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

Overview

1. Why we need a verified filesystem
2. Our approach
3. Progress so far
4. Future work

Why we need a verified filesystem

- Filesystems are everywhere.
- Yet they're poorly understood - especially by people who should.
- Modern filesystems have become increasingly complex, and so have the tools to analyse and recover data from them.
- It would be worthwhile to verify that the filesystems and the tools actually provide the guarantees they claim to provide.

What we need

- Our filesystem's operations should suffice for running a workload.
- Yet, parsimony and avoidance of redundancy are essential for theorem proving.
- How about emulating the VFS interface?
- That would mean 19 inode operations, 6 dentry operations and 22 file operations.

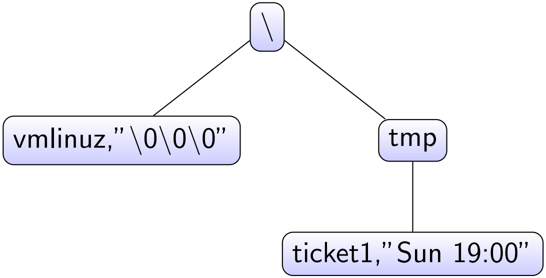
Minimal set of operations?

- There might be a better way.
- 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.
- However, they are essential when describing concurrency and multiprogramming behaviour.
- Thus, we can start modelling a minimal set of filesystem operations.

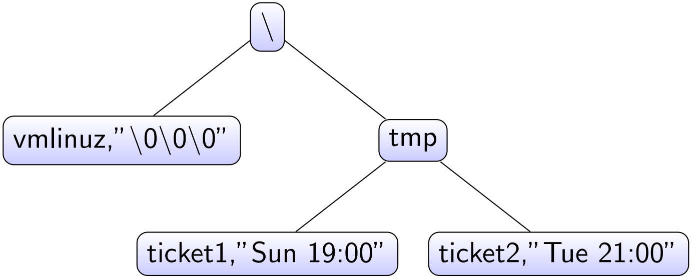
Modelling a filesystem

- What should the filesystem look like?
- How about... a tree?
- Thinking along the lines of recursive datatypes, an `alist` containing only strings or similar `alists` in its `strip-cdrs` could do the job.
- The `strip-cars` would contain the file/directory names.
- Next, we'll look at a running example, and add/delete files from such a model.

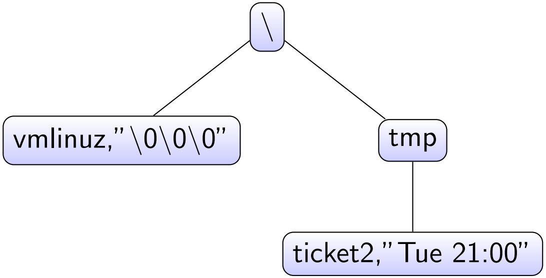
Model 1



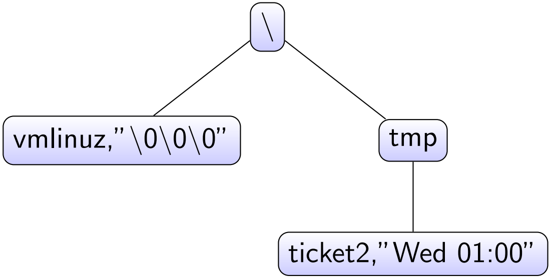
Model 1



Model 1



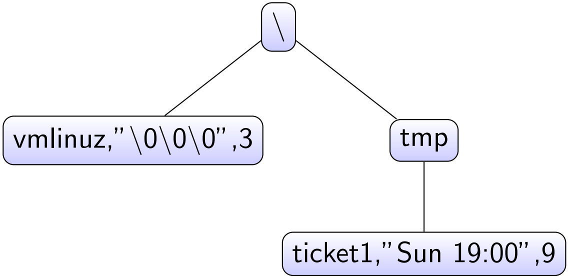
Model 1



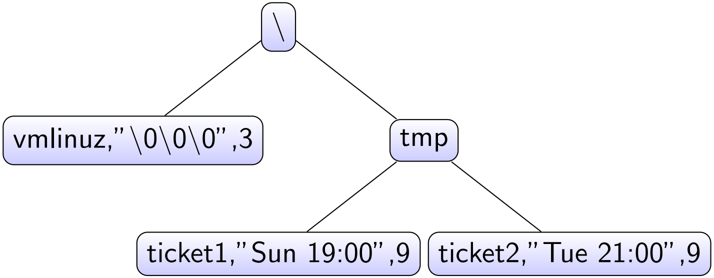
Model 2

- 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.

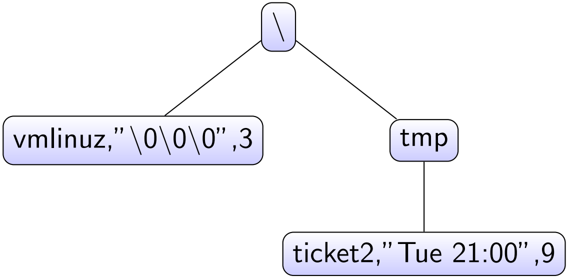
Model 2



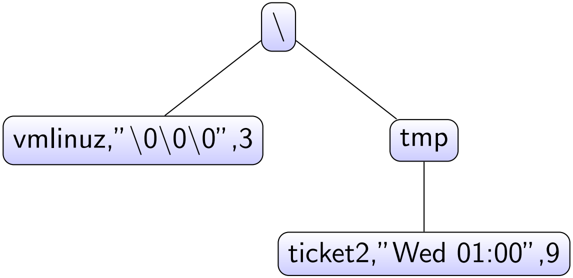
Model 2



Model 2



Model 2



Model 3

- As the next step, we would like to begin externalising the storage of file contents.
- It would also be good to break up file contents into "blocks" of a finite length.
  - Note: this would mean storing file length is no longer optional.

Model 3

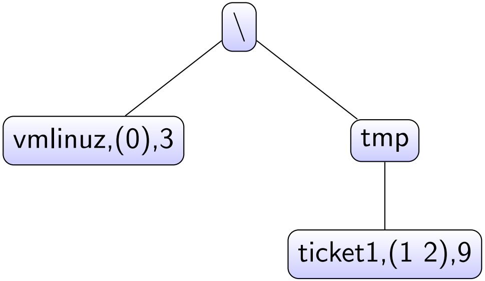


Table: Disk

\0\0\0
Sun 19:0
0

Model 3

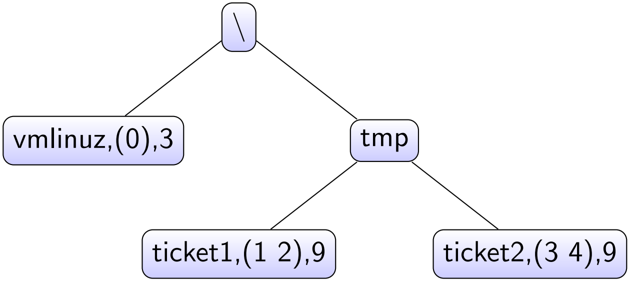


Table: Disk

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

Model 3

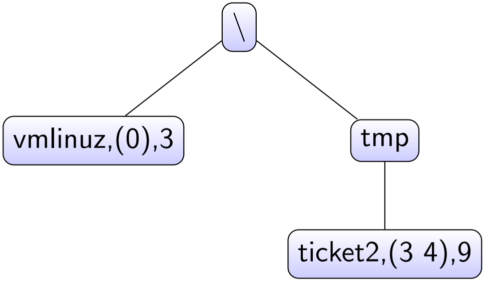


Table: Disk

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

Model 3

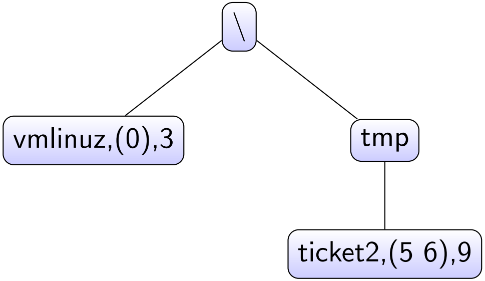


Table: Disk

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

Proof approaches and techniques

- In the fourth model, we implement garbage collection in the form of an allocation vector.
- What guarantees do we need to show that a filesystem of this kind is consistent?

Model 4

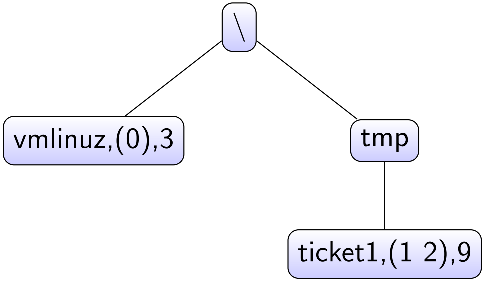


Table: Disk

\0\0\0
Sun 19:0
0

Model 4

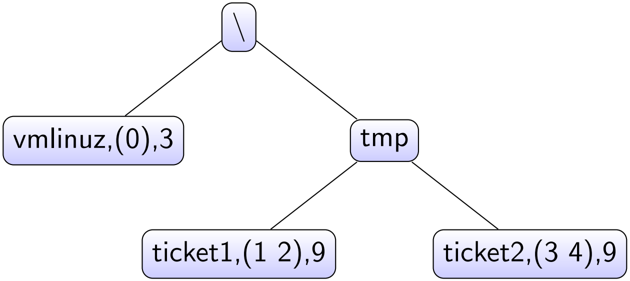


Table: Disk

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

Model 4

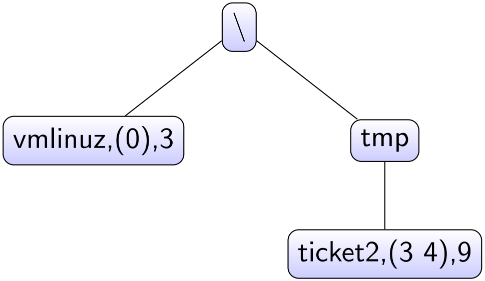


Table: Disk

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

Model 4

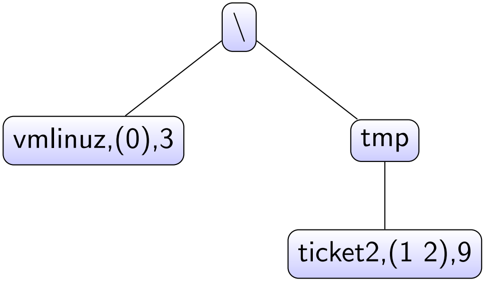


Table: Disk

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

Proof approaches and techniques

- There are many properties that could be considered for correctness, but the read-over-write theorems from the first-order theory of arrays seem like a good place to start.
  1. Reading from a location after writing to the same location should yield the data that was written.
  2. Reading from a location after writing to a different location should yield the same result as reading before writing.
- For each of the models 1, 2 and 3, we have proofs of correctness of the two read-after-write properties, based on the proofs of equivalence between each model and its successor.

Proof approaches and techniques

1. For model 4, the disk and the allocation vector must be in harmony initially and updated in lockstep.
2. Every block referred to in the filesystem must be marked "used" in the allocation vector.
   (The complementary problem - making sure unused blocks are unmarked - is more complicated because it's non-local.)
3. If n blocks are available in the allocation vector, the allocation algorithm must provide n blocks when requested.
4. No matter how many blocks are returned by the allocation algorithm, they must be unique and disjoint with the blocks allocated to other files.

Work in progress: 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 return nil in our model.
  4. Writes that fail because of permissions return an unmodified filesystem.
- Gee, how do we represent users and groups?
  1. Users are natural numbers.
  2. Groups are also natural numbers, and a vector (psst: a nat-list) holds the group associated with each user.

Future work

- Finish finitising the length of the disk and garbage collecting disk blocks that are left unused after a write or a delete operation.
- Possibly, 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.
- Linearise the tree, leaving only the disk.
- Eventually emulate the CP/M filesystem as a convincing proof of concept, and move on to fsck and file recovery 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.
- His implementation was exported into Haskell, and showed comparable performance to ext4 when run on FUSE.
- Our work is different - we're building verified models of actual filesystems with binary compatibility as the aim.
- Hyperkernel (Nelson et al, SOSP '17) is a "push-button" verification effort, but approximates by changing POSIX system calls for ease of verification.