P3 - File Systems

COP4600

Slides by: Ernesto A. Mujica | Revision by: Thurstan Ngo

File Paths

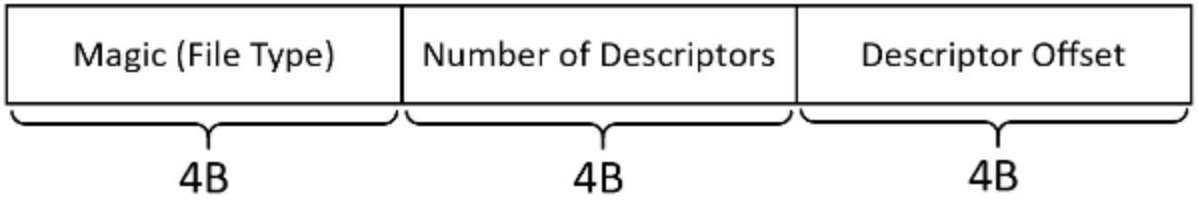
- Most of your library functions will take in a std::string representing a path to a file in the file system. Paths can represent either regular content files or directories.
- Paths will always begin with a “/” character, representing the root directory, in your Reptilian VM. When you ssh into your Reptilian VM, your terminal spawns in /home/reptilian.
- In this project, paths to content files can look like:
  - /E1M0/01.txt
- While paths to directories can look like either:
  - /F/F1
  - /F/F1/
- The extra “/” at the end of a path to a directory is valid and should be considered.

The WAD File Format - Overview

- In this project, you will be provided with WAD files, and you will need to read in and modify their contents. Your library will contain the bulk of your code, and the file system daemon will be a relatively short program that leverages the library you create.
- There are 3 main sections to a WAD file, in order:
  - The **header** contains information about where to begin reading the descriptor list.
  - The **lump data** contains the actual content for given file(s).
  - The **descriptor list** contains the relevant information about directory structure, file names, file sizes, and file content locations.

The WAD File Format - Header

- The header will always be exactly 12 bytes:
  - 4 bytes for the file magic, a 4 character ASCII string
  - 4 byte numerical value for the number of descriptors in the descriptor list
  - 4 byte numerical value for the start location of the descriptor list in the WAD file
- Each of these 3 values you take in should be stored as class member variables.
- The file magic will not change, but the number of descriptors and descriptor offset may change once directories and files are added.
  - If any of these values change, you will need to update the member variables as well as writing the new changes into the WAD file.

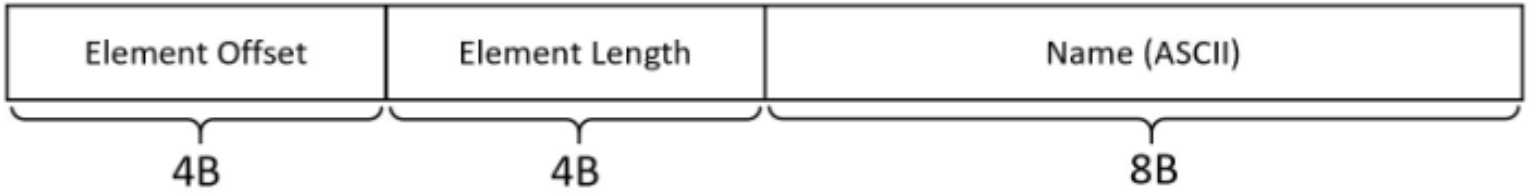


The WAD File Format - Lump Data

- The lump data is located in the middle of the WAD file, and will vary in size and length.
  - Contains the contents of files.
- A given file's descriptor will provide the location to that file's lump data as well as how many bytes starting at that location belong to that file.
- The lump data should not be read into memory until it is necessary.
- You can safely ignore all the lump data in **loadWad** until accessing the data is necessary in **writeToFile** and **getContents**.

The WAD File Format - Descriptor List

- A list of several descriptors describing files and directories. This list can be thought of as the “File Control Block” for the WAD file system.
- Each descriptor will always be 16 bytes total:
  - 4 bytes for the ‘Element Offset’, which is the location of the lump data for this file.
  - 4 bytes for the ‘Element Length’, which is the size of the file.
  - 8 bytes for the name of the file, including file extension. This means the max length a file name can be is 8 characters.
    - For ex: **file.txt** is valid, but **files.txt** is not, since it won’t fit into 8 bytes.



Directory Marker Types - Map Markers

- In this project, there will be 2 types of directories you will be working with, denoted by marker elements in the descriptor list.
- Type 1: Map Directories
  - Map directories will always have a name in the “E#M#” format where “#” is any number from 0 to 9.
  - The next 10 descriptors following a map marker will always be content files whose parent directory is the map marker.
  - Content files will never have the “E#M#” name format to prevent confusion.

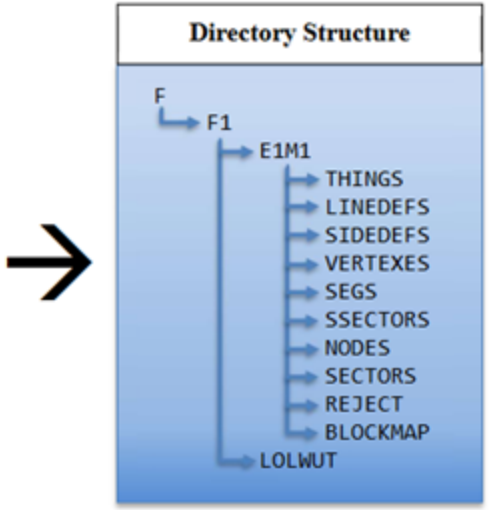
Directory Marker Types - Namespace Markers

- Type 2: Namespace Directories
  - The beginning of a namespace directory is denoted by a descriptor whose name has suffix “_START”.
  - Since “_START” is 6 characters, and we can only fit up to 8 characters in a descriptor file name, the maximum length a namespace directory can be named is 2 characters.
    - Ex: “F1_START” or “F_START”
  - The end of a namespace directory is denoted by a descriptor whose name has suffix “_END”.
  - All descriptors between the “_START” and “_END” of a namespace directory are treated as children of that directory.
  - These are the kind of directories you will be creating in **createDirectory**.

Example Directory Structure

- The first hurdle to overcome in this project is how to organize your file system in memory.
- You must traverse the flattened tree that is your descriptor list and convert it into an *actual* tree.
- This requires a depth-first search algorithm that can keep track of directory nesting (usually a stack, though recursion also works).
- You should be storing this information in an N-ary tree of some kind of struct, where each struct contains the file name, offset, length, and a way to store other files if a given descriptor is a directory.
- You are also regularly provided strings that represent file paths, and you need to use those paths to look up information about a file/directory.
- Sounds like a good candidate for a key-value system…

Offset	Length	Name
0	0	F_START
0	0	F1_START
67500	0	E1M1
67500	1380	THINGS
68880	6650	LINEDEFS
75532	19440	SIDEDEFS
94972	1868	VERTEXES
96840	8784	SEGS
105624	948	SSECTORS
106572	6608	NODES
113180	2210	SECTORS
115392	904	REJECT
116296	6922	BLOCKMAP
42	9001	LOLWUT
0	0	F1_END
0	0	F_END



```
public static Wad* loadWad(const string &path)
```

- Invokes the constructor by creating a Wad object with new, then returns the pointer to it.

```
private Wad(const string &path)
```

- Private constructor, takes in path to a WAD file from your **real** file system.
- Reads the header data and initializes your data structure(s) to represent the WAD file elements from the descriptor list. No need to read in lump data at this point.

```
public bool isContent(const string &path)
```

- Takes in path to a file from your WAD file system.
- Will return true if it is a valid path to an existing content file.
- Will return false if it is a valid path to a directory, or if the path is invalid (nonexistent).

```
public bool isDirectory(const string &path)
```

- Similar to above, but will return true for valid directories, and false for content files/nonexistent paths.

```
public int getSize(const string &path)
```

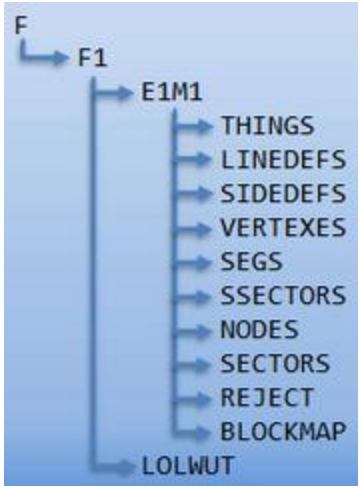
- Returns the size of the file at path. If path points to a directory or is invalid, returns -1.

```
public int getContents(const string &path, char *buffer, int length, int offset = 0)
```

- Given a valid path to an existing content file, reads length number of bytes from the file's lump data, starting at offset. Returns number of bytes successfully copied into buffer. Returns -1 if path is directory/invalid.
- Example: Let's say we have a valid path to a text file, "/F/F2/file.txt".
  Let's say this file has a size of 5 bytes, and contains data "hello".
- Case 1: length = 5, offset blank (defaults to 0).
  All 5 bytes, "hello", gets copied into buffer. Return 5.
- Case 2: length = 4, offset blank.
  4 bytes, "hell", gets copied into buffer. Return 4.
- Case 3: length = 4, offset = 1.
  4 bytes, starting at offset 1, "ello", gets copied into buffer. Return 4.
- Case 4: length = 6, offset blank.
  Length exceeds size of file, so we only copy **what we can**.
  We copy 5 bytes, "hello", into buffer. Return 5.
- Case 5: length = 8, offset = 2
  Length exceeds size of file, and we must read from offset 2.
  We can only copy 3 bytes, "llo", into buffer. Return 3.
- Case 6: length = 5, offset = 6
  Offset goes beyond end of file, so we cannot copy any bytes.
  Not an error, but no bytes are copied into buffer. Return 0.

```
public int getDirectory(const string &path, vector<string> *directory)
```

- Takes in path to a directory, and pushes back the names of all the directory's children into the passed in vector.
- Returns the number of children names copied into the vector of strings.
- Example 1: On the image on the right, directory "/F/F1" has two children: E1M1 and LOLWUT.
    - You would add those names into the vector and return 2.
- Example 2: Path = "/", the root directory. The root directory has 1 child, "F".
    - Add "F" to the vector and return 1.
- Example 3: Path = "/F/F1/E1M1/". E1M1 has 10 children.
    - Add the names of all 10 children to the vector and return 10.



`public void createDirectory(const string &path)`

- Takes in path to a directory that does not yet exist and must be created. It will be a namespace directory, meaning you will need to add a start and end descriptor, and the file name must be at most 2 characters.
- New directories can only be created in namespace directories. Map directories cannot have files or directories added to them.
- First, separate out the file name from the path, and ensure that the path before the file name is a valid, existing directory.
    - Ex: “/F/F1/F2”. “F2” is the name of the new directory to be created. “/F/F1” is its parent directory.
- The two new descriptors must be added to the very end of the parent directory, before the parent directory’s “_END” descriptor.
- You will have to create 32 bytes worth of space (16 bytes for each new descriptor) by shifting the rest of the descriptor list forward, so that you can fit the two new descriptors into the list.
- Remember to update relevant data structure(s), member variable(s), and the header of the WAD file!
- Example: **createDirectory(“/F/F1/F2”)** will result in the new descriptor list on the right.

Offset	Length	Name
0	0	F_START
0	0	F1_START
67500	0	E1M1
67500	1380	THINGS
68880	6650	LINEDEFS
75532	19440	SIDEDEFS
94972	1868	VERTEXES
96840	8784	SEGS
105624	948	SSECTORS
106572	6608	NODES
113180	2210	SECTORS
115392	904	REJECT
116296	6922	BLOCKMAP
42	9001	LOLWUT
0	0	F2_START
0	0	F2_END
0	0	F1_END
0	0	F_END

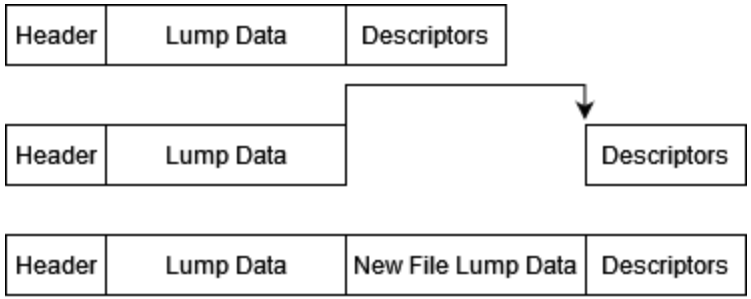
`public void createFile(const string &path)`

- Takes in path to a file that does not yet exist and must be created.
- Remember that new files can only be created in namespace directories.
- Ex: “/F/F1/F2/file.txt”. “file.txt” is the name of the new file to be created. “/F/F1/F2/” is its parent directory.
- Take caution to ensure that the file name does not contain illegal sequences for files, like “_START/_END” or “E#M#”.
- The offset and length of the file can be set to 0 initially before it has been written to.
- Example:
  createDirectory(“/F/F1/F2”);
  createFile(“F/F1/F2/file.txt”);
  - The above two calls will result in the descriptor list on the right.
- Remember to update relevant data structure(s), member variable(s), and the header of the WAD file!

Offset	Length	Name
0	0	F_START
0	0	F1_START
67500	0	E1M1
67500	1380	THINGS
68880	6650	LINEDEFS
75532	19440	SIDEDEFS
94972	1868	VERTEXES
96840	8784	SEGS
105624	948	SSECTORS
106572	6608	NODES
113180	2210	SECTORS
115392	904	REJECT
116296	6922	BLOCKMAP
42	9001	LOLWUT
0	0	F2_START
0	0	file.txt
0	0	F2_END
0	0	F1_END
0	0	F_END

```
public int writeToFile(const string &path, const char *buffer, int length, int offset = 0)
```

- Takes in path to an existing, empty file. Returns -1 if path is a directory or invalid. Returns 0 if non-empty file.
- Reads length number of bytes from buffer and writes them into the lump data of the file starting at offset.
- You will need to create space in the lump data section in the middle of the WAD file to be able to write the file contents.
- The way to do so would be to propagate the entire descriptor list forward by length bytes, creating a space for you to write your new file contents.



- Remember to update the file's length and offset in both your data structure(s) and in the file's descriptor as well as updating the header of the WAD file to represent the new position of the descriptor list.

Sidenotes for Library…

- In both **createFile** and **createDirectory**, you will need to search through the descriptor list to find the place to add the new descriptors.
- Separating out a path “/F/F1/F2/file.txt” into tokens like “F”, “F1”, and “F2” may be useful to you as well as a helper function that searches the descriptor list.
- Another possible helper function to consider would be one that can propagate the WAD file forward by a certain number of bytes, since you will have to create space in the descriptor list for the descriptors in **createFile** and **createDirectory**, and you will need to create space in the lump data before the descriptor list in **writeToFile**.

FUSE Daemon

- The FUSE daemon is a secondary program you will be writing for this project that will leverage your library to be able to mount your WAD file and traverse it as if it were a real place in your file system!
- This will be significantly shorter than your library, probably around 80-200 lines where your library might be 600+ lines. It is recommended that you **heavily** rely on the **linked resources** in the PDF since those will be able to guide you on a lot of the FUSE specific syntax.
- FUSE functions via the use of ‘callback’ functions, which replace normal file system syscalls with the functions you implement.
- You will be writing 6 FUSE callback functions, excluding main:
  get_attr, **mknod**, **mkdir**, **read**, **write**, and **readdir**.
- Each function has a use case covered by library functions you have already written, meaning that you should be able to leverage your library extensively here to do all the heavy lifting for you. Generally, you will notice they have very similar parameters to the library functions as well.

So… how do I start?

- Start early! We are nearing the end of the course and finding time to complete the project in the last couple of weeks will prove difficult, and office hours will flood.
- Code your library! Specifically, focus on the read-only functions, and return -1 for all the write functions until you have all the read functionality down. The important thing is to be able to build the data structure(s) that stores your file system, and getting **loadWad** and **getDirectory** working first.
- Test with wad_dump.cpp before you test with the provided test suite. Examine wad_dump.cpp and libtests.cpp, so that you know what behavior is expected.
- Get all library functionality working perfectly before you move on to the daemon. If you pass 35 tests on the test suite, you're good to start. Not knowing whether a bug resides in your daemon or library will make debugging extremely difficult.
- Make sure to carefully read the P3 PDF, P3 discussion slides, and P3 Canvas assignment page before asking questions in the Discord channel for P3.