Verifying filesystems in ACL2

Towards verifying file recovery tools

Mihir Mehta

Department of Computer Science
University of Texas at Austin

mihir@cs.utexas.edu

09 March, 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 might be nice, it might be nice to verify that the filesystems and the tools actually provide the guarantees they claim to provide.

Our approach

- Build a series of models, each providing a minimal set of operations and proofs of correctness of these operations.
- Per Ghemawat et al in SOSP 2003, a minimal set of operations can suffice - create, delete, open, close, read, and write files.
- Increasing the complexity of these operations with each model while proving equivalence with the previous model as we go would make the proofs tractable.
- 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.

Progress so far

- We've built three models, with the operations read, write, create, and delete.
  1. In the first model, we represent the filesystem as a tree, allow for text files (stored as strings) and directories only, and store no metadata.
  2. In the second model, we add some metadata to our tree representation - namely, file length. We introduce a rudimentary fsck and prove that our operations of writing and deleting preserve correctness under fsck.
  3. In the third model, we retain the tree but also introduce an unlimited "disk" of fixed-length character blocks. We do away with the explicit strings holding the contents of text files and instead store lists of block indices in the tree.
- For each of these models, 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

- 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?

Proof approaches and techniques

1. 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.
   What about the complementary problem - making sure unused blocks are unmarked?
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.

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.