Lab Handout 1: File Systems and System Calls

Lab created by Jerry Cain.

Problem 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?
- How large does a file need to be before the relevant inode requires the first singly indirect block number be used?
- How large does a file need to be before the relevant inode requires the first doubly indirect block number be used?
- Draw as detailed an inode as you can if it's to represent a regular file that's 2049 bytes in size.

Problem 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.

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).

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?

Problem 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.)