Verifying filesystems in ACL2

Project report, CS380L Fall 2017

Mihir Mehta

Department of Computer Science
University of Texas at Austin

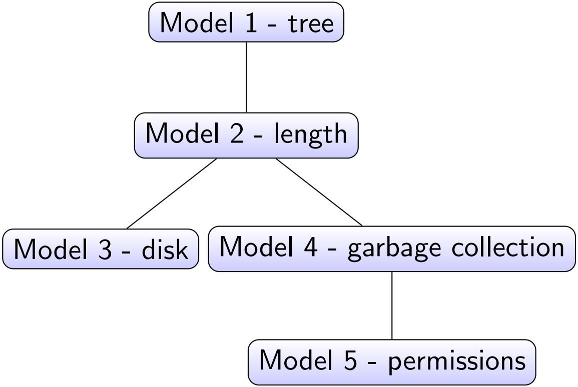
mihir@cs.utexas.edu

07 December, 2017

Why we need a verified filesystem

- 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.
- Work on this project started last year - since then I've built 5 increasingly complex models.
- The plan is to model FAT32, adding features in every model and maintaining proofs of correctness.

File system models



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.

More about the 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.
- Model 5: Model 4 with permissions for read/write for the user and others (no groups as yet)

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

- The first read-over-write theorem defines the semantics of reading from a location after writing to the same location. Formally, assuming `n = length(text)` and suitable "type" hypotheses (omitted here):

```
l1-rdchs(hns, l1-wrchs(hns, fs, start, text),
                start, n) =
text
```

- The second read-over-write theorem defines the semantics of reading from a location after writing to a different location. 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 through 5, we have proofs of correctness of the two read-after-write properties, making use of the proofs of equivalence between models and their successors.
- For model 5, the proof assumes the permissions are granted.
- The proof of the converse property - that reads and writes fail when permissions are not granted - remains to be done.

Proof example: first 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: first 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 written for this project

Source lines of code (ACL2)	6017
defun events (function definitions)	118
defthm events (lemmas and proofs)	419