CPL: Rust

Project Assignment: Developing a File System

Version 1.3

Contents

1 Introduction

In this project, you will be developing a (simple model of) a Unix-like file system.

This project has been inspired by the file system in the xv6 [1] teaching operation system and the implementation of various user-space file systems using FUSE [2], chief among which a Rust implementation called Gotenks [3]. If you want to know more about the basics of file systems or are unsure about some aspects of file systems mentioned in the assignment, and Google is not providing you with digestible information, the text parts of xv6's chapter on file systems is a good place to learn more about the basics [4].

Rather than implementing an entire user-space file system and allowing you to mount it, however, **we implemented the file system's backend** (i.e. a model of the disk controller and some operations on disk sectors) for you already, and we will ask you to build the different layers of abstraction of a classical file-system on top of the provided code. The consecutive assignments in section 3 will explain how to do this in more detail.

[1] `https://pdos.csail.mit.edu/6.828/2020/xv6.html`

[2] `https://www.kernel.org/doc/html/latest/filesystems/fuse.html`

[3] `https://reposhub.com/rust/filesystem/carlosgaldino-gotenksfs.html`

[4] Chapter 8 in `https://pdos.csail.mit.edu/6.828/2020/xv6/book-riscv-rev1.pdf`

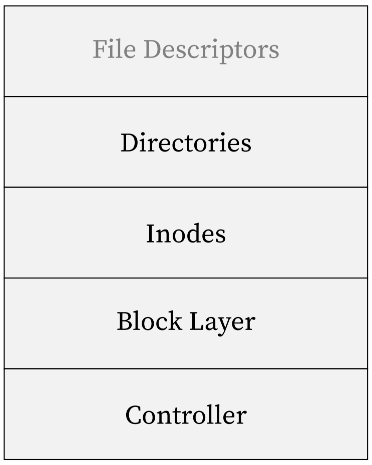

Figure 1: Representation of all abstraction layers in this assignment's file system model. We will not explicitly model file descriptors in the assignment.

More concretely, the different layers of abstraction that we will model in this assignment are shown in figure 1. We will now go over the different layers, and simultaneously refer to figure 2. This figure provides a more concrete overview of the disk layout that we will assume in this project.

The layers are as follows:

Controller The backend layer we provided for you, represented by the `Device` type in the code. Represents a model of a device controller for a storage disk and performs accesses to the raw disk data for you. As is usually the case with storage devices, accesses to the raw data can only be performed at block-level granularity, i.e. you have to read and write data in entire blocks. The file system is represented by a single open file in the code. The controller will memory-map this file during execution, and read and write the required data blocks from and to this file. At the end of its lifetime, the controller flushes any outstanding reads and writes to its backing file, to make sure they are persisted on your device. The controller layer then also contains support to reload an existing disk image, by providing it with a path in your own file system. To keep matters as simple as possible, we do **not** explicitly model a driver for the disk, and you can communicate with this controller layer directly to perform the necessary block memory accesses. In other words, we do not make a distinction between the device's block size, which is the unit for controller read-write operations, and the driver's block size, which is the unit for driver read-write operations.

To simplify reading and writing disk blocks, represented by the `Block` type in the code, we provided you with some methods to easily serialize Rust structures into disk blocks, and deserialize them from blocks. Blocks also allow reading and writing regular sequences of bytes. These methods allow for more seamless interaction with the controller layer.

Figure 2 represents the contents of a `Device` object. Although they are not explicitly shown, each of the four marked regions in this figure consists of a number of disk

`Blocks`.

Block Layer Layer that builds a **block** abstraction on top of the raw controller layer. First of all, the block layer builds the concept of **data blocks** on top of the controller. Data blocks are a specific type of blocks that make up the contents of files. They are stored on the disk, in a region called the **data block region**, shown in figure 2. Call `nbdatablocks` the total number of data blocks in the file system. An entire disk block is then provided on the disk for each data block, from block 0 to block `nbdatablocks` − 1.

It is possible that some empty space is left at the end of the data block region, if it is unnecessarily large. This is the case for most other regions as well, as figure 2 shows. Additionally, empty space within regions can be caused by the fact that the disk image is made up of blocks, and objects will not be stored across block boundaries. This second type of block-alignment related empty space is not shown on the figure.

To keep track of which blocks in the data block region have been allocated and which ones are still free, the disk contains a second region called the **bitmap region**, which contains a **single bit** for each data block, again from 0 to `nbdatablocks` − 1. If the bit corresponding to a certain data block is set to 1, then this block is currently in use. If the bit is set to 0, then the block is free and can be allocated for various purposes (e.g. when some file increases in size and requires more data block).

Second, we use a so-called **superblock** to keep track of all the metadata of the file system. The block layer has provisions to read and write this superblock. That being said, the superblock is sufficiently important and frequently accessed to warrant keeping a read-out version cached in the kernel (in our case, as we will not model an entire operating system; caching happens in the block layer). The superblock is stored in the very first block of our device image, so that the file system knows where to find it at boot time. It is the only useful data stored in this first block. Note that assuming the superblock to be the first block does not exactly correspond to real-world disk devices, as these usually have boot blocks and/or device partitions, which cause the file system superblock to be pushed further back on the disk.

Inodes Builds the well-known Unix-concept of **inodes** on top of the block layer. Inodes correspond to our mental image of a 'file' that is stored on the disk. They have an inode number to identify them, and point to a sequence of data blocks belonging to the current inode. Inodes can grown and shrink as required, under the effect of different system calls (e.g. read, write, seek, ...). Inodes can have different types; they can correspond to files, directories or be free (i.e. currently not in use). Inodes are stored in a section of the disk called the **inode region**. Since inodes are decently small compared to disk blocks, we do not create a separate bitmap region to keep track of which inodes are (un)allocated. Rather, we can find out whether an inode is in use by reading it from the disk and checking whether its type is 'free'. Call `ninodes` the total number of inodes in the file system. Part of a disk block is then provided on the disk for each inode, from inode 0 to inode `ninodes` − 1. Multiple inodes are usually stored on a single block, leaving some empty space at the end if the inode size does not divide the block size.

Directories Builds the notion of directories into the file system. Directories are just another type of inode, that contain a sequence of **directory entries** inside of their data blocks. Directory entries consist of a name (think of the string names you see when opening a directory in your file explorer) and an inode number, pointing to the inode that represents the contents of this directory entry. The top-level directory entry, called the **root directory**, has inode number 1 and will always be present in our directory structure. Figure 2 shows off the root inode, and the always empty inode 0.

Clearly the directory system allows for nesting of directories, and introduces the notion of **paths** (think e.g. `/home/kt/abc.sql`) into the file system. All (absolute) paths start at the root directory, which has path '/'. This is denoted by the '/'-character at the start of each (absolute) path.

File Descriptors **We will not model this layer in this assignment. The following explanation is just here for completeness' sake, and to help you see the link to user space.** Creates **file descriptors** on top of inodes. File descriptors are what you use when performing system calls in Unix-like operating systems, to tell the kernel which file you want to perform a certain operation (duplicate, unlink, read, write, ...) on. A file descriptor points to an inode, so that operations know where the contents of the file are located. Additionally, file descriptors keep track of a certain **offset**; an offset into the inode content at which the next read or write should commence. Three very well-known file descriptors in Unix shell programs are the standard input, standard output and standard error

Each process in an operating system has a look-up table of file descriptors, which are used to translate the integers found in system calls such as the `fd` argument in this UNIX read system call:

```
ssize_t read(int fd, void *buf, size_t count);
```

to file descriptor objects in the kernel.

2 Limitations of the model

While we model a lot of aspects relating to file systems in a Unix-like fashion, this assignment makes several simplifications that prevent this simple file system model from being used in a real-world operating system. In case you know something about file systems, it is good to clearly state these simplifications up front, to avoid confusion. The simplifications we make are as follows:

- We do not **cache** disk blocks that were read by the controller, i.e. we do not implement a disk block cache in the block layer. Since disk reads and writes are far slower than memory reads and writes, this is something that any realistic file system should do.

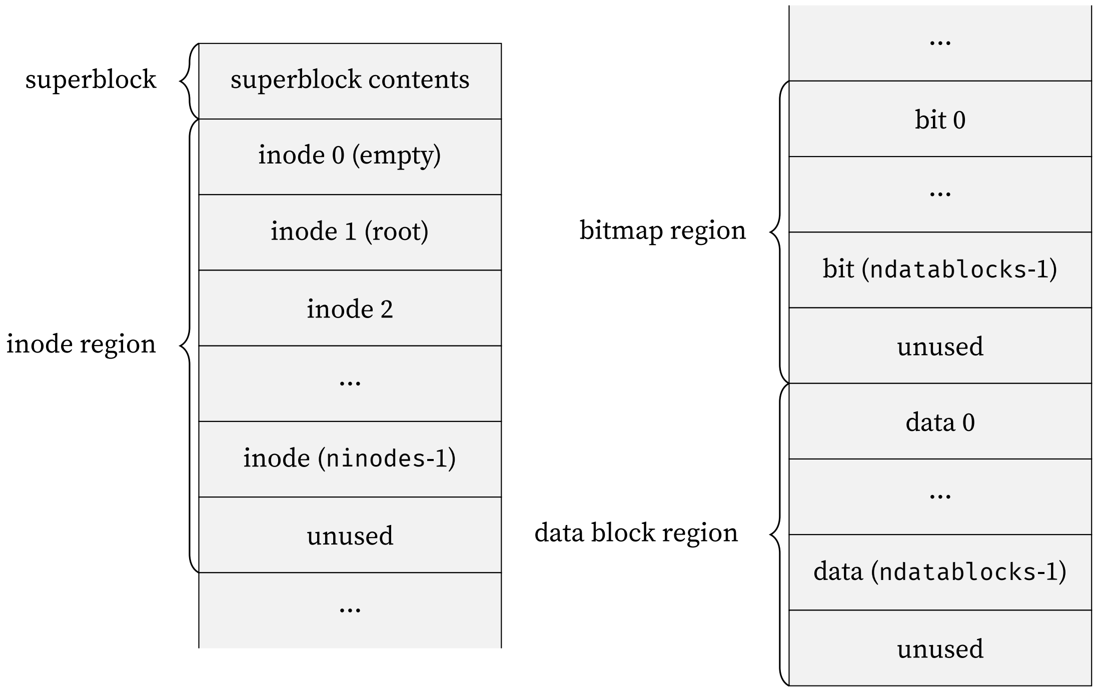

Figure 2: Layout of our file system on disk. Disk block boundaries and lost space at the end of blocks not shown.

- We do not **cache** inodes that were read from the device either. This is less critical from a performance point of view, but more important from a usability standpoint; any program that uses in-memory (i.e. not the on-disk version) representations of inodes and mutates different inodes at different times will have to make sure that it used the most up to date versions, since there is no cache that keeps the data for all in-memory inodes in a centralized fashion. The reason for both this caching simplification and the previous one, is that it is non-trivial to set up caching in an efficient way in Rust. To make sure different parts of a program or different programs can all read and write a cached version of an inode, it seems like we need to go against Rust's type system (which remembers, disallows having aliasing and mutation at the same time). If you are interested in how we can implement this type of caching in a safe manner, you should make sure to implement optional assignment g, where we will lift this second restriction.
- Often times, file systems have some notion of **transactions**, implemented by a **logging layer** abstraction. The concept is very similar to the notion of transactions in a database setting. This layer makes sure that when we execute a system call in the kernel, that requires writing multiple blocks on the disk, then this system call is either persisted in its entirety, or rolled back as if no part of it had ever occurred. The goal of this layer is to avoid having the file system in an inconsistent state at boot-time, after a hardware malfunction or kernel panic has occurred during the previous execution.
- The API's of our different layers will not guarantee correct operation in the presence of parallelism, since they do not make use of locking or some other form of **synchronization primitive**. Even on a time-shared uniprocessor, this can lead to

issues due to context switching. In other words, we assume that our API's will only ever be called by a sequence of non-overlapping, individual processes. This massively simplifies the code, and reasoning about it, since parallelism is hard.

- Usually, file systems make use of a **file descriptor** abstraction, to have a user-space addressable notion of file. Each process then keeps track of a data structure containing all file descriptors that are currently in use, and how they map to inodes on the disk. In addition to the information contained in an inode, a file descriptor also keeps track of a current index into a file, used for e.g. the `read` and `write` function calls. Since we only ever assume that one process at a time accesses our API, modeling a file descriptor layer is not as interesting. For example, the `fork` and `exec` system calls are far less interesting in this setting. For this reason (and reasons of tractability), we do not model a file descriptor layer in this assignment.

3 Assignment

We expect you to spend up to **40 hours** on this assignment. The project consists of a number of mandatory assignments and a number of optional assignments. You are not required to complete the optional assignments, that's what **optional** means.

When you complete all mandatory assignments correctly, and write readable, documented, and well-structured code, you will get at least a **passing grade**. The more optional assignments you complete, the higher your grade will be. Keep in mind that when you only hand in the mandatory assignments, and your code doesn't work correctly or is a complete (undocumented) mess, you will not get a passing grade. Therefore, we recommend that you implement at least one optional assignment. You will still lose points for handing in a complete (undocumented) mess, and we strongly advise against it. Additionally, we advise you to write your own tests in addition to those provided with the assignment, to make sure your implementation has got its bases covered.

In each assignment you will write a different file system, often building further on the previous one. We provide a **template file** for every assignment in the `solution/src` folder, **please use the right file for the right assignment**. At the top of each module, there will be some TODOs that you have to complete: indicate whether you completed the assignment, etc. You are free to add additional functions, traits, data type, modules, etc. You will often need them. However, you are not allowed to modify or rename the given ones. More concretely, you should **not change anything in the `api` folder**, or make any changes to the structure of the template files (i.e. do not remove the `FSName` type and do not change how the tests in the `api` folder are called).

The files we provided you should be sufficiently documented to be more or less self-contained. The documentation of the assignment can be built on your own machine, see the Hints & Tricks section. For this reason, the assignments below contain little more information than some pointers to the file you can get started in. Before discussing where each individual assignment is located, we first discuss how to get started on this project from a very abstract point of view.

3.1 How to Get Started

1. As this project consists of multiple modules, you won't be able to complete it using the online Rust Playground. You must **install Rust** on your computer by following the instructions at `https://www.rust-lang.org/downloads.html`.

2. **Unzip the project** and you will see the directory structure of Figure 3. We organised this crate (the name for a Rust project) in two subcrates: `cplfs_api` (in the `api` folder) and `cplfs_sol` (in the `solution` folder).

   The `cplfs_api` crate contains all code that is provided by us. You are not allowed to edit this crate whatsoever. It consists of the following elements:

   - some basic types and utility functions on them, in `types.rs`.
   - the implementation of the controller layer (in `controller.rs`, making use of the error type defined in `error_given.rs`)
   - the traits that provide the API's for the different layers of the file system, in `fs.rs`. These are the traits you will have to implement in the individual assignments below.
   - the tests we provided you with, in the `fs-tests` folder. More information on this follows in section 3.4.
   - the `lib.rs` file; see below.

   The `cplfs_sol` crate is where you will develop your own file systems. We have provided a template for each assignment. Additionally, you should not forget to fill in the file `FILLME.md` after completing the project, to provide some explanation about your self-written tests, potential noteworthy extras in your assignment, and feedback for future CPL projects.

   The `lib.rs` files are the main modules of each crate. The `Cargo.toml` files are the crate config files that define each project and its dependencies. As we will discuss in Hints & Tricks, you may depend on additional crates. You can add these dependencies by modifying the `solution/Cargo.toml` file.

3. **Compile the project** using Rust's package manager, Cargo, by running `cargo build` in the `solution` folder. Cargo is installed together with Rust. For more information about Cargo, see its tutorial: `http://doc.crates.io/guide.html`.

4. **Build the documentation** by following the instructions in the last bullet of the Hints & Tricks section.

5. **Read the documentation** of the `cplfs_sol` crate and/or **look at the source code** of `solution/src/lib.rs`.

6. **Read the first subsection** of the next section 3.2, so that you have an idea what we expect of you in the first assignment.

7. **Read the documentation** of the `a_block_support` module and **look at the source code** of `solution/src/a_block_support.rs`. This file should be sufficiently well-

```
cplfs
├── api
│   ├── Cargo.toml
│   ├── fs-tests
│   │   ├── a_test.rs
│   │   ├── b_test.rs
│   │   ├── c_test.rs
│   │   ├── d_test.rs
│   │   ├── e_test.rs
│   │   ├── f_test.rs
│   │   ├── g_test.rs
│   │   └── utils.rs
│   └── src
│       ├── controller.rs
│       ├── error_given.rs
│       ├── fs.rs
│       ├── lib.rs
│       └── types.rs
├── solution
│   ├── Cargo.toml
│   ├── FILLME.md
│   └── src
│       ├── a_block_support.rs
│       ├── b_inode_support.rs
│       ├── c_dirs_support.rs
│       ├── d_path_support.rs
│       ├── e_inode_RW_support.rs
│       ├── f_indirect_inodes.rs
│       ├── g_caching_inodes.rs
│       └── lib.rs
└── Cargo.toml
```

Figure 3: Directory structure of the project

commented to get you started, and provides pointers to the necessary traits that you should implement.

3.2 Mandatory Assignments

These are the assignments you are required to complete. Each title also states the applicable template file. This file contains (pointers to) all the necessary documentation you need, as well as some fields you need to fill in.

A: Block Layer

→ `a_block_support.rs`

Implement the block layer abstraction in figure 1. You will also need to implement methods to create and load file systems, which you will need to continuously update in the following assignments.

B: Inodes

→ `b_inode_support.rs`

Implement the inode layer from figure 1 on top of the block layer you previously implemented.

C: Directories

→ `c_dirs_support.rs`

Implement the directory layer from figure 1 on top of the inode layer you previously implemented.

3.3 Optional Assignments

You can choose which optional assignments you complete and in which order you do so. The only exception to this rule is that assignments `f` and `g` have a dependency on assignment `e`. Remember that we advise to at least complete one optional assignment. Assignment `e` should take reasonably little time to complete. My personal favorite is assignment g, as it attempts to take a peek beyond the very rigid ownership and borrowing discipline that the compiler had us enforce so far.

D: Paths

→ `d_path_support.rs`

Use the directories you previously implemented to support file system paths and a notion of current working directory.

E: Inode Read and Write

→ `e_inode_RW_support.rs`

Build on top of the inode layer you implemented before, and implement read and write methods for inodes.

F: Indirect Data Block

→ `f_indirect_inodes.rs`

Reimplement inodes, but now with a notion of an indirect data block. This allows inodes to become a lot larger than before.

G: Inode Caching

$\rightarrow$ `g_caching_inodes.rs`

Reimplement inodes. Your new implementation should support inode caching, and allow different locations in the code to keep a (possible mutable) reference to a cached entry at the same time.

3.4 Tests

Since this assignment is relatively low-level and contains a lot of nitty-gritty details, testing is very important to gain confidence in the correctness of your code. During grading we will be using a number of tests to verify whether your file system implementations correctly implement the traits we provided. A portion of these tests have been provided by us in the assignment code.

To run your own tests, you can navigate to the `solution` folder in the assignment, and run the command:

```
cargo test
```

This command will run all of your own tests. We **strongly advise you to write your own tests**. To help you get started, the source file of the first assignment (i.e. `a_block_support.rs`) will contain more detailed comments explaining how to do this, and further details how we set up our own tests as well. Additionally, it also provides 2 dummy tests that illustrate how to set up tests. For more inspiration on how to define tests, also make sure to take a look at the provided test files.

To make sure that our tests do not interfere with yours and that you do not get errors when you have not completed each individual assignment, we have hacked them into Rust's **feature** system. Each individual assignment (a through g) corresponds to a feature of the same name (a through g). When activated, each feature will run all tests in the test file `<assignment_letter>_test.rs` in the folder `api/fs-tests` (cfr. figure 3).

To run all of your own tests **and** all of our tests corresponding to feature **X**, run the following command:

```
cargo test --features="X"
```

Multiple features can also be combined in this way. For example, if you wish to run all of your tests, and all of our tests corresponding to assignments `a`, `b` and `e`, you could use the following command:

```
cargo test --features="a b e"
```

If you happen to implement each optional assignment or do not really care about receiving a lot of errors next to your implemented tests, you can run all of our tests at the same time by using the `all` feature;

```
cargo test --features="all"
```

Note that since I did not use a proper testing framework to implement the tests, failing tests might leave behind a few artifact directories, possibly containing disk images, in the `solution` folder. They do get cleaned up at the start of the next round of tests, though, so you should not worry about these too much. Just remove all of them once, as soon as your implementation does not cause tests to fail anymore, and they will not reappear again, since they get cleaned up at the end of each successful test.

4 Practical Matters

4.1 Deadline

The deadline for this project is **Sunday 2020-12-20 23h59**. There will be a Toledo assignment to submit your project.

Submit your project as a zip file of the `solution` folder, as this is the only folder you are allowed to edit. Don't forget to **complete the TODOs** at the top of each module, and **complete the `FILLME.md` file** in the `solution` folder.

Make sure your project **compiles without errors**, otherwise we will subtract points. Also, try to **fix all warnings** (and not by asking the compiler to stop generating them), they often hint at problems in your code. We also advise you to format your code as discussed in Hints & Tricks.

4.2 Plagiarism

The code that you turn in must have been written by you, and by you only. You are welcome to discuss your work with other students and with teaching assistants, but you are not allowed to copy or share pieces of code with others. You are not allowed to make your code available to others, and you are not allowed to use code made available by others.

4.3 Forum

If you have a question about the project or found a bug in the code or tests we gave you, post it on the **discussion forum on Toledo**. This forum will be actively monitored by the assistant.

This forum is not a help desk for Rust basics, questions like: "What does `&self` mean?" will not be answered, you should know how to use Google by now. Keep the rules about plagiarism in mind, so don't share code, use abstract examples.

4.4 Rust Version

We expect you to use a stable version of Rust (1.47 is the latest stable version at the time of writing). When a new stable version of Rust is released, 1.48 or even 1.49, you are allowed, but not required to switch to it. You typically don't have to worry about things no longer working with newer versions of Rust.

While browsing the Rust APIs, you will probably encounter certain methods and features that are **unstable**. These things are not available in a stable version of Rust, and require a **nightly** build.

5 Hints & Tricks

- **Documentation**: there are three important sources of information about Rust:
  - **The Rust Book (2018 edition)**: `https://doc.rust-lang.org/book/foreword.html`
    For when you can't remember what Ownership is or if you don't know how Rust does Error Handling.
  - **Rust by Example**: `https://doc.rust-lang.org/stable/rust-by-example/`
    For when you want to know what the syntax of a **struct** is or how to define **methods**.
  - **The Rust API documentation**: `http://doc.rust-lang.org/std/`
    For when you want to know what methods are available for a `Vec` or what the interface is of `Iterator`.
- **Extra dependencies**: you are free to add additional dependencies from `https://crates.io` to your project. It is certainly possible to complete all assignments without any additional dependencies, but some crates might provide useful utilities (for example the `itertools` crate) or additional data structures. The Cargo (Rust's package manager) guide[5] clearly explains how to add dependencies to your project.
- **Extra modules**: you are free to add additional modules. Moreover, you are encouraged to do so whenever a file becomes too large. The chapter on Crates and Modules in the Rust Book explains how to add modules.
- **Formatting**: use the `rustfmt`[6] tool to automatically format your code. You can simply invoke `cargo fmt --all` in the `solution/src` folder and all your code will be formatted. One less thing to worry about.
- **Generate documentation**: you can generate the documentation of both our code and your project using `rustdoc`. This can be done by running `cargo doc --open` in the root `cplfs` folder. The documentation will be generated and opened in the web browser. If you just want to generate the documentation for your solution, run this command in the `solution` directory instead. Documentation is written in the Markdown format.

[5]`https://doc.rust-lang.org/cargo/guide/dependencies.html#adding-a-dependency`
[6]`https://github.com/rust-lang-nursery/rustfmt`