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

Mihir Parang Mehta* and Warren A. Hunt, Jr.

University of Texas at Austin, Department of Computer Science,
2317 Speedway, Austin, TX 78712, USA
{mihir,
hunt}@cs.utexas.edu
http://www.cs.utexas.edu

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 the proof approach we used and its pros and cons. We describe how this work is applicable to filesystems in general, 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. Indeed, an early application of the ACL2 theorem prover was in this domain, [1], and since then multiple approaches such as [2] have been made to verify filesystem properties including crash consistency through interactive theorem provers. Non-interactive theorem provers have also been used [3], which necessitates abstracting away some of the low-level details of the filesystem; although to some extent this has been a requirement of all the approaches until now.

Here, we detail an effort to advance the state of the art by means of modelling the FAT32 filesystem at the binary level, and validating this model both through theorem proving and through co-simulation with the kernel implementation of FAT32. We begin with an overview of the model and the properties proved with examples; we proceed to a high-level explanation of the proof techniques used; 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.

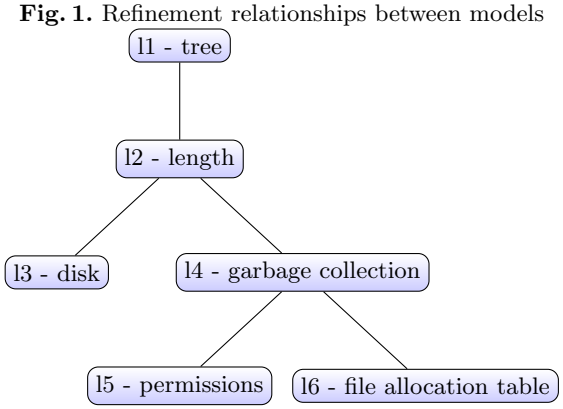
* Please note that the LNCS Editorial assumes that all authors have used the western naming convention, with given names preceding surnames. This determines the structure of the names in the running heads and the author index.

2 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. `l0` 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))))
```

3 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

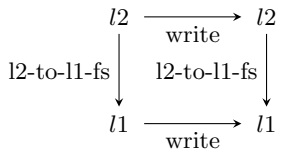
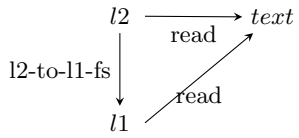


Fig. 3. l2-rdchs-correctness-1



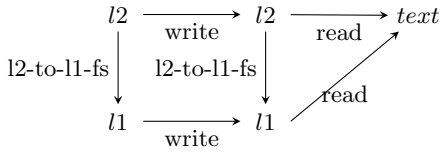
4 Some proof details

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.It is interesting to note that disabling l6-wrchs brings down the certification time for `l6` from 84 seconds to 64 seconds.

```
(defun l4-stricter-fs-p (fs alv)
  (declare (xargs :guard t))
  (and (l4-fs-p fs)
       (boolean-listp alv)
```

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



```
(let ((all-indices (l4-list-all-indices fs)))
     (and (no-duplicatesp all-indices)
          (indices-marked-p all-indices alv)))))
```

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

5 References

References

1. Bevier, W.R., Cohen, R.M.: An executable model of the synergy file system. Technical report, Technical Report 121, Computational Logic, Inc (1996)
2. 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)
3. 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