Vrije Universiteit Amsterdam

Bachelor Thesis

Sample Title

Author: Alexander Balgavy (2619644)

1st supervisor: person
daily supervisor: person

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science.

June 27, 2020

Abstract

Abstract goes here.

1 Introduction

The introduction should cover

- why reliability is important
- specifically in filesystems - why do you need a reliable filesystem?
- what's done in this paper (high-level summary)

2 Background information

2.1 C as the implementation language

The choice of an implementation language may affect the bugs or vulnerabilities that are present in the system. The de-facto standard implementation language for operating systems and their components has long been C. Many popular file systems are implemented in C, such as Ext4. **ext4Source**. C is based on typeless languages, BCPL and B, which were developed specifically for operating system programming in early Unix. A design principle of C was to be grounded in the operations and data types provided by the computer, while offering abstractions and portability to the programmer. [1]

Describe studies on issues with C, bugs found, CVEs, etc.

2.2 Possible alternatives

- Restricting C: MISRA-C, Frama-C
- Rust: esp. reference ownership
- D: allows functional contracts
- Coq: allows writing formal specification, proving it, and extracting certified program from constructive proof of its specification in OCaml, Haskell, or Scheme.
- Ada

2.3 Formal verification

Explain what it is, particularly Hoare triples. Why is it useful?

2.4 FUSE

Why use it for development, what are its limitations?

3 Related work

3.1 FSCQ

3.2 Yxv6 & Yggdrasil

3.3 Argosy

3.4 COGENT

4 Design & implementation

- Language features: strong typing, lack of pointers (access types vs in-out parameters), modularisation using packages and private parts, interfacing with C
- FUSE & the FUSE driver
- Verification: contracts (functional and data). Also - what can't be verified (at the moment)?

To explore ways of writing reliable code, a small, prototype filesystem (AdaFS) was implemented using the Ada 2012 programming language. Some parts of this filesystem are written in the SPARK 2014 language, which is a subset of Ada that removes features not amenable to formal verification, and defines new aspects to support modular, constructive, formal verification [2]. The AdaCore GNAT Community 2020 package [1] is used, which provides, among others, the compiler and prover tools. For testing purposes, a FUSE driver is written in C, and the built executables are linked with libfuse 3.9.2 [2]. The GNAT Project Manager is used to facilitate compilation of source files in different languages and linking of other required libraries.

4.1 Filesystem design

AdaFS is based on the MINIX 2 filesystem [3], with some simplifications due to time constraints. The MINIX operating system was written by Andrew Tanenbaum as an educational tool, and is compatible with UNIX, but has a more modular structure. The MINIX filesystem was chosen as a model because it is not part of the operating system, but rather runs entirely as a user program. As such, it is self-contained. Furthermore, due to its educational purpose, it is thoroughly commented and easy to understand. The second version of the file system was chosen because MINIX 3 is more complex, as it adds numerous improvements (e.g. for reliability) to place emphasis on its use in research and production [4].

A disk is formatted as an AdaFS filesystem using the *mkfs* executable. The resulting disk layout is shown in Figure 1. The disk is divided into blocks of 1024 bytes, similarly to MINIX. Blocks are collected in zones, which can be of size 2^n blocks. This abstraction of blocks into zones can make it possible to allocate multiple blocks at once.

The disk begins with a boot block that would optionally contain executable code. Then, it contains a superblock, and two bitmaps. The bitmaps are used for inode and zone allocation, and can potentially span multiple blocks. Next, there are a few blocks containing space for inodes, potentially with more than one inode per block. Finally, the rest of the blocks contain user data.

The superblock is the second block on the disk, and contains information about the layout of the filesystem. In particular, it contains the number of inodes and zones on disk, the number of inode and zone bitmap blocks, the number of the first data zone, the base-2 logarithm of the number of blocks per zone, the maximum file size, and the magic number. The magic number

[1] `https://www.adacore.com/community`

[2] `https://github.com/libfuse/libfuse`

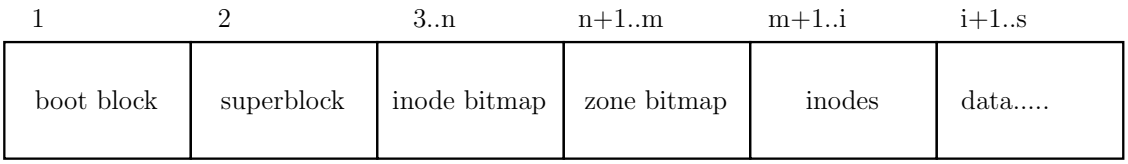
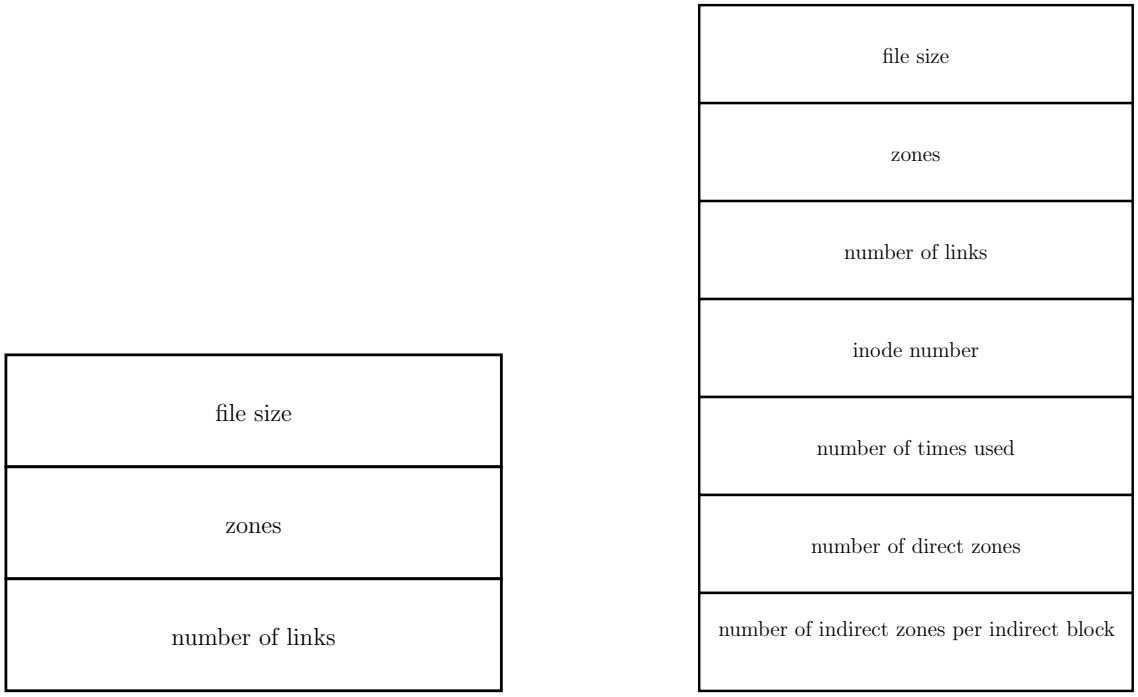


Figure 1: AdaFS disk layout. (n = number of inode bitmap blocks, m = number of zone bitmap blocks, i = number of inode blocks, s = number of blocks on disk)



(a) On disk *(b) In memory*

Figure 2: Inode representations

used to identify a correctly formatted disk is $CACA_{16}$; MINIX uses the magic number 2468_{16}, but AdaFS avoids using the same number because a disk formatted by the AdaFS *mkfs* utility is not necessarily equivalent to a disk formatted by the MINIX *mkfs* utility.

The two bitmaps keep track of available inodes and zones, where the nth bit of the inode or zone bitmap corresponds to the nth inode or zone on disk, respectively. If a bit in the inode bitmap is set to 1, that means the corresponding inode is allocated, and if it is set to 0, the inode is free. This is the same for the zone bitmap, pertaining to zones. One difference with MINIX is that, in MINIX, the first bit (bit zero) in the bitmaps must always be allocated, as the procedure that searches for a free inode or zone returns zero if no free inode/zone is found. In Ada, indexing generally starts at 1, so all bits can be used – as the first bit is bit one, this does not interfere with the allocation procedure.

The penultimate section of the disk contains inodes; the number of inodes is determined by the size of the disk. The inodes have two representations: on-disk (Figure 2a) and in-memory (Figure 2b). This allows the filesystem to make efficient use of disk space, while providing faster access to important values when the inode is loaded in memory. There are 10 zones per inode: 7 direct zones (those that contain data), 1 single indirect zone (indicates a block that contains more direct zones), 1 double indirect zone (indicates a block that contains more single indirect zones), and 1 unused zone. The unused zone is present for future use, potentially as a triple-indirect zone.

The final section spans the rest of the blocks on the disk, broken into zones, and is available to store user data. It can contain file data, or directory entries.

There are also some in-memory structures for working with open files. The filesystem keeps a process table in memory, which is indexed by process ID (PID), and keeps track of inode information for all processes using the filesystem. One entry corresponds to one process, and contains the inode numbers for the root and working directories of the process, as well as the list of open file descriptors. Each open file descriptor corresponds to an entry in the filesystem's *filp* table, which is shared among all processes and contains all the file position. The rationale for a shared file position table comes from MINIX, and is based on problems with the semantics of the *fork* system call [3]. An entry in the filp table contains the number of file descriptors using that entry, the inode number, and the file position for the inode.

Due to time constraints, a number of simplifications were made compared to MINIX. In particular, only the features necessary for basic functionality of the system were implemented. The filesystem does not cache any information, and there is no in-memory inode table; all data is written directly to disk. Furthermore, the current implementation does not keep track of file mode, owner, group, or timestamps. This results in a limited filesystem, which nonetheless completes its function as a proof-of-concept.

4.2 Language features & implementation specifics

Strong typing Ada is a strongly typed language, which helps the programmer distinguish between types that are logically different, even if their underlying representation is the same. Furthermore, the compiler will automatically catch any bugs that would be caused by assigning a value of an incorrect type to a variable. In order for a value to be assigned to a variable, two constraints must be satisfied: the value and variable must have the same type, and the value must satisfy all constraints on the variable (such as the range for an integral data type) [5]. Conversion between types is allowed, but only if it is explicit, and if the target type is an ancestor of the current type (for example, a positive integer may be converted to an integer, but not vice-versa).

Lack of pointers The C codebase of MINIX makes extensive use of pointers to refer to inodes and other structures in memory, addresses on disk, etc. Ada has access types, which are similar to pointers in some ways, but due to the complexity of verifying a program's behavior when it contains pointers, SPARK severly restricts the possibilities of using access types [2]. Thus, alternative methods must be used to provide similar functionality.

One way to solve this is with parameter modes. Ada allows specifying the mode of each parameter in a procedure, which designates how the parameter will be used in execution. If a parameter is of 'in' mode, it is only read in the procedure, and is not modified – this is the default mode. If a parameter is of 'out' mode, the value of the parameter before the call is irrelevant, as it will receive a value in the procedure. Finally, if a parameter is of 'in out' mode, it is both read and updated in the procedure. This third mode provides functionality similar to a pointer.

However, SPARK requires that functions be purely functional; that is, they cannot have parameters with a mode of 'out' or 'in out'. Here, two solutions are possible. In some cases, it may be preferred to add the return value as an 'out' parameter, and convert the function to a procedure. In other cases, it is better for the function to return multiple values, which is possible with a record type. An example is the function parse_next in Listing 1, which returns two values, wrapped in the record type parsed_res_t.

```
subtype cursor_t is Natural range path'Range;

type parsed_res_t is record
   next : adafs.name_t;
   new_cursor : cursor_t;
end record;
```

```
function parse_next (
  path : adafs.path_t;
  cursor : cursor_t) return parsed_res_t is
  ...
end parse_next;
```

Listing 1: Parse function returning the parsed component and the new cursor position

5 Results & analysis

- Include the formal verification report, what's verified in relation to CWE numbers.
- Add some performance analysis. Maybe timing? Try to do some common filesystem tasks & see how it performs?

6 Conclusion

Summary and concluding remarks, including possible future work.

References

[1] D. M. Ritchie, "The development of the c language," in *The Second ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL-II, Cambridge, Massachusetts, USA: Association for Computing Machinery, 1993, pp. 201–208, ISBN: 0897915704. DOI: 10.1145/154766.155580. [Online]. Available: https://doi.org/10.1145/154766.155580.

[2] AdaCore and Altran UK Ltd, *SPARK 2014 reference manual*, http://docs.adacore.com/live/wave/spark2014/html/spark2014_rm/index.html, Accessed on: 2020-06-26.

[3] A. Tanenbaum, *Operating systems: design and implementation*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1997, ISBN: 978-0136386773.

[4] *History of MINIX 3*, https://wiki.minix3.org/doku.php?id=www:documentation:readmore, Accessed on: 2020-06-26, 2014.

[5] J. Barnes, *Programming in Ada 2012*. Cambridge: Cambridge University Press, 2014, ISBN: 978-1-107-42481-4.