Verifying filesystems in ACL2

Towards verifying file recovery tools

Mihir Mehta

Department of Computer Science
University of Texas at Austin

mihir@cs.utexas.edu

14 November, 2017

Outline

Why we need a verified filesystem

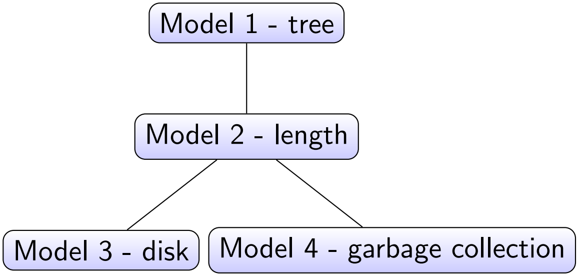
- Filesystems are everywhere, even as operating systems move towards making them invisible.
- In the absence of a clear specification of filesystems, users (and sysadmins in particular) are underserved.
- Modern filesystems have become increasingly complex, and so have the tools to analyse and recover 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 (exported) Haskell implementation performs comparably to ext4.
- 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 (FAT32) faithfully and match the resulting disk image byte-to-byte.

Outline

File system models



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 helpful for theorem proving.
- What's a necessary and sufficient set of operations?

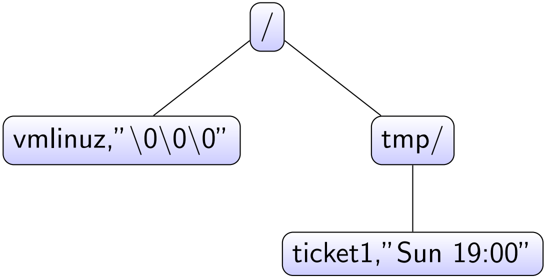
Minimal set of operations?

- 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.

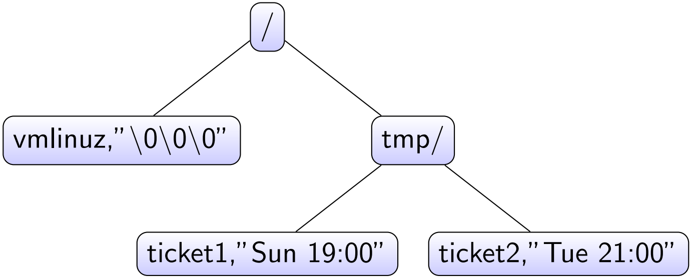
Quick overview of models

- What follows is a sequence of **increasingly refined** 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" (unbounded in length).
- Model 4: Model 3 with bounded filesystem size and garbage collection.
- In our running example, we'll observe file creation, deletion, and overwriting in each of these models in turn.

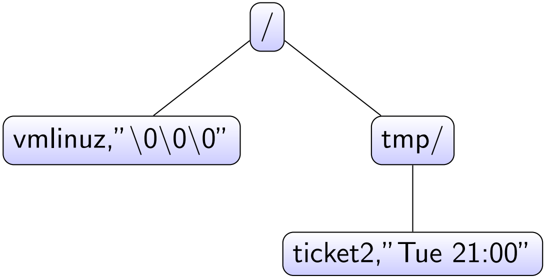
Model 1



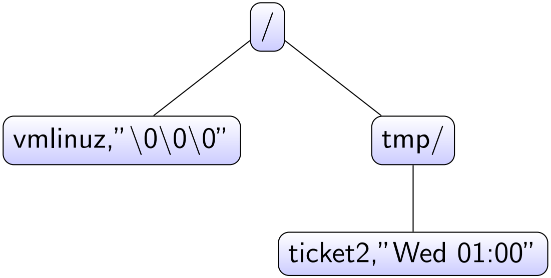
Model 1



Model 1



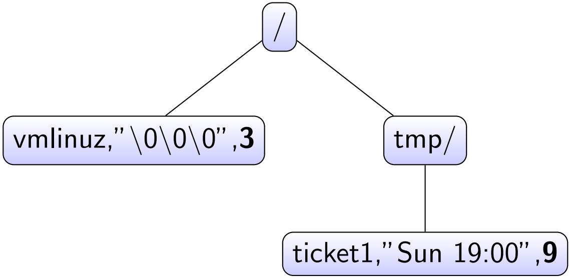
Model 1



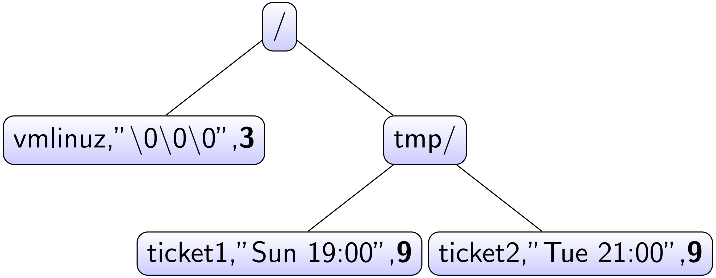
Model 2

- 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.

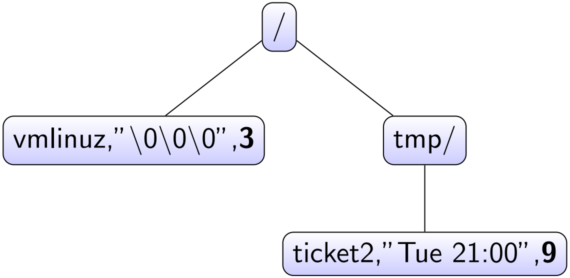
Model 2



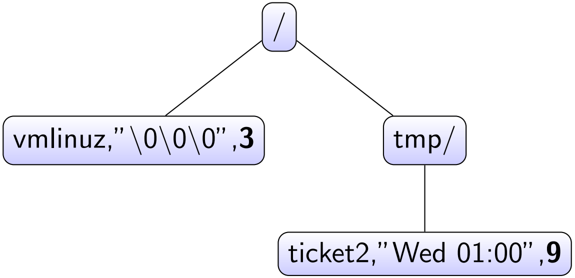
Model 2



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 constant length (8).
  - Note: this would mean storing file length is no longer optional, to avoid reading garbage past end of file at the end of a block.

Model 3

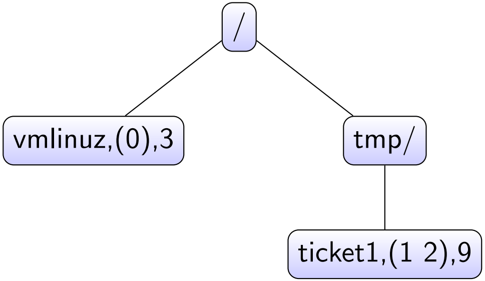


Table: Disk

0	\0\0\0
1	Sun 19:0
2	0

Model 3

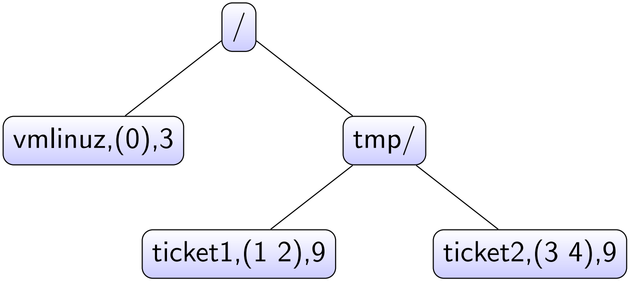


Table: Disk

0	\0\0\0
1	Sun 19:0
2	0
3	Tue 21:0
4	0

Model 3

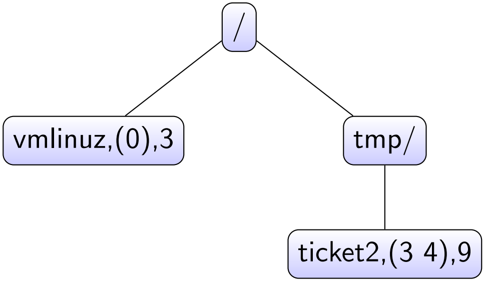


Table: Disk

0	\0\0\0
1	Sun 19:0
2	0
3	Tue 21:0
4	0

Model 3

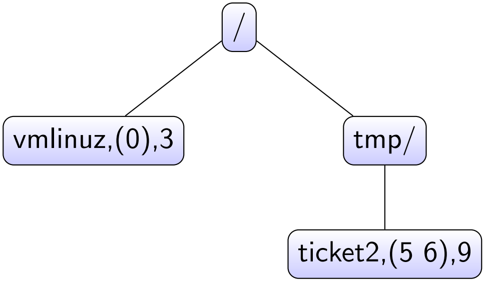


Table: Disk

0	\0\0\0
1	Sun 19:0
2	0
3	Tue 21:0
4	0
5	Wed 01:0
6	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

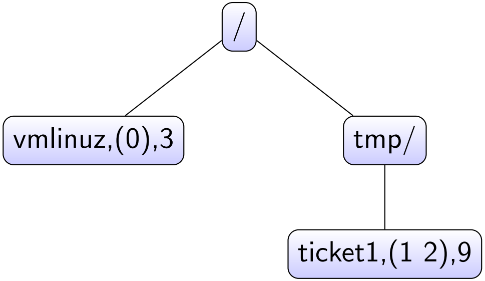


Table: Disk and allocation vector

0	\0\0\0	true
1	Sun 19:0	true
2	0	true
3		false
4		false
5		false

Model 4

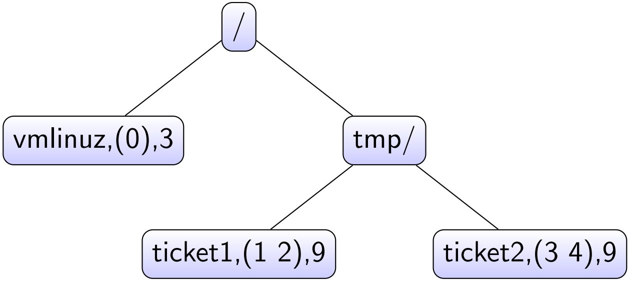


Table: Disk and allocation vector

0	\0\0\0	true
1	Sun 19:0	true
2	0	true
3	Tue 21:0	true
4	0	true
5		false

Model 4

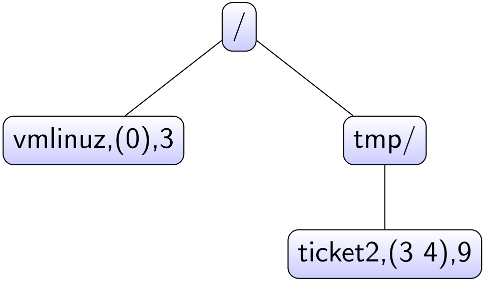


Table: Disk and allocation vector

0	\0\0\0	true
1	Sun 19:0	false
2	0	false
3	Tue 21:0	true
4	0	true
5		false

Model 4

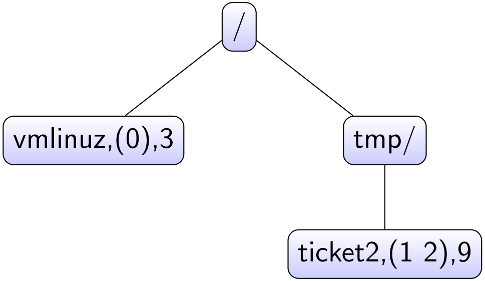


Table: Disk and allocation vector

0	\0\0\0	true
1	Wed 01:0	true
2	0	true
3	Tue 21:0	false
4	0	false
5		false

Outline

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.
- Read `n` characters starting at position `start` in the file at path `hns` in filesystem `fs`:
  `l1-rdchs(hns, fs, start, n)`
- Write string `text` characters starting at position `start` in the file at path `hns` in filesystem `fs`:
  `l1-wrchs(hns, fs, start, text)`

Proof approaches and techniques

- First read-over-write theorem: 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
```
- Second read-over-write-theorem: 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

- 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.
- 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 example: read-over-write in model 2

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

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: read-over-write in model 2

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

Figure: l2-read-over-write-1

Source analysis

Table: Code (over 3 semesters)

sloccount (lines of ACL2 code)	4964
defun events (function definitions)	106
defthm events (lemmas and proofs)	374

Outline

Future work

- Model and verify file permissions.
- 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.