failure cases is hard
- Performance optimizations lead to more tricky partial states
  - Disk I/O is expensive
  - Buffer updates in memory

# Dealing with crashes is hard

- Crashes expose many partially-updated states
  - Reasoning about all failure cases is hard
- Performance optimizations lead to more tricky partial states
  - Disk I/O is expensive
  - Buffer updates in memory

A patch for Linux's write-ahead logging (jbd) in 2012:

**"Is it safe to omit a disk write barrier here?"**

```
commit 353b67d8ced4dc53281c88150ad295e24bc4b4c5
Author: Jan Kara <jack@suse.cz>
Date:   Sat Nov 26 00:35:39 2011 +0100
Title:  jbd: Issue cache flush after checkpointing

--- a/fs/jbd/checkpoint.c
+++ b/fs/jbd/checkpoint.c
@@ -504,7 +503,25 @@ int cleanup_journal_tail(journal_t *journal)
                spin_unlock(&journal->j_state_lock);
                return 1;
        }
+       spin_unlock(&journal->j_state_lock);
+
+       /*
+        * We need to make sure that any blocks that were recently written out
+        * --- perhaps by log_do_checkpoint() --- are flushed out before we
+        * drop the transactions from the journal. It's unlikely this will be
+        * necessary, especially with an appropriately sized journal, but we
+        * need this to guarantee correctness.  Fortunately
+        * cleanup_journal_tail() doesn't get called all that often.
+        */
+       if (journal->j_flags & JFS_BARRIER)
+               blkdev_issue_flush(journal->j_fs_dev, GFP_KERNEL, NULL);

+       spin_lock(&journal->j_state_lock);
+       if (!tid_gt(first_tid, journal->j_tail_sequence)) {
+               spin_unlock(&journal->j_state_lock);
+               /* Someone else cleaned up journal so return 0 */
+               return 0;
+       }
```

> It's unlikely this will be necessary, … but we need this to guarantee correctness.
> Fortunately this function doesn't get called all that often.

# Goal: certify a file system under crashes

# Goal: certify a file system under crashes

- **FSCQ**: first certified crash-safe file system


A complete file system with a **machine-checkable proof** that its implementation meets its specification, both under **normal execution** and under any sequence of **crashes**, including crashes during recovery.

# Contributions

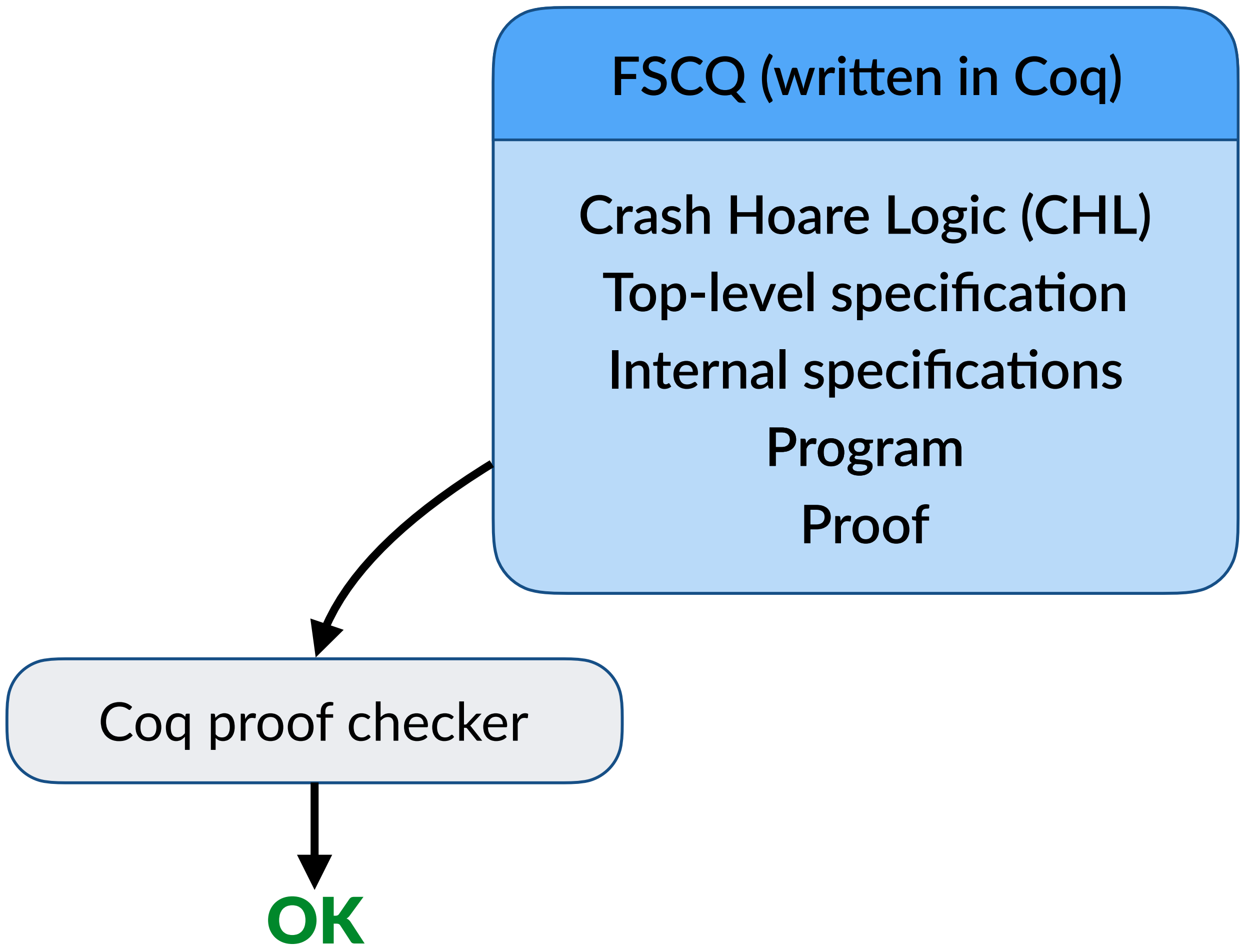
- **CHL**: Crash Hoare Logic
    - Specification framework for crash-safety of storage
    - Crash condition and recovery semantics
    - Automation to reduce proof effort
- **FSCQ**: the first certified crash-safe file system
    - Basic Unix-like file system (no hard-links, no concurrency)
    - Precise specification for the core subset of POSIX
    - I/O performance on par with Linux ext4
    - CPU overhead is high

# FSCQ runs standard Unix programs

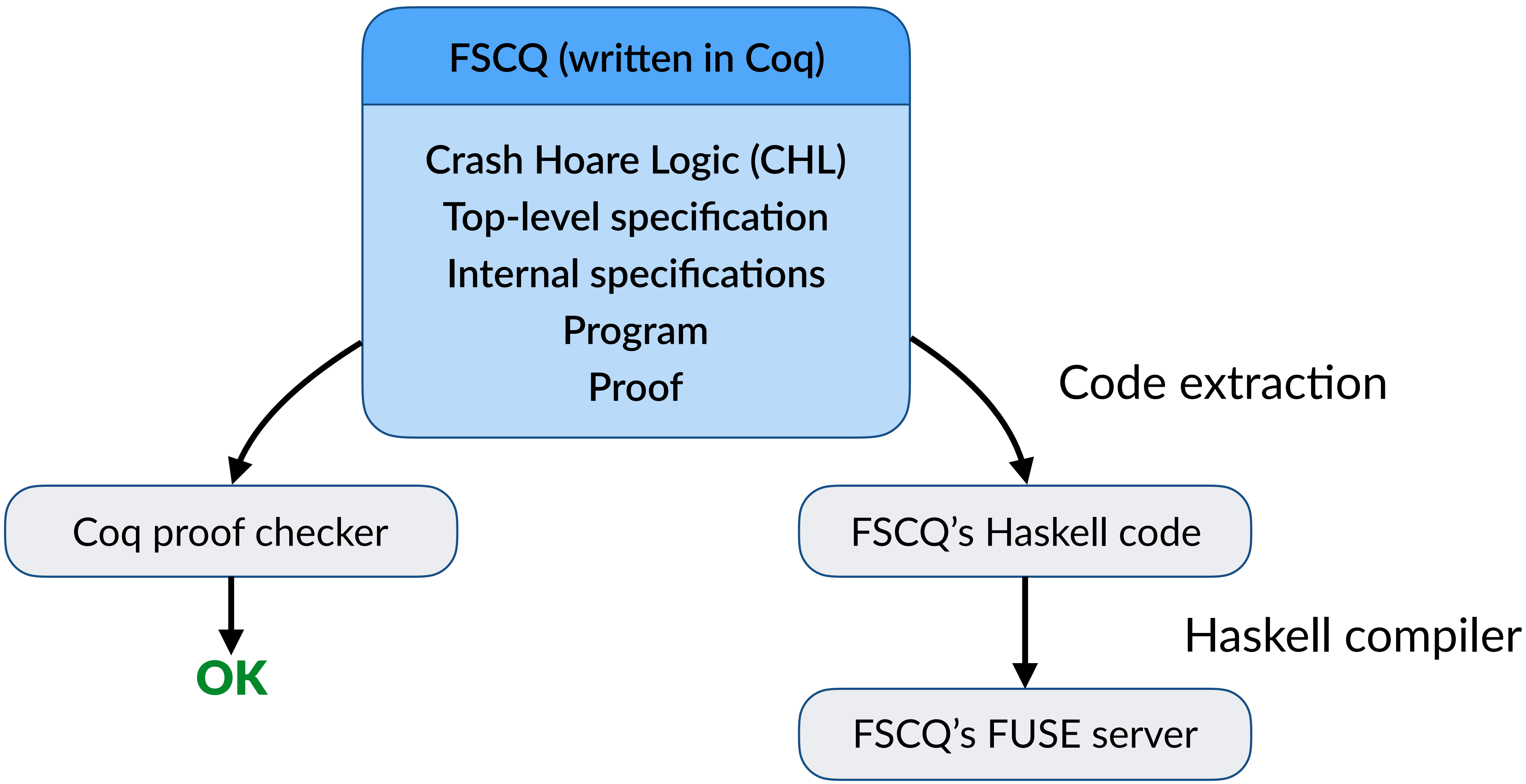
FSCQ (written in Coq)

Crash Hoare Logic (CHL)
Top-level specification
Internal specifications
Program
Proof

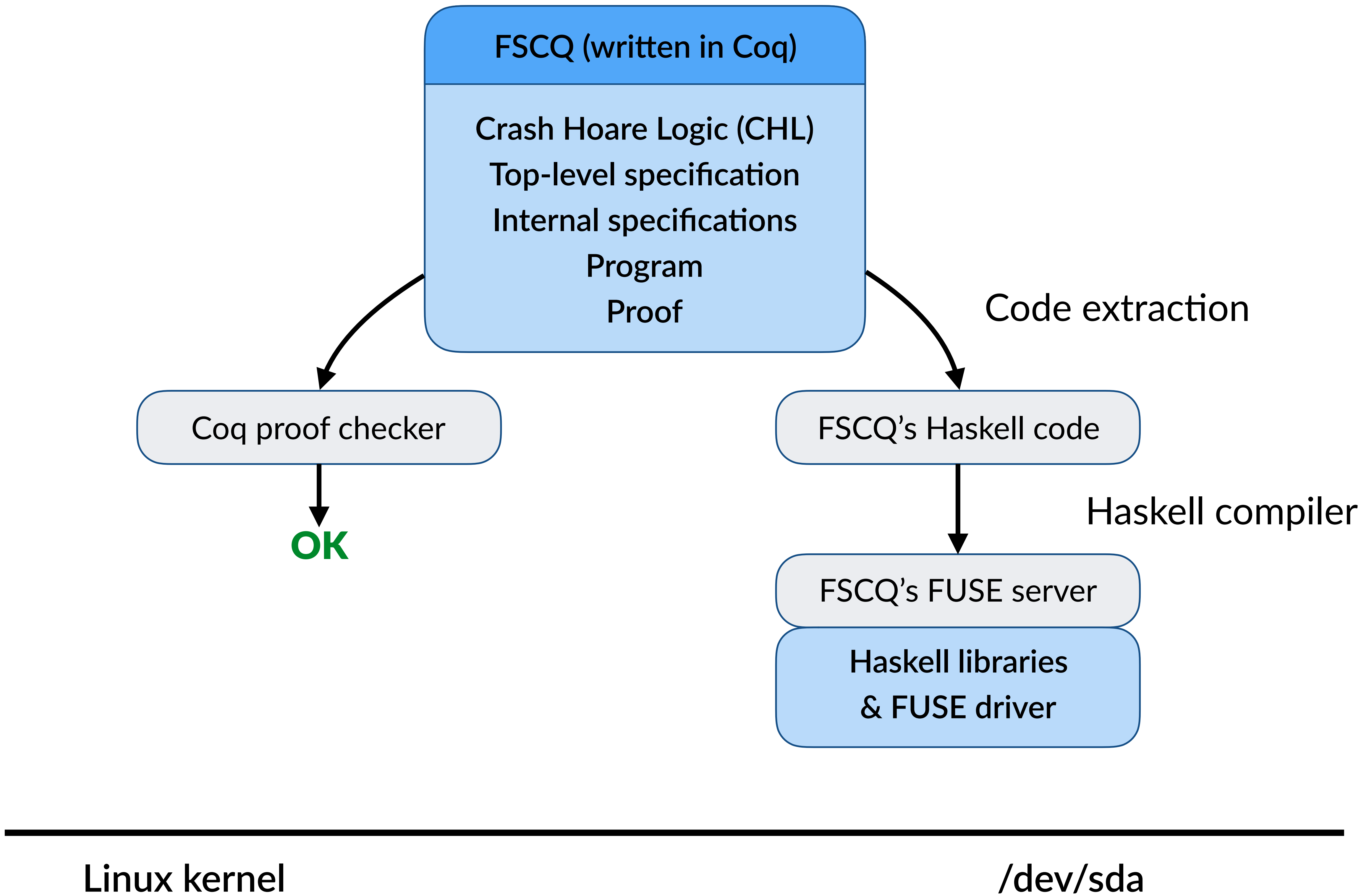
# FSCQ runs standard Unix programs



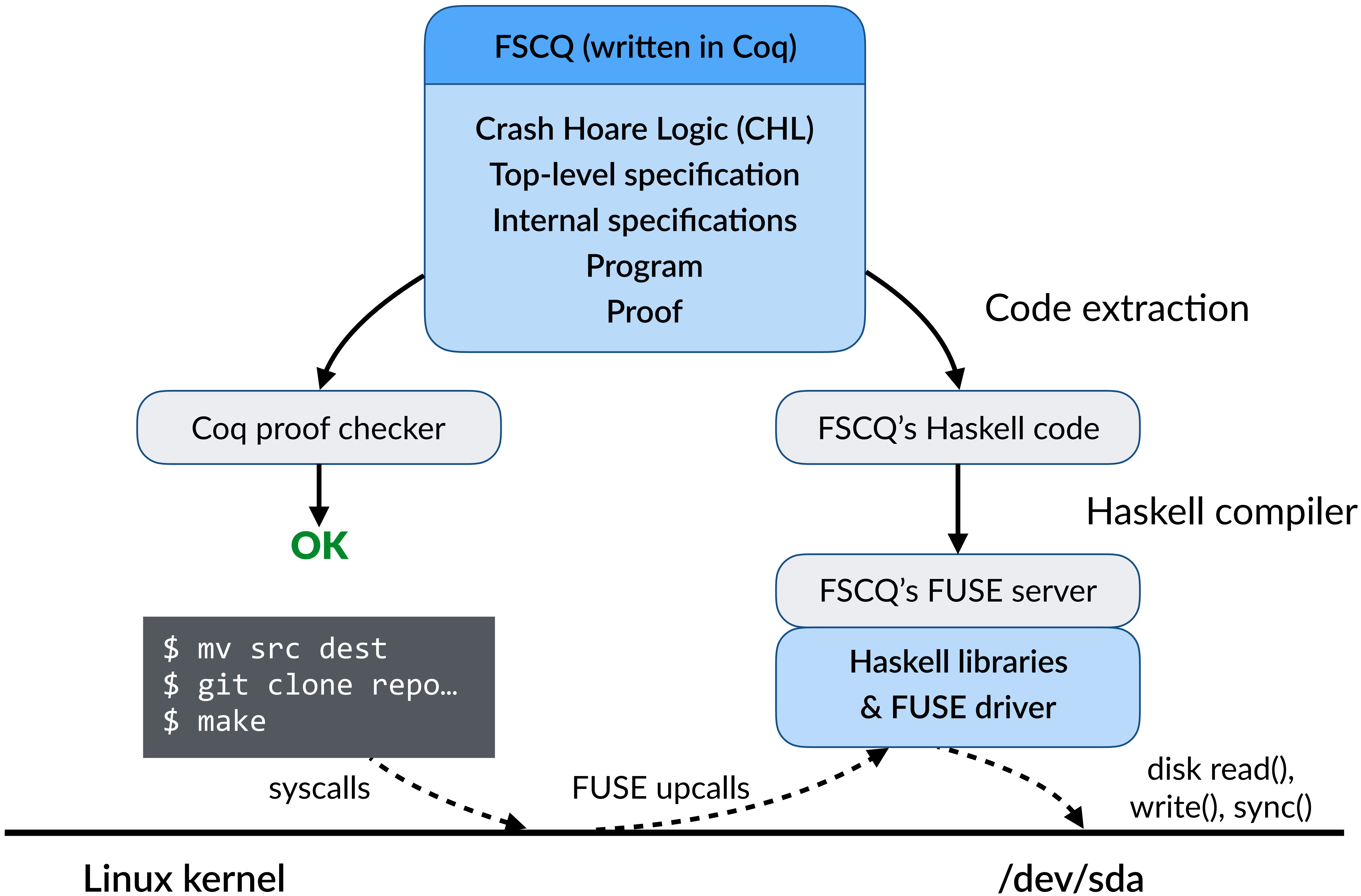
# FSCQ runs standard Unix programs



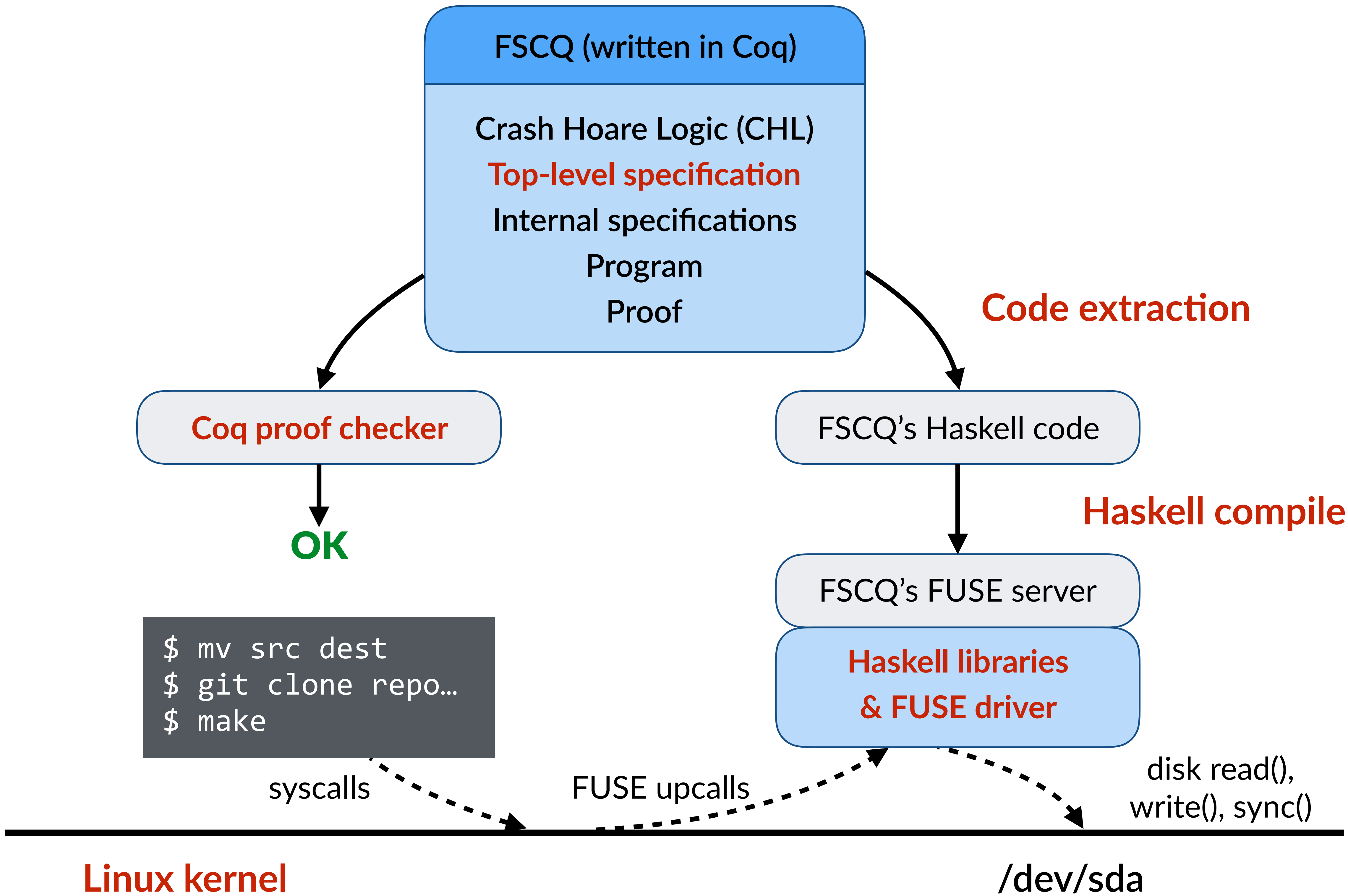
# FSCQ runs standard Unix programs



# FSCQ runs standard Unix programs



# FSCQ's Trusted Computing Base



# Outline

- Crash safety
  - What is the correct behavior after a crash?
- Challenge 1: formalizing crashes
  - Crash Hoare Logic (CHL)
- Challenge 2: incorporating performance optimizations
  - Disk sequences
- Building a complete file system
- Evaluation

# What is **crash safety**?

- What guarantee should file system provide when it crashes and reboot?
- Look it up in the POSIX standard?

# POSIX is vague about crash behavior

*[…] a power failure […] can cause data to be lost. The data may be associated with a file that is still open, with one that has been closed, with a directory, or with any other internal system data structures associated with permanent storage. This data can be lost, in whole or part, so that only careful inspection of file contents could determine that an update did not occur.*

IEEE Std 1003.1, 2013 Edition

# POSIX is vague about crash behavior

*[...] a power failure [...] can cause data to be lost. The data may be associated with a file that is still open, with one that has been closed, with a directory, or with any other internal system data structures associated with permanent storage. This data can be lost, in whole or part, so that only careful inspection of file contents could determine that an update did not occur.*

IEEE Std 1003.1, 2013 Edition

- POSIX's goal was to specify "common-denominator" behavior
- Gives freedom to file systems to implement their own optimizations

# What is **crash safety**?

- What guarantee should file system provide when it crashes and reboot?
- ~~Look it up in the POSIX standard?~~ (Too Vague)

# What is **crash safety**?

- What guarantee should file system provide when it crashes and reboot?
- ~~Look it up in the POSIX standard?~~ (Too Vague)
- A simple and useful definition is **transactional**
  - **Atomicity**: every file-system call is all-or-nothing
  - **Durability**: every call persists on disk when it returns

# What is **crash safety**?

- What guarantee should file system provide when it crashes and reboot?
- ~~Look it up in the POSIX standard?~~ (Too Vague)
- A simple and useful definition is **transactional**
  - **Atomicity**: every file-system call is all-or-nothing
  - **Durability**: every call persists on disk when it returns
- Run every file-system call inside a transaction, using **write-ahead logging**.

# Write-ahead logging

Disk

# Write-ahead logging

```
➡  log_begin()
```

# Write-ahead logging

```
➡ log_begin()
➡ log_write(2, ‘a’)
➡ log_write(8, ‘b’)
➡ log_write(5, ‘c’)
```

1. **Append** writes to the log

# Write-ahead logging

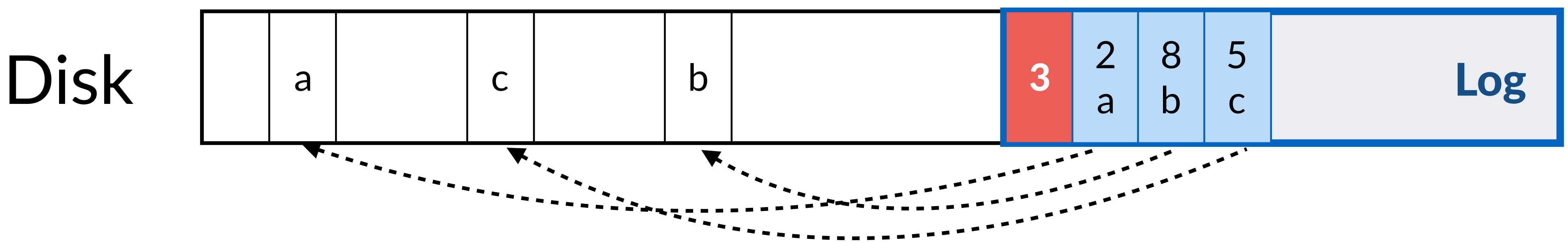
```
➡ log_begin()
➡ log_write(2, ‘a’)
➡ log_write(8, ‘b’)
➡ log_write(5, ‘c’)
➡ log_commit()
```

1. Append writes to the log
2. Set commit record

# Write-ahead logging

```
➡ log_begin()
➡ log_write(2, ‘a’)
➡ log_write(8, ‘b’)
➡ log_write(5, ‘c’)
➡ log_commit()
```

1. **Append** writes to the log
2. **Set commit record**
3. **Apply** the log to disk locations



# Write-ahead logging

```
➡ log_begin()
➡ log_write(2, 'a')
➡ log_write(8, 'b')
➡ log_write(5, 'c')
➡ log_commit()
```

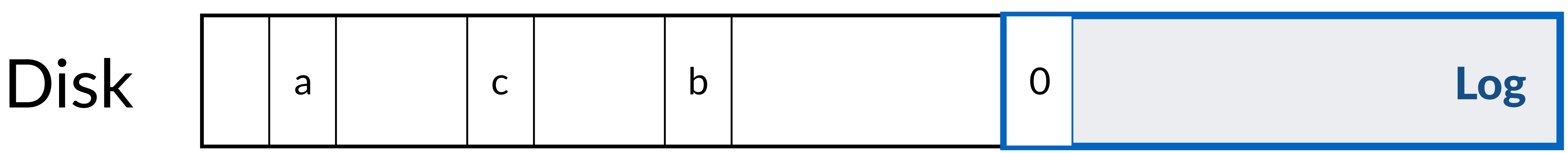
1. **Append** writes to the log
2. **Set commit record**
3. **Apply** the log to disk locations
4. **Truncate** the log

Disk

# Write-ahead logging

```
➡ log_begin()
➡ log_write(2, 'a')
➡ log_write(8, 'b')
➡ log_write(5, 'c')
➡ log_commit()
```

1. **Append** writes to the log
2. **Set commit record**
3. **Apply** the log to disk locations
4. **Truncate** the log

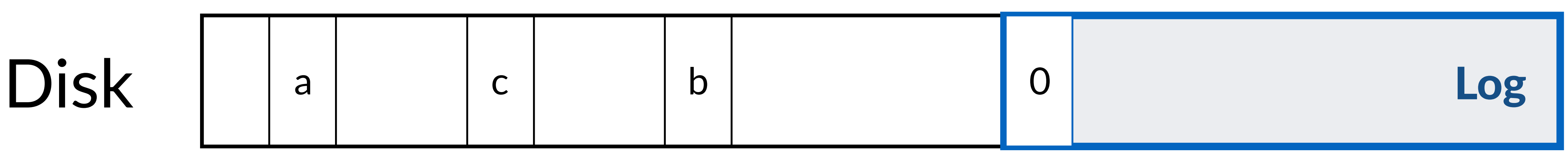



- **Recovery**: after crash, replay (apply) any **committed** transaction in the log

# Write-ahead logging

```
➡ log_begin()
➡ log_write(2, ‘a’)
➡ log_write(8, ‘b’)
➡ log_write(5, ‘c’)
➡ log_commit()
```

1. **Append** writes to the log
2. **Set commit record**
3. **Apply** the log to disk locations
4. **Truncate** the log




- **Recovery**: after crash, replay (apply) any **committed** transaction in the log
- **Atomicity**: either all writes appear on disk or none do

# Write-ahead logging

```
➡ log_begin()
➡ log_write(2, ‘a’)
➡ log_write(8, ‘b’)
➡ log_write(5, ‘c’)
➡ log_commit()
```

1. **Append** writes to the log
2. **Set commit record**
3. **Apply** the log to disk locations
4. **Truncate** the log

Disk


- **Recovery**: after crash, replay (apply) any **committed** transaction in the log
- **Atomicity**: either all writes appear on disk or none do
- **Durability**: all changes are persisted on disk when log_commit() returns

# Example: transactional crash safety

```
def create(dir, name):
    log_begin()
    newfile = allocate_inode()
    newfile.init()
    dir.add(name, newfile)
    log_commit()
```

# Example: transactional crash safety

```
def create(dir, name):
    log_begin()
    newfile = allocate_inode()
    newfile.init()
    dir.add(name, newfile)
    log_commit()
```

**… after crash …**

```
def log_recover():
    if committed:
        log_apply()
        log_truncate()
```

# Example: transactional crash safety

```
def create(dir, name):
    log_begin()
    newfile = allocate_inode()
    newfile.init()
    dir.add(name, newfile)
    log_commit()
```

**... after crash ...**

```
def log_recover():
    if committed:
        log_apply()
        log_truncate()
```

- Q: How to formally define what happens when the computer crashes?

# Example: transactional crash safety

```
def create(dir, name):
    log_begin()
    newfile = allocate_inode()
    newfile.init()
    dir.add(name, newfile)
    log_commit()
```

**... after crash ...**

```
def log_recover():
    if committed:
        log_apply()
        log_truncate()
```

- Q: How to formally define what happens when the computer crashes?
- Q: How to formally specify the behavior of “create” in presence of crash and recovery?

# Approach: Crash Hoare Logic

{pre} code {post}

| | |
|---|---|
| SPEC | disk_write $(a, v)$ |
| PRE | $a \mapsto v_0$ |
| POST | $a \mapsto v$ |

# Approach: Crash Hoare Logic

{pre} code {post}
{crash}

| | |
|---|---|
| SPEC | disk_write $(a, v)$ |
| PRE | $a \mapsto v_0$ |
| POST | $a \mapsto v$ |
| CRASH | $a \mapsto v_0 \ \vee \ a \mapsto v$ |

- **Crash condition**: all intermediate disk states (plus two end-states)

# Approach: Crash Hoare Logic

{pre} code {post}
{crash}

| | |
|---|---|
| SPEC | disk_write $(a, v)$ |
| PRE | $a \mapsto v_0$ |
| POST | $a \mapsto v$ |
| CRASH | $a \mapsto v_0 \ \vee \ a \mapsto v$ |

- **Crash condition**: all intermediate disk states (plus two end-states)
- CHL's disk model matches what most other file systems assume:
  - Writing a single block is an atomic operation, no data corruption

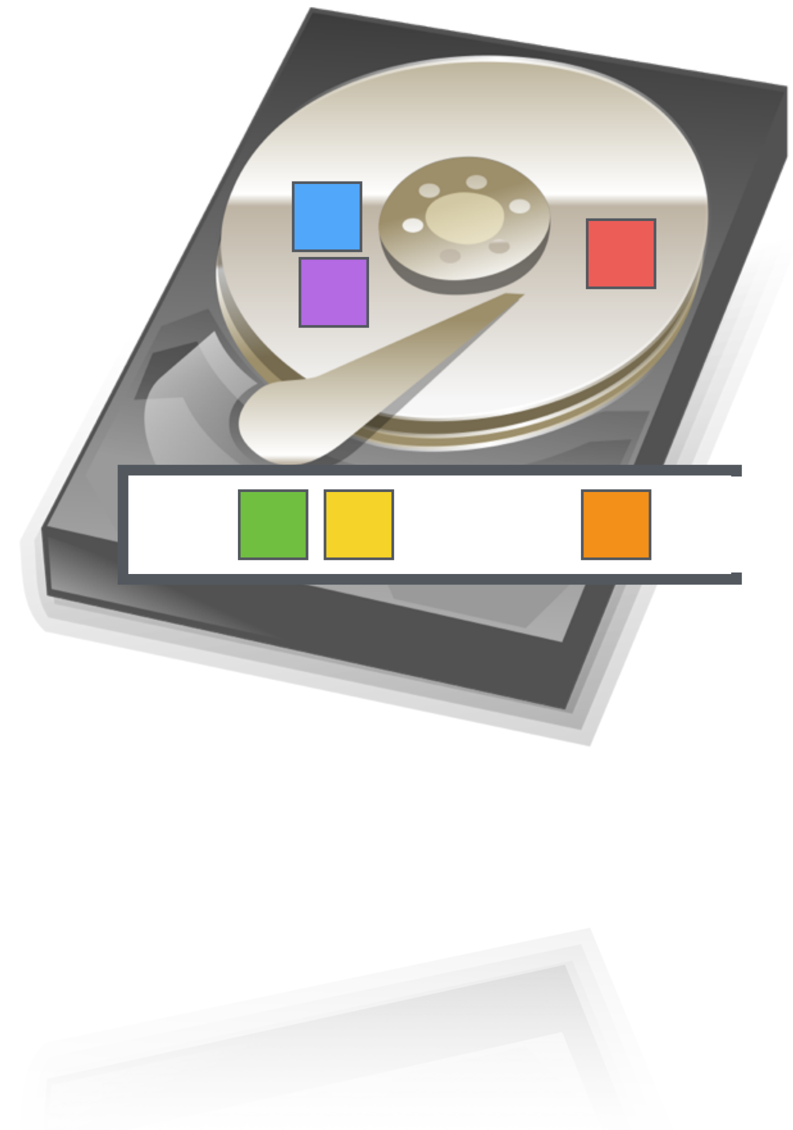
# Asynchronous disk I/O

# Asynchronous disk I/O

- For performance, hard drive caches writes in its internal volatile buffer
  - Writes **do not persist** immediately

# Asynchronous disk I/O

- For performance, hard drive caches writes in its internal volatile buffer
  - Writes **do not persist** immediately
- Disk flushes the buffer to media in background
  - Writes might be **reordered**



# Asynchronous disk I/O

- For performance, hard drive caches writes in its internal volatile buffer
  - Writes **do not persist** immediately
- Disk flushes the buffer to media in background
  - Writes might be **reordered**
- Use **write barrier** (**disk_sync**) to force flushing the buffer
  - Make data persistent & enforce ordering
  - Disk syncs are expensive!

# Formalizing asynchronous disk I/O

- **Challenge**: when crashes, the disk might lose **some** of the recent writes

# Formalizing asynchronous disk I/O

- **Challenge**: when crashes, the disk might lose **some** of the recent writes

**Q:** What are the possible disk states if crashing after the 3 writes?

```
a ⟼ 0, b ⟼ 0
disk_write(a, 1)
disk_write(b, 2)
disk_write(a, 3)
```

# Formalizing asynchronous disk I/O

- **Challenge**: when crashes, the disk might lose **some** of the recent writes

  **Q:** What are the possible disk states if crashing after the 3 writes?

  **A:** 6 cases: $a \longmapsto 0$ or $1$ or $3$, $b \longmapsto 0$ or $2$

```
a ⟼ 0, b ⟼ 0
disk_write(a, 1)
disk_write(b, 2)
disk_write(a, 3)
```

# Formalizing asynchronous disk I/O

- **Challenge**: when crashes, the disk might lose **some** of the recent writes

**Q:** What are the possible disk states if crashing after the 3 writes?

**A:** 6 cases: $a \longmapsto 0$ or $1$ or $3$, $b \longmapsto 0$ or $2$

```
a ⟼ 0, b ⟼ 0
disk_write(a, 1)
disk_write(b, 2)
disk_write(a, 3)
```

- **Idea**: use **value-sets**: $a \longmapsto \langle v_0, vs \rangle$
  - **Read** returns the latest value:
  - **Write** adds a value to the set:
  - **Sync** discards previous values:
  - **Reboot** chooses a random value:

# Formalizing asynchronous disk I/O

- **Challenge**: when crashes, the disk might lose **some** of the recent writes

  **Q:** What are the possible disk states if crashing after the 3 writes?

  **A:** 6 cases: $a \longmapsto 0$ or $1$ or $3$, $b \longmapsto 0$ or $2$

```
a ⟼ 0, b ⟼ 0
disk_write(a, 1)
disk_write(b, 2)
disk_write(a, 3)
```

- **Idea**: use **value-sets**: $a \longmapsto \langle v_0, vs \rangle$
  - **Read** returns the latest value: $v_0$
  - **Write** adds a value to the set:
  - **Sync** discards previous values:
  - **Reboot** chooses a random value:

# Formalizing asynchronous disk I/O

- **Challenge**: when crashes, the disk might lose **some** of the recent writes

**Q:** What are the possible disk states if crashing after the 3 writes?

**A:** 6 cases: $a \longmapsto 0$ or $1$ or $3$, $b \longmapsto 0$ or $2$

```
a ⟼ 0, b ⟼ 0
disk_write(a, 1)
disk_write(b, 2)
disk_write(a, 3)
```

- **Idea**: use **value-sets**: $a \mapsto \langle v_0, vs \rangle$
  - **Read** returns the latest value: $v_0$
  - **Write** adds a value to the set: $a \mapsto \langle v, \{v_0\} \cup vs \rangle$
  - **Sync** discards previous values:
  - **Reboot** chooses a random value:

# Formalizing asynchronous disk I/O

- **Challenge**: when crashes, the disk might lose **some** of the recent writes

**Q:** What are the possible disk states if crashing after the 3 writes?

**A:** 6 cases: $a \longmapsto 0$ or $1$ or $3$, $b \longmapsto 0$ or $2$

```
a ⟼ 0, b ⟼ 0
disk_write(a, 1)
disk_write(b, 2)
disk_write(a, 3)
```

- **Idea**: use **value-sets**: $a \mapsto \langle v_0, vs \rangle$
  - **Read** returns the latest value: $v_0$
  - **Write** adds a value to the set: $a \mapsto \langle v, \{v_0\} \cup vs \rangle$
  - **Sync** discards previous values: $a \mapsto \langle v_0, \varnothing \rangle$
  - **Reboot** chooses a random value:

# Formalizing asynchronous disk I/O

- **Challenge**: when crashes, the disk might lose **some** of the recent writes

  **Q:** What are the possible disk states if crashing after the 3 writes?

  **A:** 6 cases: $a \longmapsto 0$ or $1$ or $3$, $b \longmapsto 0$ or $2$

```
a ⟼ 0, b ⟼ 0
disk_write(a, 1)
disk_write(b, 2)
disk_write(a, 3)
```

- **Idea**: use **value-sets**: $a \mapsto \langle v_0, vs \rangle$
  - **Read** returns the latest value: $v_0$
  - **Write** adds a value to the set: $a \mapsto \langle v, \{v_0\} \cup vs \rangle$
  - **Sync** discards previous values: $a \mapsto \langle v_0, \emptyset \rangle$
  - **Reboot** chooses a random value: $a \mapsto \langle v', \emptyset \rangle, \; v' \in \{v_0\} \cup vs$

# CHL asynchronous disk model

| | |
|---|---|
| SPEC | disk_write $(a, v)$ |
| PRE | $\textbf{disk} \models a \mapsto \langle v_0, vs \rangle$ |
| POST | $\textbf{disk} \models a \mapsto \langle v, \{v_0\} \cup vs \rangle$ |
| CRASH | $\textbf{disk} \models a \mapsto \langle v_0, vs \rangle \lor$ |
| | $a \mapsto \langle v, \{v_0\} \cup vs \rangle$ |

# CHL asynchronous disk model

| | |
|---|---|
| SPEC | disk_write $(a, v)$ |
| PRE | **disk** $\models a \mapsto \langle v_0, vs \rangle$ |
| POST | **disk** $\models a \mapsto \langle v, \{v_0\} \cup vs \rangle$ |
| CRASH | **disk** $\models a \mapsto \langle v_0, vs \rangle \lor$ |
| | $a \mapsto \langle v, \{v_0\} \cup vs \rangle$ |

- Specifications for **disk_write**, **disk_read**, and **disk_sync** are **axioms**

# CHL asynchronous disk model

| | |
|---|---|
| SPEC | disk_write $(a, v)$ |
| PRE | $\textbf{disk} \models a \mapsto \langle v_0, vs \rangle$ |
| POST | $\textbf{disk} \models a \mapsto \langle v, \{v_0\} \cup vs \rangle$ |
| CRASH | $\textbf{disk} \models a \mapsto \langle v_0, vs \rangle \vee$ |
| | $a \mapsto \langle v, \{v_0\} \cup vs \rangle$ |

- Specifications for **disk_write**, **disk_read**, and **disk_sync** are **axioms**
- "**disk** |= …" means the **disk address space entails** the predicate

# Abstraction layers

- Each abstraction layer forms an **address space**

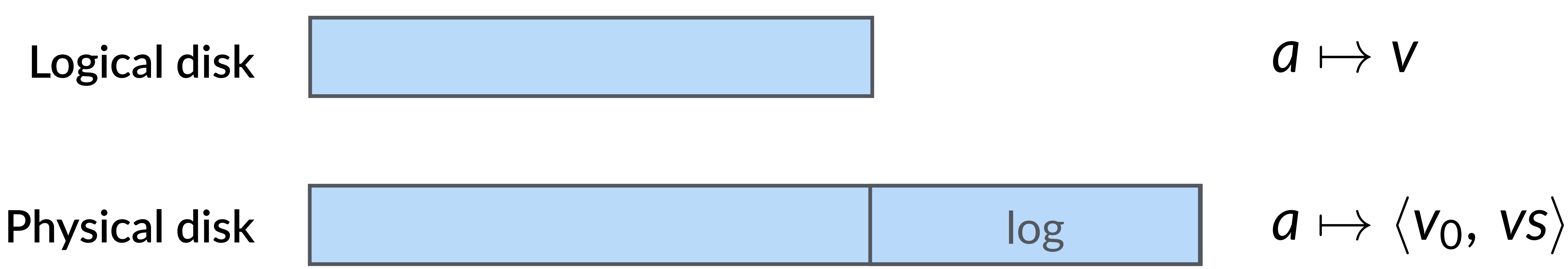
# Abstraction layers

- Each abstraction layer forms an **address space**


$a \mapsto \langle v_0, \ vs \rangle$

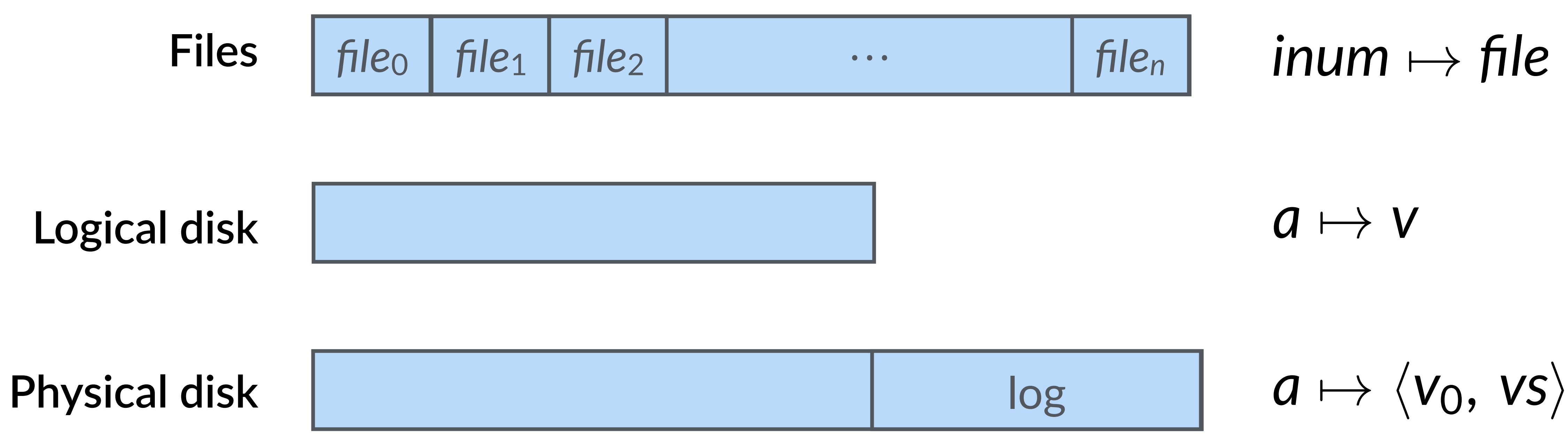
# Abstraction layers

- Each abstraction layer forms an **address space**



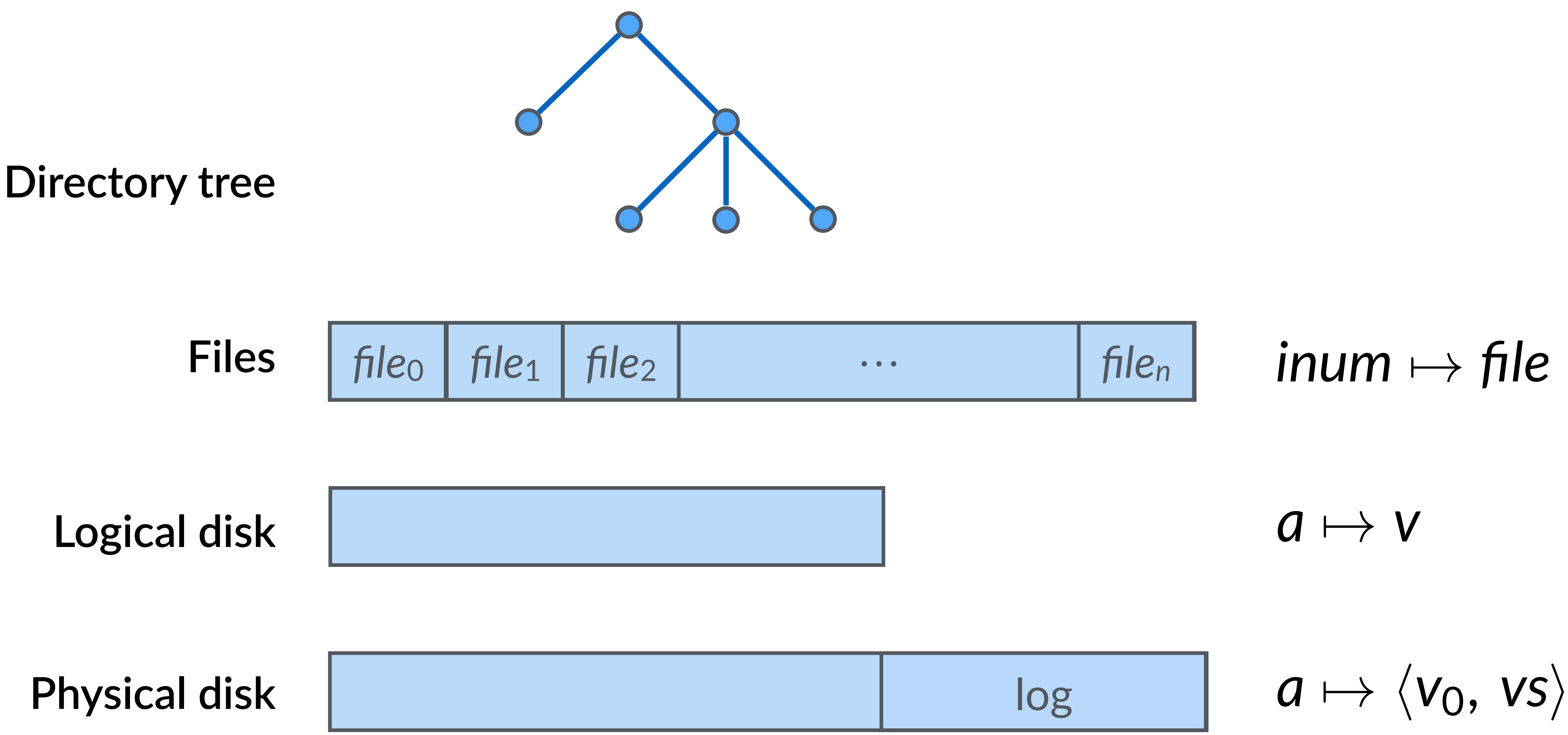
# Abstraction layers

- Each abstraction layer forms an **address space**



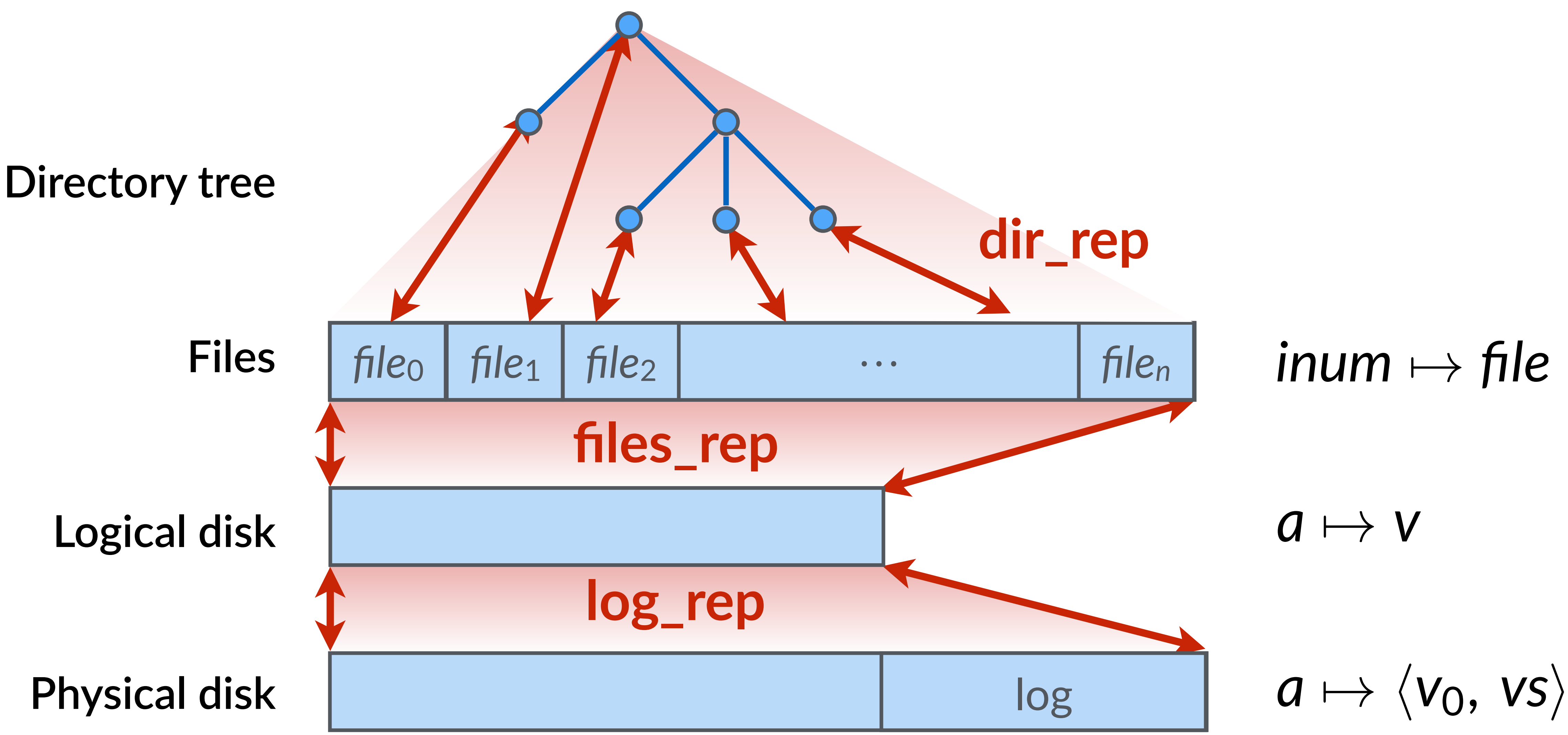
# Abstraction layers

- Each abstraction layer forms an **address space**



# Abstraction layers

- Each abstraction layer forms an **address space**
- **Representation invariants** connect logical states between layers



# Example: representation invariant

SPEC log_write $(a, v)$

PRE

**old_state** $\models a \mapsto v_0$

POST

**new_state** $\models a \mapsto v$

- **old_state** and **new_state** are "logical disks" exposed by the logging system

# Example: representation invariant

| | |
|---|---|
| SPEC | log_write $(a, v)$ |
| PRE | **disk** $\models$ **log_rep** (ActiveTxn, *start_state*, *old_state*) |
| | **old_state** $\models a \mapsto v_0$ |
| POST | **disk** $\models$ **log_rep** (ActiveTxn, *start_state*, *new_state*) |
| | **new_state** $\models a \mapsto v$ |
| CRASH | **disk** $\models$ **log_rep** (ActiveTxn, *start_state*, *any_state*) |

- **old_state** and **new_state** are "logical disks" exposed by the logging system
- **log_rep** connects transaction state to an on-disk representation
- Describes the log's on-disk layout using many $\longmapsto$ primitives

# Certifying procedures

- **bmap**: return the block address at a given offset for an inode

```
def bmap(inode, bnum):
    if bnum >= NDIRECT:
        indirect = log_read(inode.blocks[NDIRECT])
        return indirect[bnum - NDIRECT]
    else:
        return inode.blocks[bnum]
```

# Certifying procedures

- **bmap**: return the block address at a given offset for an inode

```
def bmap(inode, bnum):
    if bnum >= NDIRECT:
        indirect = log_read(inode.blocks[NDIRECT])
        return indirect[bnum - NDIRECT]
    else:
        return inode.blocks[bnum]
```

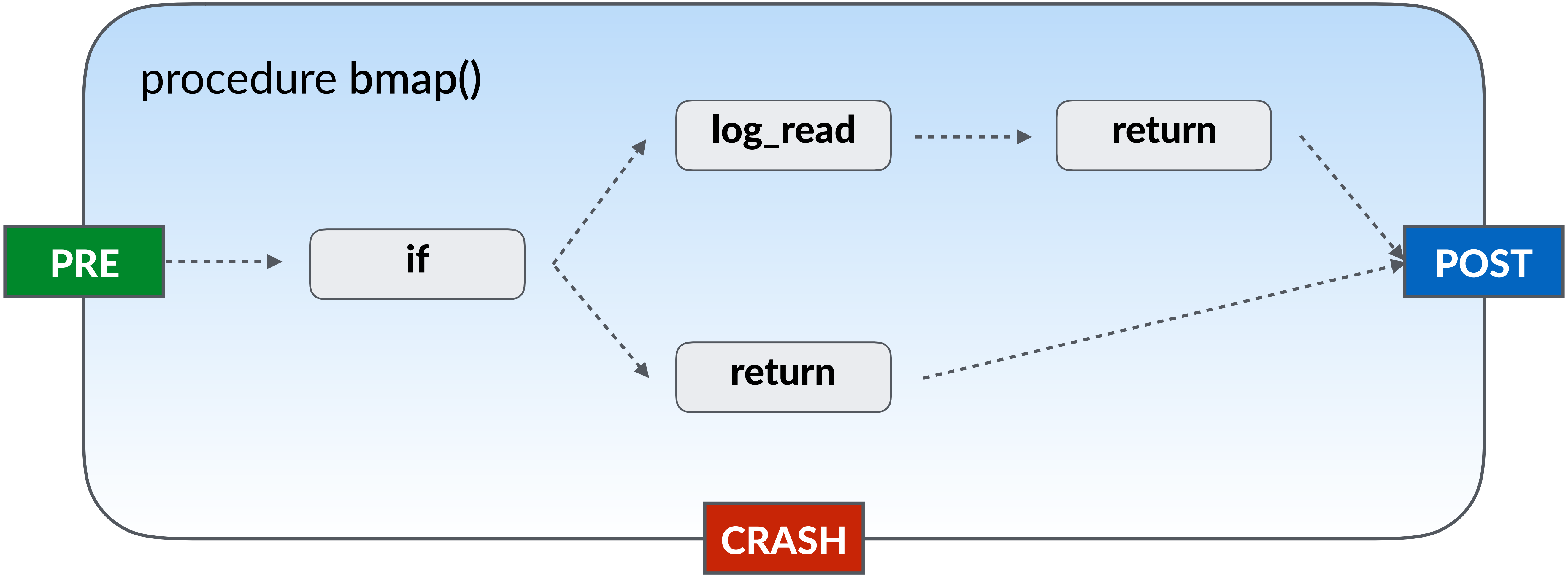
PRE

POST

CRASH

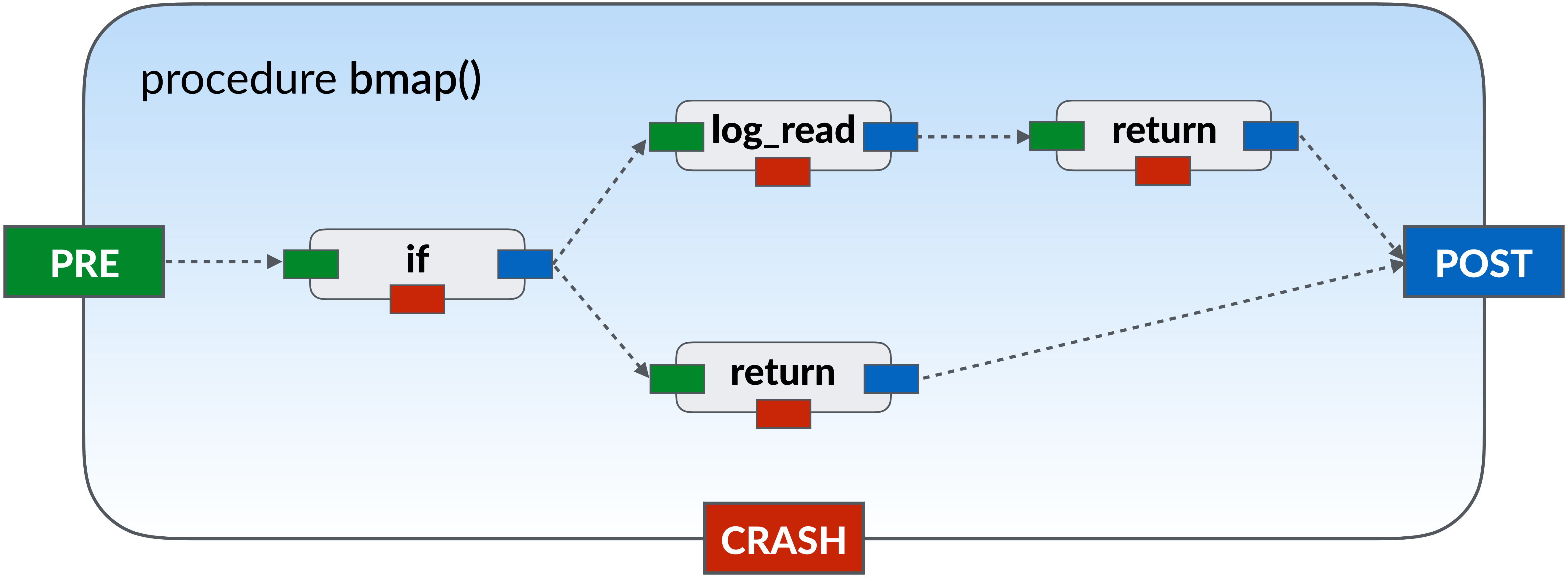
# Certifying procedures

- Follow the control flow graph



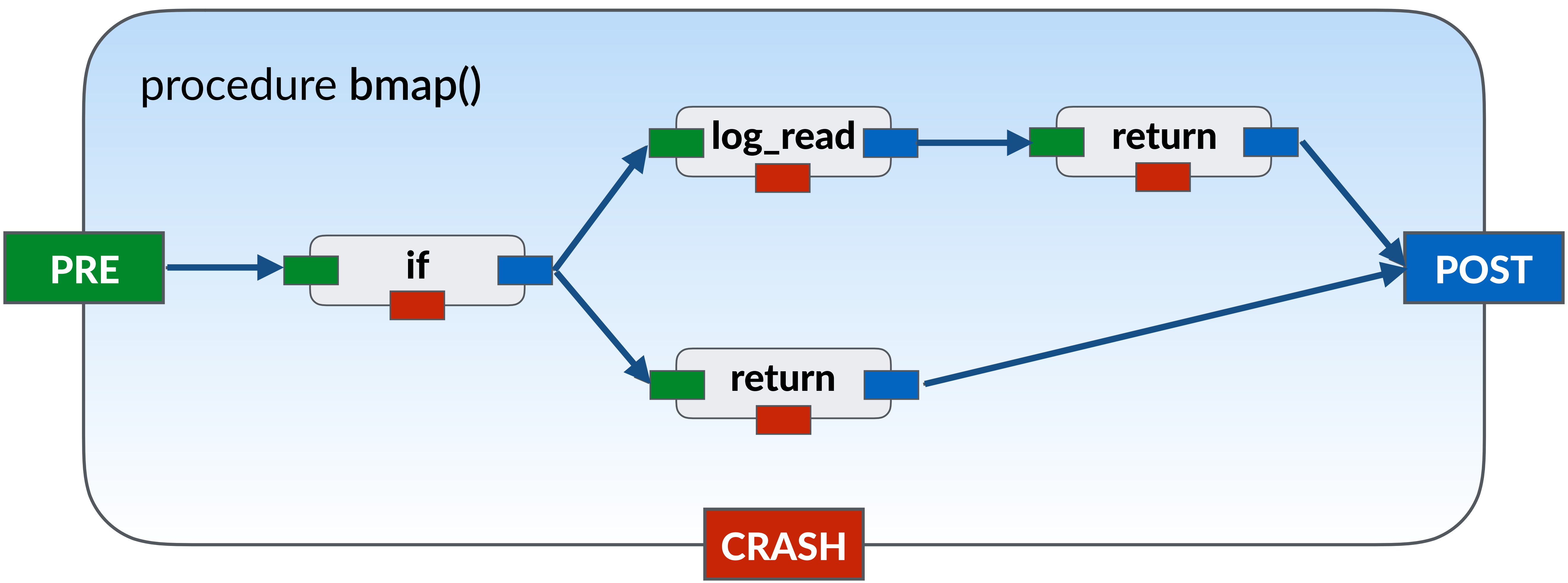
# Certifying procedures

- Follow the control flow graph
- Need pre/post/crash conditions for each called procedure



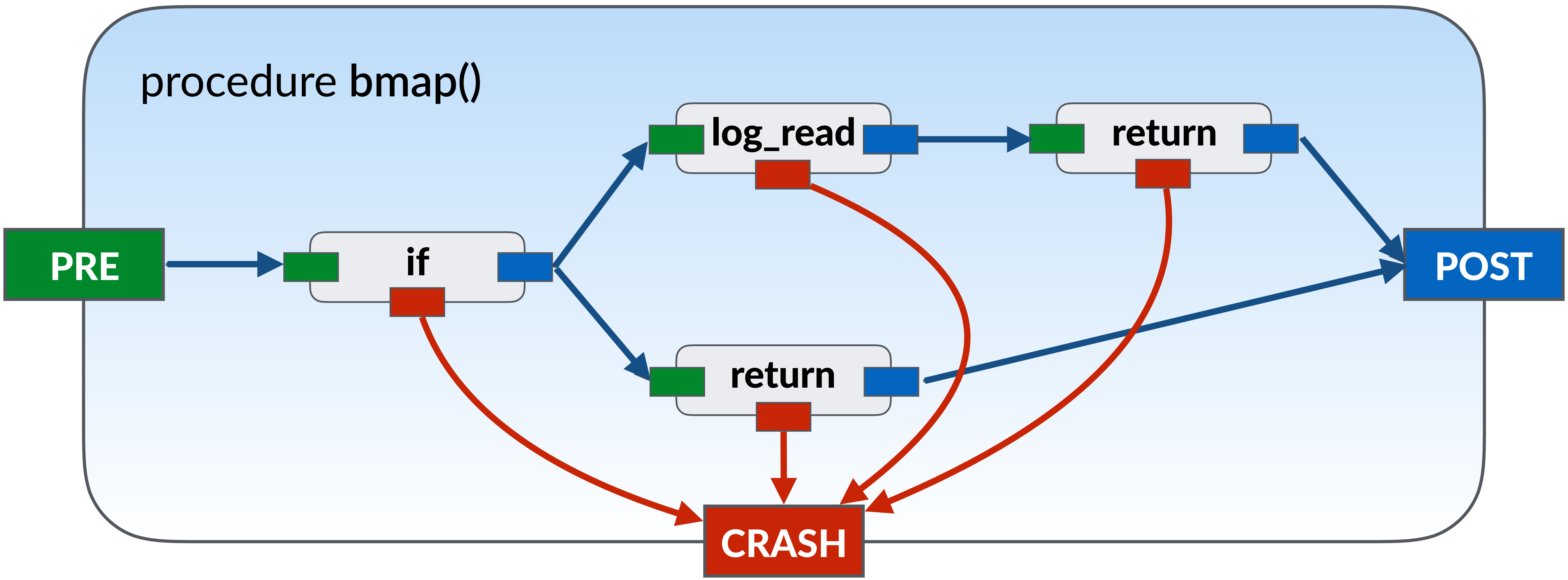
# Certifying procedures

- Follow the control flow graph
- Need pre/post/crash conditions for each called procedure
- Chain pre- and postconditions, forming **proof obligations**



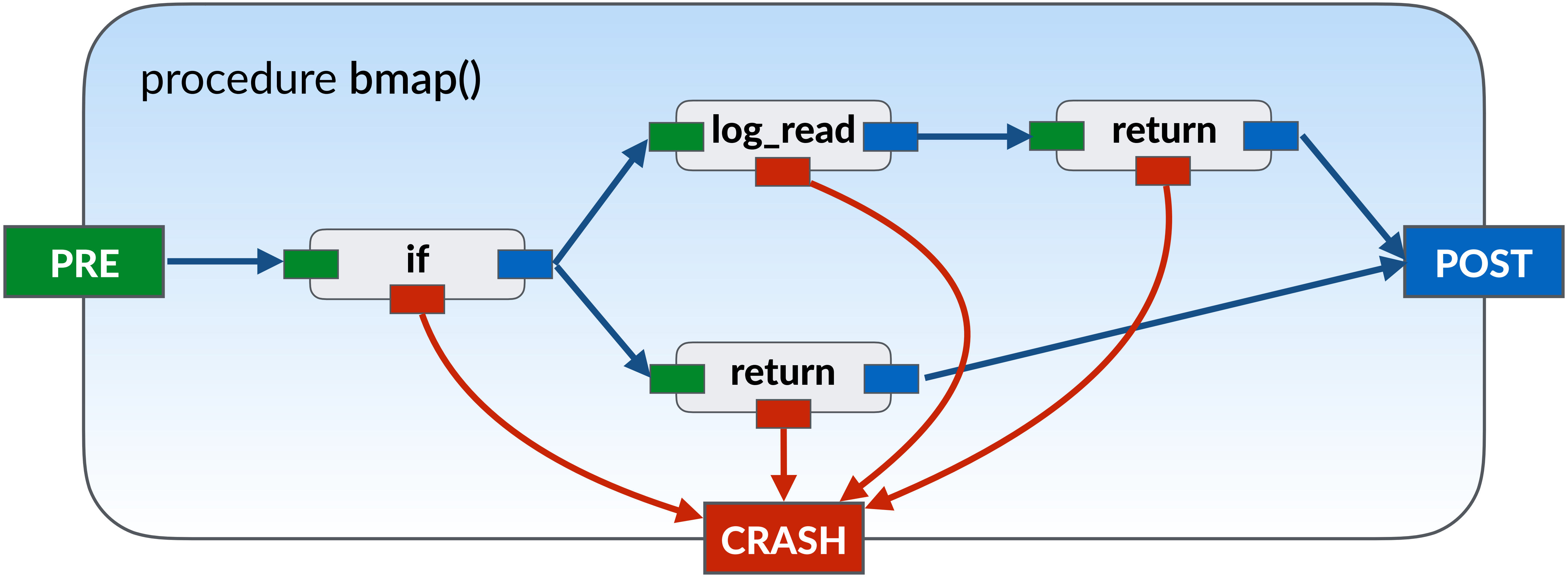
# Certifying procedures

- Follow the control flow graph
- Need pre/post/crash conditions for each called procedure
- Chain pre- and postconditions, forming **proof obligations** →
- **CHL:** combines crash conditions, get more **proof obligations** →



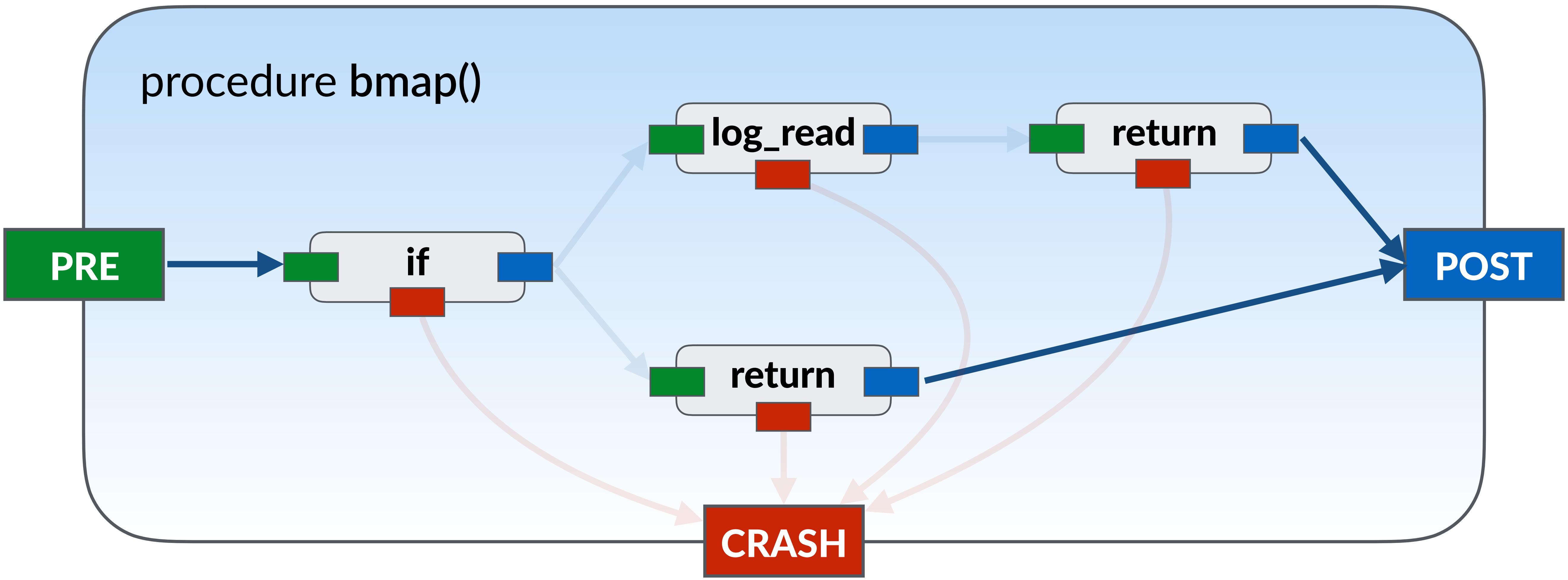
# Proof automation

- CHL follows the CFG, and generates proof obligations



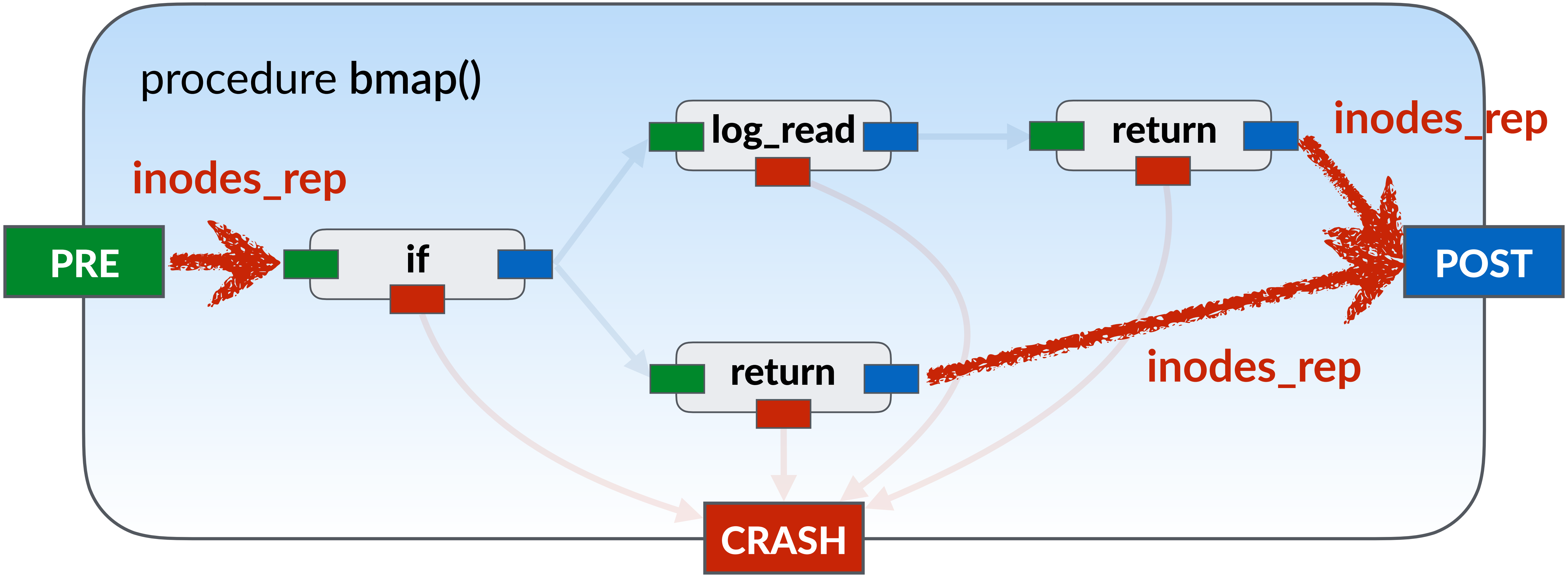
# Proof automation

- CHL follows the CFG, and generates proof obligations
- CHL solves trivial obligations automatically (common case)



# Proof automation

- CHL follows the CFG, and generates proof obligations
- CHL solves trivial obligations automatically (common case)
- Remaining proof effort: changing **representation invariants**
  - Show that rep invariant holds at entry and exit



# Specifying an entire system call (simplified)

SPEC create $(dnum,\ fn)$

PRE **disk** $\models$ log_rep(NoTxn, *start_state*)

**start_state** $\models$ dir_rep(*tree*) $\wedge$

$\exists\, path,\ tree[path].node = dnum \wedge$

$fn \notin tree[path]$

# Specifying an entire system call (simplified)

SPEC create ($dnum$, $fn$)

PRE $\textbf{disk} \models \text{log\_rep}(\text{NoTxn}, start\_state)$

$\textbf{start\_state} \models \text{dir\_rep}(tree) \wedge$

$\exists\, path,\ tree[path].node = dnum \wedge$

$fn \notin tree[path]$

POST $\textbf{disk} \models \text{log\_rep}(\text{NoTxn}, new\_state)$

$\textbf{new\_state} \models \text{dir\_rep}(new\_tree) \wedge$

$new\_tree = tree.\text{update}(path,\ fn,\ \text{EmptyFile})$

# Specifying an entire system call (simplified)

SPEC create $(dnum,\ fn)$

PRE **disk** $\models$ log_rep(NoTxn, $start\_state$)

**start_state** $\models$ dir_rep($tree$) $\wedge$

$\exists\, path,\ tree[path].node = dnum\ \wedge$

$fn \notin tree[path]$

POST **disk** $\models$ log_rep(NoTxn, $new\_state$)

**new_state** $\models$ dir_rep($new\_tree$) $\wedge$

$new\_tree = tree$.update($path,\ fn,$ EmptyFile)

CRASH **disk** $\models$ log_rep(NoTxn, $start\_state$) $\vee$

log_rep(NoTxn, $new\_state$) $\vee$

log_rep(ActiveTxn, $start\_state,\ any\_state$) $\vee$

log_rep(CommittingTxn, $start\_state,\ new\_state$)

# Specifying an entire system call (simplified)

SPEC create $(\mathit{dnum}, \mathit{fn})$

PRE
**disk** $\models$ log_rep(NoTxn, *start_state*)
**start_state** $\models$ dir_rep(*tree*) $\wedge$
$\exists\, \mathit{path},\ \mathit{tree}[\mathit{path}].\mathit{node} = \mathit{dnum} \wedge$
$\mathit{fn} \notin \mathit{tree}[\mathit{path}]$

POST
**disk** $\models$ log_rep(NoTxn, *new_state*)
**new_state** $\models$ dir_rep(*new_tree*) $\wedge$
*new_tree* = *tree*.update(*path*, *fn*, EmptyFile)

CRASH
**disk** $\models$ log_rep(NoTxn, *start_state*) $\vee$
log_rep(NoTxn, *new_state*) $\vee$
log_rep(ActiveTxn, *start_state*, *any_state*) $\vee$
log_rep(CommittingTxn, *start_state*, *new_state*)

↓

**would_recover_either** (*start_state*, *new_state*)

# Specifying an entire system call (simplified)

SPEC create ($dnum$, $fn$)

PRE **disk** $\models$ log_rep(NoTxn, $start\_state$)

**start_state** $\models$ dir_rep($tree$) $\wedge$

$\exists\, path,\ tree[path].node = dnum \wedge$

$fn \notin tree[path]$

POST **disk** $\models$ log_rep(NoTxn, $new\_state$)

**new_state** $\models$ dir_rep($new\_tree$) $\wedge$

$new\_tree = tree$.update($path$, $fn$, EmptyFile)

CRASH **disk** $\models$ **would_recover_either** ($start\_state$, $new\_state$)

# Specifying log recovery

SPEC log_recover ()

PRE **disk** $\models$ **would_recover_either** (*last_state*, *committed_state*)

POST **disk** $\models$ log_rep(NoTxn, *last_state*) $\vee$
log_rep(NoTxn, *committed_state*)

CRASH **disk** $\models$ **would_recover_either** (*last_state*, *committed_state*)

# Specifying log recovery

SPEC log_recover ()

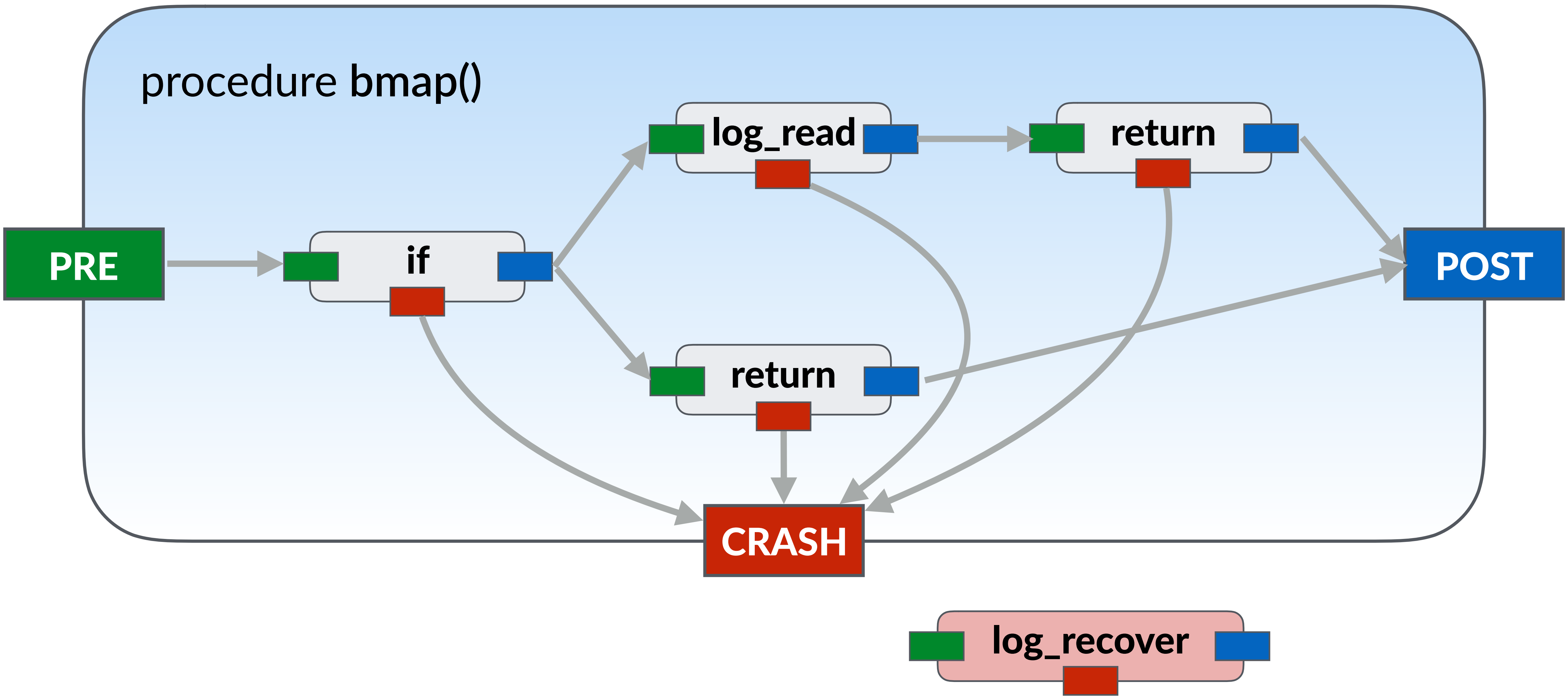
PRE **disk** $\models$ **would_recover_either** (*last_state*, *committed_state*)

POST **disk** $\models$ log_rep(NoTxn, *last_state*) $\vee$
log_rep(NoTxn, *committed_state*)

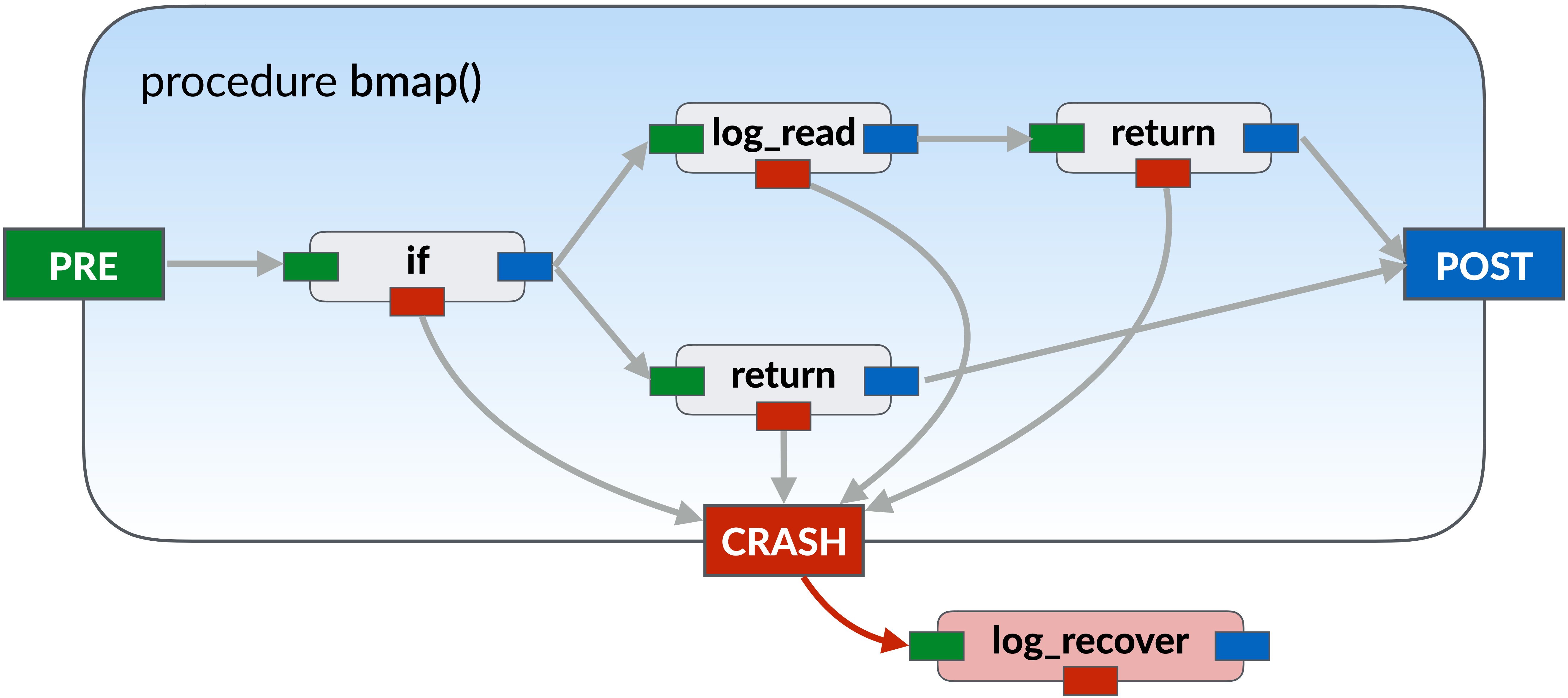
CRASH **disk** $\models$ **would_recover_either** (*last_state*, *committed_state*)

- **log_recover()** is **idempotent**:
  - Crash condition implies its own precondition
  - OK to run **log_recover()** **again** after a crash in itself

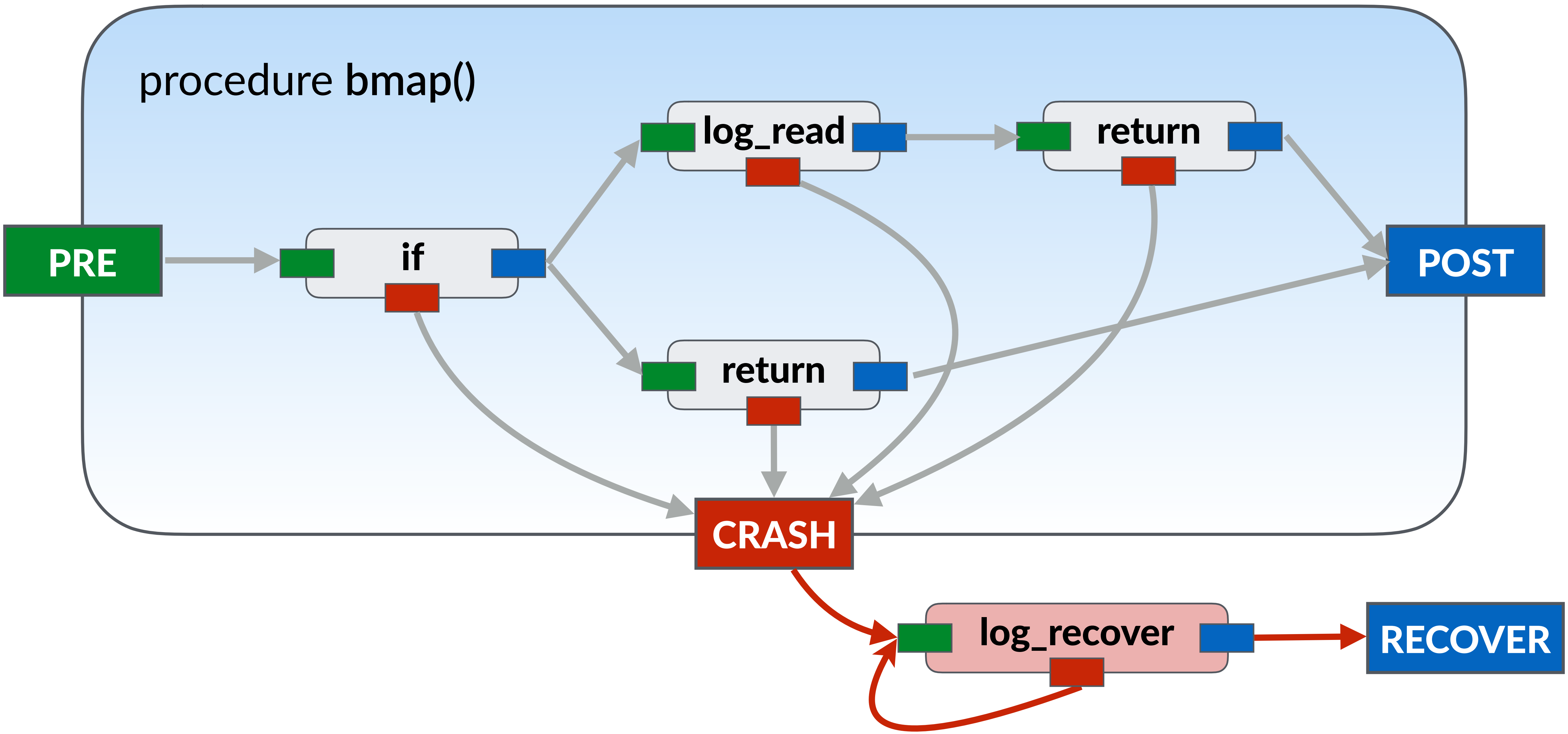
# Recovery execution semantics



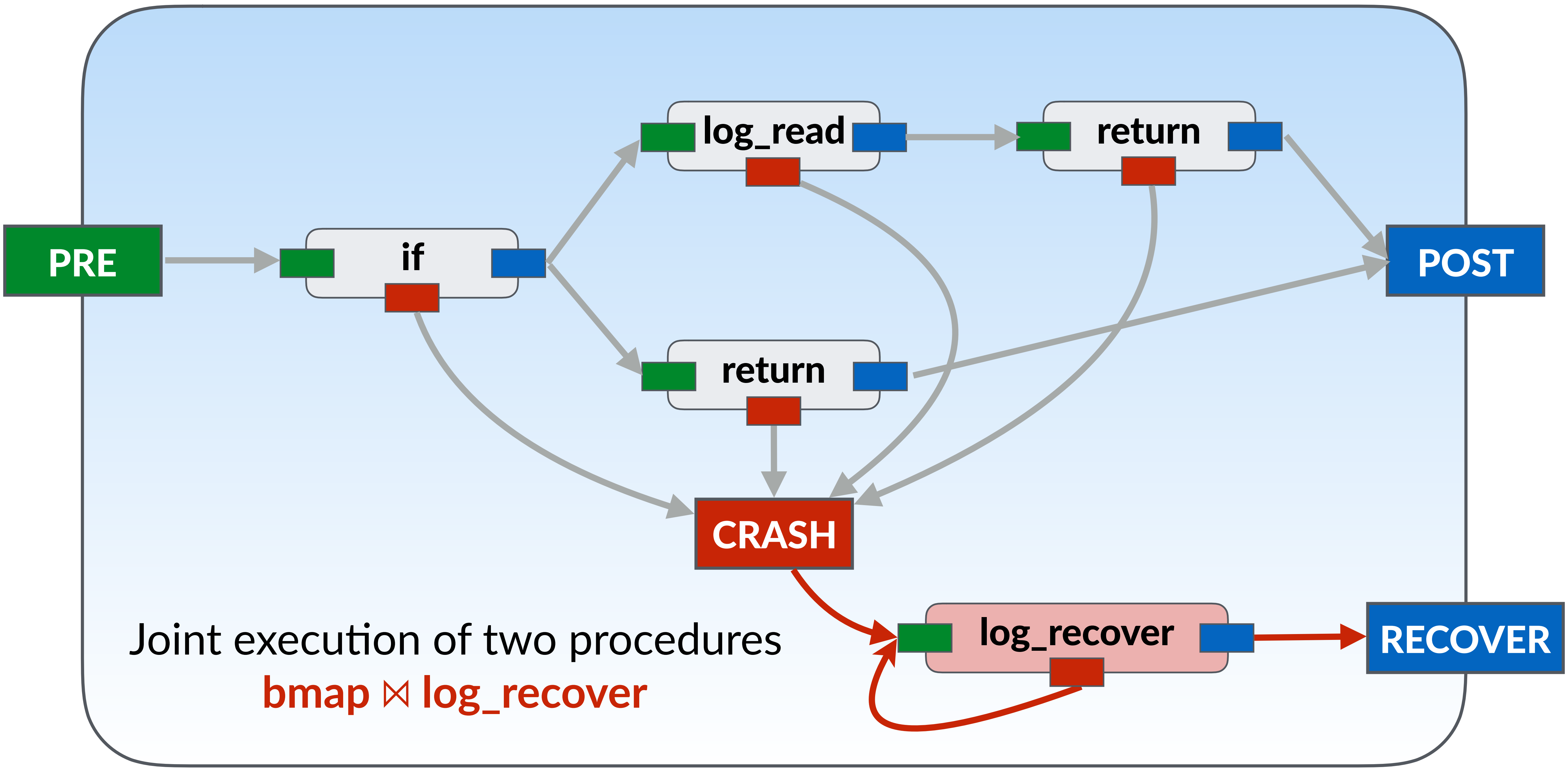
# Recovery execution semantics



# Recovery execution semantics



# Recovery execution semantics




- Whenever **bmap (or log_recover)** crashes, run **log_recover** after reboot

# End-to-end specification

| | |
|---|---|
| SPEC | create $(drum,\ fn)\ \bowtie$ **log_recover** () |
| PRE | **disk** $\models$ log_rep(NoTxn, $start\_state$) |
| | **start_state** $\models$ dir_rep($tree$) $\wedge$ |
| | $\exists\ path,\ tree[path].node = drum \wedge$ |
| | $fn \notin tree[path]$ |
| POST | **disk** $\models$ log_rep(NoTxn, $new\_state$) |
| | **new_state** $\models$ dir_rep($new\_tree$) $\wedge$ |
| | $new\_tree = tree$.update($path,\ fn,$ EmptyFile) |
| RECOVER | **disk** $\models$ log_rep(NoTxn, $start\_state$) $\vee$ |
| | log_rep(NoTxn, $new\_state$) |

- **create()** is atomic, if **log_recover()** runs after every crash
- POST is stronger than RECOVER

# CHL summary

- Key ideas: **crash conditions** and **recovery semantics**
- CHL benefit: enables precise failure specifications
  - Allows for automatic chaining of pre/post/crash conditions
  - Reduces proof burden
- CHL cost: must write crash condition for every function, loop, etc.
  - Crash conditions are often simple (above logging layer)

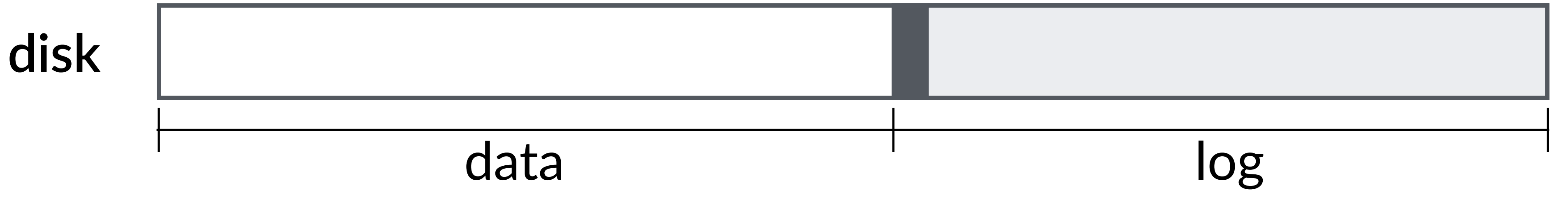
# Outline

- Crash safety
  - What is the correct behavior after a crash?
- ✔ Challenge 1: formalizing crashes
  - Crash Hoare Logic (CHL)
- Challenge 2: incorporating performance optimizations
  - Disk sequences
- Building a complete file system
- Evaluation

# FSCQ implements many optimizations

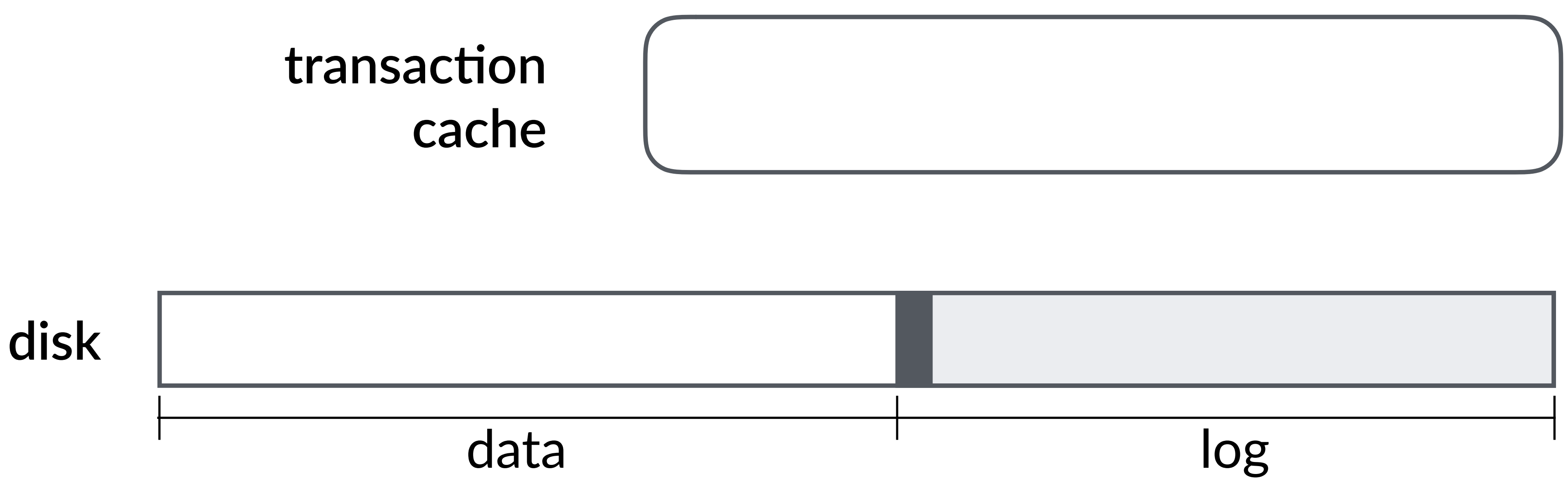
- **Group commit**
  - Buffer transactions in memory, and flush them in a single batch
  - Relax durability guarantee
- Log-bypass writes
  - File data writes go to the disk (buffer cache) directly
- Log checksums
  - Checksum log entries to reduce write barriers
- Deferred apply
  - Apply the log only when the log is full

# Example: group commit



# Example: group commit

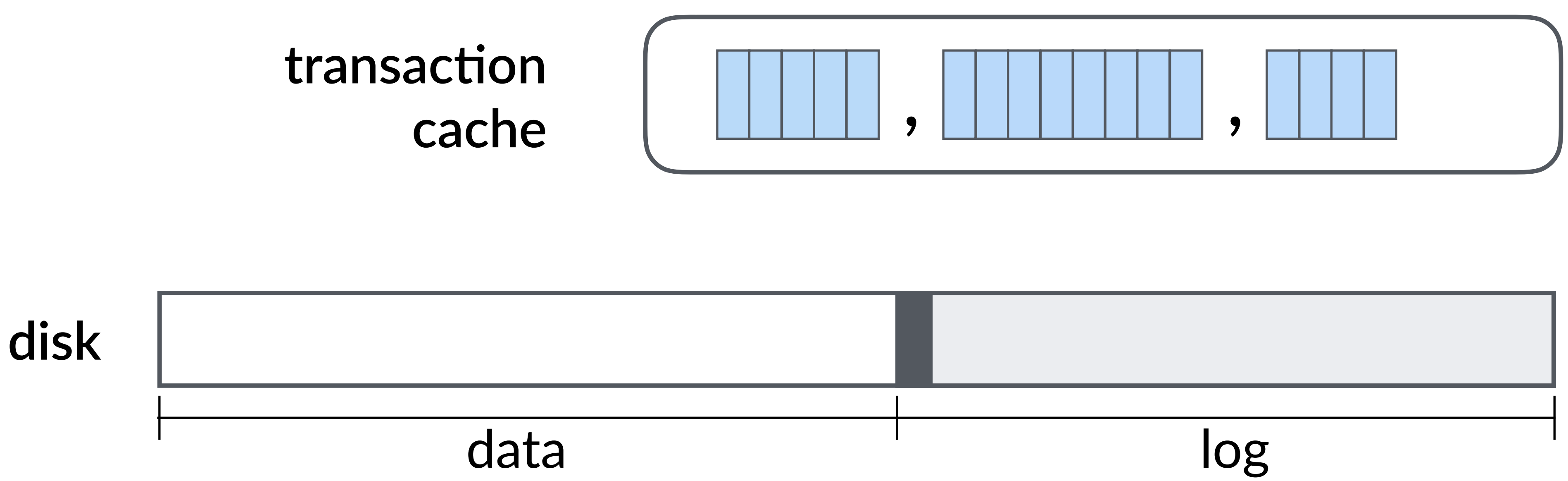
1. Each file-system call forms a transaction, and are buffered in the **transaction cache**



# Example: group commit

```
➡ mkdir('d')
➡ create('d/a')
➡ rename('d/a', 'd/b')
```

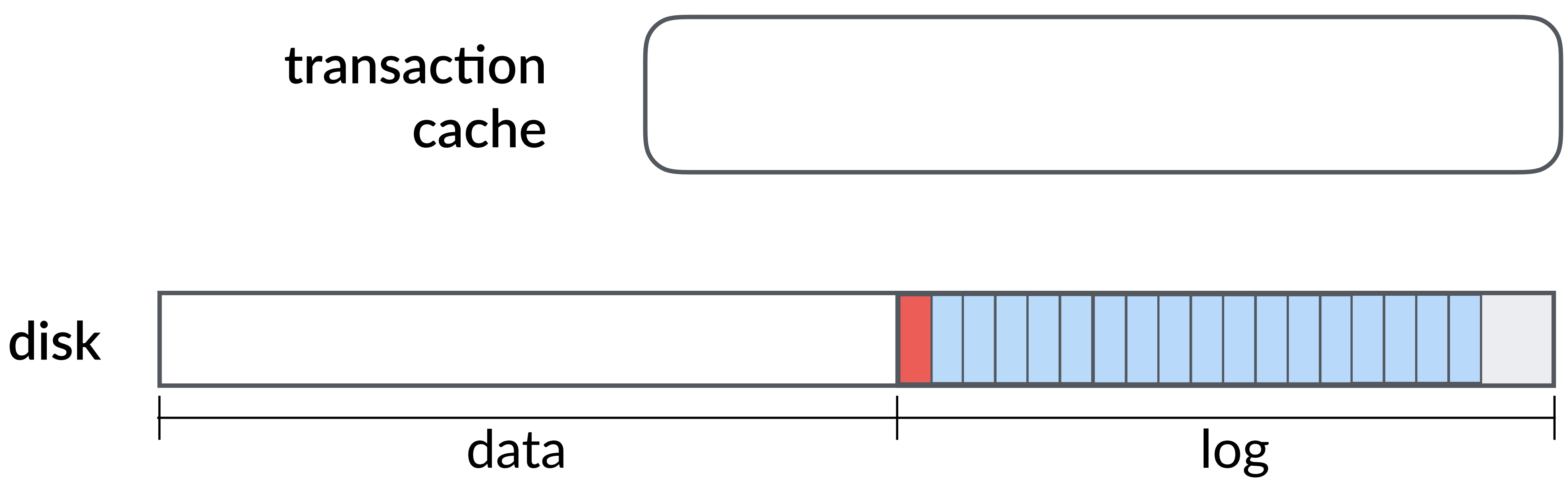
1. Each file-system call forms a transaction, and are buffered in the **transaction cache**



# Example: group commit

```
➡ mkdir('d')
➡ create('d/a')
➡ rename('d/a', 'd/b')
➡ fsync('d')
```

1. Each file-system call forms a transaction, and are buffered in the **transaction cache**
2. fsync() flushes cached transactions to the on-disk log in a batch
   - Preserve order



# Challenge: formalizing group commit

- Many more crash states (e.g., before or after mkdir() )
- On-disk state can be irrelevant to create() itself, but to some previous operations

```
➡ mkdir('d')
➡ create('d/a')
```

SPEC create(*dnum*, *fn*)

PRE **disk** $\models$ log_rep(NoTxn, *start_state*)

**start_state** $\models$ dir_rep(*tree*) $\wedge$

$\exists$ *path*, *tree*[*path*].*node* = *dnum* $\wedge$

*fn* $\notin$ *tree*[*path*]

POST **disk** $\models$ log_rep(NoTxn, *new_state*)

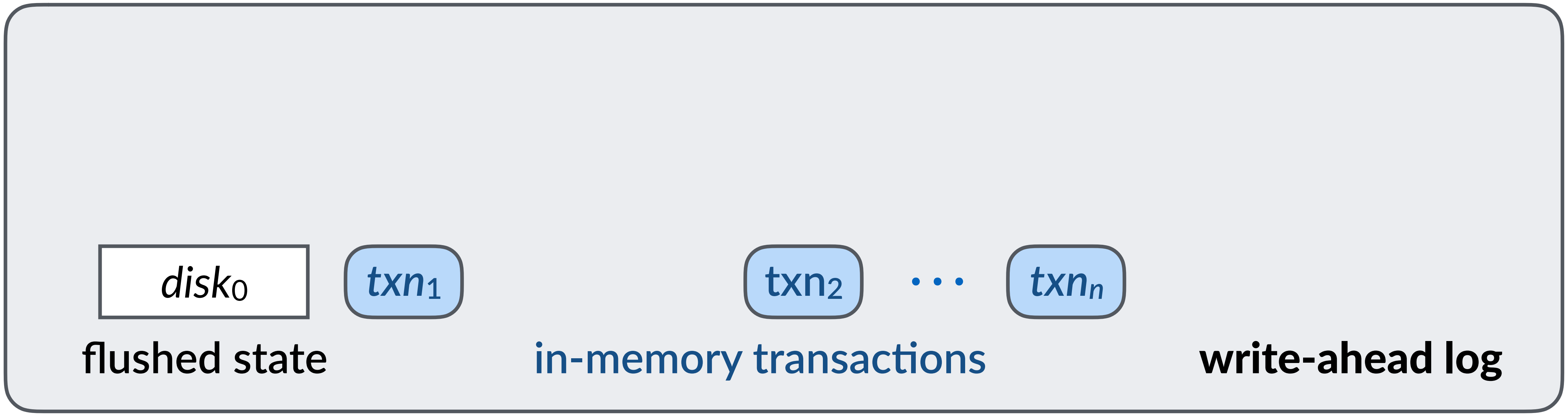
**new_state** $\models$ dir_rep(*new_tree*) $\wedge$

*new_tree* = *tree*.update(*path*, *fn*, Empty

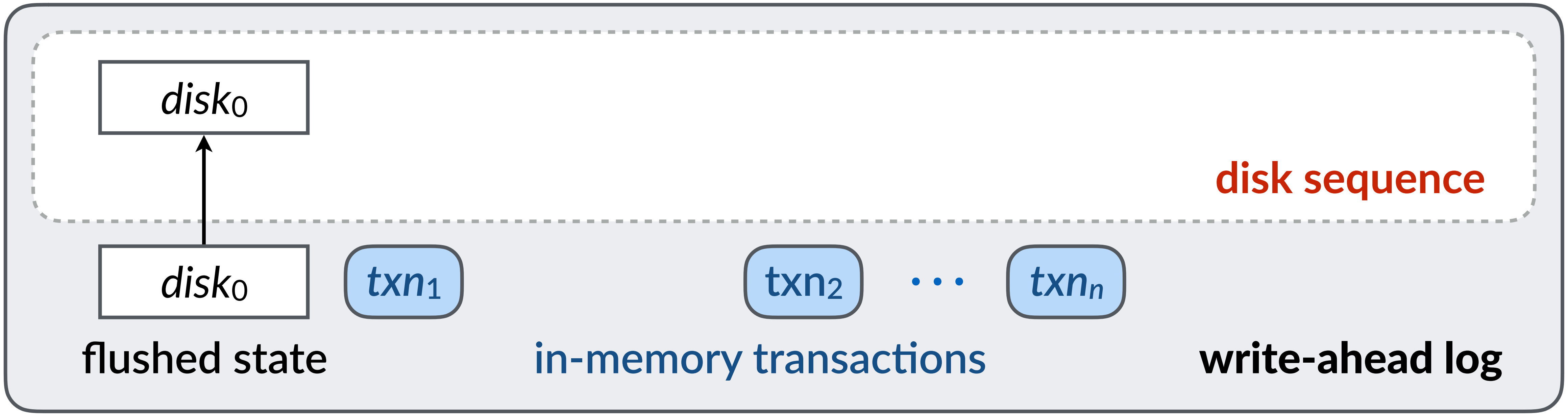
CRASH **disk** $\models$ would_recover_either (*start_state*, *new_state*

# Specification idea: **disk sequences**

# Specification idea: disk sequences

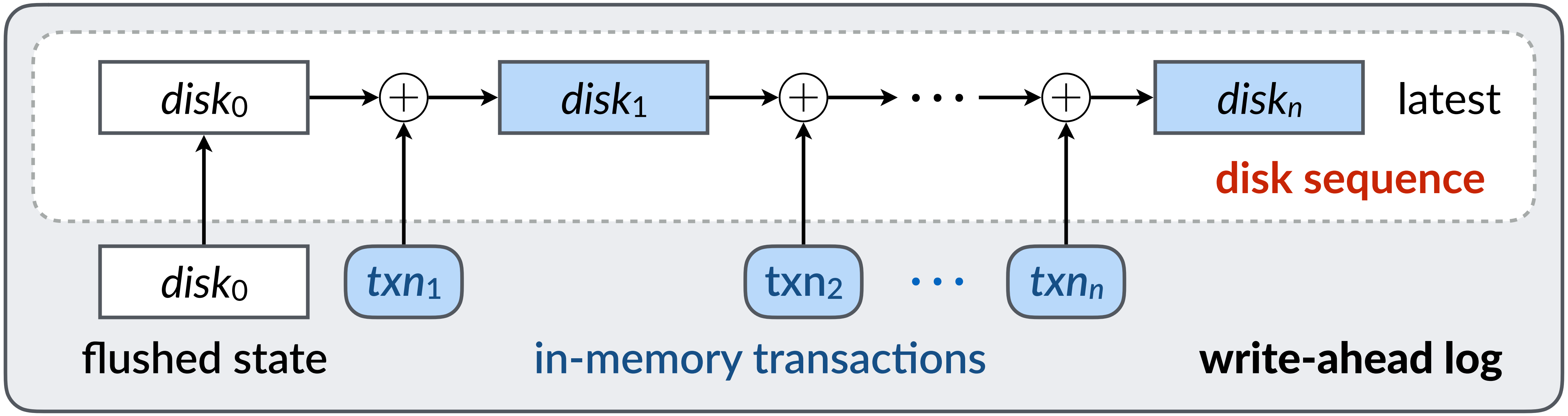


# Specification idea: **disk sequences**



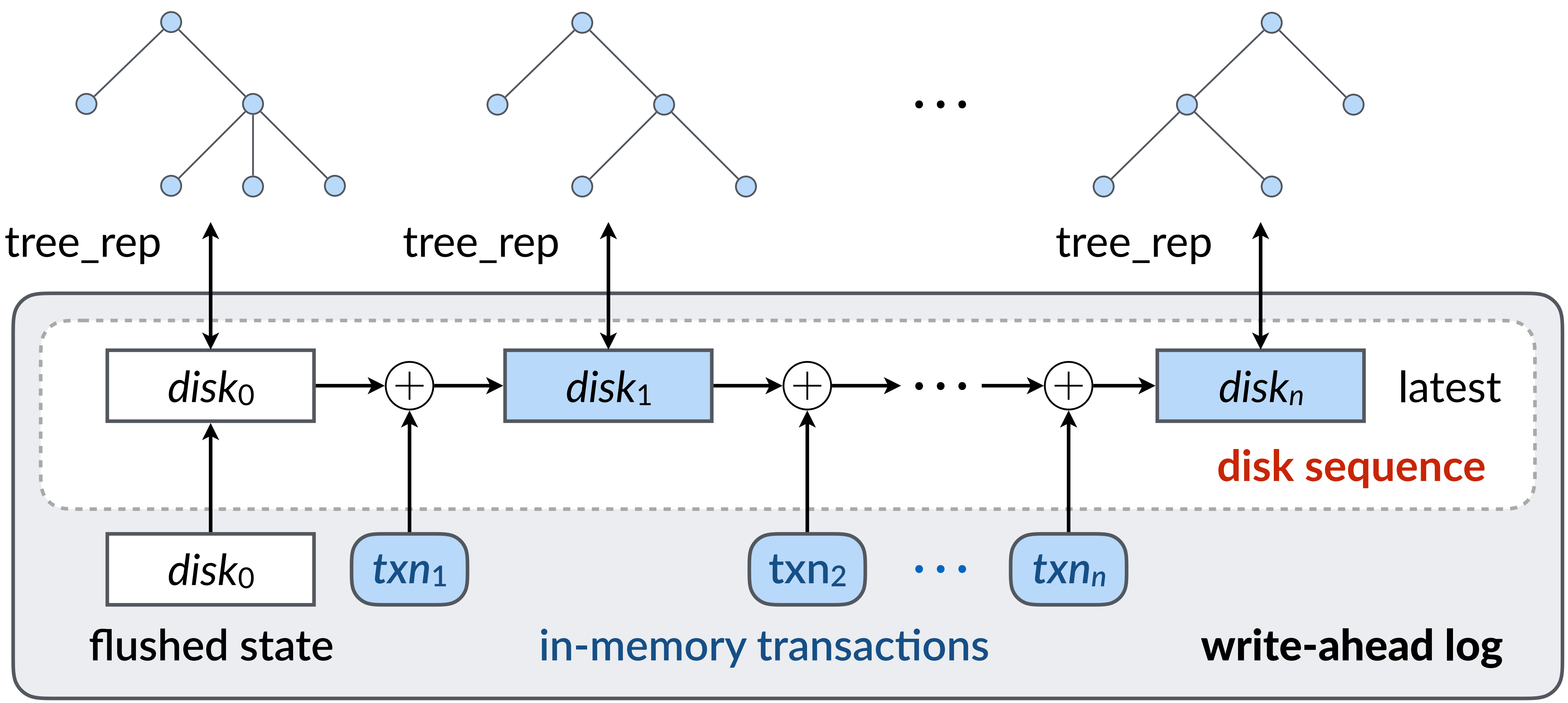
# Specification idea: **disk sequences**

- Each (cached) system call adds a new logical disk to the sequence



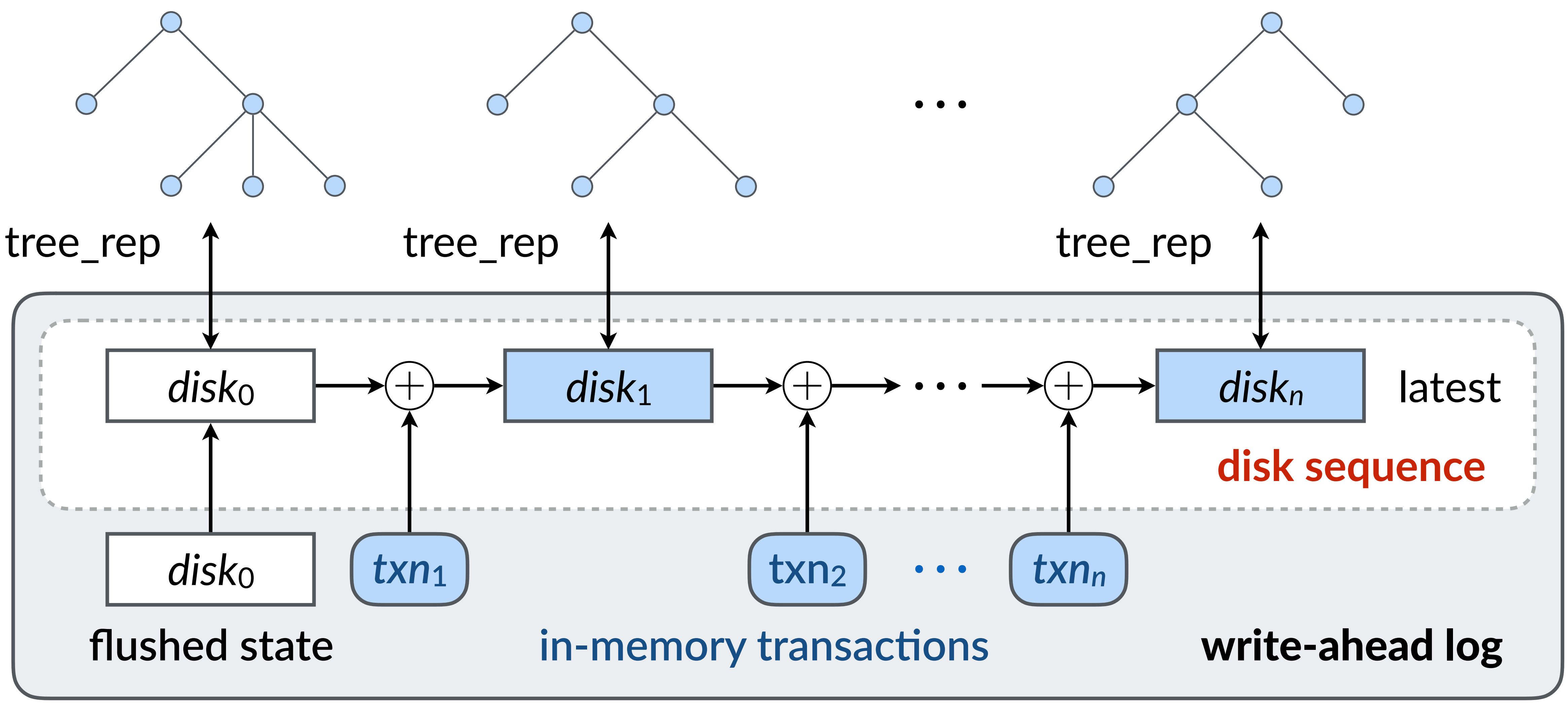
# Specification idea: disk sequences

- Each (cached) system call adds a new logical disk to the sequence
- Each logical disk has a corresponding tree



# Specification idea: disk sequences

- Each (cached) system call adds a new logical disk to the sequence
- Each logical disk has a corresponding tree
- Capture the idea that **metadata updates must be ordered**



# New specification with disk sequence

SPEC $\text{create}\,(\mathit{dnum},\ \mathit{fn})$

PRE $\textbf{disk} \models \text{log\_rep}(\text{NoTxn}, \mathit{disk\_seq})$

$\boldsymbol{\mathit{disk\_seq}}\textbf{.latest} \models \text{dir\_rep}(\mathit{tree}) \ \land$

$\exists\, \mathit{path},\ \mathit{tree}[\mathit{path}].\mathit{node} = \mathit{dnum} \land$

$\mathit{fn} \notin \mathit{tree}[\mathit{path}]$

POST $\textbf{disk} \models \text{log\_rep}(\text{NoTxn},\ \mathit{disk\_seq} +\!\!+ \{\mathit{new\_state}\})$

$\textbf{new\_state} \models \text{dir\_rep}(\mathit{new\_tree}) \ \land$

$\mathit{new\_tree} = \mathit{tree}.\text{update}(\mathit{path},\ \mathit{fn},\ \text{EmptyFile})$

CRASH $\textbf{disk} \models \textbf{would\_recover\_any}\ \textbf{(}\mathit{disk\_seq} +\!\!+ \{\mathit{new\_state}\}\textbf{)}$

- Specification isn't more complicated

# Specification for fsync on directories

SPEC fsync ($dir\_inum$)

PRE $\textbf{disk} \models \text{log\_rep}(\text{NoTxn}, \mathit{disk\_seq})$

$\boldsymbol{\mathit{disk\_seq}}\textbf{.latest} \models \text{tree\_rep}(\mathit{tree}) \;\wedge$

$\text{IsDir}(\text{find\_inum}(\mathit{tree}, \mathit{dir\_inum}))$

POST $\textbf{disk} \models \text{log\_rep}(\text{NoTxn}, \{\boldsymbol{\mathit{disk\_seq}}\textbf{.latest}\})$

CRASH $\textbf{disk} \models \textbf{would\_recover\_any}(\mathit{disk\_seq})$

- After fsync(), there is only one possible on-disk state (the latest one)

# Formalization techniques for optimizations

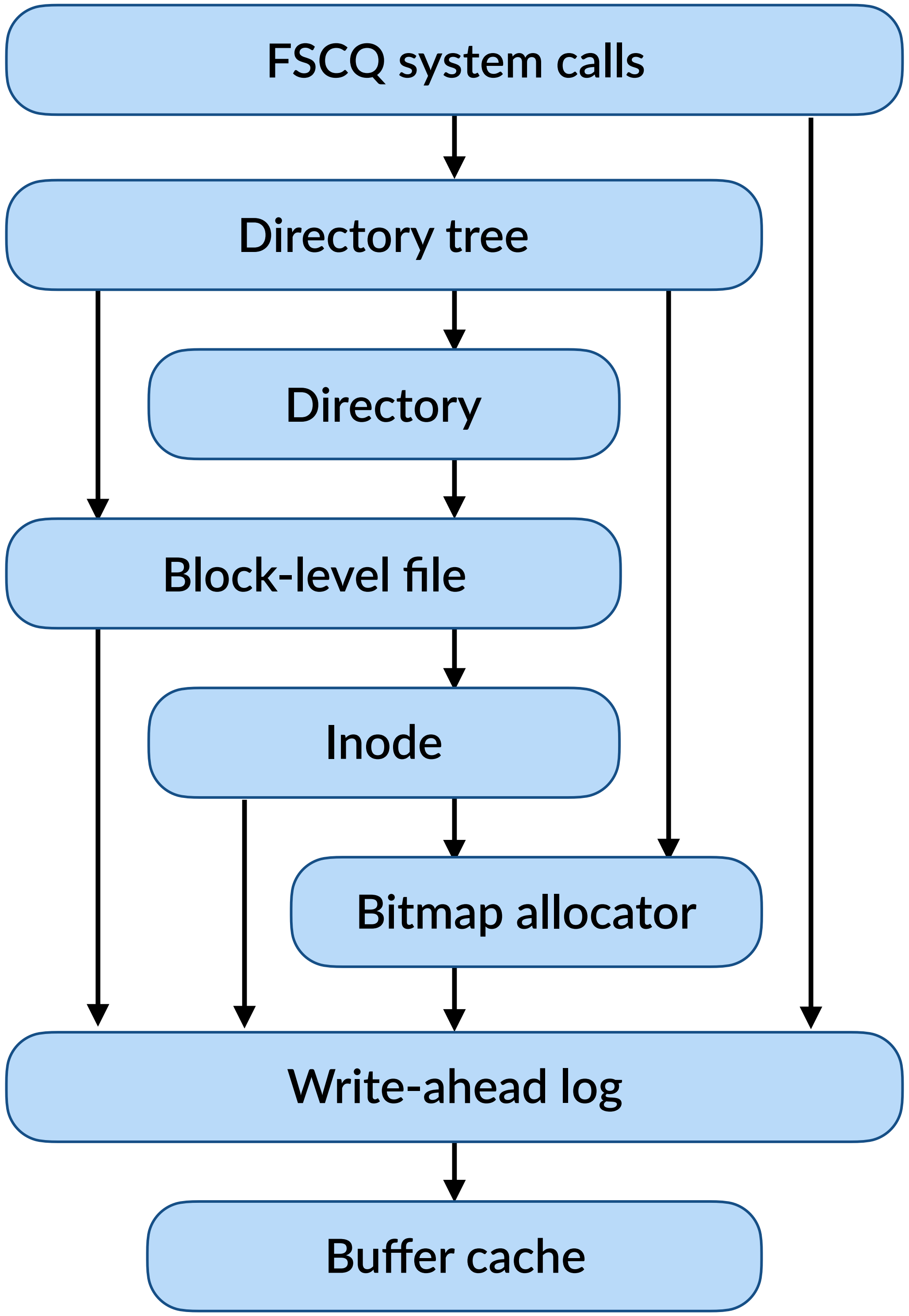
- ✓ Group commit
  - **Disk sequences**: captures ordered metadata updates
- Log-bypass writes
  - **Disk relations**: enforces safety w.r.t. metadata updates
- Log checksums
  - **Checksum model**: soundly reasons about hash collision

# Outline

- Crash safety
  - What is the correct behavior after a crash?
- Challenge 1: formalizing crashes
  - Crash Hoare Logic (CHL)
- Challenge 2: incorporating performance optimizations
  - Disk sequences
- Building a complete file system
- Evaluation

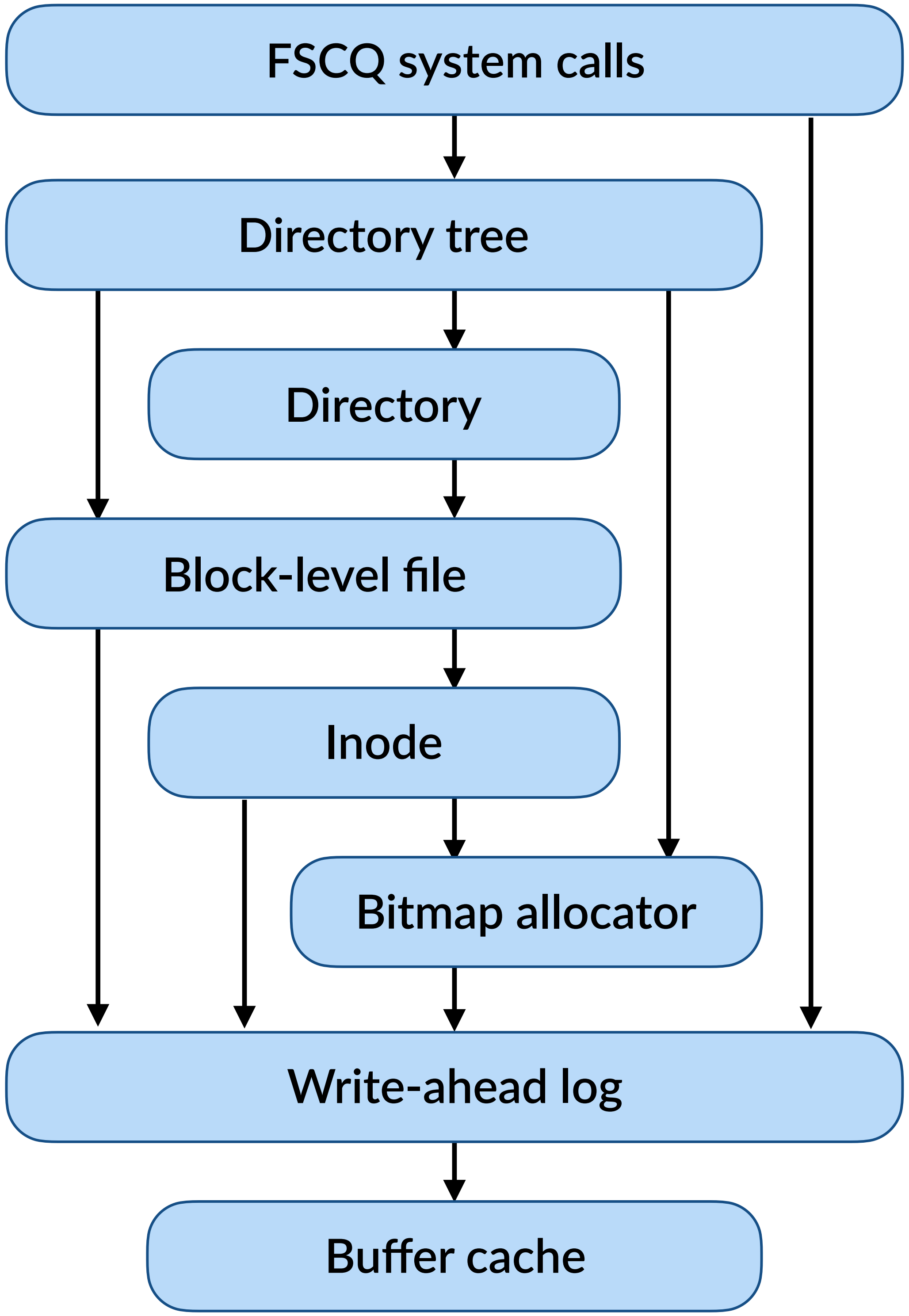
# FSCQ: building a complete file system

- File system design is close to v6 Unix (+ logging)



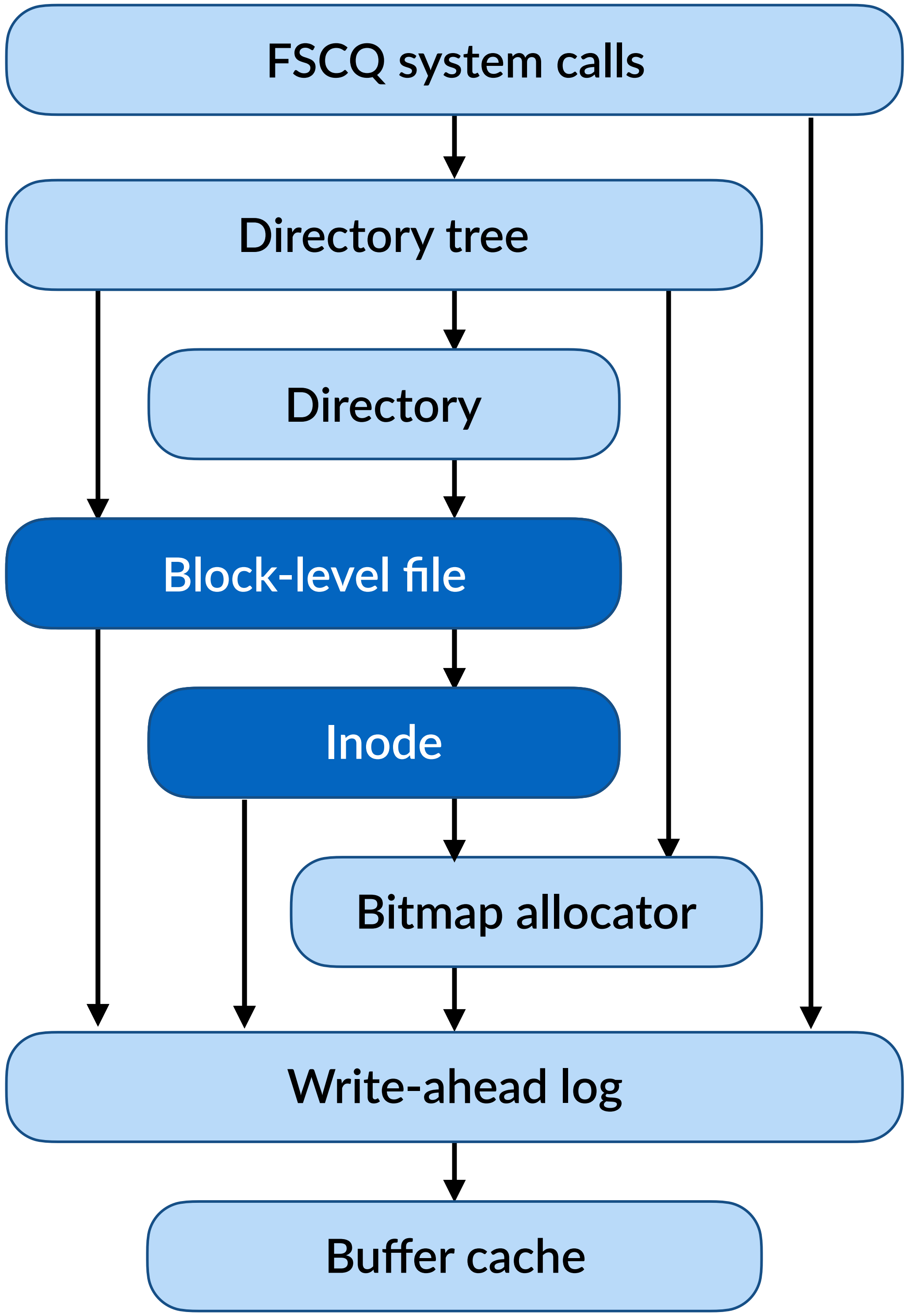
# FSCQ: building a complete file system

- File system design is close to v6 Unix (+ logging)
- Implementation aims to reduce proof effort



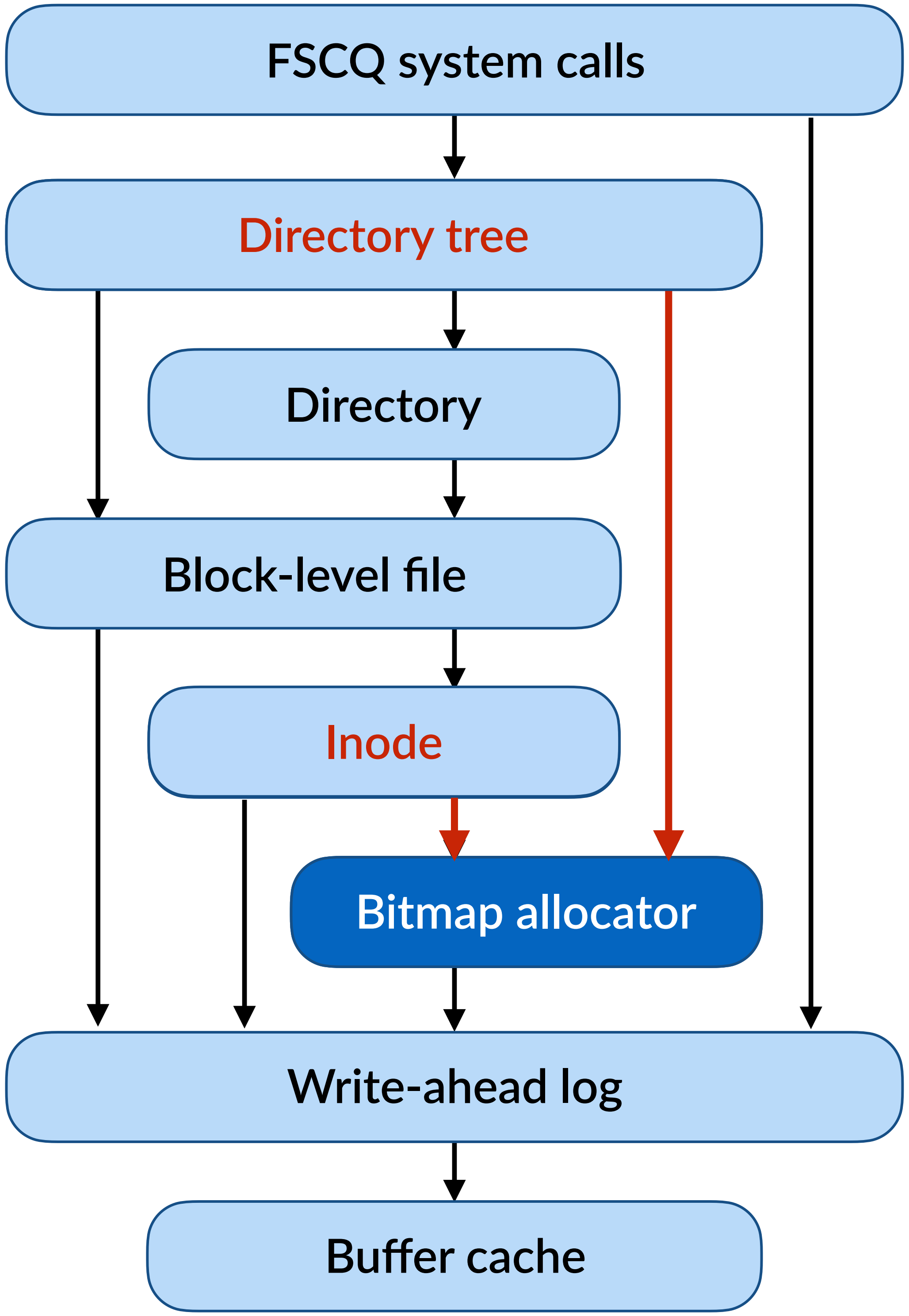
# FSCQ: building a complete file system

- File system design is close to v6 Unix (+ logging)
- Implementation aims to reduce proof effort
  - Many precise internal abstraction layers
    - e.g., split File and Inode



# FSCQ: building a complete file system

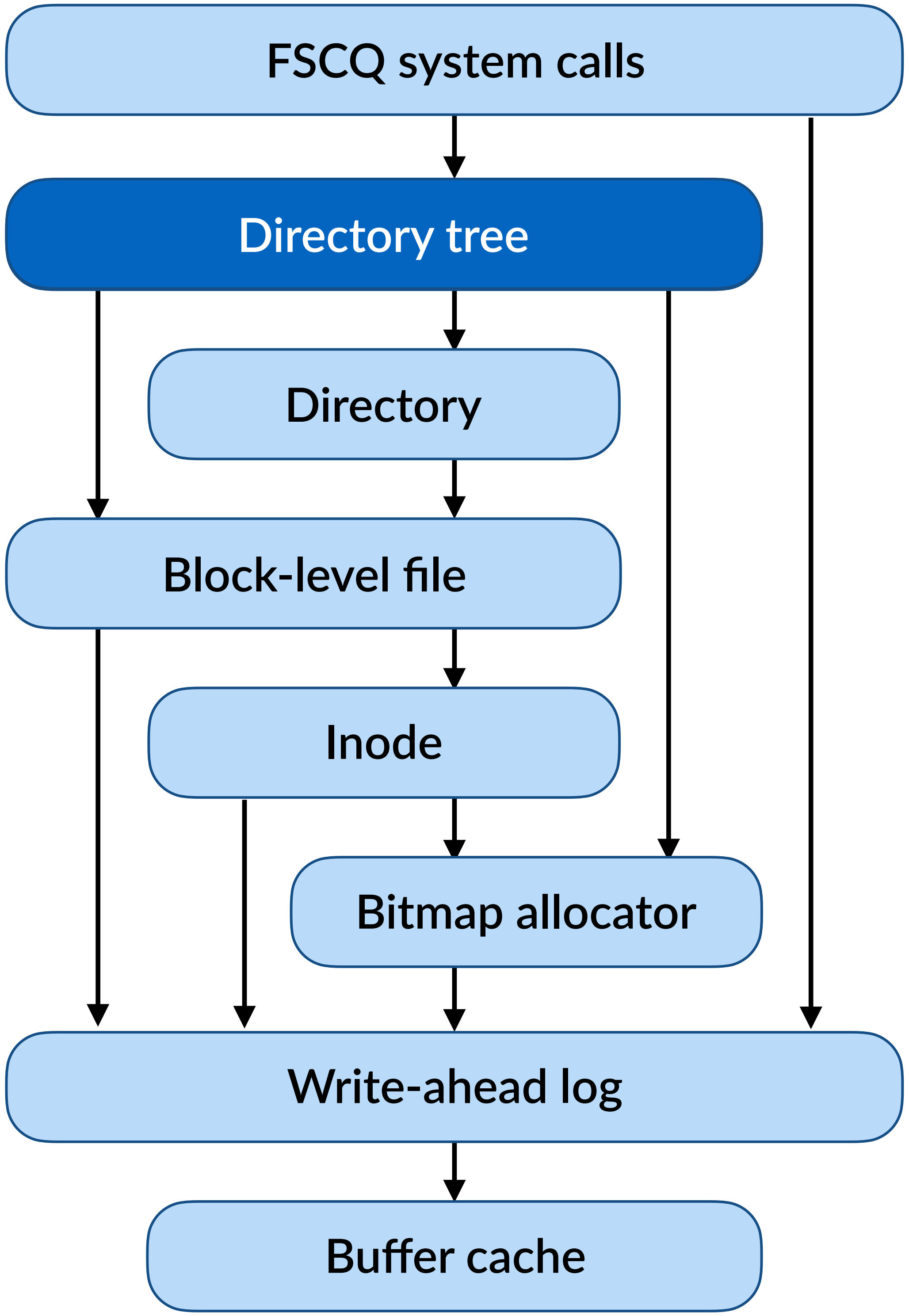
- File system design is close to v6 Unix (+ logging)
- Implementation aims to reduce proof effort
  - Many precise internal abstraction layers
    - e.g., split File and Inode
  - Reuse proven components
    - e.g., general bitmap allocator



# FSCQ: building a complete file system

- File system design is close to v6 Unix (+ logging)

- Implementation aims to reduce proof effort
  - Many precise internal abstraction layers
    - e.g., split File and Inode
  - Reuse proven components
    - e.g., general bitmap allocator
  - Simpler specifications
    - e.g., no hard link ⇒ tree spec



# Evaluation

- What bugs do FSCQ's theorems eliminate?
- How much development effort is required for FSCQ?
- How well does FSCQ perform?

# Does FSCQ eliminate bugs?

- One data point: once theorems proven, no implementation bugs in proven code
  - Did find some mistakes in spec, as a result of end-to-end checks
  - E.g., forgot to specify that extending a file should zero-fill

# Does FSCQ eliminate bugs?

- One data point: once theorems proven, no implementation bugs in proven code
  - Did find some mistakes in spec, as a result of end-to-end checks
  - E.g., forgot to specify that extending a file should zero-fill
- Systematic study
  - Categorize bugs from Linux kernel's patch history
  - Manually examine if FSCQ can eliminate bugs in each category

# FSCQ's theorems eliminate many bugs

| Bug category | Prevented? |
|---|---|
| Mistakes in logging logic<br>*e.g., combining incompatible optimizations* | ✔ |
| Misuse of logging API<br>*e.g., releasing indirect block in two transactions* | ✔ |
| Mistakes in recovery protocol<br>*e.g., issuing write barrier in the wrong order* | ✔ |
| Improper corner-case handling<br>*e.g., running out of blocks during rename* | ✔ |

# FSCQ's theorems eliminate many bugs

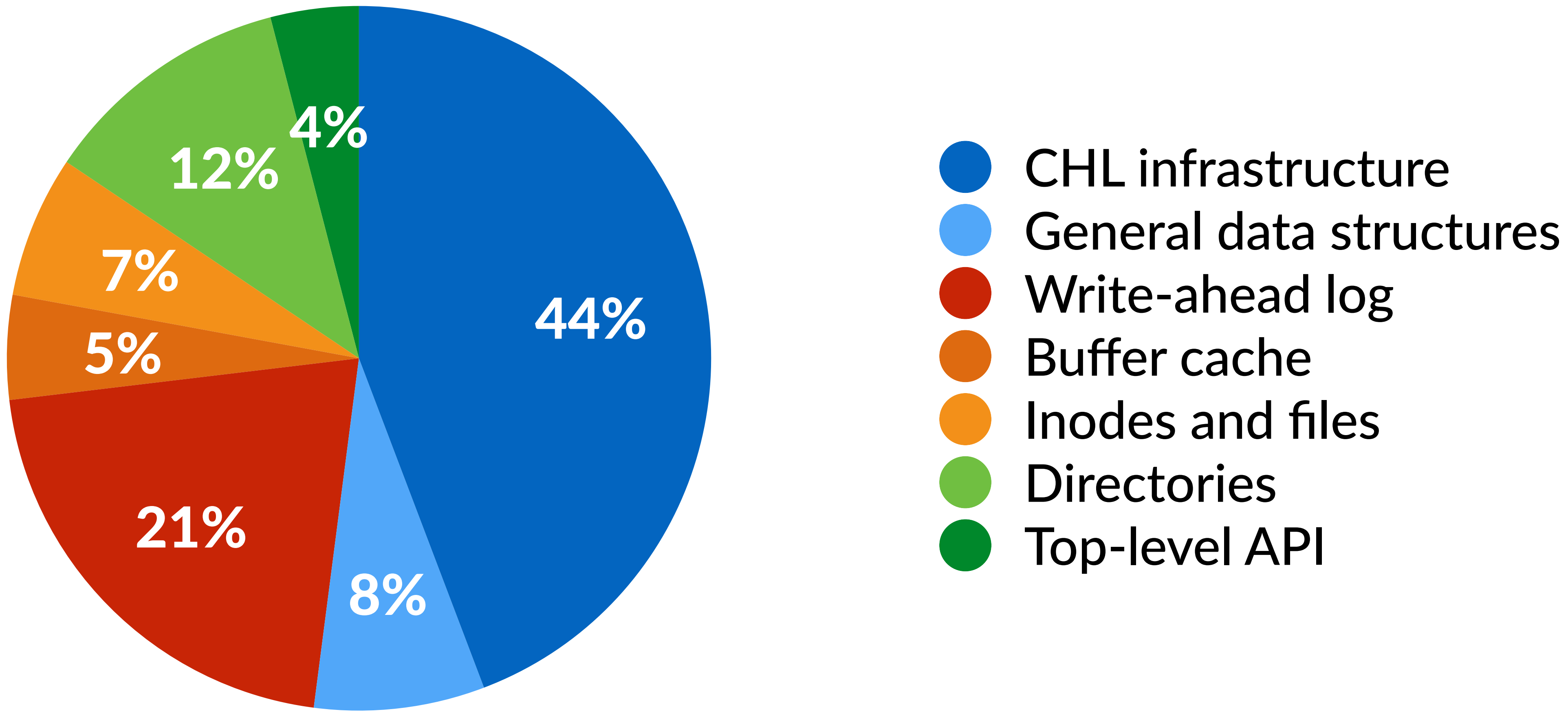
| Bug category | Prevented? |
|---|---|
| Mistakes in logging logic<br>*e.g., combining incompatible optimizations* | ✔ |
| Misuse of logging API<br>*e.g., releasing indirect block in two transactions* | ✔ |
| Mistakes in recovery protocol<br>*e.g., issuing write barrier in the wrong order* | ✔ |
| Improper corner-case handling<br>*e.g., running out of blocks during rename* | ✔ |
| Low-level bugs<br>*e.g., double free, integer overflow* | **Some (memory safe)** |
| Returning incorrect error code | **Some** |

# FSCQ's theorems eliminate many bugs

| Bug category | Prevented? |
|---|---|
| Mistakes in logging logic<br>*e.g., combining incompatible optimizations* | ✔ |
| Misuse of logging API<br>*e.g., releasing indirect block in two transactions* | ✔ |
| Mistakes in recovery protocol<br>*e.g., issuing write barrier in the wrong order* | ✔ |
| Improper corner-case handling<br>*e.g., running out of blocks during rename* | ✔ |
| Low-level bugs<br>*e.g., double free, integer overflow* | **Some (memory safe)** |
| Returning incorrect error code | **Some** |
| Concurrency | Not supported |
| Security | Not supported |

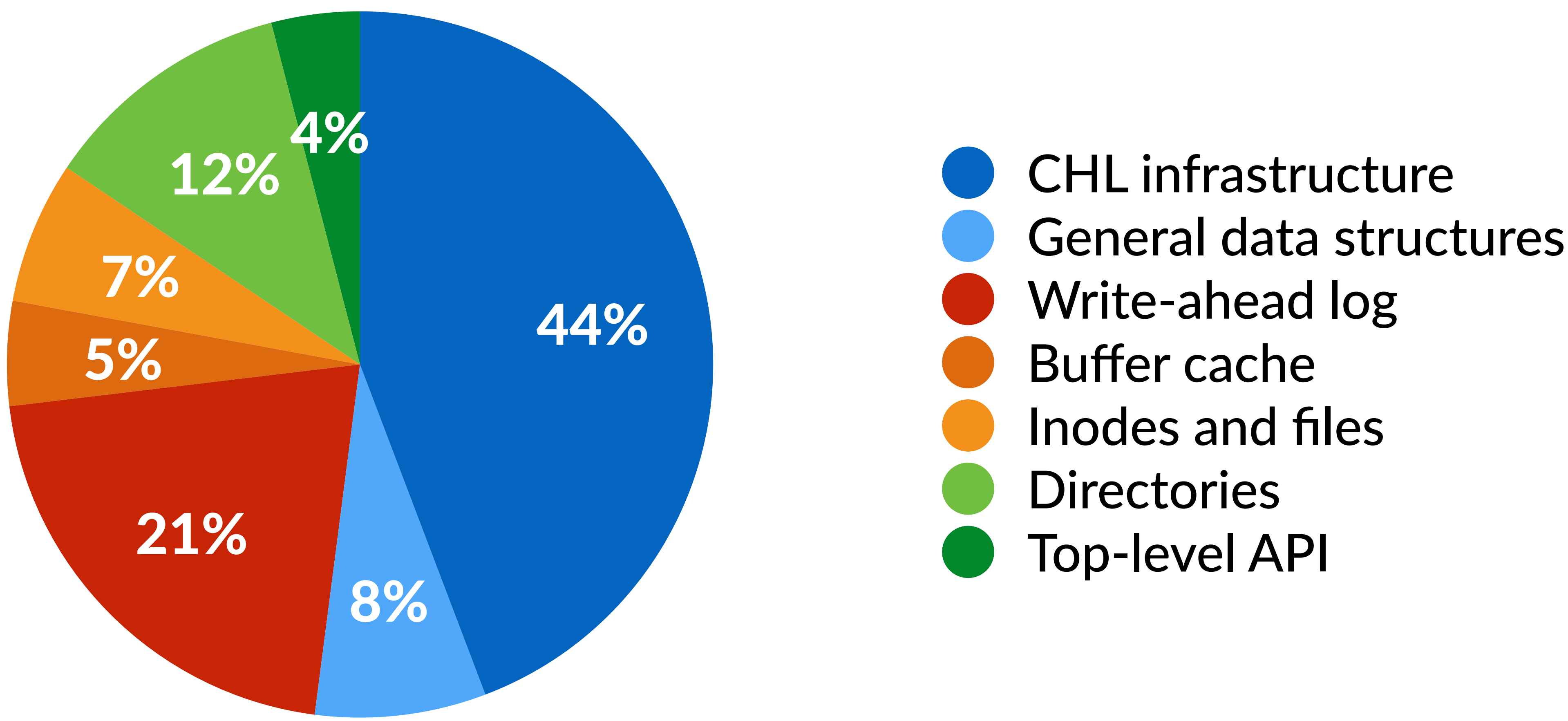
# Development effort

- Total of ~50,000 lines of **verified** code, specs, and proofs in Coq
  - > 50% **reusable infrastructure**



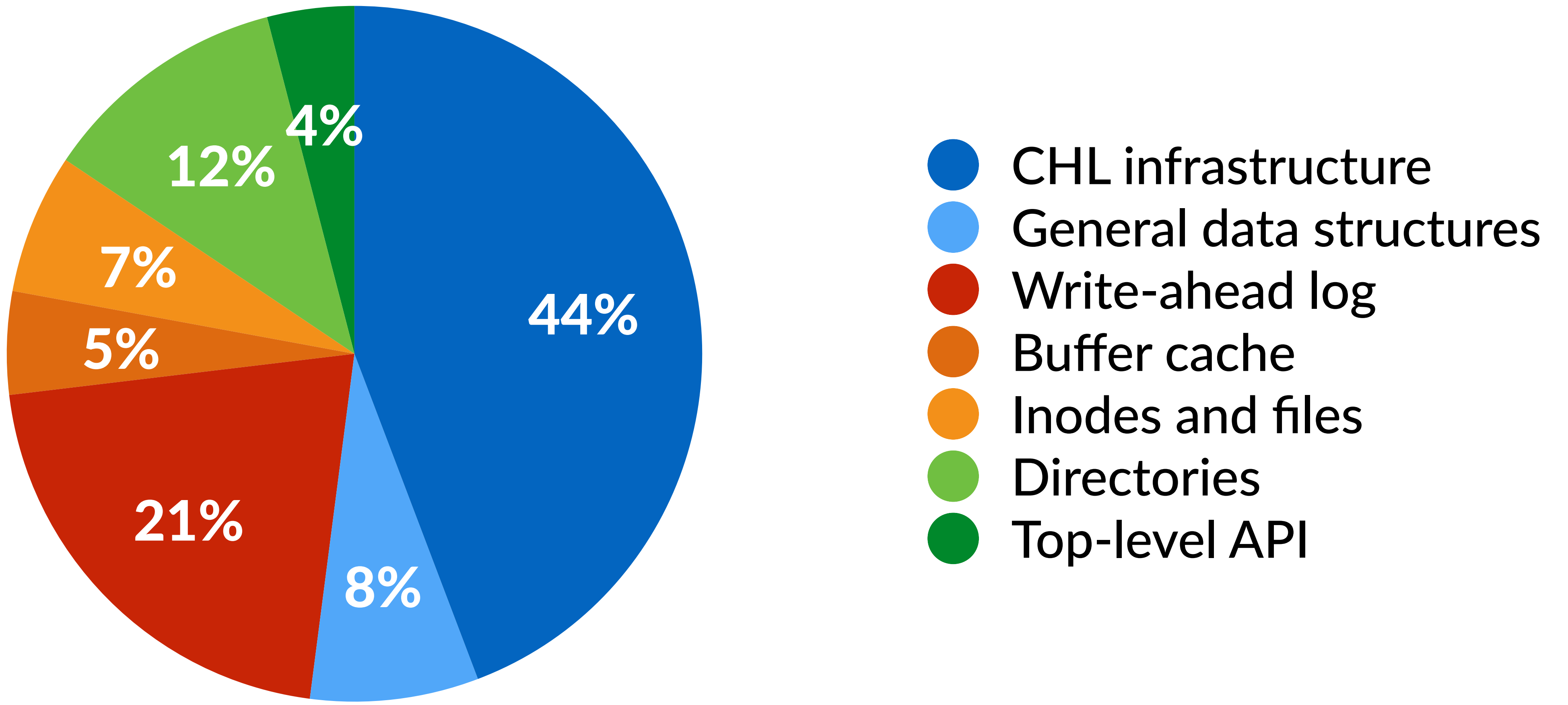
# Development effort

- Total of ~50,000 lines of **verified** code, specs, and proofs in Coq
  - > 50% **reusable infrastructure**
- Comparison: ext4 has ~60,000 lines of C code (many more features)

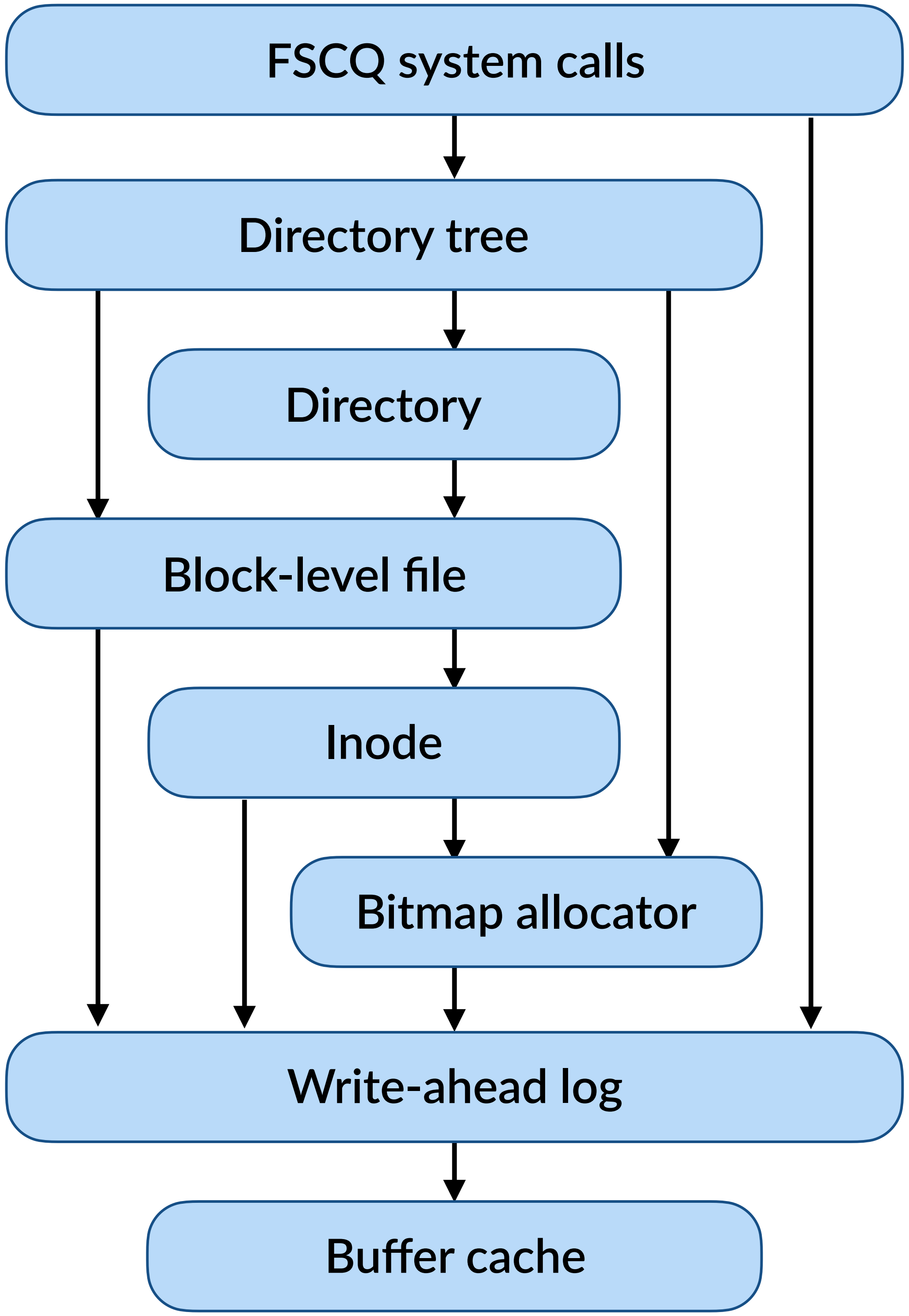


# Development effort

- Total of ~50,000 lines of **verified** code, specs, and proofs in Coq
  - > 50% **reusable infrastructure**
- Comparison: ext4 has ~60,000 lines of C code (many more features)
- What's the cost of adding new features to FSCQ?

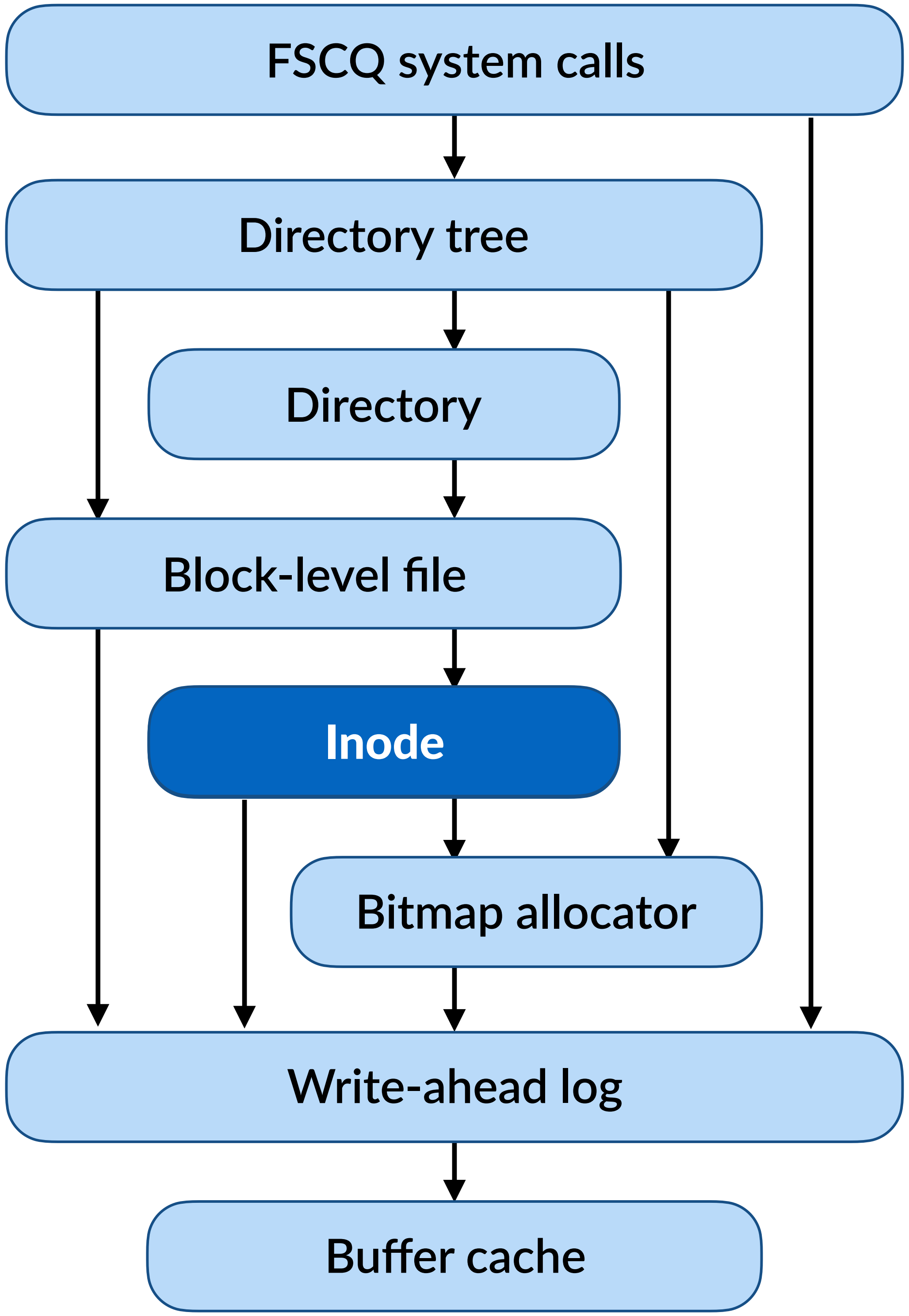


# Change effort proportional to scope of change



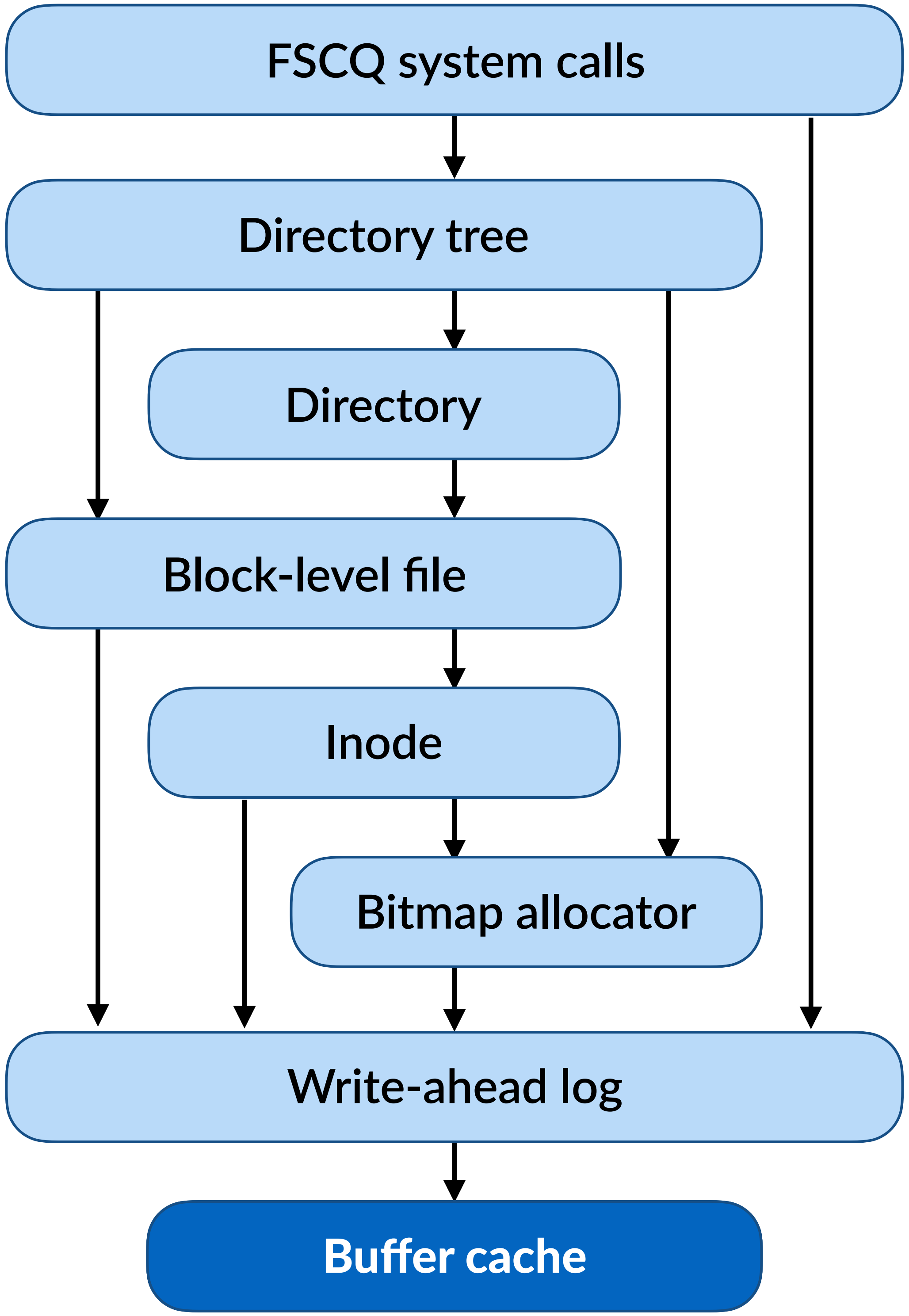
# Change effort proportional to scope of change

- Indirect blocks:
  - + 1,500 lines in Inode



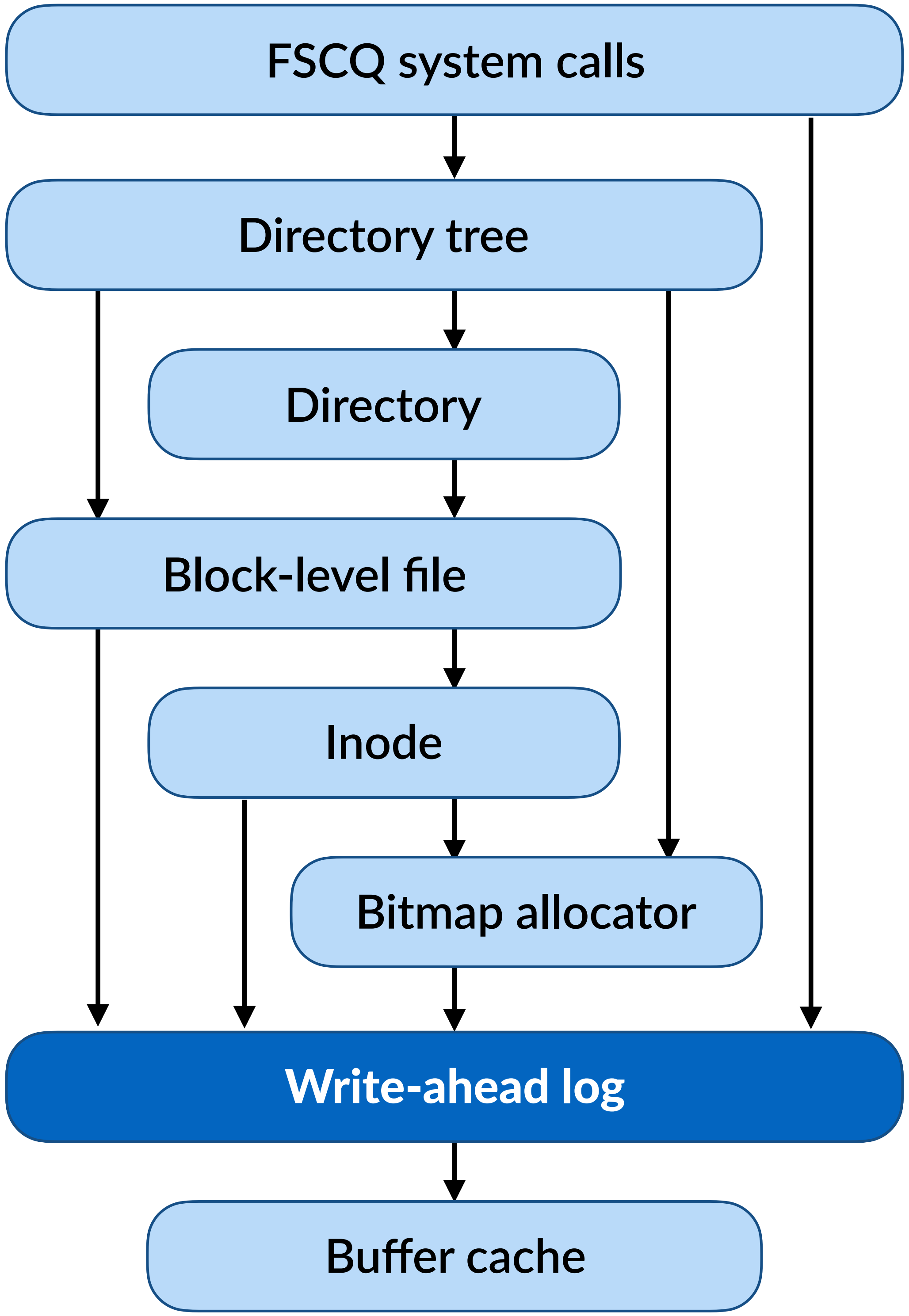
# Change effort proportional to scope of change

- Indirect blocks:
  - + 1,500 lines in Inode
- Write-back buffer cache:
  - + 2300 lines beneath log
    ~ 600 lines in rest of FSCQ



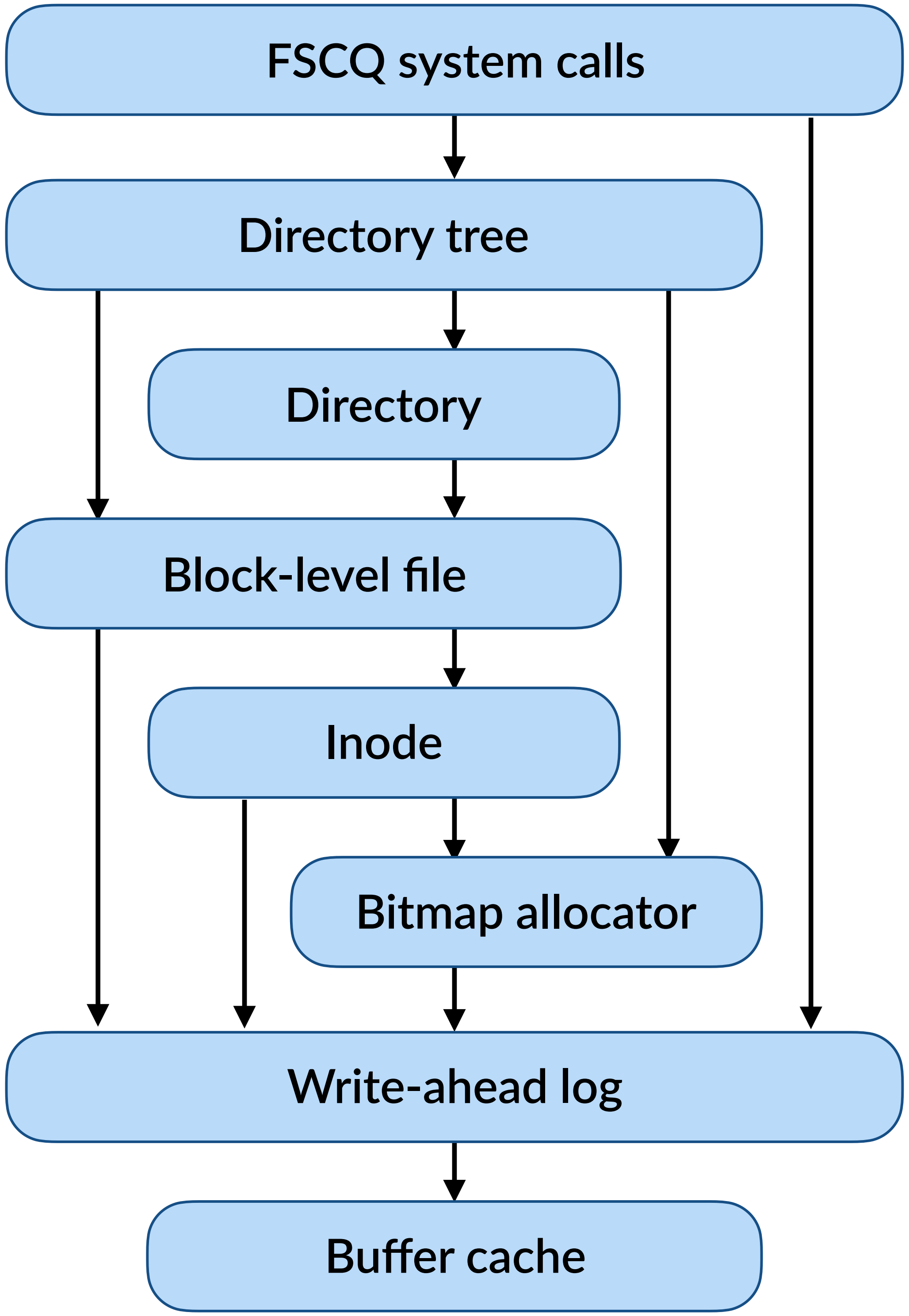
# Change effort proportional to scope of change

- Indirect blocks:
  - + 1,500 lines in Inode
- Write-back buffer cache:
  - + 2300 lines beneath log
    ~ 600 lines in rest of FSCQ
- Group commit:
  - + 1800 lines in Log
    ~ 100 lines in rest of FSCQ



# Change effort proportional to scope of change

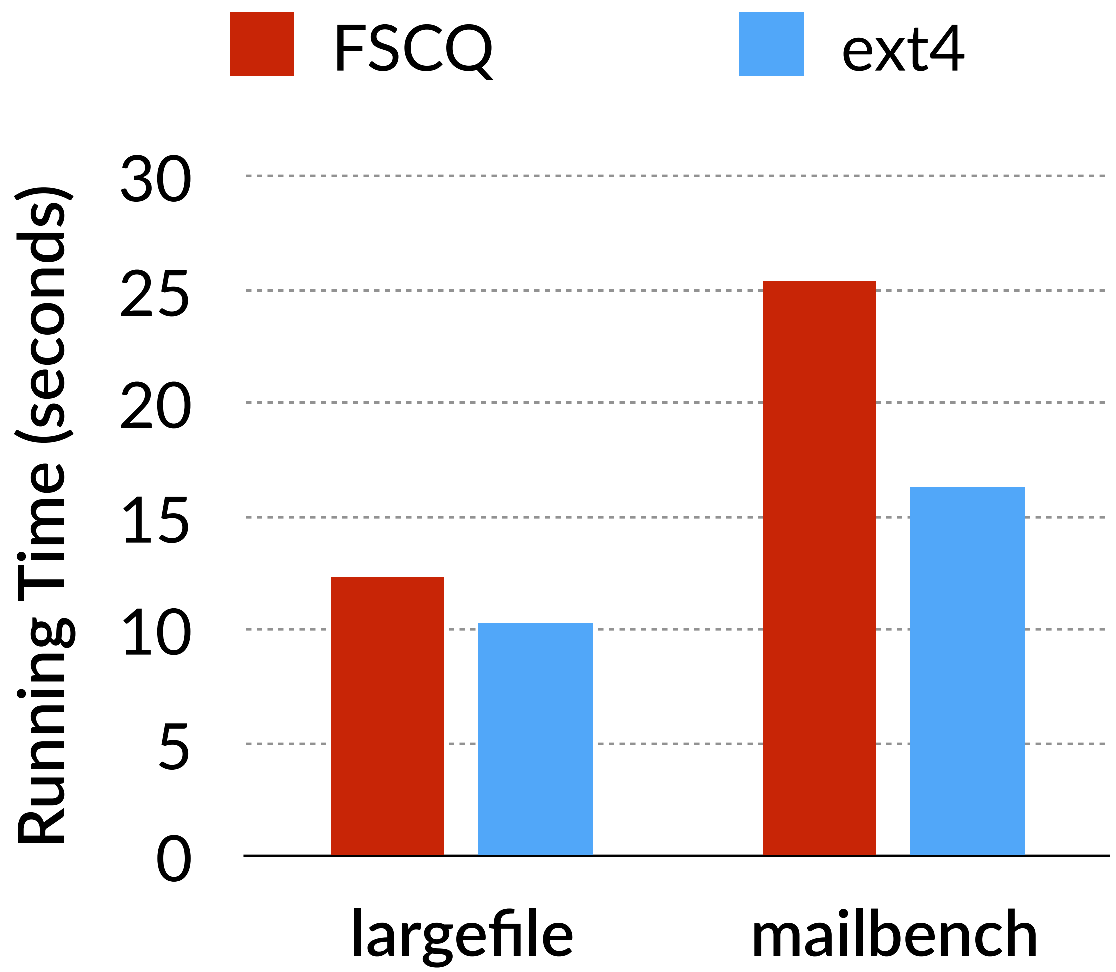
- Indirect blocks:
  - + 1,500 lines in Inode
- Write-back buffer cache:
  - + 2300 lines beneath log
    ~ 600 lines in rest of FSCQ
- Group commit:
  - + 1800 lines in Log
    ~ 100 lines in rest of FSCQ
- Changed lines include code, specs and proofs



# Performance comparison

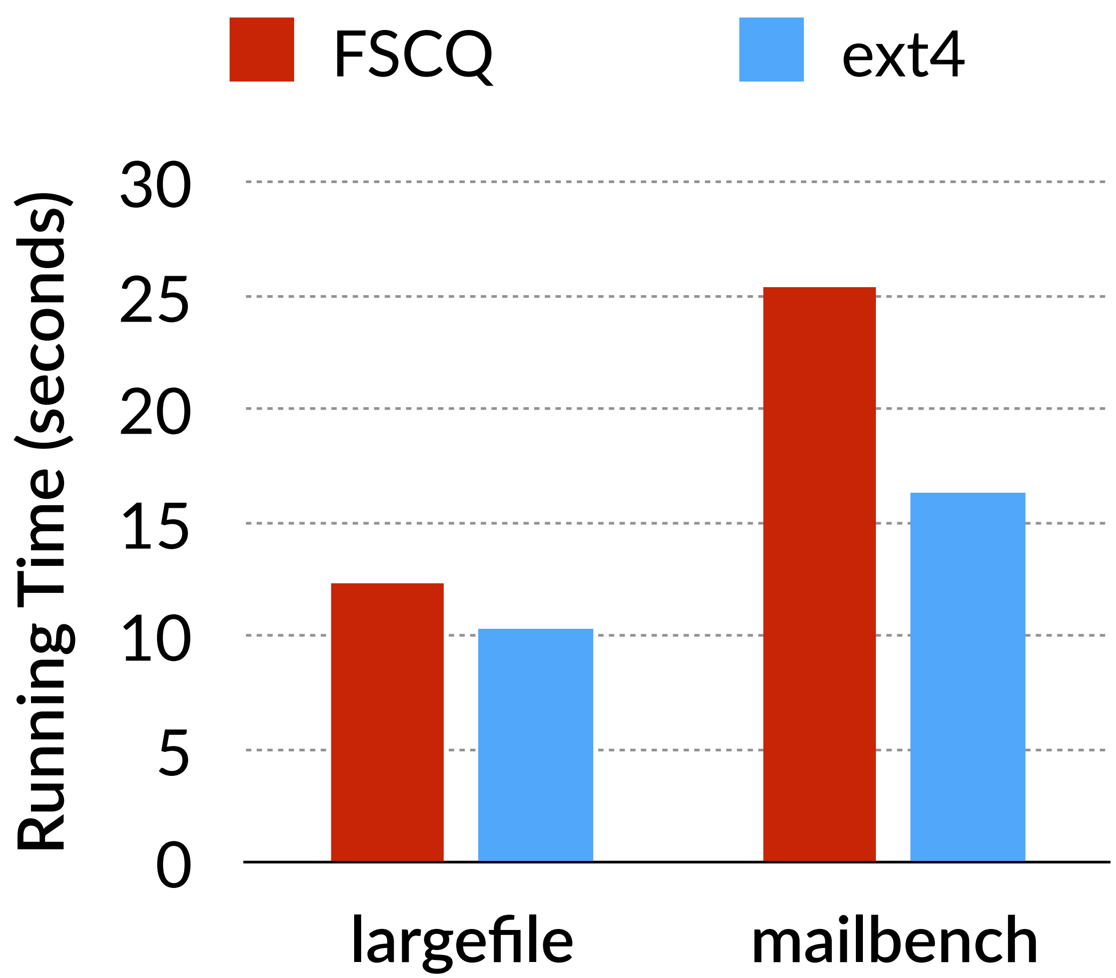
- File-system-intensive workload
  - LFS “largefile” benchmark
  - mailbench, a qmail-like mail server
- Compare with ext4 (non-certified) in default mode
  - Mount option: async,data=ordered
  - Use FUSE to forward and serialize requests (disable concurrency)
- Running on an hard disk on a desktop
  - Quad-core Intel i7-980X 3.33 GHz / 24 GB / Hitachi HDS721010CLA332
  - Linux 3.11 / GHC 8.0.1 / all file systems run on a separate partition

# FSCQ Performance




- FSCQ's CPU overhead is high
- **FSCQ's I/O performance is on par with ext4**

# FSCQ Performance




Number of disk I/Os per operation

| | largefile | | mailbench | |
|---|---|---|---|---|
| | write | sync | write | sync |
| FSCQ | **1.0** | **1.0** | **50.0** | **9.8** |
| ext4 | 1.0 | 1.0 | 38.0 | 12.3 |

- FSCQ's CPU overhead is high
- **FSCQ's I/O performance is on par with ext4**

# Future directions

- Extracting to native code
  - Reduce both CPU overhead and TCB
- Certifying crash-safe applications
  - Use FSCQ's top-level spec to certify a mail server or a KV store
- Supporting concurrency
  - Run FSCQ in a multi-user environment
  - Exploit both I/O concurrency and parallelism

# Conclusion

- CHL helps specify and prove crash safety
  - Crash conditions
  - Recovery execution semantics
- FSCQ: first certified crash-safe file system
  - Precise specification in presence of crashes
  - I/O performance on par with Linux ext4
  - Moderate development effort

# Conclusion

- CHL helps specify and prove crash safety
  - Crash conditions
  - Recovery execution semantics
- FSCQ: first certified crash-safe file system
  - Precise specification in presence of crashes
  - I/O performance on par with Linux ext4
  - Moderate development effort


https://github.com/mit-pdos/fscq-impl