Verifying filesystems in the ACL2 theorem prover: an application to FAT32

Mihir Parang Mehta

University of Texas at Austin, Department of Computer Science,
2317 Speedway, Austin, TX 78712, USA

Abstract. We describe an effort to formally verify the FAT32 filesystem, based on a specification put together from Microsoft's published specification and the Linux kernel source code. We detail our approach of proving properties through refinement of filesystem models. We describe how this work is applicable to more filesystems than solely FAT32, and enumerate possible future applications of these techniques.

Keywords: interactive theorem proving, filesystems

1 Overview

Filesystems are ubiquitous in computing, and they have been of interest to the formal verification community for nearly as long as it has existed. We have chosen to use the ACL2 theorem prover to model FAT32 down to the byte level. By starting with a high-level abstract model and refining [1] it with successive models which add more of the complexity of the real filesystem, we are able to manage the complexity of this proof, which has not yet been attempted. In the rest of this paper, we describe these models and the properties proved with examples; we proceed to a high-level explanation of our refinement proofs; and further we offer some insights about the low-level issues encountered while working the proofs. We end with some statistics pertaining to the magnitude of the proof effort and the running time of the proofs.

2 Related work

Verifying even simple filesystems is a fairly complex task; thus, the pertinent work in the literature has centred on the use of theorem provers rather than on ad hoc verification techniques. In listing a few of these verification endeavours, it's important to note that each is, to some degree, abstracted away from the byte-level reality which is our goal; yet there's a difference between the extent to which interactive and non-interactive theorem provers need these abstractions.

2.1 Interactive theorem provers

An early effort in the filesystem verification domain was by Bevier and Cohen [2], who specified the Synergy filesystem and created an executable model of the same in ACL2 [3], down to the level of processes and file descriptors. On the proof front, they certified their model to preserve well-formedness of their data structures through their various file operations; however, they did not attempt to prove, for instance, read-over-write properties or crash consistency. Later, Klein et al with the SeL4 project [4] used Isabelle/HOL [5] to verify a microkernel; while their design abstracted away file operations in order to keep their trusted computing base small, it did serve as a precursor to their more recent COGENT project [6]. Here the authors built a "verified compiler" of sorts, generating C-language code from specifications in their domain-specific in a manner guaranteed to avoid many common filesystem bugs. Elsewhere, the SibylFS project [7], again using Isabelle/HOL, provided an executable specification for filesystems at a level of abstraction that could function across multiple operating systems including OSX and Unix. The Coq prover [8] has also been used, for instance, for FSCQ [9], a state-of-the art filesystem which was built to have high performance and formally verified crash consistency properties. We note here that these systems, while working from some kind of specification, all produce a completely novel filesystem instead of attempting to specify an existing, widely used filesystem, such as FAT32 which is our technology target.

2.2 Non-interactive theorem provers

Non-interactive theorem provers such as Z3 [10] have also been used; Hyperkernel [11] is a recent effort which focusses on simplifying the xv6 microkernel until the point that Z3 can verify it with its SMT solving techniques. However, towards this end, all system calls in Hyperkernel are replaced with analogs which can terminate in constant time; while this approach is theoretically sound, it increases the chances of discrepancies between the model and the implementation which may diminish the utility of the proofs or even render them moot.

3 The FAT32 filesystem

Microsoft, in its specification [12] defines three closely related filesystems, named FAT12, FAT16 and FAT32 based on the bit-width of entries in their *file allocation table* data structure. Of these, the former two have passed almost into disuse, while FAT32 continues to be used in media of small capacity, such as USB thumb drives.

FAT32, while simple, adds some complexity compared to the filesystems which came before. Regular files, in storage, are divided into *clusters* (sometimes called *extents*) of a fixed size, constrained to be a multiple of the disk sector size, which remains constant for a given volume formatted with FAT32. Directory files are treated much the same way, with the addition of a file attribute that indicates the file is a directory. This clustering is one example of an

optimisation for long contiguous reads and writes; in the literature optimisations such as this have been shown to decrease external fragmentation at the cost of increasing internal fragmentation.

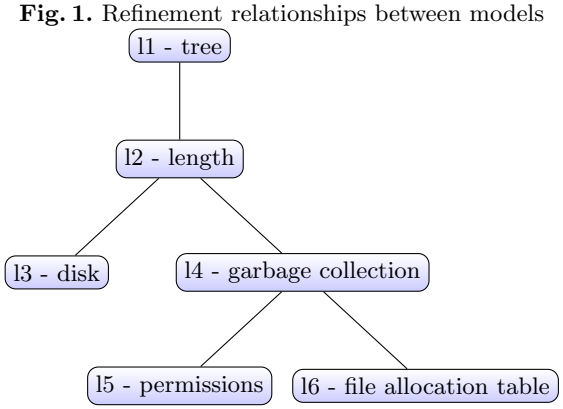
The file allocation table itself is, very simply, a linked list. It maps each cluster index used by a file to either the next cluster index for that file or an end-of-file value defined by the specification. This allows the contents of a file to be reconstructed by reading just the first cluster index from its directory entry, and reconstructing the list of clusters using the table. Unused clusters are mapped to 0 in the table; this is used for counting and allocating free clusters.

4 The models

Table 1. Models and their features

`l1`	The filesystem is represented as a tree, with leaf nodes for regular files and non-leaf nodes for directories. The contents of regular files are represented as strings stored in the nodes of the tree; the storage available for these is unbounded.
`l2`	A single element of metadata, *length*, is stored within each regular file.
`l3`	The contents of regular files are divided into blocks of fixed size. These blocks are stored in an external "disk" data structure; the storage for these blocks remains unbounded.
`l4`	The storage available for blocks is now bounded. An allocation vector data structure is introduced to help allocate and garbage collect blocks.
`l5`	Additional metadata for file ownership and access permissions is stored within each regular file.
`l6`	The allocation vector is replaced by a file allocation table, per the official FAT specification.

Fig. 1. Refinement relationships between models



At this point in development, we have six models of the filesystem, here referred to as `l1` through `l6` (see table 1). Each new model *refines* a previous model, adding some features and complexity, and thereby approaching closer to a model which is binary compatible with FAT32. These refinement relationships are shown in figure 1. `l1` is the simplest of these, representing the filesystem as a literal tree; later models feature file metadata (including ownership and permissions), externalisation of file contents, and allocation/file allocation using an allocation vector after the fashion of the CP/M file system.

Broadly, we characterise the filesystem operations we offer as either *write* operations, which do modify the filesystem, or *read* operations, which do not. In each model, we have been able to prove *read-over-write* properties which show that write operations have their effects made available immediately for reads at the same location, but also that they do not affect reads at other locations.

The first read-after-write theorem states that immediately following a write of some text at some location, a read of the same length at the same location yields the same text. The second read-after-write theorem states that after a write of some text at some location, a read at any other location returns exactly what it would have returned before the write. As an example, listings for the `l1` versions of these theorems follow.

```
(defthm l1-read-after-write-1
  (implies (and (l1-fs-p fs)
                (stringp text)
                (symbol-listp hns)
                (natp start)
                (equal n (length text))
                (stringp (l1-stat hns fs)))
           (equal (l1-rdchs hns (l1-wrchs hns fs start text) start n) text)))

(defthm l1-read-after-write-2
  (implies (and (l1-fs-p fs)
                (stringp text2)
                (symbol-listp hns1)
                (symbol-listp hns2)
                (not (equal hns1 hns2))
                (natp start1)
                (natp start2)
                (natp n1)
                (stringp (l1-stat hns1 fs)))
           (equal (l1-rdchs hns1 (l1-wrchs hns2 fs start2 text2) start1 n1)
                  (l1-rdchs hns1 fs start1 n1))))
```

By composing these properties, we can reason about executions involving multiple reads and writes, as shown in the following throwaway lemma.

```
(thm
```

```
(implies (and (l1-fs-p fs)
              (stringp text1)
              (stringp text2)
              (symbol-listp hns1)
              (symbol-listp hns2)
              (not (equal hns1 hns2))
              (natp start1)
              (natp start2)
              (stringp (l1-stat hns1 fs))
              (equal n1 (length text1)))
         (equal (l1-rdchs hns1
                          (l1-wrchs hns2 (l1-wrchs hns1 fs start1 text1)
                                    start2 text2)
                          start1 n1)
                (l1-rdchs hns1 (l1-wrchs hns1 fs start1 text1)
                          start1 n1))))
```

5 Proof methodology

In *l1*, our simplest model, the read-over-write properties were, of necessity, proven from scratch, with the use of some rather complicated induction schemes. For reference, the following code listing shows the induction scheme used for `l1-read-after-write-2`.

```
(defun induction-scheme (hns1 hns2 fs)
  (if (atom hns1)
      fs
    (if (atom fs)
        nil
      (let ((sd (assoc (car hns2) fs)))
        (if (atom sd)
            fs
          (if (atom hns2)
              fs
            (if (not (equal (car hns1) (car hns2)))
                fs
              (let ((contents (cdr sd)))
                (if (atom (cdr hns1))
                    (cons (cons (car sd)
                                contents)
                          (delete-assoc (car hns2) fs))
                  (cons (cons (car sd)
                              (induction-scheme (cdr hns1) (cdr hns2) contents))
                        (delete-assoc (car hns2) fs)))))))))))
```

In each subsequent model, the read-over-write properties are proven as corollaries of equivalence proofs which establish the correctness of read and write operations in the respective model with respect to a previous model. A representation of such an equivalence proof can be seen in figures 2, 3 and 4, which respectively show the equivalence proof for `l2-wrchs`, the equivalence proof for `l2-rdchs` and the composition of these to obtain the first read-over-write theorem for model `l2`.

Fig. 2. l2-wrchs-correctness-1

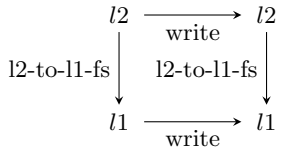


Fig. 3. l2-rdchs-correctness-1

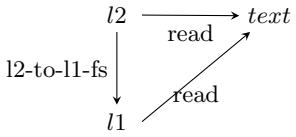
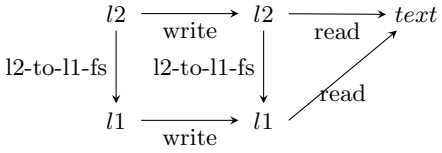


Fig. 4. l2-read-over-write-1



6 Some proof details

6.1 Invariants

As the models grow more complex, with the addition of more auxiliary data the "sanity" criteria for filesystem instances become more complex. For instance, in

l4, the predicate `l4-fs-p` is defined to be the same as `l3-fs-p`, which recursively defines the shape of a valid filesystem. However, a "sane" filesystem requires also that each disk index assigned to a regular file be marked as *used* in the allocation vector, and that it be distinct from other disk indices assigned to files across the filesystem. These properties are invariants to be maintained across write operations; they simplify the verification of read-after-write properties by ensuring that write properties do not create an "aliasing" situation in which a regular file's contents can be modified through a write to a different regular file.

These properties, in the form of the predicates `indices-marked-listp` and `no-duplicatesp`, are packaged together into the `l4-stricter-fs-p` predicate, for which a listing follows.

```
(defun l4-stricter-fs-p (fs alv)
  (declare (xargs :guard t))
  (and (l4-fs-p fs)
       (boolean-listp alv)
       (let ((all-indices (l4-list-all-indices fs)))
            (and (no-duplicatesp all-indices)
                 (indices-marked-p all-indices alv)))))
```

6.2 Performance hacking

As in all ACL2 verification efforts, our work accumulated a number of helper functions and lemmata in the service of the big-picture proofs, and these were prone to slow down our proofs somewhat. Thus, using ACL2's `accumulated-persistence` tool, we made an effort to trim the number of enabled rules by focussing on the rules which the tool suggested to be *useless*. This was important in helping us reduce the certification time for `l6` from 229 seconds to 84 seconds, but from this point onwards results were mixed. As an illustrative example, disabling the function `l6-wrchs` brought down the certification time for l6 from 84 seconds to 60 seconds, yet disabling another function, `l4-collect-all-index-lists`, had a negligible effect on other books and actually served to increase the certification time from 60 seconds to 69 seconds. Needless to say, the latter change was rolled back; a pertinent explanation can be found in the ACL2 documentation topic `accumulated-persistence-subtleties`.

7 Evaluation

At present, the codebase spans 11710 lines of ACL2 code, including 152 function definitions and 616 theorems and lemmas. Some of this data was obtained by David A. Wheeler's sloccount tool.

In table 2 we note the time taken to certify the models in ACL2, as well as some infrastructure upon which the models are built.

Table 2. Time taken to prove models

`l1`	1s
`l2`	5s
`l3`	6s
`l4`	19s
`l5`	21s
`l6`	60s
Misc.	4s

8 Future work

We are pursuing future work in a few different directions.

8.1 FAT32

Having modelled the file allocation table, the next step is to dispense with the tree representation and implement filesystem traversal by looking up entries in directory files. This will yield a model which is entirely contained in a disk data structure and which can further be validated by co-simulation with a FAT32 implementation, such as the one shipped with the Linux kernel.

8.2 Other filesystems

We plan to model a filesystem with journalling in order to prove crash consistency; we are considering ext4 and NTFS. We may also incorporate reasoning about non-determinism in multiprogramming environments where the filesystem is accessed by multiple processes concurrently.

8.3 fsck

Another goal of this work is to provide a basis for reasoning about `fsck` and other tools for sanity checking and recovering data from a filesystem. This is a large part of the motivation for pursuing binary compatibility.

Acknowledgments. This work has been supported by a grant from the NSF.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science **82**(2) (1991) 253–284
2. Bevier, W.R., Cohen, R.M.: An executable model of the synergy file system. Technical report, Technical Report 121, Computational Logic, Inc (1996)
3. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-aided reasoning: an approach. Kluwer Academic Publishers (2000)

4. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al.: sel4: Formal verification of an os kernel. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ACM (2009) 207–220
5. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic. Volume 2283. Springer Science & Business Media (2002)
6. Amani, S., Hixon, A., Chen, Z., Rizkallah, C., Chubb, P., O'Connor, L., Beeren, J., Nagashima, Y., Lim, J., Sewell, T., et al.: Cogent: Verifying high-assurance file system implementations. In: ACM SIGPLAN Notices. Volume 51., ACM (2016) 175–188
7. Ridge, T., Sheets, D., Tuerk, T., Giugliano, A., Madhavapeddy, A., Sewell, P.: Sibylfs: formal specification and oracle-based testing for posix and real-world file systems. In: Proceedings of the 25th Symposium on Operating Systems Principles, ACM (2015) 38–53
8. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: CoqArt: the calculus of inductive constructions. Springer Science & Business Media (2013)
9. Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M.F., Zeldovich, N.: Using crash hoare logic for certifying the FSCQ file system. In Gulati, A., Weatherspoon, H., eds.: 2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016., USENIX Association (2016)
10. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer (2008) 337–340
11. Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., Wang, X.: Hyperkernel: Push-button verification of an os kernel. In: Proceedings of the 26th Symposium on Operating Systems Principles. SOSP '17, New York, NY, USA, ACM (2017) 252–269
12. Microsoft: Microsoft extensible firmware initiative fat32 file system specification (Dec 2000)