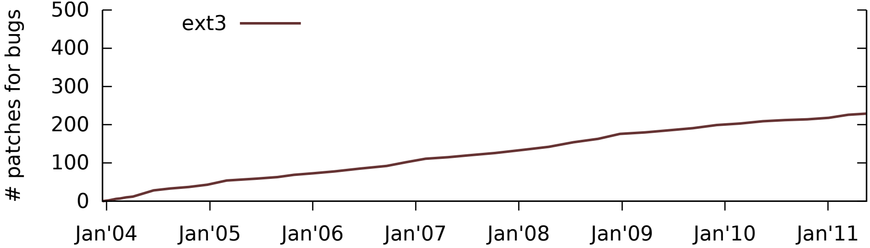
Using Crash Hoare Logic for Certifying the FSCQ File System

Haogang Chen, Daniel Ziegler, Tej Chajed,
Adam Chlipala, Frans Kaashoek, and Nickolai Zeldovich

MIT CSAIL

File systems are complex and have bugs

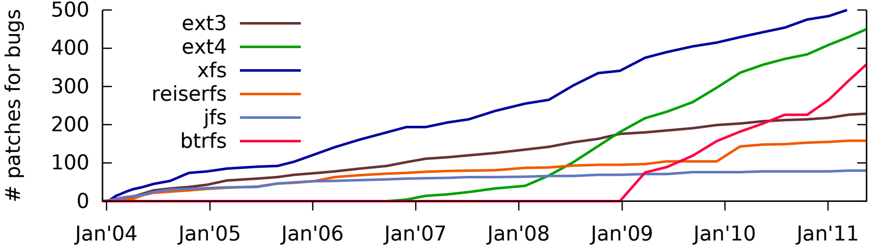
File systems are complex (e.g., Linux ext4 is ∼60,000 lines of code) and have many bugs:



Cumulative number of patches for file-system bugs in Linux; data from [Lu et al., FAST'13]

File systems are complex and have bugs

File systems are complex (e.g., Linux ext4 is ∼60,000 lines of code) and have many bugs:

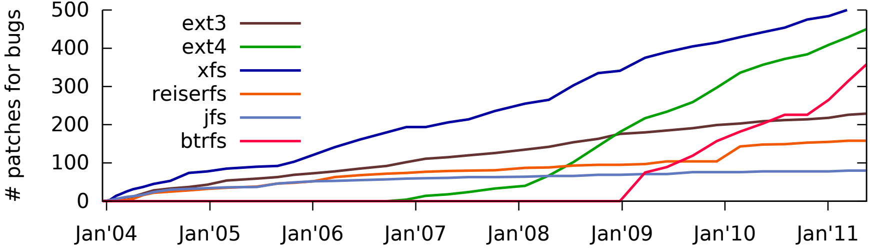


Cumulative number of patches for file-system bugs in Linux; data from [Lu et al., FAST'13]

New file systems (and bugs) are introduced over time

File systems are complex and have bugs

File systems are complex (e.g., Linux ext4 is ∼60,000 lines of code) and have many bugs:



Cumulative number of patches for file-system bugs in Linux; data from [Lu et al., FAST'13]

New file systems (and bugs) are introduced over time

Some bugs are serious: **security exploits**, **data loss**, etc.

Much research 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]

Much research 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 # bugs

Some elimination of bugs by proving:

- FS without directories [Arkoudas et al. 2004]
- BilbyFS [Keller 2014]
- UBIFS [Ernst et al. 2013]

Much research 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 # bugs

Some elimination of bugs by proving:

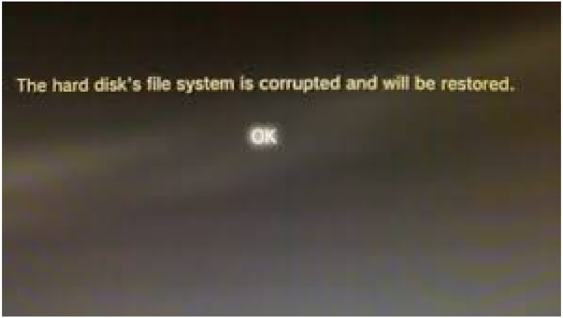
- FS without directories [Arkoudas et al. 2004]
- BilbyFS [Keller 2014]
- UBIFS [Ernst et al. 2013]

} Incomplete + no crashes

Reasoning about crashes is important

File system must recover from crash with data intact

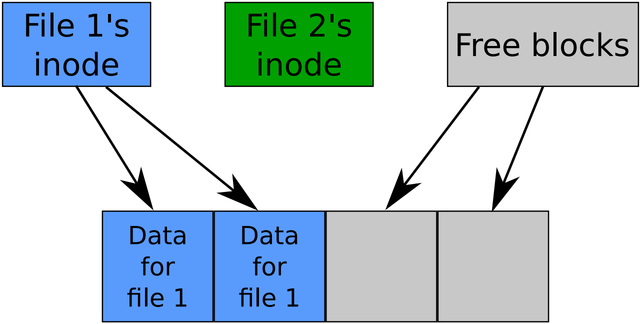
- Crash due to power failure, hardware failures, or software bugs



Risk: crashes expose subtle bugs

Crashes can expose that a file update involves *several* disk writes

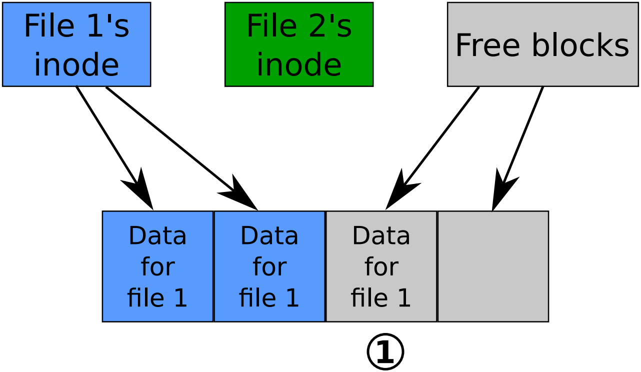
Appending to a file requires three disk writes:



Risk: crashes expose subtle bugs

Crashes can expose that a file update involves *several* disk writes

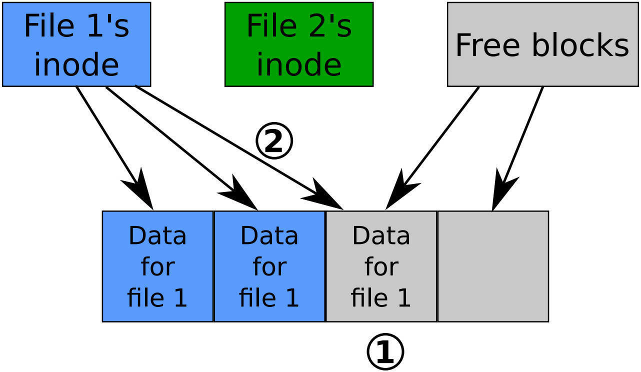
Appending to a file requires three disk writes:



Risk: crashes expose subtle bugs

Crashes can expose that a file update involves *several* disk writes

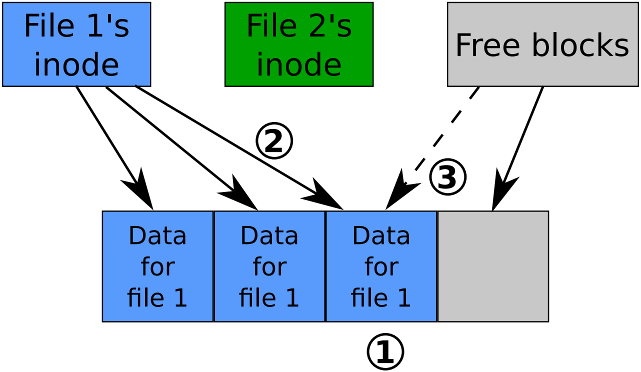
Appending to a file requires three disk writes:



Risk: crashes expose subtle bugs

Crashes can expose that a file update involves *several* disk writes

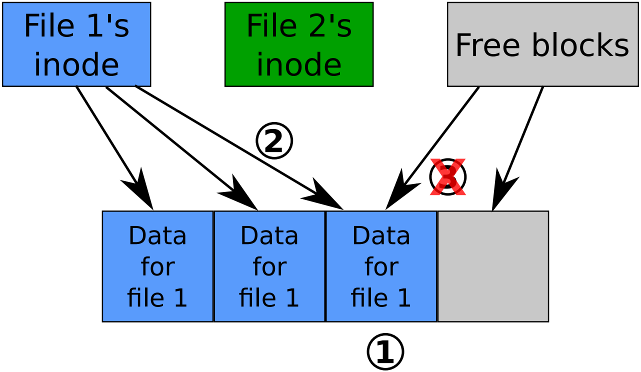
Appending to a file requires three disk writes:



Risk: crashes expose subtle bugs

Crashes can expose that a file update involves *several* disk writes

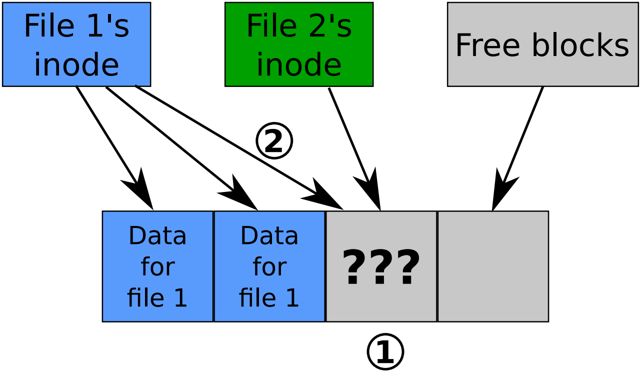
Appending to a file requires three disk writes:



Risk: crashes expose subtle bugs

Crashes can expose that a file update involves *several* disk writes

Appending to a file requires three disk writes:



Developers struggle with partially-updated states

```
commit 353b67d8ced4dc53281c88150ad295e24bc4b4c5
--- 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;
+       }
        /* OK, update the superblock to recover the freed space.
         * Physical blocks come first: have we wrapped beyond the end of
         * the log?  */
```

Mistakes cause data loss [Yang et al. 2006, Pilai et al. 2014, Zheng et al. 2014]

Goal: certify a complete file system under crashes

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

Contributions

CHL: Crash Hoare Logic for persistent storage

- Crash condition and recovery semantics
- CHL automates parts of proof effort
- Proofs mechanically checked by Coq

FSCQ: the first certified crash-safe file system

- Basic Unix-like file system (not parallel)
- Simple specification for a subset of POSIX (e.g., no `fsync`)
- About 1.5 years of work, including learning Coq

FSCQ runs standard Unix programs: mv, git, make, ...

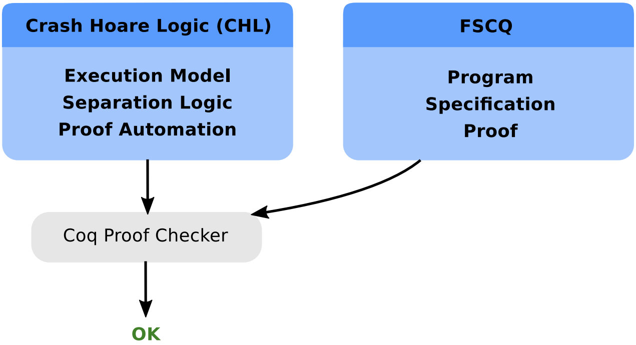
Crash Hoare Logic (CHL)

Execution Model
Separation Logic
Proof Automation

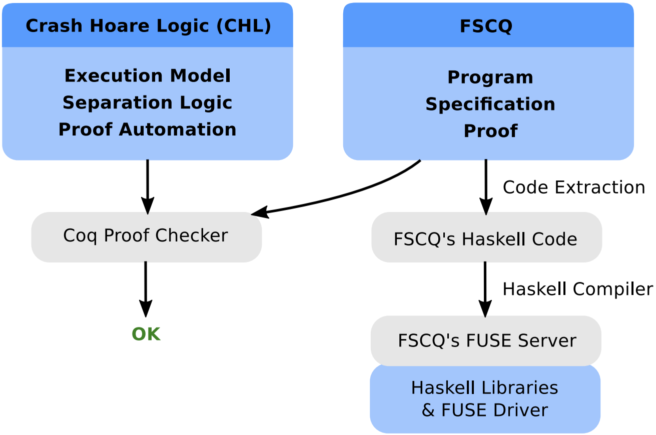
FSCQ

Program
Specification
Proof

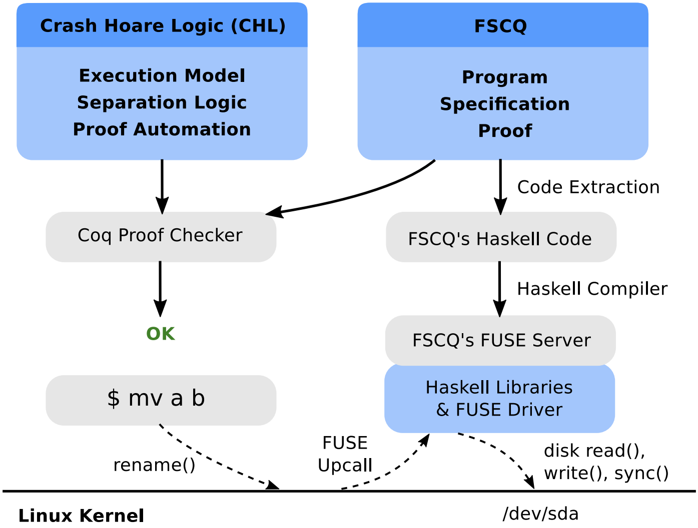
FSCQ runs standard Unix programs: mv, git, make, ...



FSCQ runs standard Unix programs: mv, git, make, ...



FSCQ runs standard Unix programs: `mv`, `git`, `make`, ...



How to specify what is “correct”?

Need a specification of “correct” behavior before we can prove anything

Look it up in the POSIX standard?

How to specify what is "correct"?

Need a specification of "correct" behavior before we can prove anything

Look it up in the POSIX standard?

> *[...] 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

- POSIX's goal was to specify "common-denominator" behavior
- File system implementations have different interpretations
- Leads to bugs in higher-level applications [Pillai et al. OSDI'14]

This work: “correct” is transactional

Run every file-system call inside a transaction

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

This work: “correct” is transactional

Run every file-system call inside a transaction

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

`log_begin` and `log_commit` implement a write-ahead log on disk

After crash, replay any committed transaction in the write-ahead log

This work: "correct" is transactional

Run every file-system call inside a transaction

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

`log_begin` and `log_commit` implement a write-ahead log on disk

After crash, replay any committed transaction in the write-ahead log

Q: How to formally specify both normal-case and crash behavior?

Q: How to specify that it's safe to crash during recovery itself?

Approach: Hoare Logic specifications

{pre} `code` {post}

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

CHL extends Hoare Logic with crash conditions

{pre} `code` {post}
{crash}

SPEC	disk_write(a, v)
PRE	$a \mapsto v_0$
POST	$a \mapsto v$
CRASH	$a \mapsto v_0 \lor a \mapsto v$

CHL's disk model matches what most other file systems assume:

- writing a single block is an atomic operation
- no data corruption

Disk model axiom specs: `disk_write`, `disk_read`, and `disk_sync`

Certifying larger procedures

pre

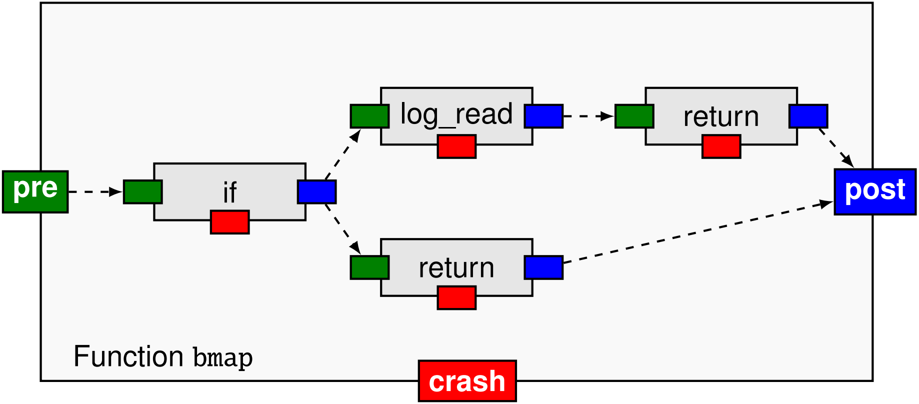
```
def bmap(inode, bnum):
    if bnum >= NDIRECT:
        indirect = log_read(inode.blocks[NDIRECT])
        return indirect[bnum - NDIRECT]
    else:
        return inode.blocks[bnum]
```

post

crash

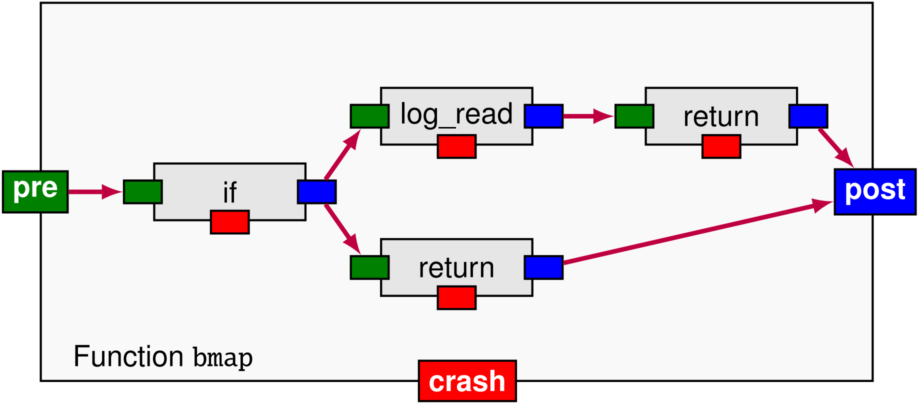
Certifying larger procedures

Need pre/post/crash conditions for each called procedure



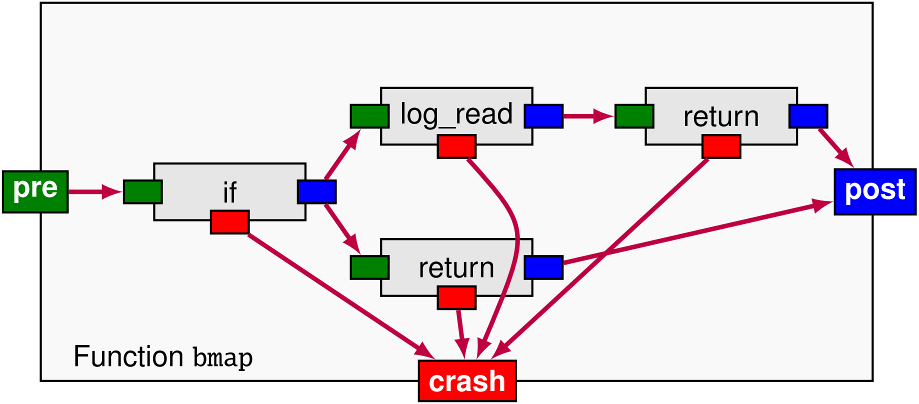
Certifying larger procedures

CHL's proof automation chains pre- and postconditions



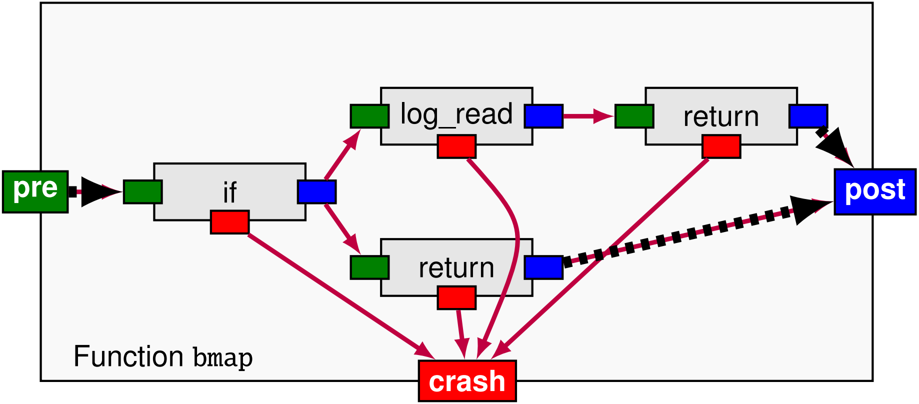
Certifying larger procedures

CHL's proof automation combines crash conditions



Certifying larger procedures

Remaining proof effort: changing representation invariants



Common pattern: representation invariant

SPEC	log_write(*a*, *v*)
PRE	**disk**: log_rep(ActiveTxn, *start_state*, *old_state*)
	old_state: $a \mapsto v_0$
POST	**disk**: log_rep(ActiveTxn, *start_state*, *new_state*)
	new_state: $a \mapsto v$
CRASH	**disk**: log_rep(ActiveTxn, *start_state*, *any*)

`log_rep` is a representation invariant

- Connects logical transaction state to an on-disk representation
- Describes the log's on-disk layout using many $\mapsto$ primitives

Specifying an entire system call (simplified)

SPEC create(*dnum*, *fn*)
PRE **disk**: log_rep(NoTxn, *start_state*)
start_state: dir_rep(*tree*) $\wedge$
$\exists$ *path*, *tree*[*path*].inode = *dnum* $\wedge$
fn $\notin$ *tree*[*path*]

Specifying an entire system call (simplified)

SPEC create(*dnum*, *fn*)
PRE **disk**: log_rep(NoTxn, *start_state*)
start_state: dir_rep(*tree*) $\wedge$
$\exists$ *path*, *tree*[*path*].inode = *dnum* $\wedge$
fn $\notin$ *tree*[*path*]
POST **disk**: log_rep(NoTxn, *new_state*)
new_state: dir_rep(*new_tree*) $\wedge$
new_tree = *tree*.update(*path*, *fn*, empty_file)

Specifying an entire system call (simplified)

SPEC create(*dnum*, *fn*)

PRE **disk**: log_rep(NoTxn, *start_state*)

start_state: dir_rep(*tree*) $\wedge$
$\exists$ *path*, *tree*[*path*].inode = *dnum* $\wedge$
fn $\notin$ *tree*[*path*]

POST **disk**: log_rep(NoTxn, *new_state*)

new_state: dir_rep(*new_tree*) $\wedge$
new_tree = *tree*.update(*path*, *fn*, empty_file)

CRASH **disk**: log_rep(NoTxn, *start_state*) $\vee$
log_rep(NoTxn, *new_state*) $\vee$
$\exists$ *s*, log_rep(ActiveTxn, *start_state*, *s*) $\vee$
log_rep(CommittedTxn, *start_state*, *new_state*) $\vee \ldots$

Specifying log recovery

SPEC	log_recover()
PRE	**disk**: log_intact(*last_state*, *committed_state*)
POST	**disk**: log_rep(NoTxn, *last_state*) $\vee$
	log_rep(NoTxn, *committed_state*)
CRASH	**disk**: log_intact(*last_state*, *committed_state*)

`log_recover` is idempotent

- Crash condition implies pre condition
- $\Rightarrow$ OK to run `log_recover` *again* after a crash

CHL's recovery semantics

`create` is atomic, if `log_recover` runs after every crash:

SPEC create(*dnum*, *fn*)
ON CRASH log_recover()
PRE **disk**: log_rep(NoTxn, *start_state*)
start_state: dir_rep(*tree*) $\wedge$
$\exists$ *path*, *tree*[*path*].inode = *dnum* $\wedge$
fn $\notin$ *tree*[*path*]
POST **disk**: log_rep(NoTxn, *new_state*)
new_state: dir_rep(*new_tree*) $\wedge$
new_tree = *tree*.update(*path*, *fn*, empty_file)
RECOVER **disk**: log_rep(NoTxn, *start_state*) $\vee$
log_rep(NoTxn, *new_state*)

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

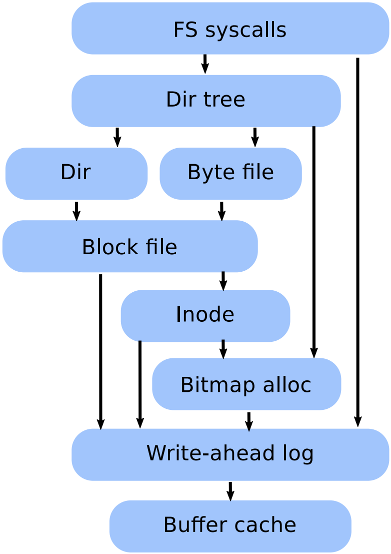
CHL cost: must write crash condition for every function, loop, etc.

- Crash conditions are often simple (above logging layer)

FSCQ: building a file system on top of CHL

File system design is close to v6 Unix, plus logging, minus symbolic links

Implementation aims to reduce proof effort



Reducing proof effort

Reuse proven components

- E.g., finding a free object in a bitmap allocator
- Typical C code: iterate over each 64-bit chunk in a 4KB block, use bitwise operations to find a zero bit
- Less proof effort: use marshaling library; decode bitmap block into 32,768-element array of 1-bit elements; loop over array

Many precise internal abstraction layers

- Files: inode; block-level file; byte-level file
- Directory: directory entries; filename encoding; tree structure

Simpler specifications

- No hard links $\Rightarrow$ logical state is a tree, not a graph

Evaluation

What bugs do FSCQ's theorems eliminate?

How much development effort is required for FSCQ?

How well does FSCQ perform?

FSCQ's theorems eliminate many bugs

One data point: once theorems proven, no implementation bugs

- 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

FSCQ's theorems eliminate many bugs

One data point: once theorems proven, no implementation bugs

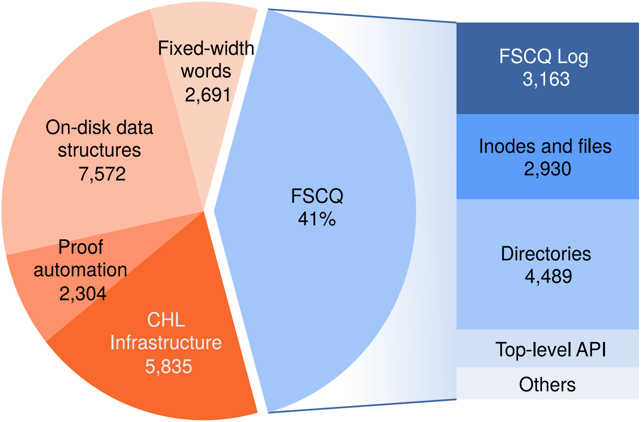
- Did find some mistakes in spec, as a result of end-to-end checks
- E.g., forgot to specify that extending a file should zero-fill

Common classes of bugs found in Linux file systems:

Bug class	**Eliminated in FSCQ?**
Violating file or directory invariants	Yes
Improper handling of corner cases	Yes
Returning incorrect error codes	Some
Resource-allocation bugs	Some
Mistakes in logging and recovery logic	Yes
Misusing the logging API	Yes
Bugs due to concurrent execution	No concurrency
Low-level programming errors	Yes

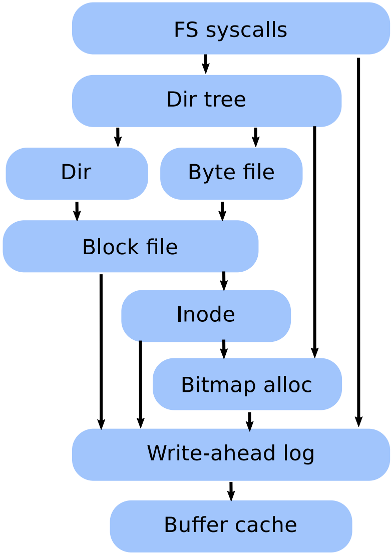
Implementing CHL and FSCQ in Coq

Total of ∼30,000 lines of **verified** code, specs, and proofs
Comparison: xv6 file system is ∼3,000 lines of code



Change effort proportional to scope of change

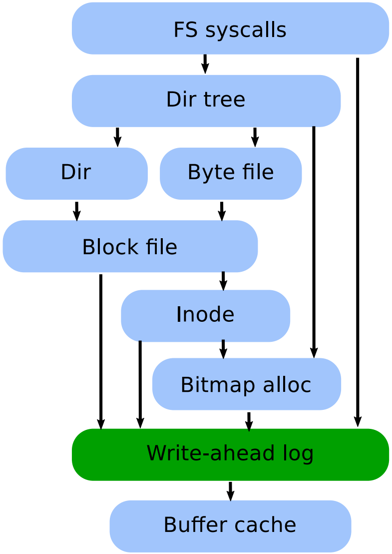
- Reordering disk writes:
  ~1,000 lines in FSCQLOG
- Indirect blocks:
  ~1,500 lines in inode layer
- Buffer cache:
  ~300 lines in FSCQLOG,
  ~600 lines in rest of FSCQ
- Optimize log layout:
  ~150 lines in FSCQLOG



Modest incremental effort, partially due to CHL's proof automation and FSCQ's internal layers

Change effort proportional to scope of change

- **Reordering disk writes:**
  ~1,000 lines in FSCQLOG
- Indirect blocks:
  ~1,500 lines in inode layer
- Buffer cache:
  ~300 lines in FSCQLOG,
  ~600 lines in rest of FSCQ
- Optimize log layout:
  ~150 lines in FSCQLOG

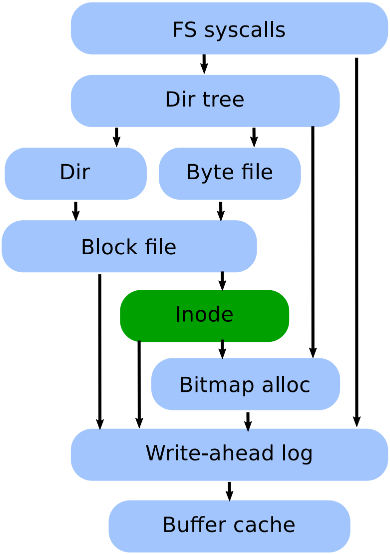


Modest incremental effort, partially due to CHL's proof automation and FSCQ's internal layers

Change effort proportional to scope of change

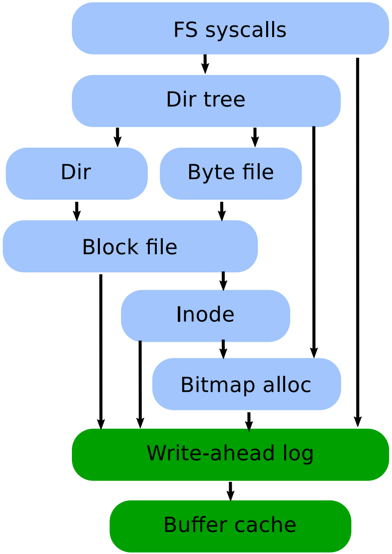
- Reordering disk writes:
  ~1,000 lines in FscqLog
- **Indirect blocks:**
  ~1,500 lines in inode layer
- Buffer cache:
  ~300 lines in FscqLog,
  ~600 lines in rest of FSCQ
- Optimize log layout:
  ~150 lines in FscqLog

Modest incremental effort, partially due to CHL's proof automation and FSCQ's internal layers



Change effort proportional to scope of change

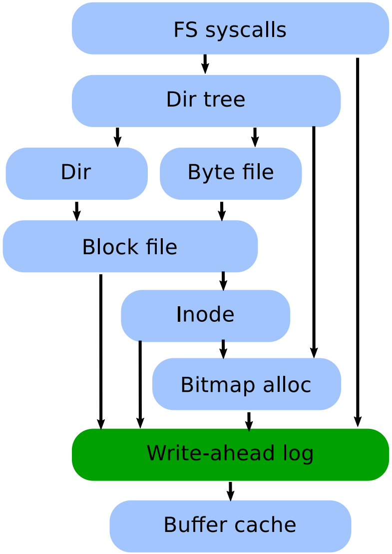
- Reordering disk writes:
  ~1,000 lines in FSCQLOG
- Indirect blocks:
  ~1,500 lines in inode layer
- **Buffer cache:**
  ~300 lines in FSCQLOG,
  ~600 lines in rest of FSCQ
- Optimize log layout:
  ~150 lines in FSCQLOG



Modest incremental effort, partially due to CHL's proof automation and FSCQ's internal layers

Change effort proportional to scope of change

- Reordering disk writes:
  ~1,000 lines in FscqLog
- Indirect blocks:
  ~1,500 lines in inode layer
- Buffer cache:
  ~300 lines in FscqLog,
  ~600 lines in rest of FSCQ
- **Optimize log layout:**
  ~150 lines in FscqLog



Modest incremental effort, partially due to CHL's proof automation and FSCQ's internal layers

Performance comparison

File-system-intensive workload

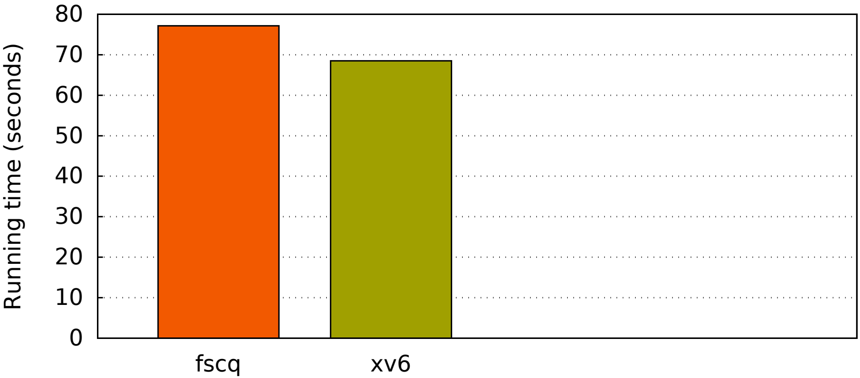
- Software development: git, make
- LFS benchmark
- `mailbench`: qmail-like mail server

Compare with other (non-certified) file systems

- xv6 (similar design, written in C)
- ext4 (widely used on Linux), in non-default *synchronous* mode to match FSCQ's guarantees

Running on an SSD on a laptop

Running time for benchmark workload



- FSCQ slower than xv6 due to overhead of extracted Haskell

Running time for benchmark workload

- FSCQ slower than xv6 due to overhead of extracted Haskell
- FSCQ slower than ext4 due to simple write-ahead logging design

Opportunity: change semantics to defer durability

- FSCQ slower than xv6 due to overhead of extracted Haskell
- FSCQ slower than ext4 due to simple write-ahead logging design
- Deferred durability (ext4's default mode) allows for big improvement

Directions for future research

Formalizing deferred durability (e.g., `fsync`)

Certifying a parallel (multi-core) file system

Certifying applications with CHL (database, key-value store, ...)

Conclusions

CHL helps specify and prove crash safety

- Crash conditions
- Recovery execution semantics

FSCQ: first certified crash-safe file system

- Usable performance
- 1.5 years of effort, including learning Coq and building CHL

Many open problems and potential for fundamental contributions

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