Lab Solution 1: File Systems and System Calls

The first and last exercises are problem set-esque questions that could easily appear on a midterm or final exam. In fact, all of the questions asked under Problem 4 were on previous midterms and finals. The middle two problems are experiments that'll require you fire up your laptop and run some programs and development tools.

Solution 1: Direct, Singly Indirect, and Doubly Indirect Block Numbers

Assume blocks are 512 bytes in size, block numbers are four-byte `int`s, and that inodes include space for 6 block numbers. The first three contain direct block numbers, the next two contain singly indirect block numbers, and the final one contains a doubly indirect block number.

- What's the maximum file size?
  - *Maximum file size is 3 * 512 + 2 * 128 * 512 + 128 * 128 * 512, or 8,521,216 bytes.*
- How large does a file need to be before the relevant inode requires the first singly indirect block number be used?
  - *3 * 512 + 1, or 1,537 bytes.*
- How large does a file need to be before the relevant inode requires the first doubly indirect block number be used?
  - *3 * 512 + 2 * 128 * 512 + 1, or 132,609 bytes.*
- Draw as detailed an inode as you can if it's to represent a regular file that's 2049 bytes in size.
  - *Can't easily inline a drawing using Quip, but it's easily described.*
    - *The first three block numbers would identify three, saturated payload blocks, and those three payload blocks would store the first 1,536 bytes.*
    - *The fourth block number–a singly, indirect one–would lead to a block of 512 bytes, although only the first eight bytes are used. The first four bytes would identify the fourth payload block–itself saturated–to store 512 bytes of payload. The next four bytes would identify a fifth payload block where only the first byte is used.*

Solution 2: Experimenting with the `stat` utility

This problem is more about exploration and experimentation, and not so much about generating a correct answer. The file system reachable from each myth machine consists of the local file system (restated, it's mounted on the physical machine) and networked drives that are grafted onto the fringe of the local file system so that all of AFS–which consists of many, many independent file systems from around the globe–all contribute to one virtual file system reachable from your local / directory.

Log into `myth52` and use the `stat` command line utility (which is a user program that makes calls to the `stat` system call as part of its execution) and prints out oodles of information about a file. Type in the following commands and analyze the output:

- `stat /`
- `stat /tmp`
- `stat /usr`
- `stat /usr/bin`
- `stat /usr/bin/g++`
- `stat /usr/bin/g++-5`

The output for each of the five commands above all produce the same device ID but different inode numbers. Read through this to gain insight on what the **`Device`** values are.

- *Output for **`stat /`** includes a size of 4096, a block count of 8, and I/O block size of 4096, an inode number of 2, and a nlinks value of 26. It's also clear it's a directory.*

```
poohbear@myth52:/usr/class/cs110$ stat /
  File: '/'
  Size: 4096          Blocks: 8          IO Block: 4096   directory
Device: 801h/2049d    Inode: 2           Links: 26
Access: (0755/drwxr-xr-x)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2019-01-15 14:34:40.650771469 -0800
Modify: 2018-04-16 17:08:24.260965019 -0700
Change: 2018-04-16 17:08:24.260965019 -0700
 Birth: -
poohbear@myth52:/usr/class/cs110$
```

  - *Directory sizes are always exposed as multiples of the true block size, which is 4096.*
  - *The sector size is 512, and **`stat`** states that it uses 8 sectors to store all of its directory entries. (I have no idea why **`stat`** uses blocks here instead of sectors; I just know it does).*
  - *The I/O block size just happens to be the same as the actual block size, but it's listed separately, because it **could** have been different. The I/O block size is the optimal byte transfer size supported by the file system hardware.*
  - *The device information is expressed as a hexadecimal (801h) and a decimal equivalent (2049d). The 801 states that the major device number is 8, and then the minor device number (or partition) within that major device is 1. If you do an **`ls -lt /dev/sda`***, you get a listing within the pseudo-file system like that below, where **`/dev/sda1`** is pseudofile representing the device driver that interfaces with the file system holding the root directory. (Special files like this are set up so that software can programmatically interact with device drivers as if they were files.)*

```
poohbear@myth52:/usr/class/cs110$ ls -lt /dev/sda*
brw-rw---- 1 root disk 8, 5 Jan 13 09:15 /dev/sda5
brw-rw---- 1 root disk 8, 1 Jan 13 09:15 /dev/sda1
brw-rw---- 1 root disk 8, 2 Jan 13 09:15 /dev/sda2
brw-rw---- 1 root disk 8, 0 Jan 13 09:15 /dev/sda
poohbear@myth52:/usr/class/cs110$
```

  - *The inode number is 2. I'm not sure what inode numbers 0 and 1 store, though–I'll update the handout if I find out.*
  - *The number of links is 26, which means that there are 26 different names leading to the root directory's payload: /. is one, and believe it or not, /.. is a second, since .. is the same as . for the root directory. If you **`ls -lt /`**, you see that there are 24 subdirectories, each of which has a .. directory entry. That's the source of the 26.*
  - *You can do similar listing for the other files as you get the same type of information.*

For each of the above commands, replace **`stat`** with **`stat -f`** to get information about the file system on which the file resides (block size, inode table size, number of free blocks, number of free inodes, etc).

- *Output for **`stat -f /`** looks like this:*

```
poohbear@myth52:/usr/class/cs110/WWW$ stat -f /
  File: "/"
```

```
    ID: 56c68aaafba5efed Namelen: 255     Type: ext2/ext3
Block size: 4096       Fundamental block size: 4096
Blocks: Total: 51833244   Free: 45817962   Available: 43179204
Inodes: Total: 13180928   Free: 12496473
poohbear@myth52:/usr/class/cs110$
```

  - *`ext2` and `ext3` are types of file systems, and the file systems on the myths are evidently a hybrid of the two.*
  - *`namelen` is the maximum length of a filename component supported by the filesystem.*
  - *the fundamental block size is the block size we've discussed in lecture, and the block size (without the fundamental prefix) is a hint as to the optimal transfer size for I/O operations (and is generally the same as the I/O Block value discussed above).*

Now log into `myth55` and run the same commands. Why are the outputs of `stat` and `stat -f` the same in some cases and different in others?

- *Each of `myth52` and `myth55` have independent file systems and might be populated slightly differently. However, the `g++` and `g++-5` executable images, while independent copies of the same file, are exactly that–independent copies of the same file, so all of `g++`'s and `g++-5`'s file properties will be the same.*

Solution 3: `valgrind` and orphaned file descriptors

Here's a very short exercise to enhance your understanding of `valgrind` and what it can do for you. To get started, type the following in to create a local copy of the repository you'll play with for this problem:

```
poohbear@myth58:~$ git clone /usr/class/cs110/repos/lab1/shared lab1
poohbear@myth58:~$ cd lab1
poohbear@myth58:~$ make
```

Now open the file and trace through the code to keep tabs on what file descriptors are created, properly closed, and orphaned. Then run `valgrind ./nonsense` to confirm that there aren't any memory leaks or errors (how could there be?), but then run `valgrind --track-fds=yes ./nonsense` to get information about the file descriptors that were (intentionally) left open. Without changing the logic of the program, insert as many close statements as necessary so that all file descriptors (including 0, 1, and 2) are properly donated back. (In general, you do not have to close file descriptors 0, 1, and 2.)

- *This should be pretty self-explanatory, so I won't include anything here beyond a remark that you need a `close` statement for every open descriptor identified by `valgrind`, and you should be careful to never `close` a previously closed descriptor.*