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 30, 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

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.

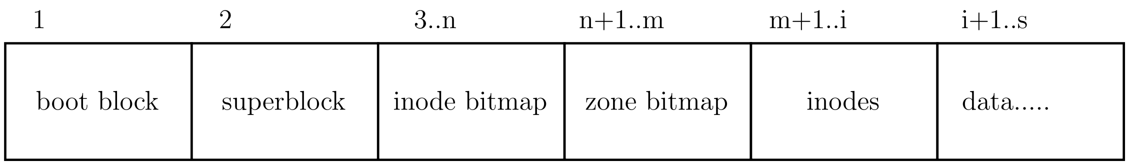
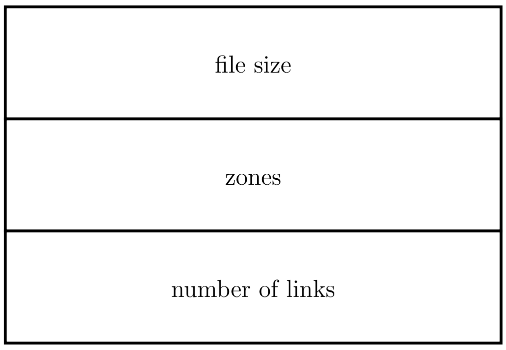


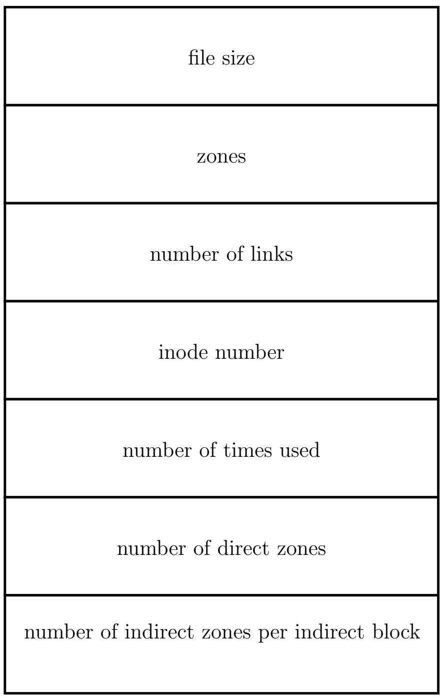
Figure 1: AdaFS disk layout. (n = number of inode bitmap blocks, m = number of zone bitmap blocks, i = number of inode blocks, s = number of blocks on disk)

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.

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



(a) On disk



(b) In memory

Figure 2: Inode representations

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 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). An example of the use of types is the definition of a character buffer of arbitrary size, followed by the definitions of different block types including a constrained version of the buffer type, shown in Listing 1.

```
type data_buf_t is array (Positive range <>) of Character;

type inode_block_t is array (1..block_size/on_disk'Size) of on_disk;
type zone_block_t is array (1..n_indirects_in_block) of Natural;
type dir_entry_block_t is array (1..block_size/direct'Size) of direct;
subtype data_block_t is data_buf_t (1..data_block_range'Last);
```

Listing 1: Block type definitions

Controlled types Controlled types allow the programmer to specify precisely what happens when a variable of a given type is declared, or when it goes out of scope. They are not permitted in SPARK, because the compiler inserts implicit calls to allow this functionality [2]. The initialization and destruction of variables of a controlled type is handled with custom initialize and finalize procedures for the type, thus the compiler needs to ensure these procedures are called in execution. However, in AdaFS, controlled types are not used for any filesystem logic, and are only used with disk I/O to make sure that e.g. the variable containing the superblock is initialized properly with the type of I/O the filesystem uses. Therefore, this was deemed acceptable for this prototype.

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 side effects, such as 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 rewrite the function as 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 2, 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 2: Parse function returning the parsed component and the new cursor position (ellipses denote code omitted for brevity)

Modularisation Ada supports modularisation in the form of packages, and forms a single translation unit, which can contain member entities such as subprograms, variables, and types. Information hiding is done by defining members as private. A package is separated into two parts, which are placed in separate files: the *specification* (the public interface for the package), and the *body* (the implementation). The compiler always checks whether the package body matches the specification, and refuses to compile the code if this is not the case. Listing 3 shows an excerpt of the specification and body of the adafs.inode package.

A package can also have child packages: for example, the adafs.inode package is a child of the adafs package. A child package has access to all member entities defined in the specification of its parent(s), including private members.

```
-- adafs-inode.ads
package adafs.inode
   with SPARK_Mode
is
   ...
   function calc_num_inodes_for_blocks (nblocks : Natural) return Natural
      with ...;
end package adafs.inode

-- adafs-inode.adb
package body adafs.inode
   with SPARK_Mode
is
```

```
  function calc_num_inodes_for_blocks
    (nblocks : Natural) return Natural
  is
    inode_max : constant := 65535;
    i : Natural := nblocks/3;
  begin
    ...
    return i;
  end calc_num_inodes_for_blocks;
end adafs.inode;
```

Listing 3: Excerpt from the adafs.inode package specification and body (ellipses denote code omitted for brevity)

It is also possible to create *generics*. Generics are somewhat similar to objects in the OOP paradigm, in that they can be instantiated with parameters. However, an important difference is that instantiation can only occur in a declarative region. Both subprograms and packages can be generic. For example, Listing 4 shows a generic function for reading data from a disk, and its instantiation. As Ada's Stream_IO requires specifying the type to be read, using a generic function helps with code reuse, with elem_t designating the type to be read, which is passed in at instantiation. The downside of generics is that they cannot be analyzed directly by SPARK, but must instead be verified from the context of instantiation (i.e., SPARK mode must be enabled in the package or subprogram that instantiates the generic) [2].

```
-- disk.ads
generic
  type elem_t is private;
function read_block (num : block_num) return elem_t;

-- disk.adb
function read_block (num : block_num) return elem_t is
  result : elem_t;
begin
  if not is_reading then
    sio.set_mode(stream_io_disk_ft, sio.in_file);
  end if;
  go_block (num);
  elem_t'read (stream_io_disk_acc, result);
  return result;
end read_block;
```

Listing 4: Generic function for reading a block of type elem_t

Interfacing with other languages Ada has a mechanism to allow interfacing with other programming languages, such as Fortran, COBOL, or C. This is done by replicating the types and subprogram signatures in Ada. The Interfaces library package [3] provides types and subprograms for this purpose. For example, for C, there are the Interfaces.C [4] and Interfaces.C.Strings [5] packages, which provide the types chars_ptr (mirroring **char**∗ in C), int (mirroring **int** in C), etc. This allows *exporting* subprograms from Ada, and calling them from a C program. Listing 5 shows

[3] `https://www.adaic.org/resources/add_content/standards/05aarm/html/AA-B-2.html`
[4] `http://www.ada-auth.org/standards/12rm/html/RM-B-3.html`
[5] `http://www.ada-auth.org/standards/12rm/html/RM-B-3-1.html`

a specification of a subprogram that is exported to C by specifying the Export and Convention aspects (as well as an external name to use when calling the subprogram from C). The subprogram is then declared as **extern** in C, and called from the C program's main function. Since the main program is written in a language different from Ada, the initialization and finalization procedures (adainit(**void**) and adafinal(**void**)) must also be declared and called before and after any other Ada subprograms, respectively. The GNAT Project Manager handles compiling and linking of files written in different languages. Unfortunately, interfacing code is not amenable to formal verification.

```
-- declarations.ads
with Interfaces.C;
package Declarations is
  function Factorial  (n : Interfaces.C.int) return Interfaces.C.int
    with Export => True,
         Convention => C,
         External_Name => "ada_factorial";
end Declarations;

-- main.c
#include <stdio.h>
extern void adainit (void);
extern void adafinal (void);
extern int ada_factorial(int n);

int main(int argc, const char *argv[]) {
  adainit();
  int n = 5;
  printf("%d\n", ada_factorial(n)); // 120
  adafinal();
}
```

Listing 5: Interfacing code written in C and Ada. declarations.adb *is omitted for brevity, but is assumed to contain an implementation of the factorial function conforming to the specification.*

FUSE & the FUSE driver FUSE is a software interface that allows running filesystem code in userspace, with FUSE bridging the gap between the filesystem and the kernel. This simplifies the development of filesystems, because access to the kernel and modification of kernel code is not necessary. FUSE allows a filesystem to be developed iteratively; i.e., first it can be implemented and tested with FUSE, and later connected to a kernel if needed.

To implement a filesystem with FUSE, the code needs to be linked with the FUSE library (libfuse), and a driver must be written to specify the filesystem's handler functions for various operations. As FUSE is written in C, the driver for AdaFS is currently also written in C, with a wrapper in Ada to convert values between C types and Ada types. FUSE specifies a **struct** with pointers to functions that should be written by the programmer for the specific filesystem they are implemented. The library also provides, among others, a function to fill file entries into a buffer, a **struct** to store open file information, and a function to get the context of the current operation (such as the PID requesting the operation). Listing 6 shows an example from the driver, with an implementation of the open file operation.

```
#define FUSE_USE_VERSION 31
```

```
#include <fuse.h>
// Declare the external filesystem open function written in Ada
extern int ada_open(const char *path, pid_t pid);

// The driver's open function
int adafs_open(const char *path, struct fuse_file_info *finfo) {
  pid_t pid = fuse_get_context()->pid;
  int fd = ada_open(path, pid);
  finfo->fh = fd;
  return 0;
}

// Register the function with FUSE
static struct fuse_operations adafs_ops = {
  .open = adafs_open
};

int main(int argc, char **argv) {
  ...
  return fuse_main(argc, argv, &adafs_ops, NULL);
}
```

Listing 6: FUSE driver implementation of open

Formal verification Unfortunately, much of the code in its current form is not amenable to formal verification. These are namely the parts involving file input and output, and functions that work with C types (i.e. the FUSE driver). However, large parts of filesystem logic *are* formally verifiable. Therefore, the code base was split into two distinct packages: the *adafs* package, which contains filesystem logic that is strictly in the SPARK language, and the *disk* package, which contains unverifiable elements such as disk I/O. The *adafs* package does not include any specifics about the type of disk being used, as the *disk* interface hides implementation details. Thus, when needed, and when an alternative is found, the *disk* package can simply be replaced with a verifiable implementation that exposes an identical interface.

For the parts that are formally verifiable, two types of contracts are available: functional and data contracts. Functional contracts describe how a subprogram should function; that is, the pre- and post-conditions for a given subprogram. They are written as boolean predicate logic expressions. Pre-conditions are evaluated before entry into the subprogram, and post-conditions are evaluated after exit from the subprogram (and can therefore mention the subprogram's result). SPARK can check these conditions at each call site to ensure that no subprogram call violates the conditions, and that the output(s) are shown to be conformant with the specification (the returned result for a function, and the **out** or **in out** parameters for a procedure).

The second type of contract available are data contracts. SPARK conducts flow analysis, which models the flow of information during a subprogram's execution. It checks for uninitialized variables, ineffective statements, and incorrect parameter modes. It is possible to specify which global variables are read, written, or both read and written in the subprogram, using the Global aspect. If no global variables are used, the value of the aspect is set to **null**. It is also possible to specify data dependencies between a subprogram's inputs and outputs.

For example, Listing 7 shows a function to get an entry from the process table, which specifies the functional and data contracts to be fulfilled for the function. The Global aspect specifies that the function only depends on the value of the package variable tab for input. The Depends aspect states that the result of the function only depends on the tab and pid variables (that is, the variable stated in the Global aspect, and the parameter of the function). The post-condition states that the is_null component of the returned variant record will be set to True if there is no entry in tab for the provided PID; otherwise, the inode number of the PIDs working directory will be non-zero. With the SPARK toolchain, we can verify that these constraints are all satisfied.

```
function get_entry (pid : tab_range) return entry_t with
  Global => (input => tab),
  Depends => (get_entry'Result => (tab, pid)),
  Post => (if tab(pid).is_null
           then get_entry'Result.is_null
           else get_entry'Result.workdir > 0);
```

Listing 7: Functional and data contracts

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:read-more, Accessed on: 2020-06-26, 2014.

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