CSCI-333 OS Project 3: ISO Filesystem in Userspace

Due: December 20th, 11:59pm

In this project you will be creating a readonly FUSE-based filesystem that gets its data from a CD image in an "ISO" file. Technically, it is an ISO-9660 filesystem. In this project, there are many details of the ISO-9660 filesystem that we will be ignoring or are handled for you. It is definitely a strange format (for example, every value that isn't a single byte is stored twice: once in little-endian and once in big-endian) but I will try to lead you through just the important parts.

If you want to read more about the ISO-9660 filesystem, take a look at the following links:

- Overview of the format and the data structures: wiki.osdev.org/ISO_9660
- The complete specification: ecma-international.org/publications/files/ECMA-ST/Ecma-119.pdf
- The Rock Ridge specification: cdrtools.sourceforge.net/private/RRIP/rrip.ps

During this project you will explore how a filesystem is structured in memory, use new low-level features such as file descriptors and memory mapping, and implement the API of a filesystem.

Part 1: Load the ISO and Find the PVD

In part 1 you will use memory mapping to bring the ISO file into memory and then find the first data structure of the ISO file system: the Primary Volume Descriptor (PVD).

Follow the first 2 TODO comments in the `load_iso()` function in part1.c. This will require using several functions you have likely never used before. Read up on them at the linked resources (or elsewhere on the internet). You must check every return value in this function, and if anything goes wrong you need to clean up and return NULL. Do not print anything out (except for debugging, remove before submitting). The functions you need are:

- `open()` and `close()`: linux.die.net/man/2/open and linux.die.net/man/2/close
- `fstat()`: linux.die.net/man/2/fstat
- `mmap()` and `munmap()`: linux.die.net/man/2/mmap

You will be filling in the ISO structure, look in the util.h file for more information about that.

Once you get that file memory mapped, it can be accessed like a pointer to a whole bunch of bytes. Those bytes aren't just random but that are organized in a specific way: according to the ISO-9660 specifications. You will need to go through the bytes and figure out what they mean. This journey starts at the Primary Volume Descriptor which describes the general information about the filesystem along with where to find the other important data structures in memory.

The PVD is just one volume descriptor, there are others that may be present as well. We have to find the correct one. The default block (or sector) size of an ISO file is 2048 or 0x800 and all of the volume descriptors are in their own blocks. Every volume descriptor starts with the same 6 bytes as listed in the VolumeDescriptor struct in the iso.h file and below:

Name	Datatype	Description
`type_code`	`uint8`	`0x01` for Primary Volume Descriptor `0xFF` for Volume Descriptor Terminator
`id`	`char[5]`	Always `"CD001"`
`version`	`uint8`	Always `0x01`

The first volume descriptor is in block/sector 16 (i.e. starts at byte 0x8000). All data before that is not used by the ISO-9660 format. You have to look at each block starting at block 16, check that it is a valid volume descriptor (using `memcmp()` to check if the id is equal to `CD001` and that the version is 1) and then check the type code. If the type code is `VD_PRIMARY` (which equals 0x01), then it is a Primary Volume Descriptor, and if it is the first one you have seen, you save its pointer to the ISO structure. If the type code is `VD_TERMINATOR` (which equals `0xFF`) then you stop searching for volume descriptors.

Besides checking the volume descriptor id and the version you must also must check to make sure that you never go too far in memory (past the end of the file) and that by the end you find a Primary Volume Descriptor. If any of these checks fail, you must cleanup everything, set `errno` to `EINVAL`, and return `NULL`.

Testing

The following should be displayed with the given ISO files:

```
$ ./part1 examples/test.iso
Primary Volume Descriptor
-------------------------
Offset in file: 0x8000
Type Code: 0x01  (always 0x01)
ID:        CD001 (always CD001)
Version:   0x01  (always 0x01)
Volume Space Size:  0x38bb 14523 blocks
Logical Block Size: 0x0800 2048 bytes/block
$ ./part1 examples/slackware-3.1.iso
Primary Volume Descriptor
-------------------------
Offset in file: 0x8000
Type Code: 0x01  (always 0x01)
ID:        CD001 (always CD001)
Version:   0x01  (always 0x01)
Volume Space Size:  0x4c8a2 313506 blocks
Logical Block Size: 0x0800 2048 bytes/block
```

(Note that the lines that start with `$` are commands that you type, don't type the `$`).

Part 2: Locate a Record

Now that you have the Primary Volume Descriptor (PVD) you can begin to look for files. The ISO 9660 filesystem is a tree-structure[1]. The `get_record()` function you will write will take an absolute path within the ISO file that starts with `"/"` and return the directory record for it (or NULL for errors). ***This is likely the hardest function in the entire project.***

There are two fields you will need from the PVD to traverse the directory tree[2]:

- Logical Block Size - bytes in a "logical block", typically 2048, but could be any value
- Root Record - the directory record of the root of the filesystem tree

If the path is just the string `"/"` (which you could find out using the `strcmp()` function), then just return the root directory record. Otherwise you will need to break the path into the separate parts (you should use the `get_path_names()` function from the util.h file for this) and look through each directory record finding the next child to descend into for each part. Due to how the directories are laid out and how many things you have to check, this is a challenge.

A single directory record (or simply `Record`) contains the following important fields:

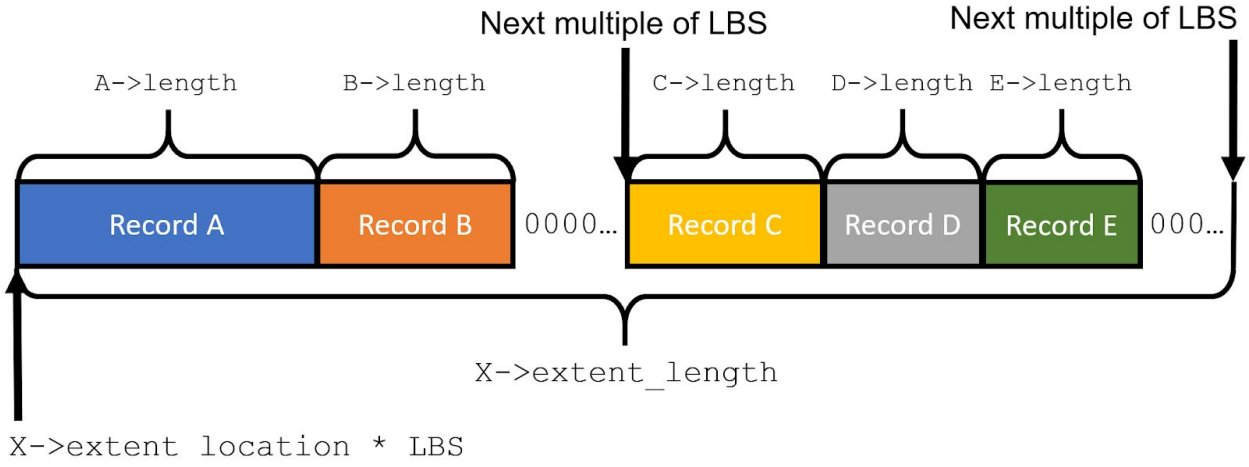
Name	Datatype	Description
`length`	`uint8`	Length of this directory record (including this byte)
`extent_location`	`uint32`	The block number of the data for this record (either the file's contents or another list of directory records)
`extent_length`	`uint32`	The number of bytes in the data for this record
`datetime`	`datetime`	File date and time (won't need till Part 3)
`file_flags`	`uint8`	Bit-flags, a combination `FILE_*` constants from the iso.h file, we care about the `FILE_DIRECTORY` flag

There is also a `filename` field, however, filenames in ISO 9660 are strange[3] so I have given you the function `get_record_filename()` to get the filename from a record. To use it, pass it a stack allocated array of 256 characters which will be filled in with the filename.

[1] Technically you could craft a DAG or even cyclic ISO 9660 file structure, but that would be cruel and actually against the official specifications, but I kind of want to try it now...
[2] You can also traverse it with the path table, but we will not use that since it has some issues and ultimately leads back to traversing parts of the directory records.
[3] Technically only 8.3 names with only capital letters, numbers, and the underscore are really allowed, and the "current" and "parent" names are not printable characters. However, the Rock Ridge extension to the ISO 9660 format that can be used to give better names among other enhancements.

Iterating the Files in a Directory

Once you have a record directory that you know is a directory (has the `FILE_DIRECTORY` flag) you can then iterate through all of the directories and files that are immediately in that directory. To do so, find the data in the raw data of the ISO file using the extent location of the `Record` for the directory (which is in blocks, you need to use the logical block size given in the PVD to convert it to bytes). This memory location can be cast to a `Record` pointer. You can then examine this `Record`, getting its filename and whether or not it is a directory. You can then advance to the next `Record` in this directory by moving `length` more bytes from the previous Record and casting again to a `Record` pointer. If the next `Record` has a length of 0 that means we need to jump to the next block (so the next multiple of the logical block size). Repeat going through `Record`s like this until you have gone a total of extent length bytes (this value comes from the original `Record` for which you used the extent location of). See the following diagram for a visual representation of this where LBS stand for the logical block size and `X` is the current directory we are trying to list the contents of.



Two of the records within a directory are special but you really don't need to handle them any differently. They refer to the current directory (`get_record_filename()` gives `"."`) and the parent directory (`get_record_filename()` gives `".."`).

Error Checks

You must check that `get_path_names()` succeeds, that the entire extent of each directory you scan is within the bounds of the ISO file data (if not set `errno` to `EINVAL`), that every intermediate path name (i.e. not the last one) actually refers to a directory (if not set `errno` to `ENOTDIR`). If the path names has a trailing slash, then the last one must also refer to a directory (if not set `errno` to `ENOTDIR`). Also, if you don't find any part of the path name then you set `errno` to `ENOENT`. In all of these cases make sure to clean up and return `NULL`.

Example Output

```
$ ./part2 examples/test.iso /
Basic File Information
------------------------
Record Length:   0x22 34 bytes
Extent Location: 0x00000018 24 blocks
Extent Length:   0x00000800 2048 bytes
Date/Time: 2019-11-17 16:09:35
Flags: 02
Raw Filename:
$ ./part2 examples/test.iso /readme
Basic File Information
------------------------
Record Length:   0x78 120 bytes
Extent Location: 0x000000e9 233 blocks
Extent Length:   0x00000081 129 bytes
Date/Time: 2019-11-17 16:09:35
Flags: 00
Raw Filename: README.;1

Rock Ridge Extension Info
------------------------
Mode:   0100444
#Links: 1
UID:     0
GID:     0
Filename: readme
Modification: 2019-11-17 16:09:35
Access:        2019-11-17 16:11:04
$ ./part2 examples/slackware-3.1.iso /Adobe/acroread_linux_b1106.tar.gz
Basic File Information
------------------------
Record Length:   0x94 148 bytes
Extent Location: 0x00024d20 150816 blocks
Extent Length:   0x004aaaa9 4893353 bytes
Date/Time: 1997-01-24 12:53:00
Flags: 00
Raw Filename: ACROREAD.TAR;1

Rock Ridge Extension Info
------------------------
Mode:   0100444
#Links: 1
UID:     0
GID:     0
Filename: acroread_linux_b1106.tar.gz
Modification: 1997-01-14 10:48:36
Access:        1997-01-24 12:53:20
```

Part 3: ISO Filesystem

The three functions you have made so far will help make the final part - the actual ISO filesystem. For this part you have to create many functions, but most are fairly straight forward since you already have created `get_record()`. As you actually implement these functions, you need to uncomment their usage in the `isofs_oper` object near the bottom of the file. Make sure that every set of functions you test and make sure everything is working since if an early function doesn't work, the later ones won't likely work either.

With each function, there is the documentation as provided by the FUSE library and also additional comments provided by me. In each of these functions you can use `GET_ISO()` to get a reference to the currently mounted ISO file. To compile, use the following command. You may add `-D_DEBUG` to near the beginning of it to see a log of the functions as they are called.

```
gcc -I/usr/local/include/osxfuse/fuse isofs.c -Wall -o isofs
                                -losxfuse
```

To actually run the program you will have to do something like:

```
./isofs -f test.iso mount
```

where `test.iso` is the ISO file to mount and `mount` is an empty directory to mount it into. You will need two terminals since this will block one of them. In the other terminal you can use basic command line tools like `cd`, `ls`, `ls -l`, `cat`, … to test the file system. Eventually, you would run this command without the `-f` argument which would cause the program to instead run in the background, but then you can't see printouts or error messages.

Note that initially you will get compile-time warnings about unused variables in the functions you have not yet written. You can ignore these until you actually get around to implementing those functions.

Basic Information

First you will implement the `isofs_statfs()` where you will fill in the provided `statvfs` structure. However, many of the values are ignored or get default values, see the code for which ones you should fill in. You will need to look through the fields in the PVD in the OSDev link on the first page and use the `get_number_of_files()` fro util.h to finish this function.

This function can be tested with the `df` program as is shown in the following.
If `test.iso` is mounted to the `mount` directory:

```
$ df solution/mount --output=fstype,source,itotal,size
Filesystem     Inodes 1K-blocks
Isofs             161     29046
```

If `slackware-3.1.iso` is mounted to the `mount` directory:

```
$ df solution/mount --output=fstype,source,itotal,size
Filesystem     Inodes 1K-blocks
Isofs             388    627012
```

Next, finish the `isofs_getattr()` function which will then all you to get information about individual files (but not the content). This function is the longer function you have to make but only because there are so many assignments to values in the `stat` structure you need to do. It is fairly straightforward. You can test this as follows on the next page.
If `test.iso` is mounted to the `mount` directory:

```
$ ls -l mount/readme
-r--r--r--. 1 root root 129 Nov 17 16:09 mount/readme
```

And for `slackware-3.1.iso`:

```
$ ls -l mount/Adobe/acroread_linux_b1106.tar.gz
-r--r--r--. 1 root root 4893353 Jan 14  1997 mount/Adobe/acroread_linux_b1106.tar.gz
```

Finally, finish the `isofs_access()` function. Only 2 lines of code are needed here. These two lines will be very common moving forward. Additionally, take a look at the last `if` statement at the end, something like this will be needed in some of the upcoming functions. This function is a bit different to directly test.

Listing Directories

Implementing `isofs_opendir()`, `isofs_readdir()`, and `isofs_releasedir()` allows you to list the contents of files (either with the `ls` and `ls -l` commands on the command line or just using Finder). The function `isofs_readdir()` will use a similar setup (albeit simplified) from what you did in `get_record()` to list the contents of a directory. You can test this as follows (remember that you need to uncomment these functions in the `isofs_oper` structure). You can also use Finder and look around (although not open files, just directories).

You can test by using the `ls` command. For `test.iso` mounted to the `mount` directory:

```
$ ls -li mount/
total 65
2 dr-xr-xr-x. 9 root root  2048 Nov 17 15:36  CSC1-120
3 dr-xr-xr-x. 2 root root  2048 Nov 17 16:08  ISOLINUX
4 -r--r--r--. 1 root root 59312 Mar  6  2017 'My Résumé.pdf'
5 -r--r--r--. 1 root root   129 Nov 17 16:09  readme
6 dr-xr-xr-x. 2 root root  2048 Nov 17 16:08  SYSLINUX
$ ls -l mount/CSCI-120/
 2 dr-xr-xr-x.  9 root  root   2048 Nov 17 15:36 .
 1 dr-xr-xr-x.  1 jbush jbush  2048 Nov 17 16:09 ..
13 dr-xr-xr-x.  2 root  root   2048 Nov 17 15:36 General
14 dr-xr-xr-x. 14 root  root   2048 Nov 17 15:39 Homework
15 dr-xr-xr-x.  4 root  root   2048 Nov 17 15:26 Information
16 dr-xr-xr-x. 13 root  root   2048 Nov 17 15:40 Labs
17 dr-xr-xr-x.  4 root  root  10240 Nov 17 15:37 Lectures
18 dr-xr-xr-x.  5 root  root   2048 Nov 17 15:26 Projects
19 dr-xr-xr-x.  2 root  root   2048 Nov 17 15:36 Quizzes
```

And for `slackware-3.1.iso`:

```
$ ls -li mount/
total 417
 2 dr-xr-xr-x.  2 root root  2048 Jan 24  1997 Adobe
 3 dr-xr-xr-x.  2 root root  8192 Jan 23  1997 bootdsks.12
 4 dr-xr-xr-x.  2 root root  8192 Jan 23  1997 bootdsks.144
 5 -r--r--r--.  1 root root 18605 Jun 28  1996 BOOTING.TXT
 6 dr-xr-xr-x.  2 root root 18432 Jan 24  1997 CGI-Developers-Guide
 7 -r--r--r--.  1 root root 16474 Jan 20  1997 ChangeLog.txt
 8 dr-xr-xr-x.  3 root root 34816 Jan 23  1997 contents
 9 dr-xr-xr-x.  5 root root  6144 Dec  2  1996 contrib
10 -r--r--r--.  1 root root 17976 Jun 10  1994 COPYING
11 -r--r--r--.  1 root root  9965 Jun 10  1996 Copyright
12 dr-xr-xr-x.  7 root root 12288 Jan 24  1997 docs
13 -r--r--r--.  1 root root 25207 Jun 28  1996 FAQ.TXT
14 dr-xr-xr-x.  5 root root 49152 Jan 24  1997 gnu
15 dr-xr-xr-x.  3 root root  2048 Aug  9  1996 install
...
```

Reading Files

Your final task will be to allow all files to be read. To do this you need to implement `isofs_open()`, `isofs_read()`, and `isofs_release()` along with uncommetting their entries in `isofs_oper`. After that you can use `cat` on the command line to test things, or simply open files in the Finder and look at their contents.

For example, the contents of the “readme” file in the test.iso file (which you could display in the terminal using mount/readme or by open it in a text editor) are:

```
This is an example ISO that has various types of files, characters sets, etc.

It also has a bootable MemTest x86 program on it.
```