There are two provided example file systems found in sub-directories of the initial project repository. One (`empty/`) contains only a root directory. The other (`fs`) contains a few files and directories as demonstrated by the commands below. I recommend that you copy one of these and test with the copy while developing to avoid corrupting the provided files. You may also restore these directories from the git repository. Or, the sample file system directories may be found in this repository: https://github.com/amigler/357-a2-starter

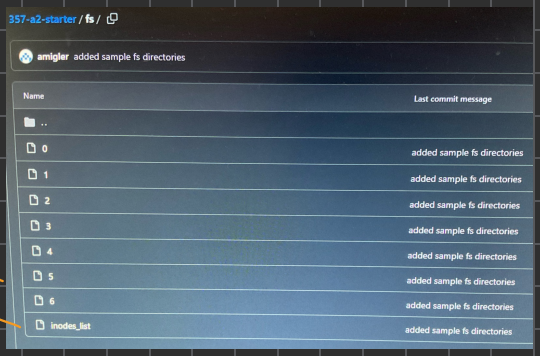
```
> ls
0 .
0 ..
1 357
> cd 357
> ls
1 .
0 ..
2 assignment2
3 final
6 assignment3.tex
> cd final
> ls
3 .
1 ..
4 final.tex
5 code.c
>
```

simulated fs

user input

inodes 0, ... n

- contents



inodes_list

- [n index]
- [t type: 'd' or 'f']

Of note, the metadata for an inode does not contain the name of the file. Instead, a directory maps names to inodes. But a directory is also a "file". This is what we will simulate.

Some Specifics

1024, some #define constant

Your program will store a fixed number of "inodes" (which is not unreasonable when simulating a physical device). The metadata for these "inodes", stored in a file named inodes_list will contain only a number (an index) and a "type" to indicate a directory ('d') or a file ('f'). The contents of the "file" associated with each "inode" are to be stored in a regular file with name matching the index number (i.e., if your program is meant to access the contents of the "file" associated with "inode" 3721, then it will open the file named "3721"; this is a simplification of tracking blocks on a physical medium).

Limit: Your program must support up to 1024 inodes (ranging from 0 to 1023). This is an entirely artificial limit and we will work to remove such limits as the quarter progresses.

The content of an example `inodes_list` file is listed below. Note that this is a binary file, not ASCII-text as we have been working with thus far, and therefore cannot be viewed or edited with a text editor such as `vim`. The example below uses the `xxd` utility to display binary data, using `-c 5` to show each record in the file on its own line (each inode record is 5 bytes: a 4-byte integer followed by a single ASCII character: d or f) In the `xxd` output, the first column (before the `:` character) represents the hexadecimal byte offset within the file. The middle columns show the binary file data in hexadecimal. The final output column displays ASCII characters where possible, or the period character for non-ASCII bytes.

```
amigler@csc$ xxd -c 5 fs/inodes_list
00000000: 0000 0000 64  ....d
00000005: 0100 0000 64  ....d
0000000a: 0200 0000 66  ....f
0000000f: 0300 0000 64  ....d
00000014: 0400 0000 66  ....f
00000019: 0500 0000 66  ....f
0000001e: 0600 0000 66  ....f
```

xxd -c 5

the '5' just shows 5 chars then separates by new-lines

n - "name"?

<4-byte int> <1-byte char 'd' or 'f'>

inodes_list format for one instance

- inodes_list contains each inode in this format:

<4b int><1b char>< . . . >< - - - -

'd'

0 1 2 3 4 5th 10th

0 byte offset

5th byte offset

inode 0 | inode 1 . . .

Directory Representation

The contents of a directory will be a sequence of file entries. Each file entry will specify, in this order, the inode number for the entry (use the `uint32_t` type defined in `stdint.h` to represent an unsigned 32-bit integer) and the name of the entry, up to 32 characters. File names less than 32 characters will include a null character within the first 32 characters (but a full 32 charactes must be stored in the file to keep each entry the same size); it is valid to use the full 32 characters for the file name, so your program must account for the lack of a null character in that case. The entries within a directory are stored as binary representations of each piece of data.

The following is an example of a directory corresponding to inode number 3, containing the files: `.`, `..`, `final.tex` and `code.c`

```
amigler@csc$ xxd -c 36 fs/3
00000000: 0300 0000 2e00 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000  ....................................
00000024: 0100 0000 2e2e 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000  ....................................
00000048: 0400 0000 6669 6e61 6c2e 7465 7800 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000  ....final.tex.......................
0000006c: 0500 0000 636f 6465 2e63 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000  ....code.c..........................
```

inode 3, which is a directory

inode number ↑

0 1 2 3 4 5 6 7 8 9 1 1 2 3 4 5 6 7 8 9 2 1 2 3 4 5 6 7 8 9 3 1

name, 32 chars

english

2e = '.'

2e2e = '..'

* all in hex, so reference ASCII chart hex

|f|i|n|a|l|.|t|e|x|

```
00000048: 0400 0000 6669 6e61 6c2e 7465 7800 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000  ....final.tex.......................
0000006c: 0500 0000 636f 6465 2e63 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000  ....code.c..........................
```

|c|o|d|e|.|c|

- 4 chars for inode # | 32 chars for english "name" of entries all stored as binary representations (displayable thru xxd -c 36)

Initial Directory

Your program will begin in the (assumed) directory at inode 0 (this is meant to be the root of the file system, typically referenced as /). Your program will then take commands from the user to traverse and manipulate the file system (see below).

or above

File Representation

The inode file corresponding to a plain file (type f) in your simulated filesystem should contain the full file name stored as ASCII text, followed by a single newline character. The following is an example of the content of inode file `4`, representing a file with name `file.tex`:

```
amigler@csc$ xxd -c 36 4
00000000: 6669 6e61 6c2e 7465 780a                                final.tex.
```

inode 4

|fi|n|a|l|.|t|e|x|\0

- the inode for dirs should fill up all 32 chars for the name, but the singular inode file should just display however many it is

Plan

- main.c
  - arg parsing & errors — ./fs_simulator <dir> — handle accordingly
  - read inodes_list, variable for "current dir"?, test prints root inode & as default
  - user input for fs_sim → [user input]
    - if-else or switch-case statements for different 'fs' commands
- make separate functions for all commands
  - mkdir & touch look similar in functionality, maybe combine
  - which command first?
    1. ls — displaying what we're seeing is great for debugging
       - also, it's just print statements so even better for debugging
    2. cd <dir> — changing into a dir feels like reading & loading the inode associated w/ it, so reusability is there
       - cd <> → ls → cd <> → ls ...
       - DBug
       - seems like a good way to test & debug
    3. mkdir <dir> or touch <file> — could do either, but mkdir seems like it can flow into the debugging easier:
       - mkdir <> → ls → cd <...>
       - DBug
    4. exit — we only mess w/ inodes_list (like saving stuff into inodes_list) after we mkdir or touch

This sequence makes sense

type

inode - index
- type 'd' or f

some array holds 1024 inodes

inode inodes_arr [1024];

MAX_INODES

0 - 1023

directory - index
- name [32 + 1] null terminator

file? - name [up to 32+1]

might not be necessary

current directory