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

Mihir Parang Mehta
Department of Computer Science
University of Texas at Austin
Austin, TX, USA
mihir@cs.utexas.edu

In this work, we present an approach towards constructing executable specifications of existing filesystems and verifying their functional properties in a theorem proving environment. We detail an application of this approach to the FAT32 filesystem.

We also detail the methodology used to build up this type of executable specification through a series of models which incrementally add features of the target filesystem. This methodology has the benefit of allowing the verification effort to start from simple models which encapsulate features common to many filesystems and which are thus suitable for re-use.

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 the FAT32 filesystem, and a proof of the read-over-write properties for FAT32 system calls. By starting with a high-level abstract model and adding more filesystem features in successive models, we are able to manage the complexity of this proof, which has not, to our knowledge, been previously attempted. Thus, this paper contributes an implementation of several Unix-like system calls for FAT32, formally verified against an abstract specification and tested for binary compatibility by means of co-simulation.

In the rest of this paper, we describe these filesystem models and the properties proved, with examples; we proceed to a high-level explanation of these proofs and the co-simulation infrastructure; and further we offer some insights about the low-level issues encountered while working 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.

Submitted to:
ACL2 workshop 2018

2.1 Interactive theorem provers

An early effort in the filesystem verification domain was by Bevier and Cohen [4], who specified the Synergy filesystem and created an executable model of the same in ACL2 [12], 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 [13] used Isabelle/HOL [16] 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 [2]. Here the authors built a verifying compiler to translate a filesystem specification in their domain-specific language to C-language code, accompanied by a proof of the correctness of this translation. Elsewhere, the SibylFS project [17], 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 [3] has also been used, for instance, for FSCQ [6], 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 [8] have also been used; Hyperkernel [15] is a recent effort which simplifies the xv6 [7] microkernel until the point where Z3 can verify its properties 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 [18], 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 filesystem features such as extents, which are essential to FAT32 and many other filesystems.

3 Program architecture and performance considerations

We have two concrete models for the FAT32 filesystem - `M2`, which is a faithful representation of a FAT32 disk image in the form of a stobj [5], and `M1`, which represents the state of the FAT32 filesystem as a directory tree. This allows us to address the practical details of updating a disk image in `M2`, which benefits from ACL2's efficient stobj array operations provides for stobjs, and abstract them away in `M1` for easier reasoning without the syntactic constraints imposed on stobj arrays.

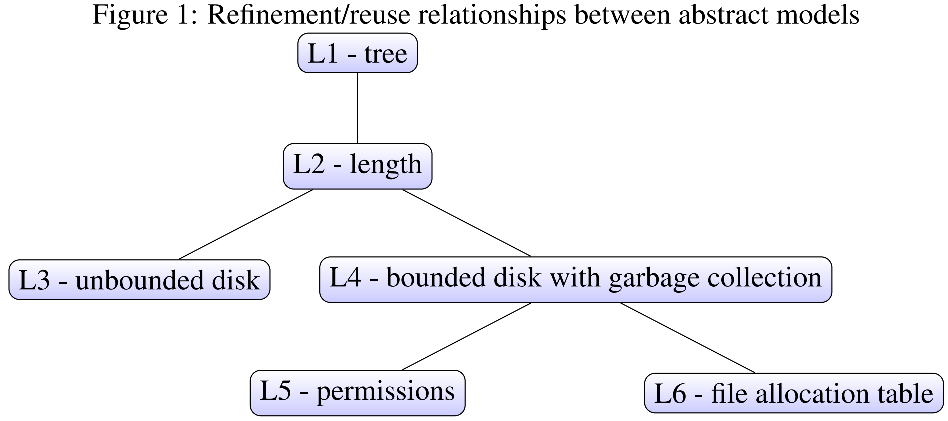
These concrete filesystem models are based upon abstract models `L1` through `L6`. These models are constructed incrementally to allow for reuse of features in general, and specifically to allow read-after-write proofs for simpler models to be reused in more complex models. In the case of models `L4` and `L6`, we are able to show a refinement relationship without stuttering [1]; however, for the other models we are able to reuse proofs without proving a formal refinement relation; these reuse relationships are summarised in figure 1. Much of the code and proof infrastructure is also shared between the abstract models and the concrete models by design. Details of the filesystem features introduced in the abstract models can be seen in table 1.

A design choice that arises in this work pertains to the level of abstraction: how operating-system specific do we want to be in our model? Choosing, for instance, to make our filesystem operations con-

Table 1: Abstract 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.

Figure 1: Refinement/reuse relationships between abstract models



form to the `file_operations` interface provided by the Linux kernel for its filesystem modules would make our work less general, but avert us from having to recreate some of the filesystem infrastructure provided by the kernel. We choose to implement a subset of the POSIX filesystem application programming interface, in order to enable us to easily compare the results of running filesystem operations on `M2` and the Linux kernel's implementation of FAT32, which in turn allows us to test our implementation's correctness through co-simulation in addition to theorem proving. As a trade-off for this choice, we are required to implement process tables and file tables, which we do through a straightforward approach similar to that used in Synergy [4].

4 The FAT32 filesystem

FAT32 was initially developed at Microsoft in order to address the capacity constraints of the DOS filesystem. Microsoft's specification for FAT32 [14], which we follow closely in our work, details the layout of data and metadata in a valid FAT32 disk image.

In FAT32 all files, including regular files and directory files, are divided into *clusters* (sometimes called *extents*) of a fixed size. The size of a cluster, like many other parameters of a FAT32 volume, is stored in a *reserved area* at the beginning of the volume and remains constant after the volume is created. The cluster size must be an integer multiple of the sector size, which in turn must be at least 512 bytes; these and other constraints are detailed in the specification. Directory files are for the most part treated the same way as regular files by the filesystem, but they differ in a metadata attribute, which indicates that the contents of directory files should be treated as sequences of directory entries. Each such directory entry is 32 bytes wide and contains metadata 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 [1] 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 example directory tree in figure 2. Here, `/tmp` is a subdirectory of the root directory (/texttt//). For the purposes of illustration, all regular files and directories in this example are assumed to span one cluster except for `/vmlinuz` which spans two clusters (3 and 4), and *EOC* refers to an "end of clusterchain" value.

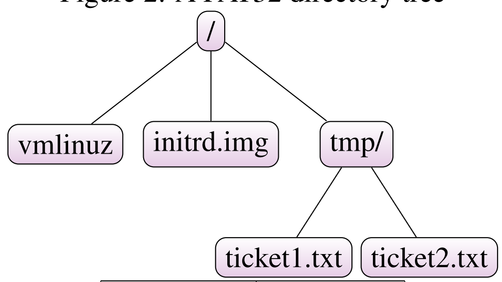
5 Proof methodology

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

[1] There is actually a range of end-of-clusterchain values in the specification, not just one. We support all values in the range.

Figure 2: A FAT32 directory tree



FAT index	FAT entry
0 (unused)	
1 (unused)	
2	*EOC*
3	4
4	*EOC*
5	*EOC*
6	*EOC*
7	*EOC*
8	*EOC*
9	0
⋮	⋮

	Directory entry in /
0	”vmlinuz”, 3
32	”initrd.img”, 5
64	”tmp”, 6
⋮	⋮

	Directory entry in /tmp/
0	”ticket1”, 7
32	”ticket2”, 8
⋮	⋮

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))))
```

In L1, our simplest model, the read-over-write properties are proven from scratch. In each subsequent model, the read-over-write properties are proven as corollaries of equivalence proofs which establish the

Figure 3: l2-wrchs-correctness-1

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

Figure 4: l2-rdchs-correctness-1

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

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`.

6 Some proof details

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

6.1 Invariants

As the abstract 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.

Figure 5: l2-read-over-write-1

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

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)))))
```

Similarly, we find it useful to package up certain invariants for the stobj `fat32-in-memory`, which we maintain while manipulating the stobj through input/output operations and file operations, in the predicate `compliant-fat32-in-memoryp`, for which a listing follows.

```
(defund compliant-fat32-in-memoryp (fat32-in-memory)
  (declare (xargs :stobjs fat32-in-memory :guard t))
  (and (fat32-in-memoryp fat32-in-memory)
       (>= (bpb_bytspersec fat32-in-memory) *ms-min-bytes-per-sector*)
       (>= (bpb_secperclus fat32-in-memory) 1)
       (>= (count-of-clusters fat32-in-memory)
           *ms-fat32-min-count-of-clusters*)
       (>= (bpb_rootclus fat32-in-memory) *ms-first-data-cluster*)))
```

6.2 Reuse

As noted earlier, in our abstract models, 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 Co-simulation

Previous work on executable specifications [9] has shown the importance of testing these on real examples, in order to validate that the behaviour shown matches that of the system being specified. In our case, this means we must validate our filesystem by testing it in execution against a canonical implementation of FAT32; in this case, we choose the implementation which ships with Linux kernel 3.10.

We use `mkfs.fat` [11], a program which produces FAT32 disk images, for our tests. When run with the `-v` flag, this program emits an English-language summary of the fields of the newly created disk image; we make use of this by writing an ACL2 program based on our model which reads the image and reproduces this summary. This validates our code for reading the various fields of the disk image and gives us a regression test to use while we modify our model to support proofs and filesystem calls.

Further, we co-simulate `cat` [10], a simple program which reads its input and copies it to its output. Its functionality is reproduced in an ACL2 program which directly interacts with the `fat32-in-memory` stobj, without using our implementations of system calls. This allows us to validate our code for reading and writing regular files and directories which span multiple clusters.

8 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.

9 Future work

The FAT32 model is still a work in progress; the set of system calls is not yet complete and the translation functions between disk images, `M2` instances and `M1` instances are not yet verified. Once we have these, we intend to use them as a basis for reasoning about sequences of filesystem operations in a program, in a manner akin to proving properties of code on microprocessor models. This is a motivation for the pursuit of binary compatibility in our work.

While FAT32 is interesting of and by itself, it lacks features such as crash consistency, which most modern filesystems provide by means of journalling. We hope to re-use some artefacts of formalising FAT32 in order to verify a filesystem with journalling, such as ext4.

We also hope to model the behaviour of filesystems in a multiprogramming environment, where concurrent filesystem calls must be able to occur without corruption or loss of data.

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.

10 Bibliography

References

[1] Martín Abadi & Leslie Lamport (1991): *The existence of refinement mappings*. Theoretical Computer Science 82(2), pp. 253–284.

[2] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell et al. (2016): *Cogent: Verifying high-assurance file system implementations*. In: *ACM SIGPLAN Notices*, 51, ACM, pp. 175–188.

[3] Yves Bertot & Pierre Castéran (2013): *Interactive theorem proving and program development: CoqArt: the calculus of inductive constructions*. Springer Science & Business Media.

[4] William R Bevier & Richard M Cohen (1996): *An executable model of the Synergy file system*. Technical Report, Technical Report 121, Computational Logic, Inc.

[5] Robert S Boyer & J Strother Moore (2002): *Single-threaded objects in ACL2*. In: *International Symposium on Practical Aspects of Declarative Languages*, Springer, pp. 9–27.

[6] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek & Nickolai Zeldovich (2016): *Using Crash Hoare Logic for Certifying the FSCQ File System*. In Ajay Gulati & Hakim Weatherspoon, editors: *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.*, USENIX Association, doi:10.1145/2815400.2815402. Available at `https://www.usenix.org/conference/atc16`.

[7] Russ Cox, M Frans Kaashoek & Robert T Morris: *Xv6, a simple Unix-like teaching operating system, 2016. URL http://pdos. csail. mit. edu/6.828/2014/xv6. html.*

[8] Leonardo De Moura & Nikolaj Bjørner (2008): *Z3: An efficient SMT solver*. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 337–340.

[9] Shilpi Goel, Warren A Hunt, Matt Kaufmann & Soumava Ghosh (2014): *Simulation and formal verification of x86 machine-code programs that make system calls*. In: *Formal Methods in Computer-Aided Design (FMCAD), 2014*, IEEE, pp. 91–98.

[10] Torbjorn Granlund & Richard M. Stallman: *cat (1)-Linux manual page, accessed: 09 Jul 2018*.

[11] Dave Hudson, Peter Anvin & Roman Hodek: *mkfs.fat (8)-Linux manual page, accessed: 09 Jul 2018*.

[12] Matt Kaufmann, Panagiotis Manolios & J. Strother Moore (2000): *Computer-aided reasoning: an approach*. Kluwer Academic Publishers.

[13] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish et al. (2009): *seL4: Formal verification of an OS kernel*. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ACM, pp. 207–220.

[14] Microsoft (2000): *Microsoft Extensible Firmware Initiative FAT32 File System Specification*. Available at `www.microsoft.com/hwdev/download/hardware/fatgen103.pdf`.

[15] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak & Xi Wang (2017): *Hyperkernel: Push-Button Verification of an OS Kernel*. In: *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, ACM, New York, NY, USA, pp. 252–269, doi:10.1145/3132747.3132748. Available at `http://doi.acm.org/10.1145/3132747.3132748`.

[16] Tobias Nipkow, Lawrence C Paulson & Markus Wenzel (2002): *Isabelle/HOL: a proof assistant for higher-order logic*. 2283, Springer Science & Business Media.

[17] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy & Peter Sewell (2015): *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, pp. 38–53.

[18] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak & Xi Wang (2016): *Push-Button Verification of File Systems via Crash Refinement.* In: *OSDI*, 16, pp. 1–16.