Verifying filesystems in ACL2

Towards verifying file recovery tools

Mihir Mehta

Department of Computer Science
University of Texas at Austin

`mihir@cs.utexas.edu`

27 October, 2017

Outline

Motivation and related work

Our approach

Progress so far

Future work

Why we need a verified filesystem

- Filesystems are everywhere, even as operating systems move towards making them invisible.
- In the absence of a clear specification of filesystems, users are underserved.
- Modern filesystems have become increasingly complex, and so have the tools to analyse and recover data from them.
- It would be worthwhile to specify and formally verify, in the ACL2 theorem prover, the guarantees claimed by filesystems and tools.

Related work

- In Haogang Chen's 2016 dissertation, the author uses Coq to build a filesystem (named FSCQ) which is proven safe against crashes in a new logical framework named Crash Hoare Logic.
- His implementation was exported into Haskell, and showed comparable performance to ext4 when run on FUSE.
- Hyperkernel (Nelson et al, SOSP '17) is a "push-button" verification effort, but approximates by changing POSIX system calls for ease of verification.
- In our work, we instead aim to model an existing filesystem faithfully and match the resulting disk image byte-to-byte.

Outline

Motivation and related work

Our approach

Progress so far

Future work

Choosing an initial model

- Our goal here is to verify the FAT32 filesystem, but we need a simpler model to begin with.
- Our filesystem's operations should suffice for running a workload.
- Yet, parsimony and avoidance of redundancy are essential for theorem proving.
- What's a necessary and sufficient set of operations?

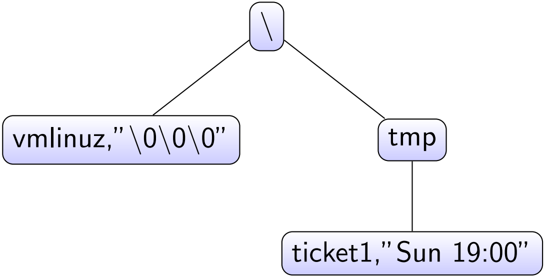
Minimal set of operations?

- The Google filesystem suggests a minimal set of operations:
  - `create`
  - `delete`
  - `open`
  - `close`
  - `read`
  - `write`
- Of these, `open` and `close` require the maintenance of file descriptor state - so they can wait.
- However, they are essential when describing concurrency and multiprogramming behaviour.
- Thus, we can start modelling a filesystem, and several refinements thereof.

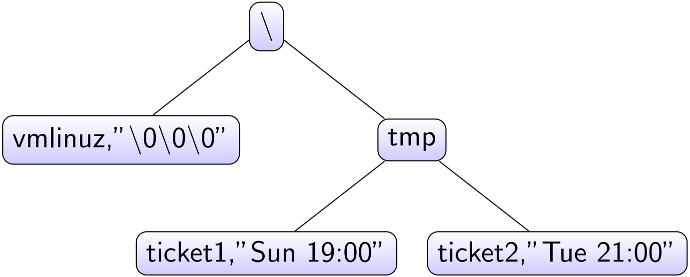
Quick overview of models

- Model 1: Tree representation of directory structure with unbounded file size and unbounded filesystem size.
- Model 2: Model 1 with file length as metadata.
- Model 3: Tree representation of directory structure with file contents stored in a "disk".
- Model 4: Model 3 with bounded filesystem size and garbage collection.

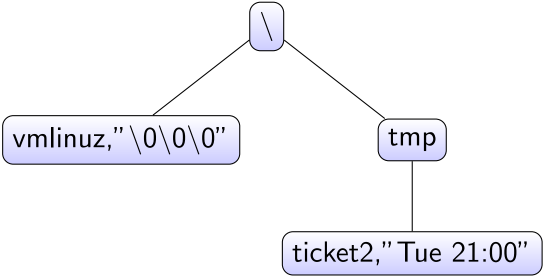
Model 1



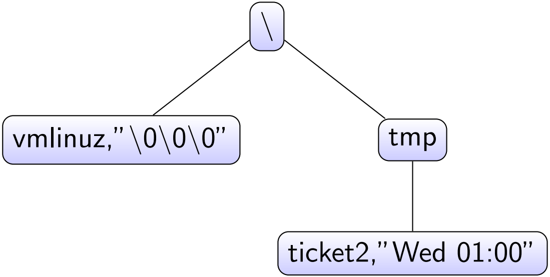
Model 1



Model 1



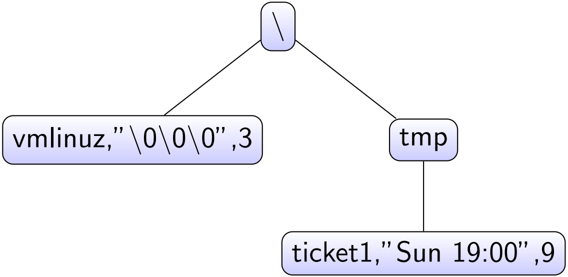
Model 1



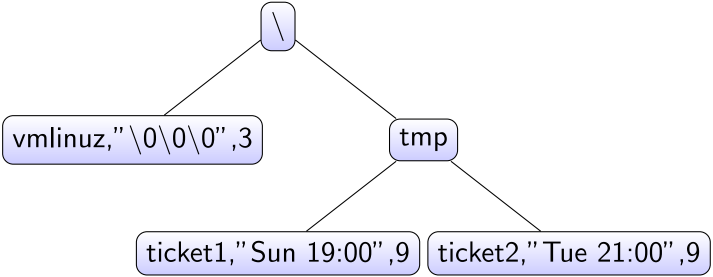
Model 2

- Model 1 supports nested directory structures, unbounded file size and unbounded filesystem size.
- However, there's no metadata, either to provide additional information or to validate the contents of the file.
- With an extra field for length, we can create a simple version of fsck that checks file contents for consistency.
- Further, we can verify that create, write, delete etc preserve this notion of consistency.

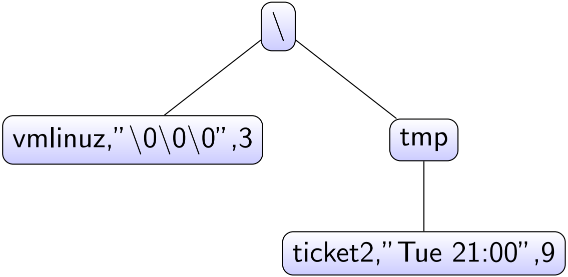
Model 2



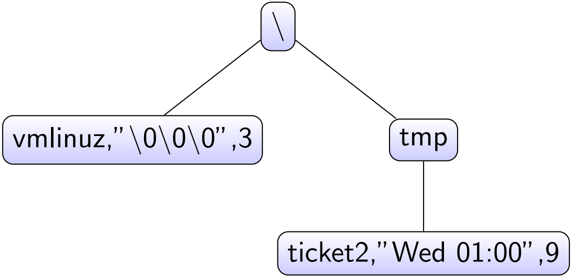
Model 2



Model 2



Model 2



Model 3

- As the next step, we focus on externalising the storage of file contents.
- We also choose to break up file contents into "blocks" of a finite length.
  - Note: this would mean storing file length is no longer optional.

Model 3

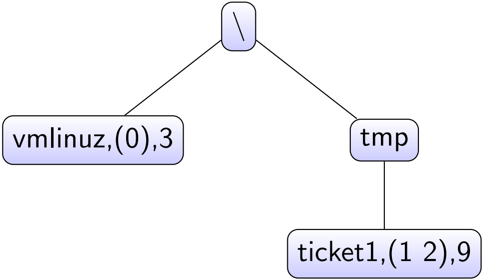


Table: Disk

\0\0\0
Sun 19:0
0

Model 3

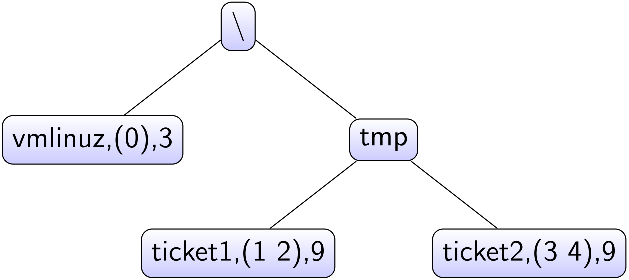


Table: Disk

\0\0\0
Sun 19:0
0
Tue 21:0
0

Model 3

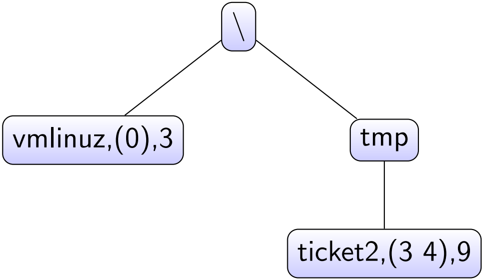


Table: Disk

\0\0\0
Sun 19:0
0
Tue 21:0
0

Model 3

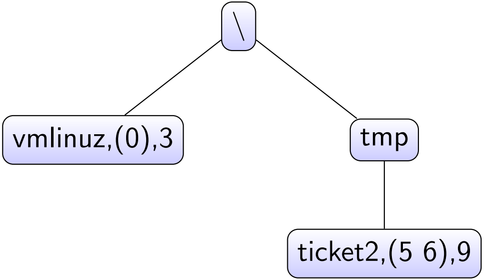


Table: Disk

\0\0\0
Sun 19:0
0
Tue 21:0
0
Wed 01:0
0

Model 4

- In the fourth model, we attempt to implement garbage collection in the form of an allocation vector.
- The allocation vector tracks whether blocks in the filesystem are in use by a file. This allows us to reuse unused blocks.

Model 4

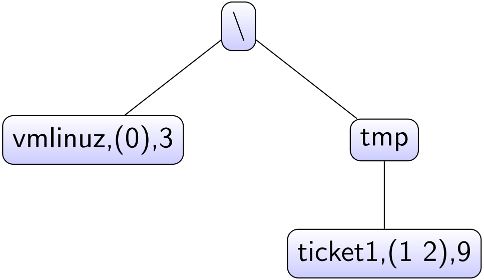


Table: Disk

\0\0\0
Sun 19:0
0

Model 4

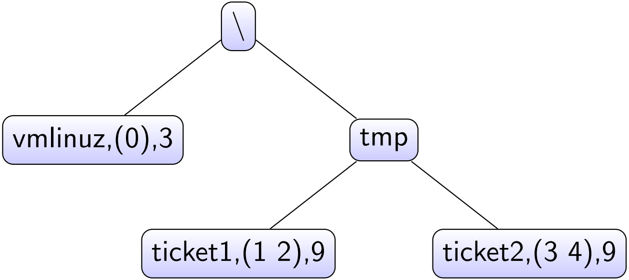


Table: Disk

\0\0\0
Sun 19:0
0
Tue 21:0
0

Model 4

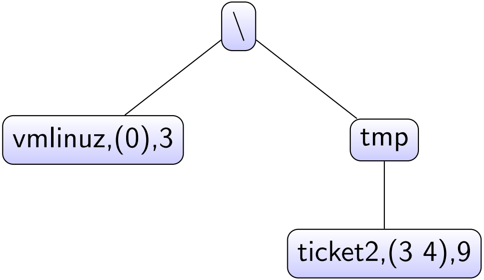


Table: Disk

\0\0\0
Sun 19:0
0
Tue 21:0
0

Model 4

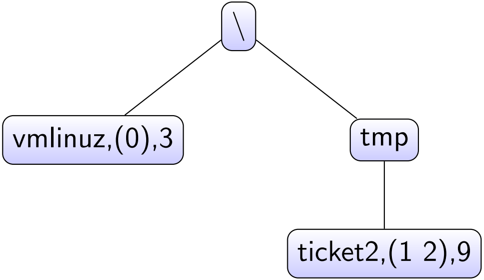


Table: Disk

\0\0\0
Wed 01:0
0
Tue 21:0
0

Outline

Motivation and related work

Our approach

Progress so far

Future work

Proof approaches and techniques

- There are many properties that could be considered for correctness, but we choose to focus on the read-over-write theorems from the first-order theory of arrays.
  1. Reading from a location after writing to the same location should yield the data that was written. Formally, assuming `n = length(text)` and suitable "type" hypotheses (omitted here):
     `l1-rdchs(hns, l1-wrchs(hns, fs, start, text), start, n) = text`
  2. Reading from a location after writing to a different location should yield the same result as reading before writing. Formally, assuming `hns1 != hns2` and suitable "type" hypotheses (omitted here):
     `l1-rdchs(hns1, l1-wrchs(hns2, fs, start2, text2), start1, n1) = l1-rdchs(hns1, fs, start1, n1)`

Proof approaches and techniques

1. For each of the models 1, 2, 3 and 4, we have proofs of correctness of the two read-after-write properties, making use of the proofs of equivalence between models and their successors.
2. Model 4 presented some unique challenges - proving the read-after-write properties required proving an equivalence between model 4 and model 2, rather than model 3.

Proof approaches and techniques

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

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

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

Outline

Motivation and related work

Our approach

Progress so far

Future work

Permissions

- What does permission checking look like in ACL2?
- Top-down: picture the theorems that would prove correctness.

    1. Read/write/execute permission is granted when the requesting user has permission for themselves/their group, or when the permission is granted to all.
    2. Converse: read/write/execute permission is denied when none of the above hold.
    3. Reads that fail because of permissions do not return a value.
    4. Writes that fail because of permissions return an unmodified filesystem.
- How can we represent users and groups?
    1. Users are natural numbers.
    2. Groups are also natural numbers, and a vector holds the group associated with each user.

Other future work

- Linearise the tree, leaving only the disk.
- Add the system call open and close with the introduction of file descriptors.
  This would be a step towards the study of concurrent FS operations.
- Eventually emulate the FAT32 filesystem as a convincing proof of concept, and move on to fsck and file recovery tools.