6.033 Computer System Engineering

Spring 2009

Hands-on 2: The UNIX Time-Sharing System

This hands-on assignment is due at the beginning of Recitation 7. Before attempting this hands-on, you should read *The UNIX Time-Sharing System*, which is also assigned for this recitation.

I. Warmup

If necessary, you may want to briefly re-read sections 5.2 and 6.2-6.4 of the paper to refresh yourself on the concepts of pipes and I/O redirection.

Note: Since the GNU Linux system commands function slightly differently to their UNIX counterparts, please ensure that your answers work on a Solaris Athena machine such as athena.dialup.mit.edu.

We'll start off with an extremely simple example that most of you are probably familiar with already:

```
athena% cd /bin
athena% ls -1 | more
```

Here, we are first changing into the /bin directory, which contains many of the executable commands for the system. The command `ls -1` gives us a listing of all the files in the current directory with one file per line. (Note that `-1` is the numeral "one", not the letter "L".) We then pipe the output from `ls` to the command `more`, which displays the results one page at a time. (Press the space bar to show the next page.) You can refer to the manual pages for `ls` and `more` to see more details and options for each command. Manual pages let you read information about various commands on UNIX systems; to use them, run

```
athena% man command
```

where *command* is the command you are interested in. If you are unfamiliar with manual pages, you may want to try running

```
athena% man man
```

for information on the man command itself. Keep in mind that the manual pages for basic commands vary from system to system (much as the commands themselves do).

Now, try this:

```
athena% cd /bin
athena% ls -1 | grep p | more
```

This runs the same `ls -1` command, but only lists the executable files which happen to contain the letter "p" somewhere in their names.

The point here is to observe that you can chain together multiple commands using the pipe character (|), and the output from each command will be passed to the input of the next, in left-to-right order. This allows you to treat any command that uses standard input and output as a primitive from which you can build more complex and useful commands.

II. Building Blocks

Now, we'd like you to figure out on your own how to solve some problems by chaining different commands together.

If you aren't already familiar with these commands, you may want to briefly skim through their man pages to familiarize yourself with what they do. You will probably need to use some of the options for the different commands in order to solve these problems.

Here are the commands you may find useful:

```
cat
fmt
grep
head
ls
ps
sort
tail
top
wc
yes (*)
```

(*) On some Athena machines, the `yes` command isn't available. However, if you are doing this assignment on Athena you can use the command `gyes`, which is functionally equivalent. `gyes` is located in the "gnu" locker, so before you can use it you need to add the locker. You can do so with the following command:

```
athena% add gnu
```

(If you are curious about Athena's locker system, you can run `man lockers` for more information. The command `whichlocker` can be used to determine which locker contains a given command. The `whichlocker` command itself resides in the "outland" locker. For more info on other lockers, look at this SIPB article)

Once you have added the gnu locker, you can use `gyes` instead of `yes`. Some Athena machines seem to lack the manual pages for both `yes` and `gyes` (hereafter referred to just as `yes`). In case the man pages are missing on the machine you are using, here is a brief description of what `yes` does. `yes` is very

simple; it just outputs a string repeatedly until killed. It takes either one or zero arguments; if you give it a string as an argument it will output that string until it is killed (you can kill a process by pressing control-c). If you give it no arguments, it will output the string "y" until it is killed, which explains why it is named `yes`.

III. Questions

For each of the outputs listed below, find one sequence of commands connected by pipes that produces the output. For each problem, turn in the command sequence that you used to generate the requested output. (Do NOT turn in the output itself.)

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

   [* Note: On some Unix/Linux systems, the dictionary has the filename **/usr/share/dict/words**]

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
```

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

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):

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

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.

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 &
```

See if you can figure out for yourself what exactly the difference is between using ";" and "&" in the examples above.

Let's explore this for ourselves. 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
```

Save the file and exit emacs. 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
```

Now let's try running the following two commands:

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

Note: If you're using a fast multicore machine for this handson, please ensure that you run the latter command on athena.dialup.mit.edu instead. 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.

6. Lastly, 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).