Verifying the FAT32 filesystem in ACL2

Mihir Mehta

Department of Computer Science
University of Texas at Austin

mihir@cs.utexas.edu

06 April, 2018

Outline

Outline

Why we need a verified filesystem

- Filesystems are everywhere, even as operating systems move towards making them invisible.
- In the absence of a clear specification of filesystems, users (and sysadmins in particular) are underserved.
- Modern filesystems have become increasingly complex, and so have the tools to analyse and recover data from them.
- It would be worthwhile to specify and formally verify, in the ACL2 theorem prover, the guarantees claimed by filesystems and tools.

Why FAT32?

- FAT32 was widely used in the days of Windows 2000; while the closely related FAT12 and FAT16 have mostly fallen into disuse, FAT32 continues to be used in removable media.
- It offers a simple design compared to other filesystems; while it lacks journalling, arguably an essential feature, it's a much easier target for verification.
- It's also not as unsophisticated as, say, the CP/M filesystem which doesn't support subdirectories. Thus, a verification effort for FAT32 can form a basis for verifying more complex filesystems.

Outline

Refinement mappings (Abadi, 1991)

- For a pair of transition systems S_1 and S_2, S_1 is said to *implement* S_2 if every externally visible behaviour allowed by S_1 is also allowed by S_2.
- This implementation relation can be proved if (but not only if) a *refinement mapping* can be discovered that maps each (state, transition) pair of S_1 to a legal (state, transition) pair of S_2.
- Proving an implementation relation of this kind, by starting with a simple system which can easily be reasoned about and refining one or more times to get to a system of sufficient complexity, is often a more tractable alternative to proving the correctness of the complex system from scratch.

Modelling a filesystem

- We opt to iteratively model a filesystem, incrementally adding features of FAT32.
- This also allows us to prove correctness in an iterative fashion, by proving equivalences and thereby reusing correctness results from previous models.
- Apart from noting that some of these models were based on the previous technology target, the CP/M filesystem, we will not dwell on these.

Modelling a filesystem

- In our most recent model, we separate our filesystem into:
  - a tree, in which non-leaf nodes represent (sub)directories and leaf nodes represent regular files;
  - a disk, containing the textual contents of regular files broken into fixed-size blocks;
  - and a file allocation table, mapping each block in a regular file to the next, this allowing us to read the contents of the entire file.

Verifying the model

- We've focussed so far on two filesystem properties, known in the literature as the *read-over-write* properties.
  1. After a write of some text at some location, a read of the same length at the same location should yield the text.
  2. After a write, a read at a different location should yield the same results as a read before the write.
- These properties have been proven for all models so far, including the present model which features a file allocation table.

Proof challenges

- How do we define a "good state" of a filesystem, which shows that reading, writing and other operations can be safely carried out?
- Answering this question involves a trade-off between simplicity (to help with verification) and generality (to model as many real-world situations as possible.)
- We choose to require:
  - that each block on the disk is attributed to at most one regular file;
  - that the clusters attributed to each non-empty regular file end with a legal EOF value, as defined by the FAT specification.
  - that each regular file is annotated with "length", a metadata field that corresponds to the actual length of the file as determined by traversing the file allocation table and reading the corresponding blocks.

Validating the model

Future work

1. Complete the FAT32 model, by means of
   - supporting variable cluster sizes,
   - moving the file allocation table onto the disk, and
   - moving all file and directory metadata from the tree to the disk.
2. Model a more complex filesystem, for instance NTFS, by re-using algorithms and proofs from the models built so far.

Outline

Related work

- FSCQ (Chen, 2016) - Coq is used to build a filesystem , proven safe against crashes in a new logical framework named Crash Hoare Logic. Implementation (exported to Haskell) performs comparably to ext4.
- SibylFS (Ridge, 2015) - a "verifying compiler" of sorts is provided to translate specs in a DSL to C implementations.
- Hyperkernel (Nelson, 2017) - xv6 microkernel is implemented with system calls changed to make them constant-time, in return, verification burden becomes lightweight enough for Z3 SMT solver.
- In our work, we instead aim to model an existing filesystem (FAT32) faithfully and match the resulting disk image byte-to-byte.