CS260 Project Proposal

Rob Bowden, David Holland, Eric Lu

April 17, 2017

1 Introduction

We are interested in file system crash recovery. Both the FSCQ and the Yggdrasil logic is single-threaded, so only one file system operation can be in progress at once, and FSCQ also needs to sync (repeatedly!) during/after each operation. These restrictions, particularly the latter, are unfortunate and we believe we can lift them.

The interesting part of this is verifying the post-crash recovery logic and functional correctness in the presence of concurrent execution. In the all-singing, all-dancing version of this project we verify full functional correctness of a complete file system, including crash recovery, and output C code that can be run in a kernel. This is not feasible in the amount of time available.

In order to finish something this semester we are taking at least the following simplification steps: (a) Using a small list of file system operations.[1] (b) Working with a model of a file system; in particular, allowing blocks to store Coq data structures directly. (c) Ignoring liveness. (d) Probably, admitting non-crash portions of the functional correctness proofs.

We are prepared to ruthlessly simplify the file system and the model further as needed to preserve the core goals.

2 Goals

There are two things we want to prove. One is correctness, specifically crash correctness; the other is serializability.

For crash correctness, the proof for each file system operation is a proof of its corresponding Hoare triple, including a proof of the crash condition. At the level of talking to the file system, operations on files like e.g. `write` should be specified in terms of traces of file operations. (These can then be shown to be a refinement of lower-level traces.) Such a trace can be written

```
Inductive fileop: Set :=
| FileWrite: bytes(*data+length*) ->
    nat (*offset*) -> fileop
| FileTruncate: nat (*filesize*) -> fileop
end.
Inductive file_trace nat (list fileop) :=
| FileTrace: forall j ops:
     j <= length(ops) -> file_trace j ops.
```

which has this crash condition:

```
forall f j ops,
exists k, j <= k /\ k <= length ops,
{{ trace_of_file f = file_trace j ops }}
crash
{{ trace_of_file f = file_trace k (take k ops) }}
```

The last operation guaranteed to be on disk is `j` but we may retain more than that in a crash. The Hoare triples for `write` and `fsync` can be written as

```
forall f j ops bytes len,
{{ trace_of_file f = file_trace j ops }}
write f bytes len
{{ trace_of_file f =
     file_trace j (ops ++ [FileWrite bytes len]) }}

forall f j ops,
{{ trace_of_file f = file_trace j ops }}
fsync f
{{ trace_of_file f = file_trace (length ops) ops }}
```

The complete correctness theorem is `/\` over all the file system operations. Serializability:

[1] Currently lookup, create, unlink, read, write, truncate, fsync, and sync.

```
Inductive vfsop: Set := ...
Inductive vfsx (*execution*): Set :=
| VfsOp: vfsop -> vfsx
| VfsSeq: vfsx -> vfsx -> vfsx
| VfsPar: vfsx -> vfsx -> vfsx
end.
Inductive Serial (*no VfsPar*):
   vfsx -> Prop := ...
Definition Serializes:
   vfsx -> vfsx -> Prop := ...
Definition Equivalent:
   vfsx -> vfsx -> Prop := ...
Theorem vfs_serializable:
   forall execution,
   exists execution',
      Equivalent execution execution' /\
      Serial execution' /\
      Serializes execution execution'.
```

`Serializes` is like `Permutation` but more complicated, so it's not immediately clear up front how best to represent it. `Equivalent` should be expressed in terms of file traces over all files.

3 Schedule

There are three separate goals that will be split amongst the three team members:

1. Writing the file system (David)
2. Proving the file system is correct according to our concurrent crash Hoare logic (Rob)
3. Proving that the concurrent crash Hoare logic is sound (Eric)

These assignments will not be strict, given that the tasks are not of equal size.

There are four weeks remaining before the deadline of May 8th. The goal for the first two weeks is to have a "proof of concept" working, whereby a single operation is verified correct under a very simplified file system model. This includes concurrency, asynchronous write back caching, write ahead logging, and recovery.

Then, for the third week, we will work on verifying the remaining operations. At this point, ideally the concurrent crash Hoare logic will have been proven sound, so Eric can move to proving things about the file system. Hopefully, the proof of concept will provide a template from which the remaining operations can be verified.

Finally, with all of the operations verified individually, in the final week we aim to prove the serializability of operations run in parallel. There are enough pieces that we hope to be able to get *something* verified, even if it means cutting multiple operations and features.

4 Future Work

Everything we don't get done in the course of the semester (toward the full version of the project) or that we've simplified away is future work: handling bit encoding of file system metadata, supporting subdirectories and the full set of directory operations, generating C code, generating C code that will fit into a real kernel, etc.

It might be interesting to emit Frama-C notations into the C output to allow crosschecking it.

An underlying goal is to produce a framework that can be used with more than one file system model, so in addition to modeling a very basic made-up file system like we're starting with, we might model one or more real file systems. Proving the recovery theorem in the original FFS paper would be an interesting exercise, but possibly maddening since it requires a much more relaxed notion of correctness than we're aiming for and that increases the complexity.

It would be interesting to reason about the complete state of the file system as seen by different processes and whether or not the state we recover to is consistent with all views, including possibly with side channels inducing additional ordering constraints.

Then there's the question of different disk models, and different degrees of control over the on-disk cache. One could look into proving temporal bounds on data loss and could also try to prove the absence of degenerate performance.

And of course, one should prove liveness, since without that it's correct to just fail after every crash.