Formalising filesystems in the ACL2 theorem prover: an application to a FAT32-like filesystem

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 Introduction and overview

Filesystems are ubiquitous in computing, providing application programs a means to store data persistently, address data by a name instead of a numeric index, and communicate with other programs. Thus, the vast majority of application programs directly or indirectly rely upon filesystems, which makes filesystem verification critically important. Here, we present a formalisation effort in ACL2 for a filesystem with a FAT32-like data organisation, and a proof of the read-over-write properties for this filesystem. 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, to our knowledge, been previously attempted. Thus, this paper contributes a case study in refinement for filesystem verification, and substantial progress towards the goal of a binary-compatible model of a FAT32, which is a real and widely-used filesystem.

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

Filesystem verification research has largely followed a pattern of synthesising a new filesystem based on a specification chosen for its ease in proving properties of interest, rather than similarity to an existing filesystem. Our work, in contrast, follows the FAT32 specification closely. In spirit, our work is closer to previous work which uses interactive theorem provers and explores deep functional properties than to efforts which use non-interactive theorem provers such as Z3 to produce fully automated proofs of simpler properties.

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.

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. A stronger effort in the same domain is Yggdrasil [12], which focusses on verifying filesystems with the use of Z3. While the authors make substantial progress in terms of the number of filesystem calls they support and the crash consistency guarantees they provide, they are subject to the limits of SMT solving which prevent them from modelling essential filesystem features such as extents, which are central to many filesystems including FAT32.

3 Refinement

One traditional approach for verification of complex systems is axiomatic, wherein the desired properties of a system are enumerated and then verified. This is in contrast with refinement, where a system is proved to refine a simpler system, possibly a state machine or a pseudocode program, which is known to show the desired properties either by inspection or by proof. (Note: the term "abstraction" is generally used to denote the inverse relationship to refinement, and we use it in that sense in this paper.) The relative merits of these approaches have been debated in the literature; Lamport [13] makes the argument that the axiomatic style is hopelessly tedious for any but the simplest systems.

In the present verification endeavour, we choose to verify refinement properties in a series of successive models. This is also the approach chosen by Yggdrasil [12]. We consider read-over-write properties to be central, and we prove them to true in all our models; however, these proofs are obtained more or less "for free" once a proof is formulated for the base model. Yet, the value of the refinement approach is attested to by the ease of verification of several incidental properties, such as the ability of write operations to succeed as long as there is sufficient space in a filesystem of finite size.

4 The FAT32 filesystem

FAT32 was initially developed at Microsoft [14] in order to address certain shortcomings of the DOS filesystem previously in use in their operating systems. While it is simple by today's standards, it does add some complexity compared to the filesystems which came before.

All files, including regular files and directory files, are divided into *clusters* (sometimes called *extents*) of a fixed size, which is decided at the time a FAT32 volume is formatted, and constrained to be a multiple of the disk sector size. Directory files differ only in a metadata attribute which indicates that their contents should be treated as a sequence of directory entries. Each such directory entry is 32 bytes wide and contains information including name, size, first cluster index, and access times for the corresponding file.

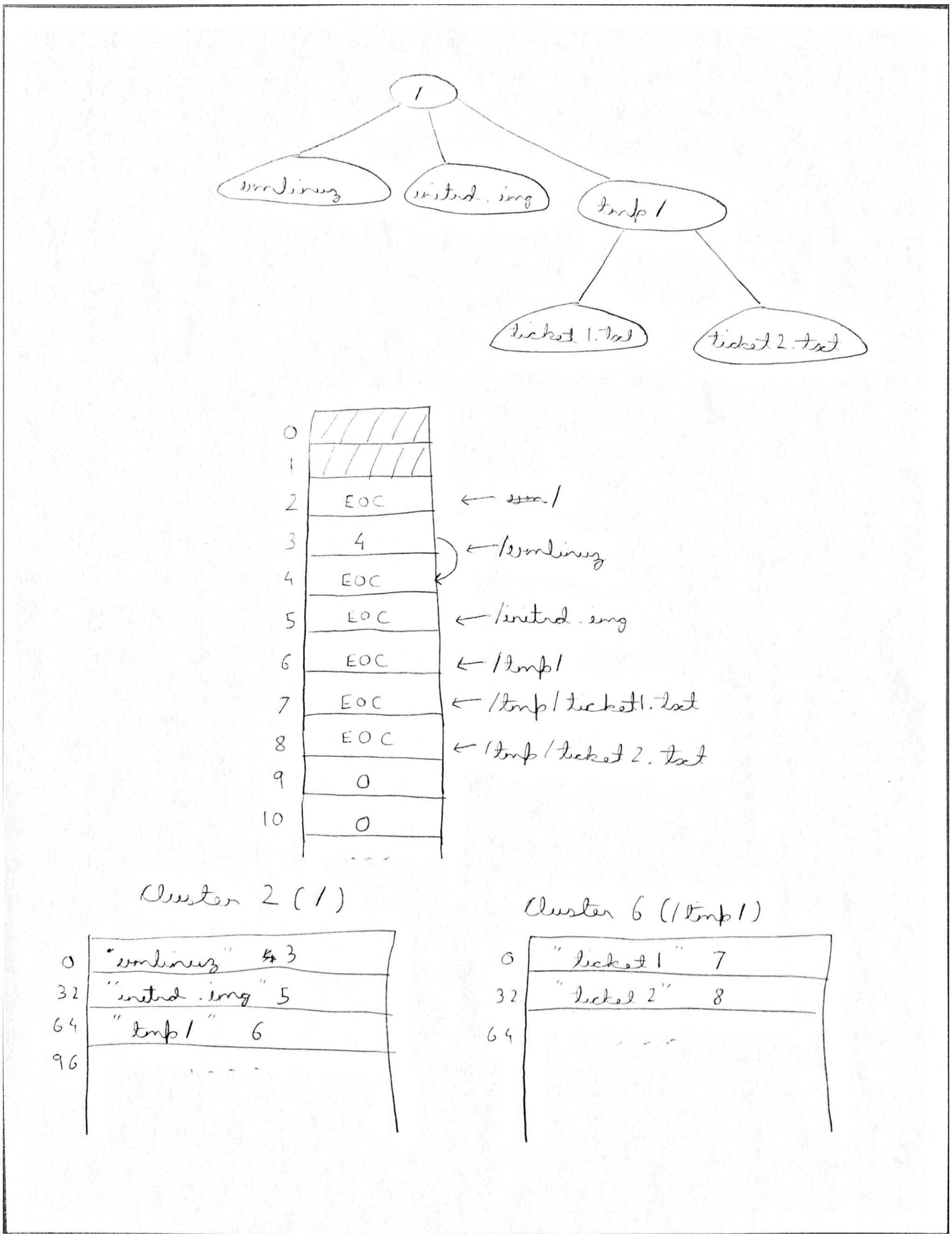
The file allocation table itself contains a number of linked lists. It maps each cluster index used by a file to either the next cluster index for that file or a special end-of-clusterchain value (footnote: there is actually a range of end-of-clusterchain values, not just one.) This allows the contents of a file to be reconstructed by reading just the first cluster index from the corresponding directory entry, and building the list of clusters using the table. Unused clusters are mapped to 0 in the table; this fact is used for counting and allocating free clusters.

We illustrate the file allocation table and data layout for a small directory tree in figure 1.

5 The models

For every read or write operation, FAT32 requires one or more lookups into the file allocation table, followed by the corresponding lookups into the data region.

Fig. 1. A FAT32 directory tree

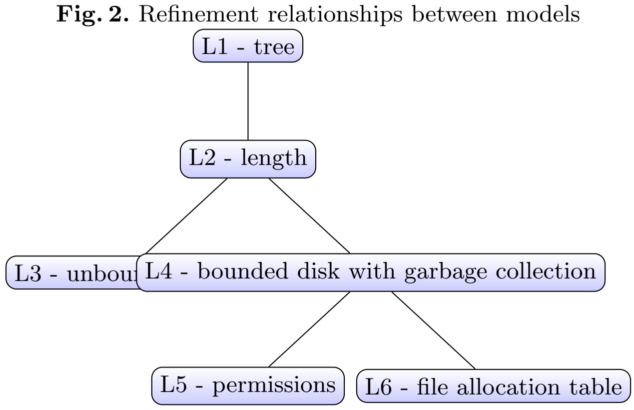


This makes proof efforts about these operations complex, which serves as the motivation for modelling the filesystem in a series of steps.

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, matching the official FAT specification.

Fig. 2. Refinement relationships between models



At this point in development, we have six models of the filesystem, here referred to as L1 through L6, described in table 1. Each model other than L1 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 2. L1 is the simplest of these, representing the filesystem as a literal directory 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 (this is a remnant of an earlier filesystem verification effort for CP/M, which we subsumed into the present work).

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 illustrated in the following throwaway proof.

```
(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))))
```

6 Proof methodology

In L1, our simplest model, the read-over-write properties were, of necessity, proven from scratch.

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 3, 4 and 5, 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. 3. l2-wrchs-correctness-1

$$\begin{tikzcd} l2 \arrow[r, "\text{write}"'] \arrow[d, "\text{l2-to-l1-fs}"'] & l2 \arrow[d, "\text{l2-to-l1-fs}"'] \\ l1 \arrow[r, "\text{write}"'] & l1 \end{tikzcd}$$

Fig. 4. l2-rdchs-correctness-1

$$\begin{tikzcd} l2 \arrow[r, "\text{read}"'] \arrow[d, "\text{l2-to-l1-fs}"'] & text \\ l1 \arrow[ur, "\text{read}"'] & \end{tikzcd}$$

7 Some proof details

We have come to rely on certain principles for the proof effort for each new model. We summarise some of these below.

Fig. 5. l2-read-over-write-1

$$\begin{tikzcd}
l2 \arrow[r, "\text{write}"'] \arrow[d, "\text{l2-to-l1-fs}"'] & l2 \arrow[r, "\text{read}"'] \arrow[d, "\text{l2-to-l1-fs}"'] & \textit{text} \\
l1 \arrow[r, "\text{write}"'] & l1 \arrow[ur, "\text{read}"'] &
\end{tikzcd}$$

7.1 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 directory tree. However, we choose to require two more properties for a "sane" filesystem.

1. Each disk index assigned to a regular file should be marked as *used* in the allocation vector - this is essential to prevent filesystem errors.
2. Each disk index assigned to a regular file should be distinct from all other disk indices assigned to files - this does not hold true, for example, in filesystems with hardlinks, but makes our proofs easier.

These properties are invariants to be maintained across write operations; while not all of them are strictly necessary for a filesystem instance to be valid, they do simplify the verification of read-after-write properties by helping us ensure that write operations 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)))))
```

7.2 Reuse

As noted earlier, using a refinement methodology allows us to derive our read-over-write properties essentially "for free"; more precisely, we are able to prove read-over-write properties simply with `:use` hints after having done the work of proving refinement through induction.

At a lower level, we are also able to benefit from refinement relationships between components of our different models. For example, such a relationship exists between the allocation vector used in `L4` and the file allocation table used in `L6`. More precisely, by taking a file allocation table and mapping each non-zero entry to `true` and each zero entry to `false`, we obtain a corresponding allocation vector with exactly the same amount of available space. This is a refinement mapping which makes it a lot easier to prove that `L4`, which uses an allocation vector, is an abstraction of `L6`, which uses a file allocation table. This, in turn, means that the effort spent on proving the invariants described above for `L4` need not be replicated for `L6`.

7.3 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`.

8 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 books for each model in ACL2 (non-cumulative), as well as some infrastructure upon which the models are built. These results

Table 2. Time taken to prove models

Misc.	4s
`L1`	1s
`L2`	5s
`L3`	6s
`L4`	19s
`L5`	21s
`L6`	60s

9 Conclusion

This work formalises a FAT32-like filesystem and proves read-over-write properties through refinement of a series of models. Further, it proves the correctness of FAT32's allocation and garbage collection mechanisms, and provides artefacts to be used in a subsequent realistic model of FAT32.

10 Future work

Our primary goal is to dispense with the tree representation and implement filesystem traversal by looking up entries in directory files. This will also involve addressing a subtle issue where reads affect the state of a filesystem by means of updating the access time, which has been analysed earlier in the context of microprocessors [15]. This will yield a model which is entirely contained in a disk data structure, without auxiliary data structures such as the tree, and which can further be validated by co-simulation with a FAT32 implementation such as that of Linux.

Next, we hope to re-use some artefacts of verifying FAT32 in order to verify a more complex filesystem, such as ext4. Choosing a filesystem with journalling will allow us to model crash consistency.

Finally, we hope to support "code proofs", by providing a basis for reasoning about filesystem operations in filesystem-specific utilities such as `fsck`, as well as other application programs. This is a large part of the motivation for pursuing binary compatibility.

Acknowledgments. This material is based upon work supported by the National Science Foundation SaTC program under contract number CNS-1525472. Thanks are also due to Warren A. Hunt Jr. and Matthew J. Kaufmann for their guidance.

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. Sigurbjarnarson, H., Bornholt, J., Torlak, E., Wang, X.: Push-button verification of file systems via crash refinement. In: OSDI. Volume 16. (2016) 1–16
13. Lamport, L.: Verification and specification of concurrent programs. In: Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems), Springer (1993) 347–374
14. Microsoft: Microsoft extensible firmware initiative fat32 file system specification (Dec 2000)
15. Goel, S., Hunt, W.A., Kaufmann, M.: Engineering a formal, executable x86 isa simulator for software verification. In: Provably Correct Systems. Springer (2017) 173–209