UF UNIVERSITY of FLORIDA
Herbert Wertheim College of Engineering

COMPUTING FOR LIFE
COP4600

P3: File Systems

Overview

Your cover in the Lizard Legion was blown, and you've been revealed as a double agent and driven out! It was all very "James Bond", if you do say so yourself, and what a daring underground helicopter escape it was… but you feel lucky to have escaped with your skin. (Literally... they would have used you to make a "human suit"!) Now that you're back on the "outside", you've been tasked with creating a scheme to allow remaining resistance fighters still within the Lizard Legion to clandestinely move information back to your organization without raising suspicion. As of late, members of the Lizard Legion have discovered the PC classic "DOOM", and it has become all the rage to build new mods for it at headquarters, so your team has decided to use mods for this title as a vehicle for exfiltration. By burying encrypted bits within textures and other game data blocks, information can be hidden within innocuous "WAD" (Where's All the Data) files.

In this project, you will implement a userspace filesystem daemon using the FUSE (Filesystem in UserSpacE) API to access data in WAD format, the standard used in a number of classic PC game titles (including DOOM and Hexen). In this critical early prototype, you have been tasked with implementing read and write access to files and directories within the WAD files as a proof-of-concept. As such, you will need to implement read and write functionality for both files and directories within your FUSE-based program. We, as your comrades-in-arms battling the Reptilian invasion, will provide sample WAD files to demonstrate the functionality of your implementation. (The resistance is counting on you!) The resistance uses university courses as cover for standard operations, so you'll submit the project via Canvas.

Structure

The project is broken into three main parts:

1) Develop a library to read from and write to WAD files and create a directory and file structure from them.
2) Implement a userspace daemon (via FUSE) to access the directory structure once mounted.
3) Test your implementation by navigating the mounted directory, examining the names and file contents, and adding directories and files of your own.

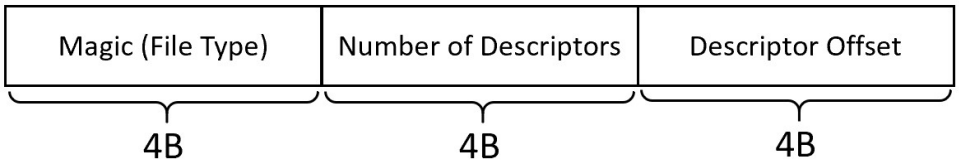
While exact implementation may vary, the daemon's parameters must match those laid out in this document, and the directory structure, naming, and file contents must be properly presented via the filesystem.

File Format

The WAD file format contains information in three sections: the ***header***, which gives basic layout information, the ***descriptors***, which describe elements in the file, and the ***lumps***, which contain the data themselves. **NOTE**: all numbers are in little-Endian format and, where applicable, are designated in bytes! Since Reptilian stores its variables in memory in little-Endian format as well, it is not necessary to perform any byte-order inversions when reading in or writing data, but this is still important information to know.

File Header

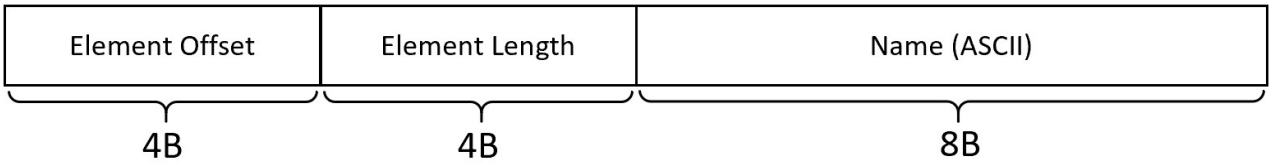
The header contains the file ***magic***, descriptor count, and location (offset) of the descriptors in the file:



The magic for a wad file is usually ASCII and always ends in the suffix **`"WAD"`** (e.g., **`"IWAD"`** or **`"PWAD"`**). It is also important to note that the descriptor list, beginning at the position indicated by the descriptor offset, is always situated at the end of the WAD file.

Descriptors

The file's descriptors contain information about elements in the WAD file – its file offset, length, and name:



Some elements will have specific naming conventions that will differentiate them from regular content files. These "marker" elements will be interpreted by the daemon as directories and should be displayed accordingly in the filesystem (see below).

Lumps

Elements in the WAD format are stored as "lumps" described by the descriptors. These lumps will be represented in the filesystem by the daemon as individual files that can be opened, read, and closed. You cannot write to existing lumps, but you will be creating empty files whose lumps you will have to write to.

Marker Elements

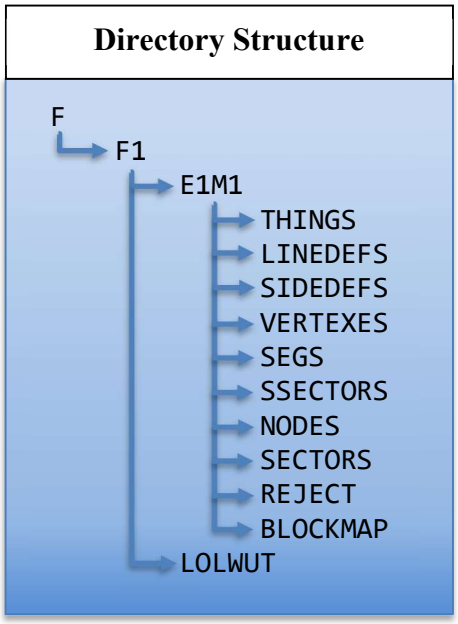
There are two primary types of marker elements in WAD files, each of which should be interpreted as directories by our daemon. The type includes map markers and namespace markers.

Map marker names are of the format **`"E#M#"`**, where **`#`** represents a single decimal digit (e.g., **`"E1M9"`**). They are followed by ten (10) map element descriptors. The elements for the next 10 descriptors should be placed inside of a directory with the map's name. Map marker directories cannot have files or directories added to them.

Namespace markers come in pairs. A namespace's *beginning* is marked with a descriptor whose name has the suffix **`"_START"`** (e.g., **`"F1_START"`**), and its ending is marked with a descriptor whose name has the suffix **`"_END"`** (e.g., **`"F1_END"`**). Any descriptors for elements falling between the beginning and ending markers for a namespace should be placed within a directory with the namespace's name (e.g., **`"F1"`**). The namespace marker's name, excluding the suffixes, will never exceed two characters. These will be the kind of directories you will be responsible for creating.

As an example, the following descriptors, in order, in the descriptor list, should result in this organization:

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



Library

Your library will contain a class to represent WAD data as described in this section.

Wad Class

The Wad class is used to represent WAD data and should have the following functions. The root of all paths in the WAD data should be **"/"**, and each directory should be separated by **'/'** (e.g., **"/F/F1/LOLWUT"**).

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

Object allocator; dynamically creates a **Wad** object and loads the WAD file data from **path** into memory. Caller must deallocate the memory using the **delete** keyword.

```
public string getMagic()
```

Returns the ***magic*** for this WAD data.

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

Returns **true** if **path** represents content (data), and **false** otherwise.

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

Returns **true** if **path** represents a directory, and **false** otherwise.

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

If **path** represents content, returns the number of bytes in its data; otherwise, returns **-1**.

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

If **path** represents content, copies as many bytes as are available, up to **length**, of content's data into the pre-existing **buffer**. If **offset** is provided, data should be copied starting from that byte in the content. Returns number of bytes copied into **buffer**, or **-1** if **path** does not represent content (e.g., if it represents a directory).

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

If **path** represents a directory, places entries for immediately contained elements in **directory**. The elements should be placed in the directory in the same order as they are found in the WAD file. Returns the number of elements in the directory, or **-1** if **path** does not represent a directory (e.g., if it represents content).

```
public void createDirectory(const string &path)
```
path includes the name of the new directory to be created. If given a valid **path**, creates a new directory using namespace markers at **path**. The two new namespace markers will be added just before the "_END" marker of its parent directory. New directories cannot be created inside map markers.

```
public void createFile(const string &path)
```
path includes the name of the new file to be created. If given a valid **path**, creates an empty file at **path**, with an offset and length of 0. The file will be added to the descriptor list just before the "_END" marker of its parent directory. New files cannot be created inside map markers.

```
public int writeToFile(const string &path, const char *buffer, int length, int offset = 0)
```
If given a valid **path** to an empty file, augments file size and generates a lump offset, then writes **length** amount of bytes from the **buffer** into the file's lump data. If **offset** is provided, data should be written starting from that byte in the lump content. Returns number of bytes copied from **buffer**, or -1 if **path** does not represent content (e.g., if it represents a directory).

NOTE: If a file or directory is created inside the root directory, it will be placed at the very end of the descriptor list, instead of before an "_END" namespace marker.

Daemon Command & Parameters

Your daemon should have name **wadfs** and should accept at a minimum three parameters – the single-threaded flag "-s", the target WAD file, and the mount directory. For example, this command should mount **TINY.WAD** in **/home/reptilian/mountdir**…

```
$ ./wadfs -s TINY.WAD /home/reptilian/mountdir
$
```

…and this should result from executing the **ls** command to show part of its contents:

```
$ ls /home/reptilian/mountdir/F/F1 -al
total 0
drwxrwxrwx. 2 root root   0 Jan  1  1970 .
drwxrwxrwx. 2 root root   0 Jan  1  1970 ..
drwxrwxrwx. 2 root root   0 Jan  1  1970 E1M1
-rwxrwxrwx. 2 root root 9001 Jan  1  1970 LOLWUT
```

We will use the following command below to unmount your filesystem:

```
$ fusermount -u /home/reptilian/mountdir
```

Your daemon should run in the background. **Do not hard-code** the debug (**-d**) or foreground (**-f**) flags!

Extra Credit

You may notice when testing with your daemon that there is an upper limit to how large files you create in your filesystem can be. Your task is to configure your library and daemon such that you are able to create large files in your filesystem (using "cp" to copy in a 200KB image file, for example). Running your daemon in debug mode (**-d**) may give you hints as to how certain calls are expected to behave.

File and Directory Requirements

Your daemon must implement, at a minimum, the following filesystem functions to provide read and write access:

1) Retrieving file and directory attributes
2) Reading from existing files, and writing to new ones
3) Reading from existing directories, and writing to new ones

Files and directories should be given full read, write, and execute permissions.

The above requirements will be achieved using, at a minimum, the following six fuse callback functions: **`get_attr, mknod, mkdir, read, write,`** and **`readdir`**

It is highly recommended to closely follow the linked resources at the bottom of this pdf to assist with your FUSE implementation. All changes to the filesystem, such as directory and file creation, must survive between mounting and unmounting.

Building with FUSE

FUSE is a userspace filesystem API that is supported directly by the Linux kernel. It allows userspace programs to provide information to the kernel about filesystems the kernel cannot interpret on its own.

Installation & Setup

To use the FUSE library, you will need to install it within Reptilian and change the FUSE permissions:

```
$ sudo apt install libfuse-dev fuse
$ sudo chmod 666 /dev/fuse
```

NOTE: if you reboot the virtual machine, you will need to re-add the FUSE permissions, as they will be reset!

Build Directives

In order to build programs using the FUSE library system, you will need to specify the file offset bits as 64 and identify the FUSE version. We recommend specifying FUSE version 26 (though this is optional):

```
$ g++ -D_FILE_OFFSET_BITS=64 -DFUSE_USE_VERSION=26 myproggy.cpp -o myproggy -lfuse
```

Submissions

You will submit the following at the end of this project:

- Report (**`p3.txt`**) in **man page format** on Canvas, including link to unlisted YouTube screencast
- Compressed tar archive (**`wad.tar.gz`**) for **libWad** library and **wadfs** daemon on Canvas

Report

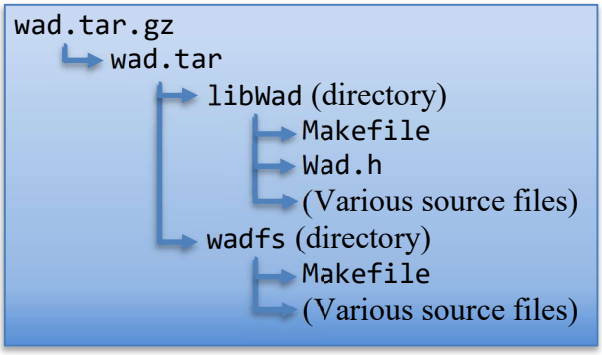
Your report will explain how you implemented the daemon, including your general architecture / program structure. It must include an explanation of how you represent the WAD file elements as a directory structure in memory, as well as how this structure was utilized in the daemon when running. It will include a description of how testing was performed along with any known bugs. The report should be no more than 600 words, cover all relevant aspects of the project, and be organized and formatted professionally – ***this is not a memo!***

Screencast

In addition to the written text report, you should submit an **unlisted YouTube** screencast (with audio) walking through your library and the daemon you wrote to provide the filesystem interface, describing your primary functions and structures (~5:30).

Compressed Archive (wad.tar.gz)

Your compressed tar file should have the following directory/file structure:



To build the library and daemon, we will execute these commands:

```
$ tar zxvf wad.tar.gz
$ cd libWad
$ make
$ cd ..
$ cd wadfs
$ make
$ cd ..
```

To run your daemon, we will execute this command:

```
$ ./wadfs/wadfs -s somewadfile.wad /some/mount/directory
```

To build another program using your library, we will execute this command:

```
$ c++ -o program_name sourcefile.cpp -L ./libWad -lWad
```

Helpful Links

You may find the following resources helpful when reading about how to implement a FUSE daemon:

https://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/html/
https://engineering.facile.it/blog/eng/write-filesystem-fuse/
https://maastaar.net/fuse/linux/filesystem/c/2019/09/28/writing-less-simple-yet-stupid-filesystem-using-FUSE-in-C/
https://www.cs.hmc.edu/~geoff/classes/hmc.cs137.201601/homework/fuse/fuse_doc.html
http://slade.mancubus.net/index.php?page=about