Topics To be Covered

Week	Broader Topic	Lecture	Topics	Tools to be covered
1	Filters / Text Processing Commands	36-40	1. Filters/ Text Processing Commands 1.1 cut command 1.2 awk command 1.2.a features of awk command 2. Grep command 3. egrep command 4. Difference between grep and egrep 5. sort command 6. uniq command 7. wc command 8. diff command 9. cmp command 10. compress and uncompress 10.1 tar 10.2 gzip 10.3 gunzip 11. Difference between tar, gzip and gunzip Practice Questions	Linux

1. Filters / Text Processing Commands: cut, awk, sort/

Filters or text processing commands are a set of powerful tools available on Linux and Unix-like systems that allow you to perform various operations on text files. They are designed to process textual data and provide functionality such as searching, filtering, transforming, sorting, and manipulating text.

1. 1 **The cut command** in Linux is used to extract specific columns or fields from a file or input stream. It allows you to select portions of lines based on delimiter characters or byte positions. The extracted columns can be output to the console or redirected to another file for further processing.

Here is the basic syntax of the cut command:

cut OPTIONS FILE

Some common options used with the cut command are:

- -f, --fields: Specifies the fields or columns to extract.
- Example: cut -f 1,3 filename will extract the first and third fields from each line of the file.

```
$ cut -d: -f '1 2' list-of-smartphones-2011.txt
Model:Company
IPhone4:Apple
Galaxy:Samsung
Optimus:LG
Sensation:HTC
IPhone4S:Apple
N9:Nokia
```

- -d, --delimiter: Specifies the delimiter character that separates the fields.

- Example: cut -d "," -f 2 filename will extract the second field using comma as the delimiter.

```
$ cut -d: -f1 list-of-smartphones-2011.txt
Model
IPhone4
Galaxy
Optimus
Sensation
IPhone4S
N9
```

- -s, --only-delimited: Skips lines that do not contain the delimiter character specified.
- Example: cut -d "," -f 2 --only-delimited filename will skip lines without a delimiter.
- -c, --characters: Specifies the character positions to extract.
- Example: cut -c 1-5 filename will extract the first five characters of each line.

```
$ cut -c 1-9 list-of-smartphones-2011.txt
Model:Com
IPhone4:A
Galaxy:Sa
Optimus:L
Sensation
IPhone4S:
N9:Nokia:
```

You can also combine multiple options to perform more complex operations, such as extracting specific fields using a different delimiter or extracting characters based on a range of positions.

Additionally, the cut command can read input from a pipe (stdin) if no file is specified. For example, you can use echo to provide input and pipe it to cut for processing.

Keep in mind that the cut command operates on fixed-width fields or fields separated by a specific delimiter. If you're working with more complex structures, such as structured data files, CSV files, or tab-separated values, other tools like awk or sed may be more suitable for processing.

For more details and advanced usage options, refer to the cut manual page (man cut) or consult the online documentation.

1. 2. **The awk command** is a versatile text processing tool that allows you to perform complex operations on text files, including data extraction, manipulation, and reporting. It operates on a per-line basis and processes input based on user-defined patterns and actions.

Here is the basic syntax of the awk command:

awk 'pattern { action }' input_file

Some key aspects of the awk command are:

- pattern: Specifies a pattern or condition that determines which lines to process. If omitted, the action block is applied to all lines.
- action: Defines the action or set of commands to be executed when the pattern is matched. It can include various operations, such as printing, calculations, conditionals, loops, and more.

1.2 a Here are a few common uses and features of the awk command:

1. **Field Extraction and Manipulation:**
    - **Accessing Fields:** awk treats each line as a set of fields, separated by a specified delimiter (by default, whitespace). You can access and manipulate these fields using the $ symbol followed by the field number. For example, $1 refers to the first field, $2 to the second field, and so on.
    - **Field Separators:** You can define a custom field separator using the -F option. For example, -F ',' sets the delimiter as a comma.
2. **Printing:**
    - **Print Statements:** You can use the print statement to output text or field values. For example, print $1 will print the first field of each line.

- Formatting Output: awk supports formatting output using printf statements. For example, printf "Name: %s, Age: %d\n", $1, $2 will print formatted output.

```
root@nginx:~# awk ' {print $0}' file.txt
Item    Model   Country         Cost

1       BMW     Germany         $25000
2       Volvo   Sweden          $15000
3       Subaru  Japan           $2500
4       Ferrari Italy           $2000000
5       SAAB    USA             $3000
```

3. **Conditionals and Loops:**

- **If-Else Statements:** You can use if and else statements to perform conditional operations based on certain criteria.
- **Loops:** awk supports while, for, and do-while loops to iterate over data or perform repetitive tasks.

```
awk '{

if (! ($3 ~ /[0-9]+$/))
  {
    print "Age is just a number but you do not have a number"
  }
else if ($3<20)
  {
    print "Student "$2,"of department", $4, "is less than 20 years old"
  }
else
  {
    print "Student "$2,"of department", $4, "is more than 20 years old"
  }

}' students.txt
```

4. **Built-in Variables and Functions:**
    - NR: Represents the current record (line) number.
    - NF: Represents the number of fields in the current line.
    - length(): Calculates the length of a string.
    - tolower(), toupper(): Converts a string to lowercase or uppercase, respectively.
    - Many more built-in variables and functions are available for text manipulation.

The awk command offers a wide range of capabilities for text processing and data extraction. It is particularly useful when working with structured data or log files. For

```
$ grep -r "linux" *
```

advanced usage and more complex scenarios, refer to the awk manual page (man awk) or explore online resources and tutorials.

2. Grep command

The grep command in Linux is used to search for text patterns within files or input streams. It allows you to find lines that match a specified pattern and display them as output. grep stands for "Global Regular Expression Print."

Here is the basic syntax of the grep command:

grep OPTIONS PATTERN FILE

Some common options used with the grep command are:

- **-i, --ignore-case**: Ignores case distinctions, so the search is case-insensitive.
    - Example: grep -i "pattern" filename

- -v, --invert-match: Selects non-matching lines. Displays lines that do not contain the specified pattern.
    - Example: grep -v "pattern" filename

- -r, --recursive: Recursively searches for the pattern in all files and directories within a directory.
    - Example: grep -r "pattern" directory

```
$ grep -r "linux" *
```

- -l, --files-with-matches: Prints only the names of files that contain the matching pattern.
    - Example: grep -l "pattern" filename
- -n, --line-number: Displays line numbers along with the matching lines.
    - Example: grep -n "pattern" filename

The grep command can also take input from a pipe (stdin) if no file is specified. For example, you can use the output of another command as input to grep.

Additionally, grep supports regular expressions, allowing you to use powerful pattern-matching capabilities. You can specify simple patterns or more complex expressions to match specific text patterns.

Here are a few examples of using grep:

- grep "word" filename: Searches for lines containing the word "word" in the specified file.
- grep -i "pattern" filename: Searches for lines containing the case-insensitive pattern match.
- grep -r "pattern" directory: Recursively searches for the pattern within all files in the specified directory.
- grep "pattern" file1 file2: Searches for the pattern in multiple files.

The grep command provides a flexible and efficient way to search for specific text patterns in files or streams. For more details and advanced usage options, refer to the grep manual page (man grep) or consult the online documentation.

3. egrep Command

The egrep command is a variant of the grep command, used for pattern matching and searching text files. It is available in Unix-like operating systems, including Linux.

The egrep command allows you to search for text using regular expressions, which are powerful patterns that can match complex string patterns. It is essentially an extended version of the grep command, supporting a wider range of regular expressions.

The basic syntax of the egrep command is as follows:

egrep [options] pattern [file(s)]

Here's a breakdown of the components:

- egrep: The command itself.

- [options]: Optional flags that modify the behaviour of the command. Some common options include:
  - -i: Ignore case (perform a case-insensitive search).
  - -v: Invert the match, i.e., show lines that do not match the pattern.
  - -r: Recursively search files in directories.
- pattern: The regular expression pattern you want to search for.
- [file(s)]: Optional argument specifying the file(s) to search. If not provided, egrep reads from standard input.

Here are a few examples to illustrate the usage of egrep:

```
bash-3.2$ cat first
Hello
bash-3.2$
bash-3.2$
bash-3.2$
bash-3.2$
bash-3.2$
bash-3.2$
bash-3.2$ egrep "hello" first
bash-3.2$
bash-3.2$ egrep "Hello" first
Hello
bash-3.2$
bash-3.2$
```

1. Search for the word "hello" in a file named file.txt:

```
egrep "hello" file.txt
```

2. Perform a case-insensitive search for "apple" in multiple files:

```
egrep -i "apple" file1.txt file2.txt
```

3. Search for lines that start with "Error" in a log file:

```
egrep "^Error" logfile.txt
```

4. Invert the match and search for lines not containing "success":

```
egrep -v "success" file.txt
```

These are just a few examples of the egrep command's usage. There are many more options and features available, which you can explore by referring to the command's documentation (man egrep).

4. Difference between grep and egrep

The main difference between grep and egrep lies in the type of regular expressions they support.

1. **Basic Regular Expressions (BRE):**

   grep uses Basic Regular Expressions by default. In BRE, certain characters have special meaning, such as ?, +, {}, |, and (). To match these special characters literally, you need to escape them with a backslash (\).

2. **Extended Regular Expressions (ERE):**

   `egrep`uses Extended Regular Expressions by default. ERE supports a wider range of special characters and metacharacters without the need for escaping. For example, ?, +, {}, |, and () have their special meaning directly without needing a backslash.

In summary, the main difference between grep and egrep is that egrep supports a more extensive set of regular expressions, making it easier to write complex patterns without the need for escaping certain characters.

However, it's worth noting that modern versions of grep (such as GNU grep) often include the -E option to enable Extended Regular Expressions, which makes grep functionally equivalent to egrep. This means you can achieve the same results using either grep -E or egrep. The egrep command is still useful for compatibility with older systems or scripts that rely on it specifically.

5. Sort Command

3. **The sort command** in Linux is used to sort the lines of a text file or input stream in a specific order. It takes the input, arranges it based on specified criteria, and produces the sorted output. By default, the sort command sorts the input in ascending order, line by line.

Here is the basic syntax of the sort command:

sort OPTIONS FILE

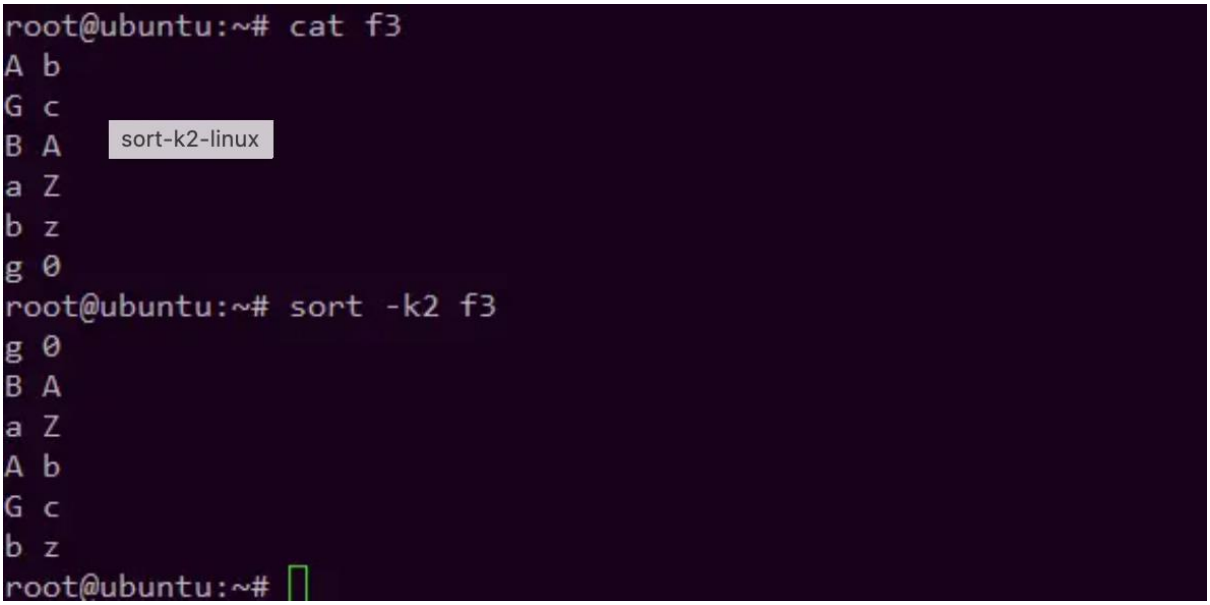
Some common options used with the sort command are:

- -r, --reverse: Sorts the input in descending order.
    - Example: sort -r filename

- -n, --numeric-sort: Sorts the input numerically instead of lexicographically.
    - Example: sort -n filename

```
root@ubuntu:~# cat f3
A b 1
G c 2
B A 5
a Z 3
b z 10
g 0 0.5
root@ubuntu:~# sort -nr f3
g 0 0.5
b z 10
a Z 3
G c 2
B A 5
A b 1
```

- -k, --key: Specifies the key or field to sort on.
    - Example: sort -k 2 filename sorts based on the second field.



- -t, --field-separator: Specifies the field separator character.
    - Example: sort -t ',' -k 3 filename sorts based on the third field using a comma as the delimiter.
- -u, --unique: Filters out duplicate lines in the sorted output.
    - Example: sort -u filename

```
root@ubuntu:~# cat f1
hi
hi
HI
Hi! How are you?
How are you!
Hello!I'm fine
root@ubuntu:~# sort -u f1
HI
Hello!I'm fine
Hi! How are you?
How are you!
hi
root@ubuntu:~#
```

The sort command can also read input from a pipe (stdin) if no file is specified. For example, you can use the output of another command as input to sort.

Additionally, the sort command can handle large files and files with multiple columns. It uses a stable sorting algorithm by default, preserving the original order of lines with equal keys.

Keep in mind that sort works on a per-line basis and treats each line as a separate unit for sorting. If you need more complex sorting requirements or custom sorting criteria, you can use the -k option to specify multiple keys and customize the sorting behavior.

For more details and advanced usage options, refer to the sort manual page (man sort) or consult the online documentation.

6. Uniq Command

The uniq command is a Unix/Linux command-line utility used to filter adjacent duplicate lines in a file or from standard input. It compares each line with the one immediately following it and removes any duplicate lines, leaving only unique lines in the output.

The basic syntax of the uniq command is as follows:

uniq [options] [input [output]]

Here's a breakdown of the components:

- uniq: The command itself.
    - [options]: Optional flags that modify the behavior of the command. Some common options include:
    - -c: Precede each line with the count of occurrences.
    - -d: Only output duplicate lines, skipping unique lines.
    - -u: Only output unique lines, skipping duplicate lines.
- [input]: Optional argument specifying the input file to process. If not provided, uniq reads from standard input.
- [output]: Optional argument specifying the output file to write the result. If not provided, the result is displayed on the terminal.

Here are a few examples to illustrate the usage of the uniq command:

1. Remove adjacent duplicate lines in a file named file.txt and display the unique lines:

uniq file.txt

```
$ sort file2 | uniq -c
    3 Budhha
    1 ChhatrapatiShahuMaharaj
    3 Dr.B.R.Ambedkar
```

2. Count the occurrences of each line in a file and display the count along with the line:

uniq -c file.txt

```
$ sort file2 | uniq -d
Budhha
Dr.B.R.Ambedkar
```

3. Display only the duplicate lines in a file:

uniq -d file.txt

```
$ sort file2 | uniq -u
ChhatrapatiShahuMaharaj
```

4. Display only the unique lines in a file:

uniq -u file.txt

5. Read input from standard input and display the unique lines:

```
echo -e "apple\nbanana\napple\norange" | uniq
```

These examples showcase some of the common uses of the uniq command. By default, uniq considers adjacent lines for comparison, so it is essential to ensure that the duplicate lines are consecutive for effective removal. For more details and advanced usage, you can refer to the uniq command's documentation (man uniq).

7 wc command

The wc command is a Unix/Linux command-line utility used to display information about the number of lines, words, and characters in a file or from standard input. It stands for "word count."

The basic syntax of the wc command is as follows:

wc [options] [file(s)]

Here's a breakdown of the components:

- `wc`: The command itself.
    - [options]: Optional flags that modify the behavior of the command. Some common options include:
    - -l: Display the number of lines.
    - -w: Display the number of words.
    - -c: Display the number of characters.

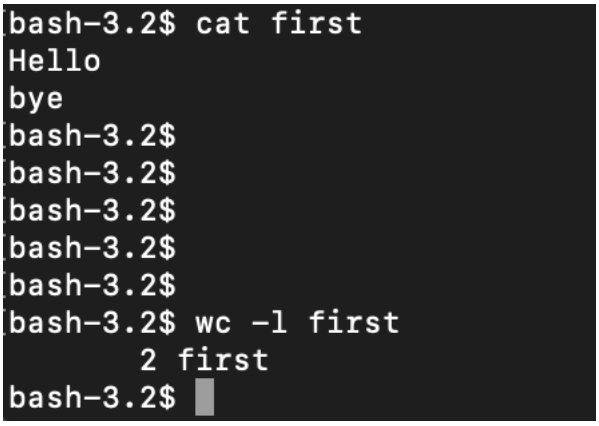
- -m: Display the number of characters (equivalent to -c).

- [file(s)]: Optional argument specifying the file(s) to process. If not provided, wc reads from standard input.

Here are a few examples to illustrate the usage of the wc command:

1. Display the number of lines, words, and characters in a file named file.txt:

```
wc file.txt
```



2. Display only the number of lines in a file:

```
wc -l file.txt
```

3. Display only the number of words in a file:


```
wc -w file.txt
```

```
bash-3.2$
bash-3.2$  wc -c first
      10 first
bash-3.2$
bash-3.2$
```

4. Display only the number of characters in a file:

```
wc -c file.txt
```

5. Read input from standard input and display the number of lines, words, and characters:

```
bash-3.2$ echo -e "Hello, world" | wc
       1       2      13
bash-3.2$
bash-3.2$
bash-3.2$
```

```
echo -e "Hello, world!" | wc
```

These examples demonstrate some of the common uses of the wc command. The output includes the total count as well as individual counts for each file specified. For more details and advanced usage, you can refer to the wc command's documentation (man wc).

8. diff Command

The diff command is a Unix/Linux command-line utility used to compare two files or directories and display the differences between them. It shows the added, removed, and modified lines or files, highlighting the changes between the compared versions.

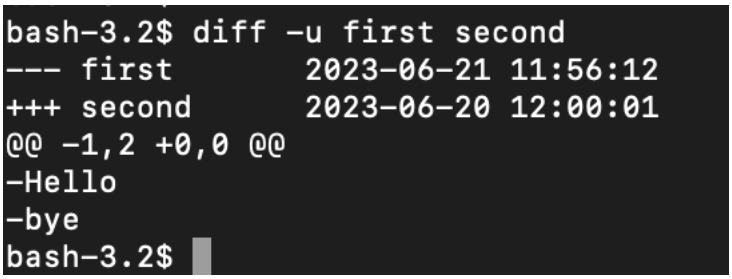
The basic syntax of the diff command is as follows:

diff [options] file1 file2

Here's a breakdown of the components:

- diff: The command itself.
  - [options]: Optional flags that modify the behavior of the command. Some common options include:
  - -u or --unified: Output a unified diff format, which provides context around the changes.
  - -c or --context: Output a context diff format, similar to unified diff but with more context lines.
  - -r or --recursive: Recursively compare directories and their contents.
- file1 and file2: The files or directories to compare. If comparing directories, the command compares corresponding files in both directories.

Here are a few examples to illustrate the usage of the diff command:



1. Compare two files and display the differences in unified diff format:

```
diff -u file1.txt file2.txt
```

2. Compare two directories and display the differences in context diff format:

```
bash-3.2$
bash-3.2$ diff -c -r first second
*** first       Wed Jun 21 11:56:12 2023
--- second      Tue Jun 20 12:00:01 2023
***************
*** 1,2 ****
- Hello
- bye
--- 0 ----
bash-3.2$
```

```
diff -c -r dir1 dir2
```

```
bash-3.2$ diff -q first second
Files first and second differ
bash-3.2$
```

3. Compare two files and suppress the output. The command exits with a status indicating whether the files are different:

```
diff -q file1.txt file2.txt
```

4. Compare two directories and generate a patch file containing the differences:

```
diff -u dir1 dir2 > patch.diff
```

These examples illustrate some of the common uses of the diff command. It provides a powerful way to identify and understand the differences between files or directories. For more details and advanced usage, you can refer to the diff command's documentation (man diff).

9 cmp command

The cmp command is a Unix/Linux command-line utility used to compare two files byte by byte. It identifies the first differing byte and reports its offset in the files. If the files are identical, no output is produced.

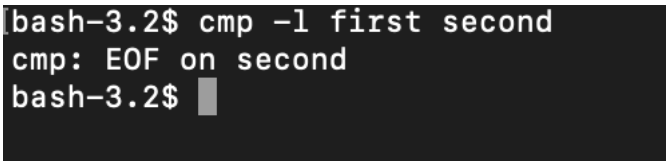
The basic syntax of the cmp command is as follows:

cmp [options] file1 file2

Here's a breakdown of the components:

- cmp: The command itself.
    - [options]: Optional flags that modify the behavior of the command. Some common options include:
    - -l or --verbose: Display the byte number and differing byte values.
    - -s or --quiet or --silent: Suppress normal output. Only return the exit status indicating whether the files are different or not.
- file1 and file2: The two files to compare.

Here are a few examples to illustrate the usage of the cmp command:



1. Compare two files and display the differing byte and its offset:

```
cmp -l file1.txt file2.txt
```

2. Compare two files and suppress normal output. Only return the exit status:

```
cmp -s file1.txt file2.txt
```

These examples demonstrate the basic usage of the cmp command. It provides a way to compare files at a binary level. For more details and advanced usage, you can refer to the cmp command's documentation (man cmp).

c. Difference between diff and cmp

The main difference between the diff and cmp commands lies in their purpose and the level of detail they provide when comparing files.

1. Purpose:
    - diff: The diff command is primarily used to compare and display the differences between two text files. It shows the added, removed, and modified lines, highlighting the changes between the compared files. It provides a more comprehensive and human-readable output that helps understand the differences at a line or context level.
    - cmp: The cmp command, on the other hand, compares two files byte by byte. It focuses on identifying the first differing byte and reporting its offset in the files. Its primary purpose is to check for binary differences between files and determine if they are identical or not.
2. Output:

- diff: The diff command generates a textual output that shows the specific lines that differ between the files, providing context and highlighting the changes. It provides a more detailed and user-friendly output format, often in the form of a unified or context diff.
- cmp: The cmp command produces output only when it encounters a difference between the compared files. By default, it reports the byte number and differing byte values. It is a more concise and low-level output, suitable for binary comparison.

3. Usage:

- diff: The diff command is commonly used for comparing text files, identifying differences in code, configurations, or any human-readable files. It is often used in conjunction with version control systems or for creating patches to apply changes between versions.
- cmp: The cmp command is useful for comparing binary files, such as executables or any files where the byte-level comparison is necessary. It can determine whether files are identical or not, based on their binary content.

In summary, diff is suitable for comparing text files, providing a detailed and contextualized output of the differences, while cmp is more focused on binary comparison, determining file identity, and reporting byte-level differences.

10. Compress and uncompress

10.1 tar command

The tar command is a Unix/Linux command-line utility used for creating, manipulating, and extracting archive files in the tar format. Tar stands for "tape archive," and it is commonly used for bundling multiple files and directories into a single archive file.

The basic syntax of the tar command is as follows:

tar [options] [archive-file] [file(s) or directory]

Here's a breakdown of the components:

- tar: The command itself.
- [options]: Optional flags that modify the behavior of the command. Some common options include:
    - -c: Create a new archive file.
    - -x: Extract files from an archive.
    - -t: List the contents of an archive.
    - -f <archive-file>: Specify the archive file to create, extract, or operate on.
    - -v: Verbose mode. Display detailed information about the files being processed.
    - -z: Compress the archive using gzip.
    - -j: Compress the archive using bzip2.
    - -C <directory>: Change to the specified directory before performing any operations.
    - [archive-file]: Optional argument specifying the name of the archive file to create, extract, or operate on. If not provided, tar reads from or writes to standard input/output.

      [file(s) or directory]: Optional argument specifying the file(s) or directory to include in the archive or extract from the archive.

Here are a few examples to illustrate the usage of the tar command:

1. Create a tar archive of a directory named myfiles:

   tar -cvf archive.tar myfiles

2. Extract the contents of a tar archive file named :

   ```
   tar -xvf archive.tar
   ```

3. List the contents of a tar archive file named :

   ```
   tar -tvf archive.tar
   ```

4. Create a compressed tar archive using gzip:

```
tar -czvf archive.tar.gz myfiles
```

5. Extract a compressed tar archive using gzip:

```
tar -xzvf archive.tar.gz
```

These examples demonstrate some of the common uses of the tar command. It provides a flexible way to create archives, add files to them, extract files, andperform various operations on tar files. For more details and advanced usage, you can refer to the tar command's documentation (man tar).

10.2 gzip Command

The gzip command is a Unix/Linux command-line utility used for compressing and decompressing files. It is commonly used to reduce the size of files and to save bandwidth when transferring files over the network.

The basic syntax of the gzip command is as follows:

To compress a file:

```
gzip [options] [file(s)]
```

To decompress a file:

```
gzip -d [options] [compressed-file(s)]
```

Here's a breakdown of the components:

- gzip: The command itself.
  - [options]: Optional flags that modify the behavior of the command. Some common options include:
  - -c: Write the compressed output to standard output instead of modifying the input file(s).

- -d: Decompress the specified file(s).
- -r: Recursively compress or decompress files in directories.
- -v: Verbose mode. Display detailed information about the compression or decompression process.
- -k: Keep the original file(s) when compressing or decompressing. By default, gzip replaces the input file(s) with the compressed or decompressed version.

- [file(s)]: The file(s) to compress. Multiple files can be specified for compression, and wildcard characters can be used to match multiple files. When decompressing, the compressed file(s) are specified.

Here are a few examples to illustrate the usage of the gzip command:

1. Compress a file named file.txt and replace it with the compressed version:

```
gzip file.txt
```

5. Compress multiple files and keep the original files:

```
gzip -k file1.txt file2.txt
```

6. Decompress a file named file.txt.gz and replace it with the decompressed version:

```
gzip -d file.txt.gz
```

7. Decompress multiple files and keep the original compressed files:

```
gzip -dk file1.txt.gz file2.txt.gz
```

8. Compress all files in a directory and its subdirectories recursively:

```
gzip -r directory/
```

These examples demonstrate some of the common uses of the gzip command. It provides a convenient way to compress and decompress files, reducing their size for storage or transmission. For more details and advanced usage, you can refer to the gzip command's documentation (man gzip).

10.3 gunzip command

The gunzip command is a Unix/Linux command-line utility used specifically for decompressing files that have been compressed with the gzip compression format. It is the counterpart to the gzip command and is often used to decompress files with the .gz extension.

The basic syntax of the gunzip command is as follows:

gunzip [options] [compressed-file(s)]

Here's a breakdown of the components:

- gunzip: The command itself.
    - [options]: Optional flags that modify the behavior of the command. Some common options include:
    - -c: Write the decompressed output to standard output instead of modifying the input file(s).
    - -k: Keep the original compressed file(s) after decompression. By default, gunzip removes the compressed file(s) once decompression is complete.
    - -v: Verbose mode. Display detailed information about the decompression process.
- [compressed-file(s)]: The file(s) to decompress. Multiple files can be specified for decompression, and wildcard characters can be used to match multiple files.

Here are a few examples to illustrate the usage of the gunzip command:

1. Decompress a file named file.txt.gz and replace it with the decompressed version:

```
gunzip file.txt.gz
```

2. Decompress a file named file.txt.gz and keep the original compressed file:

```
gunzip -k file.txt.gz
```

3. Decompress a file named file.txt.gz and output the decompressed contents to standard output:

```
gunzip -c file.txt.gz
```

4. Decompress multiple files and keep the original compressed files:

```
gunzip -k file1.txt.gz file2.txt.gz
```

These examples demonstrate the basic usage of the gunzip command. It is specifically designed for decompressing files in the gzip format. For more details and advanced usage, you can refer to the gunzip command's documentation (man gunzip).

11 Difference between tar, gzip and gunzip

tar, gzip, and gunzip are three distinct commands that serve different purposes in file compression and archiving:

1. **tar:**
    - Purpose: The tar command is used for creating archive files that bundle multiple files and directories together into a single file.
    - Syntax: tar [options] [archive-file] [file(s) or directory]
        - Key Points:
        - tar creates an uncompressed archive file by default.
        - It does not perform compression itself but can work in conjunction with compression utilities like gzip or bzip2 to create compressed archives.
        - It preserves the file structure and metadata (permissions, ownership, timestamps) of the archived files.
        - It can also perform extraction and listing operations on archive files.
2. **gzip:**
    - Purpose: The gzip command is used for compressing files using the gzip compression algorithm.
    - Syntax: gzip [options] [file(s)]
        - Key Points:
        - gzip compresses individual files, creating a compressed file with the .gz extension.
        - It replaces the original file with the compressed version by default.
        - It is typically used for compressing single files rather than creating archive files.

- gzip achieves good compression ratios with relatively fast compression and decompression speeds.

3. **gunzip:**

   - Purpose: The gunzip command is used specifically for decompressing files that have been compressed with the gzip compression format.
   - Syntax: gunzip [options] [compressed-file(s)]
     - Key Points:
     - gunzip is the counterpart to gzip and is used for decompressing files with the .gz extension.
     - It restores the original file by decompressing the .gz file.
     - It removes the compressed file after decompression by default.
     - It works solely for decompressing gzip-compressed files and does not create archives or perform other operations.

In summary, tar is used for creating archive files, gzip for compressing individual files using gzip compression, and gunzip for decompressing files that have been compressed with gzip. These commands can be used in combination to create compressed archive files, such as using tar to create an archive and gzip to compress it, and then gunzip to decompress the archive file when needed.