UNIX Assignments: Day 1

Concept: Basic commands in UNIX, Filters, Pipes

Objective: At the end of the assignment, participants will be able to:

- Execute Basic Unix commands.
- Implement the concepts of Pipes and Filters
- Work with vi editor

Problems:

Section 1:

1. List all the files and sub directories of the directory /bin.

ls `-R /bin`

`here is an explanation of each component of the ls -R /bin command in Linux:`

- **`ls`** is a command that lists the **files and directories** by **default** in **alphabetical order** in the **current directory** or a specified directory.
- **`-R`** is an **option** used with the `ls` command to list **files and directories recursively**. This means that it will list all files and directories in the specified directory as well as any **subdirectories** it contains.
- **-r** is used to list in reverse alphabetical order
- **`/bin`** is the **directory** that the **`ls`** command will be **executed** on. In Linux, **`/bin`** is a **directory** that contains **essential command-line utilities** that are required for the system to **boot and run**.

Therefore, the `ls -R /bin` command lists all files and directories in the `/bin` directory and its subdirectories, recursively. The output will include all the files and directories in the `/bin` directory as well as any subdirectories it contains.

ls : lists all files and directories in current folder by default in alphabetical order

ls -r : like ls just that in reverse alphabetical order

ls -R : like ls, lists in default alphabetical order **and** also lists the **files within the sub directories** and so on

Symbolic link => Symlink

ls -l /bin/X11/apport-collect

lrwxrwxrwx 1 root root 10 Mar 22 17:07 /bin/X11/apport-collect -> apport-bug

The `l` at the **beginning** of the **file permissions string** in the output of the `ls` command represents that the file is a symbolic link, also known as a "**symlink**". Symbolic links are a **type of file** that **acts as a reference or shortcut** to another **file** or **directory** on the system.

In your example, the file `/bin/X11/apport-collect` is a symbolic link that points to the file `/usr/bin/apport-bug`. The **arrow (`->`)** indicates the **link between the symlink** and **the file it points to**.

The **permissions** for a **symbolic link** in Linux consist of the `l` character in the beginning, followed by the **permissions of the file it points to**. In your example, the permissions for the file `/usr/bin/apport-bug` are `rwxr-xr-x`, which are displayed after the symbolic link permissions.

total 115

-rwxrwxrwx 1 a7 a7 1483 Mar 31 08:52 Armstrong.class

Here's an explanation of each component of the Linux output you provided:

- `total 115`: This is the total size of all files in the directory in 1KB blocks. In this case, it's 115KB.

- `-rwxrwxrwx`: This is the file's permissions. The first character indicates whether the file is a directory (`d`) or a regular file (`-`). The next three characters (`rwx`) indicate the owner's permissions (read, write, and execute). The next three characters (`rwx`) indicate the group's permissions. The last three characters (`rwx`) indicate everyone else's permissions.
- `1`: This is the number of hard links to the file.
- `a7 a7`: This is the owner and group of the file.
- `1483`: This is the size of the file in bytes.
- `Mar 31 08:52`: This is the date and time when the file was last modified.
- `Armstrong.class`: This is the name of the file.

Symlink

A symlink (short for symbolic link) is a **type of file** that **serves** as a **pointer** to **another file** or **directory**. When you create a symlink, you are essentially creating a **shortcut** to **another file** or **directory**. Symlinks are useful in many situations, including:

1. Providing a more convenient or memorable way to access a file or directory that is located in a different directory or on a different disk.
2. Allowing you to create **multiple paths** to the same file or directory, which can be useful in cases where you need to reference a file or directory from different locations.
3. Providing a way to redirect access to a file or directory when the original file or directory is moved or renamed.

Now, let's compare symbolic links to hard links:

A **hard link** is a **type of link** that **points** directly to the **physical location** of a **file on disk**. When you create a hard link, you are **essentially creating a second copy** of the **file** with a **different name**. Because **hard links point directly** to the **physical location of the file**, any **changes made** to the **original file** will **also be reflected** in the **hard link**, and **vice versa**. This means that hard links are **essentially the same as the original file** and **cannot be distinguished** from it by the **operating system**.

The main differences between symbolic links and hard links are:

1. Symbolic links are files that point to **another file or directory**, while hard links are **additional entries** in the filesystem's directory structure that point to the **same physical file**.
2. **Symbolic links** can **point to files or directories** on **different filesystems**, while **hard links** can **only point to files on the same filesystem**.
3. **Symbolic links** can be created for **directories as well as files**, while **hard links** can **only** be **created for files**.

4. When you delete a file that has one or more hard links, the file itself is not actually deleted until all hard links to it are removed, while deleting a file that has symbolic links will not affect the symbolic links.

In summary, symbolic links and hard links are both useful tools for managing files and directories in Linux, but they have different properties and are used in different situations.

- Creating a **symlink** for a **file** in **another directory**

ln -s [original file path if not current dir] [symbolic link path if not current dir]

ln command

Here's a breakdown of the different parts of the command:

ln: The command for creating links.

-s: The option that specifies that we want to create a symbolic link.

[original file]: The name of the file or directory that we want to create a symlink for.

[symbolic link]: The name of the symlink we want to create.

Here's an example that creates a symbolic link for a file named example.txt in the home directory:

ln -s /home/user/example.txt /home/user/symlink

This will create a symbolic link named symlink in the home directory that points to the original file example.txt.

Note that the **-s option** is **important** to specify that we want to create a **symbolic link**, as opposed to a **hard link**, which is created **by default** if you **don't specify -s**. Hard links are not the same as symbolic links and have some different properties and limitations.

ln -s /media/a7/New\Volume/DUAL_BOOT_LINUX/linux_codes/directory_1/file_symlinkcreted_of.txt /media/a7/New\ Volume/DUAL_BOOT_LINUX/output_sym_file.txt

New Volume == 'New Volume' == "New Volume"

- Create a symlink for a directory

ln -s **/media/a7/New\ Volume/DUAL_BOOT_LINUX/JavaHome/ /home/a7/Desktop/JavaHomeShortcut**

- Create a hardlink

Error:

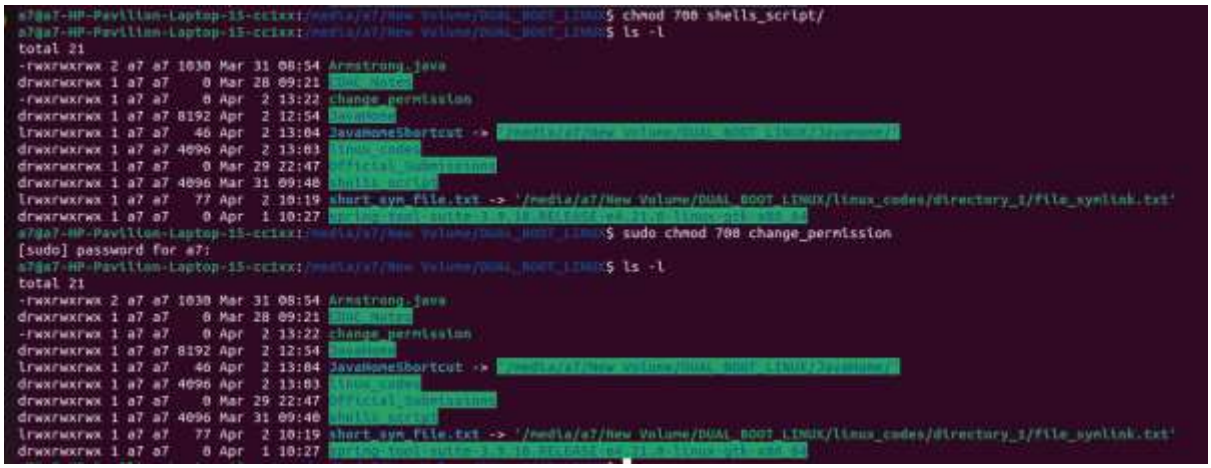
ln /media/a7/New\ Volume/DUAL_BOOT_LINUX/JavaHome/Armstrong.java /home/a7/Desktop/Armstrong_hard.java

ln: failed to create hard link '/home/a7/Desktop/Armstrong_hard.java' => '/media/a7/New Volume/DUAL_BOOT_LINUX/JavaHome/Armstrong.java': Invalid cross-device link

The error message you received indicates that you cannot create a hard link between the two files because they are on **different file systems**. Hard links can only be created between files that are on the **same file system**.

To create a hard link between two files, they must be on the same partition or volume. If the files are on different partitions or volumes, you can create a **symbolic link** instead. Symbolic links **can point to files on different file systems**.

Unable to change file permissions with simple as well as sudo mode:



There could be several reasons why you are unable to change file read and write permissions in the GUI in Linux. Here are a few possible reasons:

1. Permissions: Make sure you have the appropriate permissions to change file permissions. You must have the `write` permission on the file to be able to modify its permissions.
2. File ownership: Make sure you are the owner of the file or have sufficient privileges to modify the file's permissions. You may need to use `sudo` to change permissions on files that are owned by another user.
3. Filesystem: If the file is located on a read-only filesystem, you will not be able to modify its permissions.
4. GUI limitations: Some GUI file managers may not support changing file permissions or may have limited functionality for changing permissions. Try using the command line or a different file manager to modify the permissions.

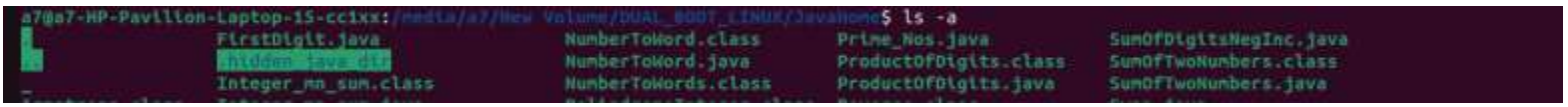
If you are still unable to change the file read and write permissions in the GUI, try using the command line to change the permissions using the `chmod` command. From the output, it looks like the file permission for Armstrong.java has been successfully changed to 700 using the `chmod` command. However, when you use the `ls -l` command to list the files in the directory, the file permission is still listed as `rwxrwxrwx`, which means read, write and execute permissions are granted to the owner, group and other users.

This could be happening because the **file is located on a separate partition or mounted drive where the permissions cannot be changed**. In such cases, you may **need to mount the drive with appropriate permissions or change the permissions of the directory that contains the file**.

You can try changing the permission of the directory where the file is located using the following command:

2. List all the files including hidden files in your current directory.

ls -a

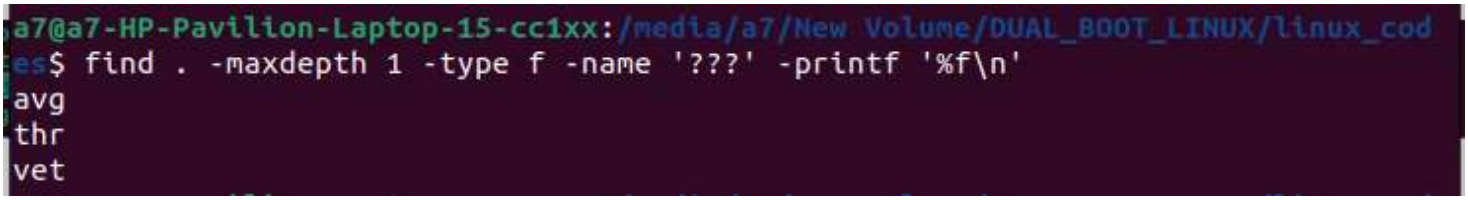


3. List all the files starting with letter ‘r’ in your current directory.

ls -N*

4. List all the files having three characters in their names, from your current directory.

ls -d ???



ls /bin/ ? => lists files with names of just one character

ls /bin/ ?? => lists files with names of just two characters

ls /bin/ ???? => lists files with names of just four characters

then why does 'ls /bin/???' print files with names of any number of characters

find . -maxdepth 1 -type f -name '???' -printf '%f\n'

Sure, let me break down the command for you:

- `find`: This command is used to search for files and directories in a directory hierarchy.
- `.`: This specifies the directory to start the search from, in this case, the current directory.
- `-maxdepth 1`: This specifies the maximum depth of the directory hierarchy to search. Here, it is set to 1, so only the current directory is searched.
- `-type f`: This specifies that we are looking for files only, and not directories.
- `-name '???'`: This specifies that we are looking for files with names that consist of exactly 3 characters.
- `-printf '%f\n'`: This tells the `find` command to print only the name of the file(s) that match the criteria, without any directory information. The `%f` is a format specifier that prints only the file name, and `\n` adds a newline after each file name.

So, when you run this command, it will search for files in the current directory that have names consisting of exactly 3 characters, and print their names without any directory information.

[command] [options] [arguments]

Command: find

Options:

- maxdepth 1: limits the search depth to the current directory only
- type f: specifies that only regular files should be returned
- name '???': specifies that the file name should have exactly three characters
- printf '%f\n': prints only the file name (without path) and a newline character

Arguments: . (current directory)

5. List all the files with extension .doc in your current directory.

file *.java VS **ls *.java** VS **dir *.java**

file *.java

ls *.java

dir *.java

6. List all the files having the first letter of their name within the range ‘l’ to‘s’, from your current directory.

ls [l-s]*

ls [l-sL-S]* : for both upper and lower case

l,s inclusive and case sensitive

7. Copy the contents of file text1 to another file text2.

cp -v PalindromeInteger.java PalindromeInteger_cpy.java

This command uses the "cp" command to copy the contents of file text1 to file text2. The first argument "text1" specifies the source file to be copied, and the second argument "text2" specifies the destination file where the contents will be copied.

If the **file text2 already exists**, the **command will overwrite its contents** with the **contents of file text1**. If the file **text2 does not exist**, the command will **create a new** file named text2 **with** the **contents of file text1**.

Note that if you want to copy the file text1 **along with its permissions** and **other attributes** to **file text2**, you can use the **"-p" option with the "cp" command**, like this:

cp -pv PalindromeInteger.java PalindromeInteger_cpy_permission_attributes_copied_aswell_with_pcommand.java

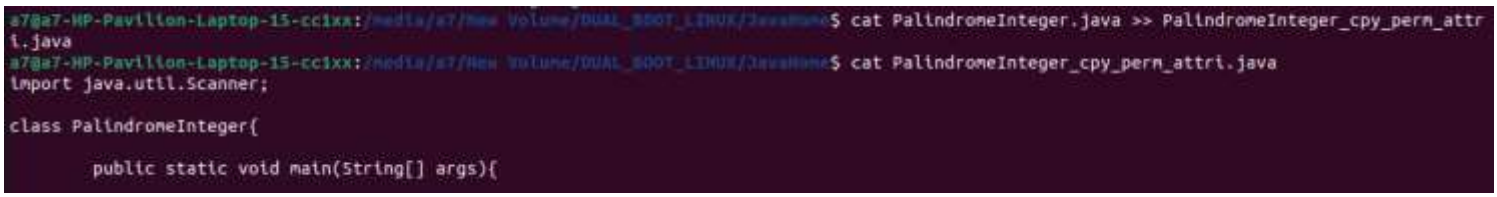
8. Append the contents of file text2 to file text1.

cat text2 >> text1

This command uses the "**cat" command** to **display the contents** of **file text2**, and then **uses the ">>" operator** to **append the output to file text1**.

The "**>>" operator** is used to **redirect the output** of a command to a **file in append mode**. It appends the **output to the end of the file**, **without overwriting** its **existing contents**.

So, in this case, the contents of file text2 will be appended to the end of file text1, and the original contents of file text1 will be preserved.



9. Count the number of files in the current directory.

ls -l | grep "^-" | wc -l

This command uses the "ls" command to list all files in the current directory, the "grep" command to filter out directories and other non-file items from the list, and the "wc" command to count the remaining lines.

Here's a breakdown of how the command works:

1. "ls -l" lists all files in the current directory with detailed information, including the file type.
2. "grep "^-" " filters out all items that do not start with a "-" character. In the output of "ls -l", the "-" character is used to indicate a regular file.

3. The filtered list of files is then piped to "wc -l", which counts the number of lines in the output. Since each line corresponds to a file, the number of lines is equal to the number of files.

The output lines starting with "." and ".." are special entries in a directory, while the line starting with "_" is a regular file or directory in the directory.

- "." represents the current directory, i.e., the directory that you are currently in. When you type "ls -a" in a terminal, you are asking to list all files and directories in the current directory, including the "." and ".." entries.
- ".." represents the parent directory of the current directory, i.e., the directory that contains the current directory. So, if you are currently in the directory /media/a7/New Volume/DUAL_BOOT_LINUX/JavaHome, ".." refers to the directory /media/a7/New Volume/DUAL_BOOT_LINUX.

So, when you run the "ls -a" command in the JavaHome directory, it will display a list of all files and directories in the directory, including the special entries "." and "..", and any regular files or directories that exist in the directory.

10. Display the output of command ls –l to a file and on the output screen.

ls -l | tee filename.txt

ls -l | tee op_on_terminal_aswellas_inthisfile.txt

This command pipes the output of the `ls -l` command to the `tee` command, which displays the output on the screen and also saves it to the file specified in the command (in this case, "filename.txt").

When you run this command, the output of the `ls -l` command will be displayed on the screen, and a file named "filename.txt" will be created in the current directory containing the same output. If the file already exists, the command will overwrite its contents.

11. From file text1 print all lines starting from 10th line.

tail -n +10 text1

```
dac@dac-Veriton-M200-H310:~/cdac/acts$ tail -n +10 file1
-rw-rw-r-- 1 dac dac   43 Mar 24 02:54 file4
-rwx------ 1 dac dac   61 Mar 22 20:49 gamma
--w--w---- 1 dac dac 1718 Mar 21 18:12 ripper.zip
-rwxrw-r-- 1 dac dac   71 Mar 22 20:46 shelling.sh
```

- `tail`: This is a Unix/Linux command used to display the last few lines of a file. It's named "tail" because it's often used to display the "tail end" of a file.
- `-n`: This is an option for the `tail` command that specifies the number of lines to display. In this case, we're using it with a `+` sign (`+10`) to tell `tail` to display all lines starting from the 10th line.
- `+10`: This argument tells `tail` to start displaying lines from the 10th line of the file.
- `text1`: This is the name of the file that we want to display lines from.

So, when you run the command **`tail -n +10 text1`**, it will display all lines starting from the 10th line of "text1". The output will be displayed on the terminal screen.

Section 2:

1. Count the total number of words in file text1.

wc -w text1 : counts words

```
a7@a7-HP-Pavilion-Laptop-15-cc1xx:/media/a7/New Volume/DUAL_BOOT_LINUX/JavaHome$ wc -w Armstrong.java
112 Armstrong.java
```

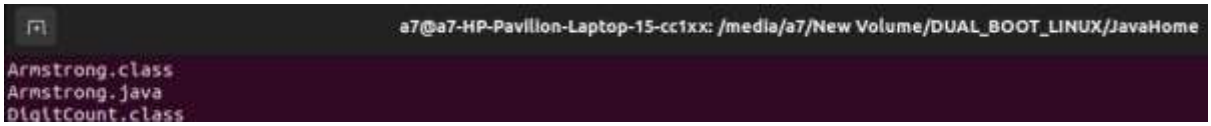
- `wc`: This is the Unix/Linux command that is used to count the number of lines, words, and characters in a file. The name "wc" stands for "word count".
- `-w`: This is an option for the `wc` command that specifies to count the number of words in a file. We're using this option to tell `wc` to count the number of words in "text1".
- `text1`: This is the name of the file that we want to count the words in. You can replace "text1" with the name of any file that you want to count the words in.

So, when you run the command `wc -w text1`, it will count the total number of words in "text1" and display the result on the screen.

If you want to count the number of lines or characters in the file, you can replace the `-w` option with `-l` to count lines or `-c` to count characters.

2. List the contents of ls command page wise.

ls | less

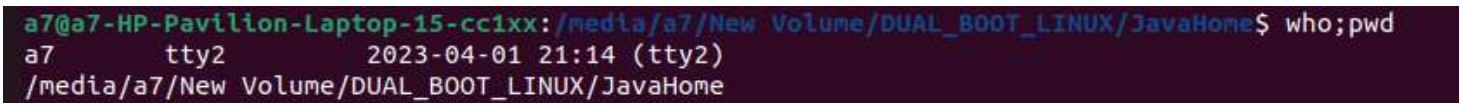


When you run this command, the output of the `ls` command will be displayed one page at a time. You can use the arrow keys or the page up/down keys to navigate through the output. To exit the `less` command and return to the terminal prompt, press the `q` key.

Note that the `|` symbol is called a "pipe" and is used to redirect the output of one command to another command. In this case, we're using the `ls` command to list the contents of the current directory, and then piping the output to the `less` command to view it one page at a time.

3. Using one single command, display the output of “who” and “pwd” commands.

   who ; pwd



The first line of the output shows the output of the `who` command. Here's what each of the columns mean:

- `a7`: This is the username of the logged-in user.
- `tty2`: This is the terminal device that the user is logged in on.
- `2023-04-01 21:14`: This is the date and time that the user logged in.
- `(tty2)`: This is a label indicating the type of terminal device.

The second line of the output shows the output of the `pwd` command, which displays the current working directory. In this case, the current working directory is `/media/a7/New Volume/DUAL_BOOT_LINUX/JavaHome`.

So the complete output of `who;pwd` tells you that the user `a7` is currently logged in on terminal device `tty2`, and that the current working directory is `/media/a7/New Volume/DUAL_BOOT_LINUX/JavaHome`.

When you run this command, the output of the `who` command will be displayed first, followed by the output of the `pwd` command. The semicolon separates the two commands and tells the shell to run them one after the other.

Note that the output of the two commands will not be combined into a single output stream. Instead, you will see the output of the `who` command first, followed by the output of the `pwd` command. If you want to combine the output of the two commands into a single stream, you can use the `&&` operator instead of the semicolon. Here's an example:

who && pwd

When you run this command, the `who` command will be executed first. If it runs successfully (i.e., without any errors), then the `pwd` command will be executed and its output will be displayed. If the `who` command fails (i.e., it returns an error), then the `pwd` command will not be executed.

4. Display the system date in following format:
    2. Today is Friday, 17 May 96

date +"Today is %A, %d %B %y"

5. Use **find** command to locate the following within your home directory tree:
    1. Files with extension .c or .pl

find ~/ -type f \(-name "*.c" -o -name "*.java" \)

- `find`: This is the command itself, which searches for files and directories within a specified path.
- `~/`: This specifies the starting path for the search. The tilde (`~`) is a shortcut that represents your home directory.
- `-type f`: This option tells `find` to only look for files, not directories or other types of files. The letter `f` stands for "file".
- `\( -name "*.c" -o -name "*.pl" \)`: This is an expression that specifies the search criteria. The backslashes and parentheses are used for grouping and escaping special characters.
- `-name`: This option specifies that we're searching by filename.
- `"*.c"`: This is a pattern that matches all files with the extension ".c".
- `-o`: This is an operator that means "or". It's used to combine multiple search criteria.
- `"*.java"`: This is a pattern that matches all files with the extension ".java".

Putting it all together, the `find` command searches for regular files within your home directory (`~/`) that have the extension ".c" or ".pl". When it finds a file that matches these criteria, it displays the full path to that file.

The `\(` and `\)` characters in the `find` command are used to group search criteria together. They are called metacharacters and are used for grouping expressions in the `find` command.

In the example command `find ~/ -type f \( -name "*.c" -o -name "*.pl" \)`, the `\( ... \)` construct groups the search criteria `-name "*.c"` and `-o -name "*.pl"` together. This means that `find` will look for files that match either of these criteria.

The reason for using parentheses is to make sure that the search criteria are evaluated together as a single unit, rather than being evaluated separately. This is important because the `-o` operator has lower precedence than the `-name` operator, which means that if we didn't use parentheses to group the criteria, `find` would first look for files with the extension ".c" and then for files with the extension ".pl", instead of looking for files that have either of these extensions.

So, in summary, the `\(` and `\)` characters in the `find` command are used to group search criteria together and to ensure that the search criteria are evaluated together as a single unit.

We used the backslash `\` before the parentheses `(` and `)` to escape them in the `find` command.

The parentheses `(` and `)` are special characters used to group search criteria together in the `find` command. In this case, we want to group the search criteria for the file extensions `.c` and `.pl` together, and we use the parentheses for this purpose.

However, the parentheses are also special characters in the shell, and they have a special meaning to the shell. Specifically, they are used to start and end a subshell in the command line.

To prevent the shell from interpreting the parentheses in the `find` command as starting and ending a subshell, we use the backslash `\` to escape them. This tells the shell to treat the parentheses as literal characters that are part of the `find` command, rather than shell **metacharacters**.

So, in summary, we use the backslash `\` before the parentheses `(` and `)` to escape them in the `find` command and prevent the shell from interpreting them as starting and ending a subshell.

2. Directories having permission 755

find ~/ -type d -perm 755

```
a7@a7-HP-Pavilion-Laptop-15-cc1xx:/media/a7/New Volume/DUAL_BOOT_LINUX/JavaHome$ find ~/ -type d -perm 755
/home/a7/Public
/home/a7/Documents
/home/a7/Templates
/home/a7/Music
/home/a7/Desktop
/home/a7/snap/snapd-desktop-integration
```

- `find`: This is the command itself, which searches for files and directories within a specified path.
- `~/`: This specifies the starting path for the search. The tilde (`~`) is a shortcut that represents your home directory.
- `-type d`: This option tells `find` to only look for directories, not files or other types of objects. The letter `d` stands for "directory".
- `-perm 755`: This option tells `find` to look for directories that have permission 755. The number 755 specifies the permission bits in octal form, where the first digit represents the file's user permission, the second digit represents group permission, and the third digit represents other permission. In this case, 755 means that the directory's owner has read, write, and execute permission, and that everyone else has read and execute permission.

When you run this command, `find` will search your home directory and all subdirectories for directories that have permission 755. When it finds a directory that matches these criteria, it will display the full path to that directory.

3. Files having permission 655

find ~/ -type f -perm 755

4. Files having inode number 12122

ls -i /path/to/directory/*

find ~/ -type f -inum 30095

```
a7@a7-HP-Pavilion-Laptop-15-cc1xx:/media/a7/New Volume/DUAL_BOOT_LINUX/JavaHome$ ls -i /media/a7/New\ Volume/DUAL_BOOT_LINUX/JavaHome
30108 Armstrong.class                                                            30128 PalindromeInteger.java
30107 Armstrong.java                                                             30103 PerfectNumber.class
30105 DigitCount.class                                                           30104 PerfectNumber.java
```

- `ls`: **This is the command itself, which lists the contents of a directory.**
- `-i`: **This option tells `ls` to list the inode number of each file.**
- `/path/to/directory/*`: **This specifies the path to the directory whose contents you want to list, followed by an asterisk (`*`) to indicate that you want to list all files in that directory.**

When you run this command, `ls` will list the name and inode number of each file in the specified directory, separated by a space. The inode number will appear at the beginning of each line.

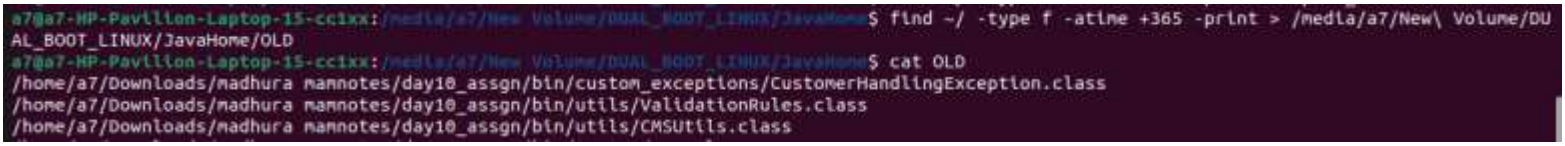
An inode (short for index node) is a data structure used by the file system to store information about a file or directory on a Unix-based operating system. Each file or directory has a unique inode number assigned to it, which is used by the operating system to keep track of the file's location on disk, ownership, permissions, timestamps, and other attributes.

Inode numbers are used by many Unix utilities to refer to files, including `ls`, `find`, `stat`, `df`, and `du`. When you list files with `ls -i`, the first column shows the inode number of each file.

The inode number is an important concept in Unix-based file systems because it allows the operating system to keep track of files even if they are renamed, moved, or deleted. Even if a file is **moved** to a **different directory or renamed**, it **retains** its **original inode number**, which is used to **locate** it on disk. This is different from other file systems, such as those used by **Windows**, which use a **file's name and location** to **identify** it

5. Files which have not been accessed for more than a year and save the list in Old_File

find ~/ -type f -atime +365 -print > ~/Old_File



- `find`: This is the command itself, which searches for files and directories within a specified path.
- `~/`: This specifies the starting path for the search. The tilde (`~`) is a shortcut that represents your home directory.
- `-type f`: This option tells `find` to only look for regular files, not directories or other types of objects. The letter `f` stands for "file".
- `-atime +365`: This option tells `find` to look for files that have not been accessed for more than 365 days. The `atime` option stands for "access time", and the `+365` argument means "more than 365 days ago".
- `-print`: This option tells `find` to print the full path of each file it finds.
- `> ~/Old_File`: This redirects the output of the `find` command to a file named "Old_File" in your home directory. The `>` symbol means "redirect output", and the `~/Old_File` path specifies the file to redirect output to.

When you run this command, `find` will search your **home directory and all subdirectories for regular files** that have not been **accessed for more than a year**. When it finds a file that matches these criteria, it will print the full path to that file and append it to the "Old_File" file in your home directory

6. Files whose size is greater than 1024 bytes

find ~/ -type f -size +1024c -print

- `find`: This is the command itself, which searches for files and directories within a specified path.
- `~/`: This specifies the starting path for the search. The tilde (`~`) is a shortcut that represents your home directory.
- `-type f`: This option tells `find` to only look for regular files, not directories or other types of objects. The letter `f` stands for "file".

- `-size +1024c`: This option tells `find` to look for files whose size is greater than 1024 bytes. The `size` option specifies the file size to look for, and the `+1024c` argument means "more than 1024 bytes". The `c` at the end of the argument specifies that the size should be measured in bytes.
- `-print`: This option tells `find` to print the full path of each file it finds.

When you run this command, `find` will search your home directory and all subdirectories for regular files that are larger than 1024 bytes. When it finds a file that matches these criteria, it will print the full path to that file
The `~/` is a shorthand representation for your home directory. When you use `~/` in a command, it is expanded to the full path of your home directory. This is useful because it allows you to reference your home directory without having to type out the full path every time.

For example, if your username is "john" and your home directory is `/home/john`, then `~/` would be expanded to `/home/john/`.

The `c` at the end of `1024c` specifies that the file size should be measured in bytes. By default, `find` measures file sizes in **512-byte blocks**. However, you can use various suffixes to specify different units of measurement. Here are some common suffixes and what they mean:

- `c`: bytes
- `k`: kilobytes (1024 bytes)
- `M`: megabytes (1024 kilobytes)
- `G`: gigabytes (1024 megabytes)
- `T`: terabytes (1024 gigabytes)

So, in the `find` command we used, `+1024c` specifies that we want to search for files whose **size is greater than 1024 bytes**. The **`c`** at the end of `1024` specifies that we want to measure the file size in **bytes**, rather than the **default 512-byte blocks**

Section 3 :

1. Using vi editor:
    1. Create a file “Data1.txt

vim Data1.txt

    2. Save the file and exit from the vi editor.

ESC :wq

    3. Write some text and and save it to a file “MyData2.txt”

```
echo "save and exit"

echo "writing some text and appending it to a file named 'MyData2.txt'" > Mydata_shello.txt
~
~
~
```

4. Repeat point (c) but after writing some text don't save and just exit "vi"

ESC :q!

2. Create a file using vi editor and enter the following text in it:

Unix Unix Unix Unix Unix

Unix is multi user operating system, Unix is multi tasking o\perating system

Everything on Unix is a file.

Unix File structure is hierarchical like an upside down tree.

Regular files cannot contain another file, or directory

Directory File Contains directory(s) and/or file(s) within it

Device files are used to represent physical devices.

Symbolic link is an indirect pointer to a file

1. Save the file without exiting vi.

ESC :w

2. Display the line number from within vi

ESC :set number | ESC :set nu

ESC :set nonumber | ESC :set nonu

3. Move first three lines of the file to the end of the file.

3dd

p

```
3 lines moved
E20: Mark not set
Press ENTER or type command to continue
```

ESC :1,3m$|'a-1,'bd _

ESC :1,3m$

- `:` enters command-line mode in Vim.
- `1,3m$` is a range that specifies the first three lines of the file (`1,3`) and the `$` address, which represents the end of the file. The `m` command moves the specified range to the end of the file.
- `|` separates the two commands.
- `'<,'>` is a range that represents the last visual selection.
- `d` is a command that deletes the selected lines.
- `"a` specifies that the deleted lines should be saved to register "a".
- `-1` is a range that specifies the line before the cursor position (i.e., the line before the moved lines).
- `,'b` is a range that specifies the first line of the deleted lines to the current line (i.e., the last line of the moved lines).
- `d _` deletes the lines in the specified range and places them in the "black hole" register (represented by `_`).

Here's a step-by-step explanation of what happens when you run this command:

1. The range `1,3` specifies the first three lines of the file, and the `$` address specifies the end of the file. The `m` command moves the specified range to the end of the file.
2. The cursor is now positioned at the end of the moved lines.
3. The `|` separator allows two commands to be executed in sequence.
4. `'<,'>` is a range that represents the last visual selection, which is the range of the moved lines.
5. `d` deletes the selected lines and places them in the "black hole" register.
6. `"a` specifies that the deleted lines should be saved to register "a".
7. `-1` specifies the line before the cursor position.
8. `,'b` specifies the first line of the deleted lines to the current line.
9. `d _` deletes the lines in the specified range and places them in the "black hole" register.
10. The contents of register "a" (the deleted lines) are now put before the cursor position using the `P` command.

After running this command, the first three lines of the file will be moved to the end of the file, and the cursor will be positioned at the beginning of the moved lines.
Sure, here's a detailed explanation of the command `:1,3m$|'a-1,'bd _`:

- `:` - This indicates that we are entering a command in Vim's command-line mode.
- `1,3` - This specifies the range of lines we want to move, i.e., lines 1 to 3.
- `m$` - This moves the specified lines to the end of the file (represented by the `$` symbol).
- `|` - This separates the first command from the second command, allowing them to be executed in sequence.
- `'a` - This indicates that we want to use the `a` register for the next command. The `a` register is a named register in Vim that can be used to store text.

- `-1` - This specifies a relative line number. In this case, it specifies the line immediately above the cursor position.
- `,'b` - This specifies a range of lines, from the first line of the deleted text (i.e., the first line of the moved lines) to the current line (i.e., the last line of the moved lines).
- `d` - This deletes the specified lines.
- `_` - This is a shorthand for the "black hole" register, which is a special register that discards the deleted text without saving it anywhere. In this case, we are using it to avoid cluttering other registers with the deleted text.

In summary, this command moves lines 1 to 3 to the end of the file, and then deletes those lines from their original location, using the `a` register to temporarily store the deleted text
Sure, here's some additional information about the `|` command and the other aspects of the command:

- `|` is not actually a pipe command in Vim, despite using the same symbol as the shell pipe. Instead, it is a command separator that allows you to run multiple commands in a single line. In the given command, `:1,3m$` and `'<,'>d _` are separate commands that are executed one after the other.
- The `-1` argument in the command `'a-1,'bd _` specifies the line number that is one line before the current cursor position. Since the command is executed after we've moved lines 1-3 to the end of the file, the current cursor position is now at the beginning of the moved lines. Therefore, `-1` specifies the last line of the moved lines.
- The `,'b` range specifies a range of lines from the first line of the deleted text (specified by `'.`) to the last line of the deleted text (specified by `'b`). The comma in between the two marks means "to", so `'.` means "the current line" and `'b` means "the line marked with the `b` mark".

In the given command, the `.` mark represents the current line, and the `b` mark is set by the `d` command (i.e., the `d` command sets the mark `'b` to the line just after the deleted text). Therefore, `,'b` specifies a range of lines from the first line of the moved text (represented by `.`) to the last line of the moved text (represented by `'b`).

To move the first three lines of a file to the end of the file using the command line, you can use the following command:

$ sed -n '4,$p' file.txt; sed -n '1,3p' file.txt

This command uses the `sed` utility to print lines from a file. The first part of the command, `sed -n '4,$p' file.txt`, prints all lines starting from line 4 to the end of the file. The second part of the command, `sed -n '1,3p' file.txt`, prints lines 1 to 3 of the file.

By combining these two commands with a semicolon, we can print the lines from 4 to the end of the file first, followed by lines 1 to 3 of the file. This effectively moves the first three lines to the end of the file.

You can redirect the output to a new file or overwrite the existing file with the command `sed -n '4,$p' file.txt; sed -n '1,3p' file.txt > newfile.txt`

4. Copy 5th line and paste above the first line.

5th line ESC yy P

You can copy the 5th line and paste it above the first line using the following steps in Vim:

1. Move the cursor to the 5th line.
2. Type `yy` to yank (i.e., copy) the entire line.
3. Move the cursor to the first line.
4. Type **`P`** **(uppercase)** to paste the copied line **above** the **current line**.

Alternatively, you can use the `:t` command to copy and paste the line in a single command:

1. In command mode, type **`:t 1`** and press Enter. This will copy the current line (i.e., the 5th line) and paste it above the first line.

Note that the `t` command in Vim is used to copy a line and paste it after a given line number. By default, the line is pasted after the current line, so in this case we specify a line number of 1 to paste the copied line above the first line.

5. Search the word ***Unix*** in forward direction

Command mode **/Unix** **n** => next **forward** occurance N => next **backward** occurance

Command mode **?Unix** **n** => next **backward** occurance N => next **forward** occurance

Here, `/` is the search command in Vim, and `Unix` is the pattern that we want to search for. When you press Enter, Vim will search for the next occurrence of the pattern in the forward direction starting from the current cursor position.

You can also search for the next occurrence of the pattern by typing the **`n` command**, or search for the previous occurrence by typing the `N` command.

6. Search the word ***Unix*** in backward direction

Command mode **/Unix** **n** => next **forward** occurance N => next **backward** occurance

Command mode **?Unix** **n** => next **backward** occurance N => next **forward** occurance

7. Replace all the occurrences of the word ***Unix*** with ***UnixOS***

```
%s/Unix/UnixOS/g
```

Sure, here's a breakdown of the `:%s/Unix/UnixOS/g` command:

- ***`%` specifies that the command applies to the entire file.***
- ***`s` is the substitute command in Vim, used for search and replace operations.***
- ***`/Unix/` is the search pattern. In this case, we're searching for the word "Unix".***
- ***`/UnixOS/` is the replace pattern. In this case, we're replacing each occurrence of "Unix" with "UnixOS".***
- ***`g` stands for "global". This flag means that every occurrence of the search pattern should be replaced, not just the first one on each line.***

So, when you run this command, Vim will search for every occurrence of "Unix" in the file, and replace it with "UnixOS" wherever it is found.

Note that if you want to confirm each replacement individually, you can add the `c` flag to the end of the command, like this: `:%s/Unix/UnixOS/gc`. This will prompt Vim to ask for confirmation before replacing each occurrence of "Unix". You can then choose to replace or skip each occurrence as desired.

:%s/Unix/UnixOS/gc

This will ask for confirmation for each occurrence of "Unix" and you can choose to replace or skip each occurrence. To replace all occurrences without confirmation, simply press `y` for each occurrence. To skip an occurrence, press `n`.

UNIX Assignments: Day 2

Concept: Shell Scripting, filters

Objective: At the end of the assignment, participants will be able to understand and
implement:

1. Regular expressions and grep
2. Shell fundamentals
3. Using basic UNIX commands and filters as building blocks
4. The commands as conditions for decision making in shell scripts
5. Shell Scripting constructs

Problems:

1. List only the directories in your current directory

ls -d */

Here, `ls` is the command to list the files and directories in the current directory. The `-d` option is used to list only the directories, not their contents. The `*/` pattern is used to match only the directories, since directories in Linux end with a forward slash (`/`).

When you run this command, it will list only the directories in your current directory, one per line.

2. Display the name and count of the directories in the current directory.

ls -l | grep "^d" | tee >(wc -l >&2) | awk '{print $9}'

- `ls -l`: lists the contents of the current directory in long format.
- `grep "^d"`: searches for lines that start with `d`, which indicates a directory in the long format output of `ls -l`.

- `tee >(wc -l >&2)`: duplicates the output of `grep` to two streams. The first stream is passed to the next command in the pipeline, `awk`, while the second stream is passed to `wc -l`, which counts the number of directories found. The `>(...)` syntax is a process substitution that allows `wc -l` to be executed in parallel with the rest of the pipeline. The `>&2` redirects the output of `wc -l` to the standard error (stderr) stream instead of stdout, so that it doesn't interfere with the output of `awk`.
- `awk '{print $9}'`: prints the 9th field of the input, which corresponds to the name of the directory. This will be printed for each line that matches the `grep` filter.

In summary, the command `ls -l | grep "^d" | tee >(wc -l >&2) | awk '{print $9}'` lists the directories in the current directory by filtering the output of `ls -l` with `grep`, counting the number of directories found with `wc -l`, and printing the directory names with `awk`. The final output displays only the names of the directories, one per line.

About awk

AWK is an acronym for "Aho, Weinberger, and Kernighan", the last names of its authors. It's a programming language that is widely used for text processing and data manipulation tasks.

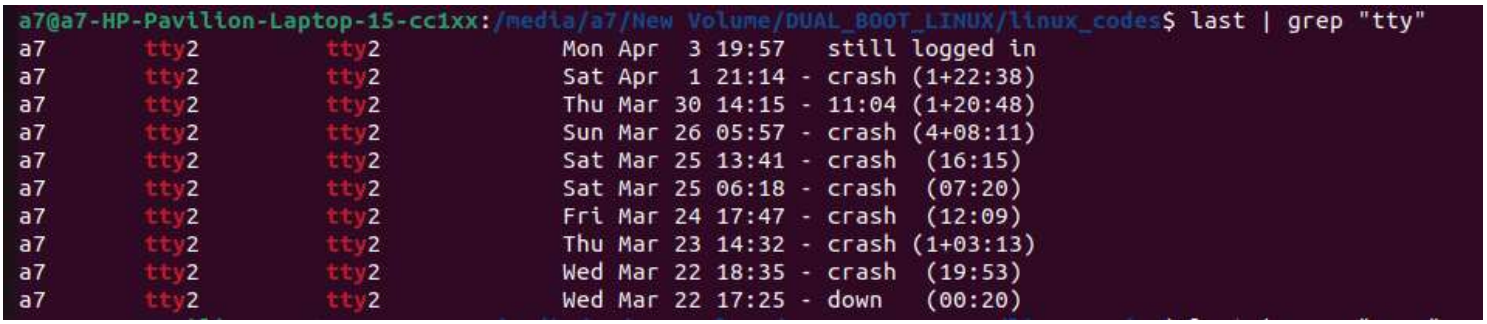
Awk is a versatile programming language used for processing text files, and it's often used for data manipulation tasks. It reads text line by line, and by default, it separates each line into fields based on whitespace (spaces, tabs, etc.).

In the given command, `awk '{print $9}'` is used to extract the ninth field from each line of input text. In the context of the `ls` command, the ninth field corresponds to the name of the directory (since we're using `ls -l` which includes detailed information about files and directories).

The **`$`** symbol is used to indicate the **field number** we want to extract. So `$9` means "**give me the ninth field of each line**"

In summary, the `awk '{print $9}'` part of the command is telling awk to extract the names of the directories from the output of the `ls` command.

3. Find out whether the users with a pattern“itp9” in their names have logged in ?



last | grep "tty" | grep -v “still logged out”

This command will display all the login records for users whose names contain "itp9". If there is any record, it means that the user has logged in at some point in time.

Note that the `last` command displays a list of all previously logged-in users and their login history. The `grep` command is used to filter the output and display only those records that match the specified pattern.
This is the output of the `last` command, which shows the login records of users.

The output has the following fields:

- User: the username of the logged in user, or "reboot" if the record is related to a system boot.
- TTY: the name of the terminal or device the user is logged in to.
- From: the hostname or IP address from where the user logged in, or "-" if the information is not available.
- Login Time: the date and time when the user logged in.
- Logout Time: the date and time when the user logged out, or "still logged in" or "still running" if the user is still logged in or the system is still running.
- Duration: the duration of the user's login session, expressed in days+hours:minutes.

The last line of the output shows the beginning of the wtmp log file, which contains the login records.

In this specific output, it shows that the system has been rebooted multiple times between March 22 and April 3, and that user "a7" has been logging in to tty2 on some of those instances. The output also shows the date and time when the user logged in and out, as well as the duration of each login session.

The `-v` option in `grep` stands for "invert match". It is used to select non-matching lines. In other words, it will exclude lines that match the given pattern. In the command `last | grep "itp9" | grep -v "still logged out"`, the second `grep` command with `-v` option is used to exclude lines that contain the phrase "still logged out".

4. Find out whether a particular user “itp9” has logged in ?

```
a7@a7-HP-Pavilion-Laptop-15-cc1xx:/media/a7/New Volume/DUAL_BOOT_LINUX/linux_codes$ last tty2
a7       tty2         tty2             Mon Apr  3 19:57   still logged in
a7       tty2         tty2             Sat Apr  1 21:14 - crash (1+22:38)
a7       tty2         tty2             Thu Mar 30 14:15 - 11:04 (1+20:48)
a7       tty2         tty2             Sun Mar 26 05:57 - crash (4+08:11)
a7       tty2         tty2             Sat Mar 25 13:41 - crash  (16:15)
a7       tty2         tty2             Sat Mar 25 06:18 - crash  (07:20)
a7       tty2         tty2             Fri Mar 24 17:47 - crash  (12:09)
a7       tty2         tty2             Thu Mar 23 14:32 - crash (1+03:13)
a7       tty2         tty2             Wed Mar 22 18:35 - crash  (19:53)
a7       tty2         tty2             Wed Mar 22 17:25 - down   (00:20)
```

last tty2

5. Assign a value “Black” to var1 and then display the following message on the terminal using this variable.

Black belt is associated with karate

var1="Black"

echo "${var1} belt is associated with karate"

var1="Black"

echo "$var1 belt is associated with karate"

6. Accept a file name and a number (x). Display x lines from the top of the file. Check if the file exists and is readable. The value of x should not exceed the total number of lines in the files. Display suitable messages in case an error is encountered.

—----------CODE—----------

```
#!/bin/bash

# check if the correct number of arguments are provided
if [ $# -ne 2 ]
then
    echo "Usage: $0 <filename> <number_of_lines>"
    exit 1
fi

# assign arguments to variables
filename=$1
num_lines=$2

# check if file exists and is readable
# if [ ! -f $filename ]; then               //the `;` is a command seperator and not any syntax
if [ ! -f $filename ]; then
    echo "Error: $filename does not exist"
    exit 1
elif [ ! -r $filename ]
then
    echo "Error: $filename is not readable"
    exit 1
fi

# check if num_lines is less than or equal to the total number of lines in the file
total_lines=$(wc -l < $filename)
if [ $num_lines -gt $total_lines ]
then
    echo "Error: $num_lines exceeds the total number of lines in $filename ($total_lines)"
    exit 1
fi

# display the first x lines of the file
head -$num_lines $filename
```

```bash
#!/bin/bash

# check if the correct number of arguments are provided
if [ $# -ne 2 ]
then
    echo "Usage: $0 <filename> <number_of_lines>"
    exit 1
fi

# assign arguments to variables
filename=$1
num_lines=$2

# check if file exists and is readable
# if [ ! -f $filename ]; then               //the `;` is a command seperator and not any syntax
if [ ! -f $filename ]; then
    echo "Error: $filename does not exist"
    exit 1
elif [ ! -r $filename ]
then
    echo "Error: $filename is not readable"
    exit 1
fi

# check if num_lines is less than or equal to the total number of lines in the file
total_lines=$(wc -l < $filename)
if [ $num_lines -gt $total_lines ]
then
    echo "Error: $num_lines exceeds the total number of lines in $filename ($total_lines)"
    exit 1
fi

# display the first x lines of the file
head -$num_lines $filename

```

—----------

7. Write a menu based script which displays the following options :
    1. Make a file.
    2. Display contents
    3. Copy the file
    4. Rename the file
    5. Delete the file
    6. Exit

    Enter your option:

    If the user selects option 1, accept a file name from the user. If the file exists, then display an error message pass the control to the menu. If the file does not exist, then allow the user to enter some data. Pressing ^D would save the contents and display the menu.

    If the user selects option 2, then accept a file name from the user. If the file exists, then display the contents of the file. If the file does not exist, then display suitable error message. After this process, display the menu to accept another option.

Selecting Option 3 allows the user to accept the source file and target file. If the source file exists and is readable, then accept the target file name. If the source file does not exist, then display suitable error message. If the target file does not exist, then copy the contents of the source file to the target file. If the target file exists, then display suitable message and go back to the menu.

Option 4 is similar to option 3 but rename the file instead of copying.

Selecting option 5 allows the user to enter a file name. If the file exists, then check to see if it is writable. If so, then delete the file with confirmation from the user. If the file does not exist, then display suitable error message.

8. Write a menu based shell script which will perform arithmetic operations on two numbers which are inputted by user. Menu should display following operations

Menu

1: Addition

2: Substraction

3: Multiplication

4: Division

5: Exit

```
#!/bin/bash

while true; do
  echo "Menu:"
  echo "1. Make a file"
  echo "2. Display contents"
  echo "3. Copy the file"
  echo "4. Rename the file"
  echo "5. Delete the file"
  echo "6. Exit"
  read -p "Enter your option: " option

  case $option in
    1)
      read -p "Enter file name: " filename
      if [ -e "$filename" ]; then
        echo "File already exists"
      else
        echo "Enter file contents (press Ctrl-D to save):"
        cat > "$filename"
      fi
      ;;
```

```
2)
  read -p "Enter file name: " filename
  if [ -e "$filename" ]; then
    cat "$filename"
  else
    echo "File does not exist"
  fi
  ;;
3)
  read -p "Enter source file name: " src
  if [ ! -e "$src" ]; then
    echo "Source file does not exist"
  elif [ ! -r "$src" ]; then
    echo "Source file is not readable"
  else
    read -p "Enter target file name: " target
    if [ -e "$target" ]; then
      echo "Target file already exists"
    else
      cp "$src" "$target"
      echo "File copied successfully"
    fi
  fi
  ;;
4)
  read -p "Enter current file name: " current
  if [ ! -e "$current" ]; then
    echo "File does not exist"
  else
    read -p "Enter new file name: " new
    if [ -e "$new" ]; then
      echo "New file name already exists"
    else
      mv "$current" "$new"
      echo "File renamed successfully"
    fi
  fi
  ;;
5)
  read -p "Enter file name: " filename
  if [ ! -e "$filename" ]; then
    echo "File does not exist"
  elif [ ! -w "$filename" ]; then
    echo "File is not writable"
  else
    read -p "Are you sure you want to delete $filename? (y/n) " choice
    if [ "$choice" = "y" ]; then
      rm "$filename"
      echo "File deleted successfully"
    fi
  fi
  ;;
6)
```

```
    exit 0
    ;;
  *)
    echo "Invalid option"
    ;;
esac

echo
done
```

```bash
#!/bin/bash

while true; do
  echo "Menu:"
  echo "1. Make a file"
  echo "2. Display contents"
  echo "3. Copy the file"
  echo "4. Rename the file"
  echo "5. Delete the file"
  echo "6. Exit"
  read -p "Enter your option: " option

  case $option in
    1)
      read -p "Enter file name: " filename
      if [ -e "$filename" ]; then
        echo "File already exists"
      else
        echo "Enter file contents (press Ctrl-D to save):"
        cat > "$filename"
      fi
      ;;
    2)
      read -p "Enter file name: " filename
      if [ -e "$filename" ]; then
        cat "$filename"
      else
        echo "File does not exist"
      fi
      ;;
    3)
      read -p "Enter source file name: " src
      if [ ! -e "$src" ]; then
        echo "Source file does not exist"
      elif [ ! -r "$src" ]; then
        echo "Source file is not readable"
      else
        read -p "Enter target file name: " target
        if [ -e "$target" ]; then
          echo "Target file already exists"
        else
          cp "$src" "$target"
          echo "File copied successfully"
        fi
      fi
```

```
41          else
42            cp "$src" "$target"
43            echo "File copied successfully"
44          fi
45        fi
46        ;;
47      4)
48        read -p "Enter current file name: " current
49        if [ ! -e "$current" ]; then
50          echo "File does not exist"
51        else
52          read -p "Enter new file name: " new
53          if [ -e "$new" ]; then
54            echo "New file name already exists"
55          else
56            mv "$current" "$new"
57            echo "File renamed successfully"
58          fi
59        fi
60        ;;
61      5)
62        read -p "Enter file name: " filename
63        if [ ! -e "$filename" ]; then
64          echo "File does not exist"
65        elif [ ! -w "$filename" ]; then
66          echo "File is not writable"
67        else
68          read -p "Are you sure you want to delete $filename? (y/n) " choice
69          if [ "$choice" = "y" ]; then
70            rm "$filename"
71            echo "File deleted successfully"
72          fi
73        fi
74        ;;
75      6)
76        exit 0
77        ;;
78      *)
79        echo "Invalid option"
80        ;;
81    esac
82
83    echo
84  done
85
```

9. Write a shell script that will remove a file taken as command line argument after taking the proper backup of file in /home/user1/backup directory

——----------CODE—----------

#!/bin/bash

```
if [ $# -ne 1 ]; then
  echo "Usage: $0 <filename>"
  exit 1
fi

filename=$1
backupdir=/media/a7/New Volume/DUAL_BOOT_LINUX/linux_codes


if [ ! -f "$filename" ]; then
  echo "Error: $filename does not exist or is not a regular file."
  exit 1
fi

if [ ! -d "$backupdir" ]; then
  echo "Creating backup directory: $backupdir"
  mkdir -p "$backupdir"
fi

backupfile="$backupdir/$(basename "$filename").$(date +%Y%m%d%H%M%S)"
cp -p "$filename" "$backupfile"
echo "Backup file created: $backupfile"

rm -i "$filename"
echo "File $filename deleted."
```

—----------

```
#!/bin/bash

# Check if the user has entered a file name as an argument.
if [ $# -ne 1 ]; then
  echo "Usage: $0 filename"
  exit 1
fi

# Set the filename and backup directory
filename="$1"
backupdir="/media/a7/New Volume/DUAL_BOOT_LINUX/linux_codes"

# Check if the file exists and is readable
if [ ! -f "$filename" ]; then
  echo "Error: File $filename not found."
  exit 1
elif [ ! -r "$filename" ]; then
  echo "Error: File $filename is not readable."
  exit 1
fi

# Create the backup directory if it does not exist
if [ ! -d "$backupdir" ]; then
  mkdir "$backupdir"
fi
```

```
# Get the current date and time to add to the backup filename
datetime=$(date '+%Y-%m-%d_%H-%M-%S')
backupfile="$backupdir/${filename}_${datetime}"

# Copy the file to the backup directory with the datetime appended
cp "$filename" "$backupfile"

# Check if the copy was successful
if [ $? -eq 0 ]; then
  echo "Backup of $filename successful."
else
  echo "Error: Backup of $filename failed."
  exit 1
fi

# Remove the original file
rm "$filename"

# Check if the removal was successful
if [ $? -eq 0 ]; then
  echo "File $filename successfully removed."
else
  echo "Error: Removal of $filename failed."
  exit 1
fi
```

—----------

```bash
#!/bin/bash

# Check if the user has entered a file name as an argument.
if [ $# -ne 1 ]; then
  echo "Usage: $0 filename"
  exit 1
fi

# Set the filename and backup directory
filename="$1"
backupdir="/media/a7/New Volume/DUAL_BOOT_LINUX/linux_codes"

# Check if the file exists and is readable
if [ ! -f "$filename" ]; then
  echo "Error: File $filename not found."
  exit 1
elif [ ! -r "$filename" ]; then
  echo "Error: File $filename is not readable."
  exit 1
fi

# Create the backup directory if it does not exist
if [ ! -d "$backupdir" ]; then
  mkdir "$backupdir"
fi

# Get the current date and time to add to the backup filename
datetime=$(date '+%Y-%m-%d_%H-%M-%S')
backupfile="$backupdir/${filename}_${datetime}"

# Copy the file to the backup directory with the datetime appended
cp "$filename" "$backupfile"

# Check if the copy was successful
if [ $? -eq 0 ]; then
  echo "Backup of $filename successful."
else
  echo "Error: Backup of $filename failed."
  exit 1
fi

# Remove the original file
rm "$filename"

# Check if the removal was successful
if [ $? -eq 0 ]; then
  echo "File $filename successfully removed."
else
  echo "Error: Removal of $filename failed."
  exit 1
fi
```

```bash
#!/bin/bash

#
if [ $# -ne 1 ]; then
  echo "Usage: $0 <filename>"
  exit 1
fi

filename=$1
backupdir=/media/a7/New Volume/DUAL_BOOT_LINUX/linux_codes

if [ ! -f "$filename" ]; then
  echo "Error: $filename does not exist or is not a regular file."
  exit 1
fi

if [ ! -d "$backupdir" ]; then
  echo "Creating backup directory: $backupdir"
  mkdir -p "$backupdir"
fi

backupfile="$backupdir/$(basename "$filename").$(date +%Y%m%d%H%M%S)"
cp -p "$filename" "$backupfile"
echo "Backup file created: $backupfile"

rm -i "$filename"
echo "File $filename deleted."
```

1. The script expects one argument, which should be the name of the file to be deleted.
2. It checks that exactly one argument has been provided. If not, it displays a usage message and exits with an error code.
3. It assigns the filename to a variable and sets the backup directory path.
4. It checks that the file exists and is a regular file. If not, it displays an error message and exits with an error code.
5. It checks if the backup directory exists. If not, it creates it using the `mkdir` command with the `-p` option to create intermediate directories as necessary.
6. It creates a backup file name by appending the current date and time to the original file name. The `basename` command is used to strip the directory path from the file name. The `cp` command is used to create a copy of the original file with the same ownership, permissions, and timestamp as the original file. The `-p` option is used to preserve the file attributes.
7. It displays a message confirming that the backup file has been created.
8. It prompts the user for confirmation before deleting the original file using the `rm -i` command.
9. It displays a message confirming that the file has been deleted.

10. Write a shell script that will accept a string from the user. Copy all contents of the file to other file without that string. Also display number of characters, lines, and words.

```
#!/bin/bash

# Accept a string from the user
echo "Enter the string to exclude from the file:"
read exclude_string

# Accept the source and target file names
echo "Enter the source file name:"
read src_file
echo "Enter the target file name:"
read tgt_file

# Check if source file exists and is readable
if [ ! -r "$src_file" ]; then
  echo "Error: $src_file does not exist or is not readable"
  exit 1
fi

# Create a temporary file for filtering
tmp_file=$(mktemp)
```

```
# Copy all lines that do not contain the exclude string to the temporary file
grep -v "$exclude_string" "$src_file" > "$tmp_file"

# Count the number of characters, lines, and words in the source file
char_count=$(wc -c < "$src_file")
line_count=$(wc -l < "$src_file")
word_count=$(wc -w < "$src_file")

# Move the temporary file to the target file
mv "$tmp_file" "$tgt_file"

# Display the counts
echo "Character count: $char_count"
echo "Line count: $line_count"
echo "Word count: $word_count"
```

Here's an explanation of each line:

1. `#!/bin/bash`: Shebang line that specifies the shell to use for interpreting the script as bash.
2. `echo "Enter the string to exclude from the file:"`: Prompts the user to enter the string to exclude from the file.
3. `read exclude_string`: Reads the user's input and assigns it to the `exclude_string` variable.
4. `echo "Enter the source file name:"`: Prompts the user to enter the source file name.
5. `read src_file`: Reads the user's input and assigns it to the `src_file` variable.
6. `echo "Enter the target file name:"`: Prompts the user to enter the target file name.
7. `read tgt_file`: Reads the user's input and assigns it to the `tgt_file` variable.
8. `if [ ! -r "$src_file" ]; then`: Checks if the source file exists and is readable.
9. `echo "Error: $src_file does not exist or is not readable"`: Displays an error message if the source file does not exist or is not readable.
10. `exit 1`: Exits the script with a non-zero exit code indicating an error.
11. `tmp_file=$(mktemp)`: Creates a temporary file and assigns its path to the `tmp_file` variable.
12. `grep -v "$exclude_string" "$src_file" > "$tmp_file"`: Uses `grep` to copy all lines that do not contain the exclude string to the temporary file.
13. `char_count=$(wc -c < "$src_file")`: Counts the number of characters in the source file and assigns it to the `char_count` variable.
14. `line_count=$(wc -l < "$src_file")`: Counts the number of lines in the source file and assigns it to the `line_count` variable.
15. `word_count=$(wc -w < "$src_file")`: Counts the number of words in the source file and assigns it to the `word_count` variable.
16. `mv "$tmp_file" "$tgt_file"`: Moves the temporary file to the target file.
17. `echo "Character count: $char_count"`: Displays the character count.
18. `echo "Line count: $line_count"`: Displays the line count.
19. `echo "Word count: $word_count"`: Displays the word count.

```
a7@a7-HP-Pavilion-Laptop-15-cc1xx:/media/a7/New Volume/DUAL_BOOT_LINUX/linux_codes$ cat > UNIX.txt
Unix is unix all else is fenix
shita7@a7-HP-Pavilion-Laptop-15-cc1xx:/media/a7/New Volume/DUAL_BOOT_LINUX/linux_codes$ cat UNIX.txt
Unix is unix all else is fenix
shita7@a7-HP-Pavilion-Laptop-15-cc1xx:/media/a7/New Volume/DUAL_BOOT_LINUX/linux_codes$ ./ExceptString.sh
Enter the string to exclude from the file:
unix
Enter the source file name:
UNIX.txt
Enter the target file name:
newUnix.txt
Character count: 35
Line count: 1
Word count: 8
a7@a7-HP-Pavilion-Laptop-15-cc1xx:/media/a7/New Volume/DUAL_BOOT_LINUX/linux_codes$ cat newUnix.txt
shit
```

This is deleting the whole line as opposed to just the word : find a better solution

```
#!/bin/bash

# Accept a string from the user
echo "Enter a string:"
read string

# Accept a filename from the user
echo "Enter a filename:"
read filename

# Check if the file exists and is readable
if [ ! -r "$filename" ]; then
  echo "Error: $filename does not exist or is not readable."
  exit 1
fi

# Create backup directory if it does not exist

# issues with space or `\` between New Volume   => `New Volume` is working at the moment
if [ ! -d "/media/a7/New Volume/DUAL_BOOT_LINUX/shells_script/backup" ]; then
  mkdir /media/a7/New Volume/DUAL_BOOT_LINUX/shells_script/backup
fi

# Create backup of the file in the backup directory
```

```
backup_file="/media/a7/New Volume/DUAL_BOOT_LINUX/shells_script/backup/$filename-$(date +%Y%m%d%H%M%S)"

cp "$filename" "$backup_file"

echo "Backup of $filename created at $backup_file"


# Copy contents of file to new file without string

new_filename="new_$filename"

grep -v "$string" "$filename" > "$new_filename"


# Display number of characters, lines, and words in the original file

char_count=$(wc -m < "$filename")

line_count=$(wc -l < "$filename")

word_count=$(wc -w < "$filename")

echo "Original file has $char_count characters, $line_count lines, and $word_count words."


# Remove the original file and rename the new file

rm "$filename"

mv "$new_filename" "$filename"

echo "File $filename has been updated without the string: $string"
```

```bash
#!/bin/bash

# Accept a string from the user
echo "Enter a string:"
read string

# Accept a filename from the user
echo "Enter a filename:"
read filename

# Check if the file exists and is readable
if [ ! -r "$filename" ]; then
  echo "Error: $filename does not exist or is not readable."
  exit 1
fi

# Create backup directory if it does not exist

# issues with space or `\` between New Volume   => `New Volume` is working at the moment
if [ ! -d "/media/a7/New Volume/DUAL_BOOT_LINUX/shells_script/backup" ]; then
  mkdir /media/a7/New Volume/DUAL_BOOT_LINUX/shells_script/backup
fi

# Create backup of the file in the backup directory
backup_file="/media/a7/New Volume/DUAL_BOOT_LINUX/shells_script/backup/$filename-$(date +%Y%m%d%H%M%S)"
cp "$filename" "$backup_file"
echo "Backup of $filename created at $backup_file"

# Copy contents of file to new file without string
new_filename="new_$filename"
grep -v "$string" "$filename" > "$new_filename"

# Display number of characters, lines, and words in the original file
char_count=$(wc -m < "$filename")
line_count=$(wc -l < "$filename")
word_count=$(wc -w < "$filename")
echo "Original file has $char_count characters, $line_count lines, and $word_count words."

# Remove the original file and rename the new file
rm "$filename"
mv "$new_filename" "$filename"
echo "File $filename has been updated without the string: $string"
```

The perfect code : which only removes just the input string and not the whole line

```
#!/bin/bash

# Accept a string from the user
echo "Enter a string:"
read string

# Accept a filename from the user
echo "Enter a filename:"
read filename

# Check if the file exists and is readable
if [ ! -r "$filename" ]; then
  echo "Error: $filename does not exist or is not readable."
  exit 1
fi

# Create backup directory if it does not exist
if [ ! -d "/media/a7/New Volume/DUAL_BOOT_LINUX/shells_script/backup/js" ]; then
  mkdir /media/a7/New Volume/DUAL_BOOT_LINUX/shells_script/backup/js
fi

# Create backup of the file in the backup directory
```

```
backup_file="/media/a7/New Volume/DUAL_BOOT_LINUX/shells_script/backup/js/$filename-$(date +%Y%m%d%H%M%S)"

cp "$filename" "$backup_file"

echo "Backup of $filename created at $backup_file"


# Copy contents of file to new file without string

new_filename="new_$filename"

while read -r line; do

    echo "${line/$string}" >> "$new_filename"

done < "$filename"


# Display number of characters, lines, and words in the original file

char_count=$(wc -m < "$filename")

line_count=$(wc -l < "$filename")

word_count=$(wc -w < "$filename")

echo "Original file has $char_count characters, $line_count lines, and $word_count words."


# Remove the original file and rename the new file

rm "$filename"

mv "$new_filename" "$filename"

echo "File $filename has been updated without the word/string: $string"
```

```bash
#!/bin/bash

# Accept a string from the user
echo "Enter a string:"
read string

# Accept a filename from the user
echo "Enter a filename:"
read filename

# Check if the file exists and is readable
if [ ! -r "$filename" ]; then
  echo "Error: $filename does not exist or is not readable."
  exit 1
fi

# Create backup directory if it does not exist
if [ ! -d "/media/a7/New Volume/DUAL_BOOT_LINUX/shells_script/backup/js" ]; then
  mkdir /media/a7/New Volume/DUAL_BOOT_LINUX/shells_script/backup/js
fi

# Create backup of the file in the backup directory
backup_file="/media/a7/New Volume/DUAL_BOOT_LINUX/shells_script/backup/js/$filename-$(date +%Y%m%d%H%M%S)"
cp "$filename" "$backup_file"
echo "Backup of $filename created at $backup_file"

# Copy contents of file to new file without string
new_filename="new_$filename"
while read -r line; do
    echo "${line/$string}" >> "$new_filename"
done < "$filename"

# Display number of characters, lines, and words in the original file
char_count=$(wc -m < "$filename")
line_count=$(wc -l < "$filename")
word_count=$(wc -w < "$filename")
echo "Original file has $char_count characters, $line_count lines, and $word_count words."

# Remove the original file and rename the new file
rm "$filename"
mv "$new_filename" "$filename"
echo "File $filename has been updated without the word/string: $string"
```

correct

```
# Copy contents of file to new file without string
new_filename="new_$filename"
while read -r line; do
    echo "${line/$string}" >> "$new_filename"
done < "$filename"
```

Incorrect

```
# Copy contents of file to new file without string
new_filename="new_$filename"
grep -v "$string" "$filename" > "$new_filename"
```

bash Copy code

```bash
# Copy contents of file to new file without string
new_filename="new_$filename"
while read -r line; do
    echo "${line/$string}" >> "$new_filename"
done < "$filename"
```

- `new_filename="new_$filename"`: This creates a new filename with "new_" added to the start of the original filename.
- `while read -r line; do`: This starts a loop that reads each line of the input file one at a time.
- `echo "${line/$string}" >> "$new_filename"`: This echoes the current line to the new file, but with the string removed from it. `${line/$string}` is a parameter expansion that removes the first occurrence of `$string` from `$line`.
- `done < "$filename"`: This ends the loop and specifies that the input for the loop should be taken from `$filename`.

Incorrect code:

perl Copy code

```perl
# Copy contents of file to new file without string
new_filename="new_$filename"
grep -v "$string" "$filename" > "$new_filename"
```

- `new_filename="new_$filename"`: This creates a new filename with "new_" added to the start of the original filename.
- `grep -v "$string" "$filename" > "$new_filename"`: This uses the `grep` command to filter out any lines that contain `$string` from `$filename`, and then redirects the output to `$new_filename`.

The corrected code loops through each line of the input file and removes the specified string from the line before writing it to the output file. The incorrect code simply uses `grep` to filter out any lines that contain the specified string from the input file.

```
while read -r line; do
```

The `-r` option used with the `read` command disables the interpretation of backslash escapes. This option is useful when reading text files that may contain backslash characters. If the `-r` option is not used, the backslash character is interpreted as an escape character and may affect the behavior of the `read` command. By default, the `read` command treats backslashes as escape characters, which can lead to unexpected results when reading text that contains backslashes. Therefore, it's a good practice to use the `-r` option with the `read` command unless you need to interpret backslash escapes.

11. Write a shell script which will generate the O/P as follows

    *

    * * * * * * * * *

    *

12. Write a shell script which will calculate the factorial of an integer entered from the keyboard.

```
#!/bin/bash

# Accept the number from the user
echo "Enter a number:"
read num

# Initialize the factorial to 1
factorial=1

# Loop from 1 to the entered number and multiply each number with the factorial
for (( i=1; i<=num; i++ ))
do
  factorial=$((factorial * i))
done

# Print the factorial
echo "Factorial of $num is: $factorial"
```

13. Write a shell script which will generate the O/P as follows

    *

```
**

***

****
```

```
#!/bin/bash

# Loop through each row
for ((i=1; i<=4; i++))
do
  # Loop through each column
  for ((j=1; j<=i; j++))
  do
    echo -n "*"
  done

  # Move to next row
  echo ""
done
```

14. Write a script which when executed checks out whether it is a working day or not?

(Note: Working day Mon-Fri)

```
#!/bin/bash

# Get current day of the week
day=$(date +%u)

# Check if it is a weekday (1 to 5)
if [ "$day" -ge 1 ] && [ "$day" -le 5 ]; then
    echo "Today is a working day"
else
    echo "Today is not a working day"
fi
```

```bash
#!/bin/bash

day=$(date +%a)

# read -p "enter 1 to 7: " day

if [ $day='Sun' || $day='Sat' ]
# if [[ $day -eq 6 || $day -eq 7 ]]
then
    echo "$day is a weekend"
else
    echo "$day is not a weekend"

fi
```

Assignment – 24-03-23

Q.1 Keep accepting an input message from the user and displaying it till the user does not type 'Bye' (not case sensitive) using a Shell script.

bash Copy code

```bash
#!/bin/bash

read -p "Enter message, I will run until you type bye: " msg

# Use command substitution to transform the input to lowercase
msg=$(echo "$msg" | tr '[:upper:]' '[:lower:]')

# Add quotes around variables to prevent word splitting and globbing
while [ "$msg" != "bye" ]
do
    echo "Enter a message, I will run until you type bye"
    read -p "Enter message: " msg

    # Use the double brackets for string comparison
    if [[ "$msg" == "bye" ]]; then
        exit
    fi
done
```

```
while [ "$msg" != 'bye' ]
```

In shell scripting, single quotes and double quotes have different meanings.

Single quotes (`' '`) are used to denote a **string literal**, which means that the **text inside the quotes** is taken **literally**, and **special characters** (such as **variables** and **escape sequences**) are **not interpreted**. This means that if you put the string 'bye' inside single quotes, it will be interpreted literally as the string "bye".

Double quotes (`" "`) are used to allow **variable substitution and interpret escape sequences**. This means that if you put a **variable** inside **double quotes**, its **value** will be **substituted into the string**. In this case, **`$msg`** is a variable that holds the user input, and we want its value to be substituted into the string comparison expression.

So in the code `while [ "$msg" != 'bye' ]`, the variable `$msg` is in double quotes so that its value is substituted into the string comparison expression, while the string 'bye' is in single quotes to indicate that it should be taken literally

why use echo in this command : **msg=$(echo "$msg" | tr '[:upper:]' '[:lower:]')**

The `echo` command is used here to pass the value of the `msg` variable to the `tr` command. The `tr` command is used to convert any uppercase letters in the `msg` variable to lowercase.

Without using `echo`, the `tr` command would not receive the value of the `msg` variable, and instead, would interpret it as a string literal "msg".

So, `echo` is used to print the value of the `msg` variable to standard output, which is then piped to the `tr` command to perform the lowercase conversion. The result of this command is then assigned to the `msg` variable.

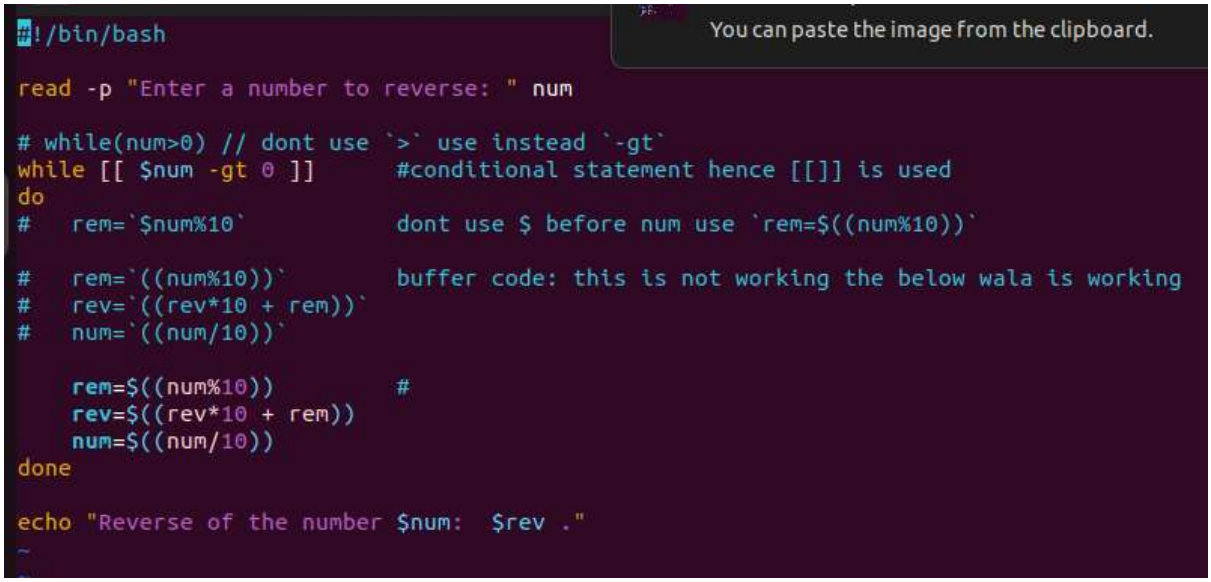
```
#!/bin/bash

read -p "Enter message, i will run till you type bye: " msg

msg=`echo $msg | tr [A-Z] [a-z]`

while [ "$msg" != 'bye' ]
do
```

```
#!/bin/bash

read -p "Enter message, i will run till you type bye: " msg

msg=`echo $msg | tr [A-Z] [a-z]`

while [ "$msg" != 'bye' ]
do

    read -p "Enter msg, i'll run till you type bye: " msg
    msg=$(echo $msg | tr [A-Z] [a-z])


    if [ "$msg" == 'bye' ]
        then
        echo "Exitting.... printing \$"
            exit
    fi

done
```


```
read -p "Enter msg, i'll run till you type bye: " msg
msg=$(echo $msg | tr [A-Z] [a-z])
if [ "$msg" == 'bye' ]
    then
    echo "Exitting.... printing \$"
        exit
fi

done
```


another way: msg=$(echo "$msg" | tr '[:upper:]' '[:lower:]')

Q.2 Write a shell script to reverse an integer (e.g. 1234 to 4321).




```
#!/bin/bash

read -p "Enter a number to reverse: " num
bkp=$num

# while(num>0) // dont use `>` use instead `-gt`
while [[ $num -gt 0 ]]      #conditional statement hence [[ ]] is used
do
#   rem=`$num%10`           dont use $ before num use `rem=$((num%10))`

#   rem=`((num%10))`        buffer code: this is not working the below wala is working
#   rev=`((rev*10 + rem))`
#   num=`((num/10))`

    rem=$((num%10))         #
    rev=$((rev*10 + rem))
    num=$((num/10))
done

echo "Reverse of the number $bkp:  $rev ."
```

Yes, there are a couple of errors in the script:

- Line 9: You should use `rem=$((num%10))` instead of `rem=((num%10))`` to perform arithmetic expansion and assign the result to `rem`.
- Line 10: Similarly, you should use `rev=$((rem*10 + rev))` instead of `rev=((rem*10 + rev))`` to perform arithmetic expansion and assign the result to `rev`.
- Line 11: You should use `num=$((num/10))` instead of `num=((num/10))`` to perform arithmetic expansion and assign the result to `num`.
- Line 14: You should enclose the variable `$num` and `$rev` in double quotes to avoid word splitting and globbing issues. Also, the value of `num` at this point would be 0, so you should reverse the order of the variables.

Refer for more: https://chat.openai.com/chat/24b7b25c-2b78-4259-b8d9-742acfccc3b9

Q.3 Create a looping menu to accept either (1) Simple interest or (2) Compound interest or (3) Exit as the option from the user. If the user selects Simple interest, accept principal amount (p), time (in years) (n) and rate of interest (r). Display simple interest using the formula interest = pnr / 100. If the user selected compound interest, also accept how many times interest was applied/calculated every year (t). Display compound interest using the formula interest = p x (1 + r/n) raised to the power (n x t)

```
#!/bin/bash

while true; do
  echo "Please select an option:"
  echo "1. Simple interest"
  echo "2. Compound interest"
  echo "3. Exit"
  read -p "Enter your choice: " choice

  case $choice in
    1)
      read -p "Enter principal amount: " p
      read -p "Enter time in years: " n
      read -p "Enter rate of interest: " r
      interest=$(echo "scale=2; $p * $n * $r / 100" | bc)
      echo "Simple interest is: $interest"
      ;;
    2)
      read -p "Enter principal amount: " p
      read -p "Enter time in years: " n
      read -p "Enter rate of interest: " r
      read -p "Enter number of times interest is calculated per year: " t
      interest=$(echo "scale=2; $p * (1 + $r / $n)^($n * $t)" | bc)
      echo "Compound interest is: $interest"
      ;;
    3)
      echo "Exiting..."
      exit 0
      ;;
    *)
      echo "Invalid choice. Please try again."
      ;;
  esac
done
```

#!/bin/bash

while true: do
  echo "Please select an option:"
  echo "1. Simple interest"
  echo "2. Compound interest"
  echo "3. Exit"
  read -p "Enter your choice: " choice

```
  case $choice in
    1)
      read -p "Enter principal amount: " p
      read -p "Enter time in years: " n
      read -p "Enter rate of interest: " r
      interest=$(echo "scale=2; $p * $n * $r / 100" | bc)
      echo "Simple interest is: $interest"
      ;;
    2)
      read -p "Enter principal amount: " p
      read -p "Enter time in years: " n
      read -p "Enter rate of interest: " r
      read -p "Enter number of times interest is calculated per year: " t
      interest=$(echo "scale=2; $p * (1 + $r / $n)^($n * $t)" | bc)
      echo "Compound interest is: $interest"
      ;;
    3)
      echo "Exiting..."
      exit 0
      ;;
    *)
      echo "Invalid choice. Please try again."
      ;;
  esac
done
```

Sudo apt install tree -y

Tree -L 1

/bin +