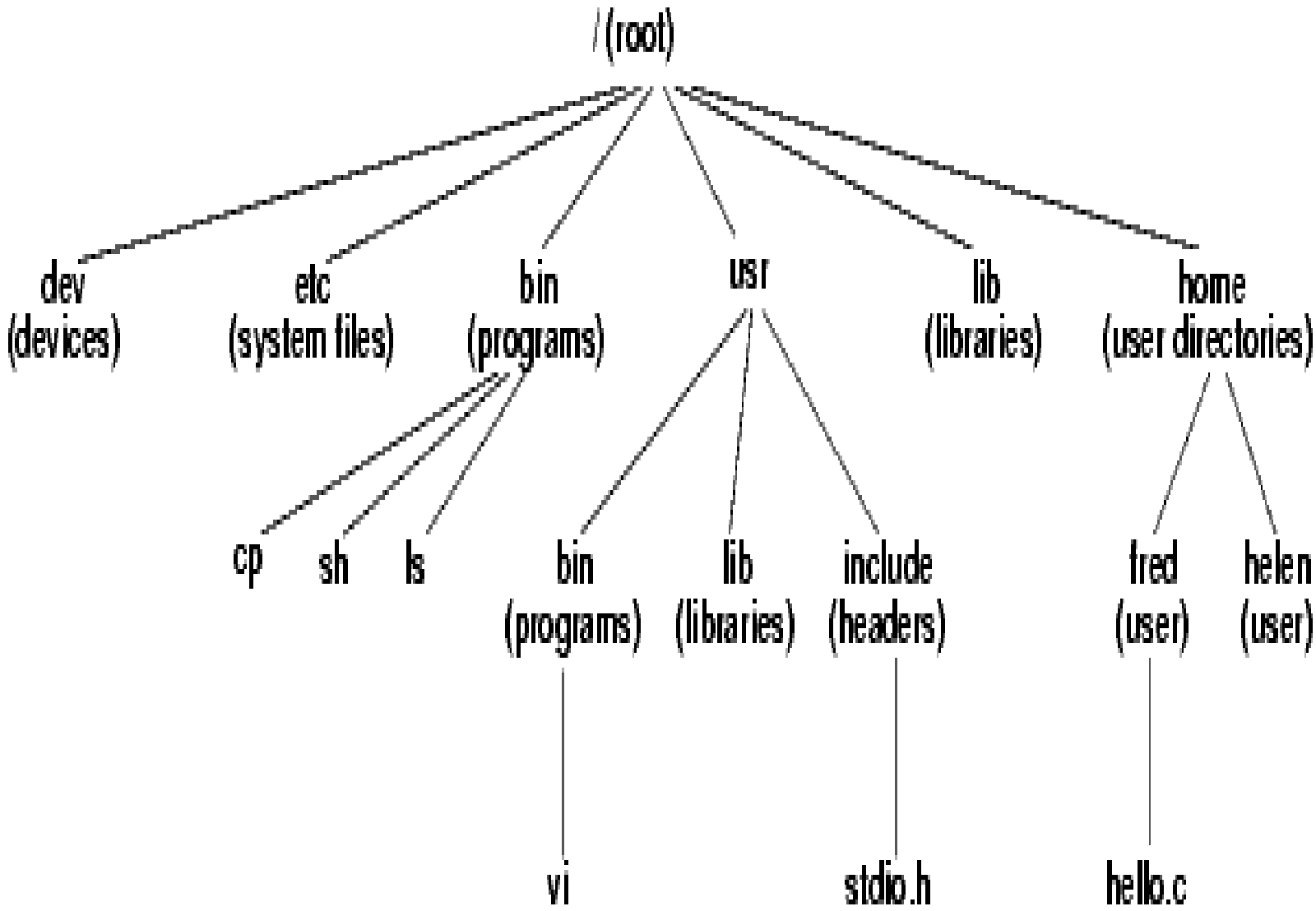
# UNIT 3
# **UNIX FILE STRUCTURE**

- The UNIX file system is a hierarchical arrangement of directories and files.

- Everything starts in the directory called root, whose name is the single character /.


/(root)
dev
(devices)
etc
(system files)
bin
(programs)
usr
lib
(libraries)
home
(user directories)
cp
sh
ls
bin
(programs)
lib
(libraries)
include
(headers)
fred
(user)
helen
(user)
vi
stdio.h
hello.c

# DIRECTORIES

- A directory is a file that contains directory entries.

- The attributes of a file are such things as the type of file (regular file, directory), the size of the file, the owner of the file, permissions for the file (whether other users may access this file), and when the file was last modified.

- The stat and fstat functions return a structure of information containing all the attributes of a file.

- **Filename: -**The names in a directory are called filenames.

- The only two characters that cannot appear in a filename are the slash character (/) and the null character.

- The slash separates the filenames that form a pathname (described next) and the null character terminates a pathname.

- **Pathname: -**A sequence of one or more filenames separated by slashes and optionally starting with a slash, forms a pathname.

- A pathname that begins with a slash is called an absolute pathname; otherwise, it’s called a relative pathname.

- **Working Directory: -** Every process has a working directory, sometimes called the current working directory.

- A process can change its working directory with the chdir function.

- **Home Directory: -** When we log in, the working directory is set to our home directory. Our home directory is obtained from our entry in the password file.

# FILES AND DEVICES

- Files are stored on devices such as **Hard Disks** and **Floppy Disks**.

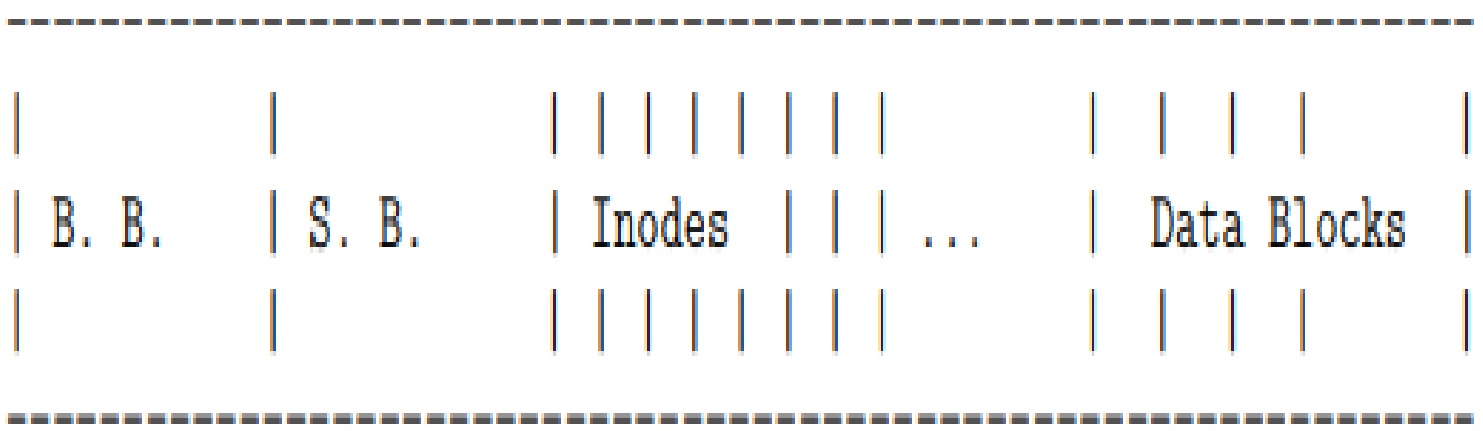
- Operating System defines a File System on Devices which is usually **Hierarchical file system** including UNIX.

- **FILE SYSTEM-** A Group of files and its relevant information forms File System and is stored on Hard Disk.

- On a Hard Disk, a Unix File system is stored in terms of blocks where each block is equal to 512 bytes.

- The Block size can be increased up to 2048 bytes.

1. Boot block
2. Super block
3. I nodes
4. Data blocks.



- **Boot Block: -** A boot block located in the first few sectors of a file system. The boot block contains the initial bootstrap program used to load the operating system.

- **Super Block: -** A super block describes the state of the file system: the total size of the partition, the block size, pointers to a list of free blocks, the inode number of the root directory, magic number, etc.

- **I – nodes: -** There is a one to one mapping of files to inodes and vice versa.

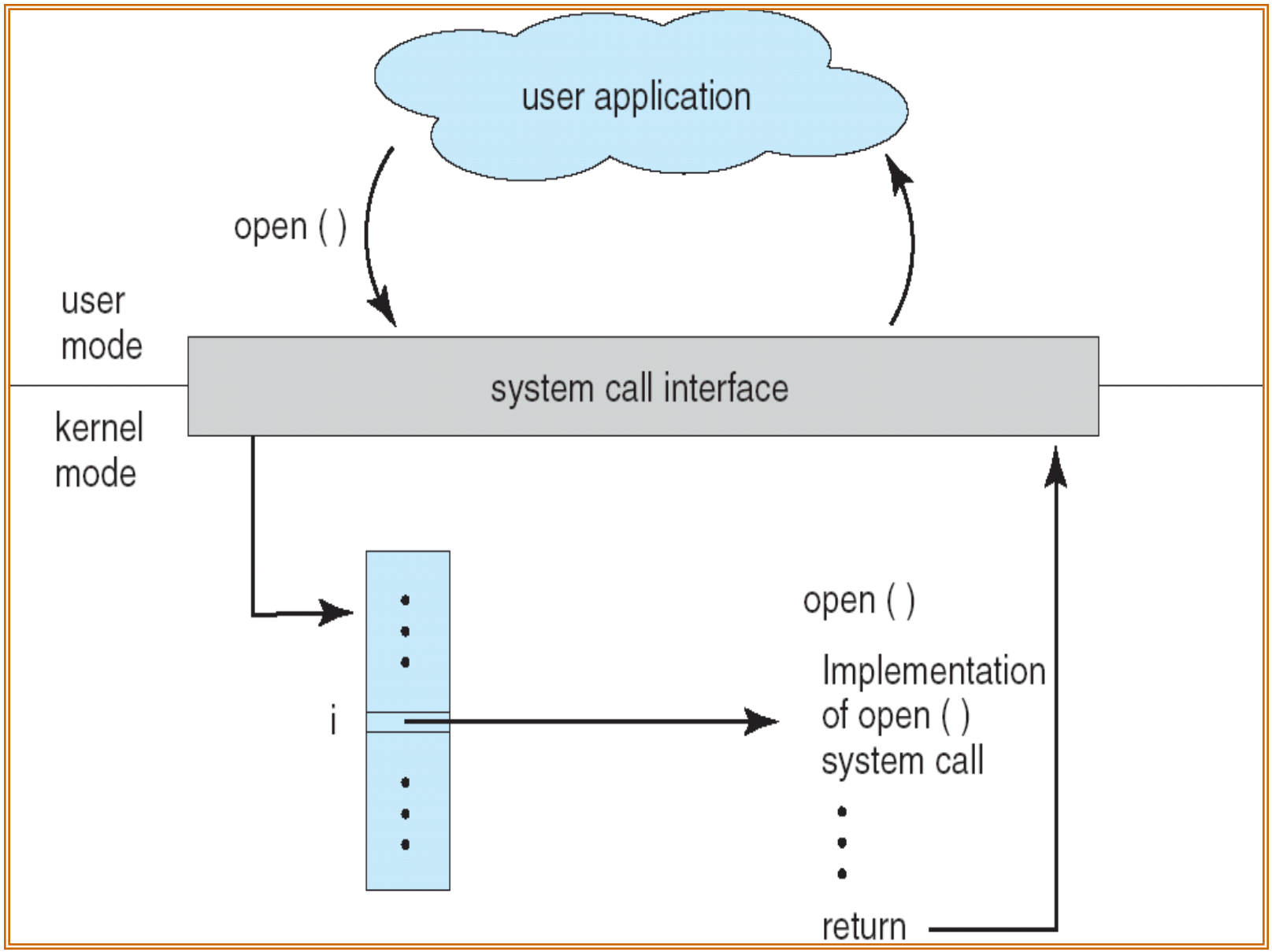
- Thus, while users think of files in terms of file names, Unix thinks of files in terms of inodes.

- **Data blocks: -** Data blocks containing the actual contents of files

# SYSTEM CALLS

- In computing, a **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on.

- A system call is a way for programs to **interact with the operating system**.


user application
open ( )
user mode
system call interface
kernel mode
i
open ( )
Implementation of open ( ) system call
return

- **Types of System Calls: -**
- There are 5 different categories of system calls

- **Process control:** end, abort, create, terminate, allocate and free memory.
- **File management:** create, open, close, delete, read file etc.
- **Device management**
- **Information maintenance**
- **Communication**

| S.NO | Types of System calls | WINDOWS | UNIX |
|---|---|---|---|
| 1 | Process Control | CreateProcess()<br>ExitProcess()<br>WaitForSingleObject() | fork()<br>exit()<br>wait() |
| 2 | File Manipulation | CreateFile()<br>ReadFile()<br>WriteFile()<br>CloseHandle() | open()<br>read()<br>write()<br>close() |
| 3 | Device Manipulation | SetConsoleMode()<br>ReadConsole()<br>WriteConsole() | ioctl()<br>read()<br>write() |
| 4 | Information Maintenance | GetCurrentProcessID()<br>SetTimer()<br>Sleep() | getpid()<br>alarm()<br>sleep() |
| 5 | Communication | CreatePipe()<br>CreateFileMapping()<br>MapViewOfFile() | pipe()<br>shmget()<br>mmap() |
| 6 | Protection | SetFileSecurity()<br>InitlializeSecurityDescriptor()<br>SetSecurityDescriptorGroup() | chmod()<br>umask()<br>chown() |

- **File structure related system calls**
- These calls return a file descriptor that identifies the I/O channel.
- creat()
- open()
- close()
- read()
- write()
- lseek()

- dup()
- link()
- unlink()
- stat()
- fstat()
- access()
- chmod()
- chown()
- umask()
- ioctl()

# LIBRARY FUNCTIONS

- For calculating string length, there exists a standard function like strlen(), for opening a file, there exists functions like open() and fopen().

- We call these functions as standard functions as any application can use them.

- These standard functions can be classified into two major categories:
- Library function calls.
- System function calls.

- **Library functions Vs System calls: -**

- The functions which are a part of standard C library are known as Library functions.

- For example the standard string manipulation functions like strcmp(), strlen() etc are all library functions.

- The functions which change the execution mode of the program from user mode to kernel mode are known as system calls.

- These calls are required in case some services are required by the program from kernel.

- For example, if we want to change the date and time of the system or if we want to create a network socket then these services can only be provided by kernel and hence these cases require system calls.

- For example, socket() is a system call.

- **Types of library functions: -**

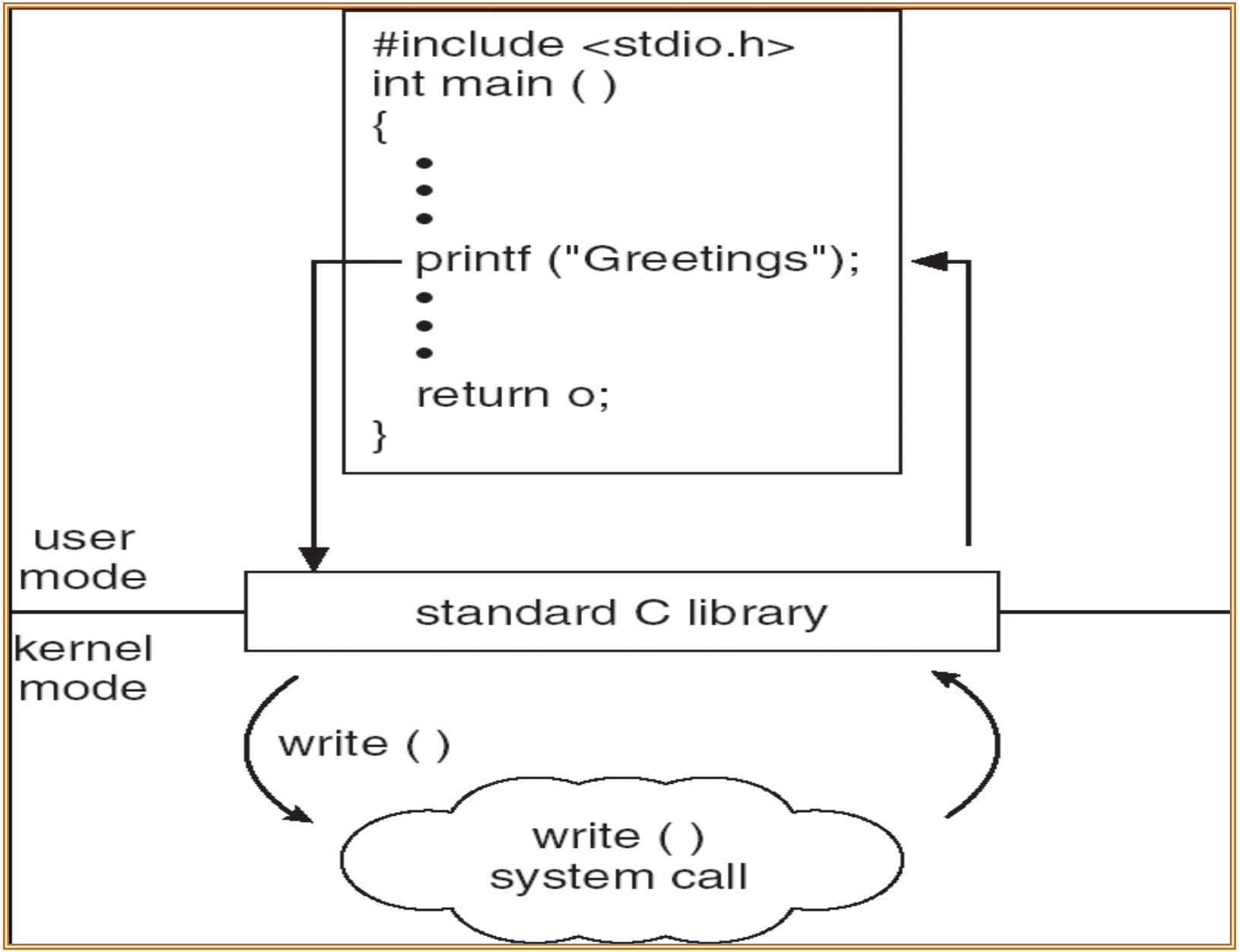
- Library functions can be of two types:

- Functions which do not call any system call.
- Functions that make a system call.

- There are library functions that do not make any system call.

- For example, the string manipulation functions like strlen() etc fall under this category.

- Also, there are library functions that further make system calls, for example the fopen() function which a standard library function but internally uses the open() system call.


#include <stdio.h>
int main ( )
{
printf ("Greetings");
return o;
}
user mode
kernel mode
standard C library
write ( )
write ( )
system call

# LOW LEVEL FILE ACCESS

- Provides direct access to files and devices.
- Is complex (Buffer management is to done by the programmer)
- When using I/O functions, low-level I/O is faster as compared to the high-level I/O.
- Uses a “file descriptor” to track the status of the file.

- **Low-level I/O functions are used for:**
- Accessing files and devices directly.
- Reading binary files in large chunks.
- Performing I/O operations quickly and efficiently.

- The low-level I/O system in C provides functions that can be used to access files and devices.
  - Open()
  - Close()
  - Read()
  - Write()

# open()

- Used to Open the file for reading, writing or both.

**Syntax in C language**

```
#include<sys/types.h>
#includ<sys/stat.h>
#include <fcntl.h>
int open (const char* Path, int flags [, int mode ]);
```

- **Parameters: -**

- **Path: -** path to file which you want to use.
- Use absolute path begin with “/”, when you are not work in same directory of file.

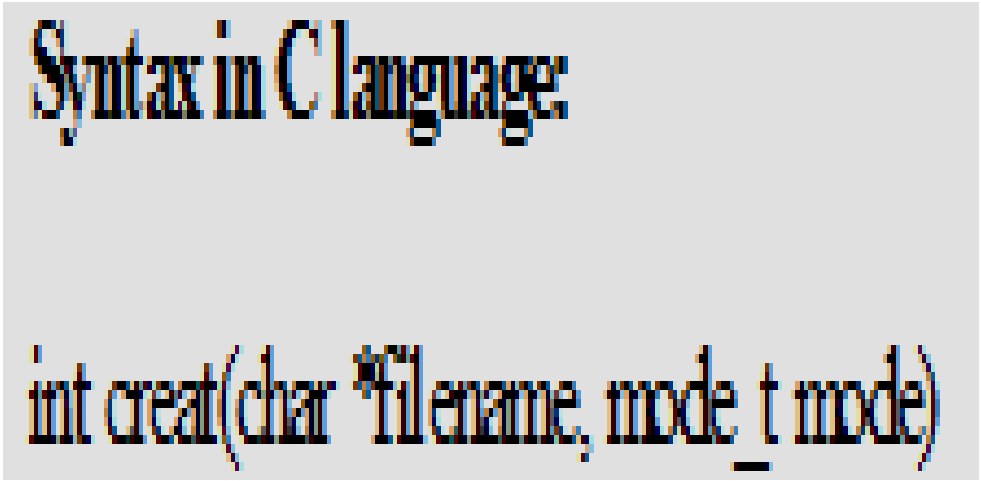
- Use relative path which is only file name with extension, when you are work in same directory of file.

- **Flags: -** How you like to use

- **O_RDONLY**: read only
- **O_WRONLY**: write only
- **O_RDWR**: read and write
- **O_CREAT**: create file if it doesn’t exist
- **O_EXCL**: prevent creation if it already exists

# creat()

- Used to Create a new empty file.



**Parameter: -**

- **filename: -** name of the file which you want to create
- **mode: -** indicates permissions of new file.

- **Returns: -**
- return first unused file descriptor (generally 3 when first creat use in process beacuse 0, 1, 2 fd are reserved).

- return -1 when error

# read()

- From the file indicated by the file descriptor fd, the read() function reads cnt bytes of input into the memory area indicated by buf. A successful read() updates the access time for the file.

Syntax in C language

```
size_t read (int fd, void* buf, size_t cnt);
```

**Parameters**

**fd:** file descriptor

**buf:** buffer to read data from

**cnt:** length of buffer

# File descriptors

- To kernel all open files are referred to by file descriptors.
- A file descriptor is a non negative integer.
- When we open an existing or create a new file, the kernel returns a file descriptor to a process.

- Each UNIX process has 20 file descriptors and it disposal, numbered 0 through 19 but it was extended to 63 by many systems.

- The first three are already opened when the process begins
- 0: the standard input
- 1: the standard output
- 2: the standard error output.

**Returns: How many bytes were actually read**

- return Number of bytes read on success
- return 0 on reaching end of file
- return -1 on error
- return -1 on signal interrupt

- **Important points**
- **buf** needs to point to a valid memory location with length not smaller than the specified size because of overflow.

- **fd** should be a valid file descriptor returned from open() to perform read operation because if fd is NULL then read should generate error.

- **cnt** is the requested number of bytes read, while the return value is the actual number of bytes read. Also, sometimes read system call should read less bytes than cnt.

# write

- Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.

```
#include <fcntl.h>

size_t write (int fd, void* buf, size_t cnt);
```

**Parameters**

- **fd:** file descriptor
- **buf:** buffer to write data to
- **cnt:** length of buffer

- **Returns: How many bytes were actually written?**
- return Number of bytes written on success
- return 0 on reaching end of file
- return -1 on error
- return -1 on signal interrupt

# close

- Tells the operating system you are done with a file descriptor and Close the file which pointed by fd.

**Syntax in C language**

```
#include <fcntl.h>
int close(int fd);
```

- **Parameter**
- **fd :**file descriptor

- **Return**
- **0** on success.
- **-1** on error.

# lseek

- The UNIX system file system treats an ordinary file as a sequence of bytes. Generally, a file is read or written sequentially -- that is, from beginning to the end of the file.

- Sometimes sequential reading and writing is not appropriate. Random access I/O is achieved by changing the value of this file pointer using the lseek() system call.

- **Syntax: -**

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fildes, off_t offset, int whence);
```

- **Description: -**
- The **lseek**() function repositions the offset of the open file associated with the file descriptor *fildes* to the argument *offset* according to the directive *whence* as follows:

| Tag | Description |
| --- | --- |
| **SEEK_SET** | |
| | The offset is set to *offset* bytes. |
| **SEEK_CUR** | |
| | The offset is set to its current location plus *offset* bytes. |
| **SEEK_END** | |
| | The offset is set to the size of the file plus *offset* bytes. |

- **Return Value: -**
- Upon successful completion, **lseek**() returns the resulting offset location as measured in bytes from the beginning of the file. Otherwise, a value of *(off_t)-1* is returned and *errno* is set to indicate the error.

- **Notes: -**
- When converting old code, substitute values for *whence* with the following macros:

| old | new |
|---|---|
| 0 | SEEK_SET |
| 1 | SEEK_CUR |
| 2 | SEEK_END |
| L_SET | SEEK_SET |
| L_INCR | SEEK_CUR |
| L_XTND | SEEK_END |

# stat, fstat, lstat System Calls

- **stat(): -** stat() function is used to access the files information such as the type of file owner of the file, file access permissions, file size etc.
- **Syntax: -**
- #include<sys/types.h>
- #include<sys/stat.h>
- int stat(const char *filename, struct stat *buff);

- Given a filename, stat() function will retrieve the files information into a stat structure pointed to by ‘buff’.

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;  /* number of blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

- **fstat(): -** fstat() function obtains the information about the file that is already open.

- **Syntax:** -
- #include<sys/types.h>
- #include<sys/stat.h>
- int  fstat(int fd, struct stat *buff);

- fstat() retrieves information about the file opened with file descriptor fd into the stat structure pointed to by ‘buff’.

- **lstat(): -** The lstat() function is similar to the stat() function, that is, it is also used to access the information about a named file.

**Syntax: -**

- #include<sys/types.h>
- #include<sys/stat.h>
- int lstat(const char *filename, struct stat *buff);

- The lstat() retrieves the information about the filename into a stat structure pointed to by buff.

- **lstat**() is identical to **stat**(), except that if *path* is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

- Returns ‘0’ on success and ‘-1’ on error.

# ioctl()

- The **ioctl**() function manipulates the underlying device parameters of special files.

- In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with **ioctl**() requests.

- **Syntax: -**
- #include <sys/ioctl.h>
- int ioctl(int *d*, int *request*, ...);

- **Description: -**
- The argument *d* must be an open file descriptor.
- The second argument is a device-dependent request code.

- The third argument is an untyped pointer to memory. It's traditionally **char** ***argp*** (from the days before **void** * was valid C)

# dup and dup2 system calls

- **dup(): -** The dup() system call creates a copy of a file descriptor.

- It uses the lowest-numbered unused descriptor for the new descriptor.
- If the copy is successfully created, then the original and copy file descriptors may be used interchangeably.

- They both refer to the same open file description and thus share file offset and file status flags.
- **Syntax: -**

```
int dup(int oldfd);

oldfd: old file descriptor whose copy is to be created.
```

- **dup2(): -** The dup2() system call is similar to dup() but the basic difference between them is that instead of using the lowest-numbered unused file descriptor, it uses the descriptor number specified by the user.

- **Syntax: -**

```
int dup2(int oldfd, int newfd);

oldfd: old file descriptor
newfd new file descriptor which is used by dup2() to create a copy.
```

# link() system call

- **link**() creates a new link (also known as a hard link) to an existing file.

- **Syntax: -**

  **#include <unistd.h>**

  **int link(const char ****oldpath***, const char ****newpath***);**

- **Description: -**
- If *newpath* exists it will *not* be overwritten. This new name may be used exactly as the old one for any operation; both names refer to the same file (and so have the same permissions and ownership) and it is impossible to tell which name was the `original’.

- **Return Value: -** On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

- **Note: -** Hard links, as created by **link**(), cannot span file systems. Use **symlink**() if this is required.

# symlink()

- **Syntax: -**
- **#include <unistd.h>**
- **int symlink(const char **oldpath*, const char **newpath*);**

- **Description: -**
- **symlink**() creates a symbolic link named *newpath* which contains the string *oldpath*.

# unlink() system call

- delete a name and possibly the file it refers to

- **Syntax: -**

**#include <unistd.h>**

**int unlink(const char **pathname*);**

**Description**

- **unlink**() deletes a name from the file system.

- If that name was the last link to a file and no processes have the file open the file is deleted and the space it was using is made available for reuse.

**Return Value**

- On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

# UNIT 4
# **UNIX PROCESS**

- A process, in simple terms, is an instance of a running program.

- If, for example, three people are running the same program simultaneously, there are three processes there, not just one.

- The operating system tracks processes through a five-digit ID number known as the **PID(**Process Identification Number**)** or the **process ID**.

- Each process in the system has a unique **PID**.

- Unix is a timesharing system, which means that the processes take turns running. Each turn is a called a time slice.

- On most systems this is set at much less than one second.

- The reason this turns-taking approach is used is fairness. We don’t want a 2-second job to have to wait for a 5-hour job to finish

# What is process

- A **process** is a program in execution in memory or in other words, an instance of a program in memory.

- Any program executed creates a process.

- A program can be a command, a shell script, or any binary executable or any application.

- However, not all commands end up in creating process, there are some exceptions.

- Similar to how a file created has properties associated with it, a process also has lots of properties associated to it.

**Process attributes:**

- A process has some properties associated to it:
- **PID**
- **PPID**
- **TTY**
- **UID**

**PID: - Process-Id.**

- Every process created in Unix/Linux has an identification number associated to it which is called the process-id.

- The PID is unique for a process at any given point of time. However, it gets recycled.

**PPID** : Parent Process Id:

- Every process has to be created by some other process.
- The process which creates a process is the parent process, and the process being created is the child process.
- The PID of the parent process is called the parent process id(PPID).

**TTY**:

- Terminal to which the process is associated to. Every command is run from a terminal which is associated to the process.

- However, not all processes are associated to a terminal. There are some processes which do not belong to any terminal. These are called daemons.

**UID**: User Id-

- The user to whom the process belongs to. And the user who is the owner of the process can only kill the process(Of course, root user can kill any process).

- When a process tries to access files, the accessibility depends on the permissions the process owner has on those files.

## **File Descriptors**:

- File descriptors related to the process: input, output and error file descriptors.

- **List the processes**:
- $ ps

| PID | TTY | TIME | CMD |
|---|---|---|---|
| 1315012 | pts/1 | 0:00 | -ksh |
| 2490430 | pts/1 | 0:00 | ps |

- **ps** is the Unix command which lists the active processes and its status.
- The ps command output shows 4 things:

  **PID :** The unique id of the process
  **TTY:** The terminal from which the process or command is executed.
  **TIME:** The amount of CPU time the process has taken
  **CMD:** The command which is executed.

- 2 processes are listed in the above case:

  **1**. -ksh : The login shell, which we are working on, is also a process which is currently running.

  **2**. ps : The ps command which we executed to get the list also creates a process.

- **Parent & Child Process**:
- Every process in Unix has to be created by some other process.

- Hence, the ps command is also created by some other process.

- The 'ps' command is being run from the login shell, ksh.

- The ksh shell is a process running in the memory right from the moment the user logged in.

- So, for all the commands triggered from the login shell, the login shell will be the parent process and the process created for the command executed will be the child process.

- In the same lines, the 'ksh' is the parent process for the child process 'ps'.

- $ ps -o pid,ppid,args

  | PID | PPID | COMMAND |
  |---|---|---|
  | 2666744 | 3317840 | ps -o pid,ppid,args |
  | 3317840 | 1 | -ksh |

The PID of the ksh is same as the PPID of the ps command which means the ksh process is the parent of the ps command. The '-o' option of the ps command allows the user to specify only the fields which he needs to display.

**Init Process:**

- If all processes of the user are created by the login shell, who created the process for the login shell?

- In other words, which is the parent process of the login shell?

- When the Unix system boots, the first process to be created is the *init* process.

- This init process will have the PID as 1 and PPID as 0.

- All the other processes are created by the init process and gets branched from there on.

- Note in the above command, the process of the login shell has the PPID 1 which is the PID of the *init* process.

# PROCESS STRUCTURE

- The kernel has a process table where it stores the state of the process and other information about the process.

- The information of the entry and the u-area of the process combined is the *context* of the process.

**PROCESS STATES**

- Every process in the system can be in one of six states.
- The six possible states are as follows:

1) ***Running,*** which means that the process is currently using the CPU.

2) ***Runnable,*** which means that the process can make use of the CPU as soon as it becomes available.

3) ***Sleeping,*** which means that the process is waiting for an event to occur.

For example, if a process executes a “read()” system call, it sleeps until the I/O request completes.

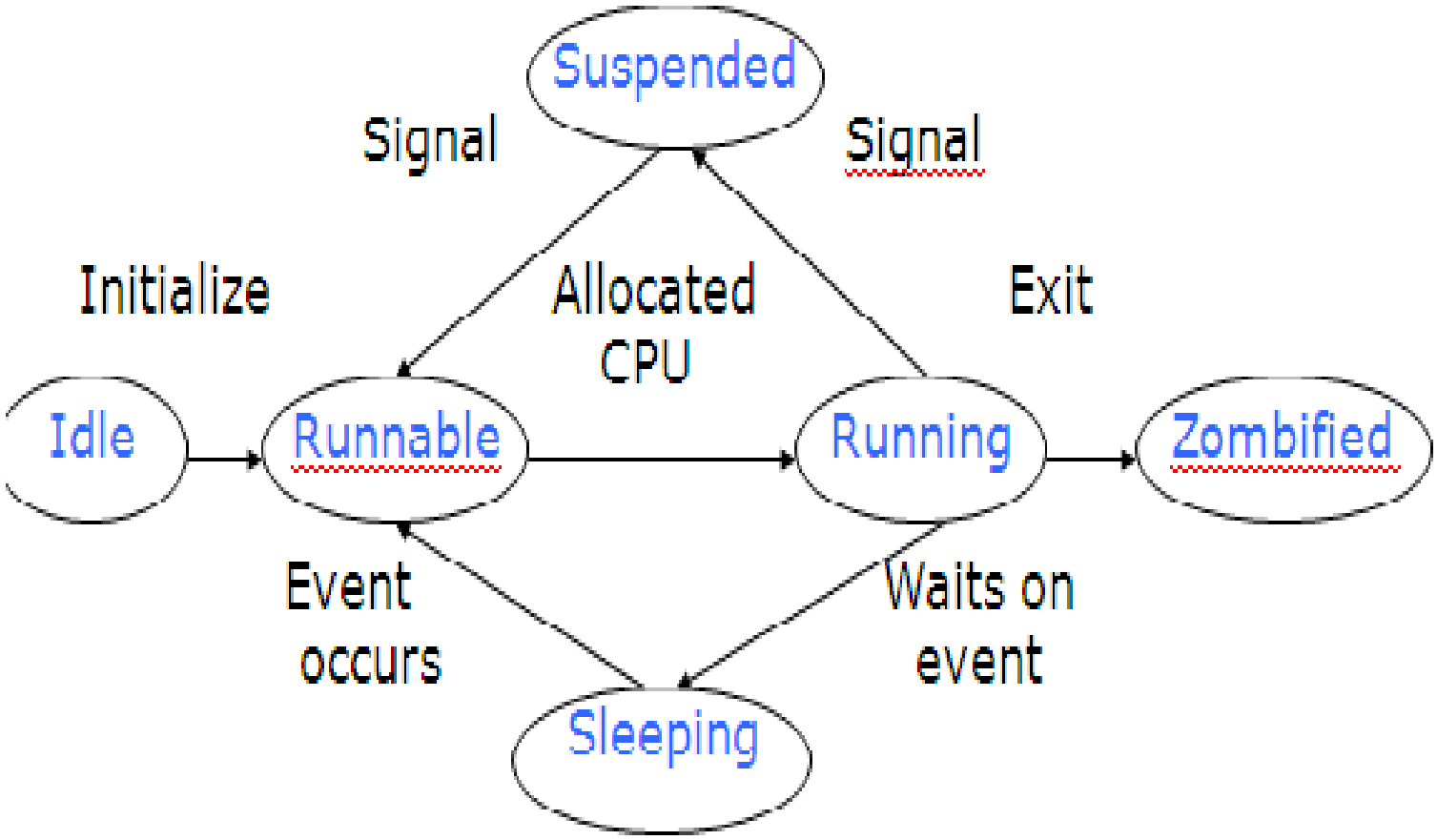
4) ***Suspended,*** which means that the process has been “frozen” by a signal such as SIGSTOP.

It will resume only when sent a SIGCONT signal.

5) ***Idle,*** which means that the process is being created by a “fork() system call and is not yet runnable.

6) ***Zombified,*** which means that the process has terminated but has not yet returned its exit code to its parent.

- A process remains a zombie until its parent accepts its return code using the “wait()” system call.




[ *Process States* ]

## PROCESS COMPOSITION

- Every process is composed of several different pieces:
- ***a code area,*** which contains the executable(text) portion of a process
- ***a data area,*** which is used by a process to contain static data
- ***a stack area,*** which is used by a process to store temporary data
- ***a user area,*** which holds housekeeping information about a process
- ***page tables,*** which are used by the memory management system

- **User Area**

- Every process in the system has some associated “housekeeping” information that is used by the kernel for process management.

- This information is stored in a data structure called a user area. Every process has its own user area.

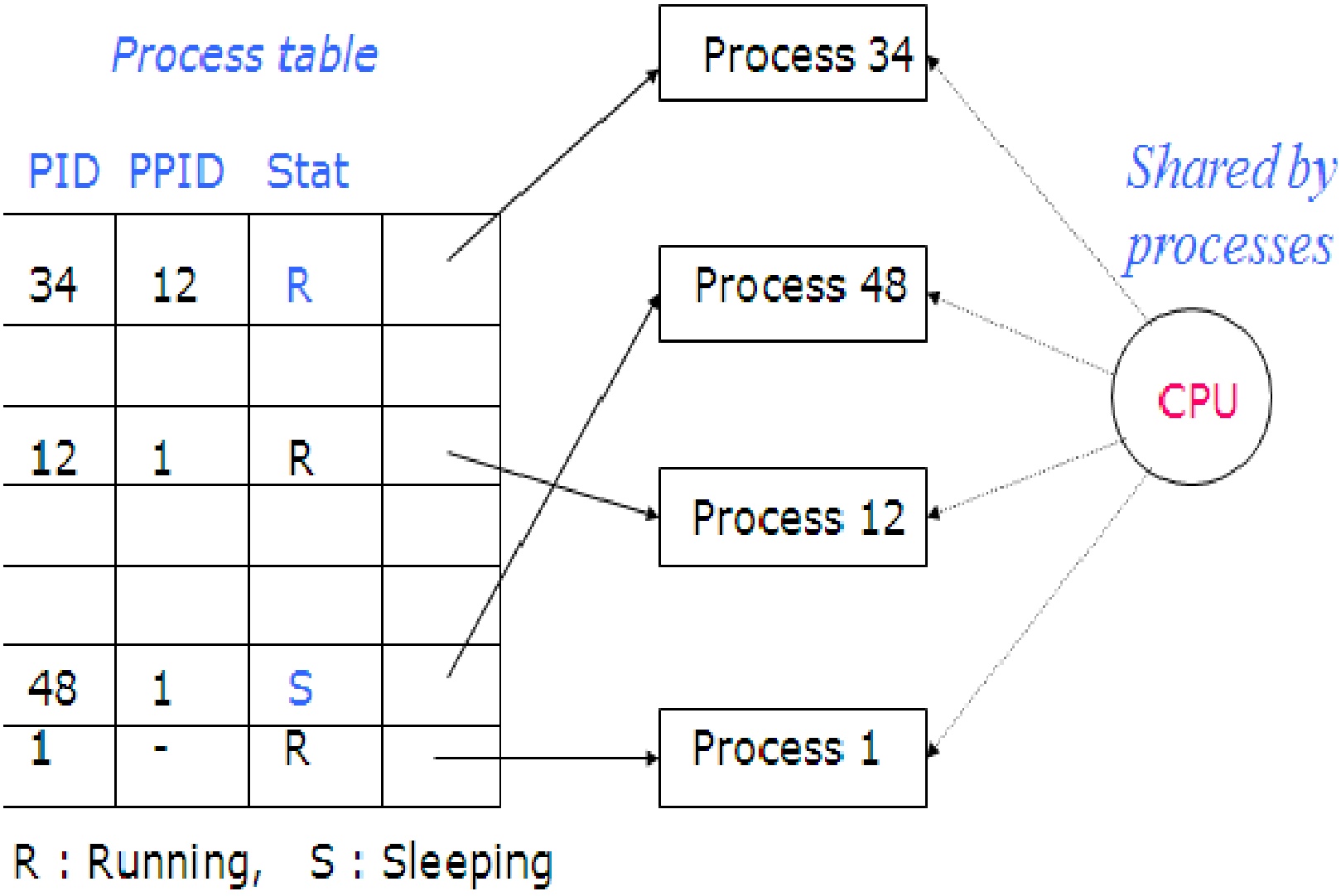
- User areas are created in the kernel's data region and are only accessible by the kernel; user processes may not access their user areas.

## The Process Table

- There is a single kernel data structure of fixed size called the process table that contains one entry for every process in the system.

- The process table is created in the kernel's data region and is accessible only by the kernel.

- Each entry contains the following information about each process:
- its process ID(PID) and parent process ID(PPID)
- its real and effective user ID(UID) and group ID(GID)
- its state ( running, runnable, sleeping, suspended, idle, or zombified )
- the location of its code, data, stack, and user areas
- a list of all pending signals


Process table
PID PPID Stat
34 12 R
12 1 R
48 1 S
1 - R
Process 34
Process 48
Process 12
Process 1
Shared by processes
CPU
R : Running, S : Sleeping

# UNIT 5
# SIGNALS

Program must sometimes deal with unexpected or unpredictable events, such as :

- a floating point error
- a power failure
- an alarm clock “ring”
- the death of a child process
- a termination request from a user( i.e.,Control-C )
- a suspend request from a user ( i.e., Control-Z )

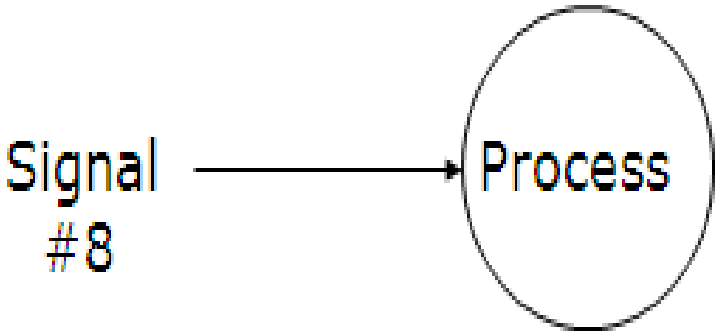
- These kind of events are sometimes called *interrupts*, as they must interrupt the regular flow of a program in order to be processed.

- When UNIX recognizes that such an event has occurred, it sends the corresponding process a **signal.**

- There is a unique, numbered signal for each possible event.

**For example:-**

- if a process causes a floating point error, the kernel sends the offending process signal number 8:



- Any process can send any other process a signal, as long as it has permission to do so.

- A programmer may arrange for a particular signal to be ignored or to be processed by a special piece of code called a signal handler.

- The process that receives the signal

  1) suspends its current flow of control,

  2) executes the signal handler,

  3) and then resumes the original flow of control when the signal handler finishes

- Signals are defined in “/usr/include/sys/signal.h”.

  or #include<signal.h>

- The default handler usually performs one of the following actions:
  - terminate the process and generate a core file **( *dump* )**
  - terminate the process without generating a core image file (***quit*** )
  - ignore and discard the signal ( ***ignore*** )
  - suspend the process **( *suspend* )**
  - resume the process

- *Sending Signals*
- There are several methods of delivering signals to a program or script. One of the most common is for a user to type **CONTROL-C** or the **INTERRUPT key** while a script is executing.
- When you press the ***Ctrl+C*** key, a **SIGINT** is sent to the script and as per defined default action script terminates.

- The other common method for delivering signals is to use the **kill command**, the syntax of which is as follows −

- $ kill -signal pid
- Here **signal** is either the number or name of the signal to deliver and **pid** is the process ID that the signal should be sent to.

- **For Example –**

- $ kill -1 1001

- The above command sends the HUP or hang-up signal to the program that is running with **process ID 1001**.

- To send a kill signal to the same process, use the following command

- $ kill -9 1001

- This kills the process running with **process ID 1001**.

# LIST OF SIGNALS

- - Here’s a list of the System V predefined signals, along with their respective macro definitions, numerical values, and default actions, as well as a brief description of each:

| Macro | # | Default | Description |
|---|---|---|---|
| SIGHUP | 1 | quit | hang up |
| SIGINT | 2 | quit | interrupt |
| SIGQUIT | 3 | dump | quit |
| SIGILL | 4 | dump | illegal instruction |
| SIGTRAP | 5 | dump | trace trap( used by debuggers ) |
| SIGABRT | 6 | dump | abort |
| SIGEMT | 7 | dump | emulator trap instruction |
| SIGFPE | 8 | dump | arithmetic execution |
| SIGKILL | 9 | quit | kill( cannot be caught, blocked, or ignored ) |
| SIGBUS | 10 | dump | bus error( bad format address ) |
| SIGSEGV | 11 | dump | segmentation violation(out-of-range address) |
| SIGSYS | 12 | dump | bad argument to system call |
| SIGPIPE | 13 | quit | write on a pipe or other socket with no one to read it. |

| | | | |
|---|---|---|---|
| SIGALRM | 14 | quit | alarm clock |
| SIGTERM | 15 | quit | software termination signal(default signal sent by kill ) |
| SIGUSR1 | 16 | quit | user signal 1 |
| SIGUSR2 | 17 | quit | user signal 2 |
| SIGCHLD | 18 | ignore | child status changed |
| SIGPWR | 19 | ignore | power fail or restart |
| SIGWINCH | 20 | ignore | window size change |
| SIGURG | 21 | ignore | urgent socket condition |
| SIGPOLL | 22 | ignore | pollable event |
| SIGSTOP | 23 | quit | stopped(signal) |
| SIGSTP | 24 | quit | stopped(user) |
| SIGCONT | 25 | ignore | continued |
| SIGTTIN | 26 | quit | stopped( tty input ) |
| SIGTTOU | 27 | quit | stopped( tty output ) |
| SIGVTALRM | 28 | quit | virtual timer expired |
| SIGPROF | 29 | quit | profiling timer expired |
| SIGXCPU | 30 | dump | CPU time limit exceeded |
| SIGXFSZ | 31 | dump | file size limit exceeded |

- **Unreliable Signals**

Def: Unreliable Signals..

- Signals could get lost without notice by process

**Why?**

- The action for a signal was reset to its default each time the signal occurred.
- The process could only ignore signals, instead of turning off the signals

# Interrupted System Calls

- In earlier UNIX systems, if a process caught a signal while the process was blocked in a "slow" system call, the system call was interrupted.

- The system call returned an error and errno was set to EINTR.

- This was done under the assumption that since a signal occurred and the process caught it, there is a good chance that something has happened that should wake up the blocked system call.

**Slow system calls**

- The system calls are divided into two categories: the "slow" system calls and all the others.
- The slow system calls are those that can block forever:

- The system calls that were automatically restarted are:
  - ioctl
  - read
  - readv
  - write
  - writev

- Functions that are always interrupted when a signal is caught.
  - wait
  - waitpid

# kill and raise Functions

- The kill function sends a signal to a process or a group of processes.
- The raise function allows a process to send a signal to itself.

```
#include <signal.h>

int kill(pid_t pid, int signo);
int raise(int signo);

/* Both return: 0 if OK, −1 on error */
```

- “kill()” sends the signal with value *sigCode* to the process with PID *pid*.
- **Conditions on pid:**

1. *pid* > 0 : signal sent to the process with UID pid
2. *pid* == 0: signal sent to “all” processes with the same gid of the sender (excluding proc 0, 1, 2)
3. *pid* < 0: signal sent to “all” processes with process gid == |pid|

The call:

```
raise(signo);
```

is equivalent to the call:

```
kill(getpid(), signo);
```

- **raise()** function can only send a signal to the program that contains it.
- **raise()** cannot send a signal to other processes.
- To send a signal to other processes, the system call **kill()** should be used.

# alarm and pause Functions

- The alarm function sets a timer that will expire at a specified time in the future.
- When the timer expires, the SIGALRM signal is generated.
- If we ignore or don’t catch this signal, its default action is to terminate the process.

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

- Seconds specifies expiration delay.

- alarm(0) cancels the alarm if not yet expired. Returns number of seconds left.

```
#include <stdio.h>
   main()
   {
  alarm(3);  /* Schedule an alarm signal in three seconds */
      printf("Looping forever… \n");
      while(1)
         printf("This line should never be executed \n");
   }
```

$ alarm ---> run the program.

Looping forever…

Alarm clock ---> occurs three seconds later.

$ .

- **The pause function suspends the calling process until a signal is caught.**

```
#include <unistd.h>

int pause(void);
```

# abort Function

- The abort function causes abnormal program termination.

```
#include <stdlib.h>

void abort(void);
```

- The abort function sends the SIGABRT signal to the caller. Processes should not ignore this signal.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
   FILE *fp;
   printf("Going to open nofile.txt\n");
   fp = fopen( "nofile.txt","r" );
   if(fp == NULL)
   {
      printf("Going to abort the program\n");
      abort();
   }
   printf("Going to close nofile.txt\n");
   fclose(fp);
   return(0);
}
```

# Sleep function

- sleep for the specified number of seconds.

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

## SLEEP COMMAND SUPPORTS BELOW UNITS

- s for seconds; this is a default one if you don’t specify any letter after the integer.
- m for minutes.
- h for hours.
- d for days.

- Delay execution of a command by one hour or even for a day.

  **sleep 1h**

  **sleep 1d**

# BASIC UTILITIES

1. who
2. date
3. tty
4. stty
5. bc
6. cal
7. man
8. lpr

# BASIC UTILITIES

9. passwd
10. clear
11. uname
12. echo
13. script

# 1. who

- The who command displays all users currently logged into the system.
- **Example :** $who

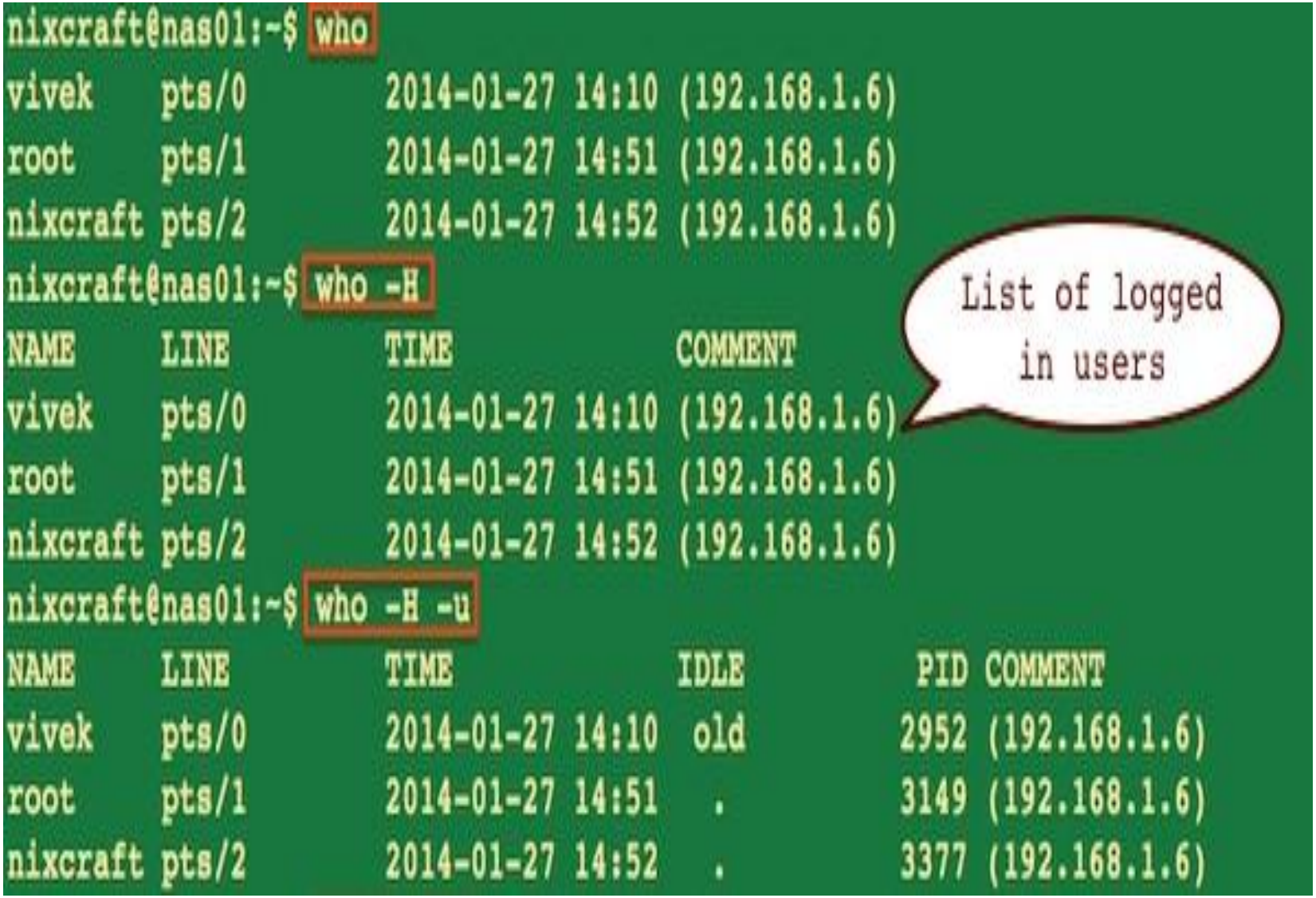
**Option:-**

- -u:- Just knowing that someone is logged in is not sufficient, however. You also want to know that he or she is active (or) not. This is know as idle time.
- **Example:-** $ who -u

- -uH:- Another helpful option, especially for new UNIX users, is the header.
- Example:- $ who –uH

## who am i

- If you key who am i  as the command, the system return your user id.
- Example:- $who am i


nixcraft@nas01:~$ who
vivek pts/0 2014-01-27 14:10 (192.168.1.6)
root pts/1 2014-01-27 14:51 (192.168.1.6)
nixcraft pts/2 2014-01-27 14:52 (192.168.1.6)
nixcraft@nas01:~$ who -H
NAME LINE TIME COMMENT
vivek pts/0 2014-01-27 14:10 (192.168.1.6)
root pts/1 2014-01-27 14:51 (192.168.1.6)
nixcraft pts/2 2014-01-27 14:52 (192.168.1.6)
nixcraft@nas01:~$ who -H -u
NAME LINE TIME IDLE PID COMMENT
vivek pts/0 2014-01-27 14:10 old 2952 (192.168.1.6)
root pts/1 2014-01-27 14:51 . 3149 (192.168.1.6)
nixcraft pts/2 2014-01-27 14:52 . 3377 (192.168.1.6)
List of logged in users

# 2. Date

- date command is used to display the system date and time.
- Date command is also used to set date and time, but only by a system administrator.

- **Syntax:-**

  date [OPTION] … [+FORMAT]

**Options with Examples:-**

- **date (no option):-** With no options, the date command displays the current date and time.

  **Example:-** $ date

- **-u:-** Displays the time in GMT (Greenwich Mean Time) / UTC(Coordinated Universal Time ).
- **Example:-** $ date –u

**Command:**

```
$date
```

**Output:**

```
Tue Oct 10 22:55:01 PDT 2017
```

**Command:**

```
$date -u
```

**Output :**

```
Wed Oct 11 06:11:31 UTC 2017
```

# List of Format specifies used with date command:

TABLE 1.1 **date** *Arguments*

| Format Code | Explanation |
|---|---|
| a | abbreviated weekday name, such as Mon |
| A | full weekday name, such as Monday |
| b | abbreviated month name, such as Jan |
| B | full month name, such as January |
| d | day of the month with two digits (leading zeros), such as 01, 02, . . . , 31 |
| e | day of the month with spaces replacing leading zeros, such as 1, 2, . . . , 31 |

Continued

TABLE 1.1 **date** *Arguments*

| Format Code | Explanation |
|---|---|
| D | date in the format mm/dd/yy, such as 01/01/99 |
| H | military time two-digit hour, such as 00, 01, . . . , 23 |
| I | civilian time two-digit hour, such as 00, 01, . . . , 12 |
| j | Julian date (Day of the year), such as 001, 002, . . . , 366 |
| m | numeric two-digit month, such as 01, 02, . . . , 12 |
| M | two-digit minute, such as 00, 01, . . . , 59 |
| n | newline character (used to display date on multiple lines) |
| p | display am or pm |
| r | time in format hour:minute:second with am/pm, such as 01:15:33 pm |
| R | time in format hour:minute, such as 13:15 |
| S | seconds as a decimal number [00–61], allows for leap seconds |
| t | tab character |
| T | time in format hour:minute:second, such as 13:15:48 |
| U | week number of year, such as 00, 01, . . . , 53 |
| W | week of year [00–53] with Monday being first day of week; all days preceding the first Sunday of the year are in week 0 |
| y | year within century (offset from %C) as a decimal number [00–99] |
| Y | year as ccyy (4 digits) |
| Z | time zone name, or no characters if no time zone is determinable |

**Syntax:-**

- $date “+Today’s date is : %D . The time is : %T”
- For the date command, the format is a plus sign (+) followed by the text and a series of format codes all enclosed in double quote marks.
- Each code is preceded by a percentage sign (%) that identifies it as a code.

**Output:-**

- Today’s date is :06/12/19.The time is: 08:29:30

# 3. tty

- In UNIX, everything is a file. Even any hardware device connected to the system is represented as a special file. So that a terminal is also represented as a file.
- tty is short for teletype, but it’s more popularly known as terminal.
- The tty command basically prints the filename of the terminal connected to standard input.

**Example:-**

- $tty

  /dev/ttyq0
- The output shows that the name of the terminal is /dev/ttyq0 (or) more simply, ttyq0.
- In UNIX, the name of a terminal usually has the prefix tty.

# 4. Set Terminal (stty) Command

- The set terminal command sets or unsets selected terminal input/output options.
- When the terminal is not responding properly, the set terminal command can be used to reconfigure it.

Syntax:- $ stty

- If we use the stty without any options or arguments, it shows the current common setting for your terminal.

## Set terminal with options:-

- The set terminal command can be used with two options (-a and -g).

- With the –a option, it displays the current terminal option settings.

- With the –g option, it displays selected settings in a format that can be used as an argument to another set terminal command.

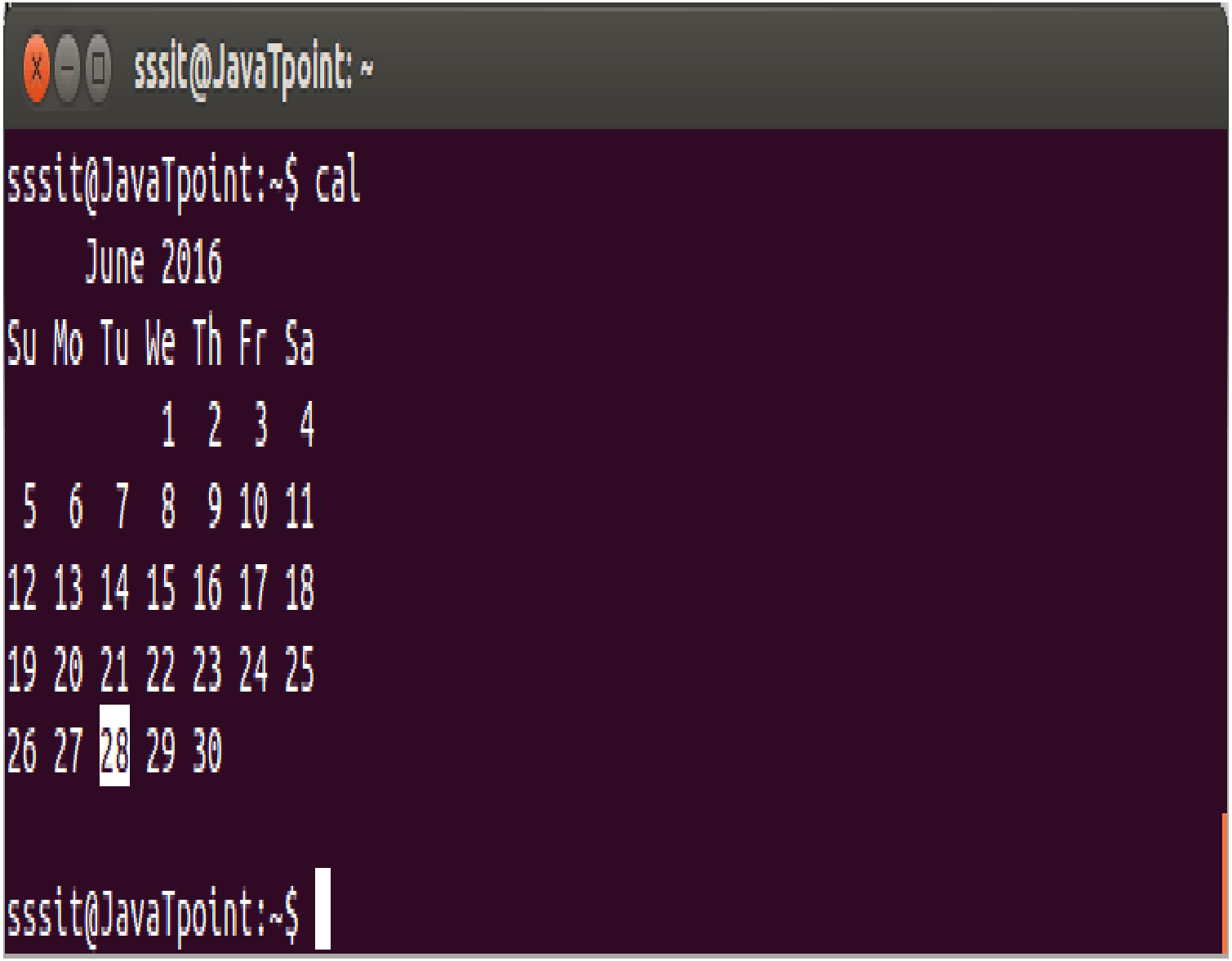
# 5. bc command

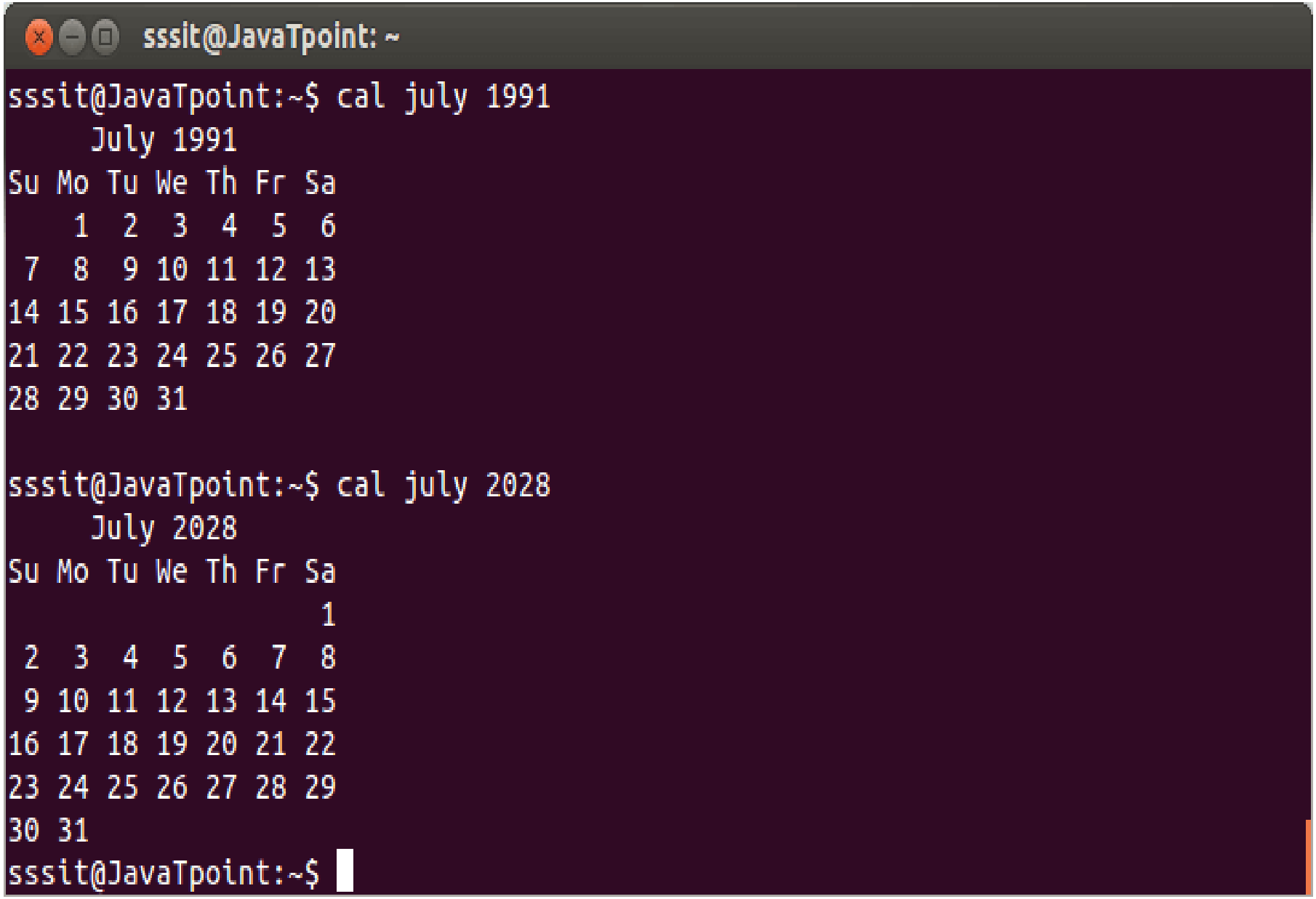
- The bc command is one of the most interesting commands in UNIX.
- It turns UNIX into a calculator.
- Syntax :- echo “134+18” |bc

# 6. Calendar (cal) command

- The calendar command, cal, displays the calendar for a specified month or for a year.
- It is an example of a command that has no options but uses arguments.
- Its general format is

  cal option [[month] year]

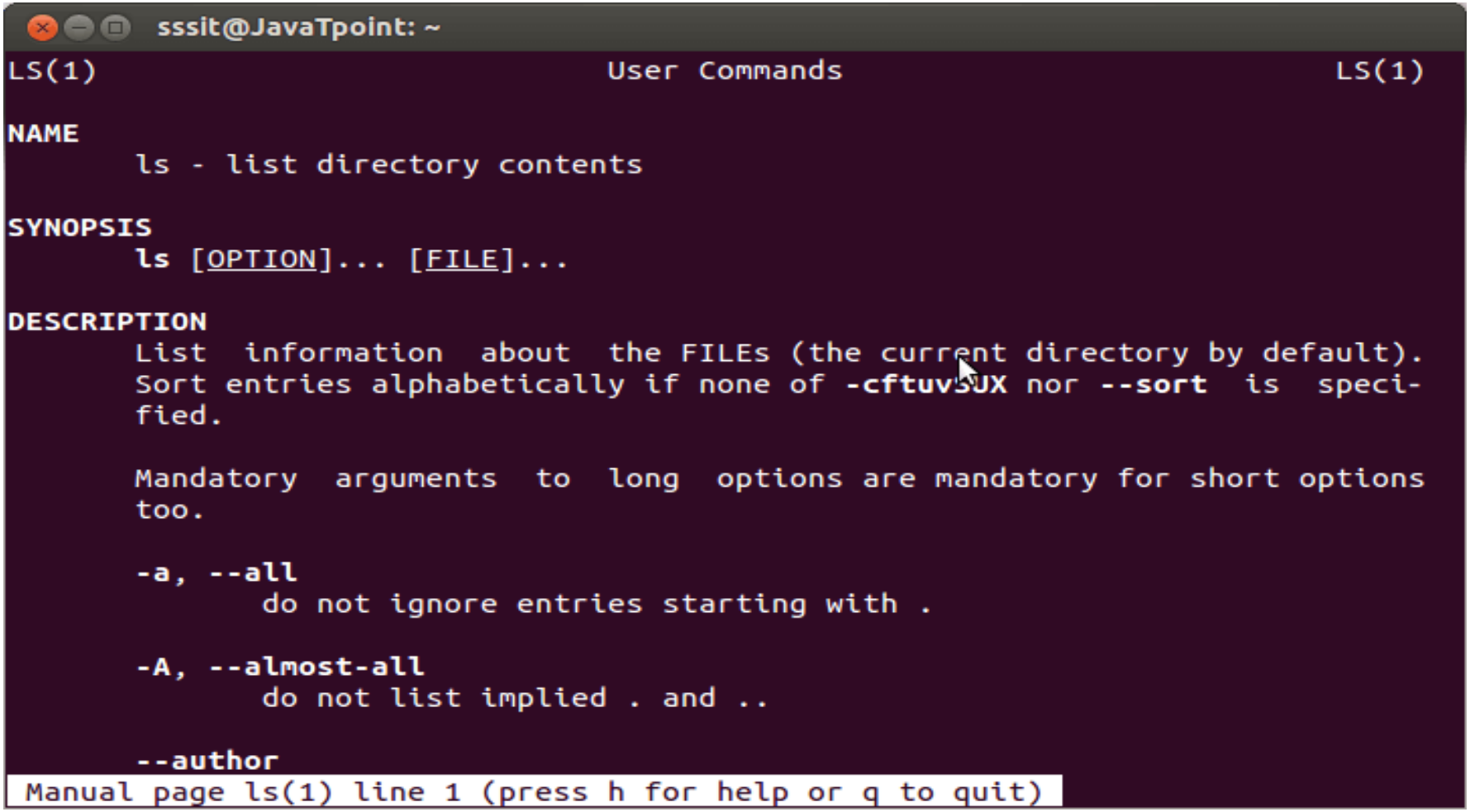

sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ cal
June 2016
Su Mo Tu We Th Fr Sa
1 2 3 4
5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
sssit@JavaTpoint:~$


sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ cal july 1991
July 1991
Su Mo Tu We Th Fr Sa
1 2 3 4 5 6
7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
sssit@JavaTpoint:~$ cal july 2028
July 2028
Su Mo Tu We Th Fr Sa
1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
sssit@JavaTpoint:~$

# 7. man command

- One of the most important UNIX commands is man.
- The man command displays online documentation. When you can't remember exactly what the options are for a command, you can quickly check the online manual and look up the answer.

# $ man ls


sssit@JavaTpoint: ~
LS(1) User Commands LS(1)
NAME
ls - list directory contents
SYNOPSIS
ls [OPTION]... [FILE]...
DESCRIPTION
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is speci-
fied.
Mandatory arguments to long options are mandatory for short options
too.
-a, --all
do not ignore entries starting with .
-A, --almost-all
do not list implied . and ..
--author
Manual page ls(1) line 1 (press h for help or q to quit)

- The man command with an option of –k and it will display information, including commands, about the topic.

  $ man –k sort
- If you want to know what UNIX sort utilities are available, you can enter the command and get a list of sort utilities.

# 8. lpr command

- The most common print utility is line printer (lpr).
- The line printer utility prints the contents of the specified files to either the default printer or to a specified printer.
- Multiple files can be printed with the same command.

Example:-

- $ lpr file1
- The command prints one file to the standard printer.

- $ lpr file1 file2 file3
- The command prints three files to the standard printer.

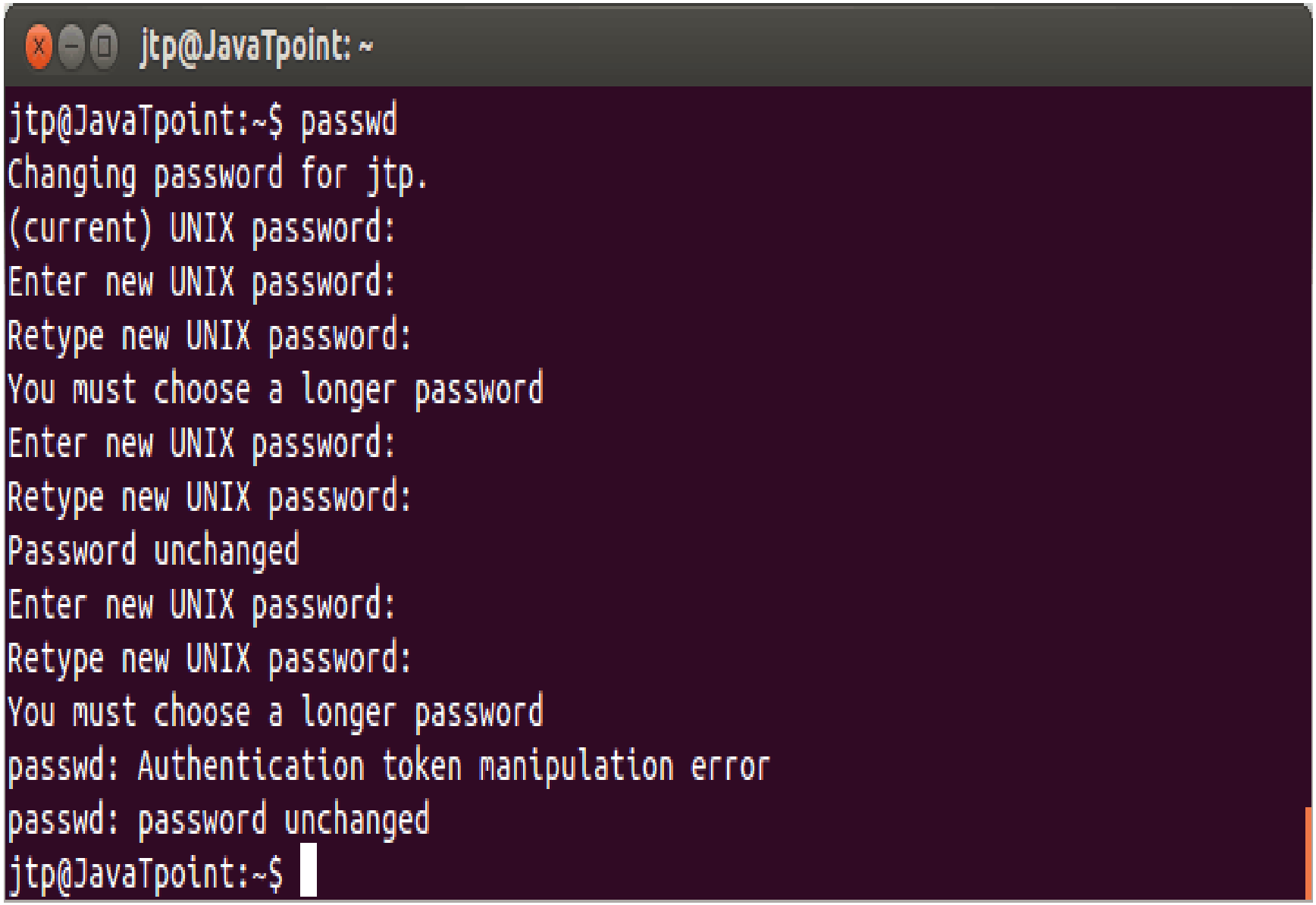
- To direct the output to a specified printer, we use the –P option. The name of the printer immediately follows the option with no spaces.

- $ lpr –Plp0 file1 file2 file3

- The command prints three files to printer lp0.

# 9. Change Password (passwd)

- The password command, passwd, is used to change your password.
- It has no options or attributes but rather does its work through a dialog of questions and answers.
- $ passwd


jtp@JavaTpoint: ~
jtp@JavaTpoint:~$ passwd
Changing password for jtp.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
You must choose a longer password
Enter new UNIX password:
Retype new UNIX password:
Password unchanged
Enter new UNIX password:
Retype new UNIX password:
You must choose a longer password
passwd: Authentication token manipulation error
passwd: password unchanged
jtp@JavaTpoint:~$

# 10. Clear Screen (clear)

- The clear command clears the screen and puts the cursor at the top.
- It is available in  most systems.

Syntax:-

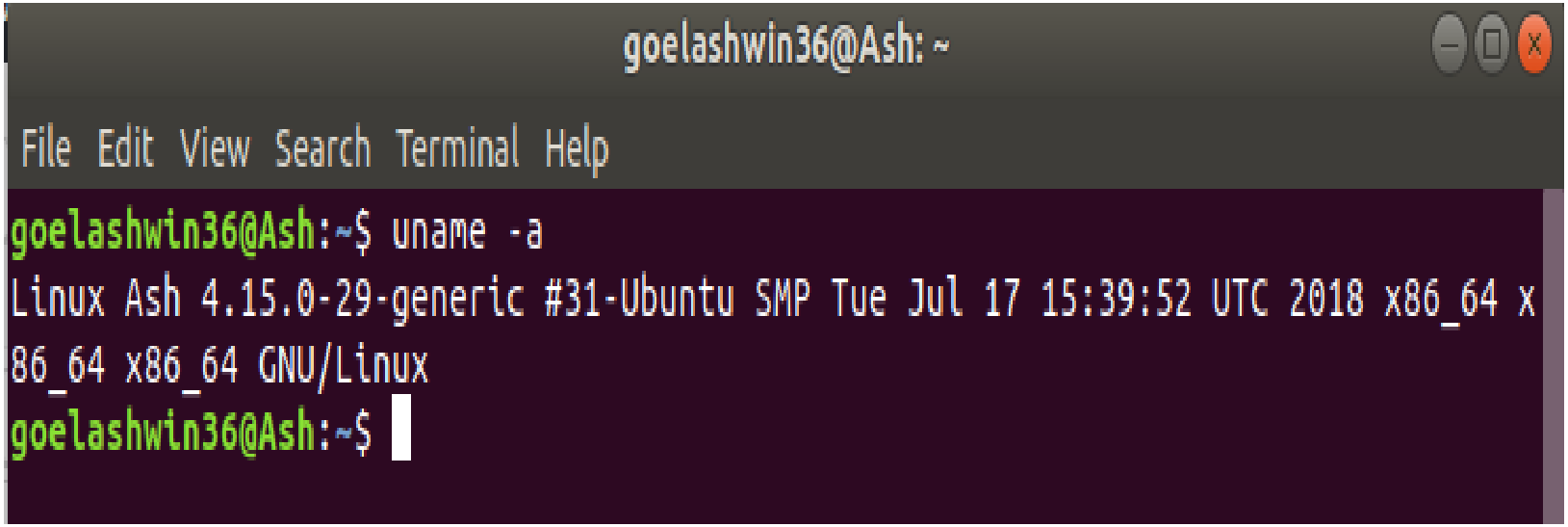
- $ clear

# 11. System Name (uname)

- Each UNIX system stores data, such as its name about itself. To see these data, we use the uname command.

  Syntax: -
- $uname

- We can display all of the data using the all option (-a)
- **Syntax:**
- $uname -a
- **Output:**

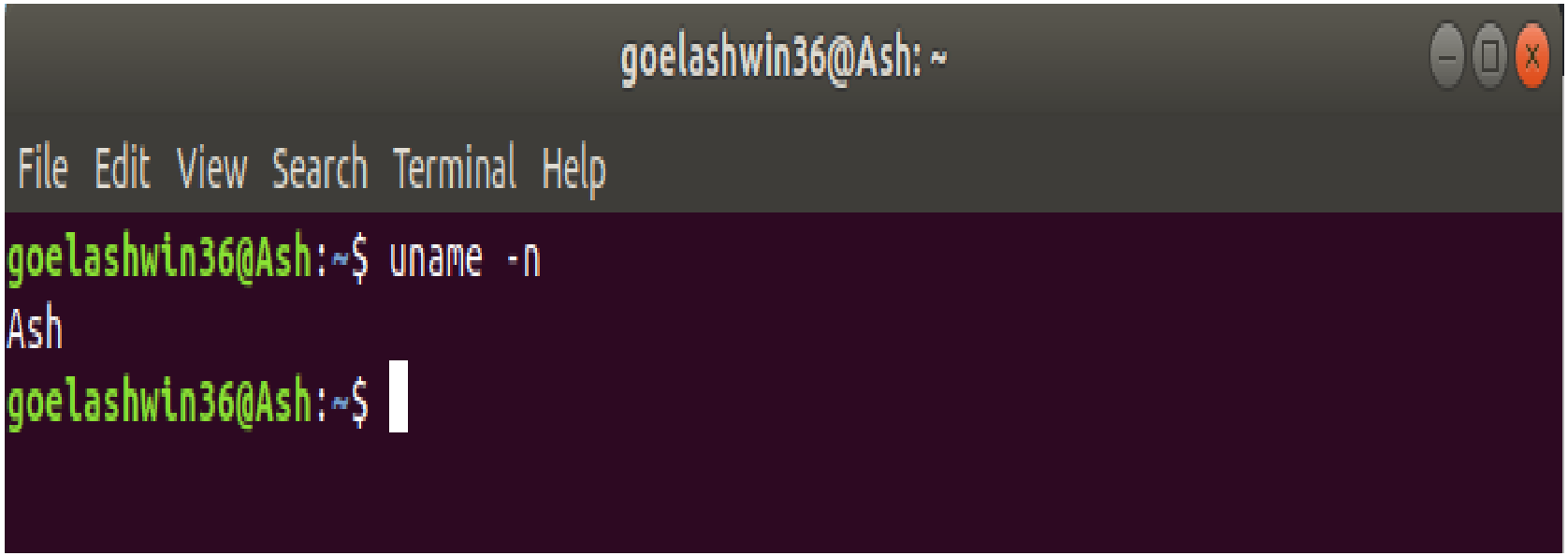


- We can specify only the name (-n).

  **Syntax:**

  $uname -n

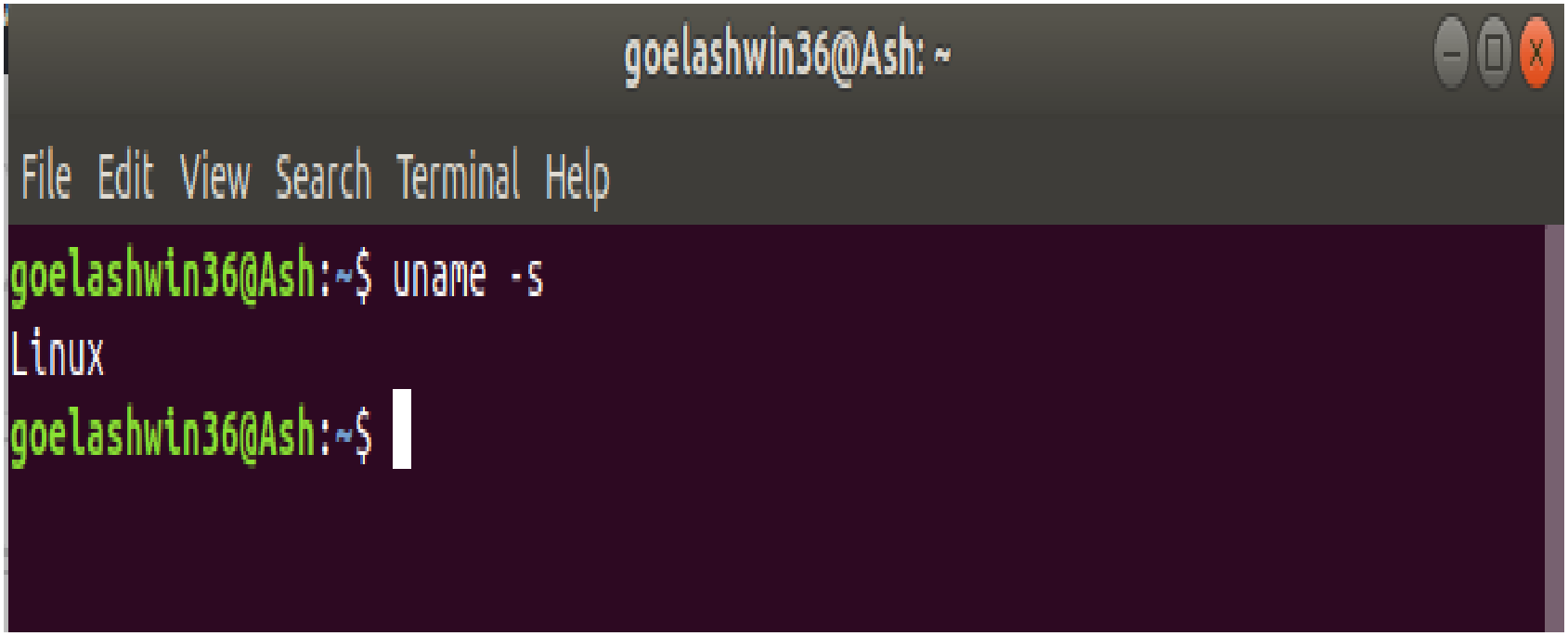
  **Output:**



- **-s option**: It prints the kernel name.

  **Syntax:**

- $uname -s
- **Output:**

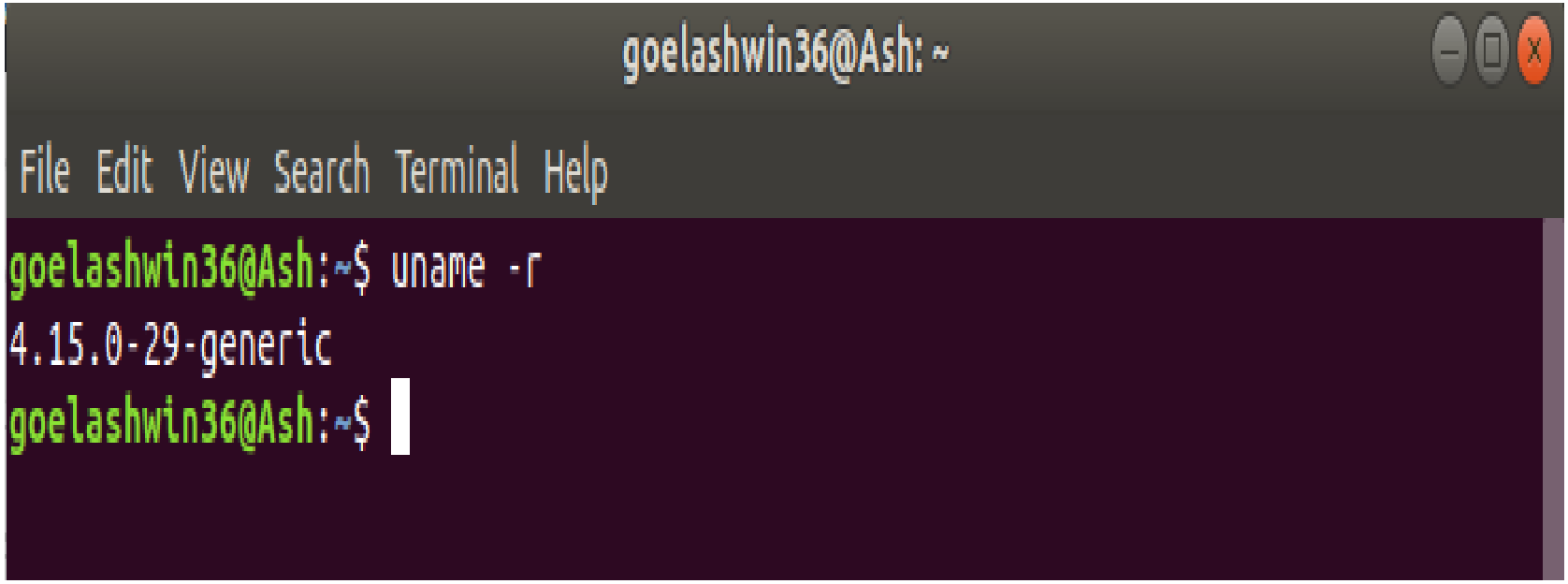


- **-r option**: It prints the kernel release date.

  **Syntax:**

  **$uname –r**

  **Output:**



- Options can be combines, for example, to display the operating system and its release, use –sr

# 12. echo

- **echo** command is used to display line of text/string that are passed as an argument .
- $echo Welcome to UNIX

# 13. Script command

- The script command can be used to record an interactive session.
- When you want to start recording, key the command.
- To stop the recording, key exit.
- You may have to use ctrl + d to log out after the exit command.

**Example:-**

- $script myfilename
- To append to the file rather than erase it, we use the append option (-a)

**Example:-**

- $script –a filename

```
idelab63@idelab63:~$ script welcome
Script started, file is welcome
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

idelab63@idelab63:~$ ls
Desktop  Downloads     Music  Public  typescript  welcome
Documents  examples.desktop  Pictures  Templates  Videos

idelab63@idelab63:~$ cal
   February 2016
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29

idelab63@idelab63:~$ echo "1097+543" | bc
1640

idelab63@idelab63:~$ exit
exit
Script done, file is welcome
```

```
idelab63@idelab63:~$ cat welcome
Script started on Thursday 11 February 2016 10:00:14 PM IST
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

idelab63@idelab63:~$ ls
Desktop   Downloads      Music   Public   typescript  welcome
Documents examples.desktop Pictures Templates Videos
idelab63@idelab63:~$ cal
   February 2016
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29
```

```
idelab63@idelab63:~$ echo "1097+543" | bc
1640
idelab63@idelab63:~$ exit
exit

Script done on Thursday 11 February 2016 10:00:55 PM IST
```

# FILE HANDLING SYSTEM CALLS USING STANDARD I/O

- File is a collection of numbers, symbols and text placed onto the disk.

- Thus, files allow us to store information permanently on to the disk and then access then when needed.

**File types:-**

- There are two types of files.
- Sequential file
- Random access file.

Functions that are available in standard library:

| S.No | Function | Operation |
|---|---|---|
| 1 | fopen() | Creates a new file for read/write operation. |
| 2 | fclose() | Closes a file associated with file pointer. |
| 3 | closeall() | Closes all files opened with fopen(). |
| 4 | fgetc() | Reads the character from current pointer position and advances the pointer to the next character. |
| 5 | fprintf() | Writes all types of data values to the file. |
| 6 | fscanf() | Reads all types of data values from a file. |
| 7 | fputc() | Writes characters one by one to a file. |
| 8 | putw() | Writes an integer to the file. |
| 9 | getw() | Reads an integer from the file. |
| 10 | fread() | Reads structured data written by fwrite(). |
| 11 | fwrite() | Writes block of structured data to the file. |
| 12 | fseek() | Sets the pointer position anywhere in the file. |
| 13 | feof() | Detects the end of file. |
| 14 | ferror() | Reports error occurred while read/write operations. |
| 15 | perror() | Prints compilers error messages along with user defined messages. |
| 16 | ftell() | Returns the current pointer position. |
| 17 | rewind() | Sets the record pointer at the beginning of the file. |
| 18 | unlink() | Removes the specified file from the disk. |
| 19 | rename() | Changes the name of the file. |

- **Opening of file: -**

  **Syntax:-**
- FILE *fp;
- fp=fopen(“data.txt”,”r”);

- It is necessary to write FILE in the uppercase. The function fopen() will open a file “data.txt” in read mode.

- **Reading a file:-**
- Once the file is opened using fopen(), its contents are loaded into the memory(partly or wholly).

- The pointer points to the very first character of the file.

- The fgetc() is used to read the contents of the file.

- The syntax for fgetc() is

  ch=fgetc(fp);

- where fgetc() reads the character from current pointer position and advances the pointer position so that the next character is pointed.

- **Text Modes:-**

- r ---> opens text file for reading only.
- w ---> opens a text file for writing only.
- a ---> opens text file for appending only.
- r+ ---> opens text file for reading and writing.
- w+ ---> opens text file for reading and writing.
- a+ ---> opens a text file for read or write

- **w(write):-** This mode opens a new file on the disk for writing. If the file already exists, it will be overwritten without confirmation.

  **Syntax:-**

  **fp=fopen(“data.txt”,”w”);**

- Here, data.txt is the file name and “w” is the mode.

- **r(read):-**This mode searches a file and if it is found the same is loaded into the memory for reading from the first character of the file.

- The file pointer points to the first character and reading operation begins.

- If the file doesn’t exist, then compiler returns NULL to the file pointer.

- Using pointer with if statement we can prompt the user regarding failure of operation.

- **Syntax:-**

```
fp=fopen(“data.txt”,”r”);
if(fp==NULL)
{
        printf(“File does not exist”);
}
        (OR)
if(fp=(fopen(“data.txt”,”r”))==NULL)
{
        printf(“File does not exist”);
}
```

- Here, data.txt is opened for reading only. If the file does not exist the fopen() returns NULL to file pointer ‘fp’.

- **append(a):-**This mode opens a pre existing file for appending data.
- The data appending process starts at the end of the opened file.
- The file pointer points to the last character of the file.
- If the file doesn’t exist, then new file is opened i.e., if the file does not exist then the mode of “a” is same as “w”.

- Due to some or other reasons if file is not opened in such a case NULL is returned.

- File opening may be impossible due to insufficient space on to the disk and some other reasons.

- **Syntax:-**

  **fp=fopen(“data.txt”,”a”);**

- Here, if data.txt file already exists, it will be opened. Otherwise a new file will be opened with the same name.

- **w+(write+read):-** This mode starts for file search operation on the disk.
- In case the file is found, its contents are destroyed.
- If the file is not found, a new file is created.

- It returns NULL if it fails to open the file. In this file mode new contents can be written and there after reading operation can be done.

**Syntax:-**

**fp=fopen(“data.txt”,”w+”);**

Here, data.txt file is open for reading and writing operation.

- **a+(append+read):-** In this file operation mode the contents of the file can be read and records can be added at the end of file.

- A new file is created in case the concerned file does not exist.

- Due to some or the other reasons if a file is unable to open then NULL is returned.

- **Syntax:-**
- **fp=fopen(“data.txt”,”a+”);**

- Here, data.txt is opened and records are added at the end of file without affecting the previous contents.

- **r+ (read + write):-** This mode is used for both reading and writing.

- We can read and write the record in the file. If the file does not exist, the compiler returns NULL to the file pointer.

- **Syntax:-**

```
fp=fopen(“data.txt”,”r+”);
if(fp==NULL)
    printf(“\n File not found”);
```

- Here, data.txt is opened for the read and write operation.

- If fopen() fails to open the file it returns NULL.

- The if statements check the value of file pointer fp; and if it contains NULL a message is printed and program terminates.

- **Closing a file:-**
- The file that is opened from the fopen() should be closed after the work is completed i.e., we need to close the file after reading and writing operations are completed.

- **Syntax:-**

```
fclose(file_pointer);
```

- To close one or more files at a time the function fcloseall() is used.

- **Syntax:-**

  fcloseall();

- **FILE I/O:-**
- After opening the file, the next thing needed is the way to read or write the file. These functions are classified as:-

- Character I/O functions.
- String I/O functions.
- Formatted I/O functions.
- Block I/O functions.

## Character I/O functions:-

- ‘C’ provides a set of functions for reading and writing character by character or one byte at a time.

- These functions are defined in the standard library.
  1. fgetc()
  2. fputc()

- **fgetc():-** fgetc() is used to read a character from a file.

- **Syntax:-**

  fgetc(FILE *stream);

- **fputc():-** This function is used to write a single character into a file. If an error occurs it returns EOF.

- **Syntax:-**
- **fputc(ch, FILE *stream);**

- **String I/O functions:-**

- If we want to read a whole line in the file then each time we will need to call character input function.
  - fgets()
  - fputs()

- **fgets():-** This function reads string from a file pointed by file pointer. It also copies the string to a memory location referred by an array.

- **Syntax:-**

  **fgets(str, size, FILE *stream);**

  Here, str is a name of a character array,

  size is an integer value.

- **fputs():-** This function is useful when we want to write a string into the opened file.

- **Syntax:-**
- **fputs(str,FILE *stream);**

- **Formatted I/O functions:-**
- If the file contains data in the form of digits, real numbers, character and strings, then character I/O functions are not enough as the values would be read in the form of characters.
  - fprintf()
  - fscanf()
- These functions are used for formatted input and output. These are identical to scanf() and printf().

- **fprintf():-** This function is used for writing characters, strings, integers, floats etc to the file.

- Hence this function is called the formatted function.

- It contains one more parameter that is file pointer, which points the opened file.

- **Syntax:-**
- **fprintf(fp, ”control string”, arguments list);**

- Here, the parameter fp associated with a file that has been opened for writing.

- A control string specifies the format specifiers.

- Argument list contains variables separated by commas.

- **fscanf():-** This function reads character, strings, integer, floats etc from the file pointed by file pointer.

- This is also a formatted function.

- **Syntax:-**
- **fscanf(fp, ”control string”, arguments list);**

- Here, the parameter fp associated with a file that has been opened for writing.

- A control string specifies the format specifiers.

- Argument list contains variables separated by commas.

- **Block I/O functions:- (Structure Read and Write)**
- Block I/O functions read/write a block. A block can be a record, a set of records or an array or a structure.
- These functions are also defined in standard library.
  - fread()
  - fwrite()
- These two functions allow reading and writing of block of data.

- **fread():-** This function is used for reading an entire block from a given file.

- **Syntax:-**
- **fread(&structure_variable, int size, int num,FILE *fp);**

- Here, **structure_variable** is the pointer or address of block of memory (structure).
- **size** is the size of the structure.
- **num** is the number of structure variables.
- **fp** is the pointer to the datatype **FILE**.

- **fwrite():-** This function is used for writing an entire structure block to a given file.

- **Syntax:-**

- **fwrite(&structure_variable, int size, int num,FILE *fp);**

- Here, **structure_variable** is the pointer or address of block of memory(structure).
- **size** is the size of the structure.
- **num** is the number of structure variables.
- **fp** is the pointer to the datatype **FILE**.

- **Random access functions:-**
- Sequential access files allow reading the data from the file in sequential manner which means that data can only be read in sequence.
- Random access files allow reading data from any location in the file. The functions are
- fseek()
- ftell()
- rewind()

- **fseek():-**

  fseek() is used to move the file position to a desired location within the file.
- **Syntax:-**

  **fseek(file_ptr,offset,position);**

  where
- file_ptr is a pointer to the file.
- offset is a number or variable of type long
- position is an integer number.

- The offset specifies the number of positions to be moved from the location specified by position. The position can take one of the following three values

| Values | Meaning |
|---|---|
| 0 | Beginning of file |
| 1 | Current position |
| 2 | End of file. |

- **ftell():-**
- ftell() takes a file pointer and returns a number of type long, that corresponds to the current position. This function is useful in saving the current position of a file, which can be used later in the program.
- **Syntax:-**
- **n=ftell(fp);**
- where n would given the relative offset of the current position.

- **rewind():-**
- rewind takes a file pointer and resets the position to the start of the file.
- **Syntax:-**
- **rewind(fp);**

- **fflush()**

- The C library function

  **int fflush(FILE *stream)** flushes the output buffer of a stream.

- **Declaration: -**

  Following is the declaration for fflush() function.

  int fflush(FILE *stream)

- **Parameters: -**

  **stream** − This is the pointer to a FILE object that specifies a buffered stream.

- **Return Value: -**
- This function returns a zero value on success. If an error occurs, EOF is returned and the error indicator is set (i.e. feof).

# Example: A Simple getchar()

```
int getchar(void) {
    static char c;
    if (read(0, &c, 1) == 1)
        return c;
    else return EOF;
}
```

- Read one character from `stdin`
  - File descriptor `0` is `stdin`
  - `&c` points to the buffer
  - `1` is the number of bytes to read
- Read returns the number of bytes read
  - In this case, `1` byte means success

# Making getchar() More Efficient

- Poor performance reading one byte at a time
  - Read system call is accessing the device (e.g., a disk)
  - Reading one byte from disk is very time consuming
  - Better to read and write in *larger chunks*

# Starting new process

- When you start a process (run a command), there are two ways you can run it –

  Foreground Processes

  Background Processes

**Foreground Processes**

- By default, every process that you start runs in the foreground.

- It gets its input from the keyboard and sends its output to the screen.

- You can see this happen with the **ls** command. If you wish to list all the files in your current directory, you can use the following command

$ls ch*.doc

- This would display all the files, the names of which start with **ch** and end with **.doc** –

ch01-1.doc

ch010.doc

ch02.doc

ch03-2.doc

ch04-1.doc

ch040.doc

- The process runs in the foreground, the output is directed to my screen, and if the **ls** command wants any input (which it does not), it waits for it from the keyboard.

- While a program is running in the foreground and is time-consuming, no other commands can be run (start any other processes) because the prompt would not be available until the program finishes processing and comes out.

- **Background Processes**
- A background process runs without being connected to your keyboard.

- If the background process requires any keyboard input, it waits.

- The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

- The simplest way to start a background process is to add an ampersand (**&**) at the end of the command.

- $ls ch*.doc &

- This displays all those files the names of which start with **ch** and end with **.doc** −

ch01-1.doc

ch010.doc

ch02.doc

ch03-2.doc

ch04-1.doc

ch040.doc

- Here, if the **ls** command wants any input (which it does not), it goes into a stop state until we move it into the foreground and give it the data from the keyboard.

- That first line contains information about the background process - the job number and the process ID.
- You need to know the job number to manipulate it between the background and the foreground.

- [1] + Done ls ch*.doc &
- $

- The first line tells you that the **ls** command background process finishes successfully. The second is a prompt for another command.

- **Listing Running Processes**
- It is easy to see your own processes by running the **ps** (process status) command as follows −

- $ps

| PID | TTY | TIME | CMD |
|---|---|---|---|
| 18358 | ttyp3 | 00:00:00 | sh |
| 18361 | ttyp3 | 00:01:31 | abiword |
| 18789 | ttyp3 | 00:00:00 | ps |

- One of the most commonly used flags for ps is the **-f** ( f for full) option, which provides more information

```
$ps -f
UID        PID  PPID  C    STIME    TTY      TIME  CMD
amrood    6738  3662  0  10:23:03  pts/6    0:00  first_one
amrood    6739  3662  0  10:22:54  pts/6    0:00  second_one
amrood    3662  3657  0  08:10:53  pts/6    0:00  -ksh
amrood    6892  3662  4  10:51:50  pts/6    0:00  ps -f
```

| S.No. | Column & Description |
|---|---|
| 1 | **UID**<br>User ID that this process belongs to (the person running it) |
| 2 | **PID**<br>Process ID |
| 3 | **PPID**<br>Parent process ID (the ID of the process that started it) |
| 4 | **C**<br>CPU utilization of process |
| 5 | **STIME**<br>Process start time |
| 6 | **TTY**<br>Terminal type associated with the process |
| 7 | **TIME**<br>CPU time taken by the process |
| 8 | **CMD** |

- There are other options which can be used along with **ps** command

| S.No. | Option & Description |
|---|---|
| 1 | **-a**<br>Shows information about all users |
| 2 | **-x**<br>Shows information about processes without terminals |
| 3 | **-u**<br>Shows additional information like -f option |
| 4 | **-e**<br>Displays extended information |

# Stopping Processes

- Ending a process can be done in several different ways.
- Often, from a console-based command, sending a CTRL + C keystroke will exit the command.
- This works when the process is running in the foreground mode.

- If a process is running in the background, you should get its Job ID using the **ps** command. After that, you can use the **kill** command to kill the process as follows.

- $ps -f

$ps -f

| UID | PID | PPID | C | STIME | TTY | TIME | CMD |
|---|---|---|---|---|---|---|---|
| amrood | 6738 | 3662 | 0 | 10:23:03 | pts/6 | 0:00 | first one |
| amrood | 6739 | 3662 | 0 | 10:22:54 | pts/6 | 0:00 | second one |
| amrood | 3662 | 3657 | 0 | 08:10:53 | pts/6 | 0:00 | -ksh |
| amrood | 6892 | 3662 | 4 | 10:51:50 | pts/6 | 0:00 | ps -f |

- $kill 6738

  Terminated
- Here, the **kill** command terminates the **first_one** process.

- If a process ignores a regular kill command, you can use **kill -9** followed by the process ID as follows −

- $kill -9 6738Terminated

# Zombie and Orphan Processes

- Normally, when a child process is killed, the parent process is updated via a **SIGCHLD** signal.

- Then the parent can do some other task or restart a new child as needed.

- However, sometimes the parent process is killed before its child is killed.

- In this case, the "parent of all processes," the **init** process, becomes the new PPID (parent process ID).

- In some cases, these processes are called **orphan processes.**

- When a process is killed, a **ps** listing may still show the process with a **Z** state.
- This is a **zombie process**. The process is dead and not being used.
- These processes are different from the orphan processes.
- They have completed execution but still find an entry in the process table.

- ***Daemon Processes***

- Daemons are system-related background processes that often run with the permissions of root and services requests from other processes.

- A daemon is a process that runs in the background, usually waiting for something to happen that it is capable of working with.

**The top Command: -**

**top** is a basic Unix command which is very useful for observing the current state of your Unix system, by default presenting you the list of top users of your system's resources – CPU shares and memory.

- Here is the simple syntax to run top command and to see the statistics of CPU utilization by different processes −
- $top

```
ubuntu$ top
top - 13:29:09 up 2 days,  7:13,  4 users,  load average:
0.07, 0.02, 0.00
Tasks: 148 total,   1 running, 147 sleeping,   0 stopped,
0 zombie
Cpu(s):  0.6%us,  0.5%sy,  0.0%ni, 97.3%id,  1.6%wa,
0.0%hi,  0.0%si,  0.0%st
Mem:   4051792k total,  4026104k used,    25688k free,
359168k buffers
Swap:  4096492k total,    24296k used,  4072196k free,
2806484k cached
```

## Job ID Versus Process ID

- Background and suspended processes are usually manipulated via **job number (job ID)**.

- This number is different from the process ID and is used because it is shorter.

- In addition, a job can consist of multiple processes running in a series or at the same time, in parallel.

- Using the job ID is easier than tracking individual processes.

- **Types of Processes**
- Parent and Child process
- Zombie and Orphan process
- Daemon process

# Process Control

**Process Identifiers**

- Every process has a unique process ID, a non-negative integer.
- As processes terminate, their IDs can be reused.

- There are some special processes, but the details differ from implementation to implementation:

- Process ID 0: scheduler process (often known as the **swapper**), which is part of the kernel and is known as a system process

- Process ID 1: init process, invoked by the kernel at the end of the bootstrap procedure.

```
include <unistd.h>

pid_t getpid(void);
/* Returns: process ID of calling process */

pid_t getppid(void);
/* Returns: parent process ID of calling process */

uid_t getuid(void);
/* Returns: real user ID of calling process */

uid_t geteuid(void);
/* Returns: effective user ID of calling process */

gid_t getgid(void);
/* Returns: real group ID of calling process */

gid_t getegid(void);
/* Returns: effective group ID of calling process */
```

# INTER PROCESS COMMUNICATION (IPC)

- Inter Process Communication (IPC) refers to a mechanism, where the operating systems allow various processes to communicate with each other.

- This involves synchronizing their actions and managing shared data.

- Inter Process Communication (IPC) is a mechanism that involves communication of one process with another process.
- This usually occurs only in one system.
- Communication can be of two types.

1. Between related processes initiating from only one process, such as parent and child processes.
2. Between unrelated processes, or two or more different processes.

- Following are some important terms that we need to know before proceeding further on this topic.

- **Pipes**: - Communication between two related processes. The mechanism is half duplex meaning the first process communicates with the second process. To achieve a full duplex i.e., for the second process to communicate with the first process another pipe is required.

- **FIFO**: - Communication between two unrelated processes. FIFO is a full duplex, meaning the first process can communicate with the second process and vice versa at the same time.

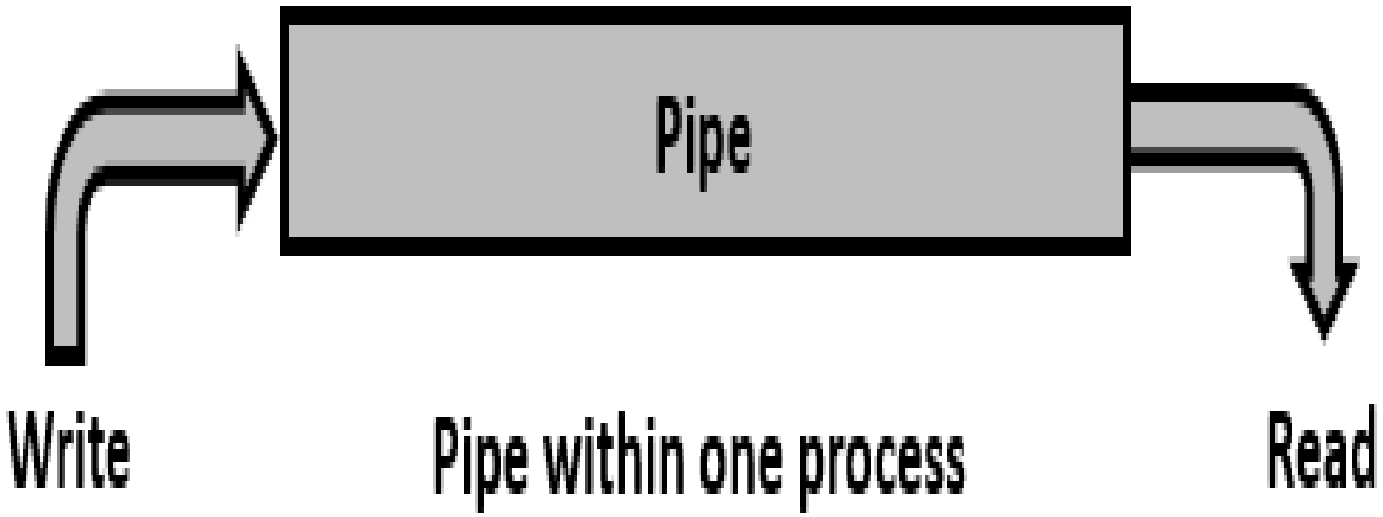
- **Message Queues**: - Communication between two or more processes with full duplex capacity. The processes will communicate with each other by posting a message and retrieving it out of the queue. Once retrieved, the message is no longer available in the queue.

- **Shared Memory**: - Communication between two or more processes is achieved through a shared piece of memory among all processes. The shared memory needs to be protected from each other by synchronizing access to all the processes.

- **Signals**: - Signal is a mechanism to communication between multiple processes by way of signalling. This means a source process will send a signal (recognized by number) and the destination process will handle it accordingly.

- **PIPES: -**
- Pipe is a communication medium between two or more related or interrelated processes.
- It can be either within one process or a communication between the child and the parent processes.
- Communication can also be multi-level such as communication between the parent, the child and the grand-child, etc.

- Communication is achieved by one process writing into the pipe and other reading from the pipe.
- To achieve the pipe system call, create two files, one to write into the file and another to read from the file.


Pipe
Write
Pipe within one process
Read

- **Syntax: -**

  #include<unistd.h>

  int pipe(int pipedes[2]);

- This system call would create a pipe for one-way communication i.e., it creates two descriptors, first one is connected to read from the pipe and other one is connected to write into the pipe.

- Descriptor pipedes[0] is for reading and pipedes[1] is for writing.

- Whatever is written into pipedes[1] can be read from pipedes[0].

- This call would return zero on success and -1 in case of failure.

- #include <sys/types.h>

  #include <sys/stat.h>

  #include <fcntl.h>

  int open(const char *pathname, int flags);

  int open(const char *pathname, int flags, mode_t mode);

- The arguments passed to open system call are pathname (relative or absolute path),
- flags mentioning the purpose of opening file (say,

  opening for read, O_RDONLY,

  to write, O_WRONLY,

  to read and write, O_RDWR,

  to append to the existing file O_APPEND,

  to create file, if not exists with O_CREAT and so on)

- The required mode providing permissions of read/write/execute for user or owner/group/others. Mode can be mentioned with symbols.
- Read – 4, Write – 2 and Execute – 1.

```
#include<unistd.h>
int close(int fd)
```

- The above system call closing already opened file descriptor.

- This system call returns zero on success and -1 in case of error.

```
#include<unistd.h>
ssize_t read(int fd, void *buf, size_t count)
```

- The above system call is to read from the specified file with arguments of file descriptor fd, proper buffer with allocated memory (either static or dynamic) and the size of buffer.

```
#include<unistd.h>
ssize_t write(int fd, void *buf, size_t count)
```

- The above system call is to write to the specified file with arguments of the file descriptor fd, a proper buffer with allocated memory (either static or dynamic) and the size of buffer.

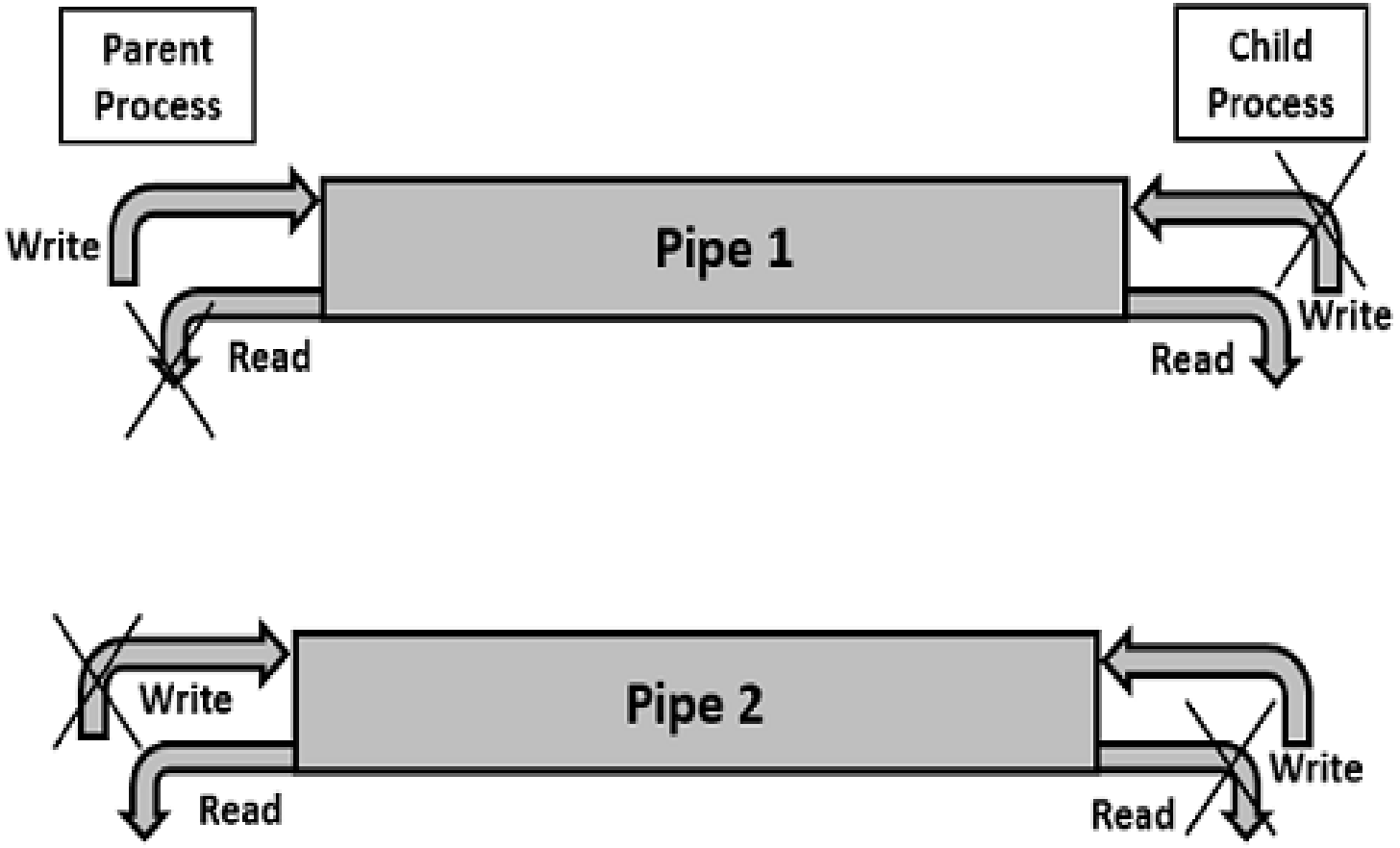
- **Two-way Communication Using Pipes**

- Pipe communication is viewed as only one-way communication i.e., either the parent process writes or the child process reads or vice-versa but not both.
- However, what if both the parent and the child need to write and read from the pipes simultaneously, the solution is a two-way communication using pipes.

- Two pipes are required to establish two-way communication.

- Following are the steps to achieve two-way communication –
- **Step 1** – Create two pipes. First one is for the parent to write and child to read, say as pipe1. Second one is for the child to write and parent to read, say as pipe2.
- **Step 2** – Create a child process.

- **Step 3** – Close unwanted ends as only one end is needed for each communication.
- **Step 4** – Close unwanted ends in the parent process, read end of pipe1 and write end of pipe2.
- **Step 5** – Close the unwanted ends in the child process, write end of pipe1 and read end of pipe2.
- **Step 6** – Perform the communication as required.


Parent Process
Child Process
Write
Pipe 1
Read
Read
Write
Write
Pipe 2
Read
Read
Write

- **FIFOs: -**
- Pipes were meant for communication between related processes.
- Can we use pipes for unrelated process communication, say, we want to execute client program from one terminal and the server program from another terminal? The answer is No.
- Then how can we achieve unrelated processes communication, the simple answer is Named Pipes.

- Even though this works for related processes, it gives no meaning to use the named pipes for related process communication.

- We used one pipe for one-way communication and two pipes for bi-directional communication. Does the same condition apply for Named Pipes.

- The answer is no, we can use single named pipe that can be used for two-way communication (communication between the server and the client, plus the client and the server at the same time) as Named Pipe supports bi-directional communication.

- Another name for named pipe is **FIFO (First-In-First-Out)**. Let us see the system call (mknod()) to create a named pipe, which is a kind of a special file.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int mknod(const char *pathname, mode_t mode, dev_t dev);
```

- The **pathname** along with the attributes of mode and device information. The pathname is relative, if the directory is not specified it would be created in the current directory.

- The **mode** specified is the mode of file which specifies the file type such as the type of file and the file mode as mentioned in the following tables.

| S.NO | FILE TYPE | DESCRIPTION |
|---|---|---|
| 1 | S_IFBLK | Block special |
| 2 | S_IFCHR | Character special |
| 3 | S_IFIFO | FIFO special |
| 4 | S_IFREG | Regular file |
| 5 | S_IFDIR | Directory |
| 6 | S_IFLNK | Symbolic Link |

- The **dev** field is to specify device information such as major and minor device numbers.

| S.NO | FILE MODE | DESCRIPTION |
|---|---|---|
| 1 | S_IRWXU | Read, write, execute/search by owner |
| 2 | S_IRUSR | Read permission, owner |
| 3 | S_IWUSR | Write permission, owner |
| 4 | S_IXUSR | Execute/search permission, owner |
| 5 | S_IRWXG | Read, write, execute/search by group |
| 6 | S_IRGRP | Read permission, group |
| 7 | S_IWGRP | Write permission, group |
| 8 | S_IXGRP | Execute/search permission, group |
| 9 | S_IRWXO | Read, write, execute/search by others |
| 10 | S_IROTH | Read permission, others |
| 11 | S_IWOTH | Write permission, others |

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode)
```

- This library function creates a FIFO special file, which is used for named pipe.
- The arguments to this function are file name and mode.

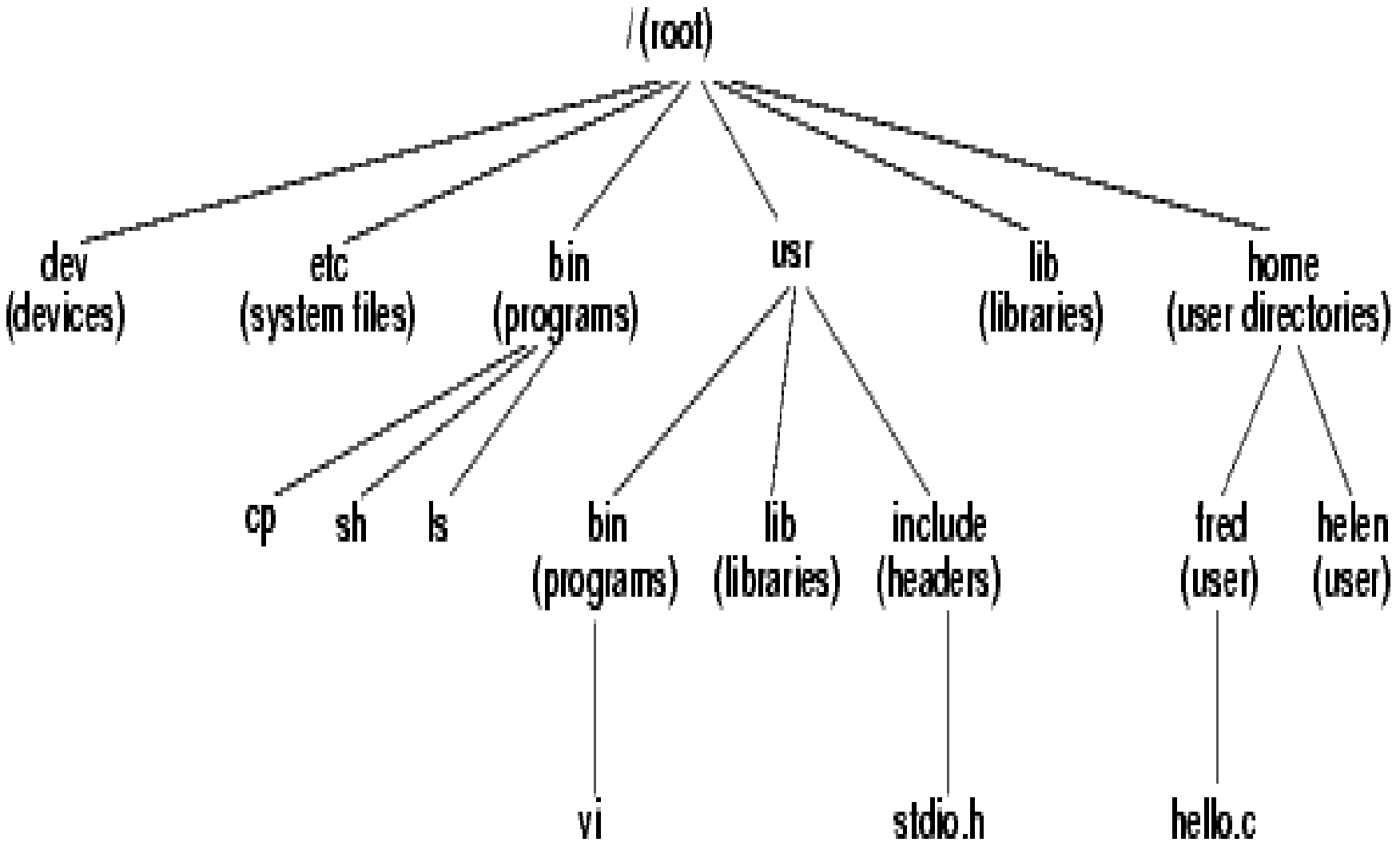
- The file name can be either absolute path or relative path.

- If full path name (or absolute path) is not given, the file would be created in the current folder of the executing process.

- The file mode information is as described in mknod() system call.

# Introduction to Unix File System

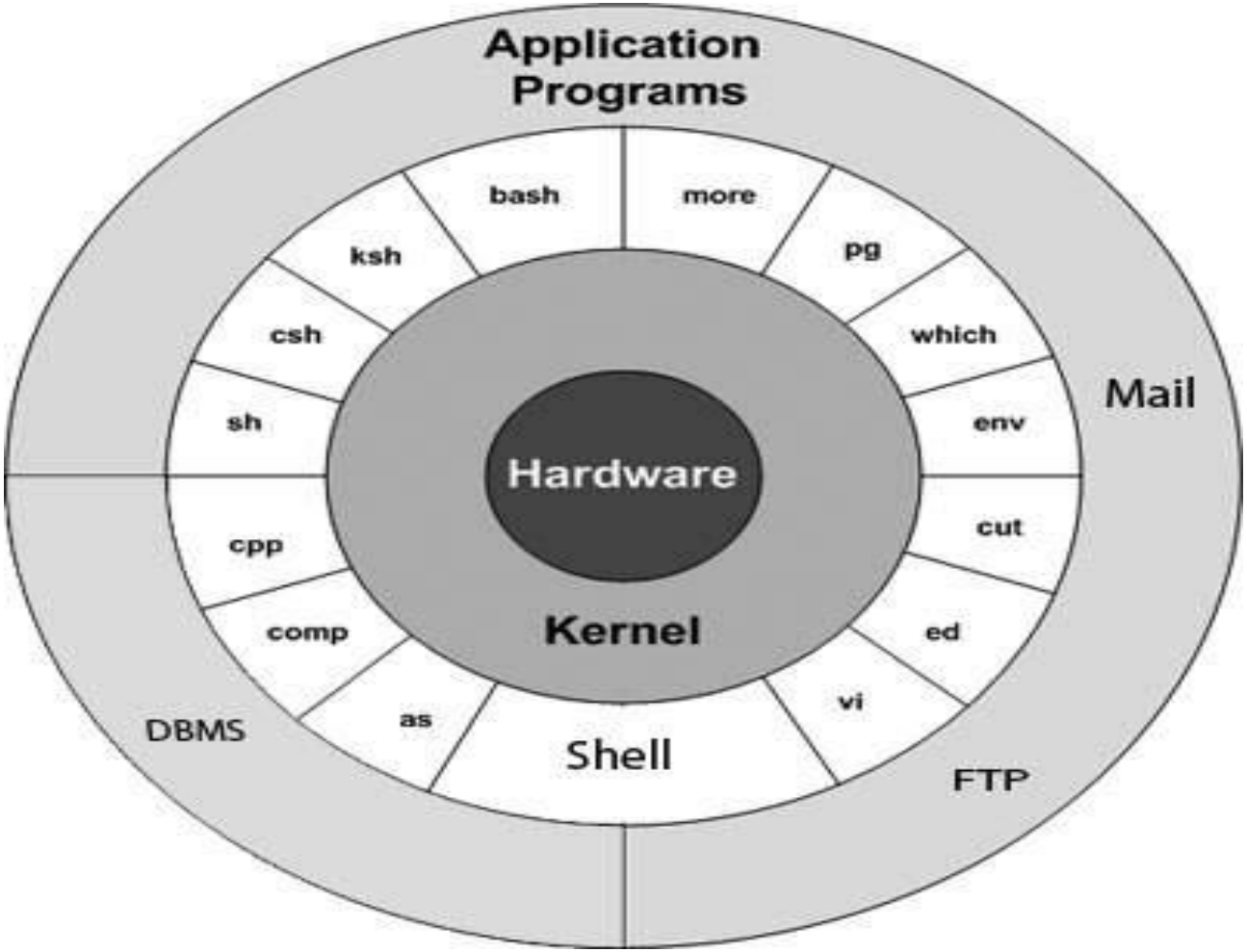
- Unix is an open Operating System.
- All data in Unix is organized into files. All files are organized into directories.
- These directories are organized into a tree like structure called the **FILE SYSTEM**.
- Files in Unix system are organized into multi level hierarchy structure known as a directory tree.


/ (root)
dev
(devices)
etc
(system files)
bin
(programs)
usr
lib
(libraries)
home
(user directories)
cp
sh
ls
bin
(programs)
lib
(libraries)
include
(headers)
fred
(user)
helen
(user)
vi
stdio.h
hello.c

# UNIX Features

- Portable.
- Multi users.
- Multi tasking.
- Networking.
- Device Independences.
- Organized File System.
- Utilities.
- Services.

# UNIX System Architecture



- **Hardware:-** The hardware includes all the parts of a computer including clocks, timers, devices, parts etc. in the UNIX OS architecture.

- **The Kernel:-** The kernel is the heart of the Unix system. The kernel is a part of the operating system. It interacts directly with the hardware of the computer through a device that is built into the kernel.

The main functions of the kernel are Memory Management, Controlling access to the computer, Maintaining the file system, Handling interrupts, Handling errors, Performing input and output services, Allocate the resources of the computer among users.

- **The Shell:-** Shell is the utility that processes your requests. when you type in a command at the terminal, shell interprets the command and calls the program that you want.

- There are various commands like cp, mv, cat, grep, id, wc, nroff, a.out and more.

- **Application Layer:-** It is the outermost layer that executes the given external applications.

# VI EDITOR

- The VI editor is a screen editor available on most UNIX systems.
- When you invoke the vi editor, it copies the contents of a file to a memory space known as a **buffer**.
- Once the data have been loaded into the buffer, the editor presents a screen full of the buffer to the user for editing.
- If the file does not exist, an empty buffer is created.

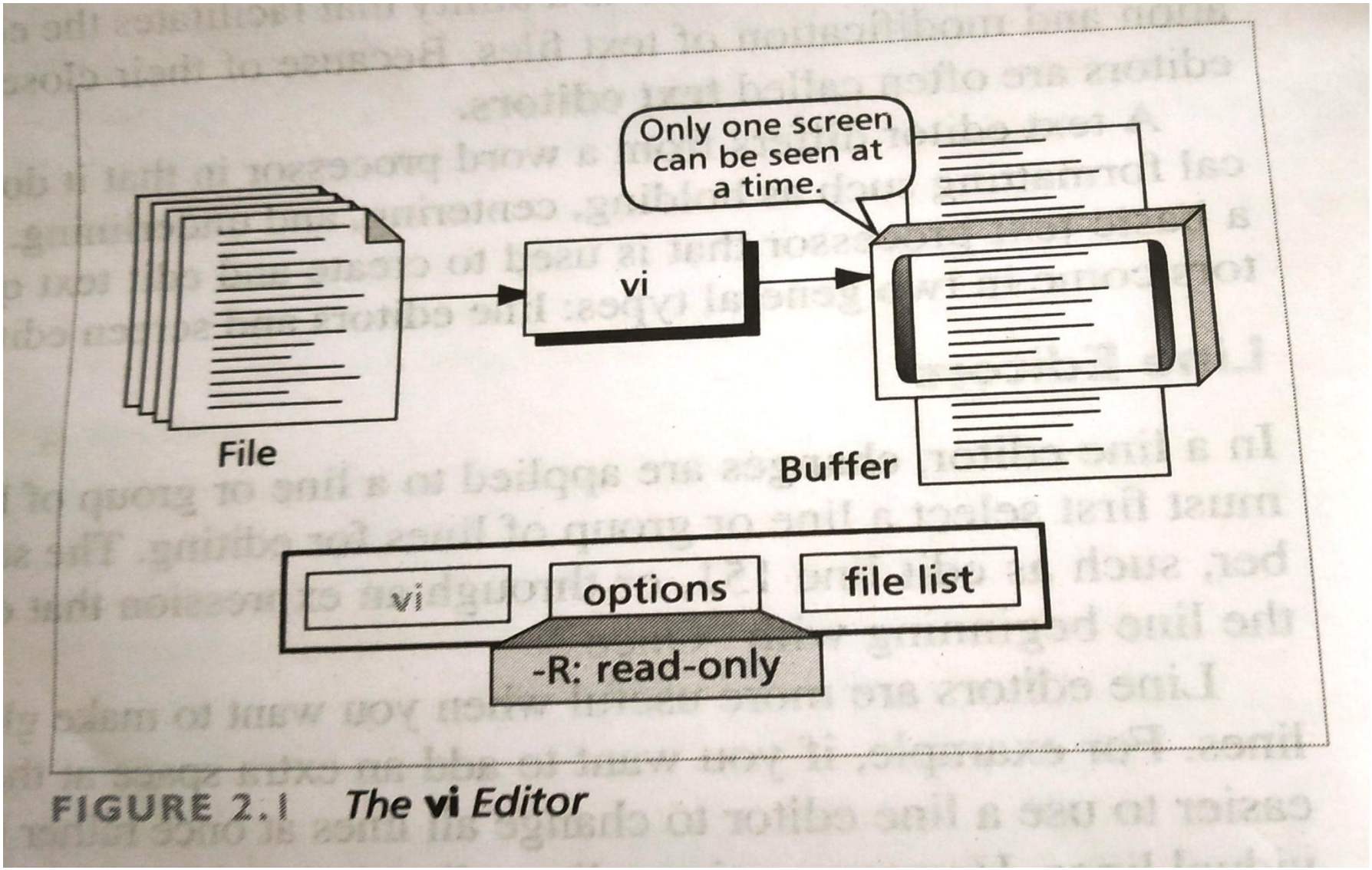



FIGURE 2.1 *The* **vi** *Editor*

- There are following way you can start using vi editor –

| Command | Description |
|---|---|
| **vi filename** | Creates a new file if it already does not exist, otherwise opens existing file. |
| **vi -R filename** | Opens an existing file in read only mode. |
| **view filename** | Opens an existing file in read only mode |

# MODES

- The vi editor uses two basic modes:

  the command mode and the text mode.

  **Command Mode:-**
- When the vi editor is in the command mode, any key that is pressed by the user is considered a command.
- Commands are used to move the cursor, to delete or change part of the text, or to perform many other operations.

**Text Mode:-**

- When the vi editor is in the text mode, any key that is pressed by the user is considered text.

- The keyboard acts as a typewriter.

- In the text mode, the characters typed by the user, if they are printable characters, are inserted into the text at the cursor.

- This means that to add text in a document, we should first place the cursor at the desired location. To place the cursor, however, we must be in the command mode.

- The typical operations, therefore, is to place the cursor with a command, switch to the text mode and edit the text, then switch back to the command mode for the next operation.

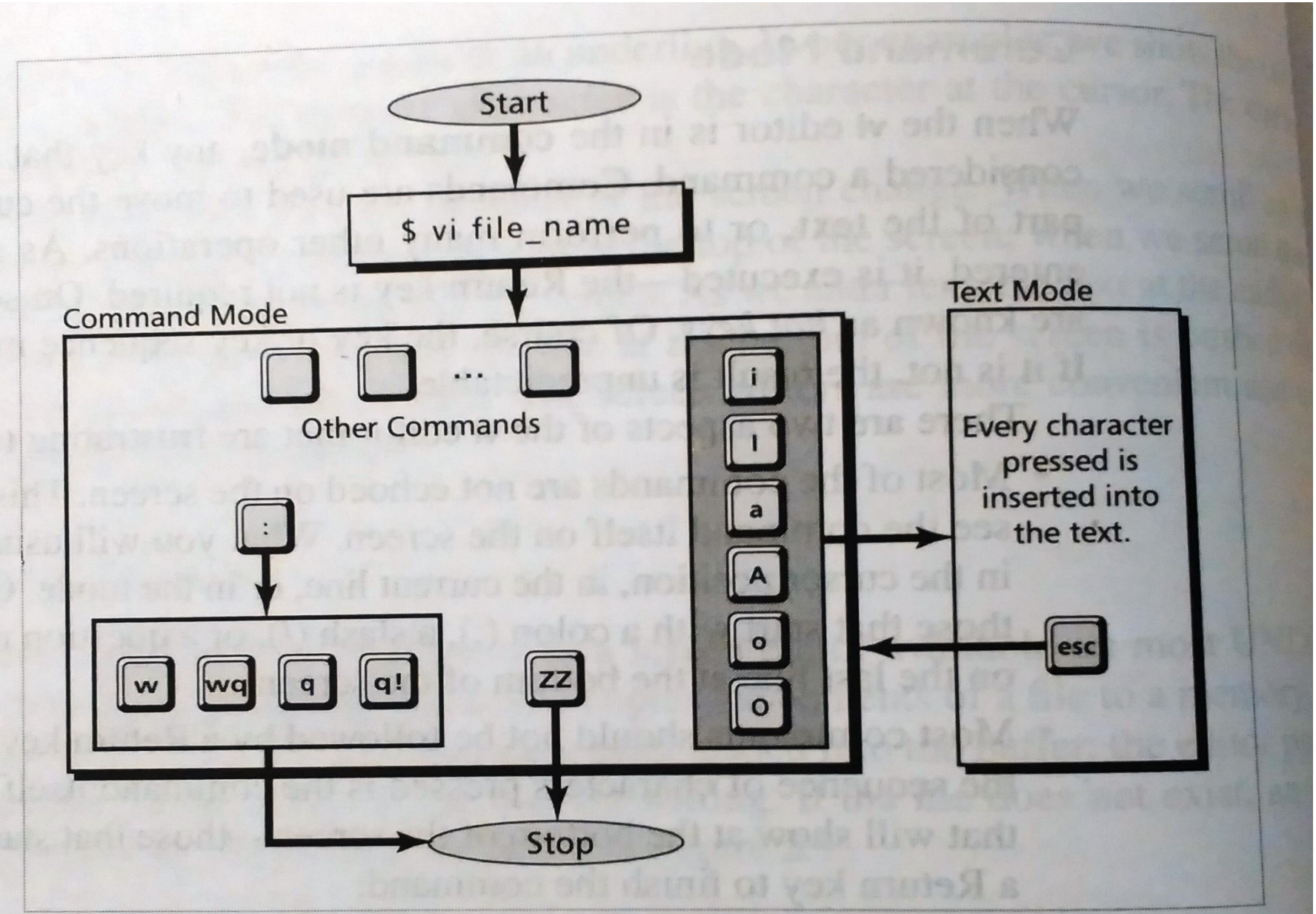



FIGURE 2.2 **vi** *Modes*

# Changing Modes

- It is clear that we must switch back and forth between vi command and text modes.
- To tell vi to do something, it must be in the command mode, to edit text, it must be in the text mode.
- To invoke vi, you type the following command at the UNIX prompt:

**$ vi filename**

- When you invoke vi, you are always in the command mode. During the session, you can move back and forth between the command mode and the text mode.

- To exit vi, you must be in command mode.

- There are six commands that take you to the text mode (a, A, i, I, o and O). Use any of these commands, vi switches immediately to the text mode.

- When you are in the text mode, you press the Escape key (esc) to go to the command mode.

- When you enter vi, you are in the command mode. To exit vi, you must be in the command mode.

# COMMANDS

ADD TEXT Commands.

- To insert text, you need to be in the text mode. The vi editor contains several commands to change the mode to text.

# 1. ADD TEXT COMMANDS

| Command | Description |
|---|---|
| **i** | Inserts text before current location. |
| **I** | Inserts text at the beginning of the current line. |
| **a** | Appends text after current character. |
| **A** | Appends text at the end of the current line. |
| **o** | Opens an empty text line for new text after the current line. |
| **O** | Opens an empty text line for new text before the current line. |

# Insert Commands (i and I)

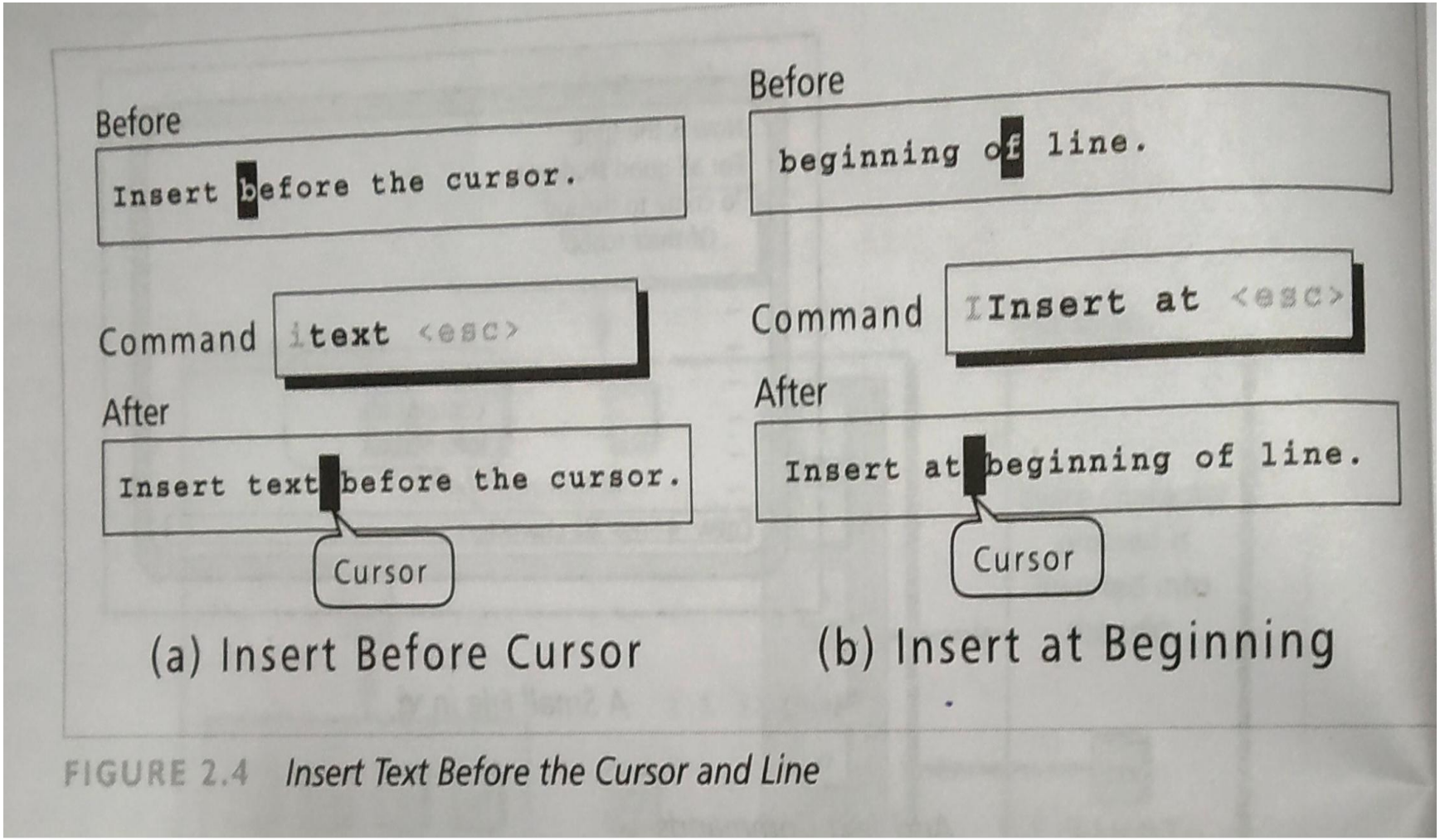



FIGURE 2.4 *Insert Text Before the Cursor and Line*

# Append Commands (a and A)

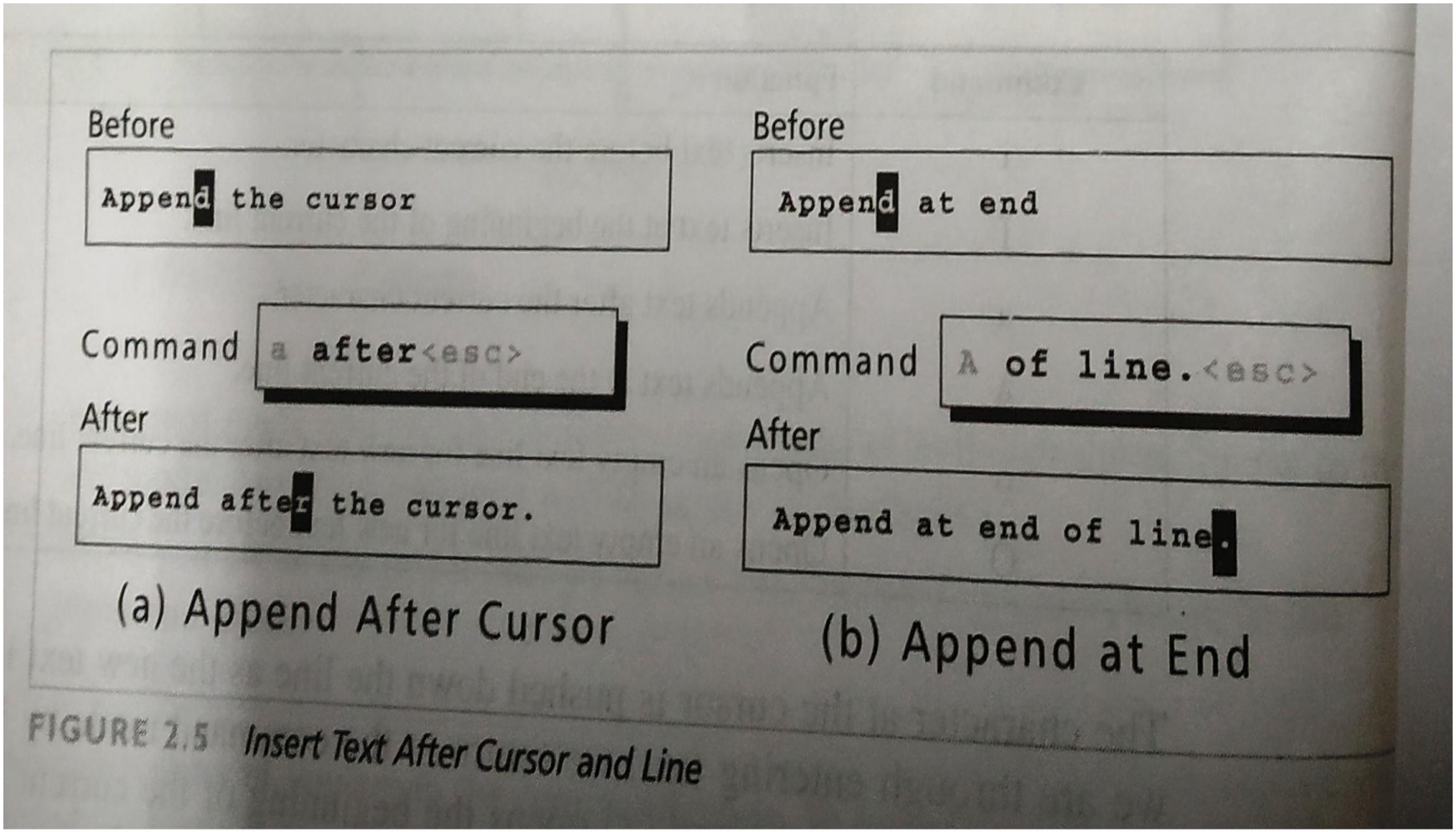



FIGURE 2.5 *Insert Text After Cursor and Line*

# New Line Commands (o or O)

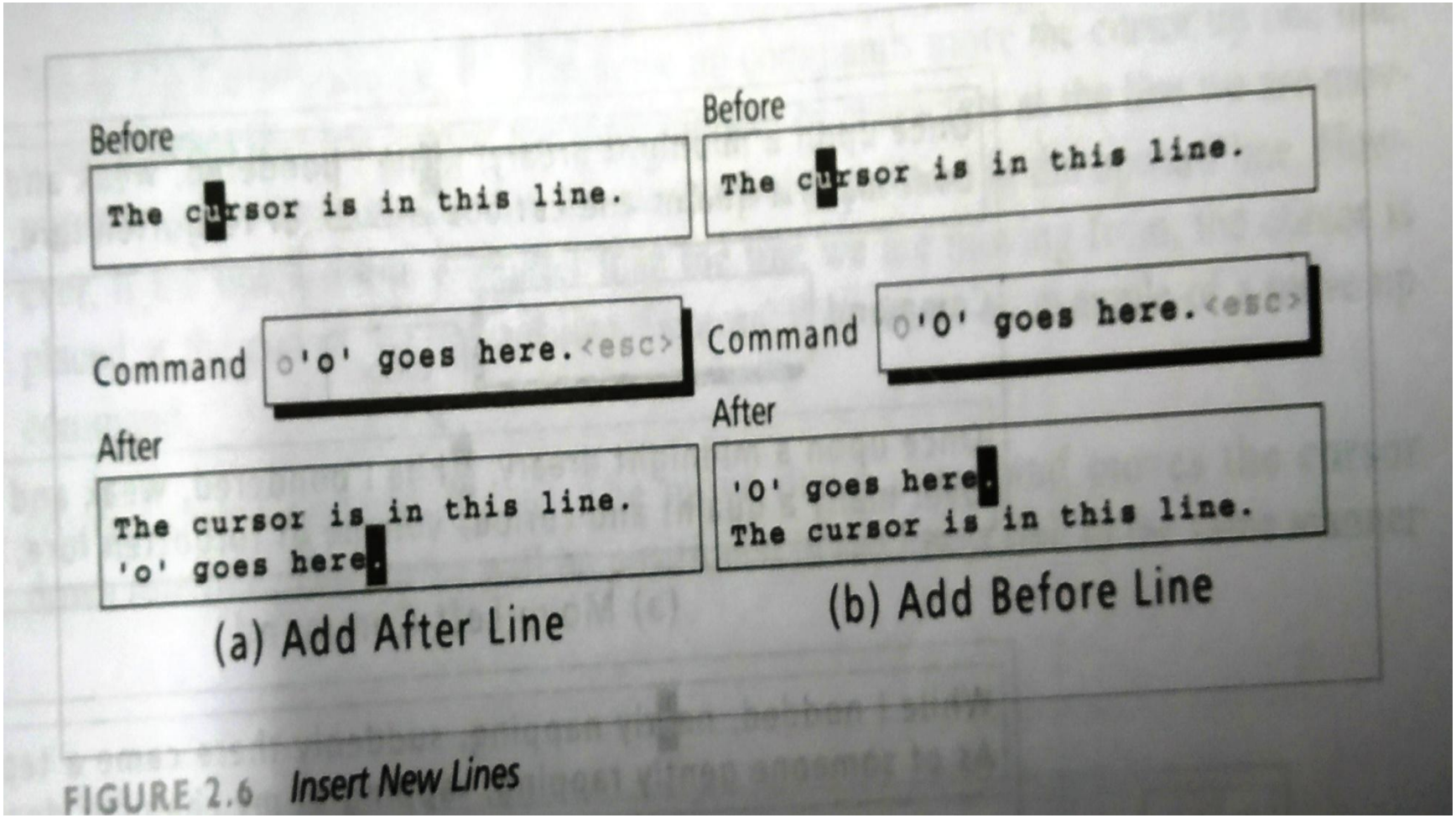



FIGURE 2.6 *Insert New Lines*

# 2. CURSOR MOVE COMMANDS

- To edit text, we need to move the cursor to the text to be edited.
- The cursor move commands are effective only in the command mode.
- After the execution of a move command, the vi editor is still in the command mode.
- There are many cursor move commands,

TABLE 2.2 *Cursor Move Commands*

| Command | Function |
|---|---|
| Horizontal Moves | |
| h, ⇐ , Backspace | Moves the cursor one character to the left. |
| l, ⇒, Spacebar | Moves the cursor one character to the right. |
| 0 | Moves the cursor to the beginning of the current line. |
| $ | Moves the cursor to the end of the current line. |
| Vertical Moves | |
| k, ⇑ | Moves the cursor one line up. |
| j, ⇓ | Moves the cursor one line down. |
| - | Moves the cursor to the beginning of the previous line. |
| +, Return | Moves the cursor to the beginning of the next line. |

# 3. DELETION COMMANDS

| Command | Function |
|---|---|
| **x** | Deletes the current character. |
| **dd** | Deletes the current line. |

# 4. JOIN COMMAND

- Two lines can be combined using join command (J). The command can be used anywhere in the first line.

- After the two lines have been joined, the cursor will be at the end of the first line.

# 5. SCROLLING COMMANDS

| Command | Function |
|---|---|
| **ctrl + y** | Scrolls up one line. |
| **ctrl + e** | Scrolls down one line. |
| **ctrl + u** | Scrolls up half a screen (12 lines) |
| **ctrl + d** | Scrolls down half a screen (12 lines) |
| **ctrl + b** | Scrolls up whole screen (24 lines) |
| **ctrl + f** | Scrolls down whole screen (24 lines) |

- Line Scroll Commands (ctrl + y and ctrl + e).
- Half Screen Commands (ctrl + u and ctrl + d).
- Full Page Commands (ctrl + b and ctrl + f).

# 6. UNDO COMMANDS

| Command | Function |
|---|---|
| **u** | Undoes only the last edit |
| **U** | Undoes all changes on the current line. |

# 7. SAVING AND EXIT COMMANDS

| Command | Function |
|---|---|
| **:w** | Saves the contents of the buffer without quitting vi. |
| **:w filename** | Writes contents of buffer to new file and continues. |
| **ZZ** | Saves the contents of the buffer and exits. |
| **:wq** | Saves the contents of the buffer and exits. |
| **:q** | Exits the vi (if buffer changed will not exit). |
| **:q!** | Exits the vi without saving. |

# DIRECTORY HANDLING SYSTEM CALLS

- The basic handling system calls provided by unix operating system are

1. opendir()
2. readdir()
3. rewinddir()
4. closedir()

5. mkdir()
6. rmdir()
7. umask()
8. seekdir()
9. telldir()

- **opendir(): -** opendir() function opens the director passed to it.

- **Syntax: -**

  #include <sys/types.h>

  #include <dirent.h>

  DIR *opendir(const char **name*);

  DIR *fdopendir(int *fd*);

- **Description: -**

- The **opendir**() function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream.

- The stream is positioned at the first entry in the directory.

- The **fdopendir**() function is like **opendir**(), but returns a directory stream for the directory referred to by the open file descriptor *fd*.

- After a successful call to **fdopendir**(), *fd* is used internally by the implementation, and should not otherwise be used by the application.

- **Return Value: -** The **opendir**() and **fdopendir**() functions return a pointer to the directory stream.

- On error, NULL is returned, and *errno* is set appropriately.

- **readdir(): -** The readdir() function takes pointer to a DIR structured returned by opendir() to read the directory.

- **Syntax: -**

  **#include <linux/types.h>**

  **#include <linux/dirent.h>**

  **int readdir(unsigned int** *fd*, **struct dirent ****dirp*, **unsigned int** *count*);

**Description: -**

**readdir**() reads one dirent structure from the directory pointed at by fd into the memory area pointed by dirp. The parameter count is ignored; at most one dirent structure is read. The dirent structure is declared as follows:

```
struct dirent
{
    long d_ino;              /* inode number */
    off_t d_off;             /* offset to this dirent */
    unsigned short d_reclen;   /* length of this d_name */
    char d_name [NAME_MAX+1];  /* filename (null-terminated) */
}
```

d_ino is an inode number. d_off is the distance from the start of the directory to this dirent. d_reclen is the size of d_name, not counting the null terminator. d_name is a null-terminated filename.

- **Return Value: -**
- On success, 1 is returned. On end of directory, 0 is returned. On error, -1 is returned, and *errno* is set appropriately.

- **rewinddir(): -** The rewinddir() function rewinds , that is, reposition the pointer at the first entry in the directory.

- **Syntax: -**

  **#include <sys/types.h>**

  **#include <dirent.h>**

  **void rewinddir(DIR **dirp*);**

- **Description: -** The **rewinddir**() function resets the position of the directory stream *dirp* to the beginning of the directory.

- **Return Value: -**

  The **rewinddir**() function returns no value.

- **closedir(): -** The closedir()  function closes the directory passed to it.

- **Syntax: -**

  #include <sys/types.h>

  #include <dirent.h>

  int closedir(DIR *dirp);

- **Description: -**
- The **closedir**() function closes the directory stream associated with *dirp*. A successful call to **closedir**() also closes the underlying file descriptor associated with *dirp*. The directory stream descriptor *dirp* is not available after this call.

- **Return Value: -** The **closedir**() function returns 0 on success. On error, -1 is returned, and *errno* is set appropriately.

- **mkdir(): -** mkdir() function is used to create directories. It creates a new, empty directory.

- **Syntax: -**

  **#include <sys/stat.h>**

  **#include <sys/types.h>**

  **int mkdir(const char **pathname*, mode_t *mode*);**

- **Description: -**
- **mkdir**() attempts to create a directory named *pathname*.
- The parameter *mode* specifies the permissions to use. It is modified by the process's *umask* in the usual way: the permissions of the created directory are (*mode* & ~*umask* & 0777). Other mode bits of the created directory depend on the operating system.

- **Return Value: - mkdir**() returns zero on success, or -1 if an error occurred (in which case, *errno* is set appropriately).

- **rmdir(): -** rmdir() function is useful to delete the directories.

- **Syntax: -**

  **#include <unistd.h>**

  **int rmdir(const char **pathname*);**

- **Description: - rmdir**() deletes a directory, which must be empty.

- **Return Value: -** On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

- **umask(): -** The umask() function sets the new umask value of the calling process and returns the old umask value. This function never return an error.

- **Syntax: -**

  #include<sys/types.h>

  #inlcude<sys/stat.h>

  mode_t umask(mode_t new_umask);

- The new_umask argument is specified as the bitwise ‘OR’ of any of the file access permission constants such as S_IRUSR, S_IWUSR, S_IXUSR etc;

- **seekdir(): -** set position of directory stream

- **Syntax: -**

  **#include <sys/types.h>**

  #include <dirent.h>

  void seekdir(DIR *dirp, long int loc);

- **Description: -**
- The seekdir() function sets the position of the next readdir() operation on the directory stream specified by dirp to the position specified by loc. The value of loc should have been returned from an earlier call to telldir(). The new position reverts to the one associated with the directory stream when telldir() was performed.

- If the value of loc was not obtained from an earlier call to telldir() or if a call to rewinddir() occurred between the call to telldir() and the call to seekdir(), the results of subsequent calls toreaddir() are unspecified.

- **Return Value: -**

  The seekdir() function returns no value.

- **telldir(): -** current location of a named directory stream

- **Syntax: -**

  #include <dirent.h>

  long int telldir(DIR *dirp);

- **Description: -**
- The telldir() function obtains the current location associated with the directory stream specified by dirp. If the most recent operation on the directory stream was a seekdir(), the directory position returned from the telldir() is the same as that supplied as a loc argument for seekdir().

- **Return Value: -**

  Upon successful completion, telldir() returns the current location of the specified directory stream

# fork Function

- An existing process can create a new one by calling the fork function.

```
#include <unistd.h>

pid_t fork(void);

/* Returns: 0 in child, process ID of child in parent, −1 on error */
```

- The new process created by fork is called the **child process**.

- This function is called once but returns twice.

- The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.

- **The two main reasons for fork to fail**

- If too many processes are already in the system, which usually means that something else is wrong.

- If the total number of processes for this real user ID exceeds the system's limit.

# vfork Function

- Both **fork()** and **vfork()** are the **system calls** that creates a new process that is identical to the process that invoked fork() or vfork().
- Using **fork()**allows the execution of parent and child process simultaneously. The other way, **vfork()** suspends the execution of parent process until child process completes its execution.

- The primary difference between the fork() and vfork() system call is that the child process created using fork has separate address space as that of the parent process.

- On the other hand, child process created using vfork has to share the address space of its parent process.

| BASIS FOR COMPARISON | FORK() | VFORK() |
| --- | --- | --- |
| Basic | Child process and parent process has separate address spaces. | Child process and parent process shares the same address space. |
| Execution | Parent and child process execute simultaneously. | Parent process remains suspended till child process completes its execution. |
| Modification | If the child process alters any page in the address space, it is invisible to the parent process as the address space are separate. | If child process alters any page in the address space, it is visible to the parent process as they share the same address space. |
| Copy-on-write | fork() uses copy-on-write as an alternative where the parent and child shares same pages until any one of them modifies the shared page. | vfork() does not use copy-on-write. |

# exit Functions

- _exit, _Exit - terminate the current process

```
#include <unistd.h>

void _exit(int status);

#include <stdlib.h>

void _Exit(int status);
```

- The function **_exit**() terminates the calling process "immediately".

- Any open file descriptors belonging to the process are closed; any children of the process are inherited by process 1, *init*, and the process's parent is sent a **SIGCHLD**signal.

# wait function

- When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent.
- Because the termination of a child is an asynchronous event (it can happen at any time while the parent is running).
- This signal is the asynchronous notification from the kernel to the parent.

- The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler. The default action for this signal is to be ignored.

- A process that calls wait or waitpid can:

1. Block, if all of its children are still running

2. Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched.

3. Return immediately with an error, if it doesn't have any child processes

```
#include <sys/wait.h>

pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);

/* Both return: process ID if OK, 0 (see later), or -1 on error */
```

- The differences between these two functions are:

- The wait function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking.

- The waitpid function doesn’t wait for the child that terminates first; it has a number of options that control which process it waits for.

## wait3 and wait4 Functions

- Most UNIX system implementations provide two additional functions: wait3 and wait4, with an additional argument *rusage* that allows the kernel to return a summary of the resources used by the terminated process and all its child processes.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

pid_t wait3(int *statloc, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);

/* Both return: process ID if OK, 0, or −1 on error */
```

# exec Functions

- One use of the fork function is to create a new process (the child) that then causes another program to be executed by calling one of the exec functions.
- When a process calls one of the exec functions, that process is completely replaced by the new program which starts executing at its main function.

- The process ID does not change across an exec, because a new process is not created.

- exec merely replaces the current process (its text, data, heap, and stack segments) with a brand-new program from disk.

- There are seven different exec functions:

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );
int execv(const char *pathname, char *const argv[]);
int execle(const char *pathname, const char *arg0, ...
        /* (char *)0, char *const envp[] */ );
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );
int execvp(const char *filename, char *const argv[]);
int fexecve(int fd, char *const argv[], char *const envp[]);

/* All seven return: −1 on error, no return on success */
```

- The first four take a pathname argument, the next two take a filename argument, and the last one takes a file descriptor argument.

Differences among the seven exec functions

| **Function** | ***pathname*** | ***filename*** | ***fd*** | **Arg list** | **argv[]** | **environ** | **envp[]** |
|---|---|---|---|---|---|---|---|
| execl | * | | | * | | * | |
| execlp | | * | | * | | * | |
| execle | * | | | * | | | * |
| execv | * | | | | * | * | |
| execvp | | * | | | * | * | |
| execve | * | | | | * | | * |
| fexecve | | | * | | * | | * |
| (letter in name) | | p | f | l | v | | e |

# system Function

- It is convenient to execute a command string from within a program.

```
#include <stdlib.h>

int system(const char *cmdstring);
```

UNIX System process control primitives:

- fork creates new processes
- exec functions initiates new programs
- exit handles termination
- wait functions handle waiting for termination

- **Process control commands in Unix are:**
- **bg -** put suspended process into background
- **fg -** bring process into foreground
- **jobs -** list processes

- **Jobs Command :** Jobs command is used to list the jobs that you are running in the background and in the foreground.

- If the prompt is returned with no information no jobs are present.
- **Syntax :**

  jobs [JOB]

- **Options**
- **-l** Lists process IDs in addition to the normal information.
- **-n** List only processes that have changed status since the last notification.
- **-p** Lists process IDs only.
- **-r** Restrict output to running jobs.
- **-s** Restrict output to stopped jobs.

# DIRECTORY RELATED UTILITES

1. pwd
2. ls
3. mkdir
4. cd
5. rmdir

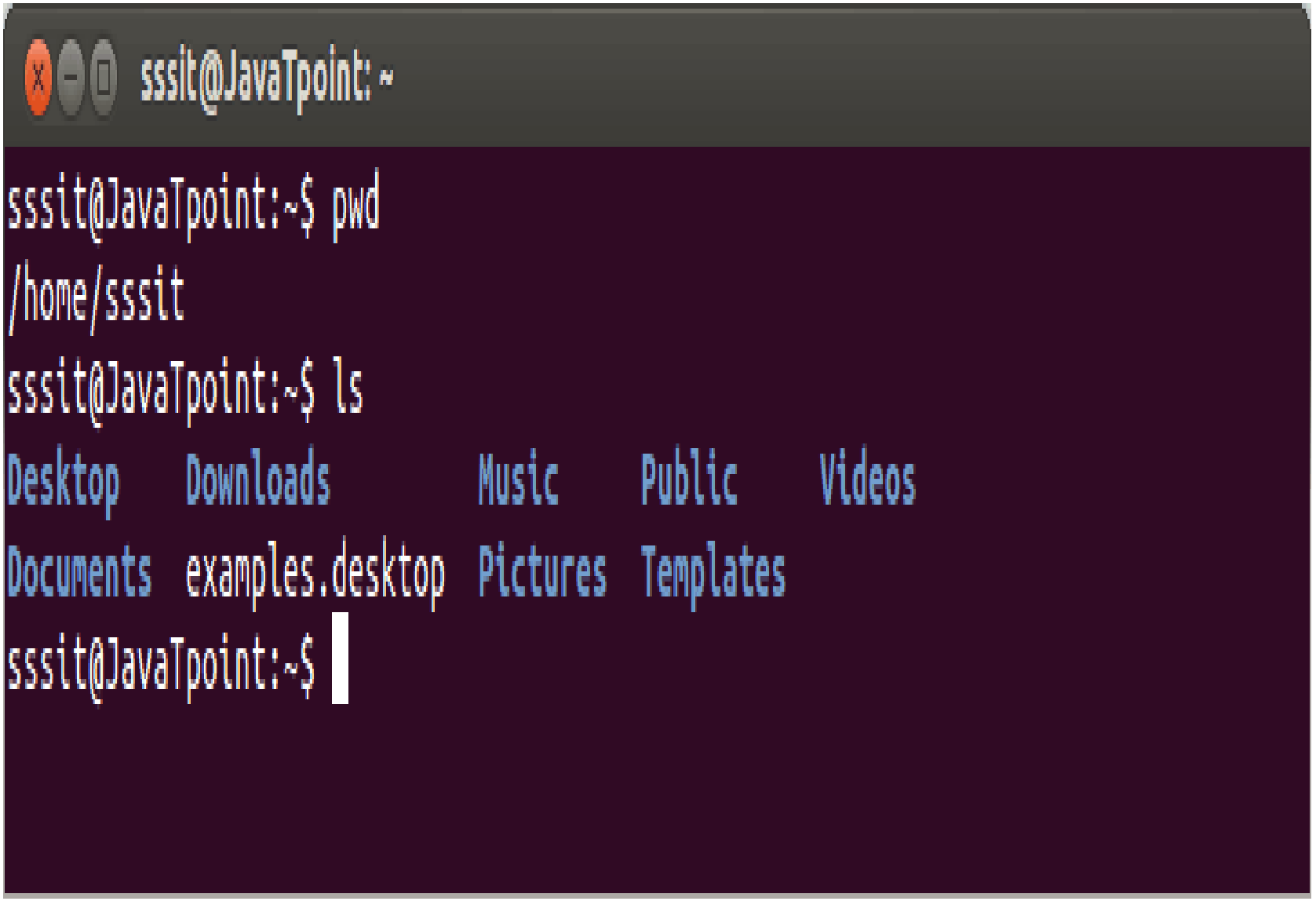
# 1. pwd

- The command used to determine the current directory is print working directory (pwd).

- It has no options and no attributes.

**Example:-**

- $pwd


sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ pwd
/home/sssit
sssit@JavaTpoint:~$ ls
Desktop Downloads Music Public Videos
Documents examples.desktop Pictures Templates
sssit@JavaTpoint:~$

# 2. ls

- The list command lists the contents in a directory. Depending on the options used, it can list files, directories or subdirectories.

**Syntax:-**

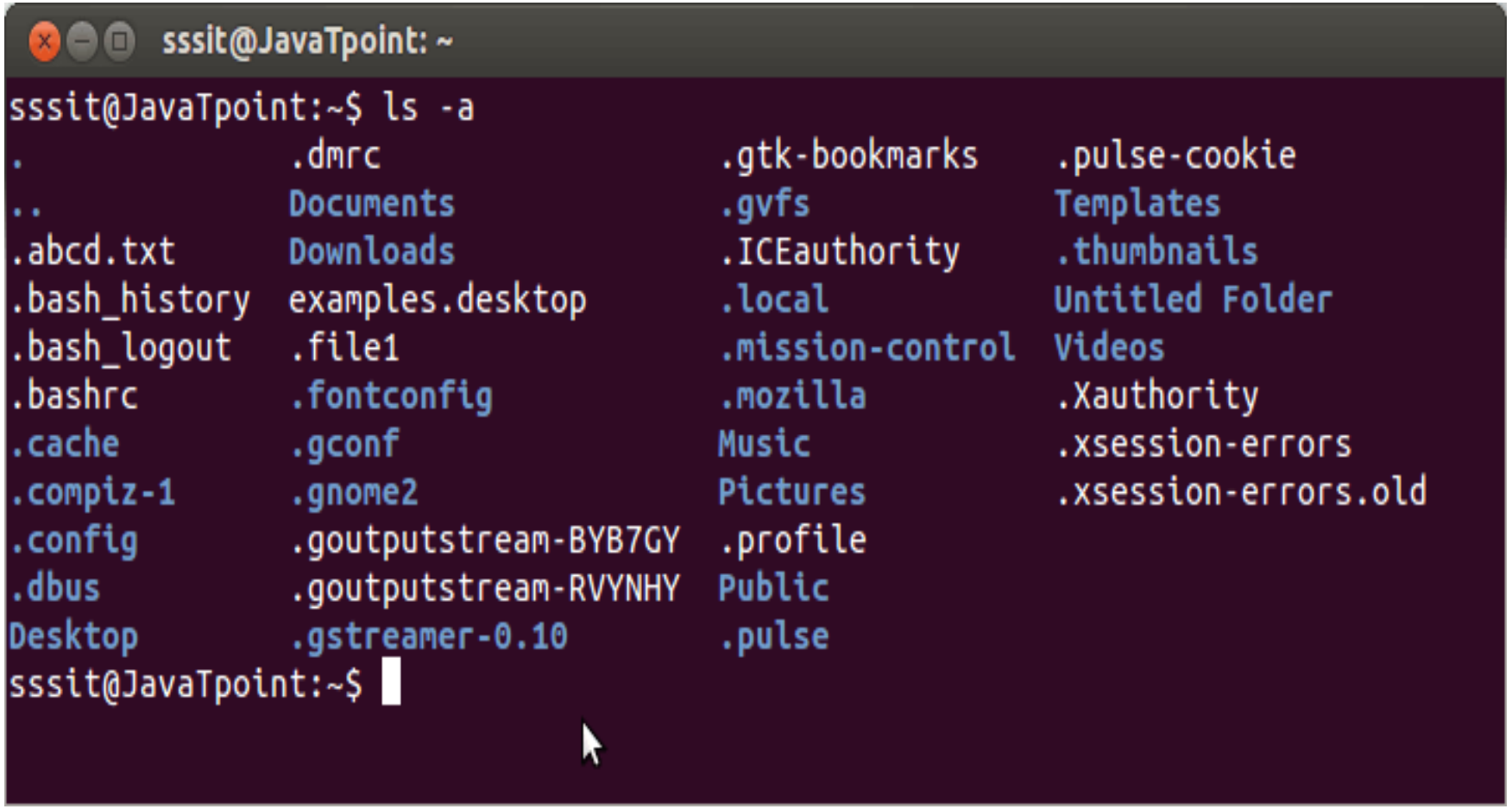
- $ls [options] [path]

**Example:-**

- $ls

# ls command options

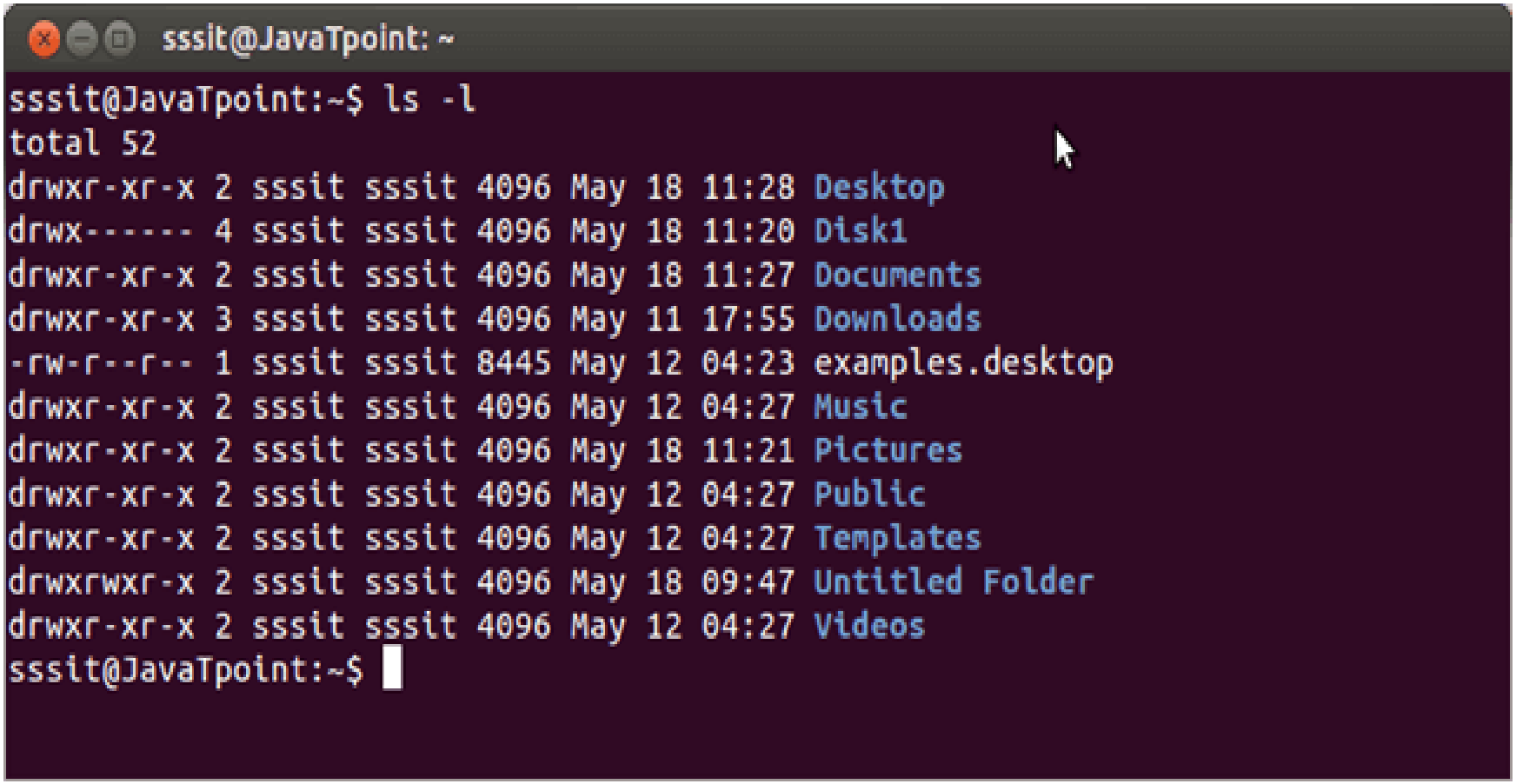
1. ls –a:- List all files including hidden files.



2. ls –A:- List all files including hidden files except for “.” and “. .”.

3. ls –R:- List all files recursively, descending down the directory tree from the given path.

4. ls –l:- List the files in long format. i.e., with an index number, owner name, group name, size and permissions.



5. ls –o:- List the files in long format but without the group name.

6. ls –g:- List the files in long format but without the owner name.

7. ls –i:- List the files along with their index number.

8. ls –s :- List the files along with their size.

9. ls –S:- Sort the list by size, with the largest at the top.

10. ls –r:- Reverse the sorting order.

11. ls -1:- there will be situations in which you want the filenames printed as a column rather than several files in one line

# 3. mkdir

- To create a new directory, you use the make directory (mkdir) command.

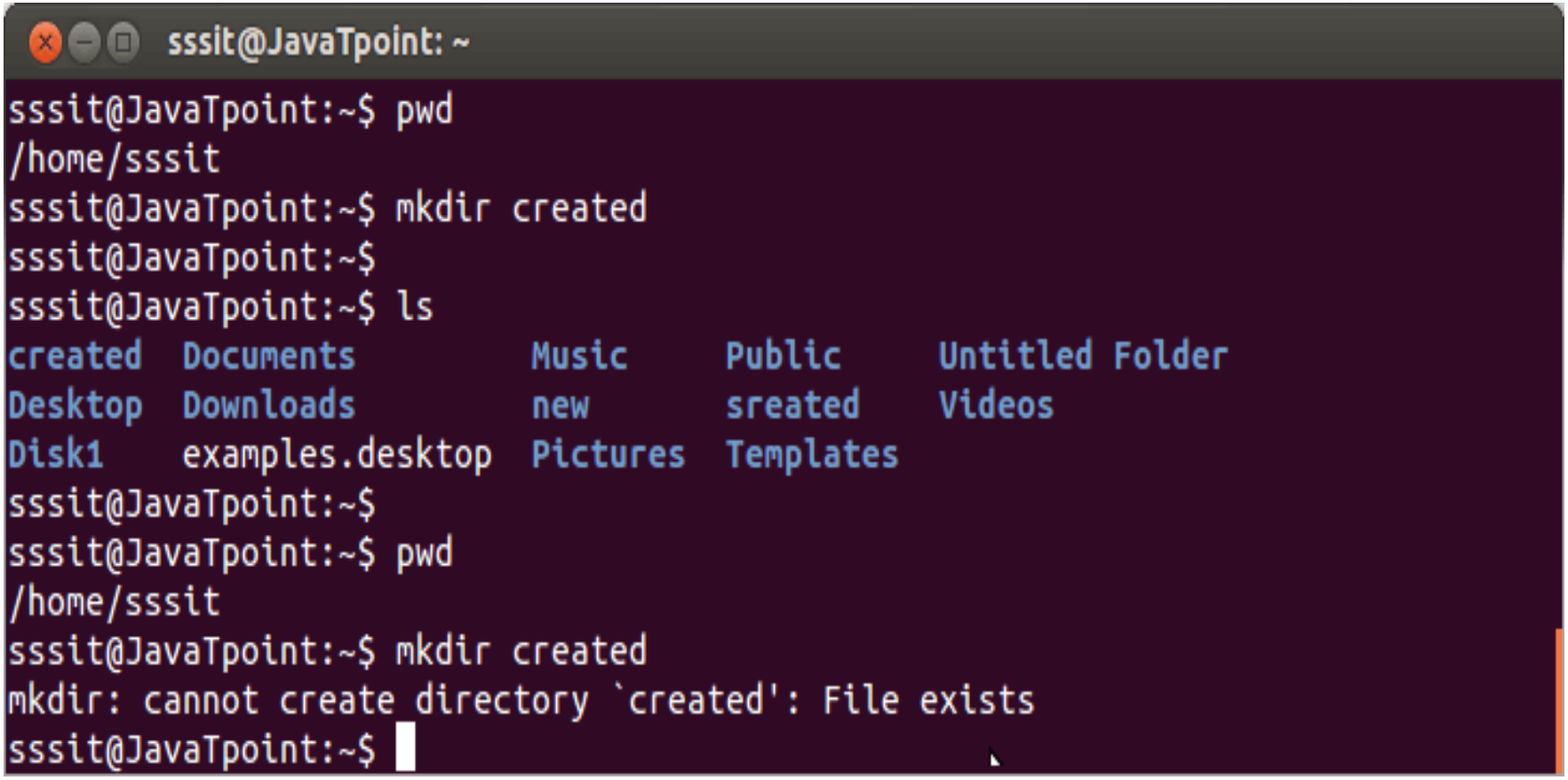
- It has two options : permission mode and parent directories.

**Syntax:-**

- mkdir [options . . . ] [directories . . .]

## Example:-

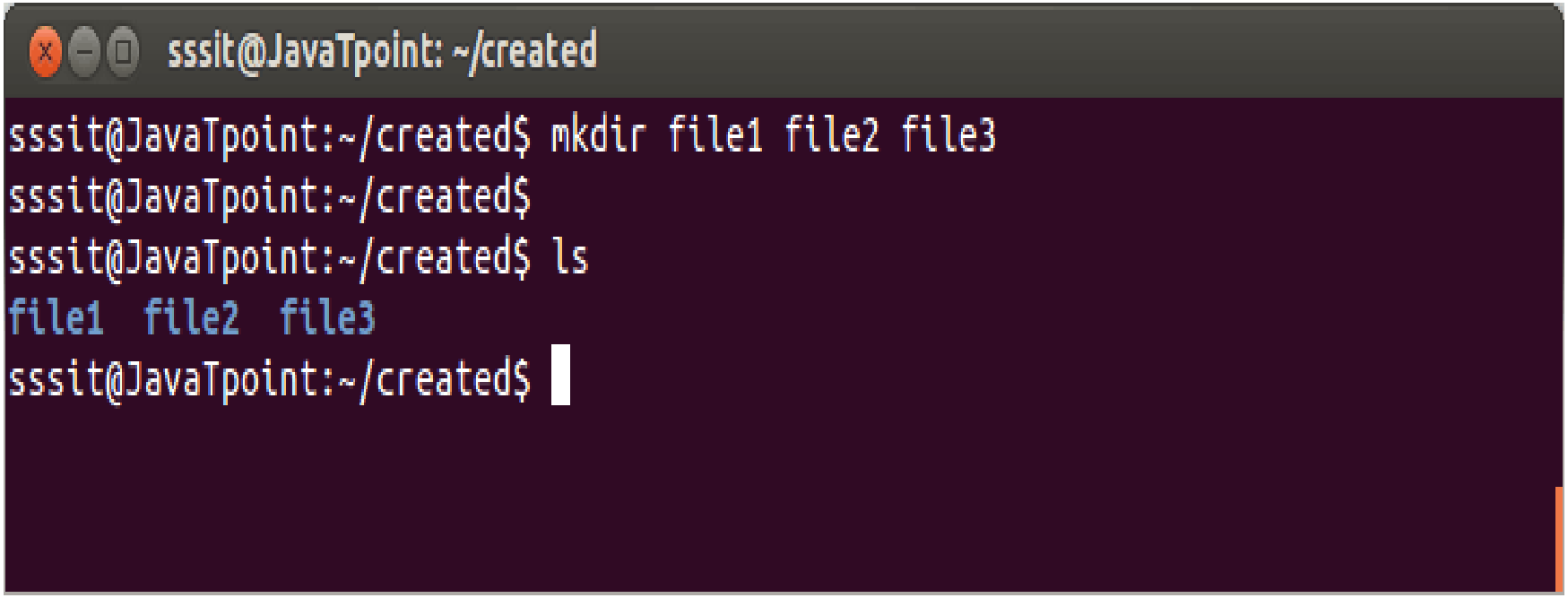
- $mkdir dharani



# To make multiple directories

**Syntax:**

- mkdir **<dirname1> <dirname2> <dirname3>** ...



# 4. cd

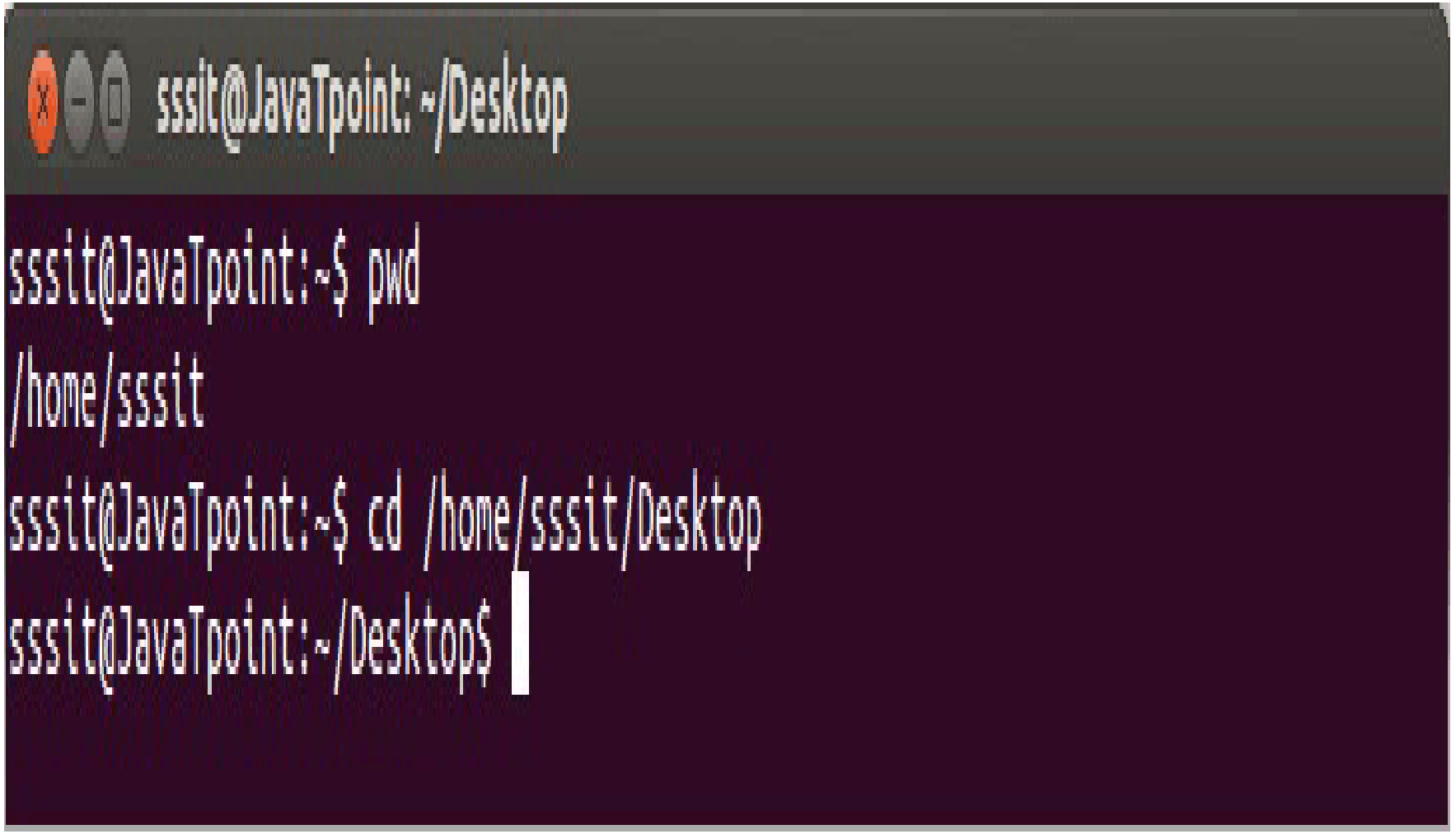
- The cd stands for ‘change directory’ and this command is used to change the current directory i.e., the directory in which the user is currently working.

  **Syntax:-**

- $cd <dirname>

  **Example:-**

- $cd dharani


sssit@JavaTpoint: ~/Desktop
sssit@JavaTpoint:~$ pwd
/home/sssit
sssit@JavaTpoint:~$ cd /home/sssit/Desktop
sssit@JavaTpoint:~/Desktop$

# 5. rmdir

- When a directory is no longer needed, it should be removed. The remove directory (rmdir) command deletes directories.

- But will not be able to delete a directory including a sub-directory. It means, a directory has to be empty to be deleted.

**Syntax:**

- $rmdir **<dirname>**

**Example:**

- $rmdir created

**5.ls**

ls [options] [names]

- If no names are given, list the files in the current directory.
- With one or more names, list files contained in a directory name or that match a file name.
- The options let you display a variety of information in different formats.

Options

-a :List all files, including the normally hidden . files.

-b :Show nonprinting characters in octal.

-c :List files by inode modification time.

-C :List files in columns (the default format, when displaying to a terminal device).

-d :List only the directory's information, not its contents. (Most useful with -l and -i.)

-f :Interpret each name as a directory (files are ignored).

-g :Like -l, but omit owner name (show group).

-i :List the inode for each file.

-l :Long format listing (includes permissions, owner, size, modification time, etc.).

-L :List the file or directory referenced by a symbolic link rather than the link itself.

-m :Merge the list into a comma-separated series of names.

-n :Like -l, but use user ID and group ID numbers instead of owner and group names.

-o :Like -l, but omit group name (show owner).

-p :Mark directories by appending / to them.

-q :Show nonprinting characters as ?.

-r :List files in reverse order (by name or by time).

-R :Recursively list subdirectories as well as current directory.

-s :Print sizes of the files in blocks.

-t :List files according to modification time (newest first).

-u :List files according to the file access time.

-x :List files in rows going across the screen.

-1 :Print one entry per line of output.

Examples

List all files in the current directory and their sizes; use multiple columns and mark special files:

ls -asCF

# File Handling Utilities

1. Create file (cat)
2. Edit file (sed)
3. Display file (more)
4. Print file (lpr)

# 1. CREATE FILE (cat)

- The most common tool to create a text file is a text editor such as vi.
- Other utilities, such as **cat,** that are useful to create small file.
- The cat command is the most universal and powerful tool.

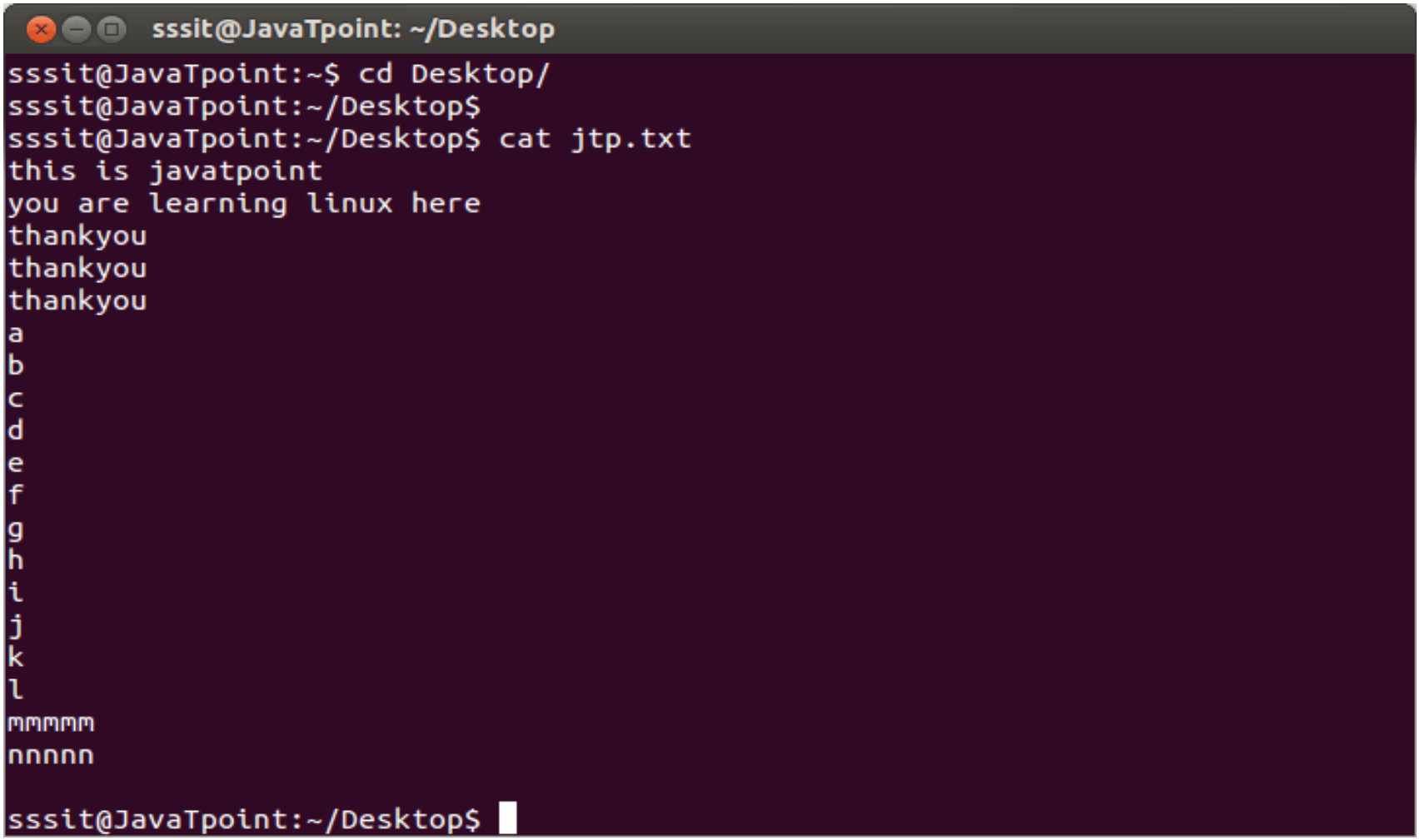
- It can be used to display the content of a file, copy content from one file to another, concatenate the contents of multiple files, display the line number, display $ at the end of the line, etc

**To display file content:-**

- The cat command can be used to display the content of a file.

**Syntax:-** $cat <filename>

**Example:-** $cat jtp.txt

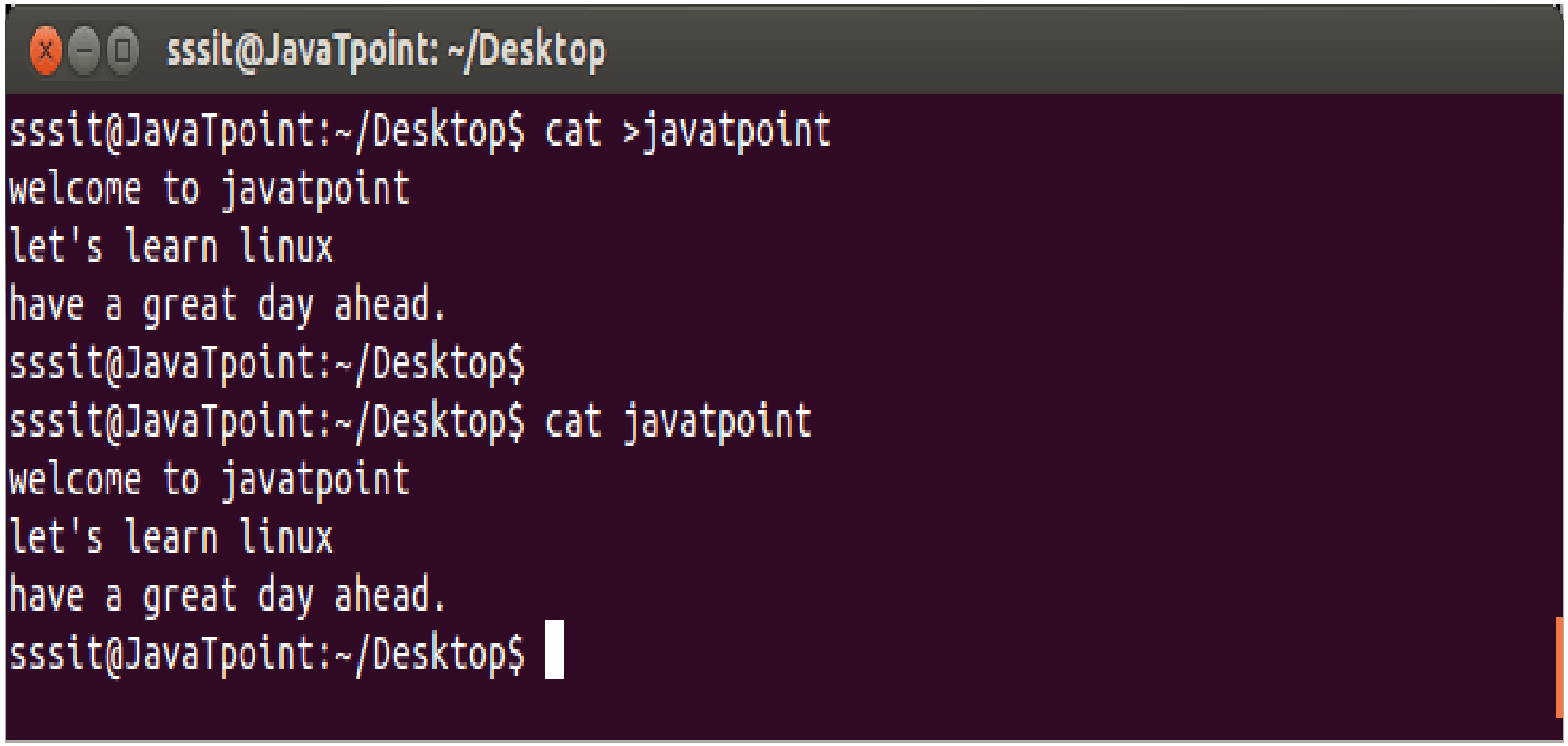


1. **To create a file:-**

   The cat command can be used to create a new file with greater than sign (>).

   **Syntax:-** cat > [filename]

   **Example:-** $ cat > hai

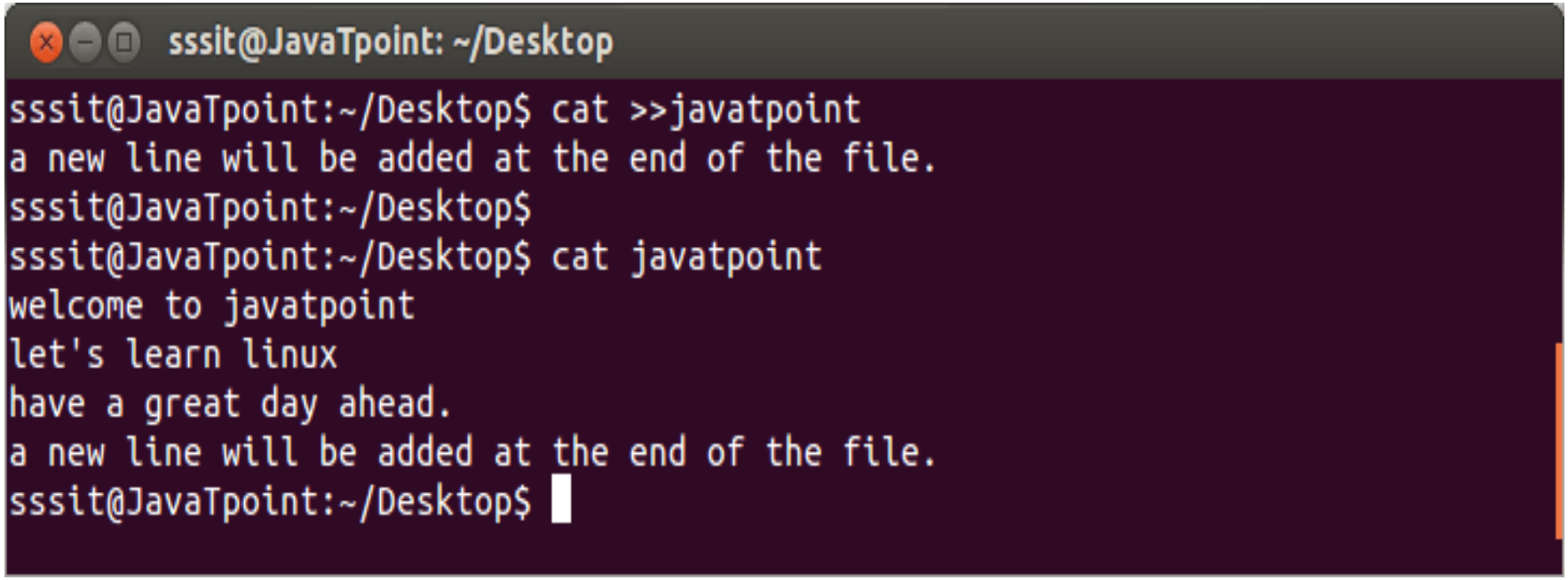


## 2. To Append the content of a file:-

The cat command with double greater than sign (>>) append something in your already existing file.

**Syntax:-** $cat >> (filename)

**Example:-** $cat >> hai

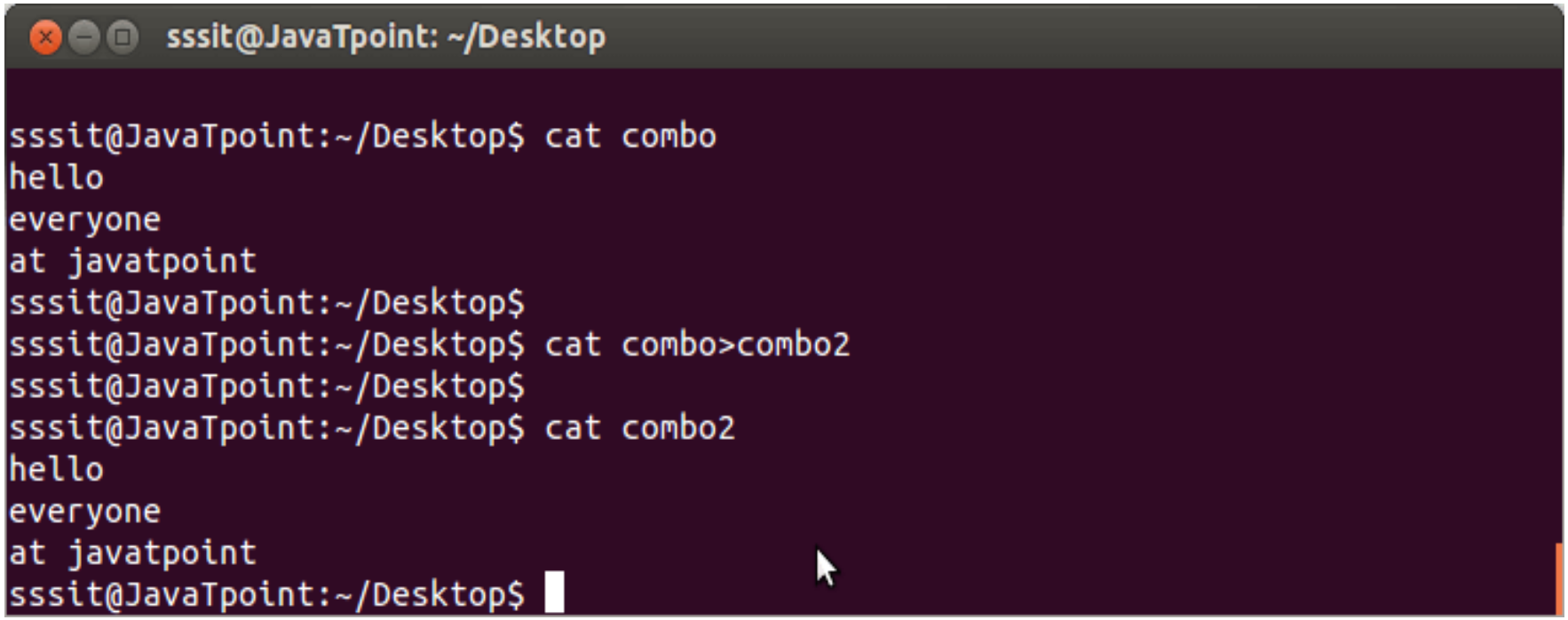


## 3. To copy file:-

The cat command can be used to copy the content of a file into another file.

**Syntax:-** cat (old file name) > (new file name)

**Example:-** $cat combo>combo2

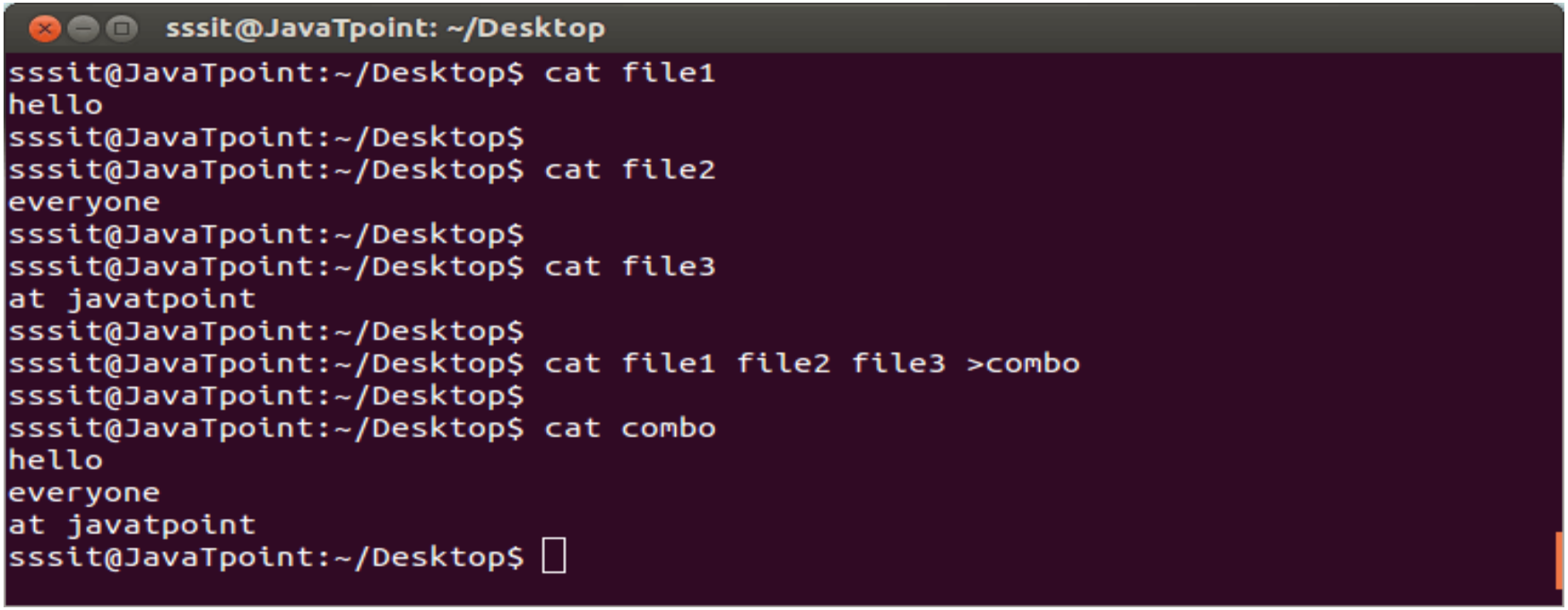


## 4. To concatenate file:-

The cat command can be used to concatenate the contents of multiple files in a single new file.

**Syntax:-** cat <file1> <file2>.....> <new file>

**Example:-** $cat file1 file2 file3 > combo



5. **To Insert a new file:-**

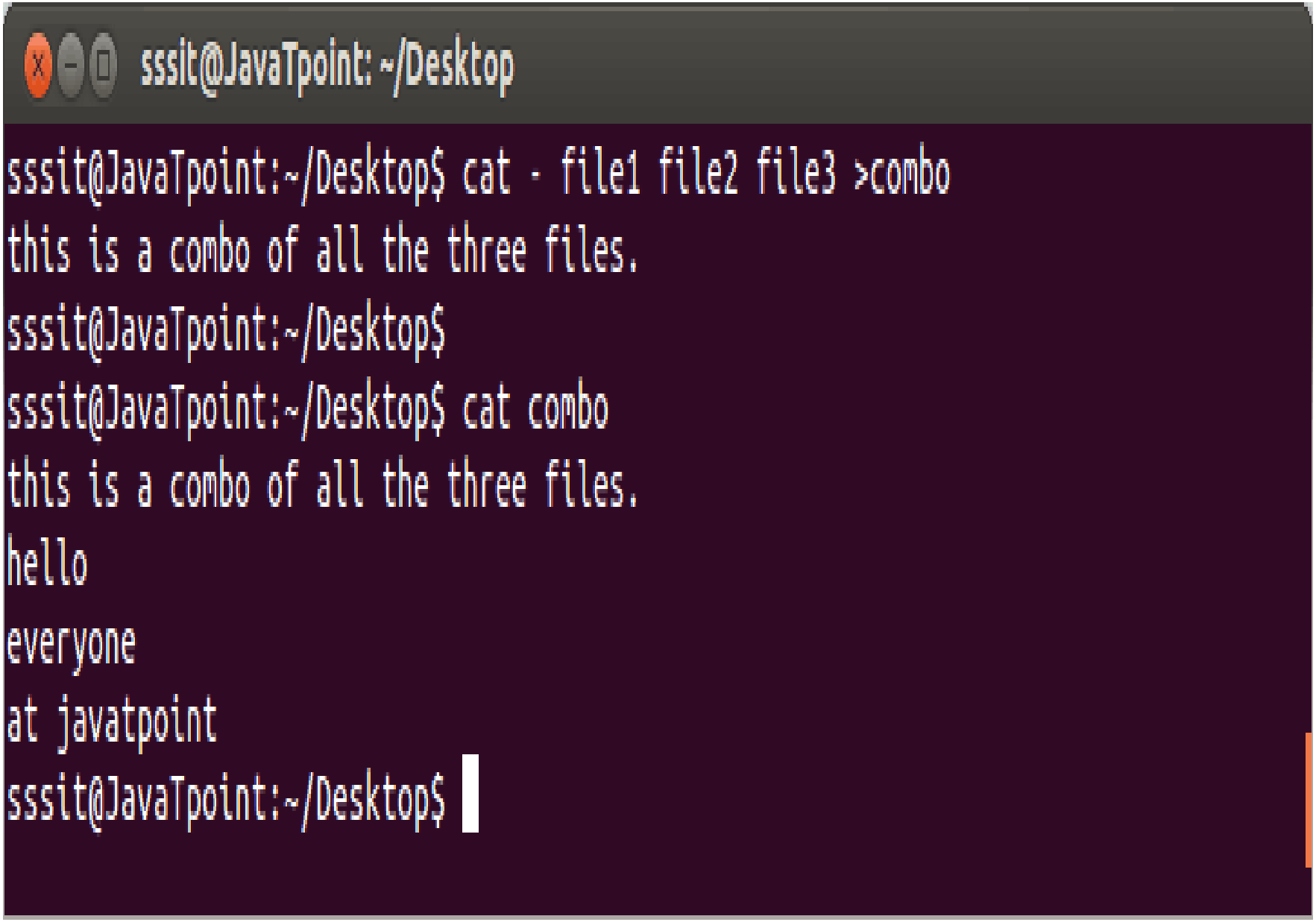
   A new line will be inserted while concatenating multiple files by using a hyphen (-).

   **syntax:**

cat -
**<filename1> <filename2>**. . . . **> <new** filena
me**>**

**Example:**

cat - file1 file2 file3 **>**combo

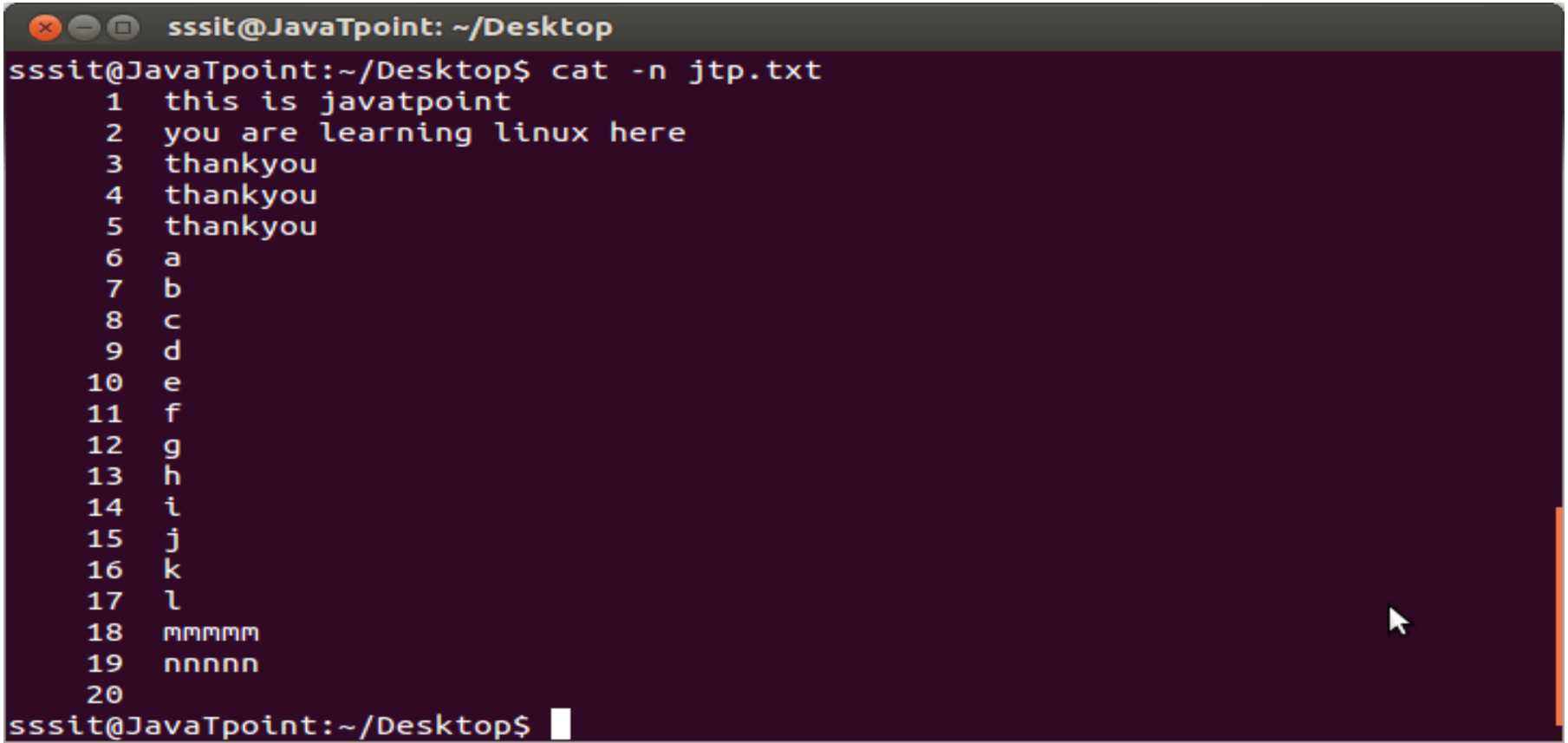

sssit@JavaTpoint:~/Desktop
sssit@JavaTpoint:~/Desktop$ cat - file1 file2 file3 >combo
this is a combo of all the three files.
sssit@JavaTpoint:~/Desktop$
sssit@JavaTpoint:~/Desktop$ cat combo
this is a combo of all the three files.
hello
everyone
at javatpoint
sssit@JavaTpoint:~/Desktop$

## 6. cat –n command:-

The 'cat -n' option displays line numbers in front of each line in a file.

**Syntax:** cat -n **<fileName>**

**Example:** cat -n jtp.txt

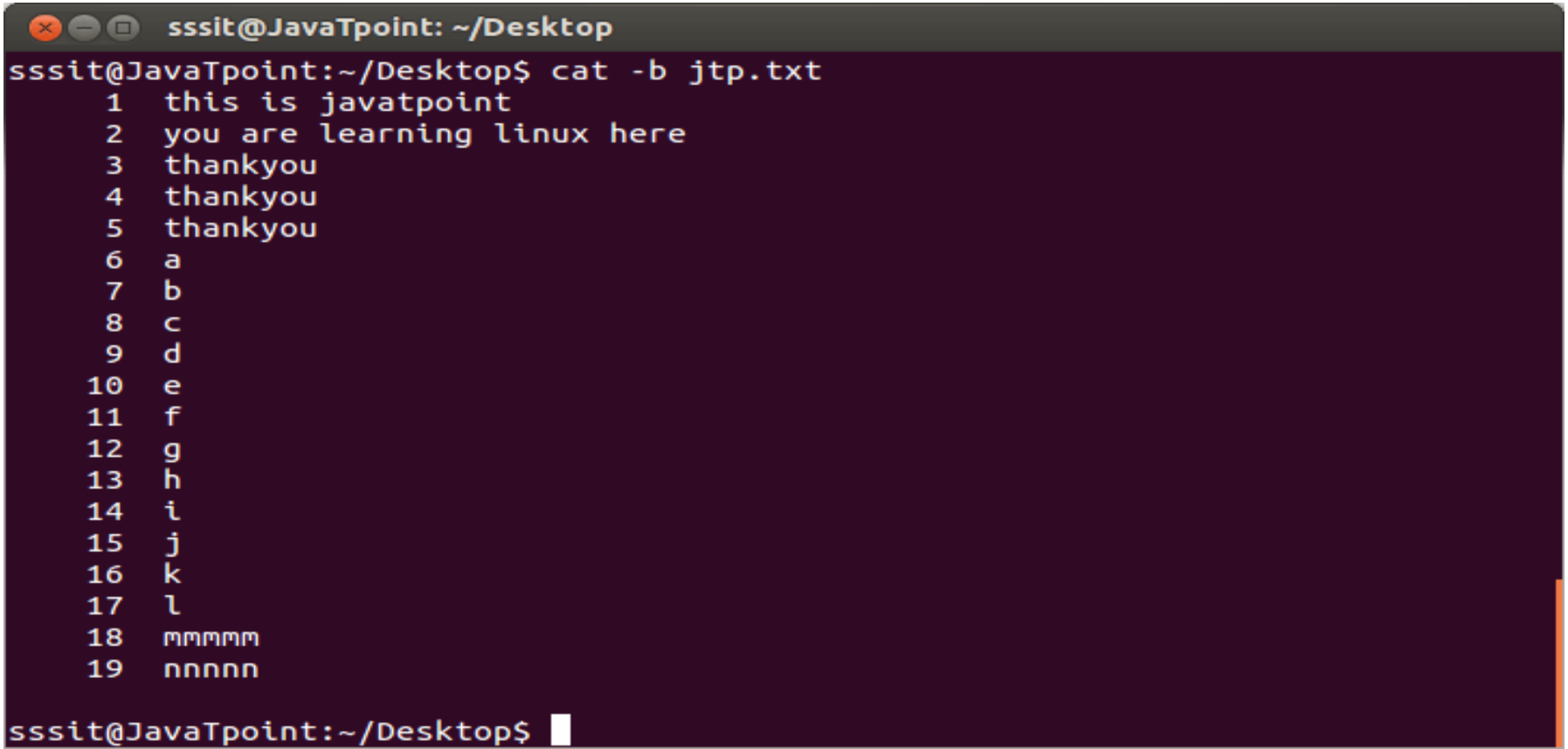


## 7. cat –b:-

The 'cat -b' option removes the empty lines.

**Syntax:** cat -b (file name)

**Example:** cat -b jtp.txt

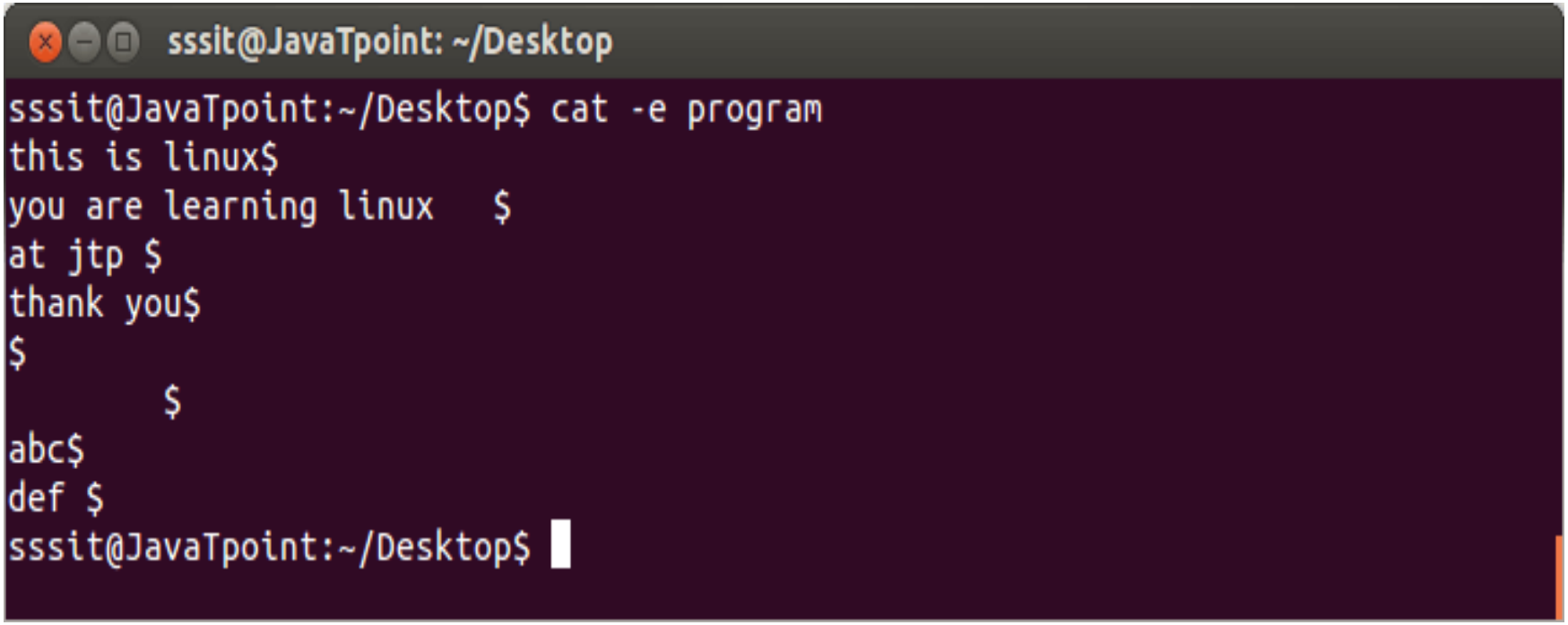


## 8. cat –e command:-

The 'cat-e' option displays a **'$'** sign at the end of every line.

**Syntax:** cat -e **<fileName>**

**Example:** cat -e program

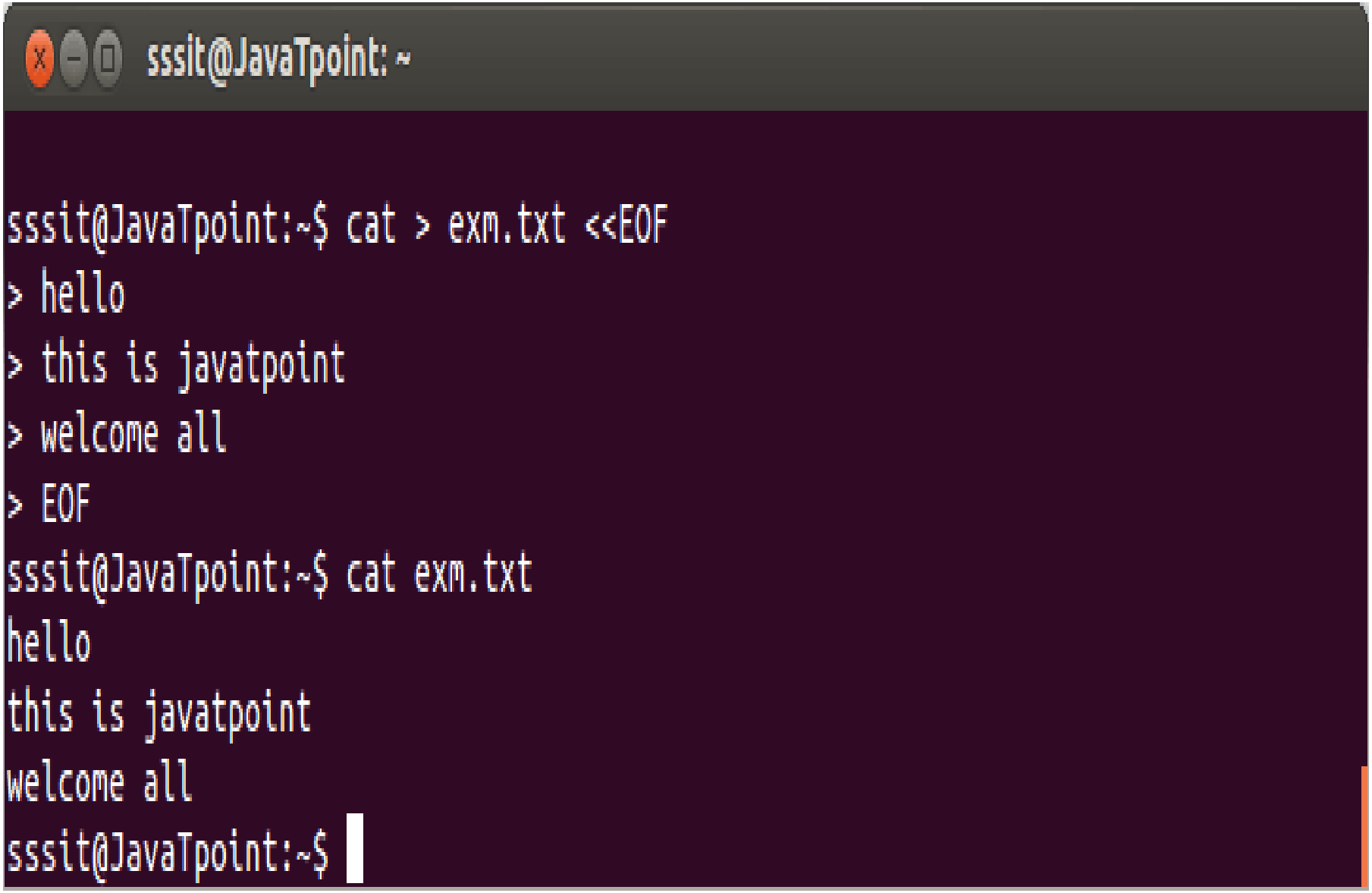


## 9. cat command (as an end marker): -

- The 'cat << EOF ' option displays an end marker at the end of a file.
- It is called **here directive** and file content will be saved at the given end marker.
- The file can be saved with the help of **'ctrl + d '** keys also. It works like the end marker.

  **Syntax:** cat **<< EOF**

  **Example:** cat **>** exm.txt **<< EOF**


sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ cat > exm.txt <<EOF
> hello
> this is javatpoint
> welcome all
> EOF
sssit@JavaTpoint:~$ cat exm.txt
hello
this is javatpoint
welcome all
sssit@JavaTpoint:~$

# 2. EDIT FILE (sed)

- UNIX provides several utilities to edit text files. The most common is a basic text editor such as vi.
- In addition, there are others that, such as **sed**, that provide powerful search and edit tools.

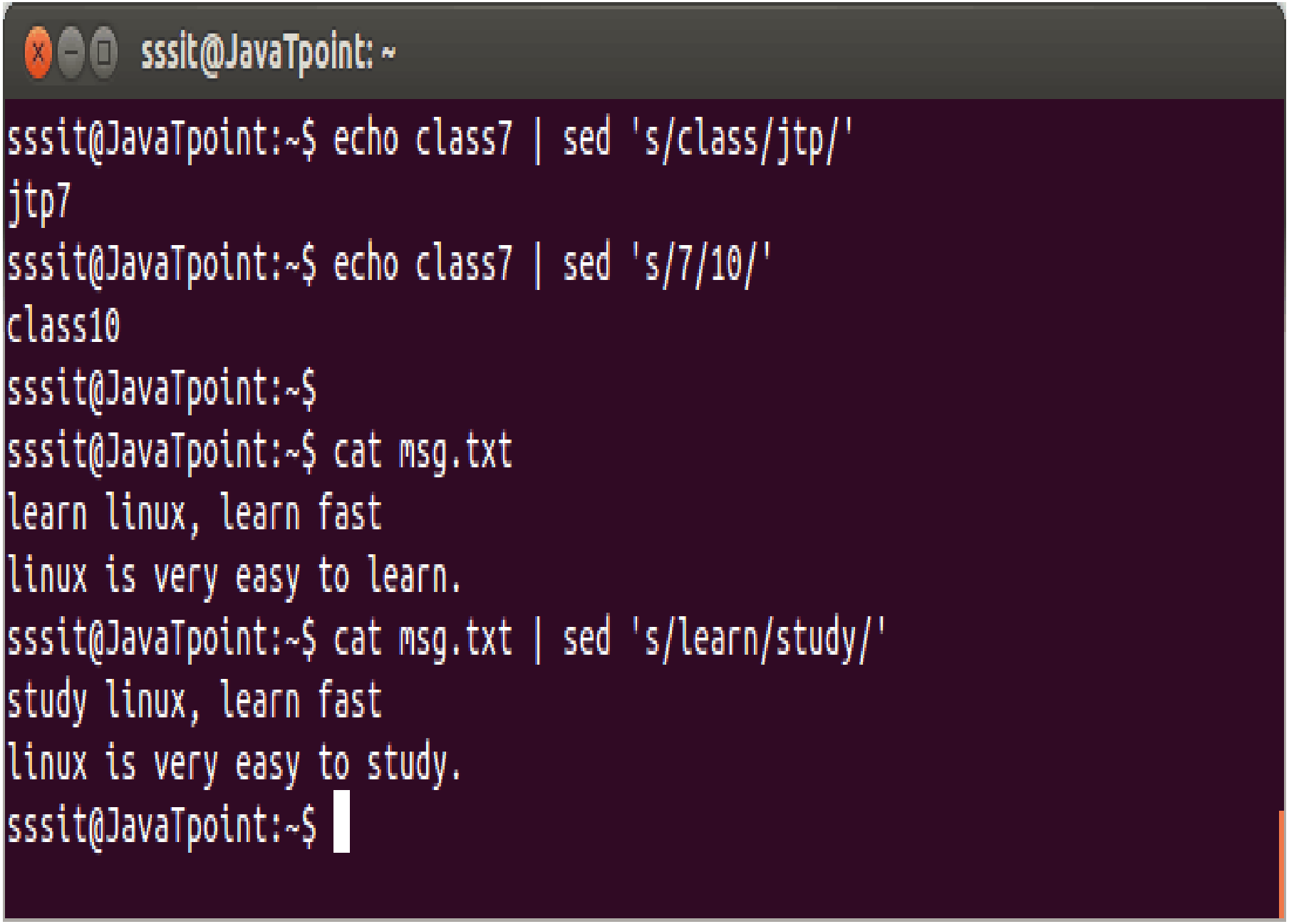
- All of the basic edit utilities can create a file, but only some can edit one.

**Syntax:**

- command | sed 's/**<oldWord>**/**<newWord>**/'

**Example:**

- echo class7 | sed 's/class/jtp/'
- echo class7 | sed 's/7/10/'
- cat msg.txt | sed 's/learn/study/'


sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ echo class7 | sed 's/class/jtp/'
jtp7
sssit@JavaTpoint:~$ echo class7 | sed 's/7/10/'
class10
sssit@JavaTpoint:~$
sssit@JavaTpoint:~$ cat msg.txt
learn linux, learn fast
linux is very easy to learn.
sssit@JavaTpoint:~$ cat msg.txt | sed 's/learn/study/'
study linux, learn fast
linux is very easy to study.
sssit@JavaTpoint:~$

```
idelab63@idelab63: ~
this is a book
hai
how r u
idelab63@idelab63:~$ cat welcome
this is a book
hai
how r u
idelab63@idelab63:~$ cat welcome | sed 's/book/study/'
this is a study
hai
how r u
idelab63@idelab63:~$ cat > unix
welcome to unix lab
hello
idelab63@idelab63:~$ ls
Desktop  Documents  Downloads  examples.desktop  Music  Pictures  Public  script
.odt  Templates  typescript  unix  Videos  welcome
idelab63@idelab63:~$ cat unix
welcome to unix lab
hello
idelab63@idelab63:~$ cat unix | sed 's/lab/class/'
welcome to unix class
hello
idelab63@idelab63:~$
```

**Global Replacement**

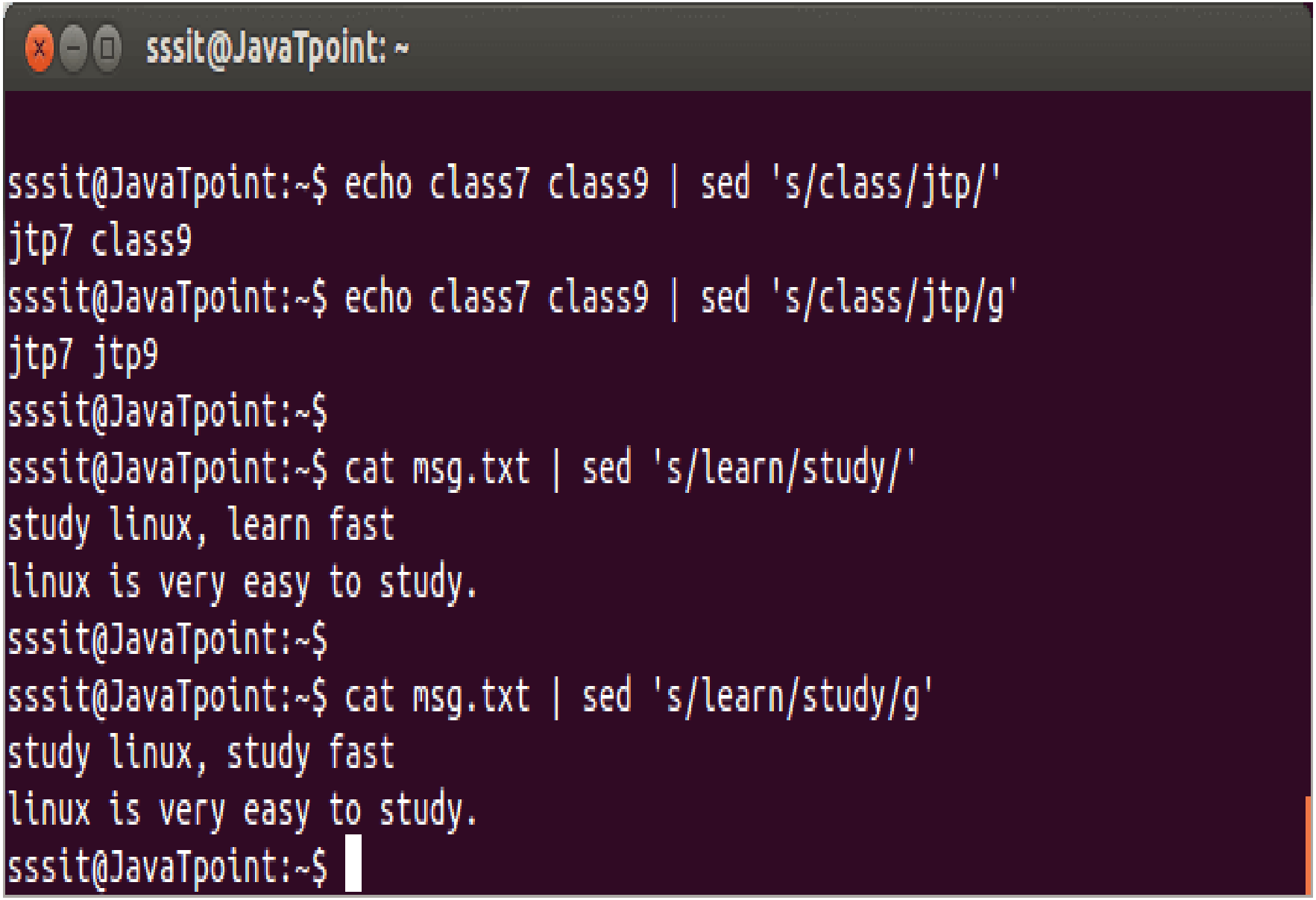
- To edit every word we have to use a **global** replacement **'g'**. It will edit all the specified word in a file or string.

**Syntax:**

- command | sed 's/**<oldWord>**/**<newWord>**/

**Example:**

- echo class7 class9 | sed 's/class/jtp/g'
- cat msg.txt | sed 's/learn/study/g'


sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ echo class7 class9 | sed 's/class/jtp/'
jtp7 class9
sssit@JavaTpoint:~$ echo class7 class9 | sed 's/class/jtp/g'
jtp7 jtp9
sssit@JavaTpoint:~$
sssit@JavaTpoint:~$ cat msg.txt | sed 's/learn/study/'
study linux, learn fast
linux is very easy to study.
sssit@JavaTpoint:~$
sssit@JavaTpoint:~$ cat msg.txt | sed 's/learn/study/g'
study linux, study fast
linux is very easy to study.
sssit@JavaTpoint:~$

**Removing a Line:-**

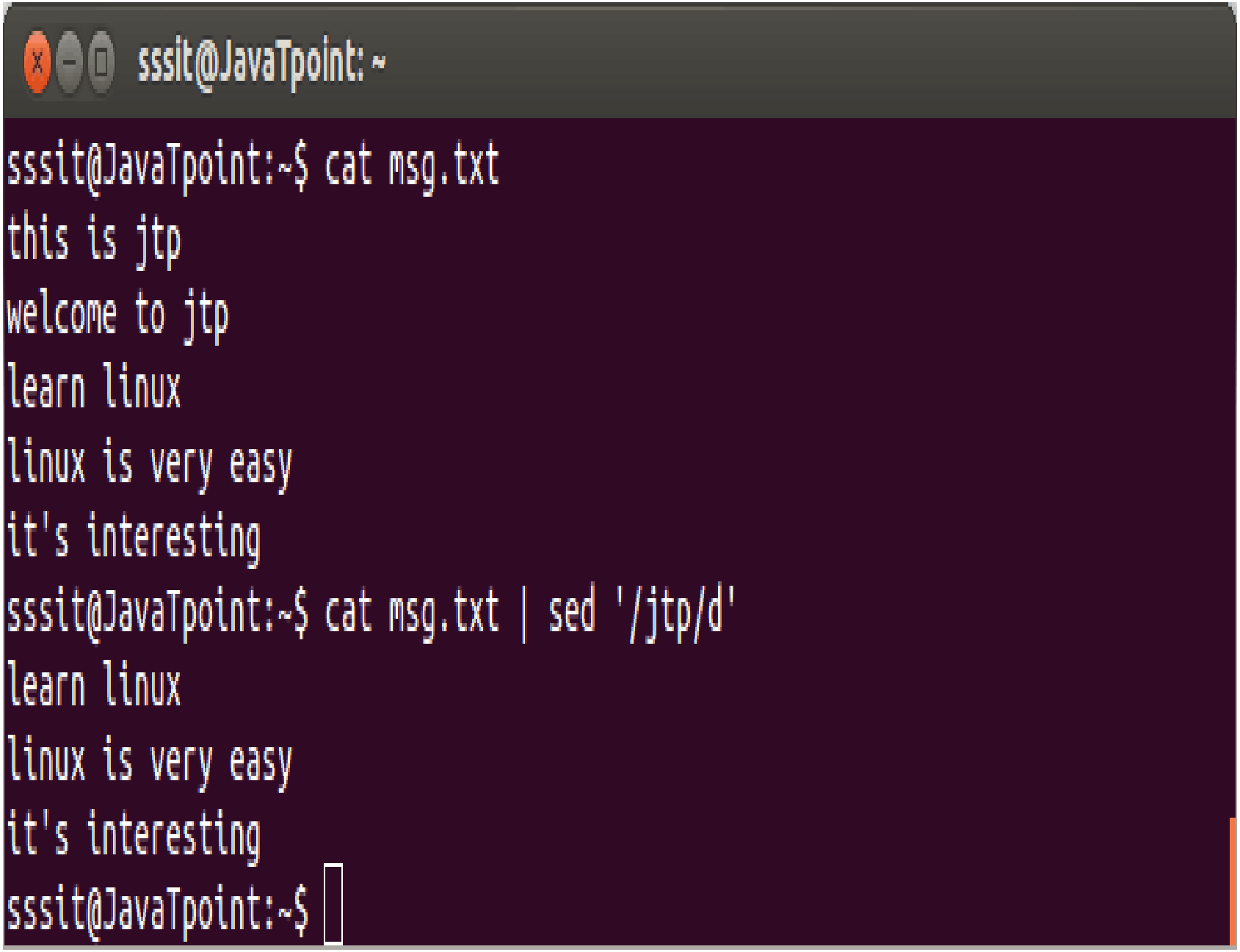
- The 'd' option will let you to remove a complete line from a file.
- You only need to specify a word from that line with 'd' option and that line will be deleted.
- But please note that all the lines having that same word will be deleted.

**Syntax:**

- cat **<fileName>** | sed '/**<Word>**/d'

**Example:**

- cat msg.txt | sed '/jtp/d'


sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ cat msg.txt
this is jtp
welcome to jtp
learn linux
linux is very easy
it's interesting
sssit@JavaTpoint:~$ cat msg.txt | sed '/jtp/d'
learn linux
linux is very easy
it's interesting
sssit@JavaTpoint:~$

# 3. DISPLAY FILE (more)

- As 'cat' command displays the file content. Same way 'more' command also displays the content of a file.
- Only difference is that, in case of larger files, 'cat' command output will scroll off your screen while 'more' command displays output one screen ful at a time.

**Syntax:** more **<file** name**>**

**Example:** more /var/log/udev

# Options

| Option | Explanation |
|---|---|
| -c | Clears screen before displaying |
| -d | Displays error messages |
| -f | Does not screen wrap long lines. |
| -l | Ignores form feed characters. |
| -r | Displays control characters in format ^C |
| -s | Squeezes multiple blank lines (leaving only one blank line in output) |
| -u | Suppresses text underlining |
| -w | Waits at end of output for user to enter any key to continue |
| -lines | Sets the number of lines in a screen (default is screen size -2) |
| +nmbr | Starts output at the indicated line number (nmbr) |
| +/ptrn | Locates first occurrence of pattern (ptrn) and starts output two lines before it. |

- If there is more than one screen of data, more displays one screen, less two lines.

- At the bottom of the screen, it displays the message “- - -more- - - (dd%)”.

- This message indicates that there are ore lines in the file and how much has been displayed so far.
- To display the next screen, key the space bar.

# 4. Print file

- The most common print utility is line printer (lpr).

# Operations Common to Both

- The operations that are common to both directories and regular files are
  1. copy (cp)
  2. move (mv)
  3. rename (mv)
  4. link (ln)
  5. delete (rm)
  6. find (find)

# 1. Copy (cp) command

- The copy (cp) utility creates a duplicate of a file, a set of files, or a directory.

- If the source is a file , the new file contains an exact copy of the data in the source file.

- If the source is a directory, all of the files in the directory are copied to the destination, which must be a directory.

- If the destination file already exists, its contents are replaced by the source file contents.

- The cp command copies both text and binary files.

**Syntax:-**

- cp **<existing** file name**> <new** file name**>**

# cp command Option

- The copy command has three options:

  preserve attributes (-p)

  interactive (-i)

  recursion (-r)

- **Preserve Attributes Option:-**

  When the destination file exists, its permissions, owner and group are used rather than the source file attributes.

- We can force the permissions, owner and group to be changed, however by using the preserve (-p) option.

  **Example:-**

- $ cp –p file1 file2

- **Interactive Option:-**

  We can guard against a file being accidentally deleted by a copy command by using the interactive (-i) option.

- When the interactive option is specified, copy asks if we want to delete an existing file.
- If we reply y or yes, the file is replaced. If we reply n or no, the copy is cancelled.

**Example:-**

- $cp –i file1 file2

**Recursive copy:-**

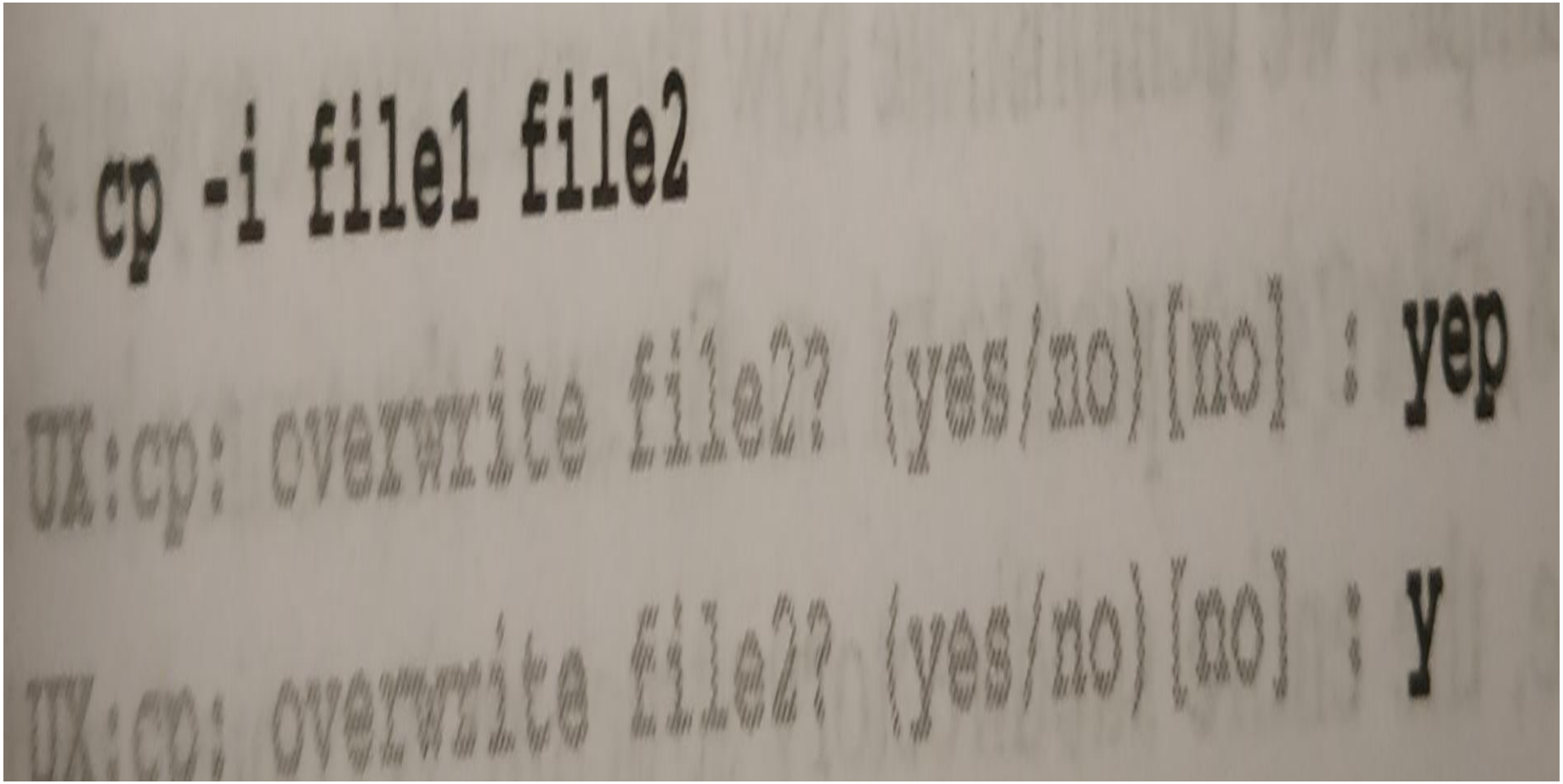
- Another way we can copy a collection of files is with the recursive (-r) copy.

- The recursive copy copies the whole directory and all of its subdirectories to a new directory.

  **Example:-**

- $cp –r DirA DirB

```
$ ls -l
total 2
-rw-r--r--    1 gilberg  staff         120 May 25 15:46 file1
-rw-------    1 gilberg  staff         120 May 27 11:03 file2

$ cp -p file1 file2

$ ls -l
total 2
-rw-r--r--    1 gilberg  staff         120 May 25 15:46 file1
-rw-r--r--    1 gilberg  staff         120 May 25 15:46 file2
```


$ cp -i file1 file2
UX:cp: overwrite file2? (yes/no)[no] : yep
UX:cp: overwrite file2? (yes/no)[no] : y

```
$ ls
DirA
$ ls DirB
Cannot access DirB: No such file or directory
$ cp -r DirA DirB

$ ls
DirA   DirB

$ ls DirB
file1  file2
```

# 2. MOVE (mv) Command

- The move (mv) command is used to move either an individual file, a list of files, or a directory.

- After a move, the old file name is gone and the new file name is found at the destination.

- This is the difference between a move and a copy.

- After a copy, the file is physically duplicated, it exists in two places.
- **Syntax:-**

  mv <source file> <destination file>

- The first argument is the name of the file to be moved.
- The second argument is its destination or, in the case of a rename, its new name.

# mv Options

- Move has only two options:

  Interactive (-i)

  Force (-f).
- **Interactive:-** if the destination file already exists, its contents are destroyed unless we use the interactive flag (-i) to request that move warn us.
- **Syntax:-** $mv –i file1 mvDir

```
$ ls
b.txt   c.txt   d.txt   geek.txt

$ cat geek.txt
India

$ cat b.txt
geeksforgeeks

$ mv -i geek.txt b.txt
mv: overwrite 'b.txt'? y

$ ls
b.txt   c.txt   d.txt

$ cat b.txt
India
```

- **Force:-** When we are not allowed to write a file, we are asked if we want to destroy the file or not.

- If we are sure that we want to write it, even if it already exists, we can skip the interactive message with the force (-f) option.

- **Syntax:-** $mv –f file1 mvDir

```
$ ls
b.txt  c.txt  d.txt  geek.txt

$ cat b.txt
geeksforgeeks

$ ls -l b.txt
-r--r--r--+ 1 User User 13 Jan  9 13:37 b.txt

$ mv geek.txt b.txt
mv: replace 'b.txt', overriding mode 0444 (r--r--r--)?

$ ls
b.txt  c.txt  d.txt  geek.txt

$ mv -f geek.txt b.txt

$ ls
b.txt  c.txt  d.txt

$ cat b.txt
India
```

# 3. Rename (mv) Command

- UNIX does not have a specific rename command.

- Recall that the move (mv) command with a new name (second argument) renames the file.

# 4. Link (ln) command

- The ln command is used to create links between files.

- A link in UNIX is a pointer to a file. Like pointers in any programming languages, links in UNIX are pointers pointing to a file or a directory

- Creating links is a kind of shortcuts to access a file. Links allow more than one file name to refer to the same file, elsewhere.

- There are two types of links :

  Soft Link or Symbolic links

  Hard Links

- These links behave differently when the source of the link (what is being linked to) is moved or removed.

- Symbolic links are not updated (they merely contain a string which is the pathname of its target).

- Hard links always refer to the source, even if moved or removed.

- For example, if we have a file a.txt. If we create a hard link to the file and then delete the file, we can still access the file using hard link.

- But if we create a soft link of the file and then delete the file, we can't access the file through soft link and soft link becomes dangling.

- Basically hard link increases reference count of a location while soft links work as a shortcut (like in Windows)

**Syntax:-**

- ln [options] source destination

**Example:-**

- $ln file1 file2

# ln Options

- Link has three options:

  Symbolic.

  Interactive.

  Force.

- **Symbolic:-** The default link type is hard. To create a symbolic link, the symbolic option (-s) is used.

**Example:-**

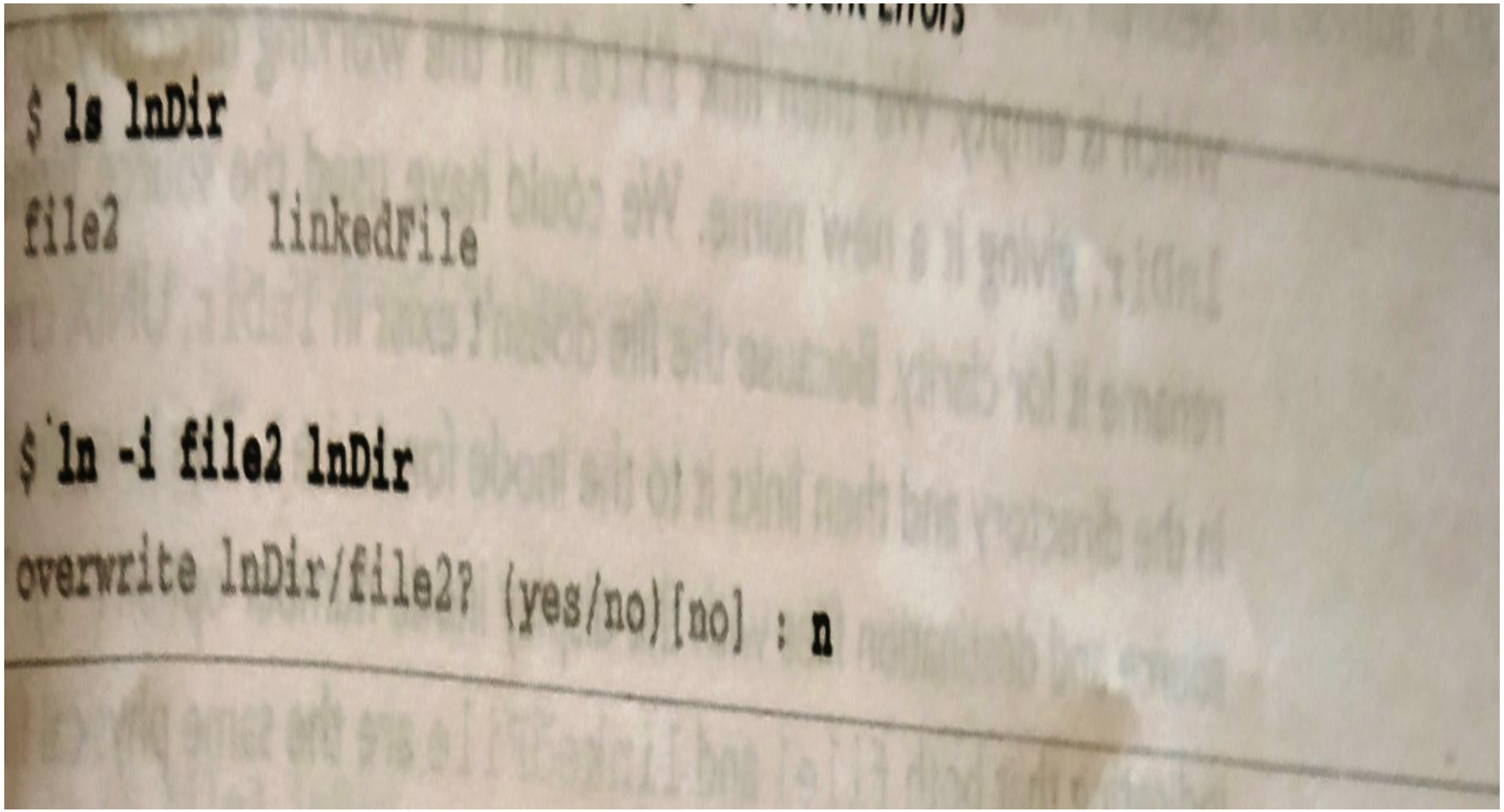
- $ln –s file2 1nDir

- **Interactive :-** If the destination file already exists, its contents are destroyed unless we request to be warned by using the interactive flag (-i).

- When the interactive flag is on link asks if we want to destroy the existing file.

**Example:-**

- $ln –i file2 1nDir



- **Force:-** When we are about to overwrite a file, we are asked if we want to destroy the file or not.

- If we are sure that we want to write it, even if it already exists, we can skip the interactive message with the force (-f) option.

**Example:-**

- $ln –f file2 1nDir

```
$ ls -l lnDir
total 2
lrwxr-xr-x    1 gilberg  staff          5 May 28 15:39 file2 -> file2
-rw-r--r--    2 gilberg  staff        120 May 25 15:10 linkedFile

$ ln -f file2 lnDir

$ ls -l lnDir
total 2
-r--r--r--    2 gilberg  staff        120 May 25 17:08 file2
-rw-r--r--    2 gilberg  staff        120 May 25 15:10 linkedFile
```

# 5. Remove (rm) Command

- rm stands for **remove.**

- rm command is used to remove objects such as files, directories, symbolic links and so on from the file system like UNIX.

- rm removes references to objects from the file system, where those objects might have had multiple references.

- **By default, it does not remove directories.**

- This command normally works silently and you should be very careful while running **rm** command because once you delete the files then you are not able to recover the contents of files and directories.

  **Syntax:**

- **rm [OPTION]... FILE...**

```
$ ls
a.txt   b.txt   c.txt   d.txt   e.txt
```

```
Removing one file at a time
$ rm a.txt

$ ls
b.txt   c.txt   d.txt   e.txt

Removing more than one file at a time
$ rm b.txt c.txt

$ ls
d.txt   e.txt
```

# rm Options

1. -i (Interactive Deletion):-

   Like in cp, the -i option makes the command ask the user for confirmation before removing each file, you have to press **y** for confirm deletion, any other key leaves the file un-deleted.

- Example:-
- $rm  -i d.txt

```
$ rm -i d.txt
rm: remove regular empty file 'd.txt'? y


$ ls
e.txt
```

2. -f (Force Deletion):-

   **rm** prompts for confirmation removal if a file is **write protected**.

   The **-f** option overrides this minor protection and removes the file forcefully.

   **Example:-**

   $rm –f e.txt

```
$ ls -l
total 0
-r--r--r--+ 1 User User 0 Jan  2 22:56 e.txt

$ rm e.txt
rm: remove write-protected regular empty file 'e.txt

$ ls
e.txt

$ rm -f e.txt

$ ls
```

3. -r (Recursive Deletion):-

- With **-r(or -R)** option rm command performs a tree-walk and will delete all the files and sub-directories recursively of the parent directory.

- At each stage it deletes everything it finds.

- Normally, **rm** wouldn’t delete the directories but when used with this option, it will delete.

```
$ ls
A

$ cd A

$ ls
B   C

$ ls B
a.txt   b.txt

$ ls C
c.txt   d.txt
```

```
$ rm *
rm: cannot remove 'B': Is a directory
rm: cannot remove 'C': Is a directory

$ rm -r *

$ ls
```

# 6. find Command

- The **find** command in UNIX is a command line utility for walking a file hierarchy.

- It can be used to find files and directories and perform subsequent operations on them.

- It supports searching by file, folder, name, creation date, modification date, owner and permissions.

**Syntax:-**

- find [paths] [expression]

- Its first argument is the path that we want to search, usually from our home directory.

- The second argument is the criterion that find needs to complete its search.

**Example:-**

- $find DirC –name file3 –print

- Find and print the absolute pathname of a file.

- Assume that we are doing our monthly file backup and want to know all files that were changed in the last 30 days.

- We can use the find command to list all files whose modification date (mtime) is within the last 30 days.

  **Example:-**

- $find DirC –type f –mtime -30

```
$ find DirC -type f -mtime -30
DirC/file1
DirC/DirC1/file2
DirC/DirC2/file3
```

# Complete List of find Criteria

| Criteria | Matches ... |
|---|---|
| -name file | filename |
| -perm nnn | permissions to nnn. nnn must be an octal number. |
| -perm -nnn | permissions to bit mask, nnn. If bit mask contains 1, permission matches if it is on. |
| -type c | file type. Valid file types are: block (b), character (c), directory (d), link (l), pipe (p), file (f), socket (s) |

| Criteria | Matches ... |
|---|---|
| -link n | number of links for a file |
| -user uname | user name. Numeric user id can also be used. |
| -nouser | no name in the `/etc/passwd` file |
| -group gname | group name |
| -nogroup | no group name in the `/etc/group` file |
| -size n \| nc | exactly file size in blocks (`n`) or characters (`nc`) |
| -inum n | the inode number |
| -atime +n \| -n | file that has been accessed `n` or more days ago (`+n`) or in the last n days (`-n`). Note that find changes the access time. |
| -mtime +n \| -n | file that has been modified `n` or more days ago (`+n`) or in the last n days (`-n`). |
| -ctime +n \| -n | file that has been changed `n` or more days ago (`+n`) or in the last n days (`-n`). |
| -newer file | file's modification date is later than file's date |
| -anewer file | file's access date is later than file's date |
| -cnewer file | file's change date is later than file's date |
| -print | Displays absolute pathname on standard output |
| -exec | Executes specified command |
| -ok | Same as execute except asks for yes/no confirmation on action |
| -depth | Processes files in a directory before the file directory itself |
| -prune | Terminates the examination of files in the current directory and its subdirectories once a matching file is found. |
| -follow | When a matching file is symbolically linked, uses linked file to match criteria. |

# Security and File Permission

- The security system in UNIX, like any other operating system, is designed to control the access to resources.

**User and Groups:-**

- In UNIX, everyone who logs on to the system is called **a user**. Users are known to the system by their user ids.

- In UNIX, not every user is created equal.

- Some users have more capabilities than others. These users are known as **super users**. Also known as system **administrators**.

- Super Users need to have a lot of experience and a lot of training.

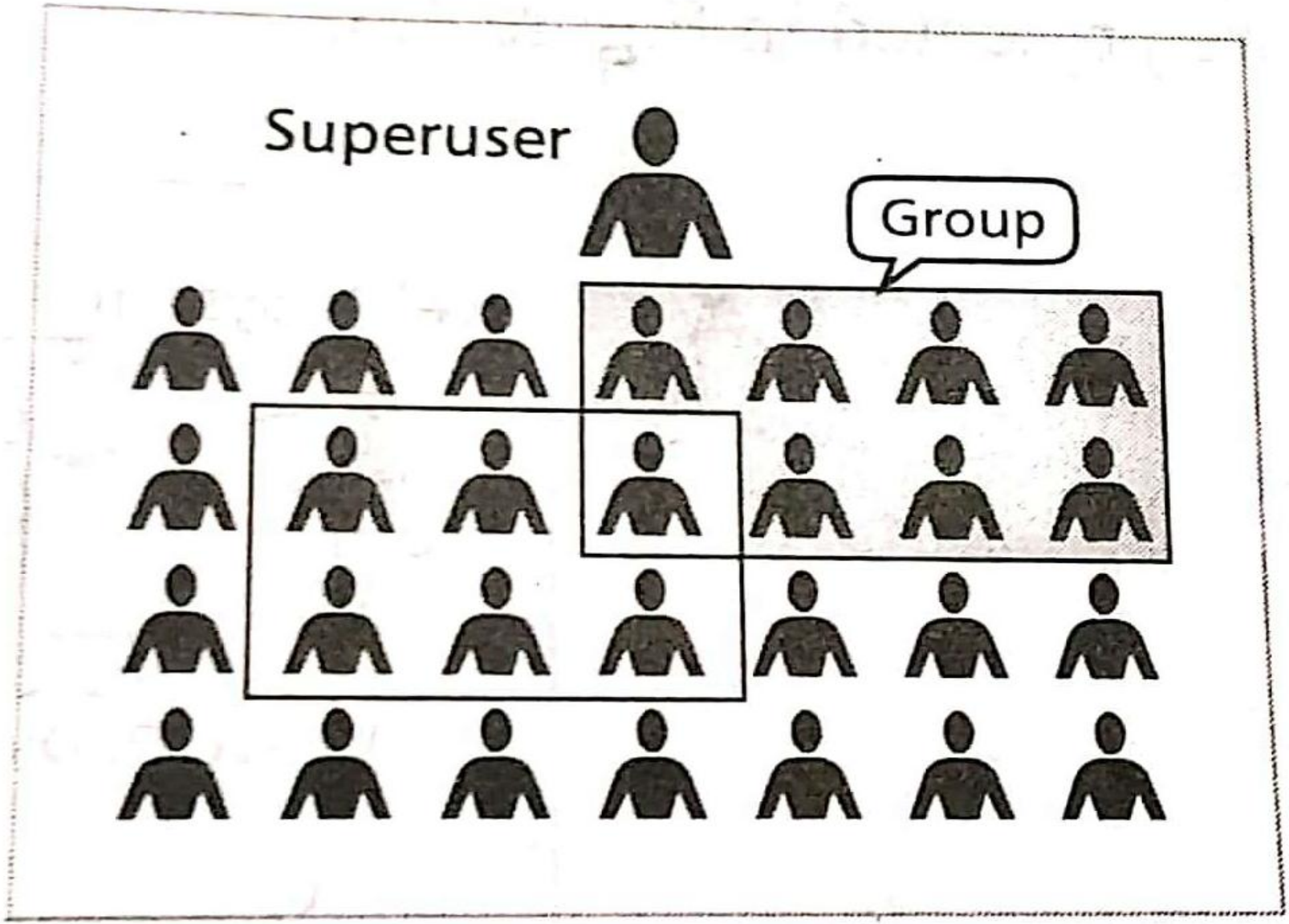



FIGURE 4.1 *Users*

**Groups:-**

- Users can be organized into groups. A team working on a project, for example, needs to share many of the same file.

- Users can belong to multiple groups.

# groups Command

- Unix provides a command, groups, to determine a user's group.

- You can check your group or any other user's group.

- If you enter the command with no user id, the system responds with your group.

- If a user belongs to multiple user groups, all of them will be listed.

  **Syntax:-** groups [options] userid

  **Example:-** $groups

# Security Levels

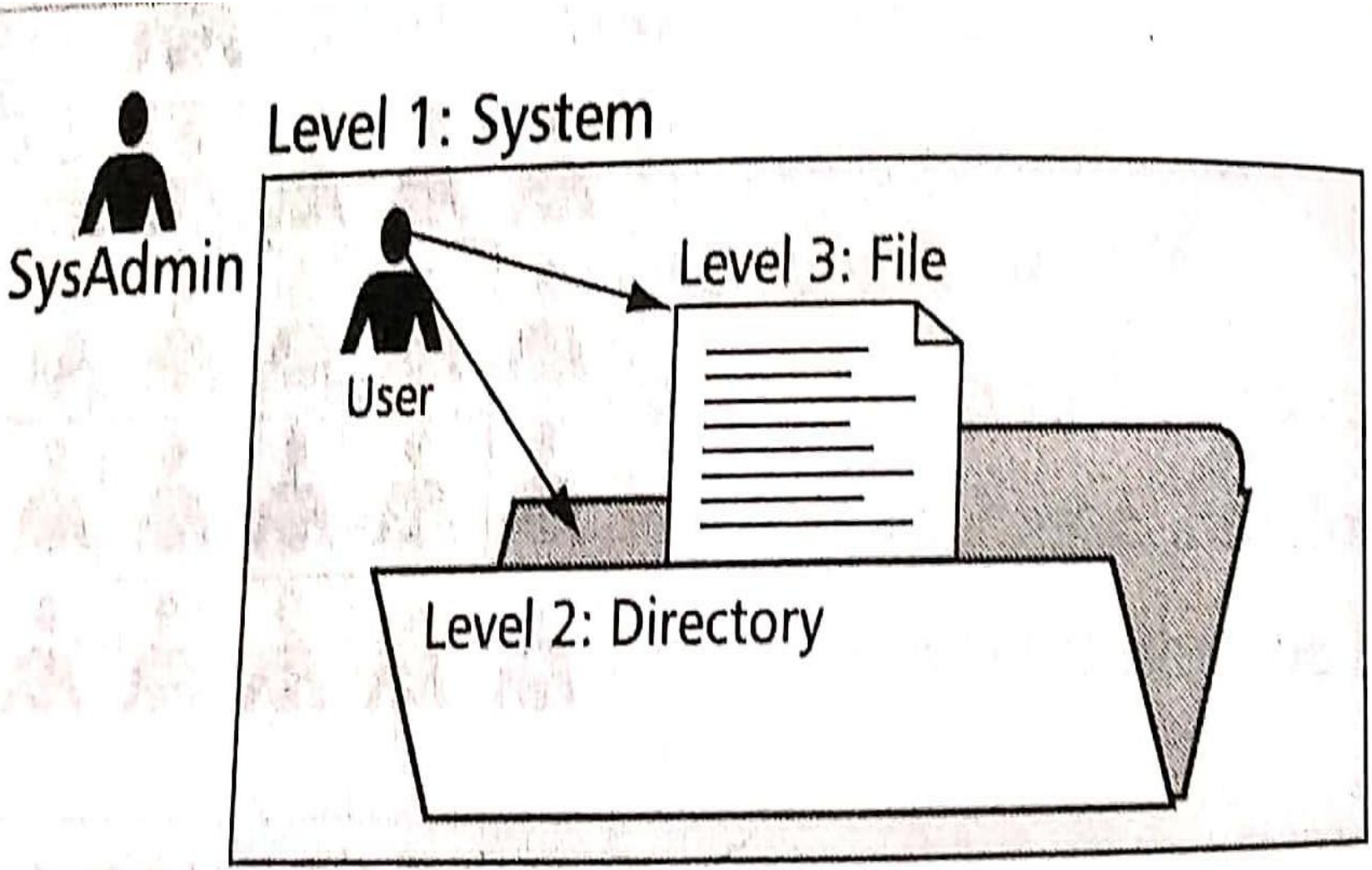
- There are three levels of security in UNIX:

  System.

  Directory.

  File.
- The system security is controlled by the system administrator, a super user.

- The directory and file securities are controlled by the users who own them.

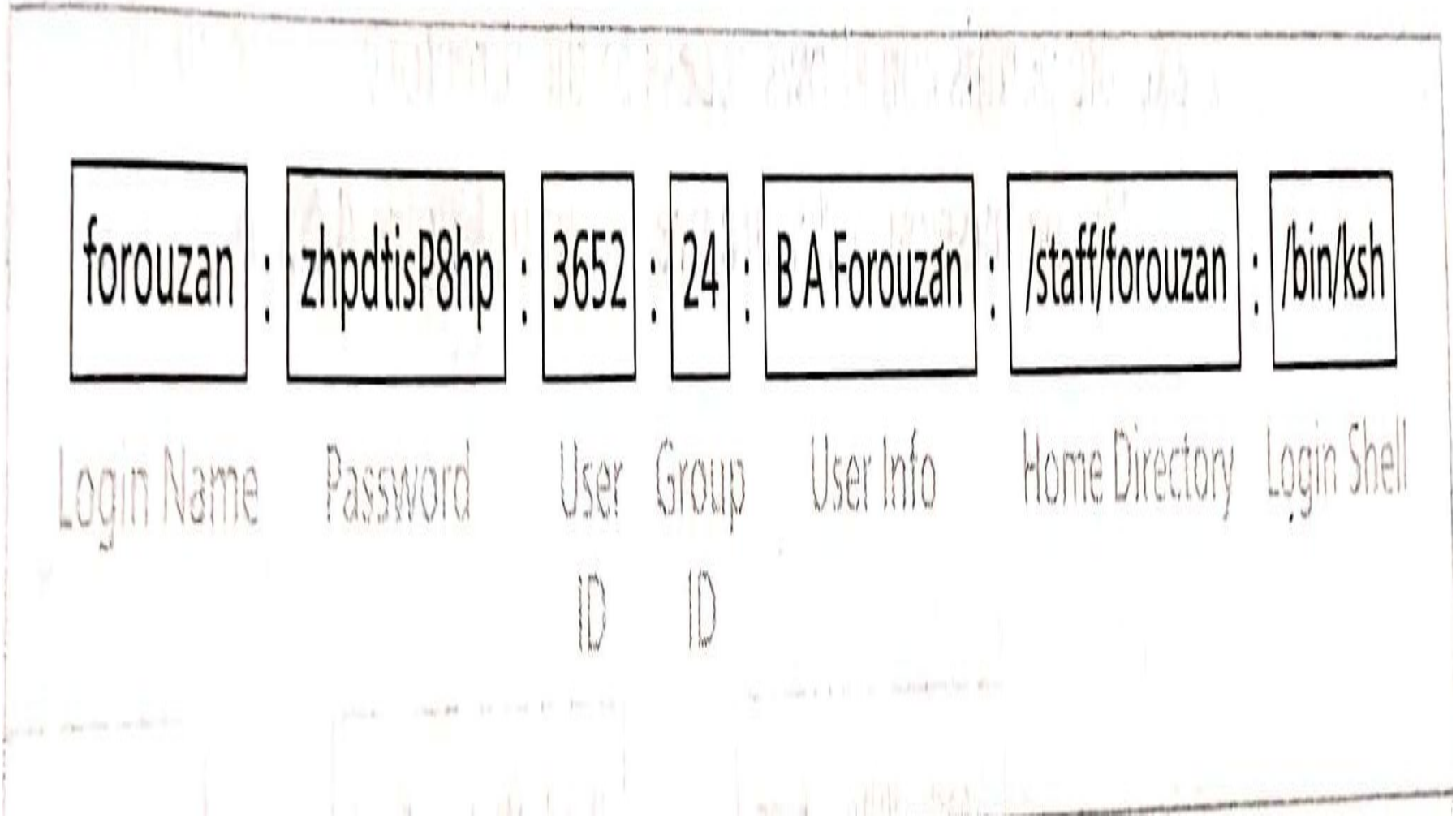

Level 1: System
SysAdmin
User
Level 3: File
Level 2: Directory

# System Security

- System security controls who is allowed to access the system. It begins with your login id and password.

- When the system administrator opens an account for you, he or she creates an entry in the system password file.

- The contents of an entry in our password file.



- **Home directory:-** the login or home directory when you first log into the system. It is represented as the absolute pathname for your home directory.

- **Login Shell:-** identifies the shell that is loaded when you login.

# Permission Codes

- Both the directory and file security levels use a set of permission codes to determine who can access and manipulate a directory or file.

- The permission codes are divided into three sets of codes.

- The **first set** contains the permissions of the **owner** of the directory or file.

- The **second set** contains the group permissions for members in a **group** as identified by the group id.

- The **third set** contains the permissions for everyone else that is, the general **public**.

- The code for each set is a triplet representing read (r), write (w) and execute (x).

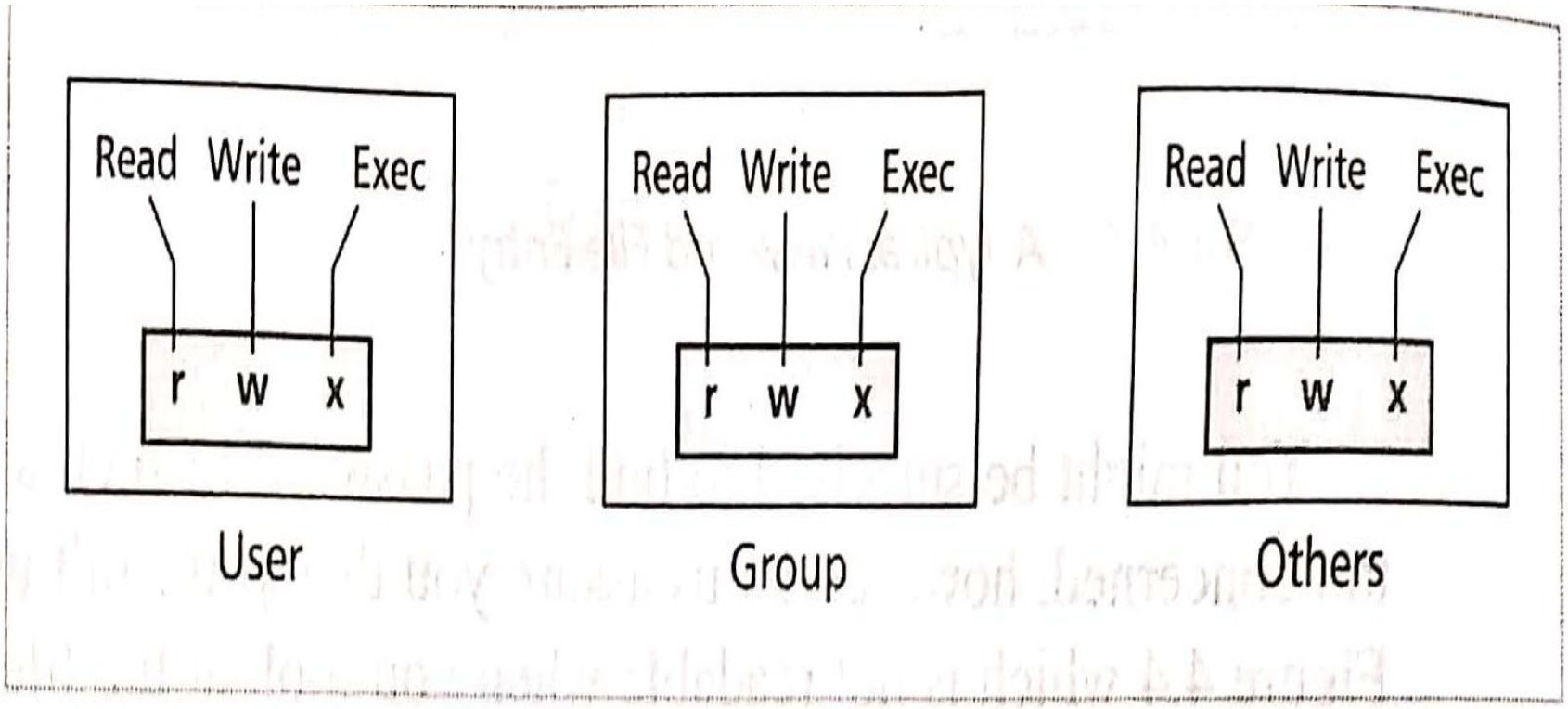



FIGURE 4.5 *Directory and File Permissions*

TABLE 4.1 *Summary of Permission Rules*

| Permission | read (r) | write (w) | execute (x) |
|---|---|---|---|
| **Directory** | Read contents of directory | Add or delete entries (files) in directory using commands | Reference or move to directory |
| **File Level** | Read or copy files in directory | Change or delete files | Run executable files |

# Directory level permissions

1. Read permission.
2. Write permission.
3. Execute permission.

# File Level Permission

1. Read permission.
2. Write Permission.
3. Execute Permission.

# Checking Permission

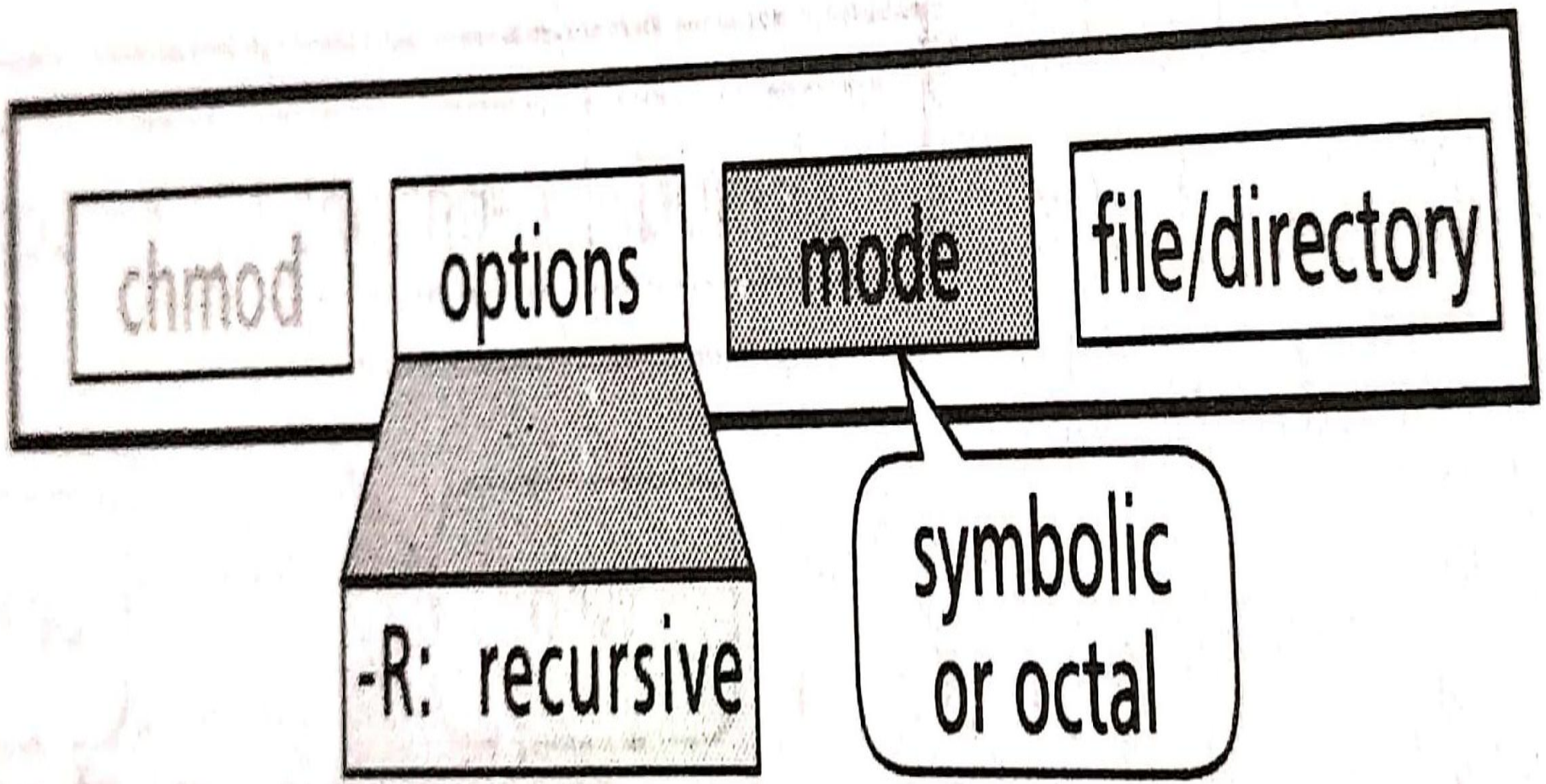
- To check the permissions of a file or directory, we use the long list command (ls -l).

# Changing Permission (chmod)

- When a directory or a file is created, the system automatically assigns default permissions.

- The owner of the directory or file can change them. To change the permissions, we use the chmod command.

**Syntax:-**


chmod
options
mode
file/directory
-R: recursive
symbolic
or octal

- There are two ways to change the permissions:

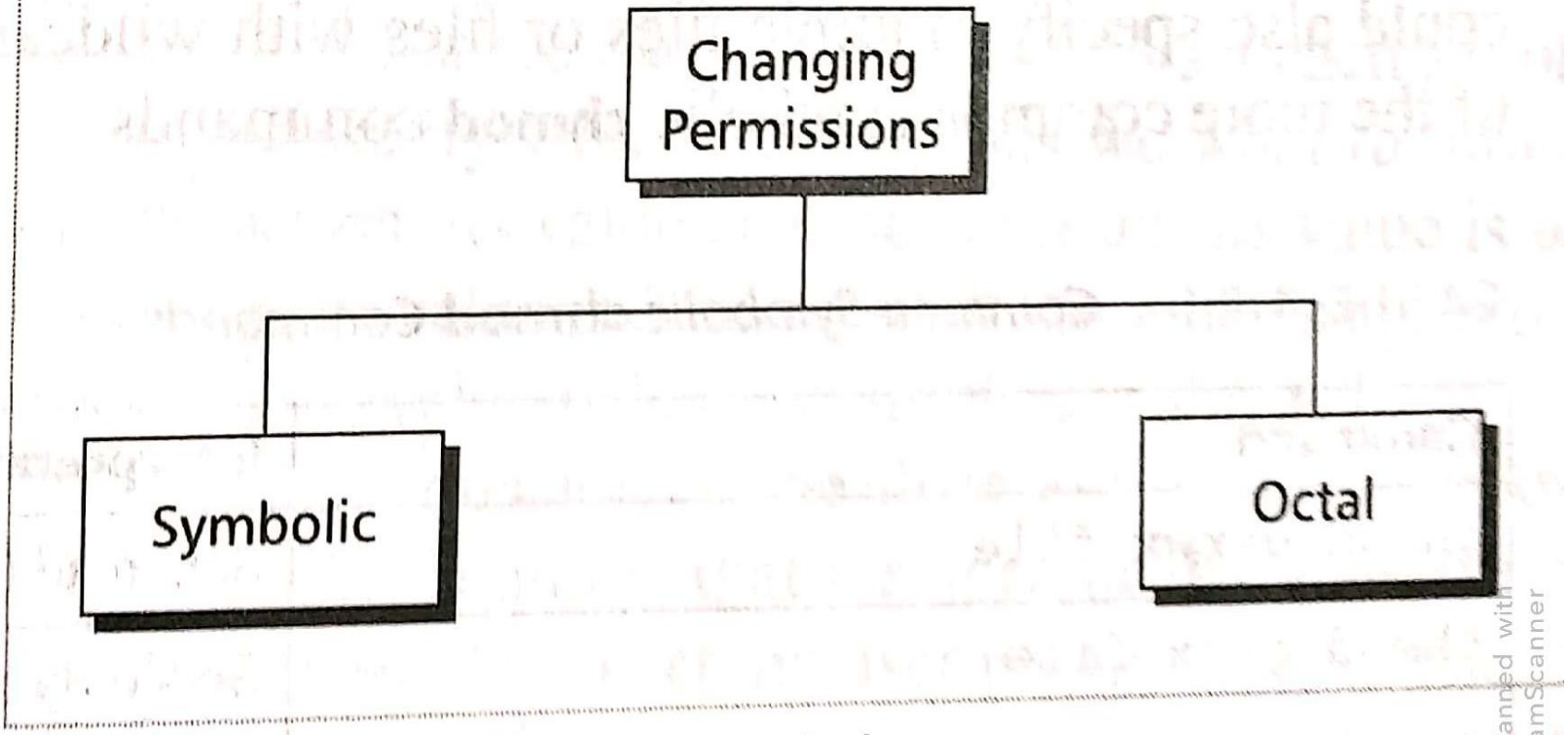
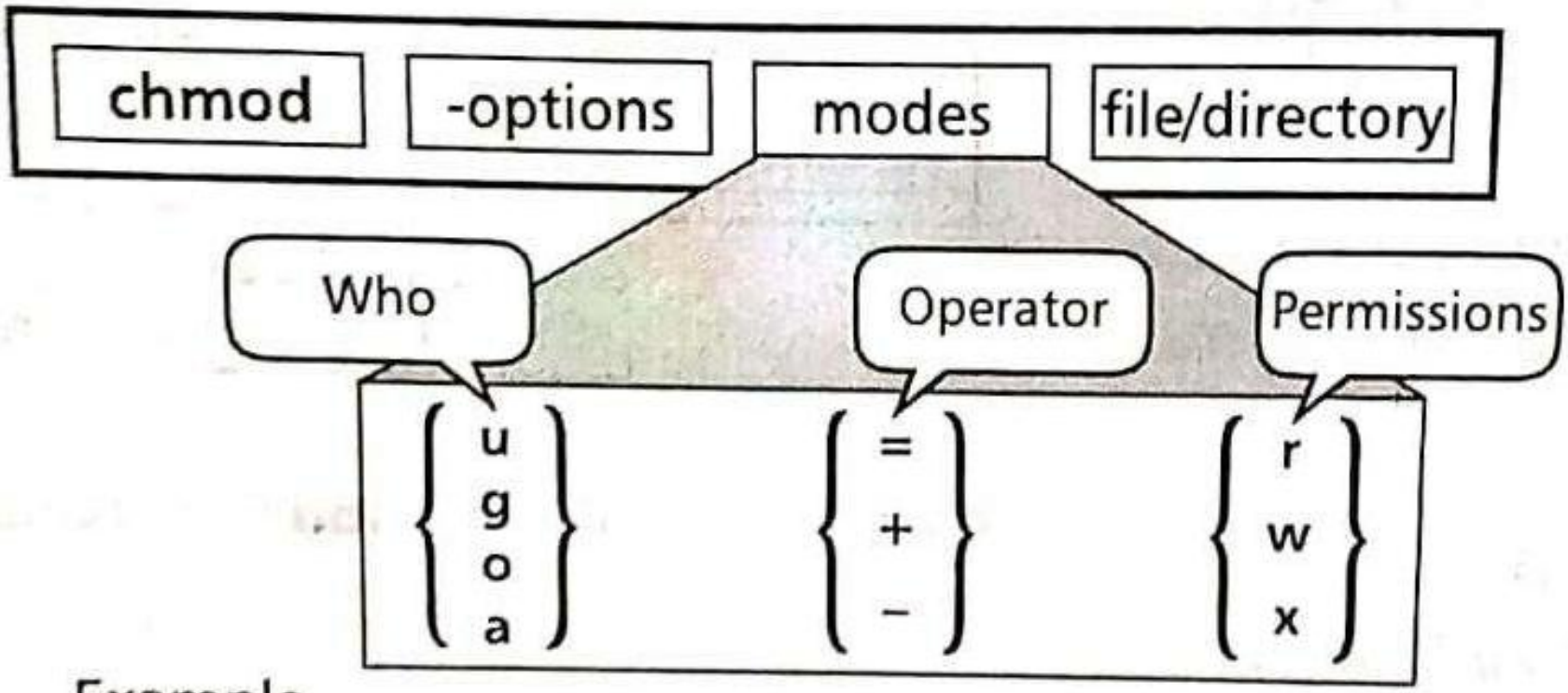



FIGURE 4.7 *Changing Permissions*

# Symbolic Codes:-




Example

```
chmod   u=rwx,g+w,o-w   memo.doc
```

TABLE 4.2 *Common Symbolic* **chmod** *Commands*

| Command | Interpretation |
|---|---|
| `chmod u=rwx file` | Sets read (**r**), write (**w**), execute (**x**) for user. |
| `chmod g=rx file` | Sets only read (**r**) and execute (**x**) for group; write (**w**) denied. |
| `chmod g+x file` | Adds execute (**x**) permission for group; read and write unchanged. |
| `chmod a+r file` | Adds read (**r**) to all users; write and execute unchanged. |
| `chmod o-w file` | Removes others' write (**w**) permission; read and execute unchanged. |

- Octal codes:-

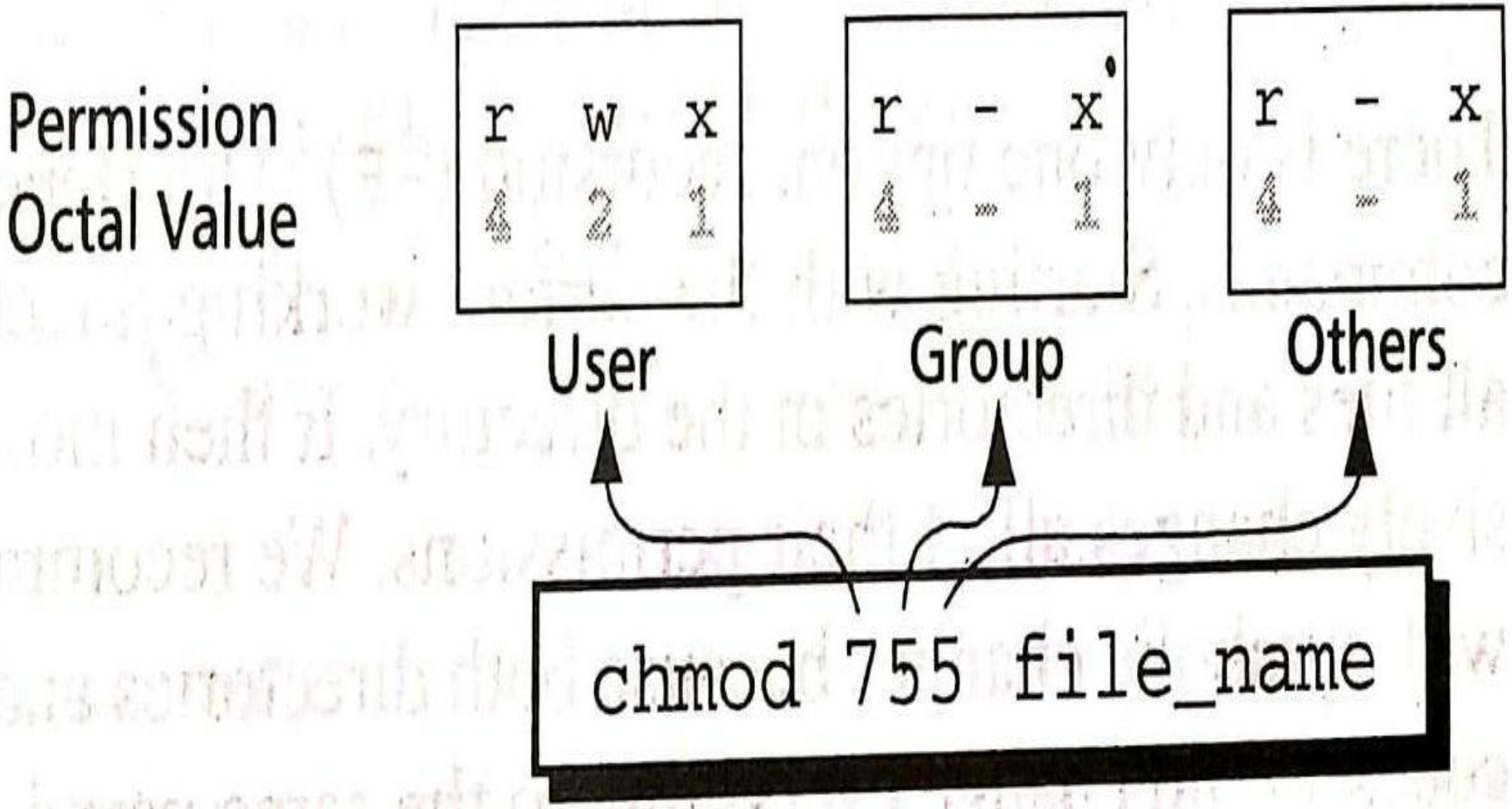

Permission
Octal Value
r w x
4 2 1
r - x
4 - 1
r - x
4 - 1
User
Group
Others
chmod 755 file_name

TABLE 4.3 Common Symbolic **chmod** Commands

| Command | Description |
|---|---|
| `chmod 777 file` | All permissions on for all three settings |
| `chmod 754 directory` | User all, group read + execute; others read only |
| `chmod 664 file` | User and group read + write, others read only |
| `chmod 644 file` | User read + write, group and others read only |
| `chmod 711 program` | User all, group and others execute only |

**Option:-**

- There is only one option, recursion (-R).

- The chmod recursion works just as in other commands. Starting with the current working directory, it changes the permissions of all files and directories in the directory.

- It then moves to the subdirectories and recursively changes all of their permissions.

```
$ ls -lR unix4sec
total 4
-rw-r--r--    1 gilberg  staff          120 Aug 30 10:36 file1
-rw-r--r--    1 gilberg  staff          120 Aug 30 10:38 file2
drwxr-xr--    2 gilberg  staff          512 Aug 30 10:39 subDirA
drwxr-xr--    2 gilberg  staff          512 Aug 30 10:39 subDirB

unix4sec/subDirA:
total 1
-rw-r--r--    1 gilberg  staff          120 Aug 30 10:39 file1A

unix4sec/subDirB:
total 1
-rw-r--r--    1 gilberg  staff          120 Aug 30 10:39 file1B

$ chmod -R o-r unix4sec

$ ls -Rl unix4sec
total 4
-rw-r-----    1 gilberg  staff          120 Aug 30 10:36 file1
-rw-r-----    1 gilberg  staff          120 Aug 30 10:38 file2
drwxr-x---    2 gilberg  staff          512 Aug 30 10:39 subDirA
drwxr-x---    2 gilberg  staff          512 Aug 30 10:39 subDirB

unix4sec/subDirA:
total 1
-rw-r-----    1 gilberg  staff          120 Aug 30 10:39 file1A

unix4sec/subDirB:
total 1
-rw-r-----    1 gilberg  staff          120 Aug 30 10:39 file1B
```

# User mask (umask) command

- The permissions are initially set for a directory or file using a three digit octal system variable, the user mask (mask).

- When a new directory or file is created, the number in the mask is used to set the default permissions.

- The default permissions are 777 for a directory and 666 for a file.

- To display the current user mask setting, use the **umask** command with no arguments.

- To set it, use the command with the new mask setting.

**Syntax:-**

- umask option code

**Example:-**

$umask

000

$umask 022

$umask

022

# Changing Ownership and Group

- Every directory and file has an owner and a group. When you create a directory or file, you are the owner and your group is the group.

- There are two commands that allow the owner and group to be changed.

- The **change ownership (chown)** command can change the owner or the owner and the group.

- The **change group (chgrp)** command can change only the group.

# Change ownership (chown)

- The owner and optionally the group are changed with the change ownership (chown) command.

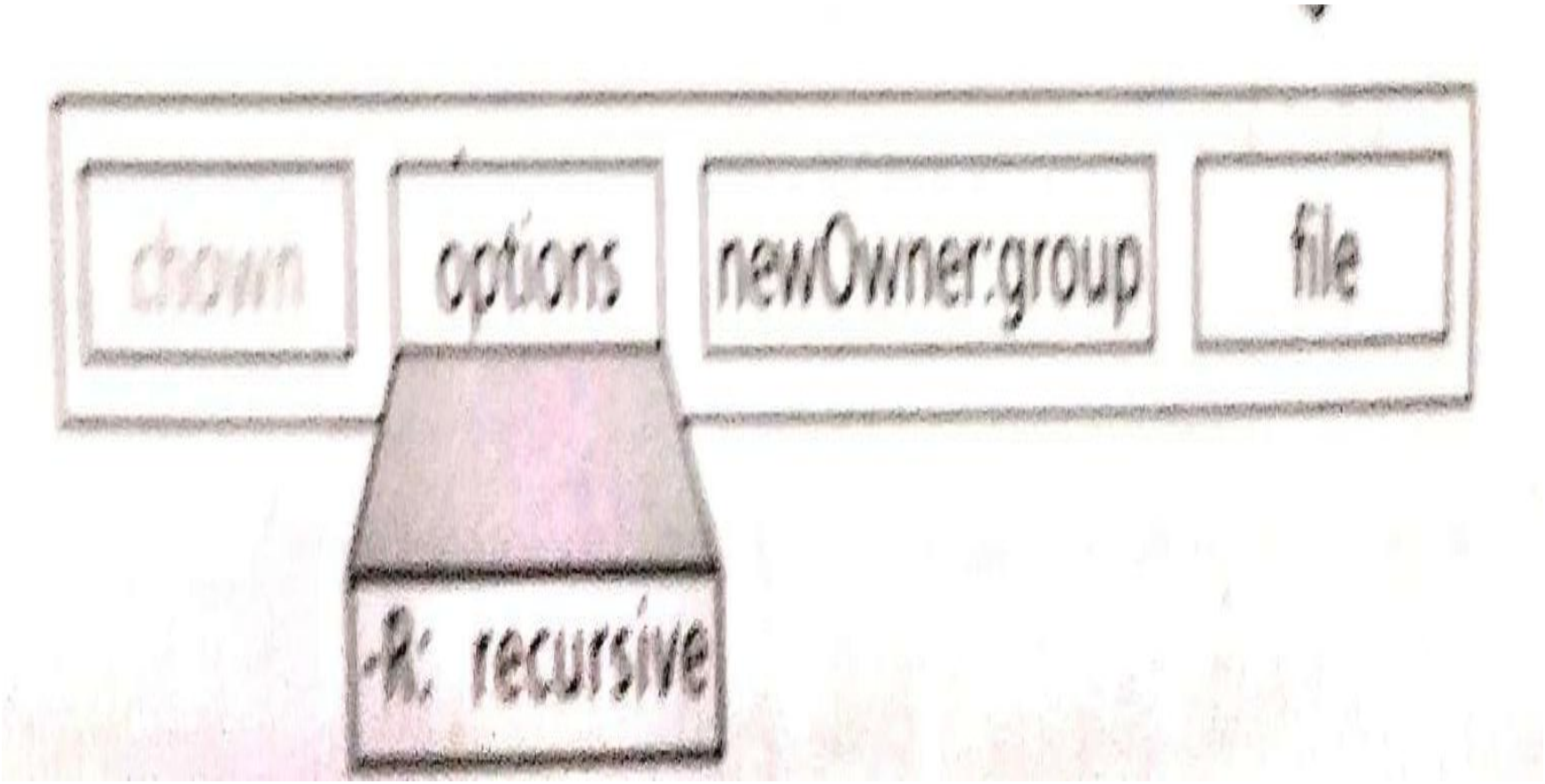
- The new owner may be a login name or a user id (UID). The group is optional.

- The group does not have to be changed when the owner is changed. Unless the new owner is not a member of the current group.

- Only the current owner or super user may change the ownership or group. This means that once the ownership is changed, the original owner cannot claim it back.

- Either the new owner or the system administrator must change it back.

# Syntax:-



**Example:-**

- $chown forouzan file1

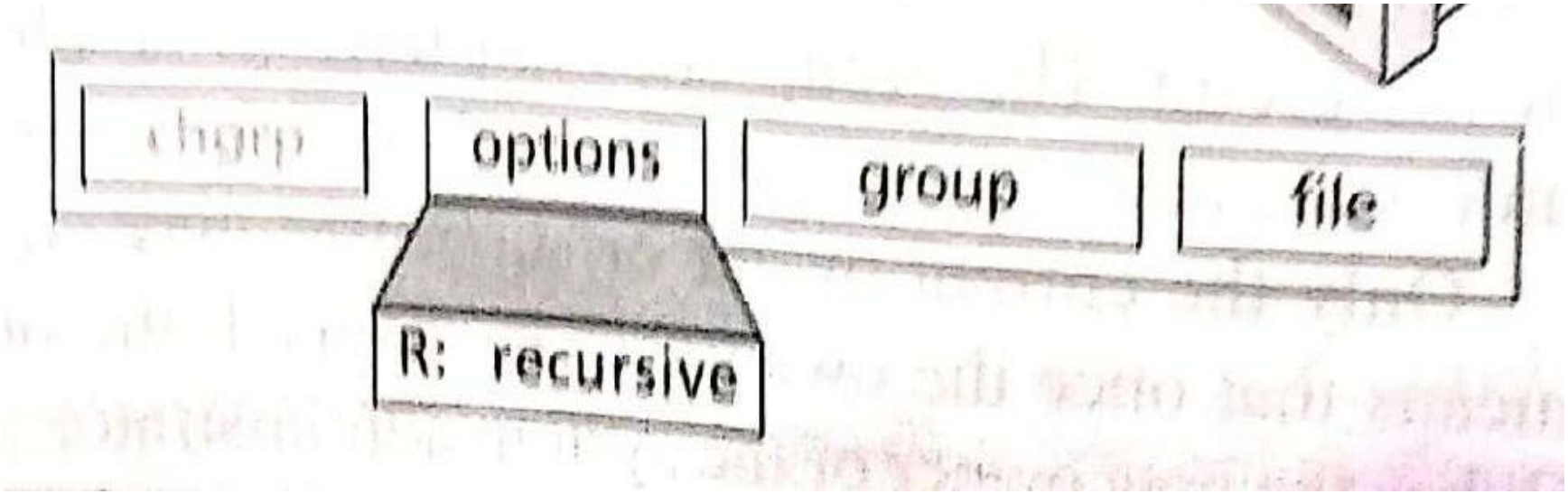
**Options:-**

- (-R):- when the recursive option is used with a directory, all files in the directory and all subdirectories and their files are changed recursively.

# Change Group (chgrp)

- To change the group without changing the owner, you use the change group (chgrp) command.

- **Syntax:-**




**Example:-**

$chgrp proj15 file2

| Command | Description | Options |
|---|---|---|
| chown | ***Synopsis:*** *chown [-option] owner [:group] list*<br>Changes the owner (and the group associated to) a list of files or directories. | R |
| groups | ***Synopsis:*** *groups [user id]*<br>Displays the user's group. | |
| umask | ***Synopsis:*** *umask [mask]*<br>Displays or sets the default permission for newly created files or directories. | |

| Command | Description | Options |
|---|---|---|
| chgrp | *Synopsis: chgrp [-option] group list*<br>Changes a group associated with a list of files or directories. | R |
| chmod | *Synopsis: chmod [-option] mode list*<br>Sets or changes the permission of a list of files or directories. | R |

# DISK UTILITIES

1. df
2. du
3. mount
4. umount

# df

- The df command, stands for **D**isk **F**ree, reports file system disk space usage.

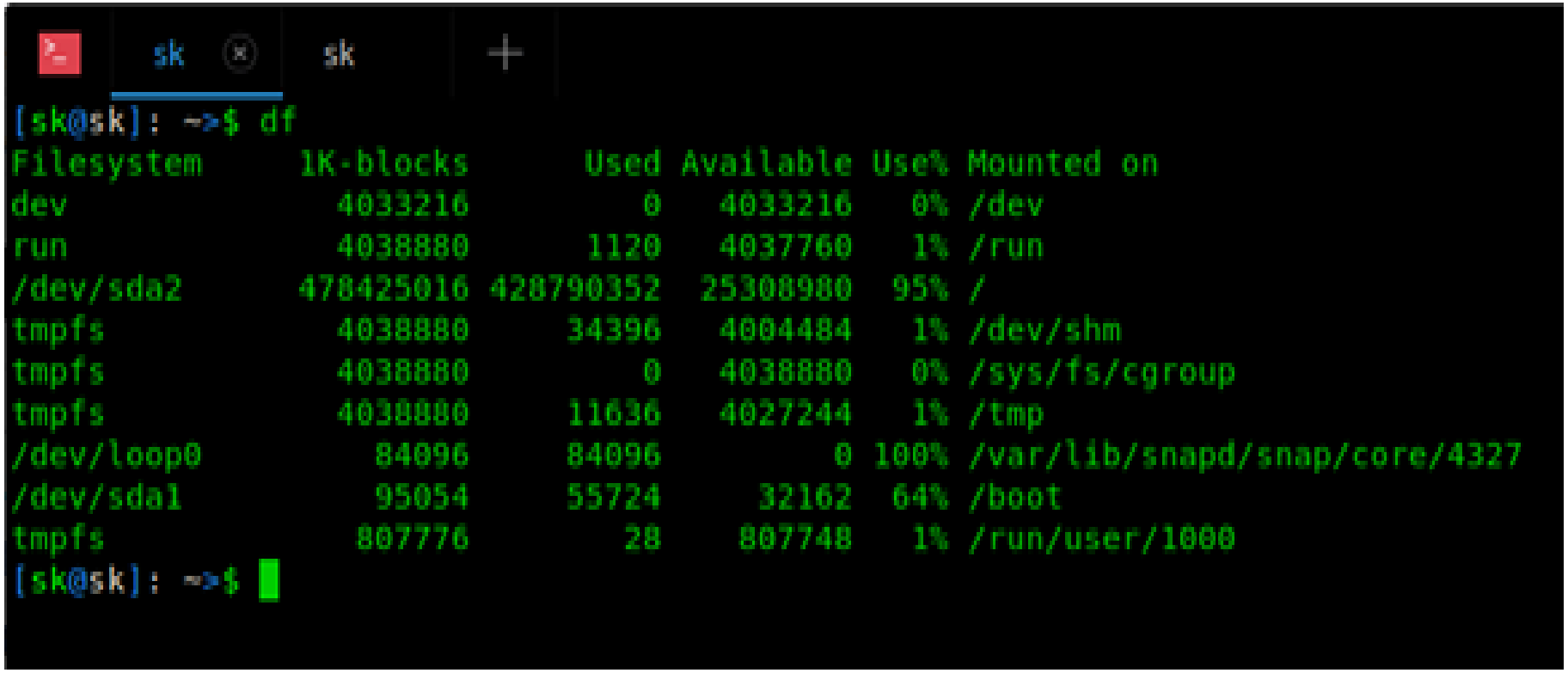
- It displays the amount of disk space available on the file system in a Linux system.

- The df command reports **how much disk space we have** (i.e free space) whereas the du command reports **how much disk space is being consumed** by the files and folders.
-

1. **View entire file system disk space usage**

   Run df command without any arguments to display the entire file system disk space.

   **Example:-** $ df



The result is divided into six columns.

- **Filesystem** – the filesystem on the system.
- **1K-blocks** – the size of the filesystem, measured in 1K blocks.
- **Used** – the amount of space used in 1K blocks.
- **Available** – the amount of available space in 1K blocks.
- **Use%** – the percentage that the filesystem is in use.
- **Mounted on** – the mount point where the filesystem is mounted.

2. **Display file system disk usage in human readable format**

   If you want to display them in human readable format, use **-h** flag.

   **Syntax:-**

   $df -h

```
Filesystem      Size  Used Avail Use% Mounted on
/dev/loop0       18G   15G  2.5G  86% /
none            4.0K     0  4.0K   0% /sys/fs/cgroup
udev            483M  4.0K  483M   1% /dev
tmpfs            99M  1.4M   97M   2% /run
none            5.0M     0  5.0M   0% /run/lock
none            492M  1.8M  490M   1% /run/shm
none            100M   20K  100M   1% /run/user
/dev/sda3       167G  157G  9.9G  95% /host
```

3. **Display disk space usage only in MB**

   To view file system disk space usage only in Megabytes, use **-m** flag.

   **Syntax:-**

   **$ df -m**

```
$ df -m
Filesystem 1M-blocks Used Available Use% M
dev 3939 0 3939 0% /dev
run 3945 2 3944 1% /run
/dev/sda2 467212 418742 24716 95% /
tmpfs 3945 26 3920 1% /dev/shm
tmpfs 3945 0 3945 0% /sys/fs/cgroup
tmpfs 3945 12 3933 1% /tmp
/dev/loop0 83 83 0 100% /var/lib/snapd/sna
/dev/sda1 93 55 32 64% /boot
tmpfs 789 1 789 1% /run/user/1000
```

4. **List inode information instead of block usage**

   We can list inode information instead of block usage by using **-i** flag.

   **Syntax:-**

   **$ df -i**

```
//using -i with df//

$df -i kt.txt
Filesystem     Inodes  IUsed    IFree Iuse% Mounted on
/dev/the2      489281     48   489233    1% /snap/core

/*showing inode info
of file system
having file kt.txt */
```

5. **Display the file system type**

   To display the file system type, use **-T** flag.

   **Syntax:-**

   **$ df -T**

```
/using -T with df//

$df -T kt.txt
Filesystem     Type      1K-blocks      Used Available U
/dev/the2      squashfs   1957124      1512   1955612

/* you can use
-T with df only
to display type of
all the mounted
file systems */
```

6. **Display only the specific file system type**

   We can limit the listing to a certain file systems. for example **ext4**. To do so, we use **-t** flag.

   **Syntax:-**

   **$ df -t ext4**

```
$ df -t ext4
Filesystem 1K-blocks Used Available Use% Mou
/dev/sda2 478425016 428790896 25308436 95% /
/dev/sda1 95054 55724 32162 64% /boot
```

7. **Exclude specific file system type**

   Some times, you may want to exclude a specific file system from the result. This can be achieved by using **-x** flag.

   **Syntax:-**

   **$ df -x ext4**

```
Filesystem 1K-blocks Used Available Use% Mounted on
dev 4033216 0 4033216 0% /dev
run 4038880 1120 4037760 1% /run
tmpfs 4038880 26116 4012764 1% /dev/shm
tmpfs 4038880 0 4038880 0% /sys/fs/cgroup
tmpfs 4038880 11984 4026896 1% /tmp
/dev/loop0 84096 84096 0 100% /var/lib/snapd/snap/core/4327
tmpfs 807776 28 807748 1% /run/user/1000
```

8. **Display usage for a folder**

   To display the disk space available and where it is mounted for a folder, for example **/home/sk/**, use this command:

   **Syntax:-**

   **$ df -hT /home/sk/**

```
$ df -hT /home/sk/
Filesystem Type Size Used Avail Use% Mounted on
/dev/sda2 ext4 457G 409G 25G 95% /
```

# du

- **du** command, short for disk usage, is used to estimate file space usage.

- The du command can be used to track the files and directories which are consuming excessive amount of space on hard disk drive.

**Syntax :**

```
du [OPTION]... [FILE]...
du [OPTION]... --files0-from=F
```

Examples :

```
du /home/mandeep/test
```

Output:

```
44    /home/mandeep/test/data
2012    /home/mandeep/test/system design
24    /home/mandeep/test/table/sample_table/tree
28    /home/mandeep/test/table/sample_table
32    /home/mandeep/test/table
100104    /home/mandeep/test
```

# Options

1. If we want to print sizes in human readable format(K, M, G), use -h option

```
du -h /home/mandeep/test

Output:
44K    /home/mandeep/test/data
2.0M    /home/mandeep/test/system  design
24K    /home/mandeep/test/table/sample_table/tree
28K    /home/mandeep/test/table/sample_table
32K    /home/mandeep/test/table
98M    /home/mandeep/test
```

2. Use -a option for printing all files including directories.

```
du -a -h /home/mandeep/test
```

Output:

```
4.0K    /home/mandeep/test/blah1-new
4.0K    /home/mandeep/test/fbtest.py
8.0K    /home/mandeep/test/data/4.txt
4.0K    /home/mandeep/test/data/7.txt
4.0K    /home/mandeep/test/data/1.txt
4.0K    /home/mandeep/test/data/3.txt
4.0K    /home/mandeep/test/data/6.txt
4.0K    /home/mandeep/test/data/2.txt
4.0K    /home/mandeep/test/data/8.txt
8.0K    /home/mandeep/test/data/5.txt
44K    /home/mandeep/test/data
4.0K    /home/mandeep/test/notifier.py
```

3. Use -c option to print total size

```
du -c -h /home/mandeep/test
```

Output:

```
44K    /home/mandeep/test/data
2.0M    /home/mandeep/test/system  design
24K    /home/mandeep/test/table/sample_table/tree
28K    /home/mandeep/test/table/sample_table
32K    /home/mandeep/test/table
98M    /home/mandeep/test
98M    total
```

4. To print sizes till particular level, use -d option with level no.

```
du -d 1 /home/mandeep/test
```

Output:

```
44    /home/mandeep/test/data
2012    /home/mandeep/test/system design
32    /home/mandeep/test/table
100104    /home/mandeep/test
```

Now try with level 2, you will get some extra directories

```
du -d 2 /home/mandeep/test
```

Output:

```
44    /home/mandeep/test/data
2012    /home/mandeep/test/system design
28    /home/mandeep/test/table/sample_table
32    /home/mandeep/test/table
100104    /home/mandeep/test
```

5. Get summary of file system using -s option

```
du -s /home/mandeep/test
```

Output:

```
100104    /home/mandeep/test
```

6. Get the timestamp of last modified using --time option

```
du --time -h /home/mandeep/test
```

Output:

```
44K   2018-01-14 22:22   /home/mandeep/test/data
2.0M   2017-12-24 23:06   /home/mandeep/test/system design
24K   2017-12-30 10:20   /home/mandeep/test/table/sample_table/tree
28K   2017-12-30 10:20   /home/mandeep/test/table/sample_table
32K   2017-12-30 10:20   /home/mandeep/test/table
98M   2018-02-02 17:32   /home/mandeep/test
```

# mount

- All files in a **Linux** filesystem are arranged in form of a big tree rooted at '**/**'.

- These files can be spread out on various devices based on your partition table, initially your parent directory is mounted(i.e attached) to this tree at '**/**', others can be mounted manually using GUI interface(if available) or using **mount** command.

- **mount** command is used to mount the filesystem found on a device to big tree structure(**Linux** filesystem) rooted at '**/**'.

- Conversely, another command **umount** can be used to detach these devices from the Tree.

## Syntax:

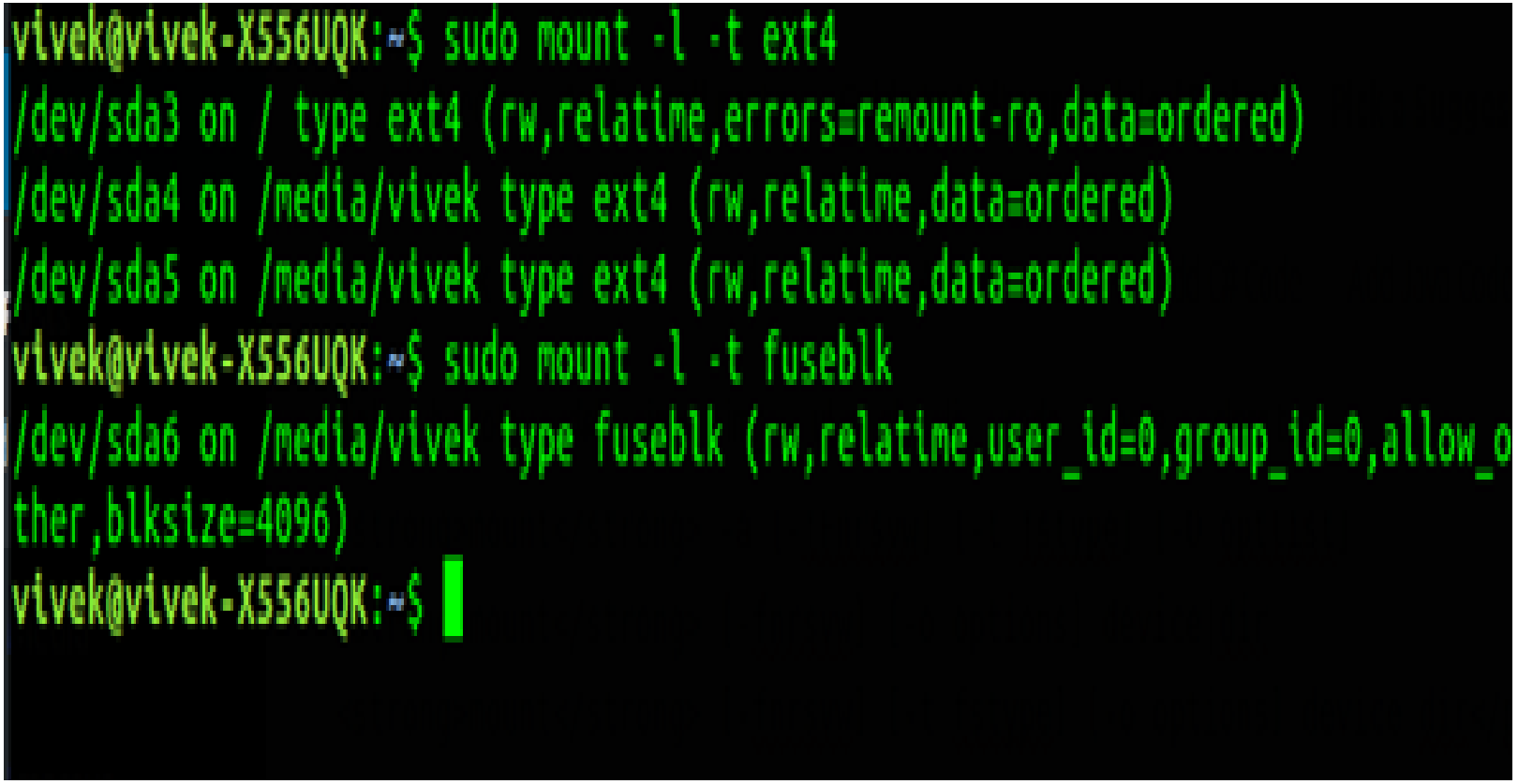
```
mount -t type device dir
```

## Other forms:

```
mount [-l|-h|-V]
mount -a [-fFnrsvw] [-t fstype] [-O optlist]
mount [-fnrsvw] [-o options] device|dir
mount [-fnrsvw] [-t fstype] [-o options] device dir
```

- **Some Important Options:**
- **l** : Lists all the file systems mounted yet.
- **h** : Displays options for command.
- **V** : Displays the version information.
- **a** : Mounts all devices described at **/etc/fstab**.
- **t** : Type of filesystem device uses.
- **T** : Describes an alternative fstab file.
- **r** : Read-only mode mounted.

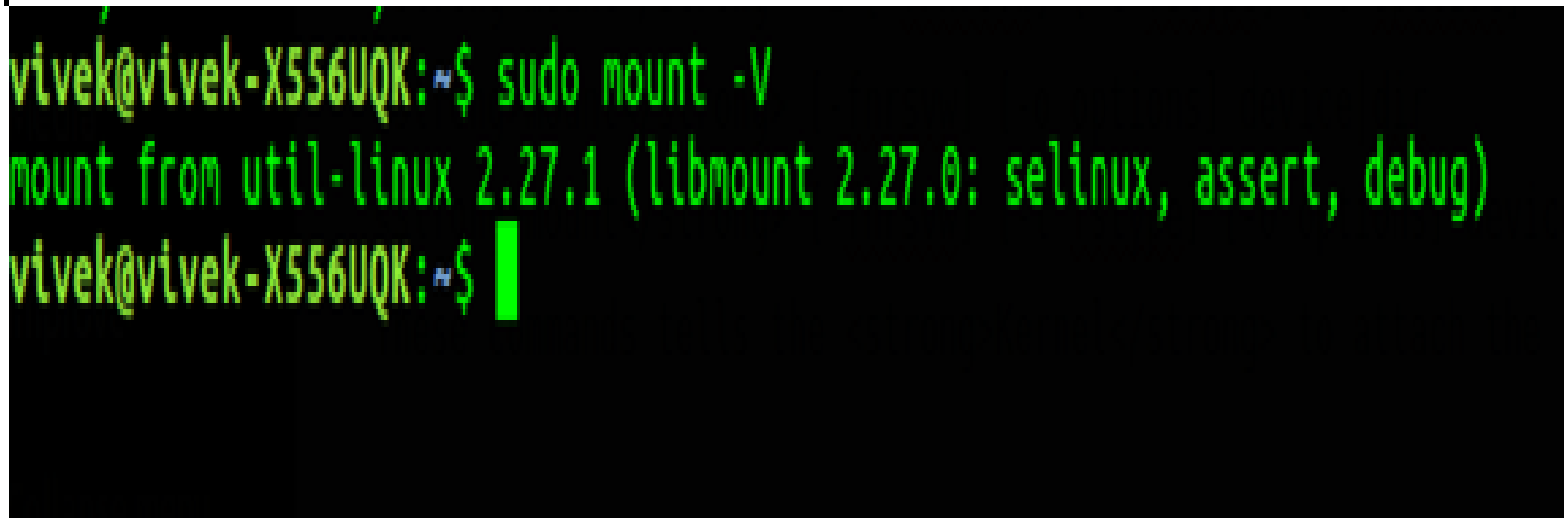
- Displays information about file systems mounted:



# Mounts file systems:

```
vivek@vivek-X556UQK:~$ sudo mount /dev/sda4 /media/vivek
vivek@vivek-X556UQK:~$ sudo mount /dev/sda5 /media/vivek
vivek@vivek-X556UQK:~$ sudo mount /dev/sda6 /media/vivek
vivek@vivek-X556UQK:~$ sudo mount -l -t ext4
/dev/sda3 on / type ext4 (rw,relatime,errors=remount-ro,data=ordered)
/dev/sda4 on /media/vivek type ext4 (rw,relatime,data=ordered)
/dev/sda5 on /media/vivek type ext4 (rw,relatime,data=ordered)
vivek@vivek-X556UQK:~$ sudo mount -l -t fuseblk
/dev/sda6 on /media/vivek type fuseblk (rw,relatime,user_id=0,group_id=0,allow_o
ther,blksize=4096)
vivek@vivek-X556UQK:~$
```

Displays version information:



# umount

- The umount command detaches the file system(s) mentioned from the file hierarchy.

- Note that a file system cannot be unmounted when it is `busy' – for example, when there are open files on it, or when some process has its working directory there, or when a swap file on it is in use.

- The offending process could even be umount itself – it opens libc, and libc in its turn may open for example locale files.

- A lazy unmount avoids this problem

**SYNOPSIS**

- umount [-hV]
- umount -a [-dflnrv] [-t vfstype] [-O options]
- umount [-dflnrv] {dir|device}…

- **OPTION :**

  **-V :-** Print version information and exit.

  **-h :-** Print a help message and exit.

  **-v :-** Run in verbose mode.

  **-n :-** Unmount without writing in /etc/mtab.

  **-r :-** In case unmounting fails, try to remount read-only.

  **-d :-** In case the unmounted device was a loop device, also free this loop device.

## EXAMPLES

1. Unmount a single mount point

```
# umount /mydata
```

2. Unmount more than one mount points

Umount allows you to unmount more than mount point in a single execution of umount of command as follows:

```
# umount /mydata  /backup
# mount | grep /mydata
# mount | grep /backup
```

4. Forcefully unmount a filesystem

umount provides the option to forcefully unmount a filesystem with option -f when the device is busy as shown below:

```
# umount -f /mnt
```

**Note :** If this doesn't work for you, then you can go for lazy unmount.

5. To unmount files and directories of a specific type, enter:

```
# umount -t test
```

This unmounts all files or directories that have a stanza in the /etc/filesystems file that contains the type=test attribute.

# TEXT PROCESSING UTILITIES

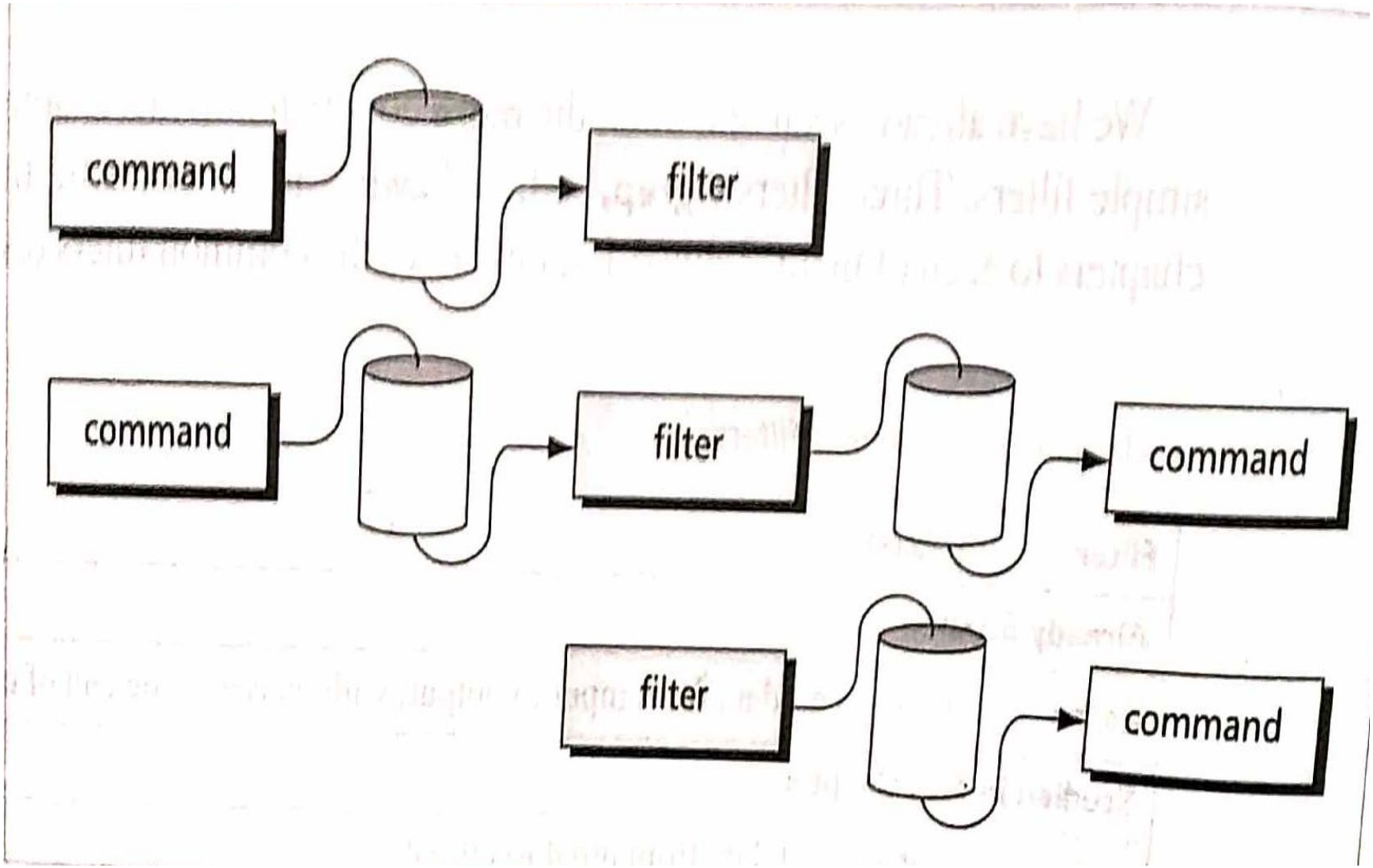
- UNIX provides a number of powerful commands to process texts in different ways. These text processing commands are often implemented as filters.

- Filters are commands that always read their input from ‘stdin’ and write their output to ‘stdout’.

- By default, when using a shell terminal, the *stdin* is from the keyboard, and the *stdout* is to the terminal.

- Filters work naturally with pipes. Because a filter can send its output to the monitor, it can be used on the left of a pipe, because a filter can receive its input from the keyboard, it can be used on the right of a pipe.

- In other words, a filter can be used on the left of a pipe, between two pipes, and on the right of a pipe.


command
filter
command
filter
command
filter
command

- This UNIX text processing commands is divided into 3 parts.

1. Unix Filters
2. Unix pipes
3. More filter commands like awk and sed.

1. head
2. tail
3. cut
4. Paste
5. sort
6. tr
7. uniq
8. wc
9. cmp
10. diff
11. comm
12. join

# head Command

- While the “cat” command copies entire files, the head command copies a specified number of lines from the beginning of one or ore files to the standard output stream.

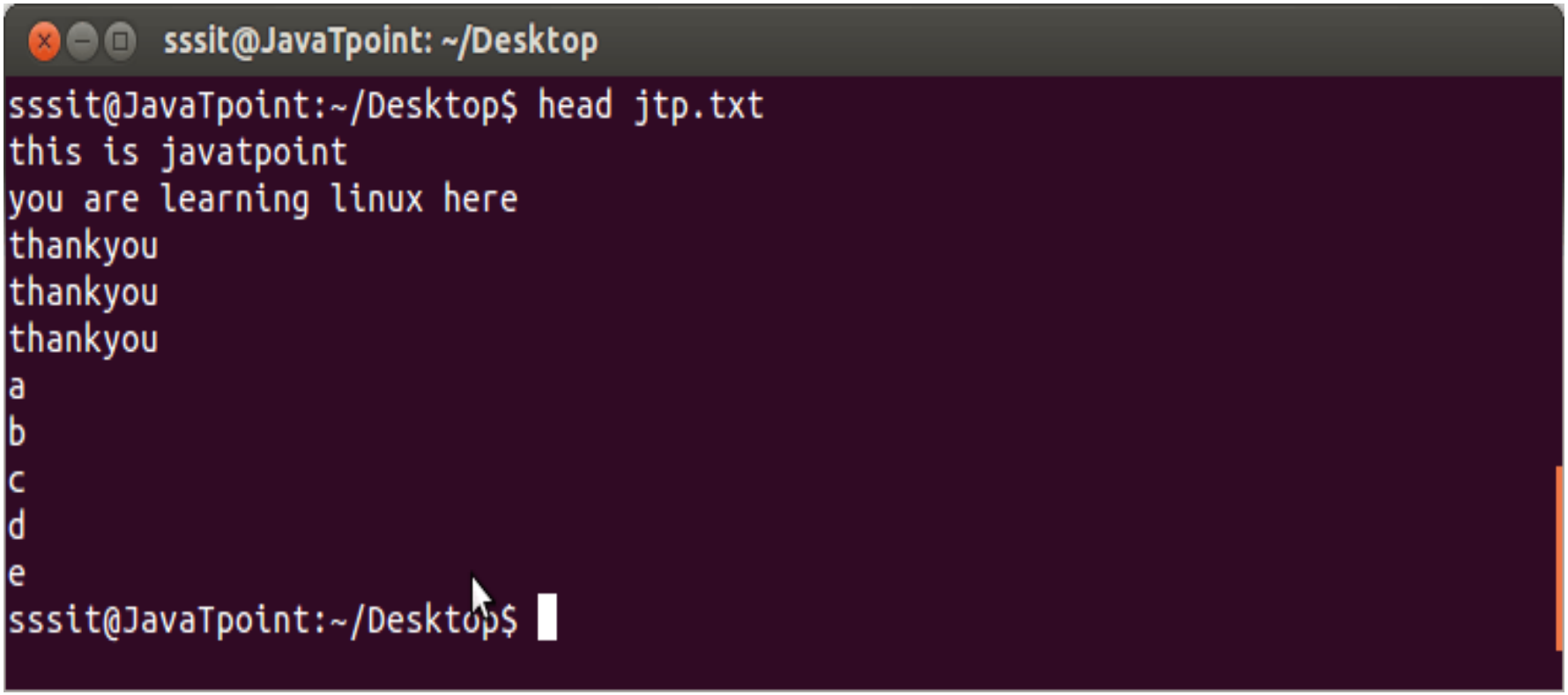
- By default, it displays starting 10 lines of any file.

**Syntax:**

- head **<file** name**>**

**Example:**

- head jtp.txt



**Head command for multiple files**

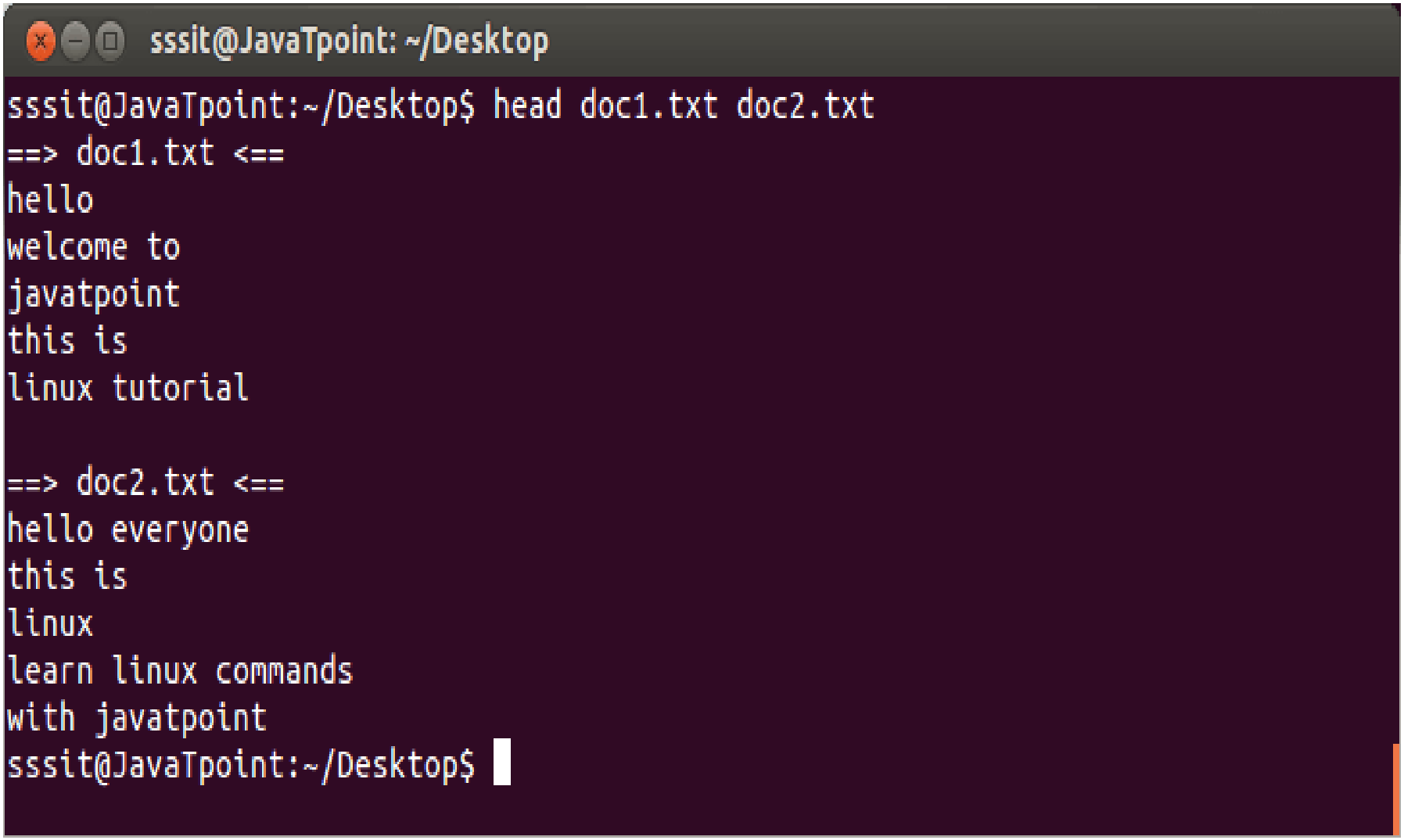
- If we'll write two file names then it will display first ten lines of each file separated by a heading.

**Syntax:**

- head **<file** name**> <file** name**>**

**Example:**

- head doc1.txt doc2.txt


sssit@JavaTpoint: ~/Desktop
sssit@JavaTpoint:~/Desktop$ head doc1.txt doc2.txt
==> doc1.txt <==
hello
welcome to
javatpoint
this is
linux tutorial
==> doc2.txt <==
hello everyone
this is
linux
learn linux commands
with javatpoint
sssit@JavaTpoint:~/Desktop$

Options:-

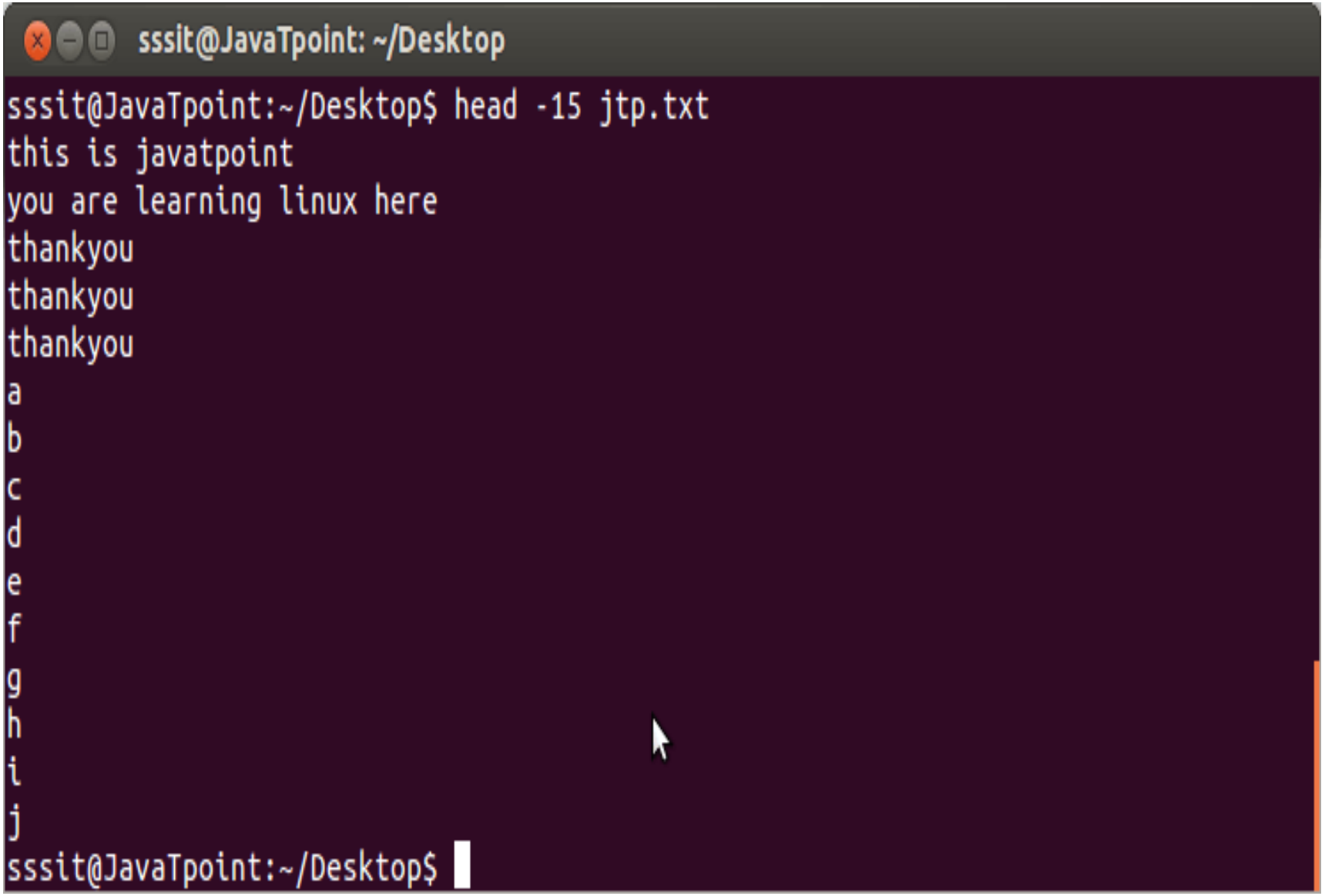
1. -n:- The 'head -n' option displays specified number of lines.

**Syntax:**

- head -n **<file** name**>**

**Example:**

- head -15 jtp.txt


sssit@JavaTpoint: ~/Desktop
sssit@JavaTpoint:~/Desktop$ head -15 jtp.txt
this is javatpoint
you are learning linux here
thankyou
thankyou
thankyou
a
b
c
d
e
f
g
h
i
j
sssit@JavaTpoint:~/Desktop$

# tail Command

- The tail command also outputs data, only this time from the end of the file.

**Syntax:-**

- tail [options] filename

**Example:-**

- $tail hai.txt

- It has several options. If the option starts with a plus sign, tail skips N-1 lines before it begins to output lines from the file and continues until it gets to the end of the file.

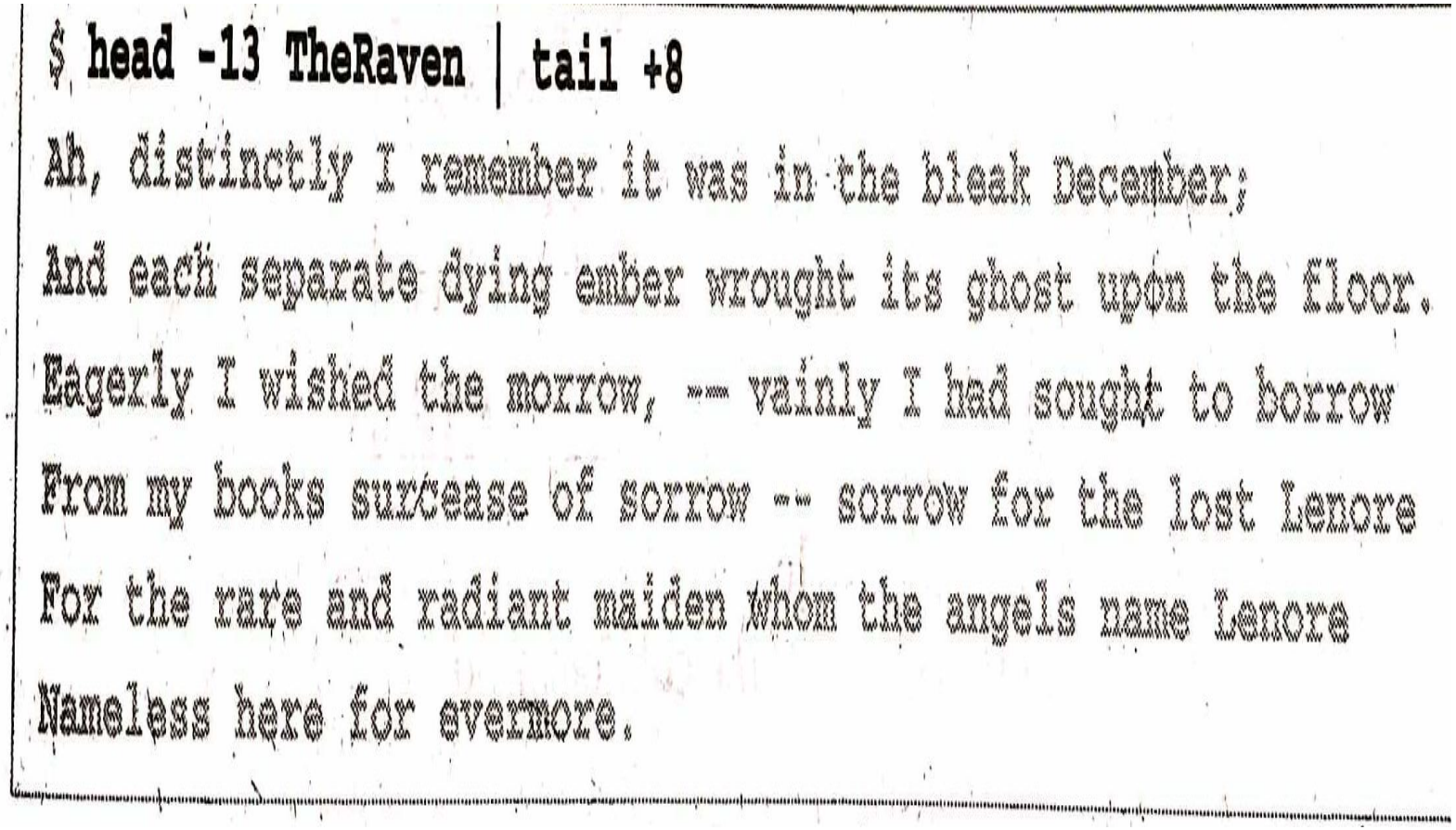
- If it starts with a minus, such as -25, it outputs the last number of lines specified in the option.

- If there are no line options, the default is the last 10 lines.

| Option | Code | Description |
|---|---|---|
| Count from beginning | +N | Skips *N* – 1 lines; copies rest to end of file. |
| Count from end | -N | Copies last *N* lines. |
| Count by lines | -l | Counts by line (default). |
| Count by characters | -c | Counts by character. |
| Count by blocks | -b | Counts by disk block. |
| Reverse order | -r | Outputs in reverse order (from bottom to top). |

```
$ tail -2r goodStudents
of their colleges.
to come to the aid
```

- We can combine the head and tail commands to extract lines from the center of a file.



# cut Command

- The basic purpose of the cut command is to extract one or more columns of data from either standard input or from one or more files.

- **Syntax:-**

  cut OPTION... [FILE]...

- Since cut looks for columns, we must have some way to specify where the columns are located.

- This is done with one of two command options. We can specify what we want to extract based on character positions within a line or by a field number.

## Specifying Character Positions:-

- Character positions work well when the data are aligned in fixed columns.

TABLE 6.3 Data for the Five Largest Cities in the United States (1990 Census)

| | | | | |
|---|---|---|---|---|
| Chicago | IL | 2783726 | 3005072 | 1434029 |
| Houston | TX | 1630553 | 1595138 | 1049300 |
| Los Angeles | CA | 3485398 | 2968528 | 1791011 |
| New York | NY | 7322564 | 7071639 | 3314000 |
| Philadelphia | PA | 1585577 | 1688210 | 1736895 |

- City is a string of 15 characters, state is 3 characters (including the trailing space), 1990 population is 8 characters, 1980 population is 8 characters, and work force is 8 characters.

- To specify that the file is formatted with fixed columns, we use the character option, -c, followed by one or more column specifications.

**Example:-**

- $cut –c1-14,19-25 censusFixed

SESSION 6.11 *Example of* **cut** *Command*

```
$ cut -c1-14,19-25 censusFixed
Chicago          2783726
Houston          1630553
Los Angeles      3485398
New York         7322564
Philadelphia     1585577
```

**Field Specification:-**

- While the column specification works well when the data are organized around fixed columns, it doesn’t work in other situations.

```
Chicago<tab>       IL<tab>  2783726<tab>  3005072<tab>  1434029
Houston<tab>       TX<tab>  1630553<tab>  1595138<tab>  1049300
Los Angeles<tab>   CA<tab>  3485398<tab>  2968528<tab>  1791011
New York<tab>      NY<tab>  7322564<tab>  7071639<tab>  3314000
Philadelphia<tab>  PA<tab>  1585577<tab>  1688210<tab>  736895
```

- To specify a field, we use the field option (-f). Fields are numbered from the beginning of the line with the first field being field number one.

**Example:-**

- $cut –f1 filename

```
$ cut -f1 censusTab
Chicago
Houston
Los Angeles
New York
Philadelphia
```

- Note:- The cut command is similar to the head and tail commands. The cut command cuts files vertically (columns), whereas the head and tail commands cut files horizontally (lines).

cut Command Options

| Option | Code | Results |
|---|---|---|
| Character | `-c` | Extracts fixed columns specified by column number. |
| Field | `-f` | Extracts delimited columns. |
| Delimiter | `-d` | Specifies delimiter if not tab (default). |
| Suppress | `-s` | Suppresses output if no delimiter in line. |

# paste Command

- Paste command is one of the useful commands in Unix or Linux operating system.

- It is used to join files horizontally (parallel merging) by outputting lines consisting of lines from each file specified, separated by **tab** as delimiter, to the standard output.

**Syntax:**

- paste [OPTION]... [FILES]...

```
$ cat state
Arunachal Pradesh
Assam
Andhra Pradesh
Bihar
Chhattisgrah

$ cat capital
Itanagar
Dispur
Hyderabad
Patna
Raipur
```

- Without any option paste merges the files in parallel.

```
$ paste number state capital
1       Arunachal Pradesh       Itanagar
2       Assam   Dispur
3       Andhra Pradesh  Hyderabad
4       Bihar   Patna
5       Chhattisgrah    Raipur
```

**Options:**

1. **-d (delimiter):** Paste command uses the tab delimiter by default for merging the files.

   The delimiter can be changed to any other character by using the **-d** option.

   If more than one character is specified as delimiter then paste uses it in a circular fashion for each file line separation.

**Only one character is specified**

```
$ paste -d "|" number state capital
1|Arunachal Pradesh|Itanagar
2|Assam|Dispur
3|Andhra Pradesh|Hyderabad
4|Bihar|Patna
5|Chhattisgrah|Raipur
```

**More than one character is specified**

```
$ paste -d "|," number state capital
1|Arunachal Pradesh,Itanagar
2|Assam,Dispur
3|Andhra Pradesh,Hyderabad
4|Bihar,Patna
5|Chhattisgrah,Raipur
```

First and second file is separated by '|' and second and third is separated b

After that list is exhausted and reused.

- **-s (serial):** We can merge the files in sequentially manner using the -s option.

  It reads all the lines from a single file and merges all these lines into a single line with each line separated by tab.

  And these single lines are separated by newline.

```
$ paste -s number state capital

1	2	3	4	5

Arunachal Pradesh	Assam	Andhra Pradesh	Bihar	Chhattisgrah

Itanagar	Dispur	Hyderabad	Patna	Raipur
```

- **Combination of -d and -s:** The following example shows how to specify a delimiter for sequential merging of files:

```
$ paste -s -d ":" number state capital

1:2:3:4:5

Arunachal Pradesh:Assam:Andhra Pradesh:Bihar:Chhattisgrah

Itanagar:Dispur:Hyderabad:Patna:Raipur
```

- **–version:** This option is used to display the version of paste which is currently running on your system.

```
$ paste --version
paste (GNU coreutils) 8.26
Packaged by Cygwin (8.26-2)
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later .
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by David M. Ihnat and David MacKenzie.
```

# SORT command

SORT command is used to sort a file, arranging the records in a particular order.

- SORT command sorts the contents of a text file, line by line.

- sort is a standard command line program that prints the lines of its input or concatenation of all files listed in its argument list in sorted order.

- By default, the entire input is taken as sort key. Blank space is the default field separator.

- **Examples**

```
Command :
$ cat > file.txt
abhishek
chitransh
satish
rajan
naveen
divyam
harsh
```

- **Sorting a file : Now use the sort command**
  **Syntax :**

```
$ sort filename.txt
```

```
Command:
$ sort file.txt

Output :
abhishek
chitransh
divyam
harsh
naveen
rajan
satish
```

- **Sort function with mix file i.e. uppercase and lower case :**

  When we have a mix file with both uppercase and lowercase letters then first the lower case letters would be sorted following with the upper case letters .

- **Example:**

```
Command :
$ cat > mix.txt
abc
apple
BALL
Abc
bat
```

```
Command :
$ sort mix.txt
Output :
abc
Abc
apple
bat
BALL
```

- **Options with sort function**

  **-o Option :**

  Unix also provides us with special facilities like if you want to write the **output to a new file**, output.txt, redirects the output like this or you can also use the built-in sort option -o, which allows you to specify an output file.

```
$ sort inputfile.txt > filename.txt
$ sort -o filename.txt inputfile.txt
```

Command:

```
$ sort file.txt > output.txt
$ sort -o output.txt file.txt
$ cat output.txt
```

Output :

```
abhishek
chitransh
divyam
harsh
naveen
rajan
satish
```

- **-r Option: Sorting In Reverse Order** : You can perform a reverse-order sort using the -r flag. the -r flag is an option of the sort command which sorts the input file in reverse order i.e. descending order by default.

```
$ sort -r inputfile.txt
```

```
Command :
$ sort -r file.txt
Output :
satish
rajan
naveen
harsh
divyam
chitransh
abhishek
```

- **-n Option** : To sort a file **numerically** used –n option. -n option is also predefined in Unix as the above options are. This option is used to sort the file with numeric data present inside.

```
Command :
$ cat > file1.txt
50
39
15
89
200
```

```
$ sort -n filename.txt
```

```
Command :
$ sort -n file1.txt
Output :
15
39
50
89
200
```

- **-nr option** : To sort a file with **numeric data in reverse order** we can use the combination of two options as stated below.

```
$ sort -nr filename.txt
```

```
Command :
$ sort -nr file1.txt
Output :
200
89
50
39
15
```

- **-k Option** : Unix provides the feature of sorting a table on the **basis of any column number by using –k option.**

```
$ cat > employee.txt
manager  5000
clerk    4000
employee  6000
peon     4500
director 9000
guard     3000
```

```
$ sort -k filename.txt
```

```
Command :
$ sort -k 2n employee.txt
guard    3000
clerk    4000
peon     4500
manager  5000
employee 6000
director 9000
```

- **-u option :** To **sort and remove duplicates** pass the -u option to sort. This will write a sorted list to standard output and remove duplicates.

  This option is helpful as the duplicates being removed gives us an redundant file.

```
Audi

BMW

Cadillac

BMW

Dodge
```

```
$ sort -u filename.txt
```

```
Command :
$ sort -u cars.txt
$ cat cars.txt
Output :
Audi
BMW
Cadillac
Dodge
```

- **-M Option :** To **sort by month** pass the -M option to sort. This will write a sorted list to standard output ordered by month name.

```
$ cat > months.txt
```

February

January

March

August

September

- **Syntax :**

```
$ sort -M filename.txt
```

```
Command :
$ sort -M months.txt
$ cat months.txt
Output :
January
February
March
August
September
```

# tr Command

- The tr command in UNIX is a command line utility for translating or deleting characters.

- It supports a range of transformations including uppercase to lowercase, squeezing repeating characters, deleting specific characters and basic find and replace.

- It can be used with UNIX pipes to support more complex translation. **tr stands for translate.**

- **Syntax:-**

```
$ tr [OPTION] SET1 [SET2]
```

## 1. How to convert lower case to upper case

To convert from lower case to upper case the predefined sets in tr can be used.

```
$cat greekfile
```

Output:

```
WELCOME TO
GeeksforGeeks
```

```
$cat greekfile | tr “[a-z]” “[A-Z]”
```

Output:

```
WELCOME TO
GEEKSFORGEEKS
```

Or

```
$cat geekfile | tr "[:lower:]" "[:upper:]"
```

Output:

```
WELCOME TO
GEEKSFORGEEKS
```

## 2. How to translate white-space to tabs

The following command will translate all the white-space to tabs.

```
$ echo "Welcome To GeeksforGeeks" | tr [:space:] '\t'
```

Output:

```
Welcome    To    GeeksforGeeks
```

## 3. How to translate braces into parenthesis

You can also translate from and to a file. In this example we will translate braces in a file with parenthesis.

```
$cat greekfile
```

Output:

```
 {WELCOME TO}
GeeksforGeeks
```

```
$ tr '{}' '()'   newfile.txt
```

Output:

```
(WELCOME TO)

GeeksforGeeks
```

## 4. How to use squeeze repetition of characters using -s

To squeeze repeat occurrences of characters specified in a set use the -s option.

This removes repeated instances of a character (or) we can say that, you can convert multiple continuous spaces with a single space

```
$ echo "Welcome    To    GeeksforGeeks" | tr -s [:space:] ' '
```

Output:

```
Welcome To GeeksforGeeks
```

## 5. How to delete specified characters using -d option

To delete specific characters use the -d option. This option deletes characters in the first set specified.

```
$ echo "Welcome To GeeksforGeeks" | tr -d 'w'
```

Output:

```
elcome To GeeksforGeeks
```

## 6. To remove all the digits from the string, use

```
$ echo "my ID is 73535" | tr -d [:digit:]
```

Output:

```
my ID is
```

## 7. How to complement the sets using -c option

You can complement the SET1 using -c option. For example, to remove all characters except digits, you can use the following.

```
$ echo "my ID is 73535" | tr -cd [:digit:]
```

Output:

```
73535
```

Or

```
$ echo “unix” | tr -c “u” “a”
```

Output:

```
Uaaa
```

# uniq Command

- The **uniq** command in Linux is a command line utility that reports or filters out the repeated lines in a file.

- In simple words, **uniq** is the tool that helps to detect the adjacent duplicate lines and also deletes the duplicate lines.

**Syntax of uniq Command :**

$uniq [OPTION] [INPUT[OUTPUT]]

# Example:

```
//displaying contents of kt.txt//

$cat kt.txt
I love music.
I love music.
I love music.

I love music of Kartik.
I love music of Kartik.

Thanks.
```

```
//...using uniq command.../

$uniq kt.txt
I love music.

I love music of Kartik.

Thanks.

/* with the use of uniq all
the repeated lines are removed*/
```

## Options

**1. Using -c option :** It tells the number of times a line was repeated.

```
//using uniq with -c//

$uniq -c kt.txt
3 I love music.
1
2 I love music of Kartik.
1
1 Thanks.

/*at the starting of each
line its repeated number is
displayed*/
```

**2. Using -d option :** It only prints the repeated lines.

```
//using uniq with -d//


$uniq -d kt.txt
I love music.
I love music of Kartik.


/*it only displayed one
 duplicate line per group*/
```

3. **Using -D option :** It also prints only duplicate lines but not one per group.

```
//using -D option//

$uniq -D kt.txt
I love music.
I love music.
I love music.
I love music of Kartik.
I love music of Kartik.

/* all the duplicate lines
are displayed*/
```

**4. Using -u option:** It prints only the unique lines.

```
//using -u option//

$uniq -u kt.txt
Thanks.

/*only unique lines are
displayed*/
```

5. **Using -w option :** Similar to the way of skipping characters, we can also ask uniq to limit the comparison to a set number of characters. For this, -w command line option is used.

```
//displaying content of f3.txt//

$cat f3.txt
How it is possible?
How it can be done?
How to use it?

//now using -w option//

$uniq -w 3 f3.txt
How

/*as the first 3 characters
of all the 3 lines are same
that's why uniq treated all these
as duplicates and gave output
accordingly*/
```

6. **Using -i option :** It is used to make the comparison case-insensitive.

```
//displaying contents of f4.txt//

$cat f4.txt
I LOVE MUSIC
i love music
THANKS

//using uniq command//
$uniq f4.txt
I LOVE MUSIC
i love music
THANKS

/*the lines aren't treated
as duplicates with simple
use of uniq*/

//now using -i option//

$uniq -i f4.txt
I LOVE MUSIC
THANKS

/*now second line is removed
when -i option is used*/
```

# wc Command

- wc stands for **word count**. As the name implies, it is mainly used for counting purpose.

- It is used to find out **number of lines**, **word count**, **byte and characters count** in the files specified in the file arguments.

- By default it displays **four-columnar output.**

- **First column** shows number of lines present in a file specified, **second column** shows number of words present in the file, **third column** shows number of characters present in file and **fourth column** itself is the file name which are given as argument.

**Syntax:**

- **wc [OPTION]... [FILE]...**

- Let us consider two files having name **state.txt** and **capital.txt** containing 5 names of the Indian states and capitals respectively.

```
$ cat state.txt
Andhra Pradesh
Arunachal Pradesh
Assam
Bihar
Chhattisgarh

$ cat capital.txt
Hyderabad
Itanagar
Dispur
Patna
Raipur
```

- **Passing only one file name in the argument.**

```
$ wc state.txt
 5  7 63 state.txt
       OR
$ wc capital.txt
 5  5 45 capital.txt
```

**Passing more than one file name in the argument.**

```
$ wc state.txt capital.txt
  5   7  63 state.txt
  5   5  45 capital.txt
 10  12 108 total
```

**Options:**

1. **-l:** This option prints the **number of lines** present in a file. With this option wc command displays two-columnar output, 1st column shows number of lines present in a file and 2nd itself represent the file name.

```
With one file name
$ wc -l state.txt
5 state.txt

With more than one file name
$ wc -l state.txt capital.txt
  5 state.txt
  5 capital.txt
 10 total
```

**2. -w:** This option prints the **number of words** present in a file. With this option wc command displays two-columnar output, 1st column shows number of words present in a file and 2nd is the file name.

```
With one file name
$ wc -w state.txt
7 state.txt

With more than one file name
$ wc -w state.txt capital.txt
   7 state.txt
   5 capital.txt
  12 total
```

**3. -c:** This option displays **count of bytes** present in a file. With this option it display two-columnar output, 1st column shows number of bytes present in a file and 2nd is the file name.

```
With one file name
$ wc -c state.txt
63 state.txt

With more than one file name
$ wc -c state.txt capital.txt
 63 state.txt
 45 capital.txt
108 total
```

- There are three unix commands that can be used to compare the contents of two files:
  - Compare (cmp)
  - Difference (diff)
  - Common (comm)

# cmp Command

- **cmp** command in Linux/UNIX is used to compare the two files byte by byte and helps you to find out whether the two files are identical or not.

**Syntax:-**

- cmp options... FromFile [ToFile]

**Example**

- $cmp file1.txt file2.txt

1. **If the files are not identical :** the output of the above command will be :

```
$cmp file1.txt file2.txt
file1.txt file2.txt differ: byte 9, line 2

 /*indicating that the first mismatch found in
 two files at byte 20 in second line*/
```

2. **If the files are identical :** you will see something like this on your screen:

```
$cmp file1.txt file2.txt
$ _
/*indicating that the files are identical*/
```

**Options:-**

**1. -b(print-bytes) :** If you want cmp displays the differing bytes in the output when used with **-b** option.

```
//...cmp command used with -b option...//

$cmp -b file1.txt file2.txt
file1.txt file2.txt differ: 12 byte, line 2 is 154 l 151 i

/* indicating that the difference is in 12
 byte ,which is 'l' in file1.txt and 'i' in file2.txt.*/
```

- The values 154 and 151 in the above output are the values for these bytes, respectively.

**2. -l option :** This option makes the cmp command print byte position and byte value for all differing bytes.

```
//...cmp command used with -l option...//

$cmp -l file1.txt file2.txt
20   12   56
21  124   12
22  150  124
23  151  150
24  163  151
25   40  163
26  146   40
27  150  151
28   12   24
29  124  145
30  157  163

/*indicating that files are different
displaying the position of differing
bytes along with the differing bytes
 in both file*/
```

**3. -s option :** This allows you to suppress the output normally produced by cmp command *i.e* it compares two files without writing any messages. This gives an exit value of 0 if the files are identical, a value of 1 if different, or a value of 2 if an error message occurs.

```
//...cmp command used with -s option...//

$cmp -s file1.txt file.txt
1

/*indicating files are different without
displaying the differing byte and line*/
```

# diff command

- diff stands for **difference**. This command is used to display the differences in the files by comparing the files line by line.

- Unlike its fellow members, cmp and comm, it tells us which lines in one file have is to be changed to make the two files identical.

- The important thing to remember is that **diff** uses certain **special symbols** and **instructions** that are required to make two files identical.

- It tells you the instructions on how to change the first file to make it match the second file.

- **Special symbols are:**

```
a : add
c : change
d : delete
```

Syntax :

```
diff [options] File1 File2
```

- Lets say we have two files with names **a.txt** and **b.txt** containing 5 Indian states

```
$ ls
a.txt   b.txt

$ cat a.txt
Gujarat
Uttar Pradesh
Kolkata
Bihar
Jammu and Kashmir

$ cat b.txt
Tamil Nadu
Gujarat
Andhra Pradesh
Bihar
Uttar pradesh
```

```
$ diff a.txt b.txt
0a1
> Tamil Nadu
2,3c3
< Uttar Pradesh
 Andhra Pradesh
5c5
 Uttar pradesh
```

```
$ cat a.txt
Gujarat
Andhra Pradesh
Telangana
Bihar
Uttar pradesh

$ cat b.txt
Gujarat
Andhra Pradesh
Bihar
Uttar pradesh

$ diff a.txt b.txt
3d2
< Telangana
```

# comm command

- comm compare two sorted files line by line and write to standard output; the lines that are common and the lines that are unique.

- Suppose you have two lists of people and you are asked to find out the names available in one and not in the other, or even those common to both.

- **comm** is the command that will help you to achieve this.
- It requires two sorted files which it compares line by line.

**Syntax :**

- $comm [OPTION]... FILE1 FILE2

```
// displaying contents of file1 //
$cat file1.txt
Apaar
Ayush Rajput
Deepak
Hemant

// displaying contents of file2 //
$cat file2.txt
Apaar
Hemant
Lucky
Pranjal Thakral
```

```
// using comm command for
comparing two files //
$comm file1.txt file2.txt
                    Apaar
Ayush Rajput
Deepak
                    Hemant
          Lucky
          Pranjal Thakral
```

**Options for comm command:**

1. **-1 :**suppress first column(lines unique to first file).

2. -**2 :**suppress second column(lin es unique to second file).

3. -**3 :**suppress third column(lines common to both files).

```
//suppress first column using -1//
$comm -1 file1.txt file2.txt
          Apaar
          Hemant
 Lucky
 Pranjal Thakral

//suppress second column using -2//
$comm -2 file1.txt file2.txt
         Apaar
Ayush Rajput
Deepak
         Hemant

//suppress third column using -3//
$comm -3 file1.txt file2.txt
Ayush Rajput
Deepak
         Lucky
         Pranjal Thakral
```

# join Command

- The join command in UNIX is a command line utility for joining lines of two files on a common field.

- **join** command is used to join the two files based on a **key field present in both the files**.

**Syntax:**

- **$join [OPTION] FILE1 FILE2**

```
// displaying the contents of first file //
$cat file1.txt
1 AAYUSH
2 APAAR
3 HEMANT
4 KARTIK

// displaying contents of second file //
$cat file2.txt
1 101
2 102
3 103
4 104
```

```
//..using join command...//
$join file1.txt file2.txt
1 AAYUSH 101
2 APAAR 102
3 HEMANT 103
4 KARTIK 104

// by default join command takes the
first column as the key to join as
in the above case //
```

## Options

1. **using -a FILENUM option :** Now, sometimes it is possible that one of the files contain extra fields so what join command does in that case is that by default, it only prints pair able lines.

   What if such unpair able lines are important and must be visible after joining the files. In such cases we can use **-a option** with join command which will help in displaying such unpair able lines.

```
//displaying the contents of file1.txt//
$cat file1.txt
1 AAYUSH
2 APAAR
3 HEMANT
4 KARTIK
5 DEEPAK

//displaying contents of file2.txt//
$cat file2.txt
1 101
2 102
3 103
4 104

//using join command//
$join file1.txt file2.txt
1 AAYUSH 101
2 APAAR 102
3 HEMANT 103
4 KARTIK 104

// although file1.txt has extra field the
output is not affected cause the 5 column in
file1.txt was unpairable with any in file2.txt//
```

```
//using join with -a option//

//1 is used with -a to display the contents of
first file passed//

$join file1.txt file2.txt -a 1
1 AAYUSH 101
2 APAAR 102
3 HEMANT 103
4 KARTIK 104
5 DEEPAK

//5 column of first file is
also displayed with help of -a option
although it is unpairable//
```

**2. using -v option :** Now, in case you only want to print unpair able lines *i.e* suppress the paired lines in output then **-v option** is used with join command.

```
//using -v option with join//

$join file1.txt file2.txt -v 1
5 DEEPAK

//the output only prints unpairable lines found
in first file passed//
```

**3. using -i option :** Now, other thing about join command is that by default, it is case sensitive.

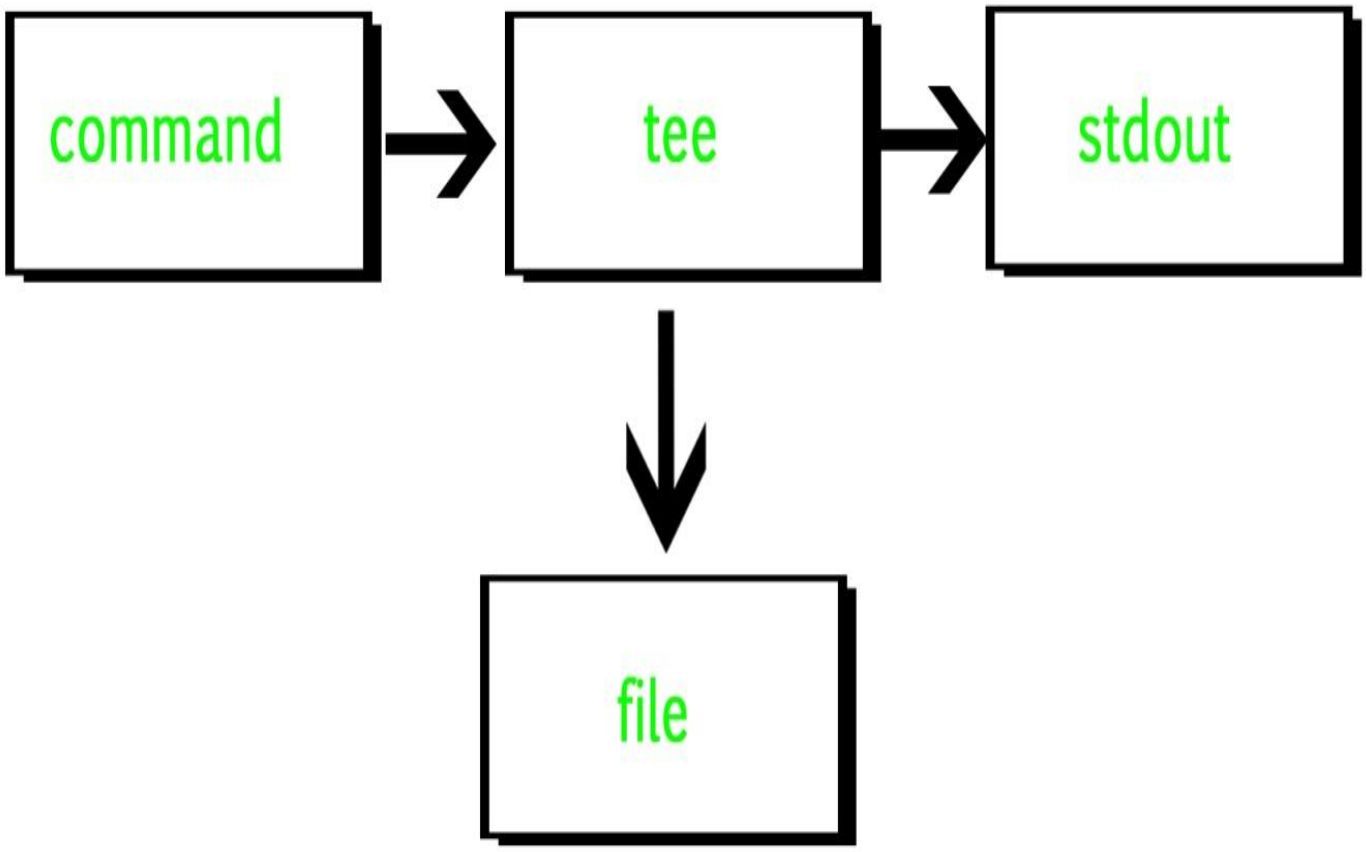
```
//displaying contents of file1.txt//
$cat file1.txt
A AAYUSH
B APAAR
C HEMANT
D KARTIK

//displaying contents of file2.txt//
$cat file2.txt
a 101
b 102
c 103
d 104
```

```
//using -i option with join//
$join -i file1.txt file2.txt
A AAYUSH 101
B APAAR 102
C HEMANT 103
D KARTIK 104
```

# tee command

- **tee command** reads the standard input and writes it to both the standard output and one or more files.

- The command is named after the T-splitter used in plumbing. It basically breaks the output of a program so that it can be both displayed and saved in a file.

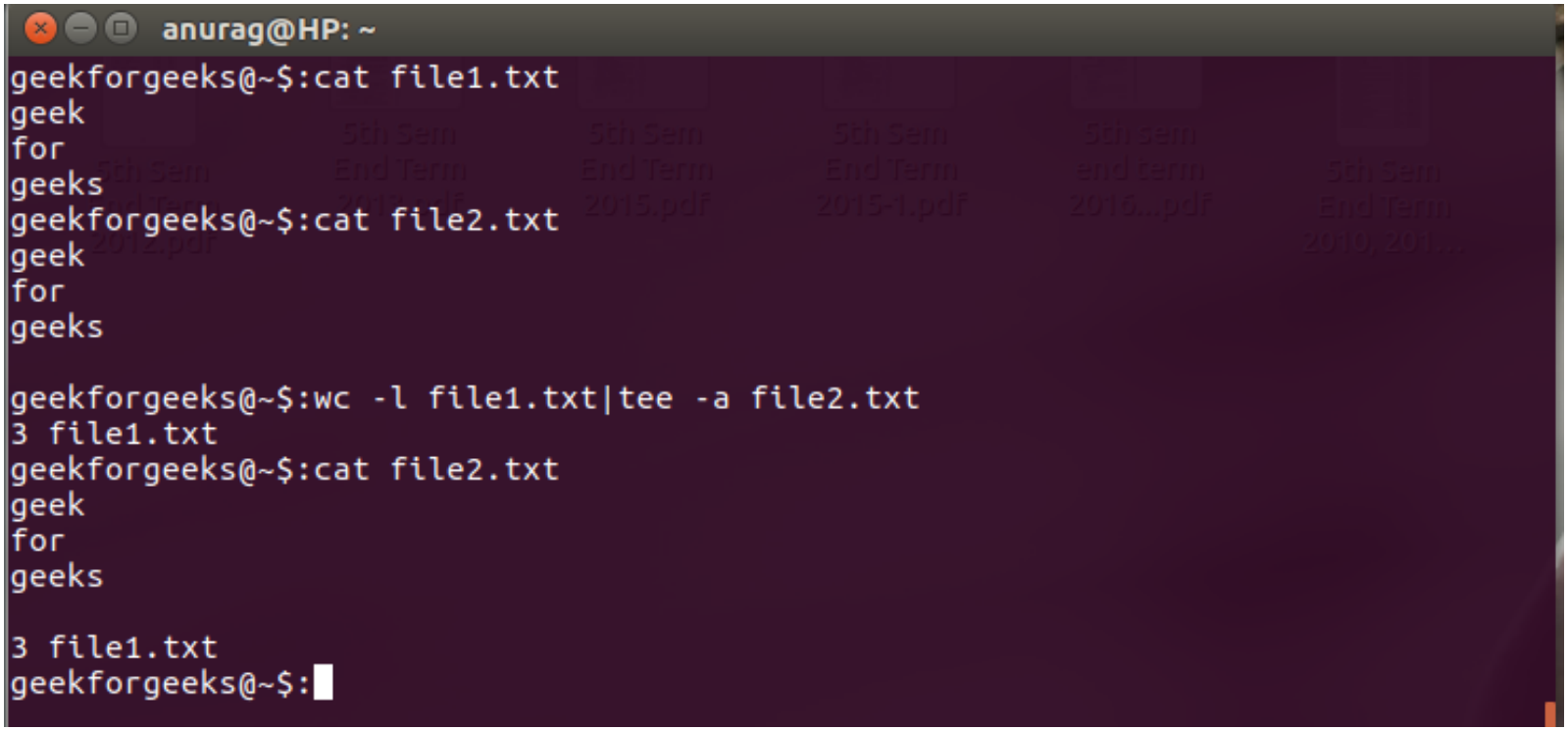

command
tee
stdout
file

**SYNTAX:**

- tee [OPTION]... [FILE]...

**Options :**

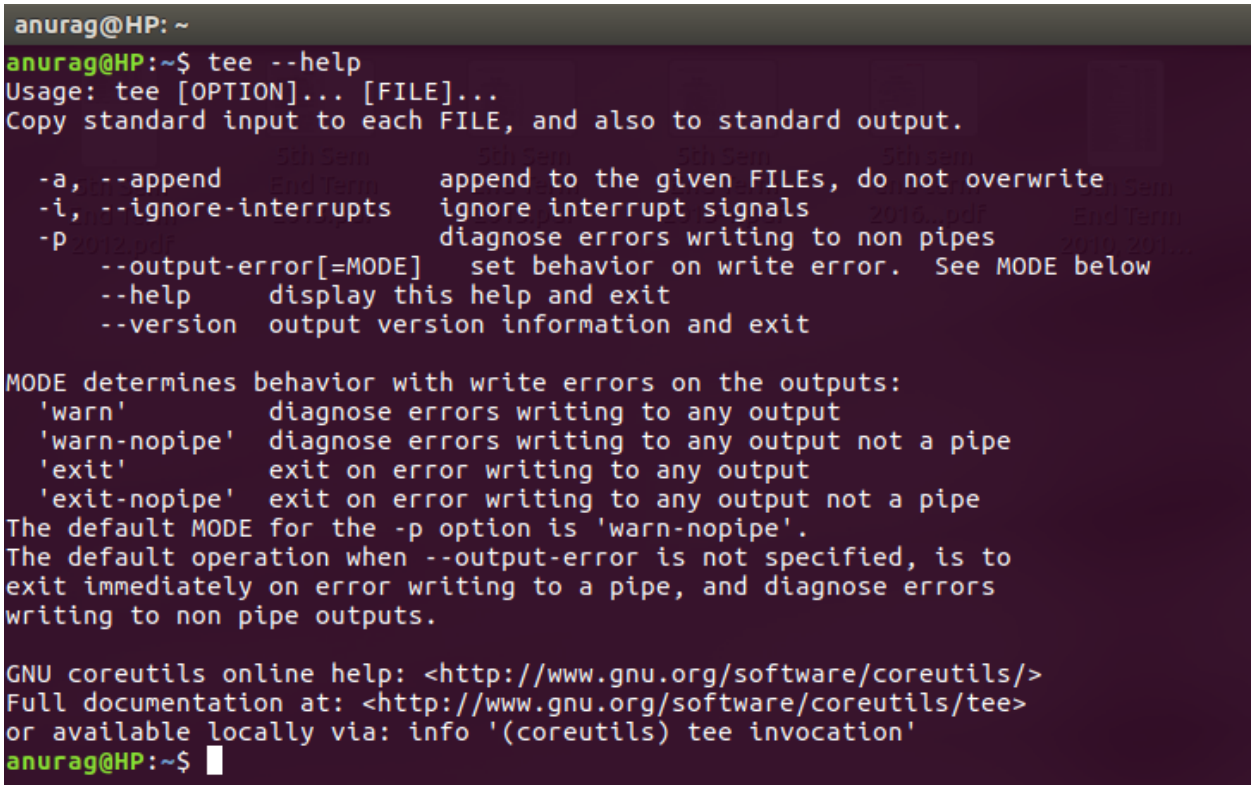
1. **-a Option :** It basically do not overwrite the file but append to the given file.



**2.–help Option :** It gives the help message and exit.

**SYNTAX :**

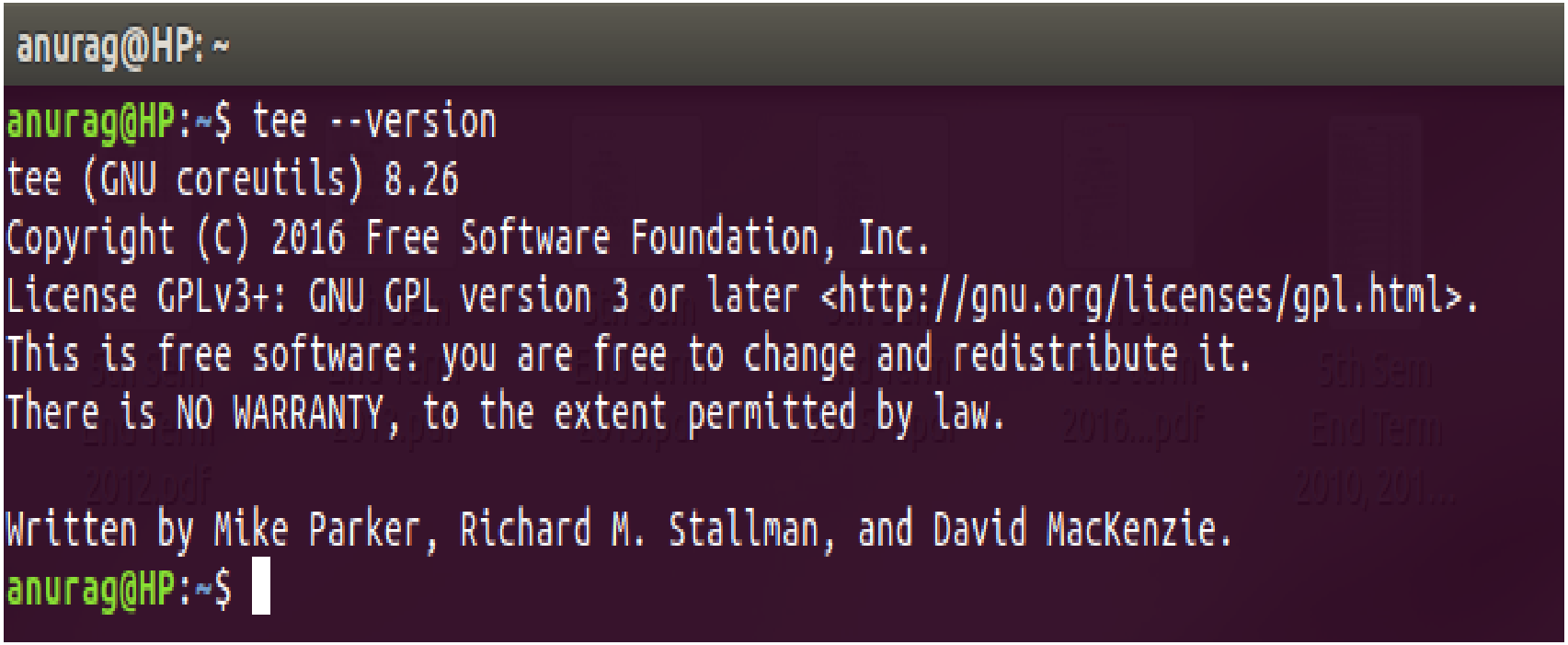
- geek@HP:~$ tee --help



**3.–version Option :** It gives the version information and exit.

**SYNTAX :**

- geek@HP:~$ tee --version



# pg Command

- The **pg** command displays the contents of text files, one page at a time.

- **pg** is a terminal pager program on UNIX and Unix-like systems for viewing text files. It can also be used to page through the output of a command via a pipe.

**Syntax:-**

- pg [options] [*file*...]

# Options

| | |
|---|---|
| *-number* | The number of lines per page. Usually, this is the number of CRT lines minus one. |
| **-c** | Clear the screen before a page is displayed, if the **terminfo** entry for the terminal provides this capability. |
| **-e** | Do not pause and display "(EOF)" at the end of a file. |
| **-f** | Do not split long lines. |
| **-n** | Without this option, commands must be terminated by a newline character. With this option, **pg** advances once a command letter is entered. |

# Finger Command

- **finger** displays the **.plan** file of a specific user, or reports who is logged into a specific machine. The user must allow general read permission on the **.plan** file.

  **Syntax**

- **finger** [options] [user[@hostname]]

# nl Command

- The **nl** command, numbers the lines in a file.

**Syntax**

- nl [*OPTION*]... [*FILE*]...

## Examples

```
cat list.txt
apples
oranges
potatoes
lemons
garlic
```

```
nl list.txt
     1	apples
     2	oranges
     3	potatoes
     4	lemons
     5	garlic
```

```
nl list.txt > nlist.txt
cat nlist.txt
```

```
1       apples
2       oranges
3       potatoes
4       lemons
5       garlic
```

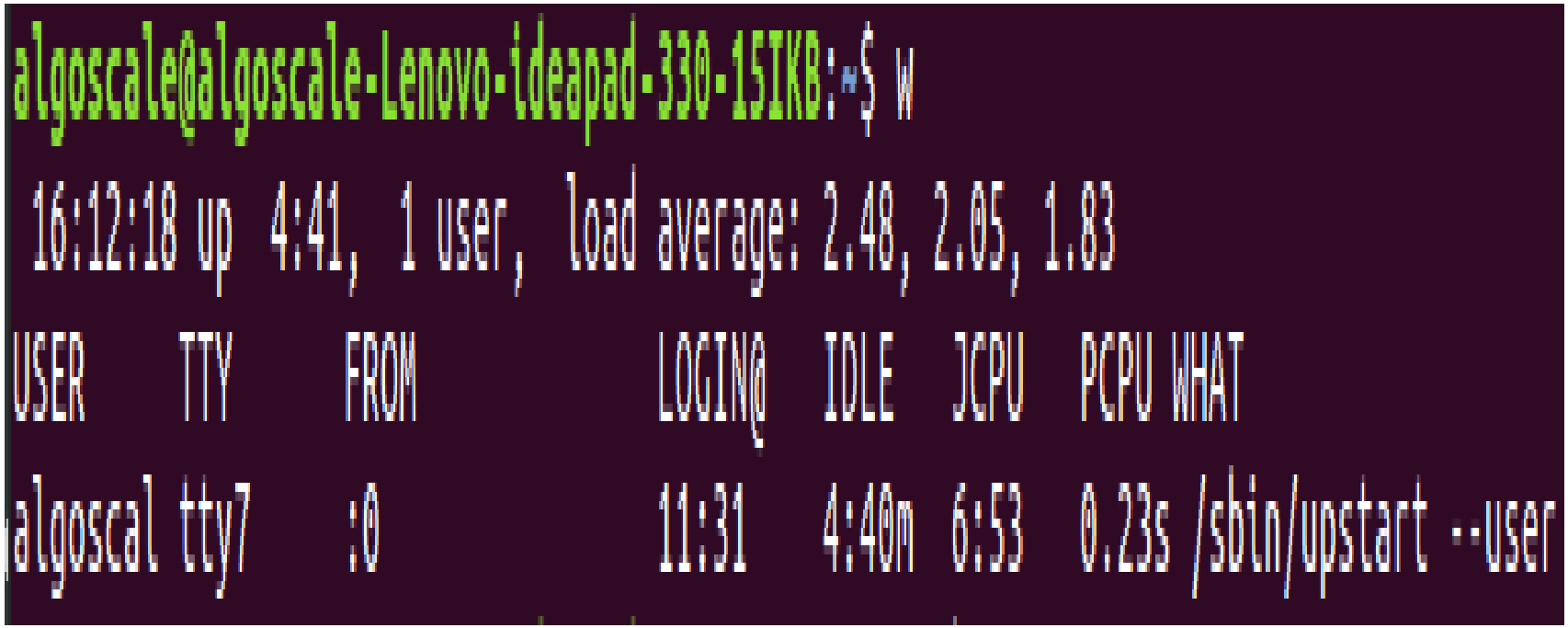
# w command

- **w** command in Linux is used to show who is logged on and what they are doing.

- This command shows the information about the users currently on the machine and their processes.

**Syntax:**

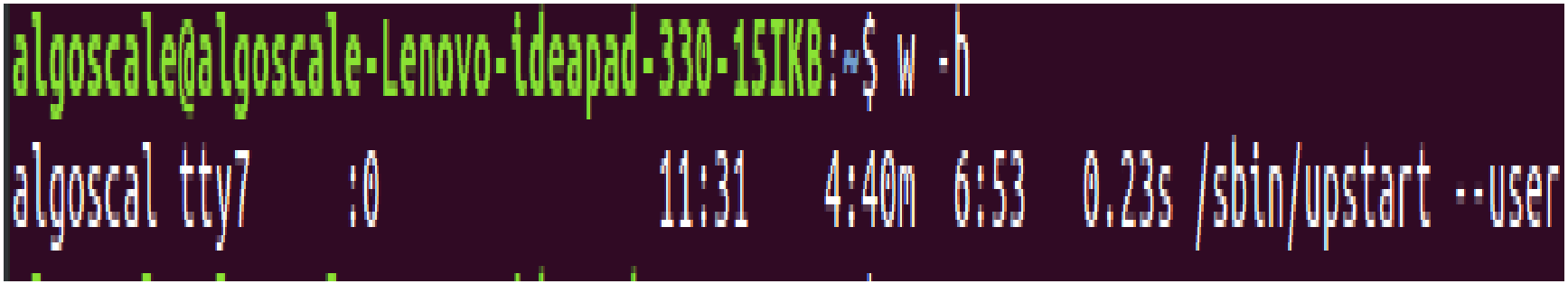
- w [options] user [...]



**Options:**

- **w -h:** This option don’t print the header.

**Syntax:-** $w -h



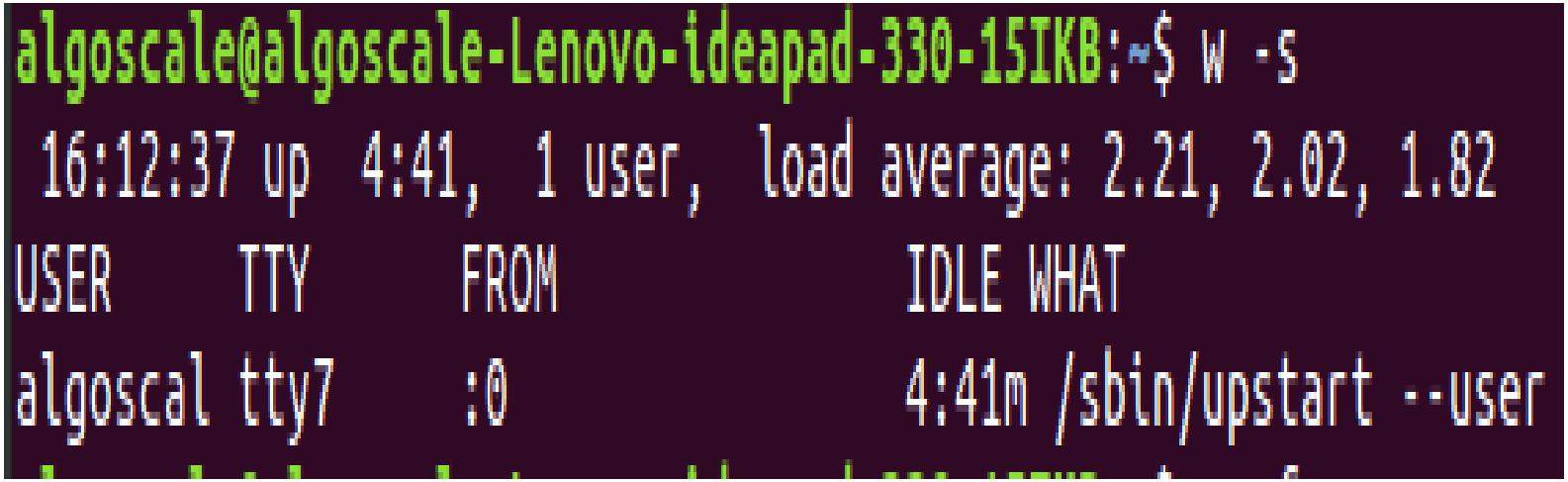
- **w -u:** This option will ignore the username while figuring out the current process and cpu times.

  **Syntax:-** $w -u

```
algoscale@algoscale-Lenovo-ideapad-330-15IKB:~$ w -u
 16:12:31 up  4:41,  1 user,  load average: 2.25, 2.02, 1.82
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
algoscal tty7     :0               11:31    4:40m  6:53   0.23s /sbin/upstart --user
```

- **w -s :** This option uses the short format. It will not print the login time, JCPU or PCPU times.

  **Syntax:-** $w -s



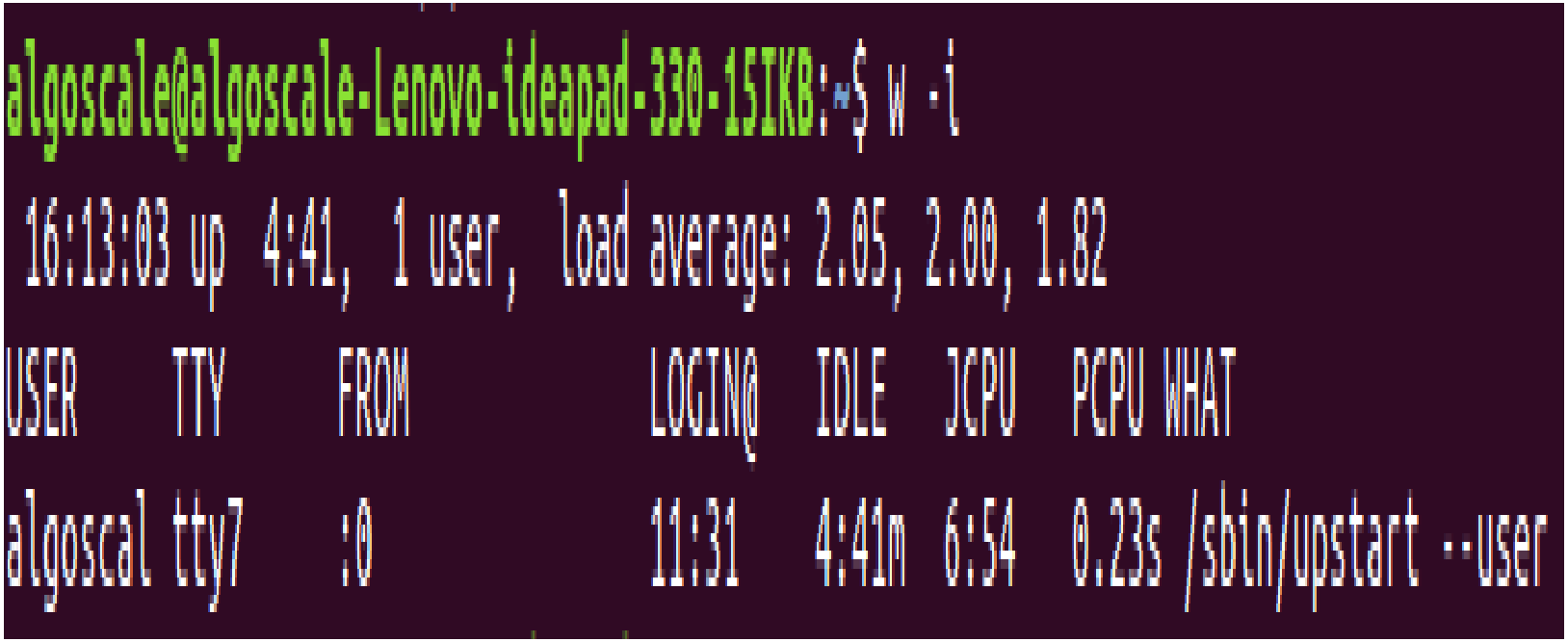
- **w –help:** This option will display help message and exit.

  **Syntax:-** $w --help

```
algoscale@algoscale-Lenovo-ideapad-330-15IKB:~$ w --help

Usage:
 w [options]

Options:
 -h, --no-header     do not print header
 -u, --no-current    ignore current process username
 -s, --short         short format
 -f, --from          show remote hostname field
 -o, --old-style     old style output
 -i, --ip-addr       display IP address instead of hostname (if possible)

     --help     display this help and exit
 -V, --version  output version information and exit

For more details see w(1).
```

- **w -i :** This option will display IP address instead of hostname for from field.

  **Syntax:-** $w -i



# ulimit Command

- The shell contains a built in command called "Ulimit" which allows you to display and set resource limits for users.

- The systems resources are defined in a file called "/etc/security/limits.conf".

- Ulimit can then be used to view these settings.

- The basic syntax of the ulimit command is: **ulimit *Options* limit**

**Display settings for current user**

- To display all of your current settings you can issue the command: "**ulimit -a**"

```
john@john-desktop:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 19868
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 19868
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

# unlink Command

- The **unlink** command calls and directly interfaces with the **unlink** system function, which removes a specified file.

- **Syntax**

  unlink *FILE*

  unlink *OPTION*

**Options**

- **--help**

  Display a help message and exit.
- **--version**

  Output version information and exit.

**Examples**

- unlink hope.txt

- Removes the file name **hope.txt**, and if there is no other hard link to the file data, the file data itself is removed from the system.

# grep command

- The grep filter searches a file for a particular pattern of characters, and displays all lines that contain that pattern.

- The pattern that is searched in the file is referred to as the regular expression (grep stands for globally search for regular expression and print out).

## Syntax:

- **grep [options] pattern [files]**

```
$cat > geekfile.txt
unix is great os. unix is opensource. unix is free os.

learn operating system.

Unix linux which one you choose.

uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

**1. Case insensitive search :** The -i option enables to search for a string case insensitively in the give file. It matches the words like “UNIX”, “Unix”, “unix”.

```
$grep -i "UNix" geekfile.txt
```

**Output:**

```
unix is great os. unix is opensource. unix is free os.

Unix linux which one you choose.

uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

**2. Displaying the count of number of matches:** We can find the number of lines that matches the given string/pattern

```
$grep -c "unix" geekfile.txt
```

Output:

```
2
```

**3. Display the file names that matches the pattern :** We can just display the files that contains the given string/pattern.

```
$grep -l "unix" *

or

$grep -l "unix" f1.txt f2.txt f3.xt f4.txt
```

**Output:**

```
geekfile.txt
```

**4. Displaying only the matched pattern :** By default, grep displays the entire line which has the matched string. We can make the grep to display only the matched string by using the -o option.

```
$ grep –o "unix" geekfile.txt
```

**Output:**

```
unix

unix

unix

unix

unix

unix
```

**5. Show line number while displaying the output using grep -n :** To show the line number of file with the line matched.

```
$ grep -n "unix" geekfile.txt
```

Output:

```
1:unix is great os. unix is opensource. unix is free os.

4:uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

# egrep command

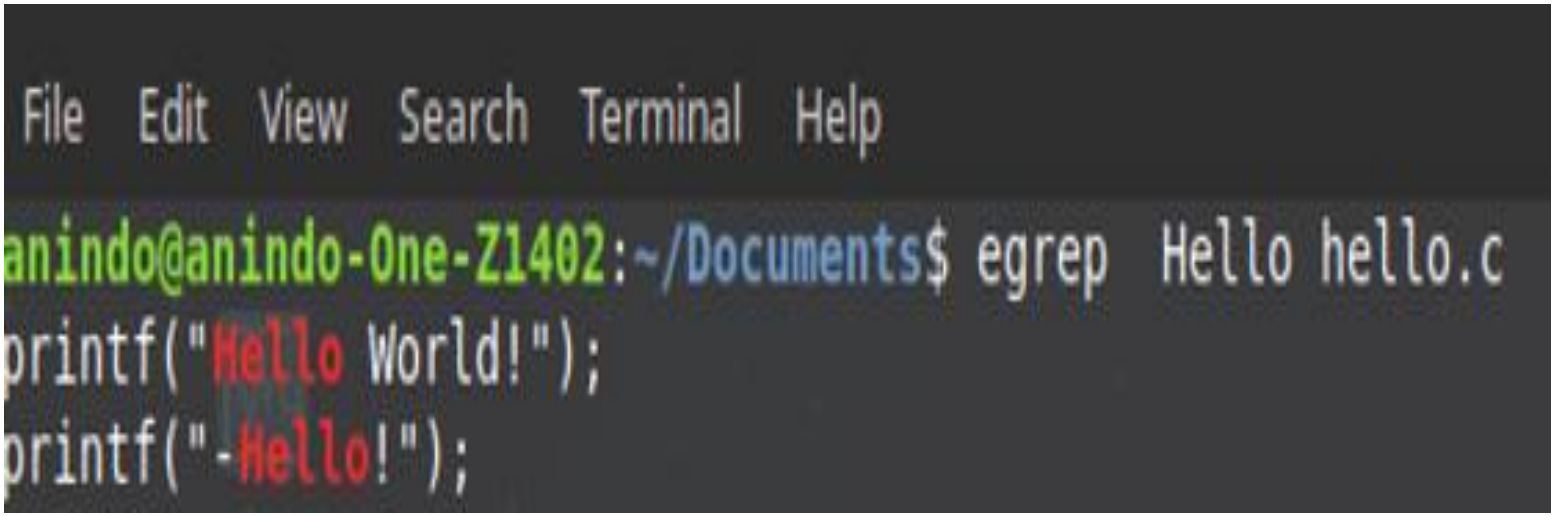
- **egrep** is a pattern searching command which belongs to the family of grep functions.

- It treats the pattern as an extended regular expression and prints out the lines that match the pattern.

**Syntax:**

- egrep [ options ] 'PATTERN' files

**Example:-**



- **Note:** The *egrep* command used mainly due to the fact that it is faster than the grep command.
- The egrep command treats the meta-characters as they are and do not require to be escaped as is the case with grep.
- This allows reducing the overhead of replacing these characters while pattern matching making egrep faster than *grep* or *fgrep*.

**Options:** Most of the options for this command are same as ***grep***

# fgrep command

- The *fgrep* filter is used to search for the fixed-character strings in a file.

- There can be multiple files also to be searched.
- This command is useful when you need to search for strings which contain lots of regular expression metacharacters, such as “^”, “$”, etc.

**Syntax:**

- fgrep [options] [ -e pattern_list] [pattern] [file]

**Options**

- **-c option:** Displaying the count of number of matches. We can find the number of lines that match the given string.

**Example:**

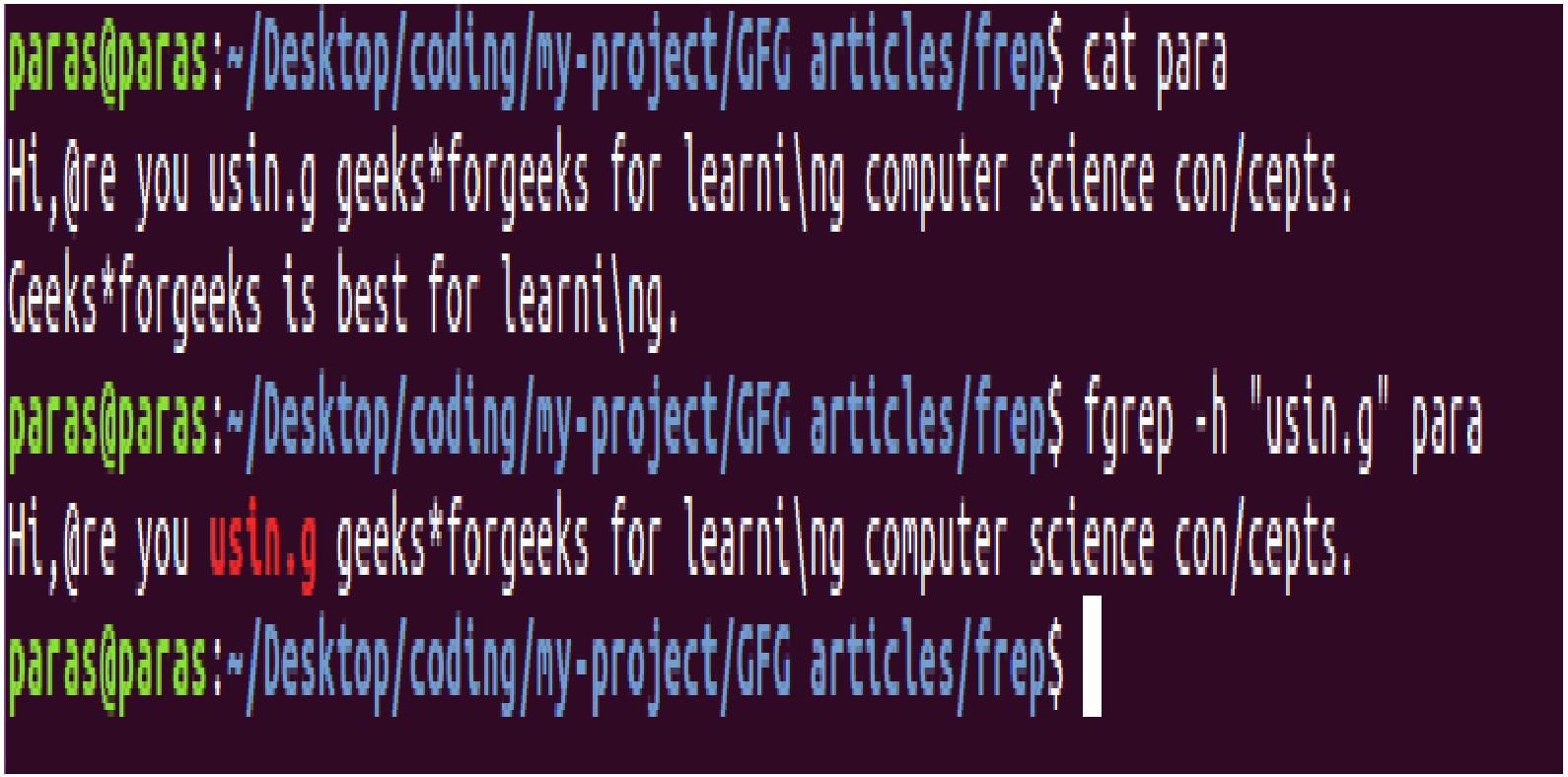
- $fgrep -c "usin.g" para


paras@paras:~/Desktop/coding/my-project/GFG articles/frep$ cat para
Hi,@re you usin.g geeks*forgeeks for learni\ng computer science con/cepts.
Geeks*forgeeks is best for learni\ng.
paras@paras:~/Desktop/coding/my-project/GFG articles/frep$ fgrep -c "usin.g" para
1
paras@paras:~/Desktop/coding/my-project/GFG articles/frep$

- **-h option:** To display the matched lines.

**Example:**

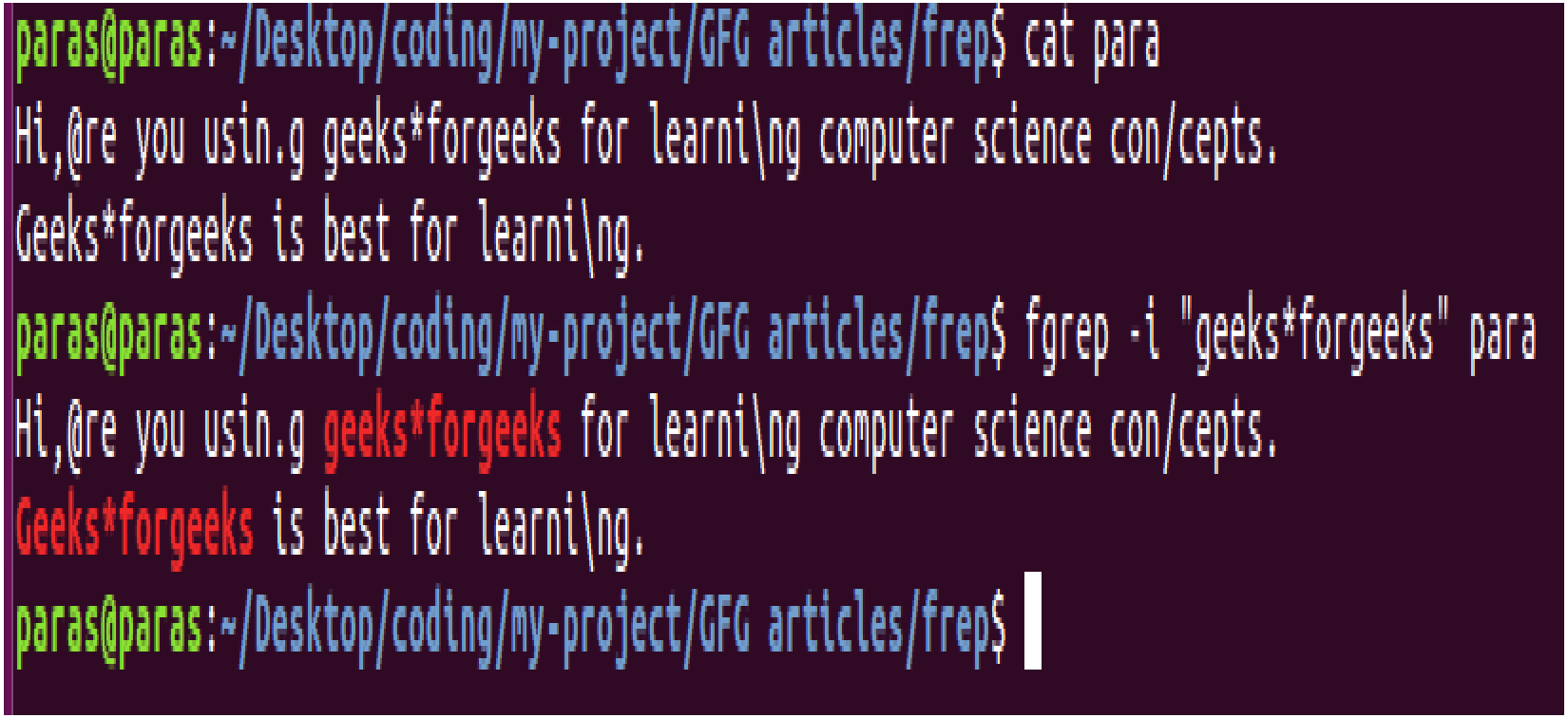
- fgrep -h "usin.g" para


paras@paras:~/Desktop/coding/my-project/GFG articles/frep$ cat para
Hi,@re you usin.g geeks*forgeeks for learni\ng computer science con/cepts.
Geeks*forgeeks is best for learni\ng.
paras@paras:~/Desktop/coding/my-project/GFG articles/frep$ fgrep -h "usin.g" para
Hi,@re you usin.g geeks*forgeeks for learni\ng computer science con/cepts.
paras@paras:~/Desktop/coding/my-project/GFG articles/frep$

- **-i option:** Used in case insensitive search. It ignore upper/lower case distinction during comparisons. It matches words like : “geeks*forgeeks”, “Geeks*forgeeks”.

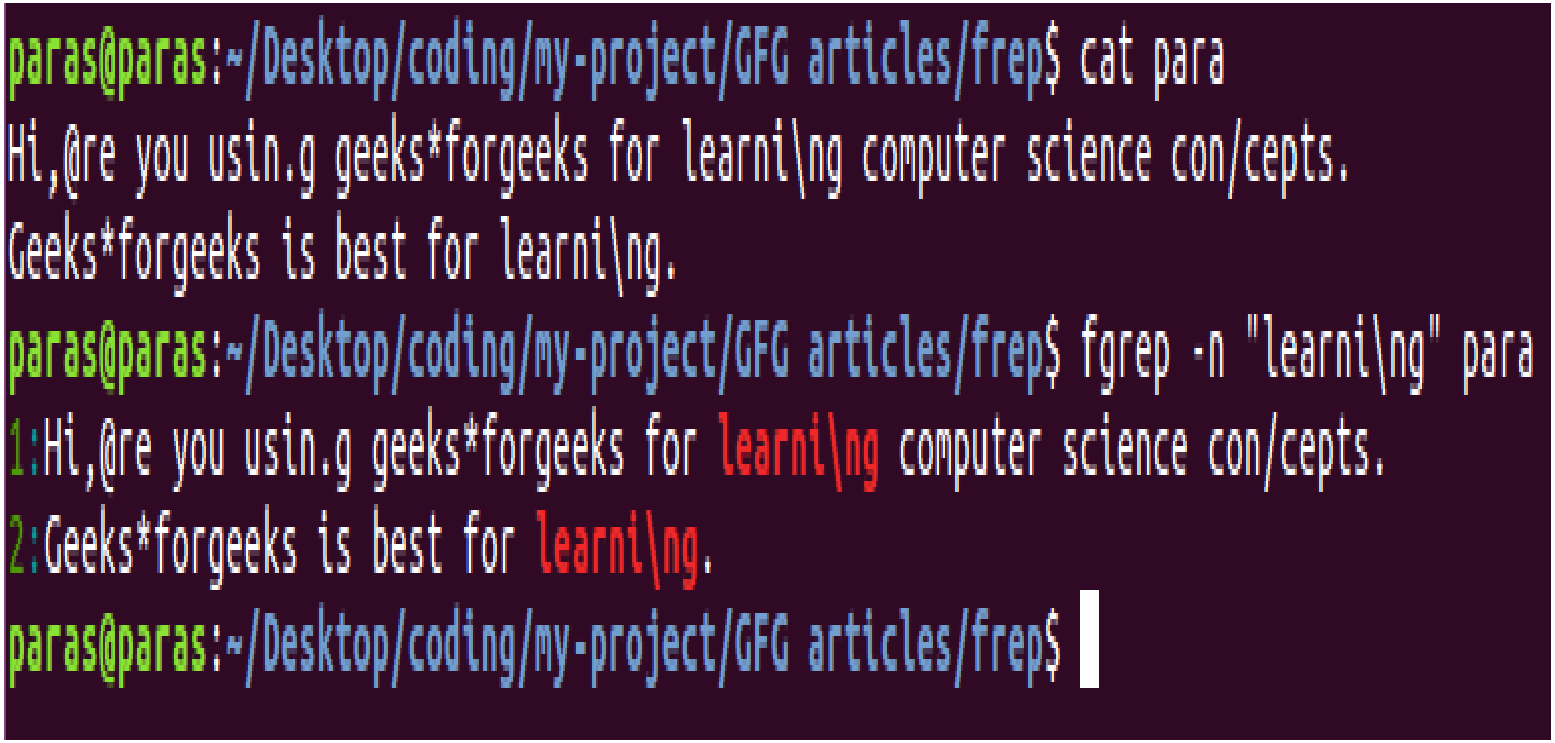
**Example:**

- fgrep -i "geeks*forgeeks" para


paras@paras:~/Desktop/coding/my-project/GFG articles/frep$ cat para
Hi,@re you usin.g geeks*forgeeks for learni\ng computer science con/cepts.
Geeks*forgeeks is best for learni\ng.
paras@paras:~/Desktop/coding/my-project/GFG articles/frep$ fgrep -i "geeks*forgeeks" para
Hi,@re you usin.g geeks*forgeeks for learni\ng computer science con/cepts.
Geeks*forgeeks is best for learni\ng.
paras@paras:~/Desktop/coding/my-project/GFG articles/frep$

- **-n option:** Precede each line by its line number in the file. It shows line number of file with the line matched.

  **Example:**
- $ fgrep -n "learni\ng" para


paras@paras:~/Desktop/coding/my-project/GFG articles/frep$ cat para
Hi,@re you usin.g geeks*forgeeks for learni\ng computer science con/cepts.
Geeks*forgeeks is best for learni\ng.
paras@paras:~/Desktop/coding/my-project/GFG articles/frep$ fgrep -n "learni\ng" para
1:Hi,@re you usin.g geeks*forgeeks for learni\ng computer science con/cepts.
2:Geeks*forgeeks is best for learni\ng.
paras@paras:~/Desktop/coding/my-project/GFG articles/frep$

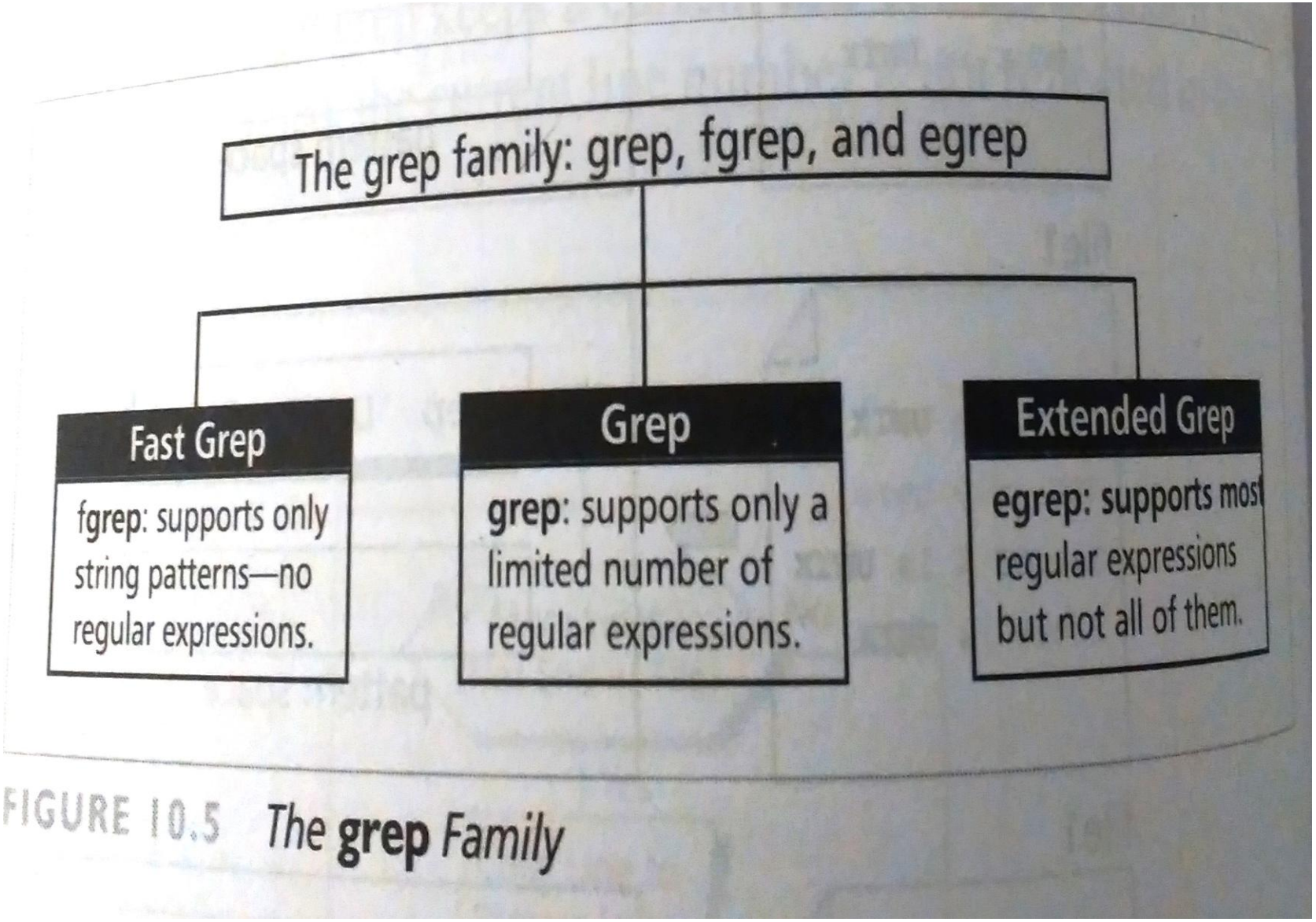



FIGURE 10.5 *The* **grep** *Family*

# BACK UP UTILITIES

1. tar
2. gzip
3. cpio

# 1. tar

- The primary function of the UNIX tar command is to create backups.
- It is used to create a ‘tape archive’ of a directory tree that could be backed up and restored from a tape-based storage device.
- The term ‘tar’ also refers to the file format of the resulting archive file.

**Syntax:**

- tar [function] [options] [paths]

**Options:-**

- **tar -c:** Create a new archive.
- **tar -A:** Append a tar file to another archive.
- **tar -r:** Append a file to an archive.
- **tar -u:** Update files in an archive if the one in the file system is newer.
- **tar -d:** Find the diff between an archive and the file system.
- **tar -t:** List the contents of an archive.
- **tar -x:** Extract the contents of an archive.

- **<u>Examples:</u>**
- **Create an archive file containing file1 and file2**

  $ tar cvf archive.tar file1 file2

- **Create an archive file containing the directory tree below dir**

  $ tar cvf archive.tar dir

- **List the contents of archive.tar**

  $ tar tvf archive.tar

- **Extract the contents of archive.tar to the current directory**

  $ tar xvf archive.tar

# 2. gzip

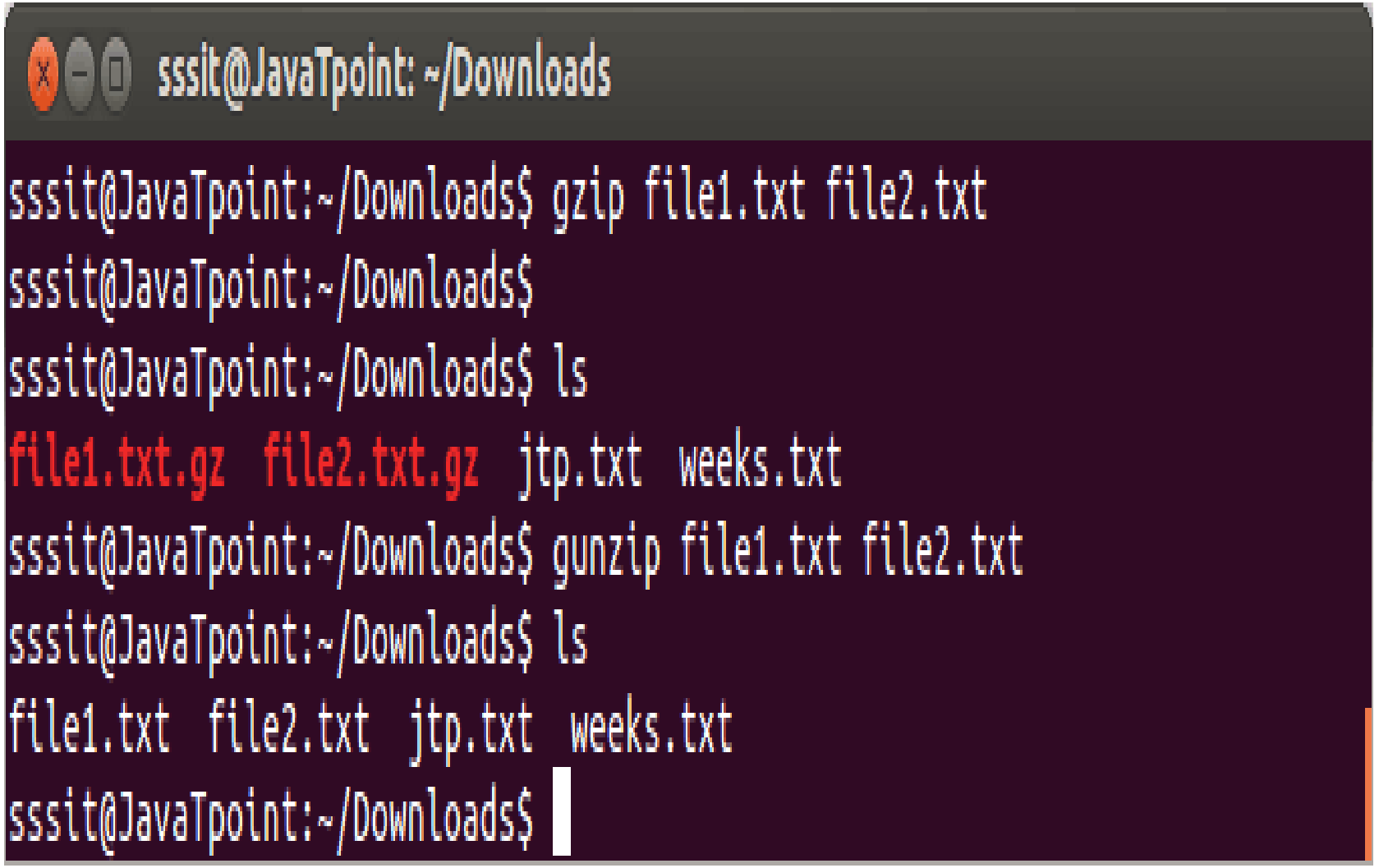
- Gzip (GNU zip) is a compressing tool, which is used to truncate the file size.
- By default original file will be replaced by the compressed file ending with extension (.gz).
- To decompress a file you can use gunzip command and your original file will be back.

**Syntax:**

```
gzip <file1> <file2> <file3>. . .
gunzip <file1> <file2> <file3>. . .
```

**Example:**

```
gzip file1.txt file2.txt
gunzip file1.txt file2.txt
```


sssit@JavaTpoint: ~/Downloads
sssit@JavaTpoint:~/Downloads$ gzip file1.txt file2.txt
sssit@JavaTpoint:~/Downloads$
sssit@JavaTpoint:~/Downloads$ ls
file1.txt.gz file2.txt.gz jtp.txt weeks.txt
sssit@JavaTpoint:~/Downloads$ gunzip file1.txt file2.txt
sssit@JavaTpoint:~/Downloads$ ls
file1.txt file2.txt jtp.txt weeks.txt
sssit@JavaTpoint:~/Downloads$

**Compressing Multi Files Together**

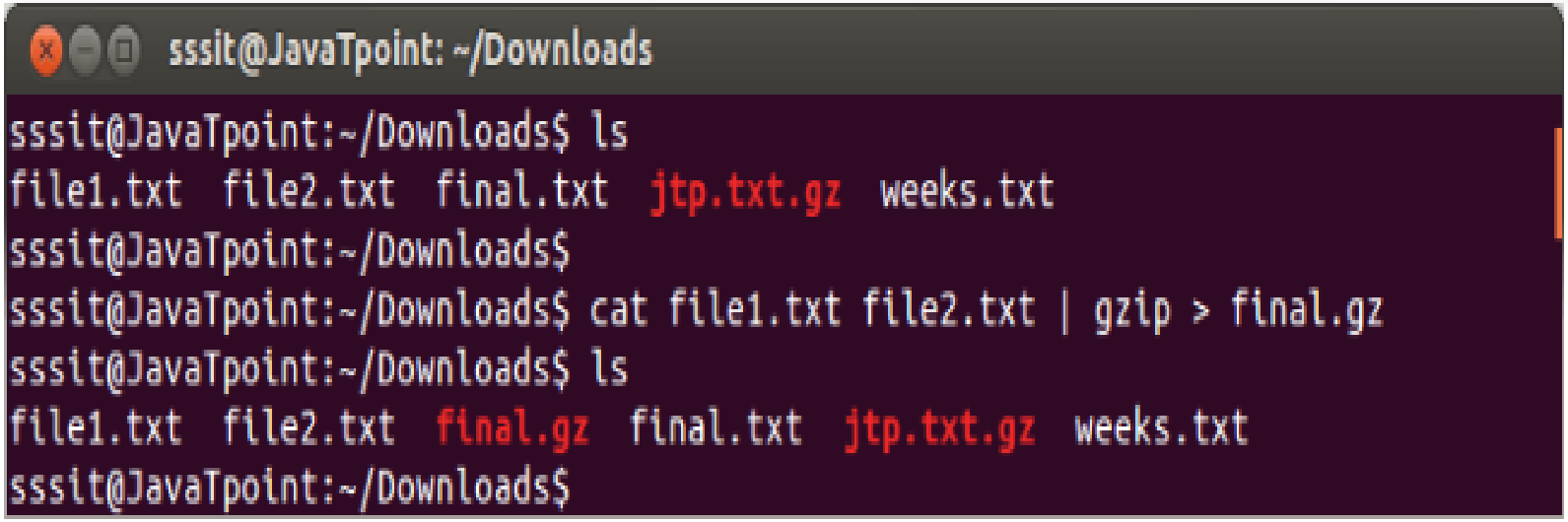
- If you want to compress more than one file together, you can use 'cat' and gzip command with pipe command.

**Syntax:**

```
cat <file1> <file2>. . | gzip > <newFile.gz>
```

**Example:**

```
cat file1.txt file2.txt | gzip > final.gz
```



**gzip -l**

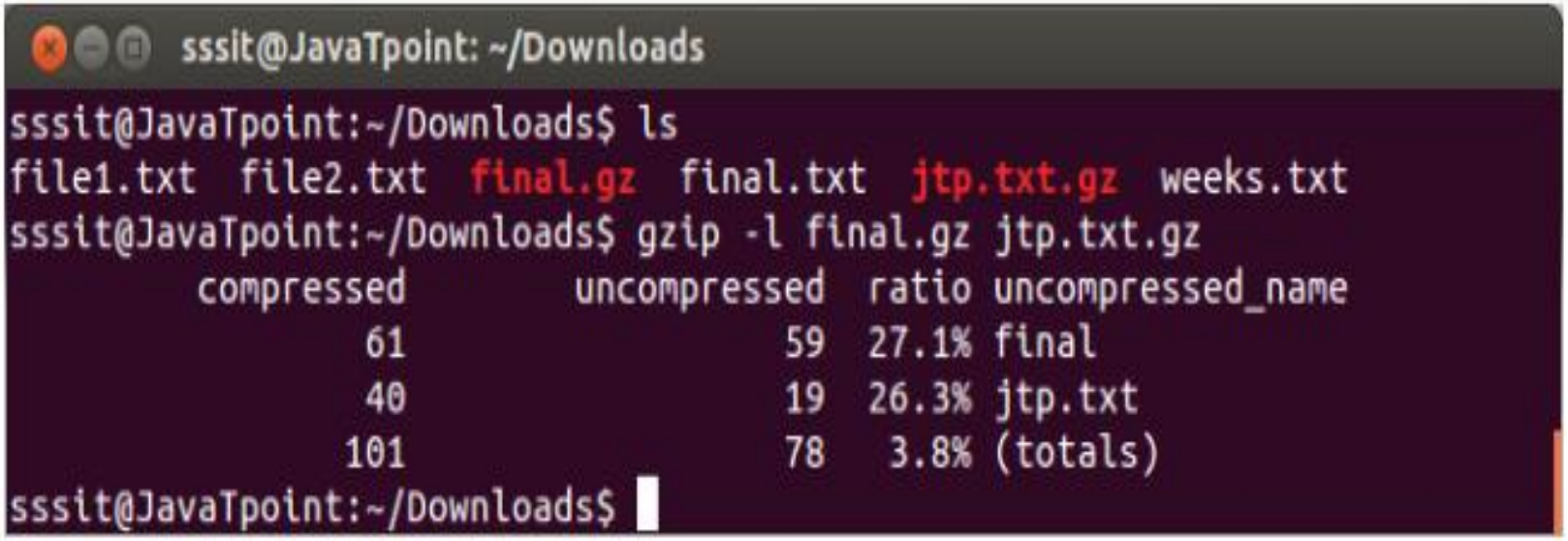
- The 'gzip -l' command tells about the compression ratio or how much the original file has compressed.

**Syntax:**

```
gzip -l <file1> <file2>. .
```

**Example:**

```
gzip -l final.gz jtp.txt.gz
```



## How To Compress A Directory

- The gzip command will not be able to compress a directory because it can only compress a single file. To compress a directory you have to use **'tar'** command.

**Syntax:**

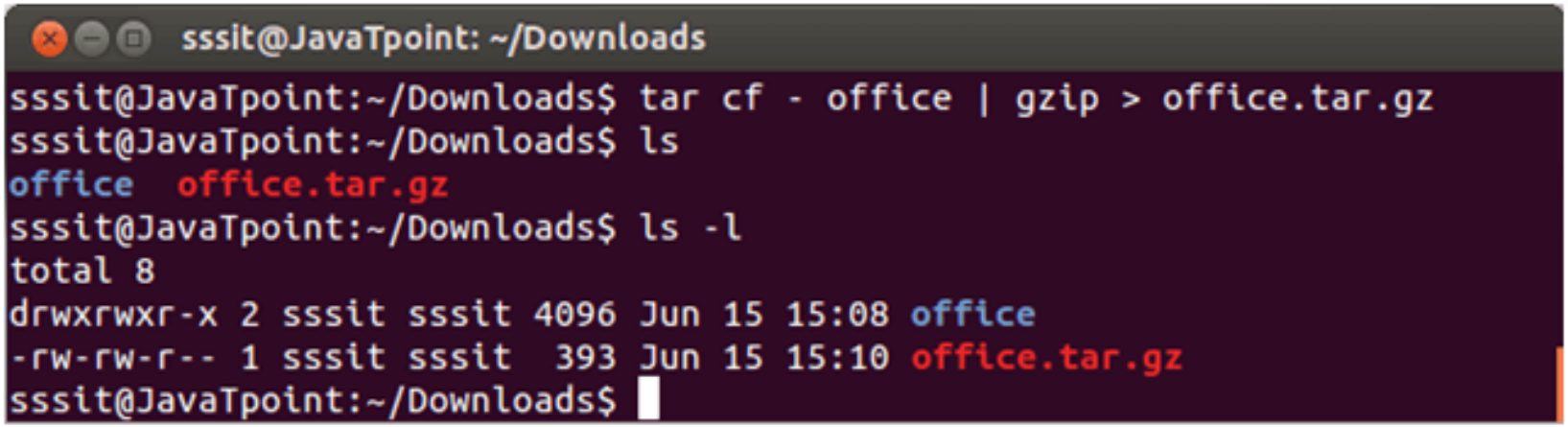
```
tar cf - <directory> | gzip > <directoryName>
```

**OR**

```
tar cvfz office.tar.gz office
```

**Example:**

```
tar cf - office | gzip > office.tar.gz
```



# 3. cpio

- **cpio** stands for "**copy in, copy out**". It is used for processing the archive files like **.cpio* or **.tar*.

- This command can copy files to and from archives.

- **Example:-**

- **Copy-out Mode:** Copy files named in name-list to the archive

  **Syntax:**

  cpio -o < name-list > archive

- **Copy-in Mode:** Extract files from the archive

  **Syntax:**

  cpio -i < archive

- **Copy-pass Mode:** Copy files named in name-list to destination-directory

  **Syntax:**

  cpio -p destination-directory < name-list

# UNIT 2
# WORKING WITH THE BOURNE SHELL

- What is Shell
- Shell Responsibilities
- Pipes and Input Redirection.
- Output Redirection.
- Here Document.
- The shell as A Programming Language.

- Shell Meta Characters
- Shell Variables.
- Shell Environment.
- Control Structures.
- Shell Script Examples.

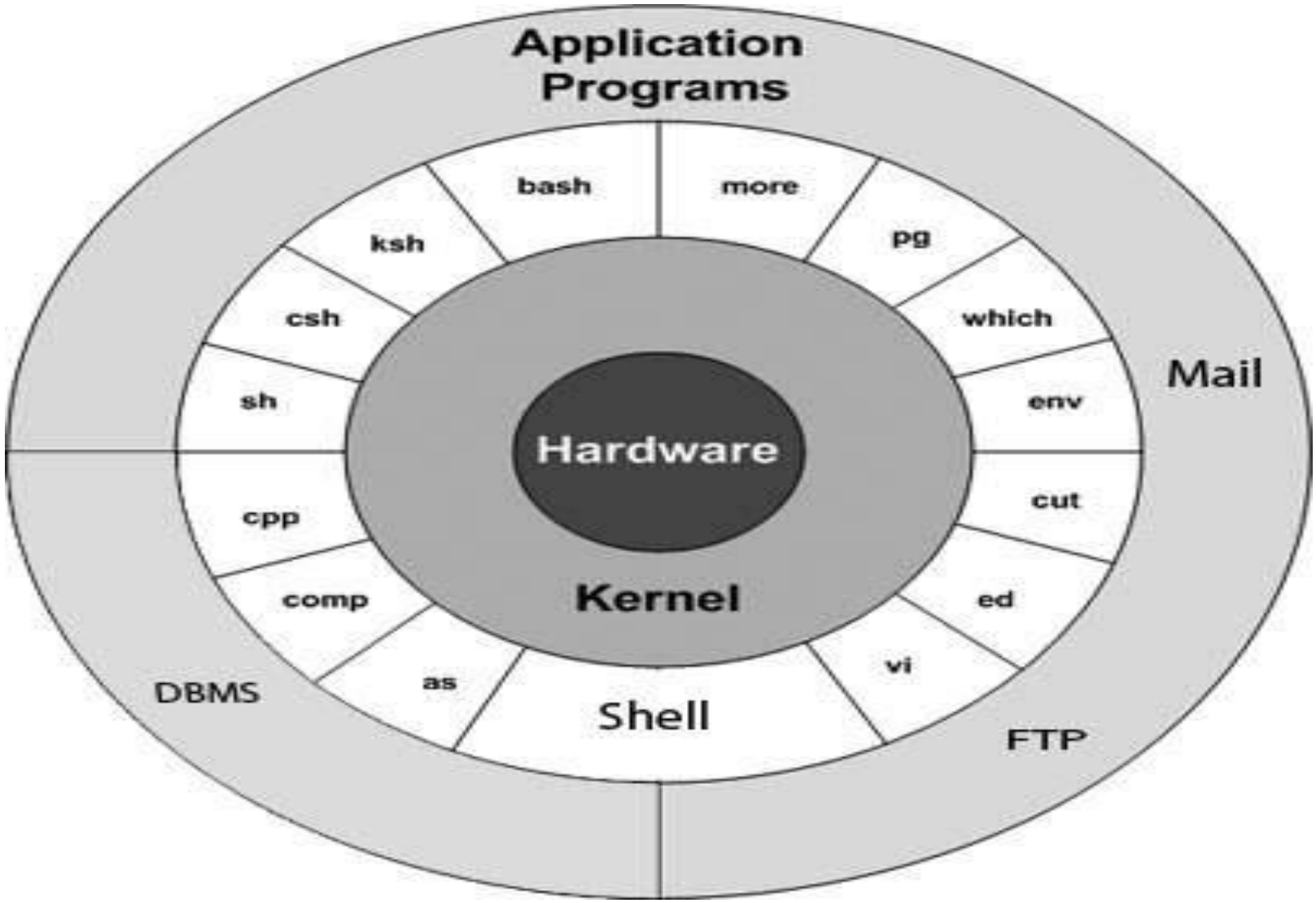
# Introduction

- When a Linux machine boots up, **init** process is initiated first then it executes the shell scripts in **/etc/rc.d** to restore the system configuration analyzing the behaviour of a system, and possibly modifying it.

- Writing shell scripts is not hard to learn, since only a fairly small set of shell-specific operations and options are to be learned.

# What is a shell

- The shell is the art of **UNIX** that is most visible to the user.
- It receives and interprets the commands entered by the user.
- To do anything in the system, we must give the shell a command.


Application Programs
bash
more
ksh
pg
csh
which
sh
env
Mail
Hardware
cut
cpp
comp
Kernel
ed
vi
as
DBMS
Shell
FTP

- If the command requires a utility, the shell requests that the kernel execute the utility.
- If the command requires an application program, the shell requests that it be run.

- There are two major parts to a shell. The first is the **interpreter.**
- The interpreter reads your commands and works with the kernel to execute them.
- The second part to the shell is a programming capability that allows you to write a **shell** (command) **script**.

- A **shell script** is a file that contains shell commands that perform a useful function. It is also known as a shell program.

- There are 4 major types of shells are used in UNIX today.

- Bourne Shell (sh)
- C Shell (csh)
- Korn Shell (ksh)
- Bourne Again Shell (bash)

- The Bourne shell, developed by **Steve Bourne** at the AT&T Labs, is the oldest. Because it is the oldest and most primitive, it is not used on many systems today.

- The C shell, developed in Berkeley by **Bill Joy,** received its name from the fact that its commands were supposed to look like C statements.

- The Korn shell, developed by **David Korn**, also of the AT&t Labs, is the newest and most powerful. Because it was developed at the AT&T Labs, it is compatible with the Bourne shell.

- The Bourne Again Shell, developed by **Steve Bourne** at the AT&T Labs. This shell is widely used with in the academic community.

- **bash** provides all the interactive features of the C shell (csh) and the Korn shell (ksh).

# Features of Bash

- Bash is **sh-compatible** as it derived from the original UNIX Bourne Shell.
- Bash can be **invoked by** single-character command line options (**-a, -b, -c, -i, -l, -r, etc.** )
- Bash **Start-up files** are the scripts that Bash reads and executes when it starts.
- Bash consists of **Key bindings** by which one can set up customized editing key sequences.
- Bash contains **one-dimensional arrays**
- Bash comprised of **Control Structures**

- Directory Stack in Bash specifies the history of recently-visited directories within a list.

- Example: **pushd** builtin is used to add the directory to the stack, **popd** is to remove directory from the stack and **dirs** builtin is to display content of the directory stack.

- Bash also comprised of restricted mode for the environment security. A shell gets restricted if bash starts with name **rbash**, or the bash --restricted, or bash -r option passed at invocation.

# Shell Responsibilities

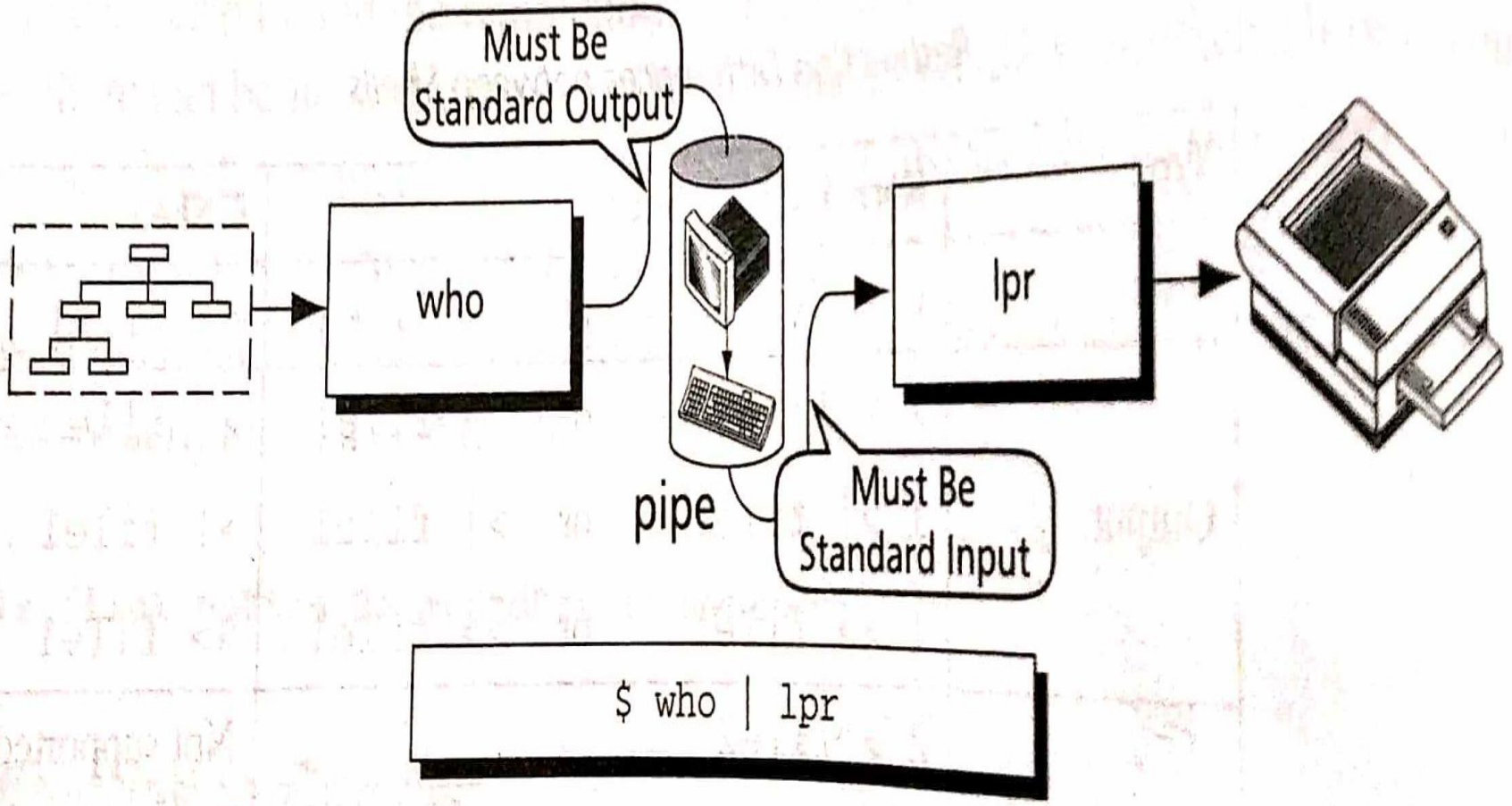
- Program Execution
- Variables & File name substitution
- I/O Redirection
- Pipeline Hookup
- Environment control
- Interpreted programming Language.

# Pipes

- We often need to use a series of commands to complete a task.
- For example, if we need to see a list of users logged into the system, we use the who command.
- However, if we need a hard copy of the list, we need two commands.

- First, we use **who** to get the list and store the result in a file using redirection.

- We can avoid the creation of the intermediate file by using a **pipe ( | )**.
- Pipe is an operator that temporarily saves the output of one command in a buffer that is being used at the same time as the input of the next command.


Must Be
Standard Output
who
pipe
lpr
Must Be
Standard Input
$ who | lpr

- The first command must be able to send its output to standard output; the second command must be able to read its input from standard input.

- General command line is as follows:
- Command1 | Command2 | Command3 | Command4

- Example: -

1. ls –l | more
2. who | lpr
3. who | more

- Pipe is an operator, not a command

# Redirection

- **Redirection** is the process by which we specify that a file is to be used in place of one of the standard files.

- With input files, we call it input redirection; with output files, we call it output redirection.

# Input Redirection

- Some commands are designed to take their input as a stream.
- This stream represents the standard input to a command.
- Standard input stream has 3 sources:

1. File
2. Keyboard (default)
3. Pipe line

- The input redirection operator is the less than character (<).
- An arrow pointing to a command, meaning that the command is to get its input from the designated file.

- Syntax:- Command < file1
- Example: - wc –l < user

```
$ wc -l users
2 users
$
```

```
$ wc -l < users
2
$
```

- In the first case, wc knows that it is reading its input from the file users.

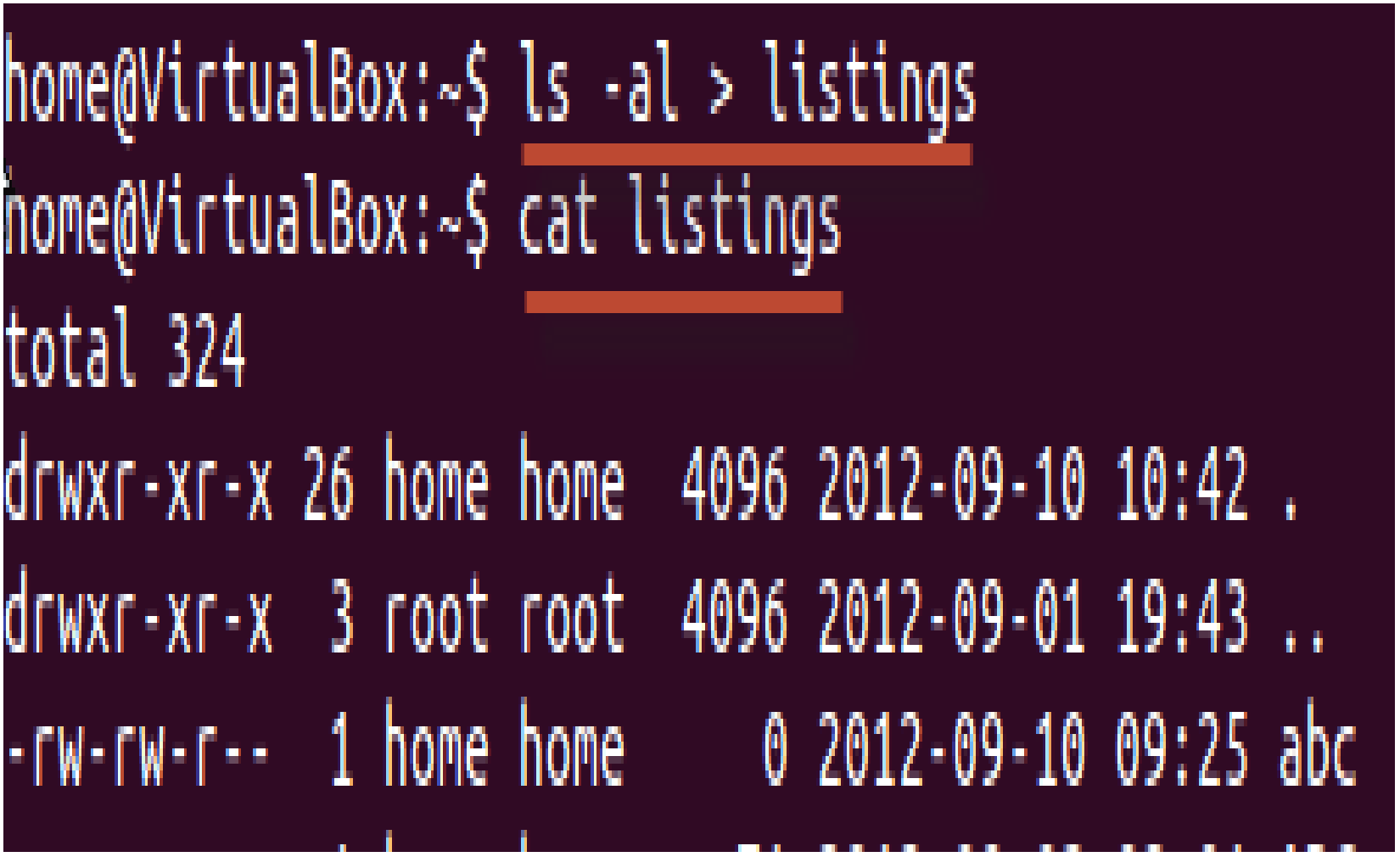
- In the second case, it only knows that it is reading its input from standard input so it does not display file name.

# Output Redirection

- When we redirect standard output, the command's output is copied to a file rather than displayed on the monitor.

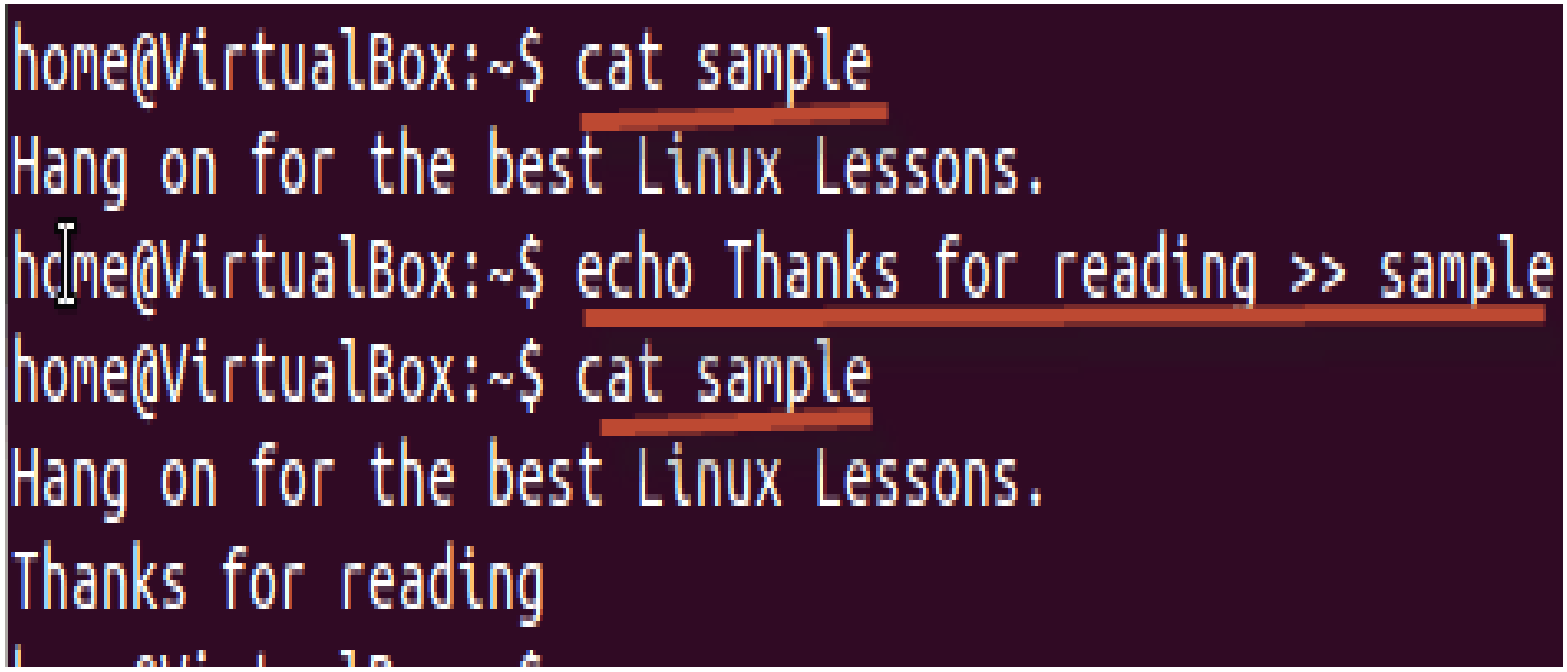
- There are two basic redirection operators for standard output: > & >>.

- The default redirection output is **terminal**.
- If you want the file to contain only the output from this execution of the command, you use one greater than token (>).
- In this case, when you redirect the output to a file that doesn't exist, UNIX creates it and writes the output.
- **Eg:** $ who > sample


home@VirtualBox:~$ ls -al > listings
home@VirtualBox:~$ cat listings
total 324
drwxr-xr-x 26 home home  4096 2012-09-10 10:42 .
drwxr-xr-x  3 root root  4096 2012-09-01 19:43 ..
-rw-rw-r--  1 home home     0 2012-09-10 09:25 abc

- On the other hand, if you want to append the output to the file, the redirection token is two greater than characters (>>).

- **Eg :** $ cat file2 >> file1


home@VirtualBox:~$ cat sample
Hang on for the best Linux Lessons.
home@VirtualBox:~$ echo Thanks for reading >> sample
home@VirtualBox:~$ cat sample
Hang on for the best Linux Lessons.
Thanks for reading

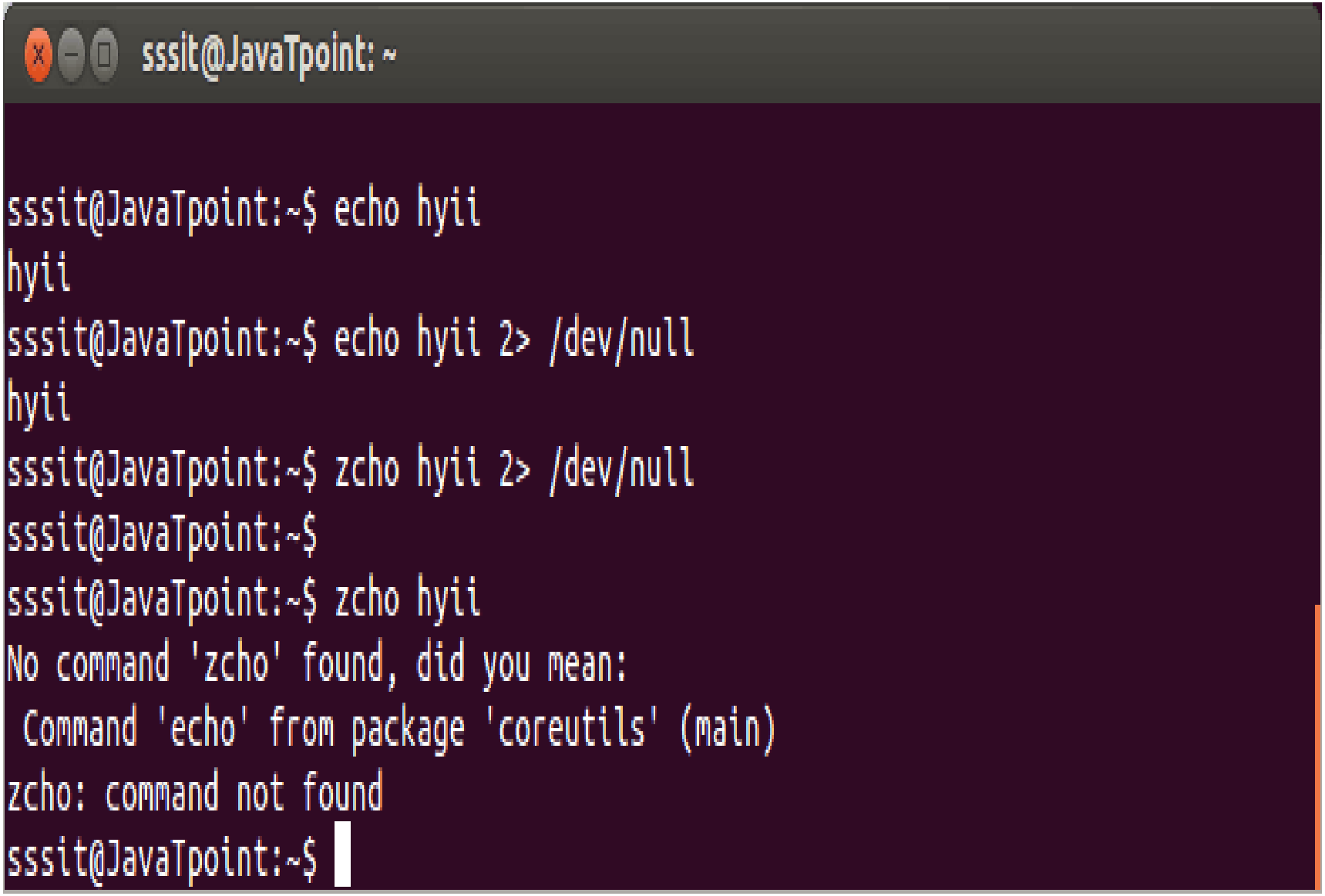
- **Disadvantages of I/O Redirection**

- Creation of temporary files.
- Memory wastage.
- Process time becomes slow

# Error Redirection

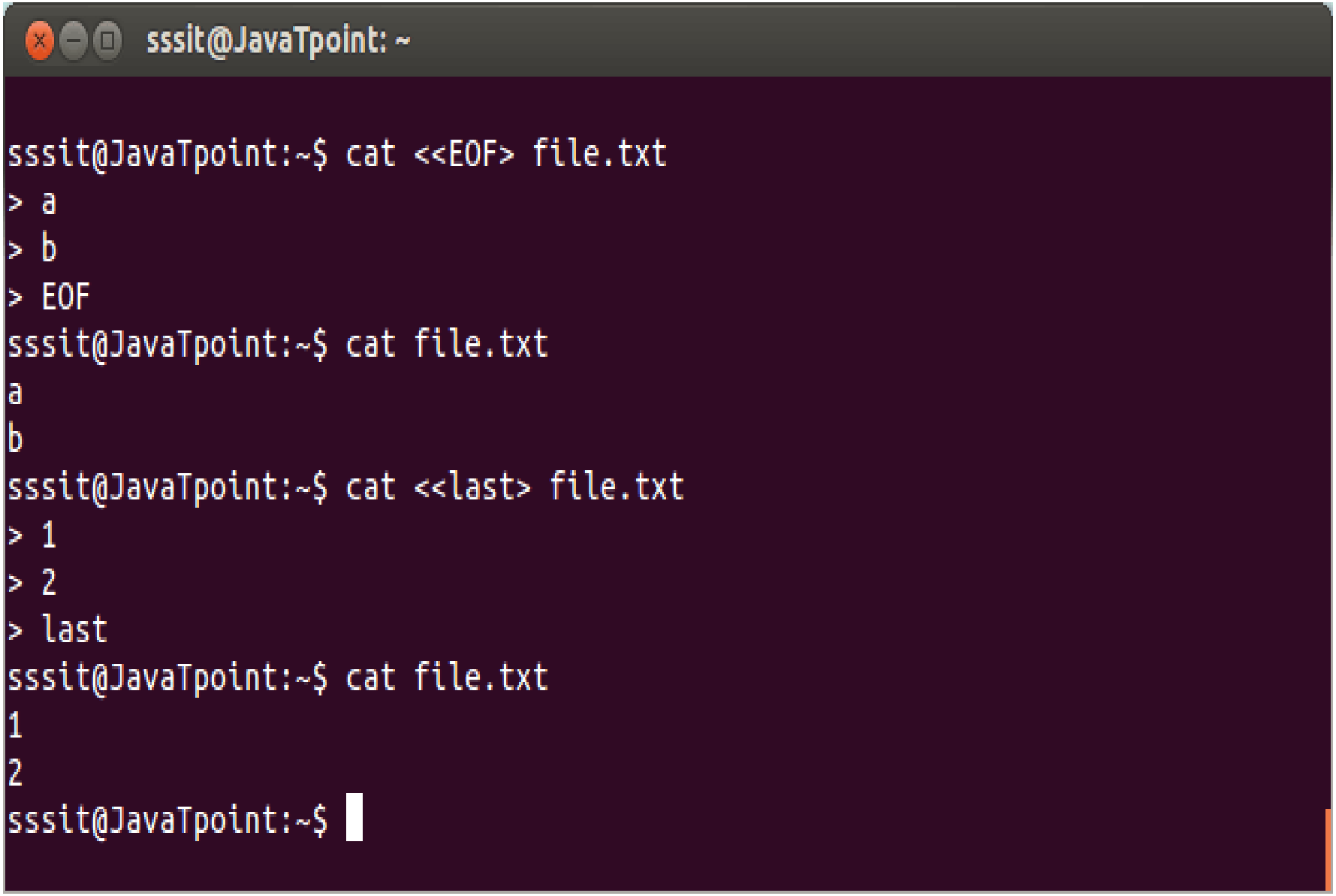
- Command '2>' redirects the error of an output.

- It helps us you to keep our display less messy by redirecting error messages.


sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ echo hyii
hyii
sssit@JavaTpoint:~$ echo hyii 2> /dev/null
hyii
sssit@JavaTpoint:~$ zcho hyii 2> /dev/null
sssit@JavaTpoint:~$
sssit@JavaTpoint:~$ zcho hyii
No command 'zcho' found, did you mean:
Command 'echo' from package 'coreutils' (main)
zcho: command not found
sssit@JavaTpoint:~$

# Here Document (<<)

- When we want to include the text in the script itself rather than read it from a file.
- This is done with the here document operator (<<).


sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ cat <<EOF> file.txt
> a
> b
> EOF
sssit@JavaTpoint:~$ cat file.txt
a
b
sssit@JavaTpoint:~$ cat <<last> file.txt
> 1
> 2
> last
sssit@JavaTpoint:~$ cat file.txt
1
2
sssit@JavaTpoint:~$

# Shell as a Programming Language

- Group of commands stored in a file is called **shell script or shell program.**

- Shell scripts are slower than compiled programs, but speed is not a constraint with certain jobs.

- Shell scripts are not recommended for number crunching.

- System administrator tasks are often best handled by shell scripts.

- The activities of the shell are not restricted to command interpretation alone.

- The shell has a whole set of internal commands that can be strung together as a language – with its own variables, conditionals and loops

# Shell meta characters

- There are some special characters that are recognized by the shell.
- **File substitution.**
- **I/O Redirection.**
- **Quoting.**
- **Process Execution**
- **Positional parameters**
- **Special parameters**

- **File substitution.**

1. ‘* ’**-** This is a ‘wildcard’, it matches any string of zero or more characters, except a leading ‘.’

   **Example: -** ls *.txt

   This will list all the files in the current directory that end with a .txt

2. **‘? ’-** This will match any single character.

   **Example: -** ls file?.txt

   This will find the files such as file1.txt, file2.txt etc.

- **I/O Redirection**

1. **>** - to redirect standard output to a file.
2. **<** - to take input from standard input devices.
3. **<<** - to give input from the terminal.
4. **>>** - to redirect output and append a file which contain data.

- **Process Execution**

1. **; -** It is used to execute more than one command.

   **Example: -** $ date ; cat file1
2. **() -** It is used to grouping more than one command.

   **Example: -** #(date;cat file) ; ls
3. **& -** to execute commands in background mode

   **Example: -** $ ls &

4. **&& -** If we pass two commands for this if first command is successfully executed then only it will execute the second command.

   **Example: -** $ ls file && echo file found

5. **|| -** it will executes second command if first command fails.

   **Example: -** $ ls file || echo not found

- **Quoting:**

1. **\ (back slash):** it negates the special property of the single character followed it.

   **Example:** - echo *

   It neglects the properties of * and displays the * as output

**2. ‘ ‘ :** Negates the special properties of all enclosed characters

Example: -

$ x=hello

$ echo ‘ < > $x ? & ‘

The output is : < > $x ? &.

3. “ “ : Negates the special properties of all enclosed characters except $, ` , \

**Example: -**

$ x=hello

$ echo “< > $x ? & “

The output is : < > hello ? &.

- **Positional parameters:** These are used to pass the parameter for shell script programs.

1. **$0 -** name of the command or script name.
2. **$* or $@ -** gives list of arguments.
3. **$# -** gives number of arguments.
4. **$1,$2,… -** first argument and second argument respectively.

# Shell variables

- A variable is a location in memory where values can be stored.
- Each shell allows us to create, store, and access values in variables.
- Each shell variable must have a name.

- The name of a variable must start with an alphabetic or underscore (_) character.
- It then can be followed by zero or more alphanumeric or underscore characters.
- There are two broad classifications of variables: user-defined and predefined.

- **User-defined variables:**
- User defined variables are not separately defined in UNIX.

- The first reference to a variable establishes it.

- The syntax for storing values in variables is the same for the Korn and Bash shells, but it is different for the C Shell.

- **Eg:** $x=10

  $ echo $x

  Output :  10.

- $ x=UNIX

  $ y=$x

  $ echo $y

  Output :  UNIX.

- For removing variables which we are defined syntax is
- $ unset variablename

- **Predefined variables:**
- Predefined variables are used to configure a user's shell environment.

- Predefined variables can be divided into two categories:

- shell variables and environmental variables.

- The shell variables are used to customize the shell itself.

- The environmental variables control the user environment and can be exported to sub shells.

- **$set** is used to display all predefined variables available in shell.

- **Shell Commands:-**

- **read:** Read values for variables; white space separated words
- **Eg:** $ read name

  Paul

  $ echo $name

  **Output:** Paul

- **set:** Display the values of all shell/system variables or predefined values and set is also used to assign values to the positional parameters.

  **Eg:** $ set `date`

  $ echo $@ or echo $*

  Output: Thu Sep 8 18:08:40 EDT 2011

- $ echo $1

  Output: Thu

- $ shift 1 // this command is used for shifting to next field

  $ echo $1

  **Output:** Sep

- **Eg:** $ set `cat f1`
- $ echo $#
- $ echo $2

- **#:** Used for comments in Shell Programming

- **printf:** printf command is used to print the code formats just like in your C
- **Example:**

  printf "sum is : %d" 100

  **Output :** 100

- **expr:** expr performs four basic arithmetic operations and the modulus function. It handles only integers, decimal portions are simple truncated or ignored.

- $x=3 y=5
- $expr 3 + 5
- output: 8

- $expr $x + $y
- output: 8

- $z=`expr $x + $y` ; echo $z
- output: 8

- $x=`expr $x + 1`
- $echo $x
- output: 4

# The Environment

- An important UNIX concept is the environment, which is defined by environment variables.

- These variables control the behavior of the system.

- Some are set by the system, others by you, yet others by the shell, or any program that loads another program.

- $env command displays only environment variables.

| Variable | Significance |
| --- | --- |
| HOME | home directory |
| PATH | Search path for commands |
| USER | login name |
| LOGNAME | as above |
| TERM | Terminal type |
| SHELL | Users login shell |

# Control Structures

- Conditional Control Structures:

1. if,
2. if-else,
3. elif,
4. case

# if

- **if: if** command is executed if it test condition is true
- **Syntax :**

**if** command is successful

**then**

Command

**fi**

# if-else

- **if-else** executes an action if the exit status of its test command is true; if false, then the **else** action is executed.
- **Syntax :**

**if** command is successful
then
    command
else
    command
fi

# elif

- **elif** allows you to nest **if** structures, enabling selection among several alternatives; at the first true **if** structure, its commands are executed and control leaves the entire **elif** structure.

- **Syntax:**

**if** command is successful

**then**

command

**elif** command is successful

**then**

command

**else**

command

**fi**

- **Using test or [ ] to Evaluate Expressions:**
- The **if** conditional can’t handle relational tests directly, but only with assistance of the **test** statement.
- **test** uses certain operators to evaluate the condition on its right and returns an exit status, which is used by **if** for making decisions.
- **test** works in 3 ways:

- Compares two numbers (like test $x –gt $y or [ $x –gt $y ]).
- Compares two strings or a single one for a null value (like test $x = $y).
- Checks a file's attributes (like test –f $file).

- Numerical comparison operators used with **test:**

| **Operator** | **Meaning** |
| --- | --- |
| -eq | Equal to |
| -ne | Not equal to |
| -gt | Greater than |
| -ge | Greater than or equal to |
| -lt | Less than |
| -le | Less than or equal to |

- String tests with **test:**

| **Test** | **True if** |
| --- | --- |
| s1 =s2 | String s1 = s2 |
| s1 != s2 | String s1 is not equal to s2 |
| -n stg | String stg is not a null string |
| -z stg | String stg is a null string |
| stg | String stg is assigned and not null |
| s1 == s2 | String s1 = s2 (Korn and Bash only) |

- File Attribute Testing with **test:**

| **Test** | **True If File** |
|---|---|
| -f file | file exists and is a regular file |
| -r file | file exists and is a readable |
| -w file | file exists and is a writable |
| -x file | file exists and is executable |
| -d file | file exists and is a directory |
| -s file | file exists and has a size greater than zero |
| -e file | file exists (korn & bash only) |
| -L file | file exists and is a symbolic link (korn & bash only) |
| f1 -nt f2 | f1 is newer than f2 (korn & bash only) |
| f1 -ot f2 | f1 is older than f2 (korn & bash only) |
| f1 -ef f2 | f1 is linked to f2 (korn & bash only) |

- **Eg:**

```
if [ -e file ]
then
        echo “File exists”
 fi
```

# case

- **case** matches the string value to any of several patterns. If a pattern is matched, its associated commands are executed.

- **Syntax: -**

- **case** expression **in**
  pattern 1)  command 1;;
  pattern 2)  command 2;;
  *) command;;
  **esac**

```
echo  -e “Menu \n 1. List of files \n 2. Todays Date \n 3. Users \n 4. Exit \n
Enter your choice: ”
read choice
case $ choice in
1)      ls;;
2)      date;;
3)      who;;
4)      exit;;
*)  echo “Invalid Option”
esac
```

# Loop Control Structures

- while,
- For
- until

# While

- **while** executes an action as long as its test command is true.

  **Syntax:**

  **while** condition is true

  **do**

  command

  **done**

- **Eg:**

```
while [ $x -eq $y ]
do
    echo   $x
done
```

# For in

- **for-in** is designed for use with lists of values; the variable operand is consecutively assigned the values in the list.
- **Syntax:**

  **for** variable **in** list

  **do**

  command
  **done**

- **Eg:**

```
for i in 1 2 3
do
echo $i
done
```

**Output:** 1 2 3

# Until

- **until** executes an action as long as its test command is false.

  **Syntax:**

  **until** command

  **do**

  command

  **done**

- **Eg:**

```
until [ $x -eq $y ]
do
    echo   $x
done
```

- **break:** break is designed for breaking the looping statements

- **Eg:**

```
for i in 1 2 3 4 5
do
if [ $ i -eq 3 ];
then
        break;
fi
echo $i
done
```

**Output:** 1 2

- **continue:** continue is designed to continue the loop at specific condition

  **Eg:**

```
for i in 1 2 3 4 5
do
if [ $ i – eq 3 ];
then
        continue;
fi
echo $i
done
```

  **Output:** 1 2 4 5

# Shell Script Example

```
echo PROGRAM TO FIND BIGGEST OF 3 NUMBERS
echo Enter 3 numbers
read a
read b
read c
```

```
if [ $a -ge $b ] && [ $a -ge $c ]
then
echo $a is big
elif [ $b -ge $c ]
then
echo $b is big
else
echo $c is big
fi
```