Assignment 1: Reading Unix v6 File Systems

Thanks to Mendel Rosenblum for this wonderful assignment. This is a real system, people!

An archaeological expedition in Murray Hill, NJ has uncovered some magnetic disk drives from the mid-1970s. After considerable effort, the dig's technical expert read the contents of the drives, and for each disk drive, she created a local file to be an exact bitwise copy of the disk's data.

The archaeologists determined the data on the disks was stored using the Version 6 Unix file system. Sadly, none of the archaeologists made it through CS110, so they haven't been able to read the image contents. Your task is to write a program that understands the Unix v6 file system to extract the file system data.

Due Date: Monday, Monday, October 10, 2016 at 11:59 p.m.

No late days can be used on this assignment.

Getting started

All coding should be done on the `myth` cluster machines, as that's where we'll be testing your submission. To get your own copy of the starter code, you should clone the master mercurial repository we've set up for you by typing:

```
myth22> hg clone /usr/class/cs110/repos/assign1/$USER assign1
```

Doing so will create an `assign1` directory within your own space, and you can descend into that directory to land on the collection of files you'll be modifying to arrive at a working product. (If no repository exists for you, email Jerry to let me know. In the meantime, clone `/usr/class/cs110/repos/assign1/guest`) so you can get started until your own repository is created, at which point you can clone that one and copy your work into it.)

If you've never used mercurial (i.e. `hg`) before, you should read the CS107 guide to using mercurial, as that's where hg is first taught. We're leveraging the same tools we've built up for CS107 and using them in CS110. Doing so will make development and grading go much more smoothly.

Building, running, testing, and committing local changes

To build the project, `cd` into your local repository and type

```
myth22> make
```

To test your work, `cd` into your local repository and type

```
myth22> ./diskimageaccess <options> <diskimagePath>
```

`<diskimagePath>` should be the path to one of the test disks located in **`/usr/class/cs110/samples/assign1/testdisks`**. We currently have three test disks: **`basicDiskImage`**, **`depthFileDiskImage`**, and **`dirFnameSizeDiskImage`**.

`diskimageaccess` recognizes two flags as valid **`<options>`**:

- `-i`: test the inode and file layers
- `-p`: test the filename and pathname layers

For example, to run both the inode and filename tests on the basic disk, you could run

```
myth22> ln -s /usr/class/cs110/samples/assign1/testdisks testdisks
myth22> ./diskimageaccess -ip testdisks/basicDiskImage
```

The expected output for running **`diskimageaccess`** on each disk image **`X`** is stored in the **`X.gold`** file inside the **`testdisks`** directory. In fact, we're leveraging even another CS107 tool: the **`sanitycheck`** tool, which you can read about right here.

In particular, if you type

```
myth22> /usr/class/cs110/tools/sanitycheck
```

at the command prompt while in your **`assign1`** directory, the **`sanitycheck`** script will exercise your implementation in precisely the same way our own grading scripts will. We won't always expose all of the functionality tests like we are for Assignment 1, but we are this time.

As you develop, it's a good idea to locally commit your work to your own repository by typing

```
myth22> hg commit
```

as doing so minimizes the chances that you'll lose your work.

Submitting `assign1`

When time comes for you to push your submission back to the master (which is the version we'll grade), do one final **`hg commit`** if needed, and then type

```
myth22> /usr/class/cs110/tools/submit
```

What we provide

The files we provide you fall into three categories.

The Version 6 Unix header files (`filsys.h`, `ino.h`, `direntv6.h`)

These are described below.

The test harness (`diskimageaccess.c`, `chksumfile.[ch]`, `unixfilesystems.[ch]`)

These files provide the infrastructure for building your code and running our tests against it.

File system module (`diskimg.[ch]`, `inode.[ch]`, `file.[ch]`, `pathname.[ch]`)

The test code we give you interfaces with the file system using a layered API that we've already designed. For each of the layers that you'll need to implement for this assignment, there is a header file that defines the interface to the layer and a corresponding `.c` file that should contain the implementation.

The layers are:

Block layer (`diskimg.[ch]`)

This defines and implements the interface for reading and writing sectors (note that the words block and sector are used interchangeably) on the disk image. We give you an implementation of this layer that should be sufficient for this assignment.

inode layer (`inode.[ch]`)

This defines and implements the interface for reading the file system's inodes. This includes the ability to look up inodes by inumber and to get the block/sector number of the nth block of the inode's data.

File layer (`file.[ch]`)

This defines and implements the interface for reading blocks of data from a file by specifying its inumber. This is one of the two layers our tests explicitly exercise.

Filename layer (`directory.[ch]`)

This defines and implements the interface for implementing Unix directories on top of files. Its primary function is to get information about a single directory entry.

Pathname layer (`pathname.[ch]`)

This defines and implements the interface to look up a file by its absolute pathname. This is the second of the two layers we explicitly test.

You are welcome to modify any of the files we give you. When making changes in the test harness, do so in a backward compatible way so that the testing scripts we give you continue to work. In other words, it's fine to add testing and debugging code, but make sure that when we run your program with `-i` and `-p`, its output has the same format with or without your changes, so our automated tests won't break.

Before you try to write any code, you'll want to familiarize yourself with the details of the Unix V6 file system by reading Section 2.5 of the Saltzer and Kaashoek textbook. You may also find it helpful to read and understand how the test script works.

Suggested Implementation Order

The starter code contains a number of unfinished functions. We suggest you attack them in the following order:

- `inode_iget` and `inode_indexlookup` in `inode.c`.
- `file_getblock` in `file.c`. After this step, your code should pass the inode level functionality tests.
- `directory_findname` in `directory.c`.
- `pathname_lookup` in `pathname.c`. Refer to Section 2.5.6 of the Salzer and Kaashoek book for this part. Afterward, all the pathname tests should pass.

Grading

We will grade your assignment based on a combination of test results and code quality.

- Code test results: 60 points
- Clean build and clean `valgrind` reports: 6 points [check out this page hosted on the CS107 course web site to learn all about `valgrind`]
- Code quality: grades on bucket system [exceptional, solid, minor-problems, major-problems, etc.]

First, we'll run your code on a set of test disk images and check that your program gives the expected output. Of course, you have access to all of the images we'll use for grading and the correct output for each of them.

If this were a real archaeological dig, you would have been given only the disk image and a computer. But to make things easier, we've provided you with starter code that tries to read all the files on the image and outputs checksums of the inodes' and files' contents. We've also computed

these checksums using a working implementation of the assignment. If your checksums match ours, then your code is correctly reading the data off the disk.

Finally, we'll read through your code and give you comments on style and overall quality. We'll look for places where you inappropriately break the layering of the file system or make other mistakes that don't necessarily cause your program to generate the wrong output. We'll also look for common C programming errors such as memory leaks and use of the heap where stack allocation is more appropriate. Finally, we'll check that your code is easy to read: It should be well-formatted, organized, and be easy to follow.

Unix v6 file system supplement

Section 2.5 of the Salzer and Kaashoek book contains most of what you need to know about the Unix v6 file system in order to do this assignment. The information below is supplementary to the textbook, so it assumes you've already read and understand the material there.

Header files and structures

In the starter code we've provided C `structs` corresponding to the file system's on-disk data structures. These `structs` have the same layout in memory as the structures have on disk. They include:

`struct filsys` (`filsys.h`)

Corresponds to the superblock of the file system. This is a slightly modified copy of the header file from Version 6 Unix.

`struct inode` (`ino.h`)

Corresponds to a single inode. Again this comes from Version 6 of Unix, with some small modifications.

`struct direntv6` (`direntv6.h`)

Corresponds to a directory entry. This was copied from section 2.5 in the textbook.

In addition, `unixfilesystem.h` contains a description of the file system layout, including the sector address of the superblock and of the start of the inode region.

Legacy of an old machine

Back in the 1970s, storage space — both on disk and in main memory — was at a premium. As a result, the Unix v6 file system goes to lengths to reduce the size of data it stores. You'll notice that many integer values in the `structs` we provided are stored using only 16 bits, rather than today's

more standard 32 or 64. (In our code we use the type `int16_t` from `stdint.h` to get a 16-bit integer, but back in the '70s, the C `int` type was 16 bits wide.)

In another space-saving move, the designers of the file system stored the inode's size field as a 24-bit integer. There's no 24-bit integer type in C, so we represent this value using two fields in the `inode struct`: `i_size1`, which contains the least-significant 16 bits of the value, and `i_size0`, which contains the most-significant 8 bits of the value. We provide a function `inode_getsize` in `inode.c` that assembles these two fields into a normal C integer for you.

The first inode

Since there is no inode with an inumber of 0, the designers of the file system decided not to waste the 32 bytes of disk space to store it. The first inode in the first inode block has inumber 1; this inode corresponds to the root directory for the file system. (See `unixfilesystem.h` for details.)

Be careful not to assume that the first inode has an inumber of 0! Off-by-one errors are the worst.

inode's `i_mode`

The 16-bit integer `i_mode` in the `inode struct` isn't really a number; rather, the individual bits of the field indicate various properties of the inode. `ino.h` contains `#define`s which describe what each bit means.

For instance, we say an inode is allocated if it points to an existing file. The most-significant bit (i.e. bit 15) of `i_mode` indicates whether the inode is allocated or not. So the C expression `(i_mode & IALLOC) == 0` is `true` if the inode is unallocated and `false` otherwise.

Similarly, bit 12 of `i_mode` indicates whether the file uses the large file mapping scheme. So if `(i_mode & ILARG) != 0` , then the inode's `i_addr` fields point at indirect and doubly-indirect blocks rather than directly at the data blocks.

Bits 14 and 13 form a 2-bit wide field specifying the type of file. This field is 0 for regular files and 2 (i.e. binary 10, or the constant `IFDIR`) for directories. So the expression `(i_mode & IFMT) == IFDIR` is `true` if the inode is a directory, and `false` otherwise.