**CSC 209 UNIX Tools**

|  |  |  |
| --- | --- | --- |
|  | **Assignment 3** |  |

**Objectives: Practicing simple UNIX commands – Access Permissions**

**Note**: **Read** the entire assignment carefully and carry out the following tasks one by one. For some steps, I may provide the required UNIX command. For others, identify the UNIX commands you need. **UNIX is case sensitive**. Create folders and files exactly as indicated below, paying special attention to case. Complete the report as indicated below and submit the same. I do not want to see copy-and-paste of your computer output in your report. By forcing you to write I am hoping that you will remember the UNIX commands used and the results observed.

1. Login to holly server using your Webmail Net-ID and password.
   1. Since we are not using the graphical user interface (GUI), you have to rely on your knowledge of UNIX commands. Take a note of how the command prompt placed by the shell appears and report below:

Shell prompt: [139] [etay1@courses2016:~]$

a. Type an appropriate command to print the working directory.

Command to print the working directory: pwd

Your login directory: /home/etay1

b. Use an appropriate command to switch to the **csc209** folder which you created in Assignment 1.

Command to switch from login directory to **csc209**: cd csc209

2. Complete the following steps:

a. Determine the current working directory.

Command to print the working directory: pwd

Your current working directory: /home/etay1/csc209

b. You should currently be located in **csc209**. Create a new folder named **asgn03** under **csc209**. Set the access permissions for **asgn03** as **rwx --- ---** by typing a command:

**chmod 700 asgn03**

Command to create the **asgn03** folder: mkdir asgn03

Command to set the access permissions: chmod 700 asgn03

c. Move to **asgn03** and determine the current working directory.

Command need to switch from **csc209** directory to **asgn03**: cd asgn03

Command to print the working directory: pwd

Your current working directory: /home/etay1/csc209/asgn03

Shell prompt: [146] [etay1@courses2016:~/csc209/asgn03]$

**From now on, unless explicitly permitted, you should not move away from this asgn03 directory.**

**If you happen to log out in the middle of the exercise, be sure to use appropriate command(s)**

**and move to asgn03 before you continue.**

2. Type the following five commands one by one and report the output:

|  |  |
| --- | --- |
| **Command** | **Output** |
| **date** | Wed Sep 21 00:01:23 EDT 2022 |
| **date +%Y** | 2022 |
| **hostname** | courses2016.brockport.edu |
| **whoami** | etay1 |
| **who am i** | etay1 pts/1 2022-09-20 23:09 (lb04float-b.brockport.edu) |

3. A command to output a string on a screen is **echo**. The command name is followed by the string(s) to be displayed. For example, **echo Great** will display **Great**. The command **echo Red White Blue** will display **Red White Blue**

Type the following two commands and observe the output:

|  |  |
| --- | --- |
| **Command** | **Output** |
| **echo Great** | Great |
| **echo Red White Blue** | Red White Blue |

In the first case, the command had no options, but one argument. In the second case, the command had no options, but three arguments.

Also we were lucky that the strings did not have any **special character** of significance to the shell. Some special characters can cause problems. Type the following commands and note how each one runs into trouble:

|  |  |
| --- | --- |
| **Command** | **Output** |
| **echo I need $50** | I need 0 |
| **echo Great!!** | echo Greatecho I need $50  Greatecho I need 0 |
| **echo <Drake>** | -bash: syntax error near unexpected token `newline' |

It turns out there are a large number of characters for which the shell attaches special meaning. As we explore Bash features we will understand them. To be safe, I suggest you **quote** all characters other than alphabetic (a .. z and A .. Z) and digits (0 .. 9).

There are three ways of **quoting**:

* Using backslash escape sequence: It preserves the literal value of the next character alone
* Using double quotes: It preserves the literal value of all characters other than dollar sign (**$)**, backquote (**`)**, and exclamation (!). That is, the shell will continue to associate the special meaning associated with dollar sign (shell variable value substitution), backquote (command output substitution) and exclamation (history substitution)
* Using single quotes: It preserves the literal value of all characters.

Type the following commands and note how each one works correctly:

|  |  |
| --- | --- |
| **Command** | **Output** |
| **echo I need \$50** | I need $50 |
| **echo Great\!\!** | Great!! |
| **echo \<Drake\>** | <Drake> |

Type the following commands and note how the first two fail, but the third one works correctly:

|  |  |
| --- | --- |
| **Command** | **Output** |
| **echo "I need $50"** | I need 0 |
| **echo "Great!!"** | echo "Greatecho "I need $50""  Greatecho I need 0 |
| **echo "<Drake>"** | <Drake> |

As we said before, within double quotes dollar sign will continue to be interpreted to mean shell variable value substitution, and exclamation will be interpreted as history substitution. That is why they fail here. But there are times when we may desire exactly that, as we will see later.

Type the following commands and note how each one works correctly:

|  |  |
| --- | --- |
| **Command** | **Output** |
| **echo 'I need $50'** | I need $50 |
| **echo 'Great!!'** | Great!! |
| **echo '<Drake>'** | <Drake> |

4. The shell maintains several variables. For example, consider the variable **USER** that keeps track of your **Net\_ID** or your username. Unlike programming languages such as Java, to access the value associated with a shell variable, we need to use a dollar sign (**$)**in front of the variable name. That is, we write **$USER** to refer to the value associate with the variable **USER**.

Type the following commands one by one and report the output:

|  |  |
| --- | --- |
| **Command** | **Output** |
| **echo $USER** | etay1 |
| **echo $HOME** | /home/etay1 |
| **echo $PS1** | [\!] [\u@\h:\w]\$ |

When you type the command **echo $USER** the shell does the following**:**

* It looks up the value of the shell variable **USER**, which is your username
* That value gets substituted in the command. That is, the command is changed to read as echo followed by your username.
* Thus, when the echo command is executed, it outputs your username.

This is what we mean by shell variable value substitution.

5. The shell variable **?** maintains the exit status of the previous command executed. It has a 0 value if the previous command executed properly. It has a non-zero value, in the range 1-255, if the previous command was not executed successfully. The value of this variable is printed by typing **echo $?**

1. Type a command: **bingo**

It is not a valid UNIX command and must fail (unless you have created a new executable named bingo). An error message will be displayed.

1. Display the value of the exit status variable by typing an appropriate command:

Command to display the value of the exit status of the previous command:

echo $?

Value displayed: 127

1. The previous command can be re-executed by typing two exclamation marks. That is, type !!

The shell will show the command re-executed and then carry it out.

Value displayed: 0

The value has changed to 0.

Reason for change in the value: the previous command executed

6. We want to create a file named **empty** of size zero bytes.

1. Type the command: **touch empty**
2. We also want to create another file whose name has a blank space in between, specifically, the name of the file is: **zero data**
3. Note that if you type **touch zero data** the shell will assume that you want to execute the **touch** command with two arguments and hence will create two files, one named **zero** and the other named **data**. But we have to make it appear as if there is only one argument. Determine and type an appropriate command.

Command to create the file named **zero data**: ­­­­­­­­­­­­­­­­­­­­­­­­­­ touch "zero data"

1. Type the command to do a directory listing (in long format) that shows all hidden and non-hidden entries under **asgn03**, which is the directory you are currently in.

Command typed: ls -a

Output observed (write the entire line) for the two files created in this step:

. .. empty zero data

7. Remove the two files you just created.

Command typed: rm empty

rm ‘zero data’

1. Type a command to do a directory listing that shows all hidden and non-hidden entries under **asgn03**, which is the directory you are currently in. (No need for long format.)

Output observed (write the entire line):

Blank

. ..

8. Remember that you are not allowed to change directory from **asgn03**. Without changing directory, perform the long listing of the file named **passwd** located in the directory **etc** which in turn is under the **root (/)** directory. Write the complete output and explain each field as to what it stands for.

Command for long listing of file: ls -l /etc/passwd

Output observed: -rw-r--r-- 1 root root 2245 Jul 26 13:16 /etc/passwd

Explain all the fields: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

“ – “ Indicates it’s a file not a directory

Read and write permissions

User root and group root

2245 bytes in the file

Date Last Modified?: Jul 26 13:16

/etc/passwd is the file name

9. Again, without changing directory, perform the long listing of the file **passwd** which is located in **/usr/bin** directory. Report and explain each field. In particular, explain the access permission that reads as “s”. Google search **unix s permission** and read.

Command for long listing of file: ls -l /usr/bin/passwd

Output observed: -rwsr-xr-x 1 root root 27856 Nov 22 2019 /usr/bin/passwd

Explain all the fields:

read, write, execute - owner

-xr execute and read permissions – group

-x execute – others

1 hard link

User root and group root

27856 bytes

Nov 22 2019

File name is in red

Explanation for **s** permission: execute permissions for the user

10. The UNIX command **file** looks into a file and makes an educated guess about the nature of the file. In its simplest form, the command name is followed by one argument – the name of the file to be checked. Make use of this command and guess the file type for each of the following files, one by one: the two passwd files in the previous two steps, and the four program files **hello.bash**, **hello.py**, **hello.c** and **Hello.java** created in **asgn01** and **asgn02**. (You need to type the command six times for the six different files. Use absolute path for the first two and relative path for the last four cases.) If the **file** command guessed incorrectly for any of the files, report that.

Use of **file** command:

Command: file /etc/passwd Output: /etc/passwd: ASCII text

Command: file /usr/bin/passwd Output:

/usr/bin/passwd: setuid ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.32, BuildID[sha1]=99f5e0979ba7cd3678db71739651a3f7decfd66d, stripped

Command: file hello.bash Output:

hello.bash: Bourne-Again shell script, ASCII text executable

Command: file hello.py Output:

hello.py: Python script, ASCII text executable

Command: file hello.c Output: hello.c: C source, ASCII text

Command: file Hello.java Output: Hello.java: ASCII text

Any incorrect guess by the command **file**?

Java didn’t get any identifiers other than ASCII

11. Consider my directory **~nyu/csc209/asgn03**

1. Do an **ls** command with long option to display the information of only my **asgn03** directory

Command: **ls -l ~nyu/csc209/asgn03**

Output: total 8

-rw-r--r-- 1 nyu domain users 797 Sep 11 2017 GCD2.c

-rw-r--r-- 1 nyu domain users 481 Sep 11 2017 GCD.c

1. Under my **asgn03** directory, I have a file named **GCD.c** Do an **ls** command with long option to display hidden and non-hidden entries under **asgn03** to see what else is available in that directory. Report.

Command to list my **asgn03** directory: ls -a ~nyu/csc209/asgn03

Four items listed (complete listing):

. .. GCD2.c GCD.c .test

12. Without changing directory, try to move the file **GCD.c** from my directory into your **asgn03** directory with the same file name. You will fail. Report the error observed.

Valid move command: mv ~nyu/csc209/asgn03/GCD.c asgn03

Error reported: mv: cannot move ‘/home/nyu/csc209/asgn03/GCD.c’ to ‘asgn03’: Permission denied

Reason for failure (explain in terms of missing access permissions; see output of Step 11a that shows that my **asgn03** directory has **rwx r-x r-x** permissions on it. State explicitly what permission on **asgn03** would have allowed you to successfully move the file **GCD.c**)

The group other was not given the write permission it would have to be rwx r-x rwx. Instead of what it is now

13. Without changing directory, try to copy the file **GCD.c** from my directory into your **asgn03** directory with the same file name. You should succeed.

Valid copy command: cp ~nyu/csc209/asgn03/GCD.c ~/csc209/asgn03

To check that you are successful, do a long listing of the **GCD.c** file in your directory

Command: ls -a

Output: . .. GCD.c

14. Compile the C program file to create an object file **GCD** (see Assignment 2 for the required command to compile a C program). Remember the option to be used to create an object file, and name the object file **GCD**. Report.

Command to compile the code: cc GCD.c -o GCD

15. The program determines the GCD of two given positive integers. The program will prompt for the two integers, compute and print the result. Execute the program by typing the name of the executable file:

Command to execute the file: ./GCD

1. Provide the two integers 12 and 18 and obtain the result.

GCD of 12 and 18 reported by the program: The required gcd is 6

1. The program will give wrong results or will be lost in an infinite loop if both integers are not positive. Run the program again and provide the integers -12 and 18 and obtain the result. (Wait till the program terminates and report the wrong result it yields.)

GCD of -12 and 18 reported by the program:

The required gcd is -2147483638

16. Run the program again and provide the two integers 12 and 0. In this case the program will **not** terminate. After a minute, suspend execution of the program by typing **Control-Z** The program is only suspended and not terminated. It can be continued from where it got suspended. Type the command: **jobs** It will show one suspended job. Resume the job by typing the command: **fg %1** After another minute, terminate execution of the program by typing **Control-C**  Type the command **jobs** again to ensure that there are no suspended jobs.

How to suspend execution? **Control-Z**

How to check suspended jobs? jobs

How to bring suspended job to foreground? fg %1

How to terminate execution? Control-C

How did you check for suspended jobs? By typing: jobs

Are there suspended jobs?

[1]- Stopped ./GCD

[2]+ Stopped ./GCD

17. Without changing directory, try to copy the file **.test** from my **asgn03** directory into your **asgn03** directory with the same file name. You will fail. Report the error observed and explain the reason for failure. (Again, focus on the permissions on **.test** and state explicitly what permission on **.test** would have allowed you to successfully copy that file.)

Valid copy command: cp ~nyu/csc209/asgn03/.test ~/csc209/asgn03

Error reported: cp: cannot open ‘/home/nyu/csc209/asgn03/.test’ for reading: Permission denied

Reason for failure (explain in terms of missing access permissions; see output of Step 11b)

Access permission for others: -

In other words no access permissions

18. Without changing the directory, using a relative path and just one command, try to determine the number of lines, number of words and number of bytes in the file **hello.c** you created under **asgn02** in Assignment 2.

Command to determine the needed information: wc ~/csc209/asgn02/hello.c

Output: 8 16 106 /home/etay1/csc209/asgn02/hello.c

1. Try the same **wc** command again with each of three options **-l**, **-w**, and **–c** one by one, and report what you determine.

**wc** command with **–l** option: wc -l ~/csc209/asgn02/hello.c # of lines: 8

**wc** command with **–w** option: wc -w ~/csc209/asgn02/hello.c # of words: 16

**wc** command with **–c** option: wc -c ~/csc209/asgn02/hello.c # of bytes: 106

1. Read the man page for the **wc** command and determine what option **-m** will help determine.

Command needed to read man pages: man wc

What will **wc** **–m** help determine: character count

**wc** command with **–m** option: wc -m ~/csc209/asgn02/hello.c # of chars: 106

**Note**: For ASCII files, each character occupies one byte. Hence the byte count and character count will be the same. However, for UNICODE files that will not be the case.

19. Without changing the directory, using a relative path, obtain the long directory listing of **hello.c** you created under **asgn02** in Assignment 2. There should be only one output line. Determine the size of the file **hello.c**. Does it match the number of bytes in the file reported by the **wc** command.

Command for long listing of **hello.c**: ls -l ~/csc209/asgn02/hello.c

Output observed:

-rw------- 1 etay1 domain users 106 Sep 13 21:49 /home/etay1/csc209/asgn02/hello.c

Size of the file in bytes: 106

Does it match the number of bytes reported in Step 18 with **wc –c**: yes

20. Without changing the directory, using a relative path, try to display **hello.c** you created under **asgn02** in Assignment 2, with line numbers. Use the **cat** command with **–n** for this.

Command to display **hello.c**: cat -n ~/csc209/asgn02/hello.c

Number of lines in **hello.c** as shown in the output: 8

Does it match the number of lines reported in Step 18 with **wc –l**: yes

1. Use the **od** command with option **–c** to display the same file. The **od** (octal dump) command with **–c** option will display the file byte by byte, with appropriate character interpretation. It will display 16 characters per line. The column of numbers on the left represents byte offsets of the first byte in each line, in octal. The last row shows the size of the file in terms of number of bytes, written in octal.

**od –c** command to display file: od -c ~/csc209/asgn02/hello.c

Size of the file, written in octal:

0000000 # i n c l u d e < s t d i o .

0000020 h > \n \t \n i n t m a i n ( i

0000040 n t a r g c , c h a r \*

0000060 a r g v [ ] ) \n { \n \t p r i n

0000100 t f ( " H e l l o , J u p i

0000120 t e r ! \ n " ) ; \n \t r e t

0000140 u r n 0 ; \n } \n \n

0000152

Convert the above octal number to decimal and show the steps:

0000000 = 0

0000020 = 2 \* (8^1) = 16

0000040 = 4 \* (8^1) = 32

0000060 = 6 \* (8^1) = 48

0000100 = 1 \* (8^2) = 64

0000120 = 1 \* (8^2) + 2 \* (8^1) = 80

0000140 = 1 \* (8^2) + 4 \* (8^1) = 96

0000152 = 1 \* (8^2) + 5 \* (8^1) + 2 \* (8^0) = 106

Size of the file, written in decimal: 106

Does it match the size of the file reported in Step 19: Yes

1. Using the **od** command with option **–c** will also interpret control characters. You should see several newline characters (shown as **\n**) that mark the end of each line. Count the number of new line characters seen.

Number of newline (**\n**) characters observed: 8

Does this above match the number of line reported above in Step 18 with **wc –l** : Yes

|  |
| --- |
| In UNIX, each file is merely a sequence of bytes. When interpreted as a sequence of characters, a file may be viewed as a sequence of lines. Each line has a sequence of characters, with the newline (**\n**) character used to mark the end of each line. UNIX does not use any special character to mark the end of the file. That is why UNIX must keep track of the exact size of the file in terms of the number of bytes. |

1. Do you see any special character at the end of the file to mark the end of the file? No