Lab Handout 1: File Systems and System Calls

The first two problems are problem set-like questions that could easily appear on a midterm or final exam. In fact, all of the questions asked under Problem 2 were on previous midterms and finals. The last two problems are experiments that'll require you fire up your laptop and run some programs and development tools.

I've created a specific channel for lab1 discussion (obliquely named #lab1), and all students are encouraged to share their ideas there. SCPD students are welcome to reach out to me directly if they have questions that can't be properly addressed without being physically present for a discussion section.

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: Short Answer Questions

Provide clear answers and/or illustrations for each of the short answer questions below. Each of these questions is either drawn from old exams or based on old exam questions. Questions like this will certainly appear on your own midterm.

1. The `dup` system call accepts a valid file descriptor, claims a new, previously unused file descriptor, configures that new descriptor to alias the same file session as the incoming one, and then returns it. Briefly outline what happens to the relevant file entry table and vnode table entries as a result of `dup` being called. (Read `man dup` if you'd like, though don't worry about error scenarios).

2. Now consider the prototype for the `link` system call (peruse `man link`). A successful call to `link` updates the file system so the file identified by `oldpath` is also identified by `newpath`. Once `link` returns, it's impossible to tell which name was created first. (To be clear, `newpath` isn't just a symbolic link, since it could eventually be the only name for the file.) In the context of the file system discussed in lecture and/or the file system discussed in Section 2.5 of the secondary textbook, explain how link might be implemented.
3. Explain what happens when you type `cd .././../.` at the shell prompt. Frame your explanation in terms of the file system described in Section 2.5 of the secondary textbook, and the fact that the inode number of the current working directory is the only relevant global variable maintained by your shell.
4. All modern file systems allow symbolic links to exist as shortcuts for longer absolute and relative paths (e.g. `search` might be a symbolic link for `/usr/class/cs110/samples/assign1/search`, and `tests.txt` might be a symbolic link for `./mytests/tests.txt`. Explain how your the absolute pathname resolution process we discussed in lecture would need to change to resolve absolute pathnames to inode numbers when some of the pathname components might be symbolic links.
5. Recall that the stack frames for system calls are laid out in a different segment of memory than the stack frames for user functions. How are the parameters passed to the system calls received when invoked from user functions? And how is the process informed that all system call values have been placed and that it's time to execute?

Problem 3: 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 glole–all contribute to one virtual file system reachable from your local / directory.

Log into `myth13` 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 `myth32` and run the same commands. Why are the outputs of `stat` and `stat -f` the same in some cases and different in others?

Now analyze the output of the `stat` utility when levied against AFS mounts where the master copies of all `/usr/class` and `/usr/class/cs110` files reside. Do this from both `myth13` and `myth32`.

- `stat /usr/class`
- `stat -f /usr/class`
- `stat /afs/ir.stanford.edu/class`
- `stat -f /afs/ir.stanford.edu/class`
- `stat /usr/class/cs110`
- `stat /afs/ir.stanford.edu/class/cs110`
- `stat -f /usr/class/cs110`

Why are most of the outputs the same for `myth13` compared to `myth32`? Which ones are symbolic links? Why are the device numbers for remotely hosted file systems so small? What about these commands?

- `stat /afs/northstar.dartmouth.edu`
- `stat -f /afs/northstar.dartmouth.edu`
- `stat /afs/asu.edu`
- `stat -f /afs/asu.edu`

What files can you see within the `dartmouth.edu` and `asu.edu` mounts?

Problem 4: `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@myth32:~$ hg clone /usr/class/cs110/repos/lab1/shared lab1
poohbear@myth32:~$ cd lab1
```

```
poohbear@myth32:~$ 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, I'll argue) left open. Without changing the "algorithm" 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 practice, you do not have to close file descriptors 0, 1, and 2.)