

Lab Handout 1 Solution: 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.

The lab checkoff sheet for all students—both on-campus and off—can be found [right here](#).

This lab was designed 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 `ints`, 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?
 -
- Draw as detailed an inode as you can if it's to represent a regular file that's 2049 bytes in size.
 - Description:
 - *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 for be used. The first four bytes would identify the fourth payload block—itsself saturated—to store 512 bytes of payload. The next four bytes would identify a fifth payload block where only the first byte is used.*

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 (repeated, 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. The inode with number 1 is `/lost+found`
- 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.*

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 `myth52` and `myth55`.

- `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 `myth52` compared to `myth55`? Which ones are symbolic links? Why are the device numbers for remotely hosted file systems so small?

- *All but one of the outputs is the same, because they all refer to the same files on remote file system that's been grafted in to contribute to the overall virtual file system that is AFS (or Andrew File System). (The output for `stat /usr/cLass` is slightly different, because that symbolic link resides on the `myth52` filesystem).*
- *This is what I get when I `stat /usr/cLass` from `myth56`: (`myth52` and `myth55` are down right now :))*

```
poohbear@myth56:/usr/class/cs110$ stat /usr/class
File: '/usr/class' -> '/afs/ir.stanford.edu/class'
Size: 26          Blocks: 0          IO Block: 4096   symbolic link
Device: 801h/2049d Inode: 3063427     Links: 1
Access: (0777/lrwxrwxrwx)  Uid: (  0/   root)  Gid: (  0/   root)
Access: 2019-01-17 13:54:08.301071882 -0800
Modify: 2017-11-30 19:16:59.467003373 -0800
Change: 2017-11-30 19:16:59.467003373 -0800
Birth: -
poohbear@myth56:/usr/class/cs110$
```

- *This is a symbolic link stored on `myth56`'s resident file system (note the `801h` again). It's clearly identified as a symbolic link (represented via inode with inumber 3063427). Its payload size is 26, which is the length of the `/afs/ir.stanford.edu/cLass` the link expands to. The surprising part is that the payload requires 0 blocks! But that's because symbolic link payloads are*

stored not in external blocks, but in the the space set aside for the inode's block numbers array. Sneaky!

- Interestingly, the output of `stat/usr/class/` (note that extra slash at the end) is different, because that extra slash dereferences the symbolic link!

```
poohbear@myth56:/usr/class/cs110$ stat /usr/class/  
File: '/usr/class/'  
Size: 309248          Blocks: 604          IO Block: 4096    directory  
Device: 28h/40d      Inode: 262274        Links: 5  
Access: (0755/drwxr-xr-x)  Uid: (  0/   root)  Gid: (  0/   root)  
Access: 2019-01-14 18:11:41.000000000 -0800  
Modify: 2019-01-14 18:11:41.000000001 -0800  
Change: 2019-01-14 18:11:41.000000000 -0800  
Birth:  
poohbear@myth56:/usr/class/cs110$
```

- Why are the major device numbers so small here? AFS uses major number 0, hence the smaller device number. AFS doesn't expose much about how remotely managed files are stored, which is why they went with a 0.

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?

My most interesting finding with `stat` and `stat -f` on those two remote directories is that the fundamental block sizes are 1024 instead of 2048. And I was able to see 20 directory entries within `/afs/northstar.dartmouth.edu` and 47 directory entries within `/afs/asu.edu`.

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.)

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

Problem 4: 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).
 - a. *The vnode table entry is left alone, but a new file descriptor is claimed and set to address the same entry in the file entry table session as the incoming one, and the reference count within that session entry would be incremented by one.*
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.
 - a. *Resolve `oldpath` to its inode number, append new directory entry to sequence of existing directory entries where `newpath` resides. Fill in entry with last component of `newpath`, and fill in inumber with the inumber of old path. Finally, increment the reference count within the inode itself to be clear the file has one more name.*
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.
 - a. *Search cwd's payload for `..`, set inumber of cwd to inumber associated with `..`; repeat three more times for `..`, then `..`, and then `.` :)*
4. All modern file systems allow symbolic links to exist as shortcuts for longer absolute and relative paths (e.g. `search_soln` might be a symbolic link for `/usr/class/cs110/samples/assign1/search_soln`, and `tests.txt` might be a symbolic link for `./mytests/tests.txt`). Explain how 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.

- a. *The absolute pathname resolution process (implemented as `pathname_Lookup` in `assign2`) should piecemeal tokenize pathnames as usual. When component (e.g. `tests.txt`) is identified as symlink, prepend expansion to unprocessed set of tokens, continue for relative paths from inumber of symlink parent, restart from inode 1 for absolute paths.*
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? How does the system call return back to the user function that invoked it?
 - a. *system call opcode and all parameters are passed through registers (`%rax` for the opcode, others for the 0 to 6 parameters)*
 - b. *system call is invoked by issuing a software interrupt, aka trap, via an assembly code instruction `syscall`. The `syscall` invokes a trap handler, and that handler builds an activation record in the kernel stack using the values passed through registers, executes the system call's implementation, places a return value in `%rax`, and then returns from the handler to the instruction after the user's `syscall`. The return address—that is, the address of the instruction following the `syscall` instruction, is placed in `%rcx`, and the `iret` instruction at the end of the system call implementation consumed the value in `%rcx` so it knows where to jump back to.*