An ext2 library

jan špaček

january 2016

1 Introduction

To begin with, here is the original goal of this project:

> It shall be a library to manipulate ext2 filesystems, maybe even a higher layer of abstraction corresponding to file system drivers in an OS (like VFS for Linux). The interesting part will be that the interface would be non-blocking using futures and promises (single-threaded) in the programming language Rust.

I will briefly describe the problem and the motivation, then I will explain how and why I failed, discuss the structure of the resulting piece of software, some technical issues and try to come up with a conclusion (and, indeed, a morale).

2 Motivation

File system is a format for saving sequences of bytes in a simple hierarchy, decorated with some metadata, inside a linear address space. These formats are understood by operating systems and abstracted away to provide uniform model of files.

Ext2 is a file system developed for Linux. Even though it has been mostly replaced by its newer revisions ext3 and ext4, it remains essentially backwards compatible (at least for reading). The main difference between ext2 and ext3 is that the latter supports journaling, which enhances performance and reliability by storing updates in a journal and replaying them later, while always keeping the device in a consistent state.

2.1 Non-blocking IO

Non-blocking IO (also referred to as asynchronous IO) means that a thread willing to perform an IO operation is not suspended until the operation finishes, but immediately continues executing. Later, when the operating system finishes the request, the thread is notified and can handle the results. This is usually accomplished by having the thread sit in an event-loop, waiting for events from the OS and dispatching them to registered handlers.

The other approach, blocking IO, is more usual, but has various downsides. First, while the process is waiting for an IO to happen, it is blocked and cannot get other things done. This works fine for processes that start by reading input, then get hot for a while and finish by writing output, but it would be disastrous for any (soft) real-time process that has to continuously respond to requests coming from the outside.

The only way to make things happen while your thread is blocked is to spawn another thread. To do n things at once, you need at least n threads. However, there are few tools widely available in modern programming environments as dangerous as a thread. It is nearly impossible for to properly coordinate so many threads by hand and avoid all race

conditions. Moreover, when the n things to be done are just waiting for events, we are paying the non-negligible cost of a thread only to make it sleep 99.9 % of its lifetime.

Event-driven programming avoids the thread synchronization hell, because there is only one thread running at a time, so no race conditions can occur while an event handler is executing. On the other hand, the sequencing constructs provided by the programming language can no longer be used to sequence IO. For example, when we want to load a list of funny pictures, blend them together and save them to a file using blocking IO, we can use the usual tools of our programming language:[1]

```
blend_pictures = \(files) {
  pictures = for file in files { read_picture(file) }
  blended = blend_pictures(pictures)
  write_picture(blended, "blended.jpg")
}
```

Note that the usual tools for composing computations worked well for composing IO operations. On the other hand, when we use callbacks to achieve the same effects with non-blocking IO, we end up with something like:[2]

```
blend_pictures = \(files, blend_callback) {
  load_pictures = \(files, pictures, load_callback) {
    if empty?(files) {
      load_callback(pictures)
    } else {
      read_picture(head(files), \(picture) {
        load_pictures(tail(files), cons(picture, pictures), load_callback)
      })
    }
  }

  load_pictures(reverse(files), \(pictures) {
    blended = blend_pictures(pictures)
    write_picture(blended, "blended.jpg", blend_callback)
  }
}
```

It is instructive to see that when imperative constructs fail, the functional come to the rescue.[3] Indeed, help comes from the language that has no imperative constructs – Haskell – in the form of `IO` monad. When we use a monad only to capture the results and allow side-effects, we end up with futures:

```
blend_pictures = \(files) {
  future_map(files, \(file) {
    read_picture(file)
  }).then(\(pictures) {
    blended = blend_pictures(pictures)
    write_picture(blended, "blended.jpg")
  })
}
```

A future represents a computation whose result will be known in the future. Futures can be composed to produce other futures using various combinators (like `future_map` or `.then` in our example). They allow us to use non-blocking IO with little cognitive overhead compared to blocking IO.

[1] In the code, I use \ to say λ, because LaTeXis, well, complicated.

[2] Of course, this works only if we have a garbage collector, so that the values captured in the closures do not get destroyed too early.

[3] Another instance of a similar phenomenon is so-called Visitor desing pattern, needed to encode variants in object-oriented languages – it is exactly the way how variants are encoded in pure λ-calculus. For me, using such a convoluted way to encode something so obvious feels like I needed to write `[->+<]` to move a value between variables :-)

I should also mention that there is another approach to doing many things at once, which avoids the need to distinguish pure and IO-based computations, showcased in languages like Go or Erlang. The first ingredient is to make threads cheap by using a lightweight system of green threads (fibers, goroutines, ...) on top of OS threads, managed by the language's runtime. The second ingredient is to provide better tools to avoid race conditions. For example, Erlang forces all values to be immutable, so there are no data races. The threads in Erlang communicate and share data using messages, which are more natural and usually easier to get right than complex interactions of mutexes and condition variables. Go does not restrict mutability, but it also provides a message-passing primitives for inter-thread[4] communication.

2.2 Rust

Rust [1] is a systems programming language developed at Mozilla Research. It guarantees memory safety without garbage collector by statically checking ownership of values and lifetimes of references. Compared to C++, probably its closest competitor, it is much more modern, featuring type inference, traits (similar to type classes in Haskell), rich data types and pattern matching, to name a few.

Rust runtime used to support green threads, but the language evolved towards smaller runtime and they were removed. However, the standard library provides access to OS threads and implements various synchronization primitives, including message passing. The strong type system can for example guarantee that values can pass thread boundaries safely, so that no data races can occur, or that a value protected by a mutex will not be accessed incorrectly.

2.3 Personal motivation

In summer 2015, I was working for a startup named Bileto on a public transport routing engine. The engine was written in C++, because the speed of the algorithm was crucial. However, the service needed to communicate with many other servers over the network, so that the size of the code that managed the communication in fact exceeded the size of the algorithms that did the actual work. This IO code was originally using blocking IO and many threads, but I rewrote it to non-blocking using `libev` and a home-made library of futures and promises.

I knew and used Rust before for my personal projects, mainly for a compiler that I developed for my maturita exam[5], so I naturally was quite interested in how Rust could be used to solve similar problems.[6]

3 High-level overview

Of course, my plans failed miserably. Indeed, I could not even find a Rust library that would allow me to read from a file asynchronously and I should soon find that the design principles that worked for C++ could not be directly translated to Rust. Designing data structures and control flow is much more demanding when the type system must be assured that everything is OK. When I think about it now, I should have probably tried harder, but I quickly decided to throw in the towel and implement the library first synchronously, and maybe rewrite it later, armed with the experience gained on the road. This rewrite is now unlikely, so I will present the synchronous version from now on.

[4] The threads in Go are called goroutines, and spawned by keyword `go`.

[5] `https://github.com/honzasp/spiral`

[6] And, of course, this library was intended to be a part of my great plan to write an OS. Thanks to Martin Mareš for opening my eyes :-)

3.1 File system structure

An ext2 filesystem [3] begins with a superblock that stores the crucial information about the filesystem concerning the layout and the details of the format. The disk is divided into blocks of 2^{10+i} bytes for some $i \geq 0$. The whole volume is split into equally-sized block groups. Every block group contains a part of the inode table and a bitmap that is used to mark blocks used in the group. There is also a bitmap that marks whether each inode from the block group is used or not. To enhance data locality, the data of an inode is primarily allocated into the block group that stores the inode, and inodes of directory entries are first placed into the block group of the directory.

The first 12 blocks of data are referenced directly from the inode. Then there is a reference to an indirect block, that contains an array of references to next blocks. Then there is a doubly indirect block, that references another indirect blocks, and finally a trebly indirect block referencing doubly indirect blocks.

The directories are just special files that contain a linked list of entries. There are various restrictions on this list (for example, no entry can cross block boundaries, the entries can only point forward, or every entry must start on a four-byte boundary), but the order or the items is not specified, so directory lookup is an $O(n)$ process. There is also an extension of the file system that allows the directories to be stored more efficiently using a hash map, but I did not implement it.

To make testing the file system easier, I created a simple FUSE wrapper that allowed me to mount the filesystem and work with it using regular Linux APIs. However, I did not create any tests.[7] Honestly, I would not dare to access any physical hard drive containing valuable data with my library.

3.2 Code organization

I tried various ways of organizing and subdividing the library. First, I tried to organize the work into several layers, inspired by [2]: the lowest layer would deal with the raw bytes, the next layer would manage inodes and inode data blocks and the top layer would provide high-level access to files and directories. However, the boundaries between the layers were not very clear and I wanted to avoid over-abstraction, so I abandoned this model.

Then I attempted to organize the code around the objects. The procedures dealing with inodes would become methods of the `Inode` object, the procedures for working with groups would be associated with `Group` and so on. This would work with a little problem: 90 % of the code deals with inodes, so I ended up with one huge module defining the inode with small modules orbiting around it.

In the end I stuck to the model used in C or in Lisps – most of the procedures are standalone functions defined in modules that are loosely grouped around the actions and data structures. There is a module `fs` that contains functions that mount and umount the filesystem, modules `encode` and `decode` for translating between internal and external representations of various data structures, `inode` for generic operations on inodes, `inode_data` for reading and writing inode data and so on. Most `struct`s are defined in module `defs`.

Every module in Rust automatically defines a namespace, but I considered it too painful to prefix every function with the name of the module that it is defined in, so I created a module named `prelude` that reexports (or, as Rust says, `pub`licly `use`s) all names from all modules. However, only `pub`lic names are available outside of modules. Also, the structure of modules is not visible to users, because the library exports only a carefully selected list of functions (so not every function that is `pub` is visible to the user). The fields of structs are also public, so they can be accessed outside of the modules they are defined in. Unfortunately, that means that the fields are also visible to the user. The solution

[7] Mainly because the cost of errors in projects like this is so low that testing does not pay for itself.

would be to define them with private fields in the root module (submodules in Rust can see private items); I will maybe do this later.

4 Low-level details

The central structure is the `Filesystem`. It stores a copy of the superblock, copies of all group descriptors and a cache of inodes. When the superblock or any of the group descriptors need to be updated, they are first modified only in memory and marked as dirty. Later, the changes are written to disk. This saves us a lot of IO, because the superblock and block descriptors are changed quite often (they store the count of free block or free inodes, for example).

Inodes are usually accessed repeatedly, so they are also cached. However, we cannot afford to store all inodes in memory, so the size of the cache is limited.[8] The algorithm that decides which inode will be removed from the cache and which will stay alive is simple: all inodes are placed in a queue in the order they are read to memory. When there are too many inodes in memory, the inode from the other side of the queue is removed. To keep frequently used inodes in memory, an inode is marked as reused when it is read or written. When the inode from the front of the queue is marked as reused, it will be enqueued again and not removed, but the mark will not be preserved. The cache is write-back so, the modifications are kept in the cache and written to disk when the inode is removed from the cache.

The user always references inodes by their number.[9] Most of the code passes around references to `Inode`s and does not directly read or write the inode from the filesystem. However, care must be taken to keep the `Inode`s in circulation and inodes on disk (or in cache) in sync. If a piece of code modified an `Inode` and another piece of code read an outdated piece of information from another copy `Inode`, both referring to the same physical inode, trouble would happen.

The in-memory structures are not in the same binary format as the structures on disk, so they need to be encoded and decoded manually. This brings a small performance penalty and a small decrease in memory consumption (we do not decode all the fields and include no padding for future extension), but the main reason is that this approach is safe, because we do not hazard with the memory guarantees of the language. It is also convenient, mostly because we can use `enum`s, and portable, because the data on disk is always stored in little-endian. There is a caveat, however: because we ignore some of the bytes when decoding inodes and superblocks, we must first read the corresponding byte range, overwrite the bytes we care about, and write back. This preserves the possibly important fields that are currently ignored. The superblock is so important that the verbatim copy of its bytes is constantly kept in memory.

To avoid any troubles with overflowing numbers, exclusively 64-bit numbers are used throughout the code. This is not strictly necessary, because most quantities can be only 32-bit (inode numbers, block numbers, ...), but it saves us a lot of explicit casting and mitigates the risk of mistakes. Rust also performs overflow checking at runtime in debug builds, increasing the safety.

Error handling is managed via standard Rust approach, the `Result` type and `try!` macro. `Result` is a simple algebraic data type with two variants representing either success or an error (like `Either` in Haskell). Explicitly checking every return value would be painful,[10] while using the monadic combinators (`.map`, `.then`) is sometimes convenient, but often it is not. However, one simple macro can change the situation. `try!(expr)` evaluates

[8] I need to confess that the size of the cache is given as a literal in the function that clears the cache, not as a properly named constant or a piece of configuration.

[9] In the code, I used a convention that `ino` means inode number, while `inode` refers to the `Inode` structure.

[10] This is the preferred way of handling errors in Go. This is also one of the reasons why I do not really like Go.

the `expr`, which is of type `Result<T, E>`[11]. If it was a success, the value of the macro is simply the associated value. On the other hand, if the result was an error, the macro returns from the current function with the same error. This exploits the fact that most of the time, errors should simply be passed up the call stack, but the possible sources of errors are still explicitly marked. In effect, we gain the comfort of language-level exceptions without their drawbacks – error handling is explicit, requires no language support and creates no corner cases (compare with C++). We can also use the monadic combinators when they are appropriate.

In fact, Rust also has a form of implicit exceptions, called thread panics. Code usually panics when there is a contract violation, for example access out of array bounds, division by zero or arithmetic overflow (in debug builds). These panics cause stack unwinding that causes destructors to run, but cannot be caught, so the whole thread is terminated. This policy forces fault isolation and tries to avoid most of the problems caused by inconsistent data after an exception (remember that a thread cannot share data with other threads).

The implementation assumes that the volume is correctly formatted and there are inputs (disk images) that can cause the library to crash. Real-world implementation should definitely be more hardened.

5 Morale

Now comes the most important part – what have we (and I) learned from this funny little piece of software?

One of the lessons we can learn is that the more restrictive a programming language is, the harder it is to write programs in it. This is one facet of the trade-off between static guarantees and flexibility. The trade-off manifests mostly in typing and the most obvious dividing line is whether the language is dynamically or statically typed. The defining feature of Rust is its memory safety without any runtime overhead, but we pay for the performance by having to think about lifetimes and memory management when designing data structures and control flow. For me, the first obstacles I encountered when trying to implement asynchronous operation were so high that I had to give up the major goal of the project. Now, I regret I did not try harder, but it is true that the language definitely stood in the way.[12]

Of course, there is the other side of the aforementioned trade-off. While it took some time to persuade the compiler to accept my code, nearly all of the errors it produced were genuine mistakes that I have made. Using a language with less static guarantees, I would have to find the mistakes by trial-and-error and probably would have to be serious with tests. This way, my code usually worked for the first time and there were relatively few cases when I had to hunt bugs. The greatest help was `e2fsck`, a tool to check and repair ext2 filesystems. I was also pleasantly surprised that `lldb`) worked very well with Rust, in fact it worked better than with (GCC-compiled) C++. And, of course, I did not encounter a single segfault or any memory safety violation.

[11] `T` is the success, `E` is the error – this is different from Haskell.

[12] In fact, when I learned I would not get a library able to asynchronously read a file without writing the library myself, I gave up almost immediately.

Bibliography

[1] *The rust programming language*. `https://www.rust-lang.org`.

[2] Abelson, Harold and Gerald Jay Sussman: *Structure and Interpretation of Computer Programs*. MIT Press, second edition edition.

[3] Poirier, Dave: *The second extended file system: Internal layout*. Technical report, 2011. `http://www.nongnu.org/ext2-doc/ext2.html`.