Introduction to Scientific Computing (BSR1015) Homework #1

Assigned January 17th, Due January 31st

1. What is a "shell"? Compare and contrast it with how you use your graphical desktop on a daily basis in terms of functionality, purpose, positives, and negatives.

* **Shell** is an out layer around the operating system [kernel](https://en.wikipedia.org/wiki/Kernel_(operating_system)), or a user interface for access to an operating system to allow users to directly communicate with the hardware. When Shell gathers input from the user, it will simultaneously launch programs onto kernel abstraction so users do not have to manually initiate programs each time.
* There are two categories of user interface in the operating systems, either [command-line interface](https://en.wikipedia.org/wiki/Command-line_interface) (CLI) or graphical user interface (GUI). While Unix and Windows practice both, Unix generally relies on use of [tex](https://en.wikipedia.org/wiki/Command-line_interface)t shell (CLI), whereas Windows depends on GUI for user interface. The differences between these two interfaces are the ways they present instructions to the operating system and visualization of the interface. Text shell (CLI) features [alphanumeric](https://en.wikipedia.org/wiki/Alphanumeric) texts as command lines typed on a keyboard, while GUI characterizes graphic bars or icons for the interface.
* For the advantages, CLI is relatively more efficient and speedy than GUI. CLI runs numerous programs at once and advances manipulation of files and remote access to another computer in the network. However, the pre-requisites of the CLI in scripting commands, syntax and keyboard controlling easily leading to unexpected results due to typos are not easy for new users. Comparatively, GUI is user-friendly. The visually intuitive windows and manipulation with a mouse and keyboard make GUI more favorable for beginning users.
* While Unix and Windows both allow performances in file management and process management, process behaviors are different between them. In Linux, child processes are forced to exit with their parent processes if parent processes are closed. In Windows, the child processes do not store information about parent processes, so when users close a parent process, the child process will still continue (Euphoria Reload3d).

1. Describe some of the features of Unix/POSIX and how the shell helps the user leverage these features. Include at least standard IO, pipes, variables, and paths.

* **POSIX** or Portable Operating System Interface is a set of standards codified by the IEEE and issued by ANSI and ISO, unifying all the UNIX-like systems for how the operating systems should function, and how processes are created across manufacturers. **Unix** is elected as a standard basis for system interface because of its manufacturer-neutral nature. Now, UNIX has branched into a large family of Unix derivatives, including Linux, Mac OS X. Unix system includes these features:

1. Multi-tasking- perform various tasks simultaneously
2. Multi-user capability- multiple users can run multiple programs, perform various tasks
3. Shared libraries- dynamically linked libraries at the run-time, not the compilation time, and every program can access the functions in a special place of memory space, without having multiple copies of them
4. Security- accessing others’ data is impossible without permission
5. Virtual memory efficiency- a modest amount of memory is consumed for all the programs
6. Portability- Unix can be run on different computers and machines due to the use of high-level language
7. Hierarchical file system- files are located in directories, when directories

* The role of **shell** in the Unix system is a command-line interpreter that provides [user interface](https://en.wikipedia.org/wiki/User_interface) for users to run commands, programs, and shell scripts in the environment and the shell will collect these inputs and interpret the commands to execute programs. The shell will display the program's output, when a program finishes execution.
* **Standard I/O**is used to redirect the programs without rewriting the program by applying an input and to capture its output. The command output processed is saved to a created temporary file, and the file name as parameter instead of a command is passed down. When we execute the command with the file name, we use the file descriptors, 0 as the standard input (stdin), carrying data from a user to a program, 1, the standard output (stdout), writing the data generated by a program to another program, and 2, the standard error (stderr), writing errors generated by a program. stdout and stderr are connected to the same output devices but stderr is not redirected when stdout is. The disadvantage of this type of redirection is the burden in time-consuming creation of one or more temporary files.
* The **pipe** is another form of redirection that chains the programs and libraries together by the standard streams for redirecting the output (including commands and files) from each process as input ([stdin](https://en.wikipedia.org/wiki/Stdin)) to the next. Note that [stderr](https://en.wikipedia.org/wiki/Stderr) of the processes in a pipeline are not passed through the pipe. Pipes store data temporarily to pseudo files without actually writing to the disk. This direct connection between programs allows operations of data being continuously transferred between programs without the need to pass through a temporary text file or through the display screen and to wait for a program being completed before another program starts. This temporary direct connection between programs enables performance of highly specialized task that the constituent programs could not perform by themselves. Here is the example of pipe use: the command “ls -l | grep aaa | less” suggests the contents in current directory being piped to the [grep](http://www.linfo.org/grep.html) command, using to search “aaa” and “aaa” argument tells grep to return only those lines printed with “aaa”.
* **Variables** are the way to pass information from the shell to programs when you run them. They can be a character string to which we assign a value or a pointer to actual data. There are three types of UNIX variables, shell, local and environmental variables.

Environmental variables are defined for the current shell and are inherited by any child processes or subshell/shell script launched by a shell. Environment variables can be set by the users, by the shell or any program. They have names with upper case and are accessed with **$** sign. These variables remain accessible until we log out of the shell. We can use **env** command to find environmental variables, such as **PATH** that specifies the directories for the shell to search for commands. **LD\_LIBRARY\_PATH** isthe user-set variable that lists multiple directories for run-time shared library loader to look for when searching for shared libraries.

1. What is process substitution? Why would you use it? How does process substitution differ from piping command together? Why would you choose one method of process substitution over another? Give an example of when you may use one method vs. another.

Process substitution is a form of redirection not specified by POSIX. Compared to command substitution, process substitution feeds the output of a process to another process (or send the results of a command to another command), instead of setting a variable to the result of a command (Linuxtopia). Process substitution does not rely on creating temporary files to hold the intermediate output to feed on subsequent commands but feeds the subsequent commands with the command (stdout) appeared as a file. The input or output of a command is read as a filename; files can directly be read from or write to another program. >() is the form that writing of the file will be passed to the substitution as the input; <() is the form that the file passed as an argument will be read from substitution to obtain the output. The feature is important for the command that reads from or writes to a file only and not stdin/stdout. This concurrent and parallel nature generally speeds up the overall action of process. Unlike pipe, process substitution can take multiple inputs and branch of processing into two different pipelines i,e duplicate the intermediate results on stdout and process them differently (Jumping Bean). Process substitution can compare the output of two different commands or the output to the same command. Without the need to run the created [named pipe](https://en.wikipedia.org/wiki/Named_pipe) as input, process substitution increases simplicity and disk space.

Typically, process substitution is used to connect multiple commands to a single command. Despite similarity, process substitution can be different from piping command together, for piping command together will send input to command only, which prevents the command that requires input as parameter, such as ldd to be processed. Process substitution can further compare the contents of two directories where different files are the output with the use of one command line, i.e. diff -u <(ls -l /home/tux) <(ls -l /home/tuxbk) (Jumping Bean). On the contrary, piping commands together is useful to highlight usage of pipes. Some tasks require running data through several tools to carry on different processing (StackExchange), and pipe may work best with filters and translators.

1. In class, we investigated soft linking. What are the differences between soft and hard linking? What are the advantages and disadvantages of a hard link? When might you use one? When would you not use one?

Hard link refers to direct linkage to the original inode (file or directory) on the disk. Without the path name to the original object, the number of inode is identical to the hard link. Changes made on the original file will be reflected on the hard links. Hard links have actual contents. Removing the source file will not affect the contents of the file. Similarly, removing the links will not affect other links or source files. In order to delete the original hard-linked object from the file system, users have to first remove all the hard links to it. Nevertheless, it is relative safer to use hard links for the wellbeing of data. Data can be lost if the links to the files ae deleted. While hard links are comparable to the main file location, destroying a couple of links will not jeopardize data, unless all the links are gone. In addition, hard links take negligible amount of storage space, since no new inodes are created when hard links are made (I used hard links to store a single file object in presence of multiple places, i.e. store the same video with various categories, by year, genre, cast, directors without reducing disk space). Performances of hard links are superior while users directly access the disk instead of accessing via another file. While hard links cannot cross file systems, they can handle moving around the file location on the same filesystem, while soft links cannot.

Soft link refers creation of file that contains the information (e.g file name and location) linked to another file. The inode number of soft link differs from source file. Once the original file is deleted, the soft links will be removed. Reversely, if the soft link is removed, the original file won’t be affected. This characteristic is a blessing and a curse. If the user moves the object to a different location, the soft link without the inode name of the object will be broken, and the system won't be able to find the object. Otherwise, users can recreate a file with the same name as the old file in the same location for new data, and the soft link will still be functioning. Unlike hard links, soft links can link files across the filesystems, and importantly soft links are specific to directories (Sources: Lifewire).

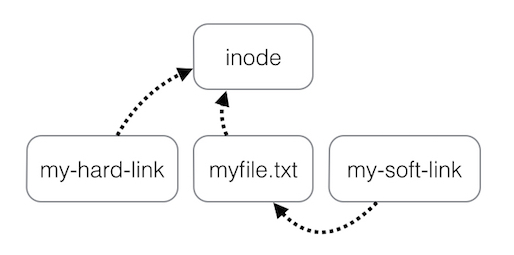


Fig 1. Visualization of mechanisms between hard and soft links found online

1. Describe the permissions, including directory traversal, for the following permission sets, and provide the octal representation.

Definition for permissions:

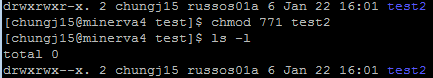
|  |  |  |  |
| --- | --- | --- | --- |
|  | Read | Write | Execute |
| File | Open file and read only | Open and modify or delete contents | Run the file. May not see the contents |
| Directories | List the files in a directory. see their names, sizes, permissions, etc. | Change the list of files in a directory. Create, delete, rename and modify files | Use the directory. Cannot modify or see the list of files. |

* **rwxrwx—x**

-permission:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Owner | group | others |
| Read | Yes | Yes | No |
| Write | Yes | Yes | No |
| Execute | No | No | Yes |

-octal representation: 771

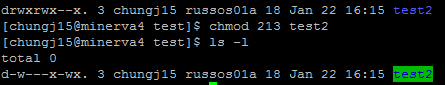
-directory traversal: 

* **-w---x-wx**

-permissions:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Owner | group | others |
| Read | No | No | No |
| Write | Yes | No | Yes |
| Execute | No | Yes | Yes |

-octal representation: 213

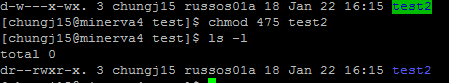
-directory traversal: 

* **r--rwxr-x**

-permissions:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Owner | group | others |
| Read | Yes | Yes | Yes |
| Write | No | Yes | No |
| Execute | No | Yes | Yes |

-octal representation: 475

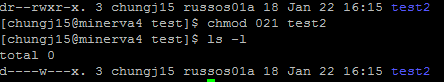
-directory traversal: 

* **----w---x**

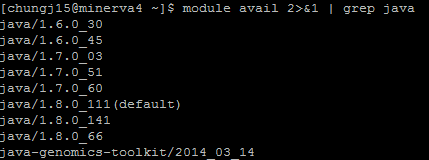
-permissions:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Owner | group | others |
| Read | No | No | No |
| Write | No | Yes | No |
| Execute | No | No | Yes |

-octal representation: 021

-directory traversal: 

1. Write a single-line command that uses commands, pipes, and redirects to search all available modules on Minerva for all versions of java, and outputs them to the screen, one per line.



1. Try the command below. Does it do what you expect? Why or why not? How is it different than “ldd $(which env)”?

ldd <(which env)

The difference between <() and $() is that <() returns the commands into a virtual variable; the name of the file is passed as an argument to the current command, whereas $() returns the command into an unnamed file, where the output is placed in the original command. Hence, “ldd <(which env)”will print “not a regular file” since the env in () doesn't refer to the original location. Instead,“ldd $(which env)” will get the results of what we expected since the env inside the () is the original path and this command is comparable to “ldd /bin/ env”.





1. Why do we need to do “./datescript” to execute the script you saw in Class #2/3’s slides? Why not just “datescript”?

To be executable, a shell script file must contain shell commands. The command does not look into every directory with that file name but the one you specify. “.” tells the command processor that the desired script is located in the current directory. This trick provides the location of shell script as well as its name, so even the file is not in $PATH, the shell knows where to look.

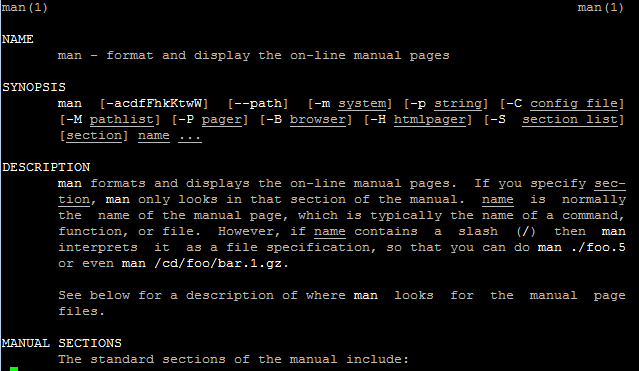


If I just type “datescript”, the command will not be found as below.

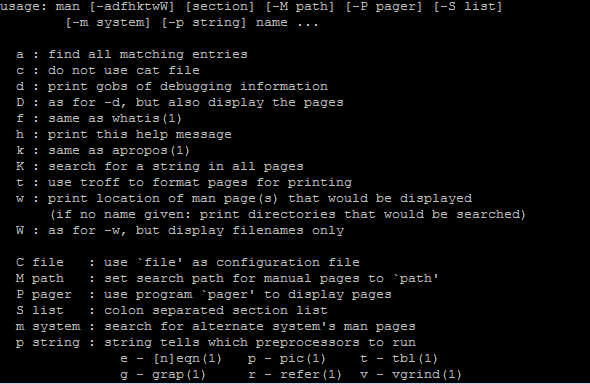


1. Give two possible ways to get help or learn about the function of a given command/program.

One of the basic ways to learn about function of a command is “man (command)”, which displays detailed manual pages for each command. Here is the example of man of the man command.



Another way to look up functions of a command is to type –h or –help following any command. “command –h | less” will display less, and concise information needed.



1. Why is it safer to remove a directory using “rmdir” than a recursive call to “rm”?

rmdir removes only the empty directories, not files, while rm removes files and rm -r recursively removes entire files within the directory. It is safer to use rmdir because it informs the users whether the file is empty to avoid accidental deletion of a directory. Rm will not prompt the user before removing files unless users alias rm to rm –i to inform rm to prompt the users before removal. Alternatively, users can unalias rm –i or choose not to be prompted before file removal by adding the –f flag to the rm command, such as rm –f. Likewise, rm -rf removes the entire directory the contents within it.