|  |  |
| --- | --- |
|  | Project 5 - FUSEs |
|  |  |
|  | Edward T, Ryan M  Cps 360  Spring 2024 |

Contents

[Setup 2](#_Toc162577785)

[Platform 2](#_Toc162577786)

[Required Software 2](#_Toc162577787)

[Build 2](#_Toc162577788)

[Repo Business 3](#_Toc162577789)

[LARD 3](#_Toc162577790)

[Description 3](#_Toc162577791)

[Axioms 3](#_Toc162577792)

[Format 3](#_Toc162577793)

[Format Explanation 5](#_Toc162577794)

[The superblock 5](#_Toc162577795)

[The I-List 5](#_Toc162577796)

[The I-Map 5](#_Toc162577797)

[The D-Pool 6](#_Toc162577798)

[readLardfs.py 6](#_Toc162577799)

[LARDIMAGE 6](#_Toc162577800)

[getIMaps 6](#_Toc162577801)

[getdPools 7](#_Toc162577802)

[IListEntry 7](#_Toc162577803)

# Setup

## Platform

We’ve tried this on WSL to see if we could get it to work without a VM for Windows users, but that was very broken. We also tried using a Debian 11 VM using VMWare, but that also didn’t work. In all fairness, that might have been a corrupted VM image. At the end of the day, we both ended up using Linux Mint 21.

## Required Software

* Linux, FreeBSD, NetBSD or MacOS X system – We only did Linux, so can’t vouch for the process on any of the others.
* Python – We were using 3.10.12 at the time of writing.
* setuptools – Python module
* contextlib2 – Python module
* pkg-config
* attr
* C compiler
* llfuse

## Build

Make sure to run the update upgrade tools to get the newest versions of software:

sudo apt update

sudo apt upgrade

To install Python on Linux Mint run the following:

sudo apt install software-properties-common

sudo add-apt-repository ppa:deadsnakes/ppa

sudo apt install python3.10

Upon installing python, we can move on to grabbing the rest of the libraries for the project. Next we need python’s package manager, pip, as well as a few linux packages:

sudo apt install pip

sudo apt install pkg-config

sudo apt install attr

After that, we can install some python packages for running llfuse, as well as the dev headers:

pip install setuptools

pip install contextlib2

sudo apt install libfuse-dev

And then, finally, we can run the magic command:

pip install llfuse

## Repo Business

Navigate to the directory in which you want to store the repository and run the following:

git clone https://github.com/Sintfoap/cps360\_fuse\_project5\_spring2024.git

Now would probably be a good time to remind you to have git installed on your linux machine.

Simply run python3 mklardfs.py to create a lardfs.img and you’re good to go!

# LARD

## Description

The LARD filesystem (lardfs) combines elements of the classic Unix filesystem (i-list/i-nodes, mode-bits, file types, names as links in directory files, unix timestamps and link count) with one from the ubiquitous FAT filesystem (linked-list sector allocation). LARD is not recommended for production, nor for direct human consumption. (It provides excellent flavor in scrambled eggs, though.)

## Axioms

* The filesystem “root” is always i-node 0.
* All numeric fields are in big endian.
* Signed integers are 2’s complement.

## Format

* superblock:
  + 8-byte ASCII magic string: "LARDFS\n\0"
  + *(all numeric fields BIG ENDIAN)*
  + 4-byte: sector size (in bytes)
  + 4-byte: image size (in sectors)
  + 4-byte: start-of-i-list (in sectors)
  + 4-byte: start-of-i-map (in sectors)
  + 4-byte: start-of-d-pool (in sectors)
* i-list: (an array of i-nodes indexed by i-number)
  + 2-byte: mode bits [0xttttsssuuugggooo]
    - MSB
    - type (4 bits):
      * 0b0011: symlink
      * 0b0010: directory
      * 0b0001: regular file
      * 0b0000: UNUSED/FREE i-node
    - setuid/setgid/sticky (su/sg/t) [3 bits]
    - user [owner] R/W/X [3 bits]
    - group R/W/X [3 bits]
    - other R/W/X [3 bits]
    - LSB
  + 2-byte: link count (0-65535)
  + 4-byte: owner-uid
  + 4-byte: owner-gid
  + 4-byte unix-timestamp: c-time (metadata-change-time)
  + 4-byte unix-timestamp: m-time (data change time)
  + 4-byte unix-timestamp: a-time (data access time)
  + 4-byte: size in bytes
  + 4-byte: first-dsector (index into imap/dpool)
* i-map: (an array of 4-byte dsector indices)
  + one per sector in the dpool
  + serves as a linked-list-style "next sector" pointer, a la FAT
  + an inode's start field gives the *first* sector of a file
  + we then look up that sector's entry in the i-map to see what the *next* sector in the file is
  + reserved values:
    - -1 (0xffffffff): unallocated
    - -2: EOF (no more sectors in chain)
* d-pool: (an array of sectors available for use)
  + one per index in the imap
* directory file structure: (array of 32-byte dir-entries)
  + 4-byte i-node
  + 28-byte ASCII char name/path-segment (e.g., "etc")

## Format Explanation

### The superblock

Just like images or elf files or what have you, this has a magic string to tell the computer what kind of file it is. We then unpack the sector size which we’re going to immediately turn around and use in relation to the next 4 fields. In understanding these fields, we only need look at the last three, image size is self-explanatory. We can treat the other three as pointers, in essence, to the start of arrays containing the data they’re named after. I’ll discuss what’s exactly in those and what they mean in relation to each other in the following sections.

### The I-List

This is what you’re going to be doing the most work in. The I-List is a list of i-nodes and their metadata. To start off, we’ve got a nice confusing 2-byte field that contains 5 pieces of information, Yay! I find it useful to look at what this would be represented as in byte format:

0bttttsssuuugggooo

The 0b just notates that this is in binary format, so no need to worry about those characters. The next four t’s represent the file type bits, and as noted above there are four different file types we’re worried about right now. The next three bits are the setuid/setgid/sticky bits, then we have the user bits, the group bits, and finally the other bits. This makes up the permissions section of files. We then have the link count, which notates how many references there are to this i-node, such as its own . reference or the .. reference in its child directory. We then have the owner uid and gid, which we don’t have to worry about very much to start with. Then all the timestamps, which will be important for information display. And then the size of bytes, which is very important for parsing, and the first-dsector. This part was super confusing when I was first working on it, this is the first index into both the i-map array and the dpool array. A bit on that next.

### The I-Map

Ok so the I-Map is basically a linked list of pointers to sectors in the dpool that we’ve allocated for our file. It’s by file, so each I-node has its own start into the I-map/dpool structures. For instance, we might have an i-node that has a first-dsector of 0, and another that has a first-dsector of 4. The ubiquitous equation for getting the relevant pointer is filePointer = startImap \* sectorSize + 4 \* dsectorPointer (dsectorPointer can be first-dsector or any resulting dsectors as you will see in a minute). After we’ve found the first dsector via first-dsector, that will be one of three values. The first is -1 (0xfffffff), which signifies an unallocated sector. We shouldn’t, and I mean error if you do, hit one of these when traversing an existing file image. The second value is -2 (0xfffffffe), which signifies EOF. When you hit this you can stop traversing the file image for the inode you were on. The final value you can get is any positive value. When you get that, that becomes your new dsectorPointer, meaning you can follow that offset into the I-Map array and dpool array to get the next sector. So given the example of the i-node that starts with a dsector of 0, we’d go to sectorSize \* startIMap + 4 \* (0) and look at the value there. Supposing it’s 1, we would then look at secotrSize \* startIMap + 4 \* (1) to get the next dsector. We keep doing that until we hit -2.

### The D-Pool

The d-pool holds two kinds of information. Files or directories, as denotated by the inode you’re searching for data for. For directory file types, you’re going to go to the first dsector section you found from the I-Map, and then you’re going to get the 4 bytes for the inode and the 28 bytes for the name. This was a bit confusing to begin with so I’ll do my best to explain. The inode designates which iNode holds the data that goes under the corresponding name. So if I’m reading inode 0 and I find an entry that has an inode number of 0 and a name of “.”, then the “.” Refers to inode 0, which is me. If I find an inode number 1 that corresponds to the name “etc”, I’ll have to go look for the data under etc to see what it is. The other type of data is file data, which is just the data that goes in the file. So if inode 0 finds a d-pool entry that points at inode 2 called motd, and inode 2 is a file type inode, then the data we pull up will be the text inside of motd. Now I hear you asking, “how do we know how much data to read from the d-pool for each inode?” and that’s an excellent question. That’s where the size on each inode we pulled in the i-list step comes in handy. For each directory inode, the size attribute will be divisible by 32, and that corresponds to the number of directory entries we should read in. For file types, it can be any arbitrary length, meaning that we just read that many bytes into the file. Now the tricky bit is when your size for an inode is larger than the sectorSize of your file image. In that case your info is going to be split up across sectors and that’s where the linked list nature of the imap comes in handy. Just traverse the imap to the next sector location and continue reading in data at that location in the dpool.

# readLardfs.py

To start out, we first made a python program capable of reading in and outputting the file tree of a LARD file system. Something to be noted here is we used the struct library for python a ton to read in binary fields from the lardfs image. It just made processing a lot easier and faster. The next few sections will be dedicated to how exactly we went about that.

## LARDIMAGE

The class lardimage contains the entirety of the file system image, and handles all the interactions between the inodes and the data in the file. This starts by reading in the superblock, which is pretty simple compared to some of the stuff we do later. We then get the iListEntries, where we read iListEntry objects into the iList attribute until we hit an entry that has a null bit in place of its mode. WHICH COME TO THINK OF IT THAT’S NOT RIGHT. REMINDER TO ME TO FIX THAT. I’M HIGHLIGHTING IT SO I DON’T FORGET. After that we read in the imap and store all the resulting sector locations in the iListEntry objects, and the same with dpool and entries.

### getIMaps

The process by which we retrieve I-Maps is a bit fun, so I’ll take some time to cover it. The basic premise is we want to read through the iList and grab all of the sector locations to store locally in each iListEntry object. In order to do so we record the size of the inode that we’re currently on in our iteration and check it against how much we’ve read in. We do this because we want to make sure that we’ve read enough entries to cover the data we will attempt to read in later. We then calculate where we’re reading from and read 4 bytes in to look for the next entry. If it’s negative 2, Great! Just check to make sure we have enough sectors to cover our size and we’re done! If it’s negative 1, someone done messed up! Report error and move on. If neither of those cases happened, then we have the next location in our list and we continue through our while loop.

### getdPools

So this one’s a bit more complex than getIMaps just because it has another option that’s pendant on a different size. Basic premise is that we want to read in all the data for either directories or files from a given inode. If it’s directories, then we’re always reading a 32 byte chunk till we reach the end of our size. If it’s files then we’re reading data in sectorSize chunks till we reach the end of our size. Pointer math is a bit more fun, and size management is more complex but other than that it’s pretty straightforward.

# IListEntry

Holds all the metadata for i-lists. The two fields to really watch out for are dsecPointers and entries. dsecPointers holds all the pointers to our available, allocated sectors in the dpool which we populate during getIMap. The entries are the fEntrys or dEntrys that we generate during the getdPool process. We do some fun bitwise calculations for separating the 3 bit and 4 bit fields in mode and user permissions from the modeBits, but other than that there’s not much to it.