CMPS 11
Intermediate to Programming
Lab Assignment 2

We have three goals in this assignment: to learn about file permissions in Unix, to get a basic introduction to the Andrew File System and it's directory access control commands, and to learn how to redirect program input and output to a file.

Unix File Permissions

Every file in a Unix system has a unique owner, and an associated group. The owner of a file is the user who created it, and the group is a collection of other users who may have access to the file. Each file also has a set of permission flags which specify separate *read*, *write*, and *execute* permissions for *User* (i.e. the owner), *Group*, and *Other* (everyone else with an account on the system.) All of this information is displayed by the `ls` command with the `-l` option. To run some examples, log on to your UCSC IC Unix account, and use your favorite editor to create a couple of text files in your cs11 directory, which you created in lab1. The contents of each file is unimportant. We will refer to them here as `junk1` and `junk2`. Do `ls -l` (that's the letter "`l`" not the number "1") at the command prompt, and you will see something like the following.

```
total 2
-rw-r--r--. 1 ptantalo users 126 Jan 11 18:23 junk1
-rw-r--r--. 1 ptantalo users  61 Jan 11 18:24 junk2
```

The first line `total 2` gives the number of files. Since you probably have other files in your cs11 directory, you will likely see a longer listing. Reading from left to right along the second line above we have:

`-rw-r--r--.`	: *permission flags* for this file (explained below)
`1`	: the *number of links* (I won't explain this, so don't worry about it)
`ptantalo`	: the *User* (owner) for this file (your cruzid, when you do it)
`users`	: the *Group* for this file
`126`	: the *size* of the file in bytes
`Jan 11 18:23`	: the *date and time* (military) of the most recent modification
`junk1`	: the *name* of the file

The permission flags are read from left to right as follows:
position 1: the directory flag: `d` for a directory, and `-` for a file
positions 2-4: read, write, execute permissions for User (owner)
positions 5-7: read, write, execute permissions for Group
positions 8-10: read, write, execute permissions for Other
position 11: should always be . indicating selinux context (not covered here, so ignore it)

The meanings of the values appearing in positions 2-10 are:
`-` in any column means that the flag is turned off
`r` in positions 2, 5, or 8 means the file is readable by User, Group, or Other (respectively)
`w` in positions 3, 6, or 9 means the file is writeable by User, Group, or Other (respectively)
`x` in positions 4, 7, or 10 means the file is executable by User, Group, or Other (respectively)

Position:	1	2	3	4	5	6	7	8	9	10	11
	directory flag	read User	write User	execute User	read Group	write Group	execute Group	read Other	write Other	execute Other	SELinux .

Thus `-rw-r--r--.` indicates a file that is readable, writeable, and not executable by its User; readable, not writeable and not executable by its Group; readable, not writable and not executable by Others on the system. The owner of a file can change its permissions by use of the `chmod` command. For instance

```
% chmod go+w junk1
```

has the effect of adding write permission to Group and Other for the file `junk1`. (As always, `%` represents the Unix command prompt.) Doing `ls -l` now gives

```
-rw-rw-rw-. 1 ptantalo users 126 Jan 13 18:23 junk1
-rw-r--r--. 1 ptantalo users  61 Jan 13 18:24 junk2
```

As you can see, the usage of `chmod` is `chmod mode filename`. In the above example, the permissions mode `go+w` is of the form (`who`)(`operator`)(`permission`), where

`who` is some combination of:
`u` : User
`g` : Group
`o` : Other
`a` : All (User, Group, and Other)

`operator` is one of:
`+` : add specified permission
`-` : delete specified permission

`permission` is some combination of:
`r` : read permission
`w` : write permission
`x` : execute permission

Do the commands `chmod go-w junk1` and `chmod a+rx junk2`, then try to predict what permission changes will take effect. Check your answer by doing `ls -l`.

Another convenient way to specify the permissions mode for a file is by giving `chmod` a sequence of 3 octal digits (0-7). Each octal digit is equivalent to 3 binary digits, and thus we are giving `chmod` a sequence of 9 binary digits, each bit corresponding to one of positions 2-10 in the string of file permission flags. For instance, the octal sequence 645 is equivalent to the binary sequence 110 100 101, which is in turn equivalent to the permission flags `rw-r--r-x` (including only positions 2-10). Do `chmod 467 junk1` and `chmod 721 junk2`, and try to predict the permission changes which result. Check your answer by doing `ls -l`. See http://www.robotroom.com/NumberSystems4.html for a description of octal to binary (and other) conversions.

Read permission on a file simply means that the specified user can view its contents (using `more`, `less` or `cat` for instance). Write permission means that the specified user can modify the contents of the file (using editing commands like `ed`, `vi`, `emacs`, `pico`, or other file manipulation operations.) If you have followed the above instructions, then files `junk1` and `junk2` will have permissions `r--rw-rwx` and `rwx-w---x` respectively (again including only positions 2-10). Thus if you (the file's owner) try to modify `junk1` (using an editor like `vi` for instance) you will get the error message: `Permission denied`.

Execute permission means that the file is a program that can be run by the specified user. To run an executable file in Unix, one simply types its name at the command prompt. Type `junk1` then `junk2`. You'll see that `junk1` gives the `Permission denied` error, while `junk2` does not. Instead, you will most likely see each line of `junk2` printed out with the error message `command not found` next to it. Here is my output.

```
junk2: line 1: lskdjflsks: command not found
junk2: line 2: jdsflfksdksdf: command not found
junk2: line 3: iouwriuoerw: command not found
```

When you attempt to execute `junk2` the command interpreter reads each line of the file, then tries to parse it as a Unix command, which may or may not succeed. Thus when a file has 'executable' permission, it does not mean that the file is able to be executed successfully, but rather that the command interpreter will *try* to execute it for the specified users. In fact, *all Unix commands are nothing more than the names of executable files*, although most such files contain binary machine language instructions instead of text.

An executable text file that contains Unix commands is often called a Shell Script. ('Shell' because that's another name for a Unix command interpreter, and 'Script' since it is a text file and not binary.) Create a new file with your favorite text editor called `prog1` containing the following lines.

```
# prog1
# this is a shell script
pwd
cp prog1 prog2
ls -l
more prog1
```

After you exit your editor do `chmod 700 prog1` to make it executable. Obviously the next thing to do is just type `prog1` to run the script, but before you do, take a moment to study the commands in the file and predict exactly what it will do. Note that anything on a line after the `#` symbol is a comment and is ignored by the shell.

The Andrew File System

The Andrew File System (AFS) is a distributed networked file system developed by Carnegie Mellon University in the 1980s. The UCSC Instructional Computing (IC) Unix Timeshare servers (unix.ic.ucsc.edu) use AFS to manage all directories and files associated with the Linux computing environment, which includes your UCSC computer account. AFS commands are not standard Unix however, so the material in this section will not necessarily pertain to other Unix systems on which you may have an account, such as the Baskin SOE servers, or your personal Linux or Mac OS X machines.

AFS provides access control levels that are finer and more flexible than the user/group/other permissions described in the previous section, but they work at the level of *directories*, not *files*. In a standard Unix system, the file permissions described above would operate on directories in the very same way that they do on files. This is not the case under AFS, where directory permissions are controlled by an *Access Control List* (ACL). These ACLs take precedence over the Unix permissions assigned to directories via `chmod`. In fact, under AFS, `chmod any_mode any_directory` has no effect on the actual access rights for that directory (although it would appear to do so, if you look at the output of `ls -l`.) In AFS, each directory has seven distinct access rights, each of which may be either on or off.

Name	Code	Permission to
read	`r`	View the contents of the files in a directory
lookup	`l`	Lookup filenames and examine the ACL of a directory

insert	`i`	Add new files and subdirectories to a directory
delete	`d`	Remove files from a directory
write	`w`	Modify file contents and change file attributes via `chmod`
lock	`k`	Lock files (not explained here, so don't worry about it)
administer	`a`	Change the ACL of a directory

The main AFS command is `fs`, which has a number of subcommands. (Type `fs help` to see a complete listing of all the subcommands to `fs`.) Of these, we are primarily interested in two: `listacl` which prints out an ACL, and `setacl` which modifies an ACL. Their usage is:

```
% fs  listacl  directory_name
% fs  setacl  directory_name  user_or_group_name  rights
```

Commands `listacl` and `setacl` can be abbreviated as `la` and `sa` respectively. For example, create a new subdirectory in `cs11` called `junk3` (using `mkdir`), then examine it's ACL by doing `fs la junk3`. You will see something like

```
Access list for junk3 is
Normal rights:
  system:anyuser rl
  ptantalo rlidwka
```

This indicates that the group `system:anyuser`, consisting of all users of AFS worldwide, has read and lookup rights. The individual user `ptantalo`, which will be your cruzid when you do this, has all rights. The ACL you get for `junk3` may be slightly different, depending on the ACL of its parent `cs11`. Generally a newly created directory will inherit the ACL of its parent. Now modify the ACL for `junk3` by doing `fs sa junk3 system:anyuser none`, then list it again using `fs la junk3`. You will see something like

```
Access list for junk3 is
Normal rights:
  ptantalo rlidwka
```

As you can see, `none` means to remove all rights. Similarly, `all` means to add all rights. For instance, if you type `fs sa junk3 system:authuser all`, then view the ACL. You should have

```
Access list for junk3 is
Normal rights:
  system:authuser rlidwka
  ptantalo rlidwka
```

The group `system:authuser` consisting of local users (at UCSC) now has all rights. For more on the Andrew File System go to https://en.wikipedia.org/wiki/Andrew_File_System.

Redirection of Program Input/Output

As mentioned in class, all running Java programs are equipped with the three data streams: stdin, stdout, and stderr. In fact the same is true of *all* Unix processes. By default, stdin represents the sequence of characters typed at the keyboard as program input. Likewise stdout and stderr represent program output, which is ordinarily sent to the terminal window. The Unix redirect operators `<`, `>`, `>>`, `>&`, and `>>&` can be used to redirect these streams to flow to/from files rather than to their defaults. Their general usage is as follows.

`command < file1`	Read standard input from `file1`. `file1` should contain exactly those characters that would ordinarily be typed at the keyboard.
`command > file2`	Write standard output to `file2` instead of the terminal. `file2` will be **created** if it does not already exist, and will be **overwritten** if it does exist.
`command >> file3`	Append standard output to `file3`. `file3` will be **created** if it does not already exist, and will be **appended to** if it does exist.
`command >& file4`	Write standard error to `file4`.
`command >>& file5`	Append standard error to `file5`.

Try this out on some Unix commands, such as:

```
% pwd > junk4
% ls -l > junk5
% ls -l >> junk5
```

Try to predict the contents of the new files `junk4` and `junk5` before viewing them. Compile the programs `HelloWorld.java` and `HelloWorld2.java` in your lab1 subdirectory, then move the executables `HelloWorld.class` and `HelloWorld2.class` into the directory cs11. If you don't know how to do this study the Unix `mv` (move) command in one of the tutorials, or do `man mv`, or Google "unix mv". Run the `HelloWorld` and `HelloWorld2` programs as follows:

```
% java HelloWorld > junk6
% java HelloWorld2 >> junk6
```

then predict the contents of the new file `junk6`. Recall that `HelloWorld4.java` from the class webpage was interactive, in that it read input from stdin. Save this file to cs11 and compile it. Prepare a file called `junk7` containing an appropriate line of text. Do

```
% java HelloWorld4 < junk7
% java HelloWorld4 < junk7 > junk8
```

Predict and then view the contents of `junk8`.

What to turn in

All the exercises you've done so far have been practice. Delete all the files and directories you've created up to now. Perform the following steps exactly as stated, and in the given order so that the file you end up with is correct.

1. Create a subdirectory called `lab2` within your `cs11` directory, and `cd` into it.
2. Create two subdirectories called `public` and `private` within `lab2`. Set their ACLs as indicated in the following table. Here `foobar` stands for your username.

	public	private
foobar	all	
system:anyuser	rl	none
system:authuser	rlid	none
system:operator	rlidwk	none

3. Copy the file `HelloWorld.java` from lab assignment 1 to your lab2 directory. Edit it so that it prints out the single line "`Hello, lab2!`". Compile it, creating the file `HelloWorld.class`. Delete the file `HelloWorld.java` from your lab2 directory. Do not delete `HelloWorld.class`.

4. Create a text file in `lab2` called `prog` containing the following shell script.

```
# prog
# shell script for lab2
pwd > result
echo >> result
fs la public >> result
echo >> result
fs la private >> result
echo >> result
ls -l >> result
echo >> result
java HelloWorld >> result
```

5. Use `chmod` to give User (yourself) read, write, and execute permissions on the file `prog`. Group and Other should have no permissions.
6. Run the shell script `prog`. Notice that a new file called `result` is created.
7. If you followed instructions to the letter, the file `result` will contain exactly 21 lines. Submit `result` with no further changes to the assignment name lab2.

This is a considerably longer assignment that lab1 so please start early and get help in lab sessions.