Building a file system with FSCQ infrastructure

Haogang Chen

Abstract

FSCQ is a file system with a machine-checkable proof (using the Coq proof assistant [2]) that its implementation meets its specification, even under crashes. FSCQ provably avoids bugs that have plagued previous file systems, such as performing disk writes without sufficient barriers or forgetting to zero out directory blocks. FSCQ built upon the *Crash Hoare Logic* (CHL) infrastructure. FSCQ uses FscqLog for crash recovery, which provides transactional disk abstraction and all-or-nothing atomicity in case of crash.

1 Overview

As a case study of using the CHL infrastructure, we built and certified the FSCQ file system. Figure 1 shows the overall components that make up FSCQ, and Figure 2 shows FSCQ's disk layout. FSCQ's design closely follows the xv6 file system. The key differences are the lack of multiprocessor support and the use of a separate bitmap for allocating inodes (instead of using a particular inode type to represent a free state). The layout block contains information about where all other parts of the file system are located on disk and is initialized by `mkfs`.

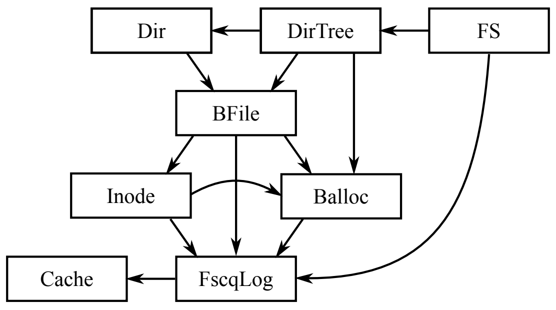

Figure 1: FSCQ components. Arrows represent function calls.

Layout block	File data	Block bitmaps	Inodes	Inode bitmaps	Log length	Log header	Log data

Figure 2: FSCQ on-disk layout.

FSCQ relies on FscqLog [1] for crash recovery. The other components provide simple implementations of standard file-system abstractions. The Cache module provides a buffer cache. Balloc implements a bitmap allocator, used for both block and inode allocation. Inode implements an inode layer; the most interesting logic here is combining the direct and indirect blocks together into a single list of block addresses. Inode invokes Balloc to allocate indirect blocks. BFile implements a block-level file interface, exposing to higher levels an interface where each file is a list of blocks. BFile invokes Balloc to allocate file data blocks. Dir implements directories on top of block-level files. DirTree combines directories and files into a hierarchical directory-tree structure; it invokes Balloc to allocate/deallocate inodes when creating/deleting files or subdirectories. Finally, FS implements complete system calls in transactions.

The rest of the paper describes the challenges we faced in specifying and proving FSCQ and the design patterns that we came up with for addressing them.

2 Using FscqLog

By building on FscqLog, we can factor out crash recovery. FSCQ updates the disk only through `log_write` and wraps those writes into transactions at the system-call granularity to achieve crash tolerance. For example, FSCQ wraps each system call like `open`, `unlink`, etc, in a FscqLog transaction, which allows us to prove that the entire system call is atomic. That is, we can prove that the modifications a system call makes to the disk (e.g., allocating a block to grow a file, then writing that block, and so on) all happen or none happen, even if the system call fails due to a crash after issuing some `log_writes`.

Furthermore, although FscqLog must deal with the complexity of asynchronous writes, it presents to higher-level software a simpler synchronous interface, because transactions hide the asynchrony by providing all-or-nothing atomicity. We were able to do this because the transaction API exposes an *abstract* name space that maps each block to unique block contents, even though the physical disk maps each block to a set of outstanding writes. As a result, software written on top of FscqLog does not have to worry about asynchronous writes.

After a crash, FSCQ's recovery program reads the layout block to determine where the log is located, and invokes FscqLog's `log_recover` to bring the disk to a consistent state.

3 Using name spaces

Since transactions take care of crashes, the remaining challenge lies in specifying the behavior of a file system and proving that the implementation meets its specification on a reliable disk. CHL's name spaces help express predicates about name spaces at different levels of abstraction. For example, consider the specification shown in Figure 3 for `file_block_write`, which writes to an existing block in a file. This specification uses separation logic in four different name spaces: the bare disk (which implements asynchronous writes and matches the `log_rep` predicate); the abstract disks inside the transaction, *old_state* and *new_state* (which have synchronous writes and match the `file_rep` predicate); the name space of files named by inode number, *old_files* and *new_files*; and finally the name space of file blocks named by offset, *old_f*.data and *new_f*.data. The use of separation logic within each name space allows us to concisely specify the behavior of `file_block_write` at all these levels of abstraction. Furthermore, CHL applies its proof automation machinery to separation logic in *every* name space. This helps developers construct short proofs about higher-level abstractions.

SPEC file_block_write($inum$, $blknum$, v)
PRE **disk**: log_rep(ActiveTxn, $start\_state$, old_state)
old_state: file_rep(old_files) $\star$ $other\_state$
old_files: $inum \mapsto old\_f \star other\_files$
old_f.data: $blknum \mapsto v_0 \star other\_blocks$
POST **disk**: log_rep(ActiveTxn, $start\_state$, new_state)
new_state: file_rep(new_files) $\star$ $other\_state$
new_files: $inum \mapsto new\_f \star other\_files \wedge$
new_f.attr = old_f.attr
new_f.data: $blknum \mapsto v \star other\_blocks$
CRASH **disk**: log_rep(ActiveTxn, $start\_state$, any)

Figure 3: Specification for writing to a file.

4 Resource allocation

File systems must implement resource allocation at multiple levels of abstraction—in particular, allocating disk blocks and allocating inodes. We built and proved correct a common allocator in FSCQ. It works by storing a bitmap spanning several contiguous blocks, with bit i corresponding to whether object i is available. FSCQ instantiates this allocator for both disk-block and inode allocation, each with a separate bitmap.

Writing a naïve specification of the allocator is straightforward: freeing an object adds it to a list of free objects, and allocating returns one of these objects. The allocator's representation invariant asserts that the free list is correctly encoded using "one" bits in the on-disk bitmap. However, the caller of the allocator must prove more complex statements—for example, that any object obtained from the allocator is not already in use elsewhere. Re-proving this property from first principles each time the allocator is used is labor-intensive.

To address this problem, FSCQ's allocator provides a `free_objects_pred`(*objlist*) predicate that can be applied to the name space whose resources are being allocated. This predicate is defined as a sequence of ($\exists v,\ i \mapsto v$) predicates for each i in *objlist*, combined using the $\star$ operator. *objlist* is typically the allocator's list of free object IDs, so this predicate states that every free object ID points to some value.

Using `free_objects_pred` simplifies reasoning about resource allocation, because it can be combined with other predicates about the objects that are currently in use (e.g., disk blocks used by files), to give a complete description of the name space in question. The disjoint nature of the $\star$ operator precisely capture the idea that all objects are either available (and managed by the allocator) or are in use (and match some other predicate about the in-use objects).

$$\begin{aligned}\texttt{file\_rep}(files) \; := \; & \exists\, free\_blocks,\ \exists\, inodes, \\ & \texttt{allocator\_rep}(free\_blocks) \star \\ & \texttt{inode\_rep}(inodes) \star \\ & \texttt{file\_block\_rep}(inodes, files) \star \\ & \texttt{free\_objects\_pred}(free\_blocks)\end{aligned}$$

Figure 4: Representation invariant for FSCQ's file layer.

For example, Figure 4 shows the representation invariant for FSCQ's file layer, which is typically applied to FscqLog's abstract disk name space, as shown in Figure 3. The abstract disk, according to Figure 4, is split up into four disjoint parts: the allocation bitmap (represented by `allocator_rep`), the inode area (represented by `inode_rep`), file data blocks (represented by `file_block_rep`), and free blocks (described by `free_objects_pred`). The allocator's representation invariant (`allocator_rep`) connects the on-disk bitmap to the list of available blocks (*free_blocks*). The `file_block_rep` function combines the inode state in *inodes* (containing a list of block addresses for each inode) and the logical file state *files* to produce a predicate describing the blocks currently used by all files. Finally, `free_objects_pred` asserts that the free blocks are disjoint from blocks used by the other three predicates.

The same pattern applies to allocating inodes as well. The only difference is that, in `file_rep`, the predicate describing the actual bitmap, `allocator_rep`, and the predicate describing the available objects, `free_objects_pred`, were both applied to the same name space (the abstract disk). In the case of inodes, the two predicates are applied to different name spaces: the bitmap predicate is applied to the abstract disk, but `free_objects_pred` is applied to the inode name space.

5 On-disk data structures

Another common task in a file system is to lay out data structures in disk blocks. For example, this shows up when storing several inodes in a block; storing directory entries in a file; storing addresses in the indirect block; and even storing individual bits in the allocator bitmap blocks. To factor out this pattern, we built the `Rec` library for packing and unpacking data structures into bit-level representations. We often use this library to pack multiple fields of a data structure into a single bit vector (e.g., the bit-level representation of an inode), and then to pack several of these bit-vectors into one disk block.

```
Definition inode_type : Rec.type := Rec.RecF ([
  ("len",  Rec.WordF addrlen);    (* #blocks *)
  ("attr", iattr_type);           (* file attrs *)
  ("iptr", Rec.WordF addrlen);    (* indirect ptr *)
  ("blks", Rec.ArrayF 5 (Rec.WordF addrlen))]).
```

Figure 5: FSCQ's on-disk inode layout.

For example, Figure 5 shows FSCQ's on-disk inode structure, in Coq syntax. The first field is `len`, storing the number of blocks in the inode, as a 64-bit integer (`Rec.WordF` indicates a word field, and `addrlen` is 64). The other fields are the file's attributes (such as the modification time), the indirect block pointer `iptr`, and a list of 5 direct block addresses, `blks`.

The library proves basic theorems, such as the fact that accesses to different fields are commutative, that reading a field returns the last write, and that pickling and unpickling are inverses of each other. As a result, code using these records does not have to prove low-level facts about layout in general.

6 POSIX specification

FSCQ provides a POSIX-like interface at the top level; the main differences from POSIX are (i) that FSCQ does not support hard links, and (ii) that FSCQ does not implement file descriptors and instead requires naming open files by inode number. FSCQ relies on the FUSE driver to maintain the mapping between open file descriptors and inode numbers.

Each system call implemented by FSCQ comes with a proven specification capturing the expected behavior of the system call and providing all-or-nothing atomicity with respect to crashes. Providing precise specifications for file-system operations is important for applications that need to implement application-level crash consistency on top of a file system [3].

For example, Figure 6 shows FSCQ's specification for its most complicated system call, `rename`, in combination with FSCQ's recovery program `fs_recover`. `rename`'s precondition requires that the directory tree is in a consistent state, matching the `tree_rep` invariant, and that the caller's current working directory inode, *cwd_ino*, corresponds to some valid path name in the tree. The postcondition asserts that `rename` will either return an error, with the tree unchanged, or succeed, with the new tree being logically described by the functions `tree_prune`, `tree_graft`, etc. These functions operate on a logical representation of the directory tree structure, rather than on low-level disk representations, and are defined in a few lines of code each. In case of a crash, the state will either have no effects of `rename` or will be as if `rename` had finished.

SPEC rename(*cwd_ino*, *oldpath*, *newpath*) ≫ fs_recover
PRE **disk**: log_rep(NoTxn, *start_state*)
  start_state: tree_rep(*old_tree*) ∧
    find_subtree(*old_tree*, *cwd*) = *cwd_tree* ∧
    tree_inum(*cwd_tree*) = *cwd_ino*
POST **disk**: ((*ret* = (COMPLETED, NoErr) ∨ *ret* = RECOVERED) ∧
      log_rep(NoTxn, *new_state*)) ∨
    ((*ret* = (COMPLETED, Error) ∨ *ret* = RECOVERED) ∧
      log_rep(NoTxn, *start_state*))
  new_state: tree_rep(*new_tree*) ∧
    mover = find_subtree(*cwd_tree*, *oldpath*) ∧
    pruned = tree_prune(*cwd_tree*, *oldpath*) ∧
    grafted = tree_graft(*pruned*, *newpath*, *mover*) ∧
    new_tree = update_subtree(*old_tree*, *cwd*, *grafted*)

Figure 6: Specification for rename with recovery.

7 Prototype implementation

We implemented, specified, and proved FSCQ correct using Coq and the CHL infrastructure. Figure 7 breaks down the source code of FSCQ and CHL. Proofs are interleaved with source code. The development effort took several researchers one year. Checking the proofs takes four hours on a 2 GHz Intel CPU with 8 GB DRAM.

Component	Lines of code
Fixed-width words	2,138
CHL infrastructure	4,835
Proof automation	2,164
On-disk data structures	2,999
Buffer cache	534
FscqLog	3,200
Bitmap allocator	435
Inodes and files	2,302
Directories	4,666
FSCQ's top-level API	721
Total	23,994

Figure 7: Combined lines of code and proof for FSCQ components.

8 Evaluation

This section answers how difficult is it to build and evolve the code and proofs for FSCQ.

One metric to evaluate the development effort is the size of the FSCQ code base, as reported in Figure 7; FSCQ consists of about 24,000 lines of code. In comparison, the xv6 file system is about 3,000 lines of C code, so FSCQ is about 8× larger, but this includes a significant amount of common CHL infrastructure, including libraries and proof machinery, which is not FSCQ-specific.

A more interesting question is how much effort is required to *modify* FSCQ, after an initial version has been developed and certified. Does adding a new feature to FSCQ require re-proving everything, or is the work commensurate with the scale of the modifications required to support the new feature? To answer this question, the rest of this section presents several case studies, where we had to add a significant feature to FSCQ after the initial design was already complete.

Indirect blocks. Initially, FSCQ supported only direct blocks. Adding indirect blocks required changing about 1,500 lines of code and proof in the Inode layer, including infrastructure changes for reasoning about on-disk objects that span multiple disk blocks (the inode and its indirect block). We made almost no changes to code above the Inode layer; the only exception was BFile, in which we had to fix about 50 lines of proof due to a hard-coded constant bound for the maximum number of blocks per file.

Buffer cache. We added a buffer cache to FSCQ after we had already built FscqLog and several layers above it. Since Coq is a pure functional language, keeping buffer-cache state required passing the current buffer-cache object to and from all functions. Incorporating the buffer cache required changing about 300 lines of code and proof in FscqLog, to pass around the buffer-cache state, to access disk via the buffer cache, and to reason about disk state in terms of buffer-cache invariants. We also had to make similar straightforward changes to about 600 lines of code and proof for components above FscqLog.

9 Discussion

Although FSCQ isn't as complete and high-performance as today's high-end file systems, our results demonstrate that this is largely due to FSCQ's simple design, and not any inherent limitations of certified software. Furthermore, the case studies in §8 indicate that one can improve FSCQ incrementally. In future work we hope to improve FSCQ's logging to batch transactions and to log only metadata; we expect this will bring FSCQ's performance closer to ext3 in its standard deployment mode. The one exception to incremental improvement is multiprocessor support, which may require global changes, and is an interesting direction for future research.

References

[1] H. Chen, D. Ziegler, A. Chlipala, M. F. Kaashoek, E. Kohler, and N. Zeldovich. Specifying crash safety for storage systems. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS)*, Kartause Ittingen, Switzerland, May 2015.

[2] Coq development team. *Coq Reference Manual, Version 8.4pl5*. INRIA, Oct. 2014. http://coq.inria.fr/distrib/current/refman/.

[3] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 433–448, Broomfield, CO, Oct. 2014.