A Formal Model of a Virtual Filesystem Switch *

Gidon Ernst Gerhard Schellhorn Dominik Haneberg Jörg Pfähler Wolfgang Reif
Institute for Software and Systems Engineering, University of Augsburg, Germany
{ernst,schellhorn,haneberg,joerg.pfaehler,reif}@informatik.uni-augsburg.de

This work presents a formal model that is part of our effort to construct a verified file system for Flash memory. To modularize the verification we factor out generic aspects into a common component that is inspired by the Linux Virtual Filesystem Switch (VFS) and provides POSIX compatible operations. It relies on an abstract specification of its internal interface to concrete file system implementations (AFS). We proved that preconditions of AFS are respected and that the state is kept consistent. The model can be made executable and mounted into the Linux directory tree using FUSE.

1 Introduction

The popularity of Flash memory as a storage technology has been increasing constantly over the last years. Flash memory offers a couple of advantages compared to magnetic storage: It is less susceptible to mechanical shock, consumes less energy and offers higher speed when reading data. However, Flash memory can only be written sequentially, and memory cells must be erased in rather large blocks before they can be written again.

Two approaches exist to deal with these special characteristics: Standard file systems can be used if the hardware has a built-in *Flash translation layer* (FTL) that emulates behavior of magnetic storage. USB drives and Solid State Disks fall into this category. In contrast, Flash file systems (FFS for short) are specifically designed to work on "raw-flash". FFS are commonly used in embedded systems such as home routers, example implementations are YAFFS and JFFS. More recently, *UBIFS* [11] has become part of the Linux kernel and represents the state of the art. An FFS can in principle be more efficient than the combination of FTL and a traditional file system.

Flash memory is beginning to be used in safety-critical applications, leading to high costs of failures and correspondingly to a demand for high reliability of the file system implementation. As an example, an error in the software access to the Flash store of the Mars Exploration Rover "Spirit" already had nearly disastrous consequences [15]. As a response, Joshi and Holzmann [12] from the NASA JPL proposed in 2007 the verification of a Flash file system as a pilot project of Hoare's Verification Grand Challenge [10] and for use in future missions.

We are developing such a verified Flash file system as an implementation of the POSIX file system interface [20], using UBIFS as a blueprint. Our goal is that it can either be used *stand-alone* or in *Linux*.

The effort is structured into layers that are connected by refinement, corresponding to the various logical parts of the file system. The top level is an abstract formal model of the file system interface as defined in the POSIX standard. It serves as the specification of the functional requirements, i.e., what it means to create/remove a file/directory and how the contents of files are accessed.

The POSIX interface addresses files and directories by *paths* and views files as a linear sequence of bytes. Such high-level concepts are typically mapped to a more efficient data representation in the file system, in particular a graph structure. In Linux, this mapping is realized by the *Virtual Filesystem Switch*

*The final publication is available at arXiv.org via http://dx.doi.org/10.4204/EPTCS.102.5

Submitted to:
SSV 2012

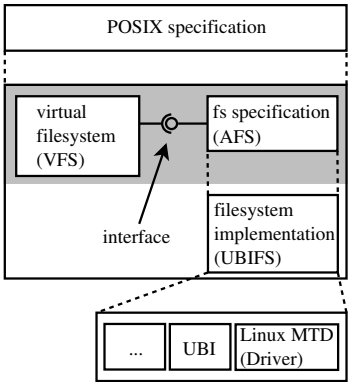

Figure 1: Components and Structure

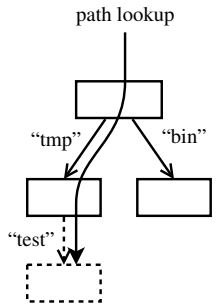

Figure 2: File system graph

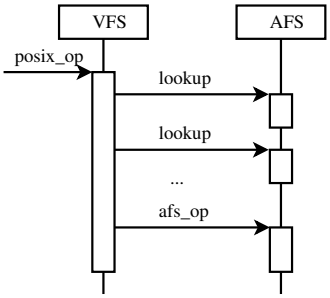

Figure 3: VFS/AFS interplay

(VFS). It implements many generic operations that are common to all file systems, e.g., permission checks and management of open file handles. VFS relies on concrete file systems – such as UBIFS – to provide lower-level operations. To that purpose, VFS defines an internal interface. The advantage of this scheme is *separation of concerns* and code reuse.

To achieve a fully verified POSIX compatible file system we have to provide our own implementation of the VFS functionality. Therefore, we define a formal specification of the main operations of the internal interface of the Linux VFS, called AFS[1](for abstract file system) in the following.

Figure 1 visualizes the different components displayed as boxes. Functional correctness is established by several nested *refinements* (or simulation relations) that are graphically shown as dotted lines. In particular, the a proof of the topmost refinement from POSIX to VFS+AFS implies the correctness of the model presented in this paper.

Note that AFS is refined to a Flash File System independently of VFS. As a consequence, VFS on top of the concrete FFS instead of AFS also refines POSIX, i.e., functional correctness propagates in a compositional fashion.

Previously published is our formalization of the core concepts of UBIFS [18] which deals with keeping the index data structures consistent. A POSIX model and the two refinements from POSIX to VFS and from AFS to UBIFS are ongoing work. We also work on the formalization of other layers, including UBI that takes the role of an FTL.

Our specification language is *Abstract State Machines* [2], which define operations over abstract data types. We use algebraic specifications to axiomatize data types, and our interactive theorem prover KIV [16] to verify properties.
In summary, the contribution of this work is:

- A model of operations common to file systems, i.e., a Virtual Filesystem Switch, that provides all essential POSIX file system interface. Standard POSIX operations are broken down to a number of AFS operations. Notably, linear access to file content is mapped to a sparse array of pages. Besides that, we implement path lookup, access permission checks and management of handles for open files. The model is thus very close to an implementation.

- The AFS model for file systems that can be plugged into the above VFS. It abstracts details of concrete file system operations into a generic specification and encodes the assumptions made by

[1]Not to be confused with the Andrew File System

the VFS. Thus, if a file system implementation respects the given contracts, it can be used directly within our VFS. In contrast to the VFS model, AFS remains as abstract as possible.

- We verified that AFS preconditions are respected by the VFS and we proved several consistency invariants about the state.

Furthermore, we have derived an executable Scala simulation from the models that can be mounted directly into the Linux directory tree using the file system in user space library (FUSE [19]). The simulation is used for testing and validates the models with respect to POSIX. For reasons of space we do not present the full models in this paper (AFS: ∼100 LOC, VFS: ∼500 LOC). Together with the proofs of invariants and the code of the simulation they are available online [6].

The remainder of this paper is structured as follows: Section 2 describes the VFS data model and operations, focusing on structural modifications of the file system. Section 3 describes the AFS data types, operations and invariants. We then turn to access of file content through file handles in Sec. 4. Section 5 discusses related work and Sec. 6 concludes.

2 The VFS Layer

This section shows, how the VFS layer realizes the top-level POSIX operations. Conceptually, the file system consists of directories and files that are organized hierarchically in a *directed acyclic graph* with the toplevel directory as root node, as visualized in Fig. 2. File system objects are addressed by paths, which are sequences of names concatenated by the separator / (resp. \ on Windows). Operations can be classified into structural modifications, such as creating/deleting files, and content modification, such as reading/writing.

Creation of a new file will be used as running example throughout the paper. The following C source code creates a file named “test” in the top-level directory named “tmp”, using the `creat(3)` operation:

```
int err = creat("/tmp/test", 0644);
```

The given path is parsed, each segment but the last is looked up in the directory tree (starting from the root) and finally a file with some access permissions (here: 0644) is created. The return value indicates success or a specific error condition. The new subgraph arising from the `creat` operation is indicated by dotted lines in Fig. 2.

The task of the VFS layer is to break down such high-level POSIX operations to several calls of AFS operations. Fig. 3 visualizes a typical sequence for structural operations like `creat`. In this case, it relies on three operations provided by the file system implementation, namely

1) lookup of the target of a single edge in the graph
2) retrieve the access permissions at each node that is passed during path lookup
3) actual creation of the file

In Linux, the AFS interface is realized by a set of function pointers. The file system specific create operations have the following signature:

```
int (*create)(struct inode *dir, struct dentry *dent,
              int mode, struct nameidata *nd);
```

The first parameter, `dir`, points to some object representing the parent directory of the new file (“/tmp” in the example above). The second parameter, `dent`, specifies the name of the new file.

The types of the formal parameters of the internal interface constitute the *VFS data model*. All file system queries and modifications are expressed in terms of these data structures.

2.1 Data Model

This section defines an algebraic specification of the VFS data model. Three main data structures represent the file system graph: *inodes*, *dentries* and *pages*. They are *communication* data structures and do not necessarily reflect the file system's runtime state and the on-disk data structures.

Inodes ("Index Nodes") correspond to the nodes of the graph, i.e., the files and directories. Inodes are uniquely identified by an inode number $ino : Ino \simeq \mathbb{N}$ and store some associated information. The sort *Inode* is formally defined as an algebraic data type:

$$\textbf{data}\ Inode\ =\ \texttt{inode}(\texttt{ino} : Ino,\ \texttt{meta} : Meta,\ \texttt{isdir} : \mathbb{B},\ \texttt{nlink} : \mathbb{N},\ \texttt{size} : \mathbb{N})$$

It has one constructor `inode` that records the inode number (`ino`), some metadata (`meta`), whether it corresponds to a file or a directory (`isdir`), the number of hard-links (inbound edges, `nlink`) and the file size resp. the number of directory entries in case of a directory inode (`size`). The abstract sort *Meta* is a placeholder for any further information that is associated with inodes. We postulate some selectors, to retrieve for example read, write and execute permissions following the ideas given in [9]. Constructor arguments are accessed by postfix selectors with the given names, for example $inode.\texttt{size}$ retrieves the size of an inode *inode*.

Dentries ("Directory Entries") correspond to the edges of the graph. They relate a parent directory to its children and are labelled with the respective *file names*. Dentries store a `name` and come in two flavors: Normal dentries point to an existing file identified by the selector `target`. Negative dentries indicate that a file name is *not* contained within a directory – they are used for example as return value of the lookup operation.

$$\textbf{data}\ Dentry\ =\ \texttt{dentry}(\texttt{name} : String,\ \texttt{target} : Ino)\ \mid\ \texttt{negdentry}(\texttt{name} : String)$$

The content of files is partitioned into uniformly sized *pages*. This has several advantages: The size of pages typically corresponds to the size of a virtual memory page, enabling caching and memory-mapped input/output. Furthermore, sparse files, i.e. files with large empty parts, can be represented efficiently by the convention that non-present pages contain zeros only. Formally, pages are arrays of bytes of a fixed length, specified by the constant `PAGE_SIZE`:

$$\textbf{type}\ Page\ =\ Array[\texttt{PAGE\_SIZE}]\langle Byte\rangle$$

2.2 Operations

To continue the example, the signature of the create operation in our model is:

```
vfs_create(path : Path, md : Meta; err : Error)
```

where paths are sequences of strings (the separator is implicit) and errors are given by an enumeration of possible error constants (where `ESUCCESS` denotes "no error"):

$$\textbf{type}\ Path = Seq\langle String\rangle \qquad \textbf{data}\ Error = \texttt{ESUCCESS} \mid \texttt{EIO} \mid \ldots$$

As a convention, we prefix VFS and AFS operations with `vfs_` resp. `afs_` to indicate that they are part of the formal model. The semicolon in the parameter list separates input parameters from reference parameters. Thus, assignments to `err` in the body of the operations are visible to the caller.

VFS (and AFS) operations are defined by *rules* of an Abstract State Machine (ASM) [2]. The language features typical programming constructs such as parallel (function) assignments, conditionals,

```
vfs_create(path, md; err)
  if path = [] then
    err := EACCESS
  else let
    ino    = ROOTINO,
    dent   = negdentry(path.last)
    path   = parent(path)
  in vfs_walk(path; ino, err);
     if err = ESUCCESS then
        vfs_maycreate(ino; dent, err);
     if err = ESUCCESS then
       afs_create(ino, md; dent, err)
```

```
vfs_walk(path; ino, err)
  err := ESUCCESS;
  let path = path in
  while path ≠ [] ∧ err = ESUCCESS do
    vfs_maylookup(ino; err);
    if err = ESUCCESS then
    let dent = negdentry(path.first)
    in  afs_lookup(ino; dent, err);
        if err = ESUCCESS then
          ino := dent.target
          path := path.rest
```

Figure 4: ASM rules of the VFS create operation

loops, nondeterministic choice and recursive procedures. ASM rules are executable, provided that the nondeterminism is resolved somehow and the algebraic operations on data types are executable.

All VFS operations perform extensive error checks. Similar to Hesselink and Lali [9], we ensure that the file system is guarded against unintended or malicious calls to operations. Specifically, all operations are total (defined for all possible values of input parameters) and either succeed or return an error *without* modifying the internal state. Some implications of these checks manifest as preconditions in AFS (and further refinements), as discussed in Sec. 3.2.

Figure 4 shows the ASM rules that realize the create operation in the syntax of our specification language. The entry point is `vfs_create`. It receives the path to the new file and ensures that it is non-empty. Subsequently, the helper routine `vfs_walk` determines the parent directory of the new file, identified by the inode number returned in `ino`. The path walk itself performs multiple lookups of directory entries by calling the AFS routine `afs_lookup(ino; dent, err)`. Informally, `afs_lookup` checks, whether the name given by `dent` is contained in the directory identified by `ino` and sets `dent.target` to the inode number of the child or returns an error. In the latter case, the whole operation is aborted. The subroutine `vfs_maycreate` ensures that the user has sufficient permissions to create a file in the parent directory and that the name is not already present. Creation of the file is then delegated to `afs_create(ino, md; dent, err)` (shown in Fig. 5), which allocates a new inode number and modifies its internal state. The new inode number is returned in `dent.target`.

The formal model supports the following operations besides `create`:

The operation `mkdir` creates a new empty directory, conversely, `rmdir` removes an existing directory, which must be empty. The operation `link` introduces hard-links to existing files, i.e., they introduce additional edges in the graph between existing nodes, so that a file becomes accessible through multiple paths. The converse operation is `unlink`, which also deletes the file on disk when the last link is removed (and the last file handle is closed, see Sec. 4.2). The most complex structural operation is `rename` (AFS rule shown in Fig. 6), which allows to change the name and optionally the parent directory of a file or directory. It performs two path walks and checks a couple of error conditions.

The operation `open` returns a *file descriptor* through which the content of the file can be accessed with `read`, `write`, and `seek`. The operation `close` invalidates a descriptor and frees associated resources. The size of a file can be changed with `truncate`, meta data is accessed with `getattr` and `setattr`. Finally, the operation `readdir` returns the filenames contained in a directory.

3 The AFS Layer

AFS is also realized as an Abstract State Machine, i.e., it is an *operational* specification of expected behavior of concrete file systems. The design goal is to remain as abstract as possible.

3.1 State

AFS maintains an internal state, consisting of files and directories. These are kept in two separate stores (partial functions), mapping inode numbers to the respective objects; we use the symbol $\rightarrow\!\!\!\!\!\!+$ to denote partial function types.

state vars $dirs : Ino \rightarrow\!\!\!\!\!\!+\ Dir, \quad files : Ino \rightarrow\!\!\!\!\!\!+\ File \qquad$ where $Ino \simeq \mathbb{N}$

The separation is motivated by the distinction into structural and content modifications: the former will affect mainly *dirs* while the latter will affect only *files*. We expect this decision to facilitate the refinement proofs between the POSIX layer and VFS. However, it comes at the cost of an extra disjointness invariant (see Sec. 3.4).

Directory entries are *contained* in the parent directory, likewise, pages are contained in the file object they belong to:

data $Dir = \texttt{dir}(\texttt{meta} : Meta,\ \texttt{entries} : String \rightarrow\!\!\!\!\!\!+\ Ino)$
data $File = \texttt{file}(\texttt{meta} : Meta,\ \texttt{size} : \mathbb{N},\ \texttt{pages} : \mathbb{N} \rightarrow\!\!\!\!\!\!+\ Page)$

Inode numbers $ino \in (\texttt{dom}(dirs) \cup \texttt{dom}(files))$ are called *allocated*, they refer to valid directories resp. files. We often omit the dom-operator for brevity, as in $ino \in dirs$. We write application of partial functions using square brackets, e.g., $dirs[ino]$ retrieves the directory identified by *ino*. Function update is written $dirs[ino] := d$ for a directory d. We require explicit allocation of inode numbers for *dirs* and *files*, $dirs +\!+ ino$ denotes the store *dirs* with an additional mapping for *ino* (to an arbitrary directory). Conversely, deallocation is written as $dirs --\ ino$. Empty stores are written as $\emptyset$. Similar conventions apply to the stores *files*, `entries` and `pages`, except that the latter two do not require explicit allocation.

3.2 Operations

ASM rules in the AFS model have the form `if` *pre* `then` *actions* with a precondition *pre*. These preconditions roughly correspond to the error checks performed by VFS. We proved that all calls from VFS indeed establish the AFS preconditions. In contrast to the axiomatic approach of Hoare-style contracts, postconditions are implicitly given by the effect of the actions, namely, the outputs and state transitions.

All structural operations, such as `vfs_create`, have corresponding AFS counterparts. Figure 5 shows the ASM rules of `afs_lookup` and `afs_create`.

The lookup operation tests whether a parent directory identified by the inode number `pino` contains an entry with a specific name, given by the (negative) directory entry `dent`. It returns a positive directory entry that points to the child's inode number or a negative directory entry and the error "no entry". The test `pino` $\in$ `dirs` represents the precondition of the operation: it may only be invoked with a valid parent inode number.

The create operation allocates a new inode number `ino`. It is added to the `entries` slot of the parent directory under `dent.name`. The new file object has the given metadata, a size of zero and no pages. The operation requires that `pino` refers to a valid directory and that the name is not already present.

```
afs_lookup(pino; dent, err)
if pino ∈ dirs then
  if dent.name ∈ dirs[pino].entries
  then let
    ino = dirs[pino].entries[dent.name]
    in dent := dentry(dent.name, ino)
       err := ESUCCESS
  else dent := negdentry(dent.name)
       err := ENOENT
```

```
afs_create(pino, md; dent, err)
if   pino ∈ dirs
   ∧ dent.negdentry?
   ∧ dent.name ∉ dirs[pino].entries
then
  choose   ino with ¬ ino ∈ dirs
         ∧ ¬ ino ∈ files ∧ ino ≠ 0 in
  dirs[pino].entries[dent.name] := ino
  files := files ++ ino
  files[ino] := file(0, md, ∅)
  dent := dentry(dent.name, ino)
  err := ESUCCESS
```

Figure 5: ASM rules of the AFS create and lookup operations

```
afs_rename(oldino, newino; olddent, newdent)
  if   oldino ∈ dirs
     ∧ newino ∈ dirs
     ∧ olddent.target ∈ dirs[oldino].entries
     ∧ ( olddent.ino ∈ files ∧ newdent.ino ∈ files
       ∨ olddent.ino ∈ dirs  ∧ newdent.ino ∈ dirs )
     ∧ (newdent.ino ∈ dirs → dirs[newdent.ino].entries = ∅)
     ∧ ;; whether olddent/newdent must be positive/negative,
       ;;         and their .target inodes are allocated
  then
    { dirs[oldino].entries := dirs[oldino].entries -- olddent.target;
      dirs[newino].entries[newdent.target] := olddent.target }
    olddent := negdentry(olddent.target)
    newdent := dentry(newdent.target, olddent.target)
    err := ESUCCESS
```

Figure 6: ASM rules of the AFS rename operations

The AFS rename operation is shown in Fig. 6. It takes two inode numbers and two directory entries. The file or directory is simply renamed if `oldino` = `newino`. Otherwise, it is additionally moved into a different parent directory. It is possible to overwrite an existing destination, if it has the same type (file resp. directory) and in the latter case the destination is empty. Furthermore there are some consistency conditions on both dentries. The state transition consists of two directory modifications. Note that sequential composition (emphasized by curly braces) in the given order is significant, since the two statements might affect the same parent directory.

The remaining non-structural AFS operations are: The pair of operations `afs_readinode` and `afs_writeinode` construct *Inode* instances and write back changes. Similarly, `afs_readpage` and `afs_writepage` read and write whole pages. The operation `afs_evict` deletes unreferenced files. The operations `afs_readdir` and `afs_truncate` complete the list.

3.3 Nondeterministic Errors

AFS operations assume infinite, perfect storage: allocation always succeeds and *dirs* and *files* can be accessed reliably. However, transient and persistent failures are quite common with real Flash hardware; allocation may fail due to insufficient memory, and the Flash device can be full. Since the AFS model

is too abstract to capture when exactly such errors arise, most AFS operations nondeterministically fail with an error code selected from a set of low-level errors (the exception is `afs_evict`, see Sec. 4.2). This is achieved by modifying the body of operations as follows:

```
{ choose e with e ∈ {EIO,...} in err := e } or { normal code }
```

3.4 Invariants

We have proved the following invariants on the AFS state: inode numbers of files and directories are disjoint, never 0 and there is a root inode (1). Invariant (2) states closure under lookup: following a directory entry leads to an allocated inode number. Directories have at most one parent (3) and there are no superfluous pages beyond the file size (4). For all *ino*, *str*, *n*:

$$0 \notin dirs \quad \text{and} \quad 0 \notin files \quad \text{and} \quad \texttt{ROOT\_INO} \in dirs \quad \text{and} \quad dirs \cap files = \emptyset \tag{1}$$

$$ino \in dirs \wedge str \in dirs[ino].\texttt{entries} \rightarrow dirs[ino].\texttt{entries}[str] \in (dirs \cup files) \tag{2}$$

$$ino \in dirs \rightarrow \#\,\texttt{links}(ino, dirs) \leq 1 \tag{3}$$

$$ino \in files \wedge n \in files[ino].\texttt{pages} \rightarrow n * \texttt{PAGE\_SIZE} < files[ino].\texttt{size} \tag{4}$$

The last page of a file (if present) contains zeros in the part that is outside the file size (5)

The proofs are not difficult, a couple of helper lemmas lead to high automation. The verification crucially relies on the AFS preconditions. For example:

- The `link` operation requires the target to be a file, otherwise, invariant (3) may be violated.
- The `rmdir` operation requires that the path is not empty, so that the root directory is never deleted (invariant (1)).

4 File Access and File Handles

The external POSIX view of file content is a sequence of bytes that is accessed indirectly through *file descriptors*, passed to the operations read and write. An example write to the file "/tmp/test" using the C interface is:

```
int fd = open("/tmp/test", O_WRONLY);
write(fd, "Hello, World!", 13);
close(fd);
```

The first line opens the file for writing, yielding a descriptor `fd`. Read and write operations take a memory buffer and a length specifying how many bytes to transfer from/to this buffer. In the example, the buffer contains the string "Hello, World!" with a length of 13 bytes. The VFS keeps track of the current read/write offset into the file. Access is sequential and writes at the end of the file implicitly increase its size.

The VFS state consists of a registry *oh* of open file handles, formalized as:

state var $oh : \mathbb{N} \nrightarrow Handle$ where

data $Handle = \texttt{handle}(\texttt{ino} : Ino, \texttt{pos} : \mathbb{N}, \texttt{mode} : Mode)$

We have proved the following invariant on the VFS state. For all *fd*:

$$fd \in oh \rightarrow oh[fd].\texttt{ino} \in files \tag{6}$$

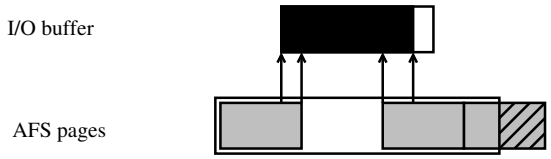

Figure 7: Read with unallocated pages

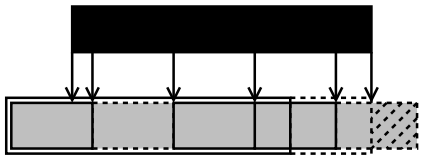

Figure 8: Write that extends the file size

The VFS client may obtain a file descriptor $fd : \mathbb{N}$ by the call to `vfs_open(path, mode; fd, err)` and release it after use with `vfs_close(fd; err)`. The `pos` slot of a handle can be modified by the operation `vfs_seek(fd; n, whence, err)`, returning the updated position (the parameter `whence` specifies whether `n` is to be understood relative to `pos` or absolute).

The signatures of read and write are `vfs_read(fd; buf, len, err)` and `vfs_write(fd, buf; len, err)`, where buffers `buf` are arrays of bytes (**type** $Buffer = Array\langle Byte\rangle$)

4.1 Read & Write

Reading and writing maps a linear buffer onto the file's array of pages. There is a number of special cases that must be considered.

An example read operation is visualized in Fig. 7. At the bottom, the file's pages are denoted by gray boxes with an outer frame indicating the file size. The hatched part of the last page does not contribute to the file's content and must always contain zeros. The white space among the pages denotes an unallocated page, which implicitly represents a range of all zeros in the file.

The rectangle in the middle denotes the destination buffer (parameter `buf`). The black part corresponds to the range that should be read (parameter `len`). The buffer may be larger than `len` (white part). The read operation loads the affected pages sequentially and copies the required parts (graphically delimited by arrows) into the buffer.

Listing 9 shows the core helper procedure that is called in a loop until done or an error occurs. Parameters `start` and `end` define the range to read, as absolute positions into the file in bytes. Of this range, `total` bytes have been processed so far (note that `done`, `buf` and `total` are passed by reference).

The procedure computes the current page and offset into that page and considers three upper bounds for the length of the range to copy in this iteration: the maximum number of bytes to transfer (`len`), the end of the current page, and the end of the file. Note that for a top-level read operation, each page is loaded at most once. Nonexistent pages are handled in AFS:

```
afs_readpage(ino, pageno; page, err)
  let pages = files[ino].pages in
    page := if (pageno ∈ pages) then pages[pageno] else emptypage
    err  := ESUCCESS
```

Writing is done similarly, an example is shown in Fig. 8, assuming the operation is executed with the same file as in Fig. 7. A write operation may extend the file at the end. In this case, the dotted lines indicate the newly allocated parts of the file: the missing second page is written, as well as an additional page at the end. The fourth page is overwritten and becomes part of the file entirely. The write operation relies on a helper routine similar to `vfs_read_block`, an excerpt is shown in Fig. 10. The affected page is loaded, modified and then written back.[2] Writes that extend the file additionally call `afs_truncate` with the new size.

[2]The `afs_readpage` could be optimized away if `n = PAGE_SIZE`

```
vfs_read_block(start, end, inode; done, buf, total, err)
  if   ¬ done ∧ start + total ≤ inode.size
     ∧ total ≤ # buf ∧ inode.ino ∈ files
     ∧ err = ESUCCESS then
  let pageno = (start + total) / PAGE_SIZE        ;; integer division
      offset = (start + total) % PAGE_SIZE        ;; and     modulo
      page   = ?
  in ;; bytes to read in this iteration
     let n = min(end         - (start + total)    ;; read size    boundary
                 PAGE_SIZE   -  offset            ;; current page boundary
                 inode.size - (start + total)) in ;; inode size   boundary
     if n ≠ 0 then
       afs_readpage(inode.ino, pageno, dirs, files; page, err)
       if err = ESUCCESS then
         buf    := copy(page, offset, buf, total, n)
         total := total + n
     else done := true
```

Figure 9: ASM rule to read a partial page

```
if n ≠ 0 then
  afs_readpage(inode.ino, pageno, dirs, files; page, err)
  if err = ESUCCESS then
    page := copy(buf, total, page, offset, n)
    afs_writepage(inode.ino, pageno, page, dirs; files, err)
```

Figure 10: ASM rule to write a partial page (excerpt)

A write that completely falls beyond file size leads to a gap in the file that must contain zeros. That no zeros have to be written in between is guaranteed by the invariants (4) and (5).

A further detail is that a read/write may be successful, even if *less* than `len` bytes were transferred, i.e., some intermediate page access failed, for example if the storage medium becomes full.

4.2 Deletion

The POSIX standard allows file handles to point to files that are not part of the directory tree any more. This situation occurs, when the last link to an open file is removed. The manual of `close(3)` specifies:

"If the link count of the file is 0, when all file descriptors associated with the file are closed, the space occupied by the file shall be freed and the file shall no longer be accessible."

Some applications actively exploit this feature to create hidden temporary files (MySQL caches, Apache SSL mutex). Replacement of files during a system update is another use case: existing files, in particular application binaries, are unlinked before the new version is written. The command `lsof +L1` can be used to detect applications that still refer to a deleted file.

Since the AFS layer does not know about open files (it can not access *oh*), VFS needs to signal explicitly when files become obsolete, which is done by a call to the operation `afs_evict`. The helper routine `vfs_putinode` is called after a file handle has been closed and after a link to a file has been removed.

```
vfs_putinode(ino; err)
  if ¬ is-open(ino, oh)
  then afs_evict(ino; err)
```

```
afs_evict(ino; err)
  if links(ino, dirs) = []
  then files := files -- ino
  err := ESUCCESS
```

where $\texttt{is-open}(ino, oh) :\leftrightarrow \exists\, fd \in oh.\ oh[fd].\texttt{ino} = ino$

As previously indicated, `afs_evict` must always succeed. The reason is that the state has already been modified and thus the corresponding VFS call may not fail any more (see Sec. 2.2). The requirement not to fail is reasonable, since `afs_evict` will be implemented as in-memory operation.[3]. The Linux VFS has a similar requirement (`void` return type of `evict`).

5 Related Work

File system correctness has been an active research topic for some time. An early model of the POSIX standard written in Z by Morgan and Sufrin is [14]. Several mechanized models have been developed related to this work [3, 9, 5, 4]. These approaches typically remain on a very high abstraction level, make strong simplifications, e.g., leave out hard-links or treat file content atomically. To our knowledge, the separation of common functionality (VFS) versus file system specific parts (AFS) has not been addressed previously.

The work of Kang and Jackson [13] is closest to our work with respect to read and write – it provides the same interface (buffer, offset, length). However, their model only deals with file content but not with directory trees or file handles. They check correctness with respect to an abstract specification for small bounded models. Kang and Jackson address further issues as well (fault tolerance and physical disk layout), that can be modularly realized as a separate layer in our refinement chain. In comparison, their read and write algorithm is less practical than ours, because it relies on an explicit representation of a list of blocks that needs to be modified during an operation.

Arkoudas et al. [1] address reading and writing of files in isolation (without file handles). Their model of file content is similar to ours (i.e., non-atomic pages). They prove correctness of read and write with respect to an abstract POSIX-style specification. However, their file system interface allows only to access *single bytes* at a time, which is a considerable simplification.

Damchoom et al. [5] start with a graph-based specification of file system operations, with an interface similar to our AFS layer. Their model differs from the VFS data model by using parent pointers to encode the graph. In [4] Damchoom et al. decompose the write operation with respect to pages, however, not down to bytes. They use shadow copies of whole files to achieve abstract fault tolerance, which is not realistic.

Ferreira et al. [7] et. al. provide a POSIX-like specification at the level of paths.

6 Discussion and Conclusion

We have presented a formal model of a Virtual Filesystem Switch and an abstract specification of its internal interface, inspired by the implementation of the POSIX file system interface in Linux.

Our model has two notable simplifications that are visible in the external interface: Symbolic links are not supported since we feel that they are a secondary concern. In POSIX, `readdir` is specified only

[3] Technically, in UBIFS, the modification has already been recorded in the on-flash journal and `evict` only removes the file and its content from the RAM index.

by a C library interface based on further data structures that out of scope of this model. We therefore chose a simple implementation of `readdir`. In contrast to the corresponding Linux system call, it is not based on directory handles and returns all entries at once.

The effort to develop these models was dominated by two factors: On one hand, we spent much time studying the Linux source code, figuring out the interplay between VFS and file system implementations and details of the semantics of operations (e.g., evict). On the other hand, we had to ensure that the simulation proofs involving POSIX, the existing UBIFS model, and further refinements will work out (e.g., nondeterministic errors). The development of a POSIX model has been overlapped with this work in order to clarify the requirements to the VFS.

We estimate that the effort related to VFS and AFS was about four months, of which the technical aspects – writing ASM rules, specifying and proving invariants and properties – took roughly one month.

The modularization into VFS and AFS allows us to focus on the FFS internals in our future work.

Three important orthogonal aspects remain for future work:

A major decision was that the VFS layer does not store or cache any inode, dentry or page objects internally. We expect that caching can be introduced by a refinement of the VFS model without the need to change AFS.

Concurrency is an essential part of the Linux VFS. It leads to locking and synchronization and introduces additional complexity. Work in this direction could benefit from Galloway et al. [8], where Linux VFS code is abstracted to a SPIN model to check correct usage of locks and reference counters.

Fault tolerance against power loss is of great interest and we are currently proving that the model can deal with crashes anytime during the run of an operation, using the temporal program logic of KIV [17].

References

[1] K. Arkoudas, K. Zee, V. Kuncak & M. C. Rinard (2004): *On Verifying a File System Implementation*. In: *ICFEM*, pp. 373–390, doi:10.1007/978-3-540-30482-1_32.

[2] E. Börger & R. F. Stärk (2003): *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, doi:10.1007/3-540-36498-6.

[3] A. Butterfield & J. Woodcock (2007): *Formalising Flash Memory: First Steps*. In: *Proc. of the 12th IEEE Int. Conf. on Engineering Complex Computer Systems (ICECCS)*, IEEE Comp. Soc., Washington DC, USA, pp. 251–260, doi:10.1109/ICECCS.2007.23.

[4] K. Damchoom & M. Butler (2009): *Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B*. In Marcel Vinícius Oliveira & Jim Woodcock, editors: *Formal Methods: Foundations and Applications*, Springer, Berlin, Heidelberg, pp. 134–152, doi:10.1007/978-3-642-10452-7_10.

[5] K. Damchoom, M. Butler & J.-R. Abrial (2008): *Modelling and Proof of a Tree-Structured File System in Event-B and Rodin*. In: *Proc. of the 10th Int. Conf. on Formal Methods and Sw. Eng. (ICFEM)*, Springer LNCS 5256, pp. 25–44, doi:10.1007/978-3-540-88194-0_5.

[6] G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler & W. Reif (2012): *KIV models and proofs of VFS and AFS*. Available at `http://www.informatik.uni-augsburg.de/swt/projects/flash.html`.

[7] M.A. Ferreira, S.S. Silva & J.N. Oliveira (2008): *Verifying Intel flash file system core specification*. In: *Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop*, School of Computing Science, Newcastle University, pp. 54–71. Available at `http://twiki.di.uminho.pt/twiki/pub/Research/VFS/WebHome/overture08sl.pdf`. Technical Report CS-TR-1099.

[8] A. Galloway, G. Lüttgen, J.T. Mühlberg & R.I. Siminiceanu (2009): *Model-Checking the Linux Virtual File System*. In: *Proc. VMCAI'09*, Springer, pp. 74–88, doi:10.1007/978-3-540-93900-9_10.

[9] W.H. Hesselink & M.I. Lali (2012): *Formalizing a hierarchical file system*. *Form. Asp. Comput.* 24(1), pp. 27–44, doi:10.1007/s00165-010-0171-2.

[10] C. A. R. Hoare (2003): *The verifying compiler: A grand challenge for computing research*. *J. ACM* 50(1), pp. 63–69, doi:10.1007/3-540-36579-6_19.

[11] A. Hunter (2008): *A Brief Introduction to the Design of UBIFS*. Available at `http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf`.

[12] R. Joshi & G.J. Holzmann (2007): *A mini challenge: build a verifiable filesystem*. *Formal Aspects of Computing* 19(2), doi:10.1007/s00165-006-0022-3.

[13] E. Kang & D. Jackson (2008): *Formal Modelling and Analysis of a Flash Filesystem in Alloy*. In: *Proceedings of ABZ 2008*, Springer LNCS 5238, pp. 294 – 308, doi:10.1007/978-3-540-87603-8_23.

[14] C. Morgan & B. Sufrin (1987): *Specification of the UNIX filing system*. In: *Specification case studies*, Prentice Hall (UK) Ltd., Hertfordshire, UK, pp. 91–140, doi:10.1109/TSE.1984.5010215.

[15] G. Reeves & T. Neilson (2005): *The Mars Rover Spirit FLASH anomaly*. In: *Aerospace Conference, 2005 IEEE*, pp. 4186–4199, doi:10.1109/AERO.2005.1559723.

[16] W. Reif, G. Schellhorn, K. Stenzel & M. Balser (1998): *Structured Specifications and Interactive Proofs with KIV*. In W. Bibel & P. Schmitt, editors: *Automated Deduction—A Basis for Applications*, chapter 1, II, Kluwer, Dordrecht, pp. 13 – 39. Available at `http://www.informatik.uni-augsburg.de/de/lehrstuehle/swt/se/publications/1998-struc_spec_proofs_kiv/`. ISBN: 978-0-7923-5132-0.

[17] G. Schellhorn, B. Tofan, G. Ernst & W. Reif (2011): *Interleaved Programs and Rely-Guarantee Reasoning with ITL*. In: *Proc. of Temporal Representation and Reasoning (TIME)*, IEEE, CPS, pp. 99–106, doi:10.1109/TIME.2011.12.

[18] A. Schierl, G. Schellhorn, D. Haneberg & W. Reif (2009): *Abstract Specification of the UBIFS File System for Flash Memory*. In: *Proceedings of FM 2009: Formal Methods*, Springer LNCS 5850, pp. 190–206, doi:10.1007/978-3-642-05089-3_13.

[19] M. Szeredi: *File system in user space*. Available at `http://fuse.sourceforge.net`.

[20] The Open Group (2008): *The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2008 Edition*. Available at `http://www.unix.org/version3/online.html`. (login required).