Formalising Filesystems in the ACL2 Theorem Prover

An Application To FAT32

Mihir Mehta

Department of Computer Science
University of Texas at Austin

mihir@cs.utexas.edu

05 November, 2018

Why filesystem verification matters

- Basis of current computing paradigm.
- Provide a means to address data by names, not numbers.
- Provide efficiency and redundancy.
- Thus, critically important to verify the properties of filesystems in common use, making them more reliable.
  - FAT32 - once widely used on Windows, and still used by a large number of embedded systems - qualifies.

The plan

- Modelling FAT32 in ACL2
- Verification through refinement
- Binary compatibility and execution efficiency
- Co-simulation testing for accuracy

Outline

FAT32

The models

Proofs and co-simulation

Related and future work

Outline

Our FAT32 model aims to have . . .

- . . . the same space constraints as a FAT32 volume of the same size.
- . . . the same success and failure conditions for file operations, and the same error codes for the latter.
- . . . a way to read a FAT32 disk image from a block device, and a way to write it back.
  - This is made easier by choosing to replicate the on-disk data structures of FAT32 in the model.

File operations in our model

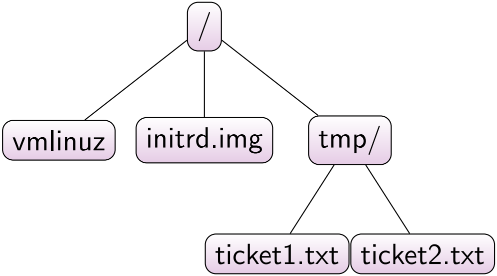
- File operations categorised into *read operations*, which do not change the state of the filesystem, and *write operations* which do.
- Generic signature for read operations:
  (read fs-inst *args*) $\mapsto$ (mv ret-val status errno)
- Generic signature for write operations:
  (write fs-inst *args*) $\mapsto$ (mv fs-inst ret-val status errno)
- Ret-val, status and errno derived from Linux syscall conventions - in the absence of pointers and global variables, they must all be returned

The FAT32 specification

In a FAT32 volume, the unit of data storage is a *cluster* (also known as an *extent*). There are three on-disk data structures.

- *reserved area*, volume-level metadata such as the size of a cluster and the number of clusters.
- *file allocation table*, collection of *clusterchains* (linked lists of clusters), one for each regular file/directory file.
- *data region*, collection of clusters.

A FAT32 Directory Tree



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

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

eoc: end of clusterchain

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

Outline

FAT32

The models

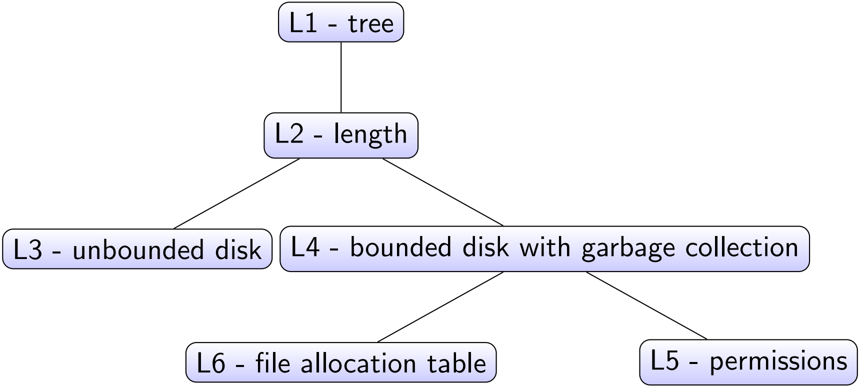
Proofs and co-simulation

Related and future work

Abstract models

- Bootstrap - begin with abstract filesystem models, in order to explore the properties we require in a FAT32 model.
- Incrementally add the desired properties in a series of models
- Wherever possible, capture common features expected in different filesystems.

Relationships between abstract models



Beginning to model FAT32

Next, in models `M1` and `M2`, we model FAT32 more concretely, providing the standard POSIX system calls in accordance with Microsoft's official specification.

- `M1` - another tree model, with nodes storing FAT32's file-level metadata.
- `M2` - a stobj model, with fields for all the metadata in the reserved area and arrays for the file allocation table and data region.

This way, we benefit from efficient stobj array operations in `M2`, and we can simplify our reasoning in `M1` by continuing to work with directory trees.

Outline

Properties proved

- *Read-over-write* properties show that write operations have their effects made available immediately for reads at the same location, and also that they do not affect reads at other locations.
- We've proved these properties for the abstract models `L1`-`L6`, and we've also proved them for our concrete models `M1` and `M2`, with the caveat that the transformations between `M1` and `M2` are not yet verified.
- Also, for models `L4`-`L6` it is proved that write operations succeed if and only if there are sufficient free blocks to satisfy the request - although this exact result does not carry over to FAT32 (deleted file directory entries can take up space).

Refinement proofs

- For the abstract models, we started by proving the read-over-write properties *ab initio* for `L1`.
- For each subsequent model in `L2`-`L6`, we have proved a refinement relationship where possible, or an equivalence where a strict refinement does not hold, with a previous model and used it to prove read-over-write properties as a corollary.
- An illustration of such a proof follows.

Proof example: first read-over-write in L2

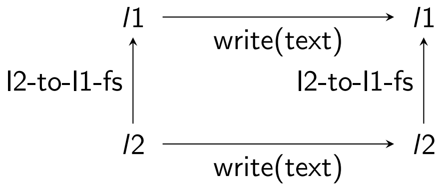


Figure: l2-wrchs-correctness-1 (write is overloaded for L2 and L1)

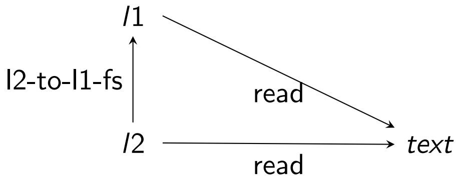


Figure: l2-rdchs-correctness-1 (read is overloaded for L2 and L1)

Proof example: first read-over-write in L2

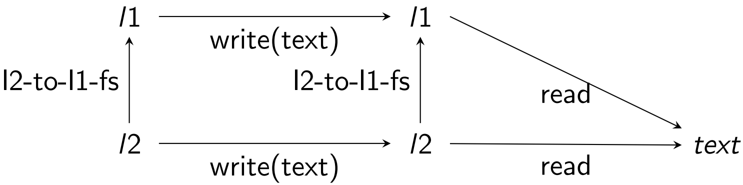


Figure: l2-read-over-write-1

Co-simulation

- Ensure that our implementation lines up with FAT32, the target filesystem.
- Support POSIX system calls - `lstat`, `open`, `pread`, `pwrite`, `close`, `mkdir` and `mknod`.
- Wherever `errno` is to be set, do what Linux does.
- Check that the output of our ACL2 programs (based on the FAT32 model) matches the utilities (including `cp` and `mkfs`) which they replicate.

Outline

Related work - interactive theorem provers

- Synergy FS (1996) - executable model with processes and file descriptors, but no read-over-write theorems (ACL2).
- COGENT (2016) - verifying compiler from a DSL to C code for a filesystem (Isabelle/HOL).
- FSCQ (2016) - high-performance filesystem with verified crash consistency properties (Coq).

Note: Our work, in contrast to the above, models an actual filesystem.

Related work - non-interactive theorem provers

- Hyperkernel (2017) - microkernel with system calls simplified until the point where useful properties can be proved through SMT solving (Z3).
- Yggdrasil (2016) - filesystem verification through SMT solving (Z3).

Future work

- Model the remaining POSIX system calls for FAT32 and use them to reason about sequences of file operations (i.e. do code proofs).
- Reuse FAT32 verification artifacts for a filesystem with crash consistency, for instance, ext4.
- Model concurrent file operations in a multiprogramming environment.

Recent progress

- Set of supported POSIX system calls expanded.
- Set of co-simulation tests, mostly based on coreutils and mtools programs, expanded based on these.
- Functions for converting `M2` instances to FAT32 disk images and back proved to be inverses of each other.
- Equivalence relation developed to allow two FAT32 disk images to be compared modulo rearrangement of data and reordering of files within directories.
  - This gives us a means to co-simulate programs which modify filesystem state, such as `rm`.

Conclusion

- FAT32 formalised, demonstrating the applicability of the refinement style to filesystem verification.
- Co-simulation infrastructure developed to validate filesystem models against a canonical implementation, such as that of Linux.
- Allocation and garbage collection algorithms certified.