# Verifying Filesystem Data Structure Properties

## Using a FAT32-like filesystem organisation

Mihir Mehta

Department of Computer Science
University of Texas at Austin

mihir@cs.utexas.edu

06 April, 2018

# Outline

# Outline

# Why we need a verified filesystem

- Ubiquity of filesystems, even as operating systems move towards making them invisible
- Increasing complexity of modern filesystems and the tools which analyse and recover data
- Inadequacy of POSIX, especially as a basis for a formal verification effort
- Opportunity to formally verify guarantees claimed by these filesystems and tools

# Why FAT32?

- Officially supported by Windows in the past and still used in USB thumb drives and the like
- Relatively simple, without journalling or transactions
- Supports, for example, nested subdirectories and long filenames
- Tractable from verification standpoint, and yet capable of providing a basis for verification of more complex filesystems

# Verification task

A formal model of FAT32 must have

- A file allocation table - this serves as a linked list for contents of regular files and directories ✓
- Clusters (a.k.a. extents) - groups of adjacent sectors, read and written all at once
- Metadata for regular files and directories
- Error codes, to signify insufficient space and the like ✓

# Outline

# Verifying through refinement

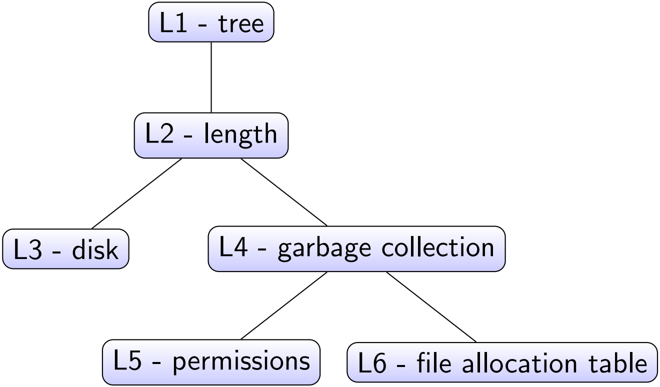
- Intuition - start simply, instead of modelling all filesystem features at once
- Justification - reasoning about input/output behaviour of a complex system is hard, but an equivalent approach is to reason about the input/output behaviour of a simple system, and prove the complex system *implements* (Abadi, 1991) the simple system
- Definition - For a pair of transition systems $S_1$ and $S_2$, $S_1$ is said to implement $S_2$ if every externally visible behaviour allowed by $S_1$ is also allowed by $S_2$.
- One way of proving this implementation relation - finding a *refinement mapping*, which maps each (state, transition) pair of $S_1$ to a legal (state, transition) pair of $S_2$.

# Models and their features

The filesystem is modelled iteratively, incrementally adding features of FAT32.

1. Filesystem represented as a tree - leaf nodes for regular files and non-leaf nodes for directories; regular file contents represented as ACL2 strings; unbounded storage.
2. *Length* added as metadata for each regular file.
3. Regular file contents divided into blocks of fixed size, which are stored in an external "disk" data structure of unbounded size.
4. Disk size bounded; allocation vector data structure (*à la* CP/M) introduced to help allocate and garbage collect blocks.
5. Metadata for *file ownership* and *access permissions* added for regular files.
6. Allocation vector replaced by file allocation table.

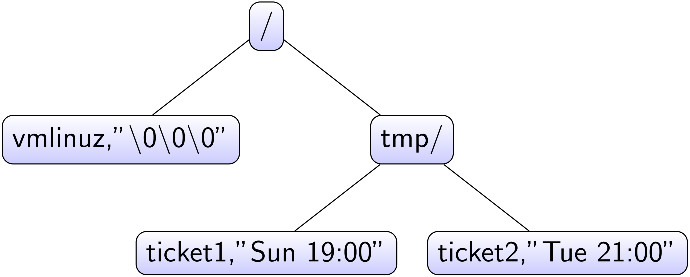
# Models and their refinement relationships



# Modelling a filesystem

- L1 filesystem representation: literal directory tree, in which non-leaf nodes represent (sub)directories and leaf nodes represent regular files
- L6 filesystem representation:
  - a tree, as above
  - a disk, containing the textual contents of regular files broken into fixed-size blocks;
  - and a file allocation table, mapping each block in a regular file to the next, this allowing us to read the contents of the entire file.

# Model 1 example



# Model 6 example

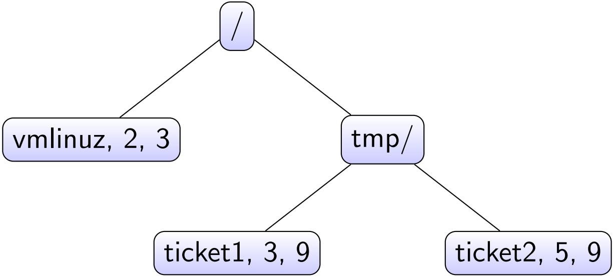



Table: Disk and allocation vector

| | | |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | \0\0\0 | EOC |
| 3 | Sun 19:0 | 4 |
| 4 | 0 | EOC |
| 5 | Tue 21:0 | 6 |
| 6 | 0 | EOC |

# Conceptualising proofs

- Initial focus on read-over-write properties (more details follow)
- Transition system formulation - filesystem instances (storing some files and directories with some metadata) become states, and file operations (reading, writing) become externally visible actions
- Small number of file operations, consistently named across models - *stat, read, create, write, unlink*
- Refinement mappings - simply find functions that map each instance of a given model to an equivalent instance of a previously verified model
- Proof burden for `L1` (base model) - satisfaction of read-over-write properties
- Proof burden for `L2` (and following models) - mapping from `L2` instances to `L1` composes correctly with file operations in both `L2` and `L1`.

# Outline

# Verifying the models

- We've focussed so far on two filesystem properties, known in the literature as the *read-over-write* properties.
  1. After a write of some text at some location, a read of the same length at the same location should yield the text.
  2. After a write, a read at a different location should yield the same results as a read before the write.
- These properties have been proven for all models so far, including the present model which features a file allocation table.

# Proof example: first read-over-write in L2

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

Figure: l2-wrchs-correctness-1

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

Figure: l2-rdchs-correctness-1

# Proof example: first read-over-write in L2

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

Figure: l2-read-over-write-1

# Proof challenges

- Many lemmas proved about `assoc`, `delete-assoc` and `no-duplicatesp` but invariants are really the core of the proof
- How do we define a "good state" of a filesystem, which shows that reading, writing and other operations can be safely carried out?
- Answering this question involves a trade-off between simplicity (to help with verification) and generality (to model as many real-world situations as possible.)
- We choose to require:
  - that each block on the disk is attributed to at most one regular file;
  - that the clusters attributed to each non-empty regular file end with a legal EOF value, as defined by the FAT specification.
  - that each regular file is annotated with "length", a metadata field that corresponds to the actual length of the file as determined by traversing the file allocation table and reading the corresponding blocks.

# Outline

# Future work

1. Complete the FAT32 model, by means of
   - supporting variable cluster sizes,
   - moving the file allocation table onto the disk, and
   - moving all file and directory metadata from the tree to the disk.
2. Validate the model through co-simulation with an implementation.
3. Model a more complex filesystem, for instance NTFS, by re-using algorithms and proofs from the models built so far.

## Related work

- FSCQ (Chen, 2016) - novel filesystem, proven safe against crashes using Coq, performs comparably to ext4.
- COGENT (Amani, 2016) - "verifying compiler" translates specs in a DSL to C implementations free of some classes of bugs.
- SibylFS (Ridge, 2015) - "executable specification" for filesystem validates or rejects filesystem traces across multiple OSes.
- Hyperkernel (Nelson, 2017) - xv6 microkernel implemented with system calls changed to make them constant-time; in return, verification burden becomes lightweight enough for Z3 SMT solver.
- Our work's distinct aim: model an existing filesystem (FAT32) faithfully and match the resulting disk image byte-to-byte.

# Conclusion

- FAT32-adjacent filesystem formalised with a binary compatible file allocation table.
- Read-over-write properties proven by means of refinement through a series of models.