Exercises for Hands-on 2: UNIX

Pipe Questions

For each question, give a series of UNIX commands that will produce the result.

Question 1. A listing of all processes that you are currently running on the Athena machine you are using, sorted by the command name in alphabetical order (i.e. a process running `acroread` should be listed before a process running `zwgc`). The output should consist *only* of the processes you are running, and nothing else (i.e. if you are running 6 processes, the output should only have 6 lines).

Question 2. The number of words in the file **/usr/share/dict/words** (*) which do not contain any of the letters a, e, i, o, or u (upper and lower case).

Question 3. A 5x6 matrix of entries of alternating 1's and 0's. It should look like this:

```
1 0 1 0 1
0 1 0 1 0
1 0 1 0 1
0 1 0 1 0
1 0 1 0 1
0 1 0 1 0
```

Question 4. A "long" listing of the smallest 5 files in the **/etc** directory whose name contains the string ".conf", sorted by *increasing* file size.

---

1) `$ ps --sort command`
2) `$ egrep -ic '^[^aeiou]+$' /usr/share/dict/words `
3) ` $ yes 1 0 | fmt -u --width=10 | head -n 6`
4) ` $ ls -lSr /etc | fgrep ".conf" | head -n 5`

File Redirection

Now we'd like to explore something slightly different, having to do with file redirection as discussed in section 6.2-6.4 of the paper. The authors explain that the following two commands are functionally equivalent (except that you have to remove the temp file afterwards in the second case). We'll use `athena%` to indicate the command-line prompt:

```
athena% ls | head -   1
athena% ls > temp; head -1 < temp
```

Question 5. Try the above commands in a few different directories. What happens if you try both of the commands in the /etc directory on athena? How else can the second command not produce the same output as the first? Can you think of any negative side effects that the second construction might cause for the user? You might want to think about the commands you used to solve the first four questions and consider their behavior

---

5) The 2nd command writes a text file to the current directory, called `temp`. This operation cannot be done in `/etc` without sudoers privilege, as it is in the root directory. The first command is only reading, not writing in the directory so it is allowed. The user's privileges cause the difference in output, and the writing of an offline file named 'temp' in the working directory is a side effect the user should be weary of. This command would not leave the directory the same as when it had begun.

The UNIX paper authors explain in section 6.3 that a user can type two commands together in parenthesis separated by a semicolon, and redirect the output to a file. The file will then contain the concatenation of the two commands. The example from the paper is roughly:

```
athena% (date ; ls) > temp1 &
```

Note that this example uses the & operator to run the process asynchronously. That is, the shell will accept and run another command without waiting for the first to finish. The authors also mention that one can use the & operator multiple times on one line. For example, we can do almost the same thing:

```
athena% (date & ls) > temp2 &
```

Let's explore the difference between using `;` and & in the examples above. First, we will write a very simple variation on the "yes" program that you encountered earlier on in this assignment. To do so, we will use the "command file" functionality described in section 6.4 of the paper. Most people call these command files "shell scripts", since they are essentially simple scripts that are executed by the shell.

First, start up a copy of emacs editing a new file called "myyes".

```
athena% emacs myyes
```

Now, enter the following lines into your file:

```
#!/bin/sh
echo y
sleep 1
echo n
```

(If you are having trouble with backspace, use the delete key.)

Save the file (Ctrl-x Ctrl-s) and exit emacs (Ctrl-x Ctrl-c). If you don't already know what the echo and sleep commands do, look them up in the man pages. Lastly, make the file executable by running the following command:

```
athena% chmod a+rx myyes
```

No question on this page.

Now let's try running the following two commands:

```
athena% (./myyes ; ./myyes) > temp3
athena% (./myyes & ./myyes) > temp4
```

Note: Some fast multicore machines may occasionally give unexpected answers, where some output is lost - originally unexpected even to the 6.033 staff. If this occurs for you on your home machine, you may want to think briefly about why this occurs.

Question 6: Compare the two temp files. Based on your understanding of file I/O in UNIX, what is going on here, and why? Is this different from what you would expect? (If there is more than one difference between the two files, it is the ordering of the letters y and n that we are interested in).

Question 7: The paper describes the Unix system call interface in some detail. In particular, the read and write system calls do not take the offset as an argument. Why did the Unix designers not include the offset as an argument to read and write? How would an application write to a specific offset in a file?

---

6) The ordering of the printed ‘y’ and ‘n’ characters differs as expected. Temp3 is created synchronously, and prints y ⇒ n ⇒ y ⇒ n (one script after the other); while temp4 is created asynchronously, so the 2nd call to `myyes` begins immediately after the first call. Thus, yielding y ⇒ y ⇒ n ⇒ n.

7) This is because Unix is designed not to have an offset, but rather have the system remember where the last I/O call left off. In other words, reads and write are sequential; the system maintains a pointer to the last byte used, and then will read/write from the immediately following byte. This eliminates the need to have an offset, or index, from the beginning of the file to where the I/O op should occur. This method of I/O eliminates the need for locks on files, as they are inadequate for preventing interference

Looking Around

When you log in, the system sets your current working directory to your home directory, your personal name space where you can create your own files, directories and links. You can view the contents of your current working directory with the ls command (see above). Use the pwd command to learn the absolute path of your current working directory. This will tell you where you are in the directory name space even if you move around in the directories.

```
pwd
```

The output of `pwd` reveals where the Athena administrators store your home directory. For example, `/afs/athena.mit.edu/user` tells us that the your home directory is stored as a user in the Athena namespace.

The stat program reports detailed information about a file including its inode number, link count, file type and other metadata. To use it, type stat followed by a file name at the command prompt. Run stat on your home directory:

```
stat .
```

No question on this page.

Creating Directories and Files

Now create a directory named 6.033-handson2 in your home directory using the `mkdir` command. You can learn more about the mkdir command with `man mkdir`.

```
mkdir 6.033-handson2
```

Use `ls` to verify that the new directory exists. Now change your current working directory to your new 6.033-handson2 directory using the `cd` command and verify that your working directory has changed using `pwd`.

```
cd 6.033-handson2
pwd
```

View the contents of your new directory using `ls -a -l`. ls normally hides the directories "." and "..", but the -a option forces it to show them

.

```
ls -a -l
```

Question 8: Change to the '.' entry in your new directory. What happens to your working directory? Next, change to the '..' entry. What happens to your working directory?

Question 9: Describe a scenario where you might need to use the '.' directory.

---

8) `cd .` moves the working directory to the current working directory, i.e. it does nothing.
`cd ..` moves the working directory to the parent of the current working directory, i.e. it goes up one level in the file system tree.

9) You may need to call the current directory when executing a shell script, such as `./myscript.sh` or if you want to move a file from the parent directory to the current directory, you could say `mv ../foo.txt ./` .

Change your current directory back to your new 6.033-handson2 directory and `stat` the current directory; note the link count. Now create a couple files in your new directory using the touch command and `stat` the directory again.

```
stat .
touch foo bar
ls
stat .
```

Question 10: What has changed in the `stat` output and why has it changed?

Now create a subdirectory baz in 6.033-handson2 and `stat` the directory once more.

```
mkdir baz
stat .
```

Question 11: What has changed in the stat output this time and why has it changed? Why does the link count only change when you create a new directory?

---

10) Nothing changes. Creating new blank files, just creates references to them in the lookup table, places links to them in the directory. When we ask for the stats of '.' we are just asking for links to that directory, not links to its contents.

11) The link count increased by one. This link count is the number of hard links, or number of files which point to its inode. By making a sub-directory, there is a link to `..` or the parent directory, thus adding another link to its link count. Nothing changes when adding files because we are looking at the stats for the current directory, not one of the files we created. Those files point to their own inode where their metadata is stored, as in UNIX a directory's contents are not actually stored in the directory; it is just an abstraction.

Creating Links

The `ln` command can create both hard links and soft (symbolic) links. First `stat` your file foo and read the output information. Then create a hard-link named foo-lnk and stat both foo and foo-lnk.

```
stat foo
ln foo foo-lnk
stat foo
stat foo-lnk
```

Note that everything about foo and foo-lnk is identical except for their names. If you modify foo you will see the modifications in foo-lnk.

```
echo Hello >> foo
cat foo-lnk
```

Now create a symbolic link to foo and note that the symbolic link differs from the original file in several ways. Creating the symbolic link does not increase the link count of foo and the symbolic link does not share an inode with foo.

```
stat foo
ln -s foo foo-slnk
stat foo
stat foo-slnk
```

Question 12: One reason for supporting symbolic links is to allow linking from one disk to another disk, but you can also create a symbolic link to a file on the same disk. Name one advantage and one disadvantage of using symbolic links on the same disk.

---

12) Advantage: can be used to build another abstraction on top of the existing file system's hierarchical tree, i.e, you can restructure the tree with symlinks to make the location of files more intuitive. It may also be advantageous in some contexts that mutating a symlink does not mutate the original file.

Disadvantage: the link is symbolic (a text string of the absolute path), and can be an arbitrary string which points to nothing. A hard link has to point to something in the filesystem, as the file would be discarded when its link-count reaches zero.

Now `cd` to the 6.033 Athena locker with the command:

```
cd /mit/6.033
```

Your home directory is accessible by the path `/mit/YOUR_USERNAME` (replace `YOUR_USERNAME` with your username). Try to change to your home directory with the command:

```
cd ../YOUR_USERNAME
```

Question 13: What happened? Why?

Like your home directory, the 6.033 locker's absolute path is much longer than `/mit/6.033` and `/mit/6.033` is only a symbolic link. You can learn the absolute path name by typing `pwd` or by typing:

```
ls -l /mit
```

Question 14: You can reach the 6.033 locker with the path `/afs/athena.mit.edu/course/6/6.033`. Why does Athena also provide the `/mit/6.033` symbolic link?

Question 15: How would you change the file system to make this command (`cd /mit/6.033; cd ../YOUR_USERNAME`) actually change to your home directory?

---

13) The system created a symbolic link to your home folder inside of `/mit`. This is because your home folder is actually located at `/afs/athena.mit.edu/user/s/t/stwhite` , and `6.033` directory has the path `/afs/athena.mit.edu/course/6/6.033`. BUT, if this was attempted with `tcsh` instead of `bash`, this command would have failed. This is because bash went to the parent of the symbolic link, while tcsh went to the parent for `/mit/6.033`'s absolute path, which was `/afs/athena.mit.edu/course/6`.

14) This symbolic link is also for simplicity to the user. Directly from the SIPB docs on lockers:
“ Lockers organize files and software on Athena, and allow them to be accessed more easily. They eliminate the need to use long pathnames like /afs/sipb.mit.edu/contrib/sipb, and allow you to quickly run programs without needing to know exactly where they are located.”

15) Concretely, use bash instead of tcsh. More abstractly, change the behavior of the `..` directory. With bash, it points to the parent of the symbolic link, while in tcsh it points to its parent in the absolute path. Running `/mit/6.033$  pwd ..` in bash and tcsh shows this difference.

The Search Path

The UNIX shell has a configuration variable named `PATH` that tells the shell where to look in the file system for programs we type on the command line. You can see your `PATH` variable with this command:

```
echo $PATH
```

You can configure your shell to search the current working directory by adding '.' to the `PATH` using these commands:

```
setenv OLDPATH $PATH
setenv PATH .:$PATH
```

Now, `cd` to `/mit/6.033` and run our demo program using the following commands. Your shell will find the "demo" program because it is in your working directory.

```
cd /mit/6.033
demo
```

Oh no, something terrible just happened! Just kidding, the demo program did not actually do anything. Verify that nothing happened with ls.

```
ls -l
```

Question 16: What happened to `ls`? Why isn't it listing files like it did before? (Hint: set your path back to its original state: `setenv PATH $OLDPATH`)

Usually, it is a bad idea of have '.' in your PATH, because it is easy to run the wrong programs by accident. Instead, you can use '.' explicitly to run programs in your working directory like this:

```
./demo
```

---

16) This happens because the $PATH variable is a list of directories that is read right-to-left. After adding the `.` in, the $PATH looks like: `.:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/bin/mh`. So, when we type `ls` the first directory that it looks for that command in is `.`, and in this case, the `/mit/6.033` directory has a simple shell script called `ls` which is found and ran instead.

System Calls

In this final question we take a quick peek the systems calls that a program issues to the operating systems using the program strace. Run the following command:

```
strace ls
```

`strace` shows all the system calls that ls makes. Look for one of the write systems calls, and answer the following question:

Question 17: What does 1 represent in the first argument of write? You may have to consult section 3.6 of the Unix paper to answer the question.

Question 18: How long did this assignment take you to complete up to this point?

---

17) the first arg in write, `write(1, "1994  2000  2006  2012\t    2017\t"..., 521994  2000  2006  2012    2017 lec     recguides) = 52` is the file descriptor, or 'filep', which is a contextual variable that references the file in subsequent read/write operations.

18) Like 6 hours, but I took the time to geek out and learn some other unix commands and read up on tcsh vs bash