DPS918 Lab 2

DPS918 Lab 2: Notes

Some important notes on this Lab:

(1) You do NOT have to do this whole Lab in one sitting! Any completed sections will be tracked.

(2) You may abort the Lab at any time by pressing Ctrl-c but you may lose any work done on an incomplete section.

(3) Instructions for each page will be displayed at the bottom of the screen.

(4) The answers expected in this Lab are derived from the Lab itself and the course notes. The course notes should be used as a reference, if required. Using Google or another search engine may provide answers that are not accepted by the Lab.

(5) This Lab is maintained by Les Czegel. Please try to resolve any problems with your instructor who will determine if Les should be contacted.

DPS918 Lab 2: Selection Menu

-----------------------------------------------------------------------------------------------------------------------------------------------------

Available selections:

1 Redirection And Pipes

2 Linking Files And Directories

3 Command Summary

4 Linux Processes

5 Regular Expressions Using grep

6 sed And awk

7 Interactive Shell Environment

8 Submit DPS918 Lab 2

The highlighted lines above indicate the incomplete parts of the Lab.

There is a total possible mark of 3, with 0.5 marks deducted for each incomplete part. Late marks will be deducted at the rate of 0.3 marks (10%) per day for Labs submitted after midnight of May 26 2017.

Available selections:

1 Standard File Descriptors

2 Redirecting Standard Output

3 Redirecting Standard Input

4 Redirecting Standard Error

5 Connecting Commands With Pipes

6 Putting it All Together

7 Review Exercise

Note that the highlighted lines indicate the incomplete sections of this part of the Lab.

The Review Exercise can only be done when the other sections have been completed.

Standard File Descriptors

When a Unix shell starts a command, it opens three connections for that command. These three connections are called the 'standard file descriptors'. Many commands use these three connections for input and output.

These are the three standard file descriptors:

Number Name Short Name

0 Standard Input stdin

1 Standard Output stdout

2 Standard Error stderr

DPS918 Lab 2.1: Standard File Descriptors

If you're using the shell interactively, then normally all three of the standard file descriptors are connected to your terminal. This means that when the program needs input, it gets it from the keyboard, and when it displays output or error messages, they appear on the screen.

Number Name Short Name

0 Standard Input stdin

1 Standard Output stdout

2 Standard Error stderr

Enter the 'date' command.

$ date

Thu May 18 07:40:24 EDT 2017

Note that the date appeared on the screen. This is because the standard output (stdout) of the date command was connected to the screen.

This connection between the stdout of the command and your screen is set up automatically by the shell.

Now try using the 'cat' command with no arguments.

IMPORTANT!: Once you enter 'cat', you will be able to type text. Try entering several lines, pressing ENTER after each line. When you are done, press Ctrl-d on a line by itself to continue with the Lab.

$ cat

abc

abc

123

123

Notice that the 'cat' command just copied whatever you entered? That is because 'cat' with no arguments just takes whatever is

entered on stdin (the keyboard) and copies it to stdout (the screen).

One of the features of Unix is that almost everything is treated as a file.

Enter the command 'who am i' to find out the file name for your terminal (look in the second column of output).

$ who am i

'who am i' reported that your terminal device is (second column of the output). That is a relative pathname based on the device directory /dev, so the full pathname for your terminal is /dev/

Check to make sure that file exists by getting a detailed listing of it.

$ ls -l /dev/

total 0

crw-rw---- 1 root video 10, 175 May 18 07:28 agpgart

crw------- 1 root root 10, 235 May 18 07:28 autofs

drwxr-xr-x 2 root root 100 May 18 07:28 block

prw------- 1 root root 0 May 18 07:27 blog

drwxr-xr-x 2 root root 80 May 18 03:27 bsg

crw------- 1 root root 10, 234 May 18 07:27 btrfs-control

lrwxrwxrwx 1 root root 3 May 18 07:28 cdrom -> sr0

drwxr-xr-x 2 root root 3080 May 18 03:28 char

crw------- 1 root root 5, 1 May 18 07:28 console

lrwxrwxrwx 1 root root 11 May 18 07:27 core -> /proc/kcore

drwxr-xr-x 4 root root 100 May 18 07:27 cpu

crw------- 1 root root 10, 62 May 18 07:28 cpu\_dma\_latency

drwxr-xr-x 6 root root 120 May 18 07:28 disk

lrwxrwxrwx 1 root root 3 May 18 07:28 dvd -> sr0

lrwxrwxrwx 1 root root 13 May 18 07:27 fd -> /proc/self/fd

crw-rw-rw- 1 root root 1, 7 May 18 07:28 full

c

Now use the 'file' command to see what kind of a file /dev/ really is.

$ file /dev/

/dev/: directory

As you saw, /dev/ is a 'character special' file, which means that it is a device which accepts one character at a time and can send one character at a time.

It is possible to use the shell to connect the standard file descriptors to other files or devices, or even to other commands. The next sections of this Lab discuss how this is done.

Redirecting Stdout

The standard output may be redirected with the symbol '>' followed by the name of the file which will contain the output.

You can think of the '>' character as being an arrow pointing from the command to the file, showing that the output from that command will go into the file.

Let's redirect the output of the 'date' command into the file 'today'.

$ date > today

Now display the contents of the 'today' file.

$ cat today

Fri May 19 00:00:41 EDT 2017

Good -- the file 'today' contained the date from the command 'date > today'

Next, try redirecting the output from 'cal' into a file named 'may'.

$ cal > may

Now display the contents of the 'may' file.

$ cat may

May 2017

Su Mo Tu We Th Fr Sa

1 2 3 4 5 6

7 8 9 10 11 12 13

14 15 16 17 18 19 20

21 22 23 24 25 26 27

28 29 30 31

Great!

Let's redirect the output of the 'whoami' command into that same file ('may').

$ whoami > may

Now display the contents of the 'may' file.

$ cat may

hhuang66

Notice that the output from the 'whoami' command replaced the output from the

'cal' command!

This is called overwriting. The symbol '>>' will append (add) to a file instead

of overwriting. You can think of the symbol as a double arrow from the command

to the file, meaning 'add the output from the command to the END of the file'.

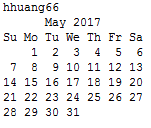
Try using '>>' to append the output from 'cal' to the file 'may' without overwriting

what is already there.

$ cal >> may

Once again, display the contents of the 'may' file.

$ cat may



Good -- the output from 'cal' was appended to the information already in the 'may' file.

Proceed to the next section on redirecting stdin.

DPS918 Lab 2.1: Redirecting Standard Input Page 1/6

-------------------------------------------------------------------------------------

Redirecting Stdin

You can use '<' to redirect standard input, just like you used '>' to redirect standard output.

You can also think of the '<' character as being an arrow pointing from the file to the command, meaning that the information will flow from the file into the input of the command.

First, let's create a little file that we can use for testing.

Let's put the current month's calendar into the file 'caltest'.

$ cal >caltest

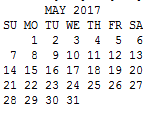
The translate command 'tr' takes its input from stdin, which is usually the terminal.

Use 'tr' to translate the contents of 'caltest' to uppercase.

The command to translate to uppercase is 'tr "[a-z]" "[A-Z]"' -- add '< caltest' to make the command take its input from the file 'caltest'.

Note that the 'tr' command can usually be used without the square brackets, but some older versions of Unix require them.

$ tr "[a-z]" "[A-Z]" < caltest



You should have seen this month's calendar appear on your screen in uppercase.

Some commands take their input from stdin if there is no file argument on the command-line. For example, you can give a filename argument to the 'cat' command; but if you do not include a filename argument, then the input is taken from stdin

(as you found out earlier).

This means that 'cat < caltest' and 'cat caltest' do the same thing.

Enter 'cat < caltest' now to prove this.

$ cat < caltest



Good!

Proceed to the next section on Redirecting stderr.

DPS918 Lab 2.1: Redirecting Standard Error

Redirecting Stderr

It is often useful to be able to separate error messages from a program's normal output. For example, you may want to redirect the output from a command into a file, but still have any error messages go to the screen so that you can tell when something is wrong.

Let's intentionally create an error while redirecting the output.

Redirecting Stderr

It is often useful to be able to separate error messages from a program's normaloutput. For example, you may want to redirect the output from a command into a file, but still have any error messages go to the screen so that you can tell when something is wrong.

Let's intentionally create an error while redirecting the output.

$ cal 1963 8 > caltest

cal: illegal month value: use 1-12

Notice that the error message appeared on the screen even though the output of the command was redirected. This is why standard error is separate from standard output.

To redirect standard error we use the same symbols as we use for redirecting standard output -- '>' and '>>' -- except that we add the file descriptor number for stderr (2) directly in front of the symbol, like this:

2> file (overwrites)

2>> file (appends)

This time, let's intentionally create an error while redirecting stderr into a file

$ cal 1963 8 2>caltest

Now display the contents of the 'caltest' file to see what error messages were produced.

$ cat caltest

cal: illegal month value: use 1-12

The 'caltest' file contained the error message from the 'cal 1963 8' command.

Good! Now proceed to the section on Connecting Commands with Pipes.

DPS918 Lab 2.1: Connecting Commands With Pipes

Connecting Commands with Pipes

In addition to connecting stdin, stdout, and stderr to files, it is possible to

connect the stdout of one command to the stdin of another.

This is done using the pipe symbol: '|'.

For example, we can take the standard output of 'cal 2017' and connect it to the standard input of 'grep' so that we can specify which lines to output on the screen.

Try it out...

$ cal 2017 | grep er

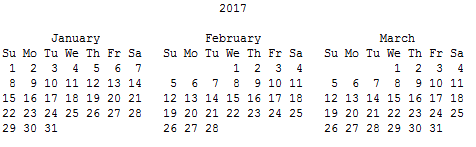
July August September

October November December

Good. Notice that the commands are connected from left-to-right. The standard output of 'cal' was connected to the standard input of 'grep'.

Let's try using 'head' to view the first 10 lines of the output from 'cal 2017'.

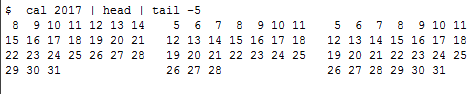
$ cal 2017 | head



Good!

Now let's add 'tail -5' to get just the last 5 lines of those first 10 lines.

$ cal 2017 | head | tail -5



Great.

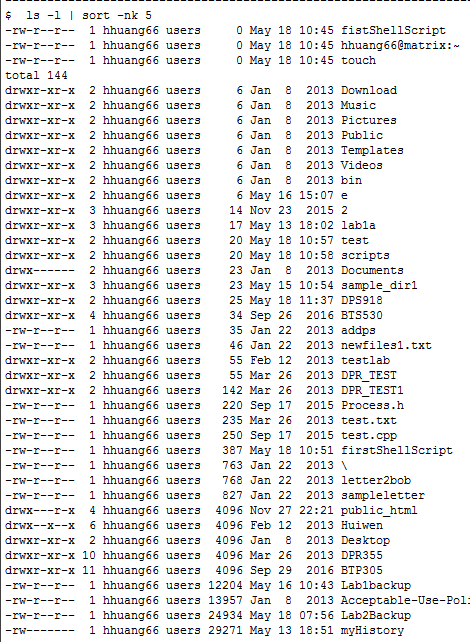
Now we'll try some other examples...

Let's use 'ls -l' to get a detailed list of files in your home directory, and then use 'sort' to sort the list by file size.

For the 'sort' command, we'll need the arguments '-k 5' to sort on the 5th column

(file size) and '-n' to sort numerically instead of alphabetically.

$ ls -l | sort -nk 5



DPS918 Lab 2.1: Connecting Commands With Pipes

Good. The files were sorted by size, with the largest at the bottom. But what if we only want to see the 2 largest files?

Try the same command, but add 'tail -2' to see only the last two files listed.

$ ls -l | sort -nk 5 | tail -2

-rw-r--r-- 1 hhuang66 users 24934 May 18 07:56 Lab2Backup

-rw------- 1 hhuang66 users 29271 May 13 18:51 myHistory

Good. That concludes the section on pipes.

Next, proceed to the section 'Putting it All Together'.

DPS918 Lab 2.1: Putting It All Together

Putting it All Together

Redirection and pipes are often used together, but it can be tricky getting the syntax right.

There are three important rules to remember:

(1) Redirection only connects commands to files

(2) Pipes only work between commands (|).

(3) Data goes into the left side of a pipe and flows out the right side.

Redirection only connects commands to files

One common mistake is to try to use the redirection symbols between commands. Remember that >, >>, < only connect commands to files.

For example, this will work:

cal -y > today

These will not work:

cal -y > sort -n

sort -n < cal -y

Pipes only work between commands

In the same way, the pipe symbol '|' can only be used to connect commands.

For example, this will work:

cal -y | sort -n

This will not work:

cal -y | outputfile

Data goes into the left side of a pipe and flows out the right side

Pipes are written left-to-right. In this example:

a | b | c

The stdout of command 'a' is connected to the stdin of 'b', and the stdout of

'b' is connected to the stdin of 'c'.

If you're going to combine file redirection with pipes, then any input redirection needs to be connected to 'a' and any output redirection needs tobe connected to 'c'.

Data goes into the left side of a pipe and flows out the right side

Therefore you could do something like this:

a <inputfile | b | c

But not:

a | b | c <inputfile

The problem with the second example is that the command shows the stdin of

'c' connected to both 'inputfile' and the stdout of 'b'.

Keep those three warnings in mind whenever you use redirection and pipes and

you will avoid the most common mistakes!

(1) Redirection only connects commands to files.

(2) Pipes only work between commands (|).

(3) Data goes into the left side of a pipe and flows out the right side.

Proceed to the Review section.

DPS918 Lab 2.1: Review Exercise

Review Exercise

The purpose of this section is to reinforce your skills with issuing UNIX commands covered in the previous sections.

You will be asked to enter a single UNIX command for the following situations. If you are unable to successfully issue the UNIX command after several attempts, a hint will be provided.

Question 1

Redirect the output of 'cal -y' into a new file called 'review1'

cal -y > review1

Question 2

Redirect the output of 'date' so that it is added to the end of the file

'review1'.

date >> review1

Question 3

Display a listing of the directory '/etc' one screen at a time.

(Remember to press <q> when you want to quit).

ls -l /etc |more

Question 4

The file 'threelines' contains 3 lines of text. Display the middle line only on the screen. You are not allowed to use the more command.

head -2 threelines | tail -1

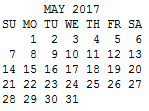
$ head -2 threelines | tail -1

Line Two

Question 5

Display the current month using 'cal', translating to uppercase using 'tr'.

$ cal | tr 'a-z' 'A-Z'



Question 6

Run the command 'mkdir old' and redirect any error messages into the file 'errors'.

mkdir old2>errors

Question 7

Search the file 'data' for all of the lines that contain the pattern 'linux' and put those lines in the file 'matches'.

grep linux data > matches

Question 8

The file 'numbers' contains a list of numbers. Write a command to place the largest one of those numbers into the file 'largest' (there should be nothing else in that file). Do not use the 'head' command in your answer.

sort -n numbers | tail -1 > largest

Congratulations!

You have successfully completed the Review Exercise!

Available selections:

1 Standard File Descriptors - Completed

2 Redirecting Standard Output - Completed

3 Redirecting Standard Input - Completed

4 Redirecting Standard Error - Completed

5 Connecting Commands With Pipes - Completed

6 Putting it All Together - Completed

7 Review Exercise - Completed

Note that the highlighted lines indicate the incomplete sections of this part of the Lab.

The Review Exercise can only be done when the other sections have been completed.

-------------------------------------------------------------------------------

Enter a menu selection or 'q' to quit: 2

DPS918 Lab 2.2: Linking Files & Directories

-------------------------------------------------------------------------------

Available selections:

1 Hard Links

2 Symbolic Links

3 Linking To Directories

4 Review Exercise

Note that the highlighted lines indicate the incomplete sections of this part of the Lab.

The Review Exercise can only be done when the other sections have been completed.

DPS918 Lab 2.2: Hard Links

Linking Files

The 'ln' command creates links to files. A link may simply be considered an additional name for a file. The link may be in the same directory or a different directory than the original filename.

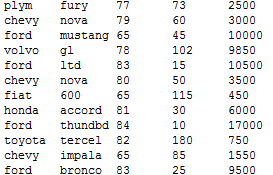
There are two types of links. In this section, we will look at hard links.

In the next section, we will look at symbolic links.

A file called 'cars' has been created in your current directory to demonstratelinking.

View the contents of this file:

$ cat cars



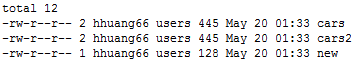
In order to create a link, we use the 'ln' command followed by the current name of the file, followed by the additional name we wish to give it.

Create a link to the 'cars' file called 'cars2':

$ ln cars cars2

Look at detailed information about the 'cars' and the 'cars2' files in the current directory:

$ ls –l



The statistics for 'cars' and 'cars2' are identical. In fact,

'cars' and 'cars2' are two names for the same physical file.

The 'number of links' shown after the permissions was 2 for both filenames. This means that the file has two links to it, one with the name 'cars' and one with the name 'cars2'.

Note that creating an additional link to a file, even if in a different directory, is not a method of backing up the file. If the hard drive fails, then the physical file can still be corrupted or destroyed regardless of the number of links.

Try the '-i' option of the ls command:

$ ls -i

798727 cars 798727 cars2 798726 new

The '-i' option of ls gives the 'inode' number of filenames. 'inode' stands for information node, and contains all the information about a file, including permissions, owner, group, create date-time,

last modified date-time, and other pieces of information.

Each file requires a unique inode. As you saw, the inodes for 'cars' and 'cars2' were the same, since they are different names for the same file.

Now delete the file 'cars':

$ rm cars

Now take a look at the remaining files:

$ ls -l

total 8

-rw-r--r-- 1 hhuang66 users 445 May 20 01:33 cars2

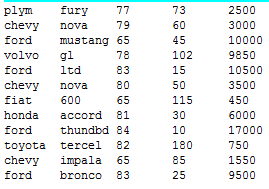
-rw-r--r-- 1 hhuang66 users 128 May 20 01:33 new

You saw that 'cars2' is still there. A file is not considered deleted until ALL hard links are removed.

Note that files are not actually physically deleted from a hard drive, and may be recovered, until their specific inodes are overwritten with new data.

To see that the file is still there, display the contents:

$ cat cars2



Now rename the 'cars2' file to 'cars', and we're back where we started:

$ mv cars2 cars

You may now proceed to the next section about 'Symbolic Links'.

DPS918 Lab 2.2: Symbolic Links

Symbolic links give another name to a file, in a way similar to hard links. But a file can be deleted even if there are remaining symbolic links.

Symbolic links are also called soft links. In order to give a file a symbolic link, the 'ln' command is used with the '-s' option.

Give 'cars' the symbolic link 'cars3':

$ ln -s cars cars3

Now take a look at the files:

$ ls –l

total 8

-rw-r--r-- 1 hhuang66 users 445 May 20 01:55 cars

lrwxrwxrwx 1 hhuang66 users 4 May 20 02:02 cars3 -> cars

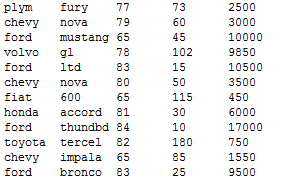
-rw-r--r-- 1 hhuang66 users 128 May 20 01:55 new

Notice that the first character (file-type) for 'cars3' was 'l', which means it's a symbolic link. And in the name field, you saw 'cars3 -> cars'.

The symbolic link 'cars3' DOES NOT point to the inode that 'cars' is pointing to. Instead, 'cars3' points at the name 'cars'.

Let's display the file pointed to by 'cars3':

$ cat cars3



'cars3' will continue to point to the name 'cars', even if 'cars' is deleted.

Let's delete 'cars' and see what happens:

$ rm cars

Now take a look at the remaining files:

$ ls -l

total 4

lrwxrwxrwx 1 hhuang66 users 4 May 20 02:02 cars3 -> cars

-rw-r--r-- 1 hhuang66 users 128 May 20 01:55 new

As you saw, the file cars has been removed, but 'cars3' is still pointing to the name 'cars'.

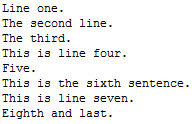
If we now create a file called 'cars', 'cars3' will automatically point to the new file.

To see how this works, rename the file 'new' to 'cars':

$ mv new cars

Now display the contents of 'cars':

$ cat cars



Now display the contents of 'cars3':

$ cat cars3

Line one.

The second line.

The third.

This is line four.

Five.

This is the sixth sentence.

This is line seven.

Eighth and last.

You may now proceed to the next section about 'Linking to Directories'.

DPS918 Lab 2.2: Linking Files & Directories

Available selections:

1 Hard Links - Completed

2 Symbolic Links - Completed

3 Linking To Directories

4 Review Exercise

Note that the highlighted lines indicate the incomplete sections of this part of the Lab.

The Review Exercise can only be done when the other sections have been completed.

Directories can be linked only symbolically, except by the system administrator.

To demonstrate, start by displaying the contents of the directory

'~dps918/2017b/sample\_dir3/linked\_directories/sample\_files':

$ ls -l ~dps918/2017b/sample\_dir3/linked\_directories/sample\_files

total 152

-rw-r--r-- 1 dps918 users 445 May 6 04:42 cars

-rw-r--r-- 1 dps918 users 135838 May 6 04:42 phones

-rwx--x--x 1 dps918 users 9571 May 6 04:42 showtree

These are the same 'cars' and 'new' files that we've already seen earlier in this Lab.

It's very cumbersome to type a long pathname such as ~dps918/2017b/sample\_dir3/linked\_directories/sample\_files.

If we were often accessing the files in this directory, we could create a link.

Let's create a symbolic link to the directory, and call it 'linkdir':

$ ln -s ~dps918/2017b/sample\_dir3/linked\_directories/sample\_files linkdir

Now we can refer to 'linkdir' instead of ~dps918/2017b/sample\_dir3/linked\_directories/sample\_files.

Copy the file 'cars' from 'linkdir' to the current directory:

$ cp linkdir/cars .

Now take a look at the files in the current directory:

$ ls -l

total 12

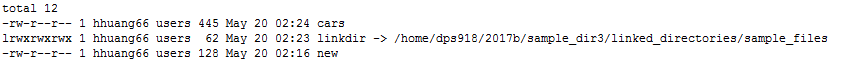
-rw-r--r-- 1 hhuang66 users 445 May 20 02:24 cars

lrwxrwxrwx 1 hhuang66 users 62 May 20 02:23 linkdir -> /home/dps918/2017b/sample\_dir3/linked\_directories/sample\_files

-rw-r--r-- 1 hhuang66 users 128 May 20 02:16 ne

You may now proceed to the Review section.

$ ls -l



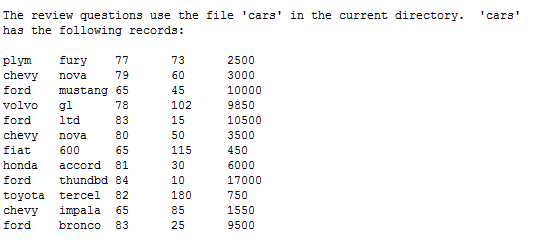
You may now proceed to the Review section.

DPS918 Lab 2.2: Review Exercise

Review Exercise

The purpose of this section is to reinforce your skills with issuing UNIX commands covered in the previous sections.

You will be asked to enter a single UNIX command for the following situations.



Question 1

Create a subdirectory called autos within the current directory:

$ mkdir autos

Question 2

Create an additional name (hard link) to the file 'cars'. The link should be called 'cars.link' and should be in the directory 'autos':

$ ln cars autos/cars.link

Question 3

Create a link to the directory

'~dps918/2017b/sample\_dir3/linked\_directories/sample\_files'. The link, called

'samples', should be in the current directory.

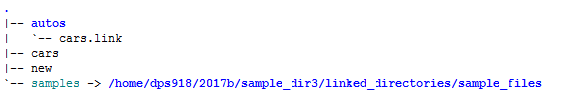
$ ln -s ~dps918/2017b/sample\_dir3/linked\_directories/sample\_files samples

Question 4

There is a command called 'showtree' in the directory you just linked to.

Execute this command using the newly created 'samples' link

$ samples/showtree



Congratulations!

You have successfully completed the Review Exercise!

DPS918 Lab 2.2: Linking Files & Directories

-----------------------------------------------------------------------------------

Available selections:

1 Hard Links - Completed

2 Symbolic Links - Completed

3 Linking To Directories - Completed

4 Review Exercise - Completed

Note that the highlighted lines indicate the incomplete sections of this part of the Lab.

The Review Exercise can only be done when the other sections have been completed.

DPS918 Lab 2.3: Command Summary

-----------------------------------------------------------------------------------

Available selections:

1 Command Summary - Part 1

2 Command Summary - Part 2

3 Command Summary - Part 3

Note that the highlighted lines indicate the incomplete sections of this part of the Lab.

Section 3 can only be done when the other sections have been completed.

DPS918 Lab 2.3: Command Summary (Part 1)

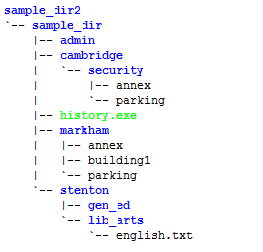
The purpose of this section is to reinforce your skills with issuing UNIX

commands covered previously.

You will be asked to enter a single UNIX command for various situations.

\*\*\* All paths must be relative, unless requested otherwise. Note that in most

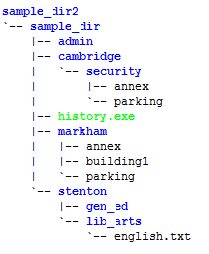
cases, only the shortest possible command line will be accepted.\*\*\*



sample\_dir2 is a subdirectory of your home directory. Move to stenton (make

stenton your current directory), using a relative-to-home pathname:

$ cd ~/sample\_dir2/sample\_dir/stenton

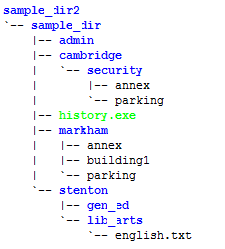


Your current directory is stenton. Display detailed information about the

markham directory itself, not the files within it:

$ ls -ld ../markham

drwxr-xr-x 2 hhuang66 users 48 May 20 15:28 ../markham



Your current directory is stenton. Place a calendar listing for the month of

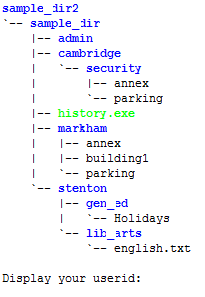
December 2017 into a file called Holidays within the gen\_ed directory:

$ cal 12 2017 >gen\_ed/Holidays

Your current directory is stenton. Append (add) the current date and time to

the Holidays file:

$ date >>gen\_ed/Holidays



Display your userid:

$ id

uid=36329(hhuang66) gid=500(users) groups=500(users)

Display all the users logged into the system:

$ who

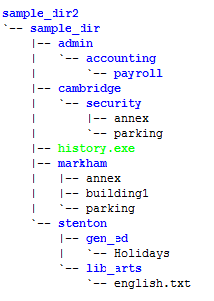
hhuang66 pts/0 May 22 23:14 (38.132.35.91)

lrincoraya pts/1 May 22 15:35 (toroon4423w-lp140-04-76-64-117-201.dsl.bell.ca)

Your current directory is stenton. Create a new directory called payroll

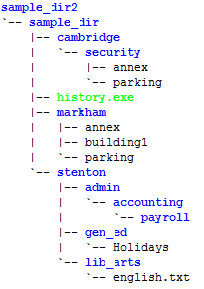
within a new directory called accounting, which should be in admin:

$ mkdir -p ../admin/accounting/payroll



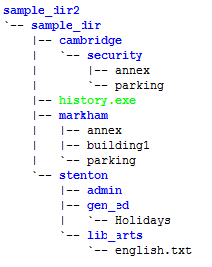
Your current directory is stenton. Move the admin directory under stenton:

$ mv ../admin .



Your current directory is stenton. Remove the accounting directory:

$ rm -r admin/accounting



Your current directory is stenton. Move admin back to sample\_dir:

$ mv admin ..

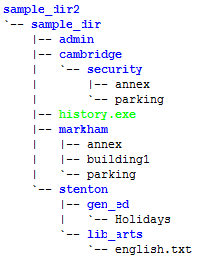
You may now proceed to the next section, Part 2 of this Command Summary.

DPS918 Lab 2.3: Command Summary (Part 2)

The purpose of this section is to reinforce your skills with issuing UNIX commands covered previously.

You will be asked to enter a single UNIX command for various situations.

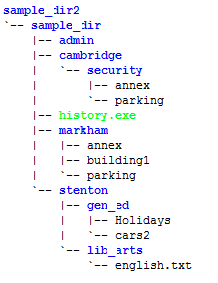
\*\*\* All paths must be relative, unless requested otherwise. Note that in most cases, only the shortest possible command line will be accepted.\*\*\*



Your current directory is stenton. Copy the file ~dps918/2017b/cars to gen\_ed, but calling the new

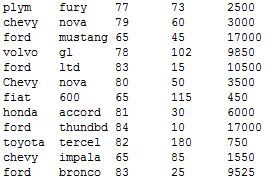
file cars2:

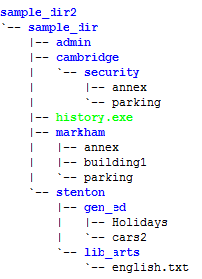
$ cp ~dps918/2017b/cars gen\_ed/cars2



Your current directory is stenton. Display the entire cars2 file:

$ cat gen\_ed/cars2

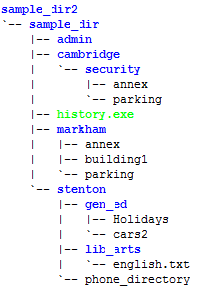




Your current directory is stenton. Copy the first 300 lines of file ~dps918/2017b/phonebook to a new

file called phone\_directory in stenton.

$ head -300 ~dps918/2017b/phonebook > phone\_directory



Your current directory is stenton. The phone\_directory file contains phone number and

other information, sorted by last name, which is the first field in each record. Display

phone\_directory one screen at a time, until you find the record for 'Bruce Byce'. Make

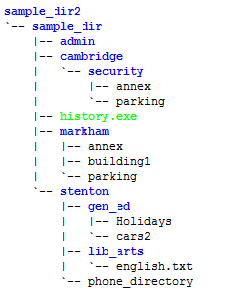
a note of his 4-digit telephone extension (you'll need it for the next question), then

quit the command:

less phone\_directory   
hit space space space space however many times necessary. note the extension then hit q   
done.

Enter the 4-digit telephone extension for 'Bruce Byce':

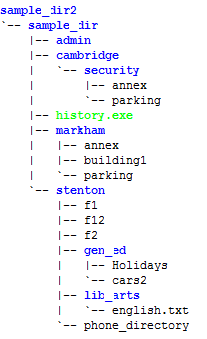
2215



Your current directory is stenton. Create empty files called f1, f2, and f12

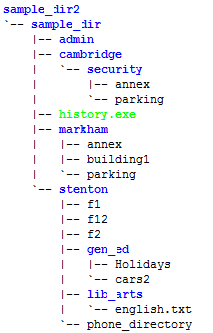
(in that order), within stenton:

$ touch f1 f2 f12



Your current directory is stenton. Move to sample\_dir:

$ cd ..



Your current directory is sample\_dir. Display all the files within sample\_dir

(and it's subdirectories) that are larger than 10k bytes:

$ find . -size +10k

./history.exe

./stenton/phone\_directory

Your current directory is sample\_dir. Edit the file cars2 using vi, and delete

a line or make any other changes you wish, and save the file:

$ vi stenton/gen\_ed/cars2

Click Delete, press Esc, :wq

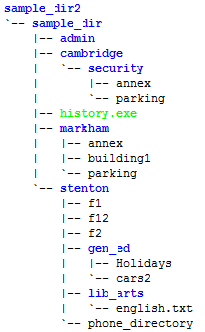
Your current directory is sample\_dir. Display the differences between the files

~dps918/2017b/cars and cars2:

$ diff ~dps918/2017b/cars stenton/gen\_ed/cars2

You may now proceed to the next section, Part 3 of this Command Summary.

DPS918 Lab 2.3: Command Summary (Part 3)



Display the contents of the file '.bashrc' within the home directory of your

instructor (userid les.czegel):

$ cat ~les.czegel/.bashrc

# .bashrc

# User specific aliases and functions

# Source global definitions

if [ -f /etc/bashrc ]; then

. /etc/bashrc

fi

# stty erase ^?

# PS1='${PWD#~/}: '

# FCEDIT=/usr/bin/vi

# export FCEDIT

# VISUAL=/usr/bin/vi

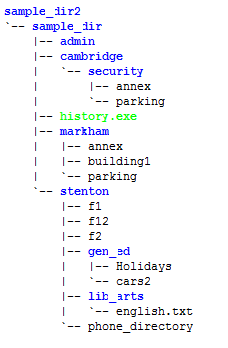
# PATH=$PATH:$HOME/scripts

# umask 077

# set -o vi

# alias ls='ls --color=never'

# mesg n



Your current directory is sample\_dir. Display the 2nd line of the file cars2,

translated to uppercase (hint: start with a 'head' command):

$ head -2 stenton/gen\_ed/cars2 | tail -1 | tr "[a-z]" "[A-Z]"

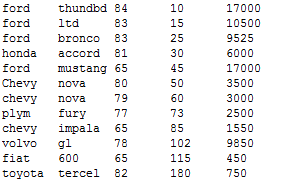
CHEVY NOVA 79 60 3000



Your current directory is sample\_dir. Display cars2 sorted numerically by the

4th field:

$ sort -nk 4 stenton/gen\_ed/cars2

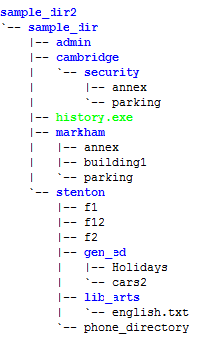


Your current directory is sample\_dir. Display only the 1st field (last name)

of each record in phone\_directory using 'cut'. Note that the field delimiter is

a space:

$ cut -d " " -f1 stenton/phone\_directory



Your current directory is sample\_dir. Find the records in cars2 that contain

the string "ford", and display the number of bytes in the result:

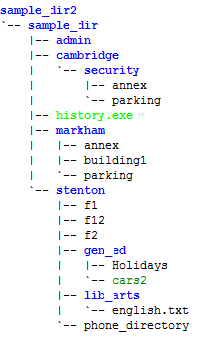
$ grep ford stenton/gen\_ed/cars2 | wc -m

151

Your current directory is sample\_dir. Change the permissions (using octal)

for cars2 so that an 'ls -l' listing would show rwxr-x--x:

$ chmod 751 stenton/gen\_ed/cars2



Your current directory is sample\_dir. Add the permission (using symbolic) for

gen\_ed so that all users can access the file cars2:

$ chmod ugo+x stenton/gen\_ed

Your current directory is sample\_dir. Create a hard link to cars2, called cars3

within sample\_dir:

$ ln stenton/gen\_ed/cars2 cars3

Congratulations!

You have successfully completed this section!

DPS918 Lab 2.3: Command Summary

--------------------------------------------------------------------------------------------

Available selections:

1 Command Summary - Part 1 - Completed

2 Command Summary - Part 2 - Completed

3 Command Summary - Part 3 - Completed

Note that the highlighted lines indicate the incomplete sections of

this part of the Lab.

Section 3 can only be done when the other sections have been

completed.

DPS918 Lab 2.4: Linux Processes

--------------------------------------------------------------------------------------------

Available selections:

1 Process Concepts

2 Managing Processes

3 Review Exercise

Note that the highlighted lines indicate the incomplete sections of

this part of the Lab.

The Review Exercise can only be done when the other sections have

been completed.

DPS918 Lab 2.4: Process Concepts Page 1/17

--------------------------------------------------------------------------------------------

Process Concepts

As mentioned in previous lessons, an operating system controls the hardware and software

resources of a computer system.

These software resources (programs) work with the various hardware resources such as

the CPU, memory, input/output devices, and networking connections.

Since the computer may have only one or a few processors to manage computer

resources as well as allow many end-users to run their own programs, a mechanism

is required to schedule and allocate CPU time for these "simultaneously"

running programs.

In Unix and Linux operating systems, a running program (any type of running

program) is referred to as a "process".

In Linux and Unix, whenever a command is issued or a program executed, a process

is initiated.

For identification purposes a PID or "Process IDentification" number is assigned

to the process. This allows the user to keep track of the process and abort it

if necessary.

To demonstrate, open another bash shell window.

Let's view the processes that are running that are allowing us to connect to

the server.

Issue the following command in the other window:

Ps

The first column (PID) shows the process number. The right column shows the

name of the program that is running.

You should see several processes that are running. One is the "bash" shell

(your login shell), and the actual "ps" command that you issued.

These processes are related to each other. To view the relationship between

the processes, issue the following command in the other window:

ps f -f

You can use this command output to trace the relationship between a process

(PID) and it's parent process (PPID):

- find the PPID of the ps, this will match

the PID of it's parent bash shell

- the PPID of that bash shell will match the

PID of it's parent

In another column, you will see either "S" or "R". These characters represent

the "state" of the process.

The state "R" indicates that the process is "Runnable" which is either running,

or queued by the scheduling program to be run. The state "S" indicates that

the process is sleeping, waiting for the child process(es) to end, in order to

"wake-up" or "run".

There are other process states in addition to "R" and "S":

T - Stopped

X - Terminated or "dead" process

Z - Defunct process, but still using resources

(dead, but still "running" - like a "Zombie")

So far, you have only seen processes that have run from your shell. You can

use the "-u" option with the ps command to view all processes relating to a user

(like "hhuang66" - i.e. you!).

Issue the following command in the other window:

ps f -fu hhuang66

Your Linux session (shell) was run from the sshd process when you connected to

the server via your SSH program.

Here are a few other "sshd" processes that are running, some are other users

that have connected to the server:



Notice that these ssh processes can be traced to a program that permits SSH

connections to the server called:

/usr/sbin/sshd

This process, in turn, can be traced back to PID 1.

This is a special program called "init" which is used to start programs after

the kernel (PID 0) was started following the server "boot-up".

The "init" program launches the "start-up" programs and controls the

processes. There are many other processes besides sshd including: program

scheduler (cron), print job management (spooler), etc...

Many of the programs that the "init" program runs are independent of the shell. In

other words, the program or process "disconnects itself" from the terminal and

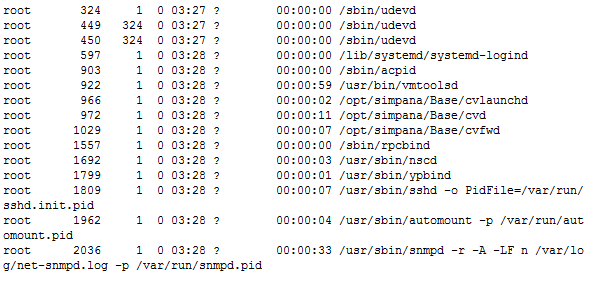
runs in the background. These types of programs are referred to as "daemons".

A user can issue the "ps -e" option to view every process on your system. There

are many programs or processes that can be identified as "daemons" simply by

viewing a "d" at the end of the program name.

For example, these are all the lines from "ps -ef" that end in "d":



A simple, but useful command when viewing process relationships is the "pstree"

command. This command will "draw" a tree-diagram showing the relationships of

processes running on a computer system.

For example, issue pstree for your userid in the other window:

pstree -A hhuang66

You can see the direct relationship of your processes.

You can also view process relationships for all processes running on the computer

system by issuing the pstree command without any arguments. You may or may not

need the -A option, depending on whether your session is set up to support the

ASCII character set.

Notice all the processes that are branching-off the ssh daemon when you enter

the following command in the other window:

pstree –A

Now that you have a basic understanding about processes you need to learn

to control them. You will be learning how to do this in the next section,

"Managing Processes".

Managing Processes

When you issue a command in Unix / Linux, a new child process is created and

executed. The parent process, your shell, is suspended or "sleeps" until the

child process terminates.

As a result, your terminal may be "tied-up" while running long and time-consuming

programs or commands.

There is a method to have child processes run in the background to "free-up"

the parent process or terminal. Processes that are run in the background,

but are still "tied" to a terminal, are referred to as "jobs".

To run a command in the background, type an ampersand "&" symbol after the

command or program name.

To demonstrate background processes, we will use a command that never ends,

"tail -f cars". This command will display the last 10 lines of the specified

file and will stay active to display any lines that are added to the file.

This is a very useful command for system administrators to view changes to a

file, such as a log file, in real time.

In another ssh window, change to a directory that you have the "cars" file,

or copy it from "~dps918/2017b/cars". Then issue the following command:

tail -f cars

DPS918 Lab 2.4: Command Substitution

Once you're satisfied that the command isn't ending, hit <Ctrl>-c (in the

other window!!!).

Now try running the command in the background:

tail -f cars &

You should have seen a number in square brackets, which is the "job number",

followed by the process ID of the new process.

Also notice that you received terminal control immediately, even though the

"tail" is still running. This is a big advantage of using background processes.

Now, issue the command again:

tail -f cars &

Now we have two background jobs running.

Get a third background job running:

tail -f cars &

To view information regarding processes or jobs that are running in the

background, issue the following command in the other window:

jobs

The diplay provides the following job or process information:

[#] Indicates job number

+ Indicates most recent job sent to background

- Indicates second most recent job sent to background

The plus sign "+" is useful, since you may want to bring the most recent job

to the foreground by simply issuing the command "fg" by itself.

You may also want to specify the job # that you want to bring to the

foreground. Let's bring job #2 to the foreground. (We could also use PID

instead of job #).

Issue the following command:

fg %2

Good. Your command "tail -f cars" is now running on your terminal or "in the

foreground".

As before, the tail has control of the terminal. Let's stop it by hitting

<Ctrl>-z (in the other window!!!).

Note that <Ctrl>-c would terminate a foreground process, while <Ctrl>-z just

stops it from running. Now issue the following command

jobs

Notice that job #2 is no longer running.

You can continue a job "running" in the background. This allows the user to

run long commands or programs in the background without occupying the terminal.

Let's run job #2 in the background again. Issue the following command:

bg %2

You can terminate a job that is running in the background by issuing the "kill"

command specifying either the job number or PID.

Unpriviledged (non-administrator) users can only kill their own processes or

jobs. Root (administrator) can kill any process.

Issue the command:

kill %3

Hit <Enter>, and you should get a message showing that job #3 was terminated.

Now, again, issue a command to display the other jobs that are running in

the background:

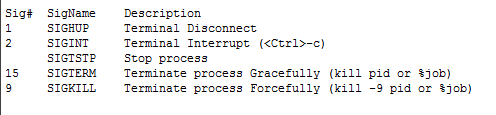
jobs

The kill command actually sent a "signal" to the job or process that was running

in the background. A signal is a special code that is sent from one process

to another.

Some signals are:



Let's send signal SIGTSTP to stop job #1 by issuing the "kill" command with

signal SIGTSTP.

This is the equivalent of <Ctrl>-z for a foreground job. It will stop the

process from running, but will not terminate it.

Enter the following command:

kill -SIGTSTP %1

Hit <Enter>, and you should get a message showing that job #1 was stopped.

Now, again, issue a command to display the background jobs:

jobs

Note that there are two signals that can be used to kill a job. Signal #15

(SIGTERM) is the default kill signal, and should always be tried first.

Signal #9 (SIGKILL) should only be used as a last resort, if signal #15 doesn't

work. This would mean that either the program was written to ignore signal #15,

or there is some kind of a problem. Signal #9 cannot be ignored, but will not

clean up resources, such as memory and page space.

Let's kill the two remaining processes, by entering the following command:

kill %1 %2

Hit <Enter>, and you should get a message showing that jobs #1 and #2 were

terminated.

Now, again, issue a command to display any background jobs:

jobs

Good. There should be no jobs that are running in the background.

That concludes this section. Now it's time to do the Review section!

DPS918 Lab 2.4: Review Page 1/7

-----------------------------------------------------------------------------------

Review Exercise

The purpose of this section is to reinforce your skills with issuing UNIX

commands covered in the previous sections.

You will be asked to enter a single UNIX command for the following situations.

Question 1

Enter the command that is used to display currently running processes, including

an option to display PPID's (parent process id's):

ps –f

Question 2

Enter the command that is used to display the background jobs that are

currently running:

jobs

Question 3

Enter the command to bring job #2 to the foreground

fg %2

Question 4

Enter the command to terminate job #3:

kill %3

Question 5

Enter the command to terminate job #3, assuming "kill %3" didn't work:

kill -9 %3

Congratulations!

You have successfully completed the Review Exercise!

DPS918 Lab 2.5: Regular Expressions Using grep

---------------------------------------------------------------------------------

Available selections:

1 Introduction to Regular Expressions

2 Characters & Character Classes

3 The Wildcard

4 Repetition

5 Anchors

6 Review Exercise

Note that the highlighted lines indicate the incomplete sections of

this part of the Lab.

The Review Exercise can only be done when the other sections have

been completed.

Regular Expressions

Before we start working on regular expressions, you must bear in mind:

REGULAR EXPRESSIONS are NOT THE SAME as AMBIGUOUS FILENAMES!

Although both are used to match patterns, they cannot be swapped, and they

use the same symbols to mean different things.

DO NOT GET THEM MIXED UP IN YOUR MIND!

DPS918 Lab 2.5: Introduction Page 2/4

---------------------------------------------------------------------------------

Regular Expressions

Regular expressions are text search patterns. They are written using a

specialized language, and there are several versions of that language.

There are several basic components used to build simple regular expressions

(or "regexps"):

- Characters and Character Classes

- Wildcard

- Repetition

- Anchors

Our sample file "names" contains these lines:

Mr. Smith

Jenn Bewlite

Ms Carmichael

Dr Ivan James

Mrs Holly Alva Beswol

Mrs James Sheepwool

Mr. Hitchcock

Miss Jennifer Olgovie

Melissa (Missy) Smith

JoJo Smythe

Dr Smooth

Mr. John Waterson

Sara Oooosterinnk

Ms Ella Moloska

James Ngenda

Dr and Mr Wilson

Creating Regular Expressions

When you write regular expressions, you are stating the text pattern that you

wish to match. We are going to use the "grep" command to test some regular

expressions.

Let's start with the first component, which is characters.

DPS918 Lab 2.5: Characters Page 1/8

---------------------------------------------------------------------------------

Characters & Character Classes

Characters may be literal characters, such as letters or digits, which match

exactly the same letter or digit.

Try this example:

$ grep "Smith" names

Mr. Smith

Melissa (Missy) Smith

In that last example, the regexp was "Smith", and it exactly matched the word

"Smith", which appeared in two lines.

Note that the quotes were not required in the last example, but it's easier

to use them all the time than to think about when they are and are not required.

Let's try matching "smith" in lowercase...

$ grep "smith" names

The last command didn't match any names because the "s" in "smith" was in

lower-case. "grep" is case-sensitive.

Generally, options such as case-sensitivity are not controlled by the regular

expression, but are controlled by options to the program which uses the

regular expression to search.

For example, in the case of grep, the "-i" option makes it ignore UPPER-

and lower-case.

$ grep -i "smith" names

Mr. Smith

Melissa (Missy) Smith

If you want to search for a character other than a letter or digit, you will

need to know if that character has a special meaning. If it does, then you

can remove the special meaning by placing a backslash in front of the character.

The confusing part is that, in some cases, a character BECOMES a special

character when a backslash is placed in front of it -- this depends on the

program you are using (for example, "grep" and "sed" are different in this way).

The only reliable solution is to check the documentation for the program you

are using.

You can also specify a literal character using a character class.

Character classes in regular expressions are basically the same as character

classes in "ambiguous filenames": they consist of lists of characters and/or

ranges of characters, written inside square-brackets.

For example, the character class [ABCD] matches the letter A or the letter

B or the letter C or the letter D. [A-D] means the same thing. [^ABCD] will

match any letter EXCEPT A or B or C or D.

Let's use a character class to match both "Ms" and "Mr" in our names file.

(Notice the space after the character class in this command).

$ grep "M[rs] " names

Ms Carmichael

Ms Ella Moloska

Dr and Mr Wilson

Notice that "M[rs] " didn't match "Mrs", because the regular expression

specifies the letter M followed by EITHER an r or an s, followed by a space.

Remember that a character class matches only ONE character.

You may now proceed to the section on the Wildcard.

DPS918 Lab 2.5: The Wildcard Page 1/3

---------------------------------------------------------------------------------

The Wildcard

A period "." character is called a wildcard, and will match any one character.

$ grep "Sm.th" names

Mr. Smith

Melissa (Missy) Smith

JoJo Smythe

Notice that the period matched only one character. This is why the regular

expression "Sm.th" did not match "Smooth".

To match two characters, you could just place two periods in a row.

$ grep "Sm..th" names

Dr Smooth

That's it for the wildcard!

Now proceed to the section on Repetition to see another way of specifying

multiple characters.

DPS918 Lab 2.5: Repetition Page 1/3

---------------------------------------------------------------------------------

The asterisk character, "\*", means that the pattern should match the previous

character zero or more times. That means that the previous character does

not have to be present at all.

Let's search for a "w", then zero or more of the letter "o", then the letter

"l".

$ grep "wo\*l" names

Jenn Bewlite

Mrs Holly Alva Beswol

Mrs James Sheepwool

Notice that in "Bewlite" there was no "o" but in "Beswol" there was one letter

"o" and in "Sheepwool" there were two.

The asterisk is often placed after the wildcard to mean any number of any

characters (the wildcard means any character, and the asterisk means zero or

more repetitions).

Let's say we wanted to search for Dr James, but we weren't sure if his first

name or initials were between the "Dr" and the "James". We could use .\*

to match anything that was in the middle.

$ grep "Dr.\*James" names

Dr Ivan James

Good!

Now proceed to the section on Anchors.

DPS918 Lab 2.5: Anchors Page 1/5

---------------------------------------------------------------------------------

Anchors

Anchors let us search for text only at the start or end of the line.

The symbols are:

^ anchor to the start of the line

$ anchor to the end of the line

Search for "Miss" in the names file.

DPS918 Lab 2.5: Anchors Page 2/5

---------------------------------------------------------------------------------

$ grep "Miss" names

Miss Jennifer Olgovie

Melissa (Missy) Smith

Notice that we matched the characters "Miss" in the middle of one line. Now

let's anchor the regular expression to the start of the line with "^".

$ grep "^Miss" names

Miss Jennifer Olgovie

We can also anchor to the end of a line.

Let's search for all of the lines with "James" in them.

$ grep "James" names

Dr Ivan James

Mrs James Sheepwool

James Ngenda

If we anchor the "James" to the end of the line, we can eliminate the records

with "James" as a first name.

$ grep "James$" names

Dr Ivan James

Good!

You may now proceed to the Review section.

DPS918 Lab 2.5: Review Page 1/9

---------------------------------------------------------------------------------

Review Exercise

The purpose of this section is to reinforce your skills with issuing UNIX

commands covered in the previous sections.

You will be asked to enter a single UNIX command for the following situations.

The data file for the review questions, named "inventory", looks like this:

Strawberry Jam,300,4

Raspberry Jam,1216,7

Blueberry Jam,96,195

Strawberry Compote,49,621

Raspberry Compote,1937,624

Blueberry compote,200,625

Frozen Strawberries,130,1941

Straw Hats,16,2047

The first field is the product name, the second field is the quantity on hand,

and the third field is the product code. The fields are separated by commas.

In each of the following questions, write a command using "grep" and a

regular expression. The data file in all cases is named "inventory".

Question 1

Display all of the lines in the file that contain the characters "Jam".

Remember that we're using "grep" and the data file is named "inventory".

$ grep "Jam" inventory

Strawberry Jam,300,4

Raspberry Jam,1216,7

Blueberry Jam,96,195

Question 2

Display all of the lines in the file that contain the word "Straw" right

before a space.

$ grep "Straw " inventory

Straw Hats,16,2047

Question 3

Display all of the lines in the file that contain "Compote" or "compote".

(Use a character class, do not use the -i option).

$ grep "[Cc]ompote" inventory

Strawberry Compote,49,621

Raspberry Compote,1937,624

Blueberry compote,200,625

Review Question 4

Display all of the lines in the file that contain "Straw" at the beginning

of the line.

$ grep "^Straw" inventory

Strawberry Jam,300,4

Strawberry Compote,49,621

Straw Hats,16,2047

Review Question 5

Display all of the lines in the file where the last field is one digit long.

Search for ',' before the field, then use a character class to make sure

it's one digit, and anchor it to the end of the line to make sure it's the

last field.

$ grep ",[0-9]$" inventory

Strawberry Jam,300,4

Raspberry Jam,1216,7

Review Question 6

Display all of the lines that contain "Straw" followed later in the line by

"Hat".

$ grep "Straw.\*Hat" inventory

Straw Hats,16,2047

DPS918 Lab 2.6: sed And awk

---------------------------------------------------------------------------------

Available selections:

1 Using sed

2 Using awk

3 Review Exercise

Note that the highlighted lines indicate the incomplete sections of

this part of the Lab.

The Review Exercise can only be done when the other sections have

been completed.

DPS918 Lab 2.6: Using sed Page 1/8

---------------------------------------------------------------------------------

Using sed

sed is a 'stream editor'. This means it can be used to change (edit) a file

non-destructively. In other words, it will send the changed file to standard

output, but the original file is left unchanged.

In this Lab, we will only use those facilities of sed and other commands that

are covered in the lectures, so you may wish to use the lectures as a reference.

The true power of sed can be seen when it's combined with other commands,

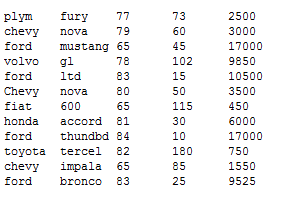
either within a series of piped commands, or within a script.

As a matter of fact, this is Unix's major strength, the ability to combine

several simple commands to get a fairly complex end result with relative ease.

In this section, we're going to create a series of piped commands to solve

a problem, using the 'cars' file that you know and love:



We have a customer who would like to buy a car, and we want to produce a

list of cars that meet her requirements. She wants to see a list of all the

cars on the lot that cost less than $10,000, except she doesn't want a chevy.

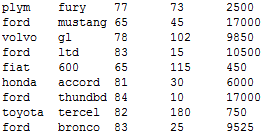
The list should be sorted from lowest cost to highest, and we want to produce

the list so that the alphabetic characters are in uppercase.

Let's start by displaying all the records in 'cars' that are not chevy's.

Use a grep command to do this, and don't forget to ignore case:

$ grep -iv "chevy" cars



So far we have:

grep -iv chevy cars

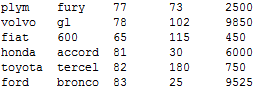
Pipe the output of the grep into a sed to delete records that match a

regular expression representing 5 digits at the end of a record (DO NOT

use a repetition factor, such as {5}, for this). This will delete the cars

that have a price greater than $9,999:

$ grep -iv chevy cars |sed '/[0-9][0-9][0-9][0-9][0-9]$/ d'



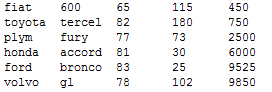
o far we have:

grep -iv chevy cars | sed '/[0-9][0-9][0-9][0-9][0-9]$/ d'

Now let's sort the cars by price. Pipe the output of the sed into a sort

to do this, and don't forget to sort numerically on the 5th field:

$ grep -iv chevy cars | sed '/[0-9][0-9][0-9][0-9][0-9]$/ d' | sort -nk 5



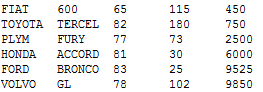
So far we have:

grep -iv chevy cars | sed '/[0-9][0-9][0-9][0-9][0-9]$/ d' | sort -nk 5

Finally, let's display the output in uppercase. Pipe the output of the

sort into a tr command:

$ grep -iv chevy cars| sed '/[0-9][0-9][0-9][0-9][0-9]$/ d' | sort -nk 5 |tr "[a-z]" "[A-Z]”



Good.

Let's go on to awk.

DPS918 Lab 2.6: Using awk Page 1/7

------------------------------------------------------------------------------------------

Using awk

awk is a superset of the C programming language. Fortunately, a very small subset of

awk functionality is enough to handle many situations that arise both on the command

line and within scripts.

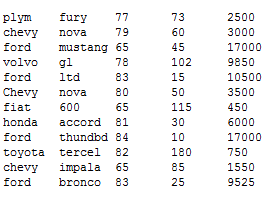
awk can make substantial changes to a file, and like sed, it will send the changed

file to standard output, but the original file is left unchanged.

In this Lab, we will only use those facilities of awk and other commands that are

covered in the lectures, so you may wish to use the lectures as a reference.

Once again, we're going to use the 'cars' file:



We have another customer who would like to buy a car, and we need a list of cars that

meet his requirements. He wants to rebuild a classic car built between 1975 and 1983.

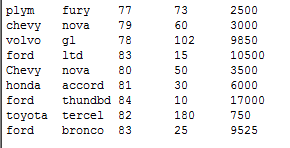
He's willing to pay up to $9,000, but he cares less about price than he does about

low mileage, so the list should be sorted from lowest mileage to highest.

Let's start by displaying all the records in 'cars' that are newer than 1974. Use an

awk command to do this, selecting those records that have a third field greater than 74:

$ awk '$3>74' cars



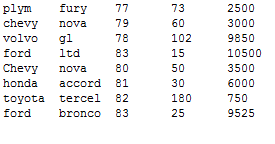
So far we have:

awk '$3 > 74' cars

Now let's pare the list down to the cars that are older than 1984. Pipe the output of

the awk into a second awk, selecting those records that have a third field less than 84:

$ awk '$3 > 74' cars |awk '$3<84'



So far we have:

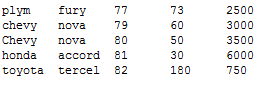
awk '$3 > 74' cars | awk '$3 < 84'

Now let's delete the cars that are more than $9,000. Pipe the output of the second

awk into a third awk to do this, selecting those records that have a fifth field less

than or equal to 9000:

$ awk '$3 > 74' cars | awk '$3 < 84' | awk '$5 <=9000'



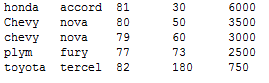
So far we have:

awk '$3 > 74' cars | awk '$3 < 84' | awk '$5 <= 9000'

Finally, let's sort the cars by mileage. Pipe the output of the third awk into a sort

to do this, and don't forget to sort numerically on the 4th field:

$ awk '$3 > 74' cars | awk '$3 < 84' | awk '$5 <= 9000' | sort -nk4



Good!

You may now proceed to the Review section.

DPS918 Lab 2.6: Review Page 1/7

------------------------------------------------------------------------------------------------------------------------------------------

Review Exercise

The purpose of this section is to reinforce your skills with issuing UNIX commands covered in the previous sections.

You will be asked to enter a single UNIX command for the following situations.

The data file for the review questions, named "inventory", looks like this:

Strawberry Jam,300,4

Raspberry Jam,1216,7

Blueberry Jam,96,195

Strawberry Compote,49,621

Raspberry Compote,1937,624

Blueberry compote,200,625

Frozen Strawberries,130,1941

Straw Hats,16,2047

The first field is the product name, the second field is the quantity on hand,

and the third field is the product code. The fields are separated by commas.

In each of the following questions, write a command using "sed" or

"awk". The data file in all cases is named "inventory".

Question 1

Display only the fifth line in the file, using sed.

Remember that the data file is named "inventory".

$ sed -n '5 p' inventory

Raspberry Compote,1937,624

Question 2

Display all of the lines in the file, changing the characters "Jam" to "Marmalade", using sed.

Remember that the data file is named "inventory".

$ sed 's/Jam/Marmalade/g' inventory

Strawberry Marmalade,300,4

Raspberry Marmalade,1216,7

Blueberry Marmalade,96,195

Strawberry Compote,49,621

Raspberry Compote,1937,624

Blueberry compote,200,625

Frozen Strawberries,130,941

Straw Hats,16,2047

Question 3

Display all of the lines in the file showing only the quantity and product name, in that order and separated by a space, using awk. Product

name is the first field, and quantity is the second field.

Remember that the data file is named "inventory".

$ awk -F"," '{ print $2, $1 }' inventory

300 Strawberry Jam

1216 Raspberry Jam

96 Blueberry Jam

49 Strawberry Compote

1937 Raspberry Compote

200 Blueberry compote

130 Frozen Strawberries

16 Straw Hats

Question 4

Display all of the lines in the file with less than 100 items in inventory, using awk. Quantity is the second field.

Remember that the data file is named "inventory".

$ awk -F',' '$2 <100' inventory

Blueberry Jam,96,195

Strawberry Compote,49,621

Straw Hats,16,2047

Congratulations!

You have successfully completed the Review Exercise!

Available selections:

1 Using sed - Completed

2 Using awk - Completed

3 Review Exercise - Completed

Note that the highlighted lines indicate the incomplete sections of

this part of the Lab.

The Review Exercise can only be done when the other sections have

been completed.

DPS918 Lab 2.7: Interactive Shell Environment

------------------------------------------------------------------------------------------------------------------------------------------

Available selections:

1 Environment Variables

2 Variable Assignment & Quoting

3 Command Substitution

4 Using The alias Command

5 Variables

6 Review Exercise

Note that the highlighted lines indicate the incomplete sections of

this part of the Lab.

The Review Exercise can only be done when the other sections have

been completed.

DPS918 Lab 2.7: Environment Variables Page 1/16

------------------------------------------------------------------------------------------------------------------------------------------

Environment Variables

There are many pieces of information that can be passed to programs through command-line options and arguments. This can lead to a lot of typing.

Fortunately, Unix/Linux provides a way to reduce the amount of typing by enabling some information to be stored in 'Environment Variables', which are passed to all child processes.

Every process receives environment variables from its parent, and every process passes environment variables to processes that it starts (the child processes). Shells also permit you to view, create, remove, and alter environment variables.

The command to view all of your shell's current environment variables is 'env'. Try it out now...

$ env

LESSKEY=/etc/lesskey.bin

NNTPSERVER=news

MANPATH=/usr/lib/mpi/gcc/openmpi/share/man:/usr/local/man:/usr/local/share/man:/usr/share/man

XDG\_SESSION\_ID=165

HOSTNAME=matrix

XKEYSYMDB=/usr/X11R6/lib/X11/XKeysymDB

HOST=matrix

TERM=vt100

SHELL=/bin/bash

PROFILEREAD=true

HISTSIZE=9999

TMPDIR=/tmp

SSH\_CLIENT=10.40.84.170 30821 22

MORE=-sl

SSH\_TTY=/dev/pts/7

LC\_ALL=C

ANT\_HOME=/usr/local/ant

JRE\_HOME=/usr/java/jdk/jre

USER=hhuang66

…

That was too much information for your screen! Try using 'more' to view it one screen at a time.

Remember! - press 'q' to quit from 'more' when you are ready to continue.

$ env | more

LESSKEY=/etc/lesskey.bin

NNTPSERVER=news

MANPATH=/usr/lib/mpi/gcc/openmpi/share/man:/usr/local/man:/usr/local/share/man:/usr/share/man

XDG\_SESSION\_ID=165

HOSTNAME=matrix

XKEYSYMDB=/usr/X11R6/lib/X11/XKeysymDB

HOST=matrix

TERM=vt100

SHELL=/bin/bash

PROFILEREAD=true

HISTSIZE=9999

TMPDIR=/tmp

SSH\_CLIENT=10.40.84.170 30821 22

MORE=-sl

SSH\_TTY=/dev/pts/7

LC\_ALL=C

ANT\_HOME=/usr/local/ant

JRE\_HOME=/usr/java/jdk/jre

USER=hhuang66

There were many variables in there. Some of the key ones are:

HOME -- contains your home directory

MAIL -- specifies where your e-mail mailbox file is located

PATH -- tells the system which directories to search for commands

PS1 -- sets your primary shell prompt

TERM -- tells programs what type of terminal you're using

To set a variable, just enter its name, then an equal sign, then the value. Do not use any spaces or tabs.

Let's assign the value '5' to the variable 'FOO':

$ FOO=5

Now use 'env' to see if 'FOO' is in the list (remember to use 'more'!).

$ env | more

LESSKEY=/etc/lesskey.bin

NNTPSERVER=news

MANPATH=/usr/lib/mpi/gcc/openmpi/share/man:/usr/local/man:/usr/local/share/man:/usr/share/man

XDG\_SESSION\_ID=165

HOSTNAME=matrix

XKEYSYMDB=/usr/X11R6/lib/X11/XKeysymDB

HOST=matrix

T

Even though variable 'FOO' has been given a value, it is not yet an environment variable -- it is a shell variable accessible only by

your shell. To make it into an environment variable, we must 'export' it so that child processes receive its value.

$ export FOO

Now use 'env' again to see if 'FOO' is in the list (remember to use 'more'!).

$ env | more

LESSKEY=/etc/lesskey.bin

NNTPSERVER=news

MANPATH=/usr/lib/mpi/gcc/openmpi/share/man:/usr/local/man:/usr/local/share/man:/usr/share/man

XDG\_SESSION\_ID=165

H

Now other programs can use the value of 'FOO'.

You can also remove an environment variable with the 'unset' command.

Let's remove 'FOO'.

> unset FOO

One of the most important environment variables is 'PATH'.

PATH contains a list of directories, separated by colons. When you enter a command, the shell will search those directories to find the command.

Your PATH is currently set to:

/home/uli101/bin:/usr/lib/mpi/gcc/openmpi/bin:/home/hhuang66/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/X11R6/bin:/usr/games:/usr/local/ant/bin:/usr/java/jdk/bin:/usr/lib/mit/bin:/usr/lib/mit/sbin:/opt/oracle12c/instantclient\_12\_1:/opt/oracle12c/instantclient\_12\_1:/home/hhuang66/scripts

Change your PATH to only include the root directory '/'.

> PATH=/

Now try a simple 'ls' command.

$ ls

bash: ls: command not found

ls was not found because it is not found in any of the directories in the PATH.

(PATH has been automatically reset for you).

You can retrieve the value of any environment variable using a dollar sign and the variable name. You can put this in any command.

For example...

$ echo $PATH

/home/uli101/bin:/usr/lib/mpi/gcc/openmpi/bin:/home/hhuang66/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/X11R6/bin:/usr/games:/usr/local/ant/bin:/usr/java/jdk/bin:/usr/lib/mit/bin:/usr/lib/mit/sbin:/opt/oracle12c/instantclient\_12\_1:/opt/oracle12c/instantclient\_12\_1:/home/hhuang66/scripts

Notice that you only need to place a dollar sign in front of a variable name if you are retrieving the variable value. In any other

situation we leave it off.

We can combine retrieving a variable with assigning a value to a variable. Let's say you wanted to add the directory '/sbin' to your PATH.

------------------------------------------------------------------------------------------------------------------------------------------------

$ PATH="$PATH:/sbin"

In the last example, we placed quotes around the value that we were assigning just in case there were spaces in the value. It is a good

idea to always use quotes around the value when assigning a variable.

Exported environment variables are inherited by child processes, so every program that you start will receive their values if you set

them up as soon as you login.

The file '~/.bash\_profile' for the bash shell or '~/.profile' for ksh is run when you login, so you can add any environment variable

set-up lines to that file. Note that since it is a regular text file, you can use any regular text editor to change it.

That concludes this section.

Remember:

To create a variable: variablename="value" To export a variable: export variablename To access a variable:

$variablename (in a command) To remove a variable: unset variablename To see environment variables: env To see ALL

variables: set

And to have an environment variable created every time you login, place the appropriate commands in ~/.bash\_profile (bash) or ~/.profile (ksh).

You may now proceed to the next section about 'Variable Assignment & quoting'.

DPS918 Lab 2.7: Variable Assignment & Quoting Page 1/10

---------------------------------------------------------------------------------

To set a variable, just enter its name, then an equal sign, then the value.

Do not use any space or tabs.

Let's assign the value 'Ontario' to the variable 'province':

$ province=Ontario

To retrieve the value of a variable, we place a dollar sign in front of the

variable name. Try the following:

$ echo My province is $province

My province is Ontario

Let's see what happens when we try to use spaces around the '=' sign:

$ city = Brampton

bash: city: command not found

An error message results because the variable 'city' is misinterpreted as a

command by the shell. Similarly, let's see what happens when we try to use

spaces within the character string:

$ city=My city is Brampton

bash: city: command not found

Again we get a misinterpretation by the shell. Fortunately, we can use

spaces within the character string by using quotes around the entire string.

Either single or double quotes will work in this case:

$ city='My city is Brampton'

Double quotes are also called weak quotes, because they allow variables to

be substituted by their values:

$ echo "My province is $province"

My province is Ontario

Single quotes are also called strong quotes, because they do not allow variable substitution:

$ echo 'My province is $province'

My province is $province

You may now proceed to the next section about 'Command Substitution'.

DPS918 Lab 2.7: Command Substitution Page 1/5

---------------------------------------------------------------------------------

Command substitution takes the output of a command and treats it as a

character string.

The older form of command substitution uses backquotes:

$ echo "My user name is `whoami`"

My user name is hhuang66

The newer form of command substitution is more powerful, since it allows

nesting of command substitution. It uses '$' followed by parentheses

surrounding the command:

$ echo "My user name is $(whoami)"

My user name is hhuang66

The result of command substitution can be used anywhere a string can be used.

For example, it may be placed into a variable:

$ username=$(whoami)

Now the variable 'username' can be displayed:

$ echo "My user name is $username"

My user name is hhuang66

You may now proceed to the next section about 'Using the alias Command'.

DPS918 Lab 2.7: Using The alias Command Page 1/12

---------------------------------------------------------------------------------

The 'alias' command can be used to give a command a different name.

For example:

alias display=echo

In this case, whenever the shell finds a 'display' command, it will substitute

the 'echo' command. Try it:

DPS918 Lab 2.7: Using The alias Command Page 2/12

---------------------------------------------------------------------------------

$ alias display=echo

Now try using the new 'display' command:

$ display "My user name is $(whoami)"

My user name is hhuang66

An alias can be removed using the 'unalias' command:

$ unalias display

Now try using the 'display' command again:

$ display "My user name is $(whoami)"

bash: display: command not found

The 'alias' command can also be used to add options or arguments to a command.

For example:

alias mkdir='mkdir -p'

or:

alias md='mkdir -p'

In this case, whenever the shell finds a 'mkdir' or 'md' command, it will

substitute the 'mkdir -p' command. Try it:

$ alias md='mkdir -p'

Now try using the new 'md' command. In your current (home) directory, create

the directory 'lab2\_testdir/a/b/c/d':

$ md lab2\_testdir/a/b/c/d

Now check that the directory 'lab2\_testdir/a/b/c/d' exists:

$ ls -ld lab2\_testdir/a/b/c/d

drwxr-xr-x 2 hhuang66 users 6 May 26 00:44 lab2\_testdir/a/b/c/d

The 'alias' command can be convenient, but it has potential drawbacks.

For example, if you ask a user to execute the 'md' or other aliased commands,

they may not have the same aliases as you do.

Similarly, you might use the 'md' or other aliased commands in scripts that

you write for others, and they may not have the aliases.

Remove the 'md' alias:

$ unalias md

You may now proceed to the next section about 'Variables'.

DPS918 Lab 2.7: Variables Page 1/19

---------------------------------------------------------------------------------

Variables

Besides the user-defined variables and environment variables seen in earlier

sections, there are lots of pre-defined variables that can be used on the

command line and within scripts.

Positional parameters are available as the variables $1, $2, $3, and so on.

For example, if we write a script called 'script1', and call it as follows:

script1 Toronto Brampton 'North York'

then the value of variable $1 is the string 'Toronto', $2 has the value

'Brampton', and $3 has the value 'North York'.

Similarly, these variables can be set on the command line by using the 'set'

command. For example:

set Toronto Brampton 'North York'

then the value of variable $1 is the string 'Toronto', $2 has the value

'Brampton', and $3 has the value 'North York'. Try this now:

$ set Toronto Brampton 'North York'

Now display the value of the variable $1:

$ echo $1

Toronto

Now display the value of the variable $3:

$ echo $3

North York

DPS918 Lab 2.7: Variables Page 8/19

---------------------------------------------------------------------------------

The variable $\* represents all of the positional parameters as a single string,

with the parameters separated by a single space. Display the value of the

variable $\*:

$ echo $\*

Toronto Brampton North York

The variable $@ represents all of the positional parameters as separate strings.

The difference between $\* and $@ will be easier to show when looping is

discussed in a later section.

The variable $# represents the number of positional parameters. Display the

value of the variable $#:

$ echo $#

3

The command 'shift' shifts all of the positional parameters by one position.

The value of the first positional parameter disappears, so that $1 has the

value of the second positional parameter, and so forth.

Remember that our three parameters were 'Toronto', 'Brampton', and 'North York'.

Now try a shift command:

$ shift

Now display the values of the variables $1 and $2:

$ echo "$1 and $2"

Brampton and North York

Now display the values of the variables $3 and $#:

$ echo "$3 and $#"

and 2

The variable $$ contains the process id number, or PID, of the current

process. All processes running on the system, including this Lab, have a

unique process id. This is very useful when creating temporary files within

a script, allowing temporary files to have a unique name regardless of how

many times the script is being executed at the same time.

Again, this Lab is an example of that, where multiple students can simultaneouly

run the Lab program without interfering with each other. Display the value

of the variable $$:

$ echo $$

9981

The variable $? contains the 'exit status' of the last command that was

executed. The 'exit status' is usually used as an indication of the success

or failure of a command. Zero indicates success, non-zero indicates failure.

This is very useful in scripting, where we can determine what the script will

do next based on whether the previous command was successful or not.

That concludes this section.

Now it's time to do the Review section!

DPS918 Lab 2.7: Review Exercise Page 1/9

---------------------------------------------------------------------------------

Review Exercise

The purpose of this section is to reinforce your skills with issuing UNIX

commands covered in the previous sections.

You will be asked to enter a single UNIX command for the following situations.

Question 1

The following command has been executed:

set 10 20 30 40 50

Enter the value of the variable $3:

30

Question 2

The following command has been executed:

set 10 20 30 40 50

Enter the value of the variable $#:

5

Question 3

The following commands have been executed:

set 10 20 30 40 50

shift

shift

Enter the value of the variable $3:

50

Question 4

The following commands have been executed:

set 10 20 30 40 50

shift

shift

Enter the value of the variable $#:

3

Question 5

The following commands have been executed:

set 10 20 30 40 50

shift

shift

Enter the value of the variable $\*:

Question 6

Enter the command to display a line that looks like the following:

My process id is ####

where '####' is the process id of this Lab (Hint: use a variable, not a

command substitution):

$ echo "My process id is $$"

My process id is 9981

Question 7

What command would you use to add the directory '/usr/share/bin' to the end

of your current PATH?

> PATH="$PATH:/usr/share/bin"

Congratulations!

You have successfully completed the Review Exercise!

DPS918 Lab 2.7: Interactive Shell Environment

---------------------------------------------------------------------------------

Available selections:

1 Environment Variables - Completed

2 Variable Assignment & Quoting - Completed

3 Command Substitution - Completed

4 Using The alias Command - Completed

5 Variables - Completed

6 Review Exercise - Completed

Note that the highlighted lines indicate the incomplete sections of

this part of the Lab.

The Review Exercise can only be done when the other sections have

been completed.

DPS918 Lab 2.8: Submit Lab 2 Page 1/2

---------------------------------------------------------------------------------

Total marks obtained is 3 out of 3.

Lab 2 Section 1 was completed

Lab 2 Section 2 was completed

Lab 2 Section 3 was completed

Lab 2 Section 4 was completed

Lab 2 Section 5 was completed

Lab 2 Section 6 was completed

Lab 2 Section 7 was completed

Submission Confirmation

You may now submit Lab 2 by typing 'Yes'. If you do NOT want to submit at

this time, type 'm' or 'M'.

Congratulations!

You have successfully submitted Lab 2!