**BASIC COMMANDS IN LINUX**

**AIM:** To study and execute the commands in UNIX

**A.**

**FILE RELATED COMMANDS:**

1. **Pwd Command:**

**DESCRIPTION:** It displays the present working directory

**SYNTAX:** pwd

**OUTPUT:**

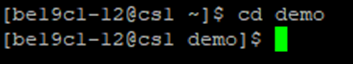


1. **Cd Command:**

**DESCRIPTION:** It changes the working directory i.e., change the current working to some other folder.

**SYNTAX:** cd [option] [directory]

**OUTPUT:**



1. **Ls Command:**

**DESCRIPTION:** It lists the content/files listed in a particular UNIX directory

**SYNTAX:** ls [options] [filename]

**OUTPUT:**



1. **Rm command:**

**DESCRIPTION:** It removes files or directories

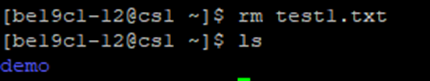
**SYNTAX:** rm [option …] filelist

**OUTPUT:**

Before using rm command:



After using rm command:



1. **Mv command:**

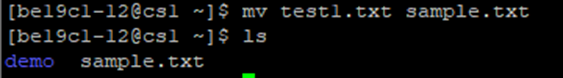
**DESCRIPTION:** It is used to move files or directories from one place to another or renames the files.

**SYNTAX:** mv [option] [sourceFile] [destFile]

If destination file doesn’t exist then it will be created else it will overwrite and the source file is deleted.

**OUTPUT:**

Renaming the file present in directory



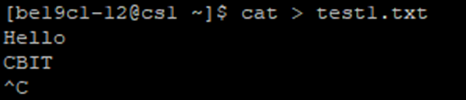
1. **Cat Command:**

**DESCRIPTION:** It allows us to create single or multiple files, view content of file, concatenate files and redirect output in terminal or files.

**SYNTAX:** cat [option …] [file …]

**OUTPUT:**

Text file is created

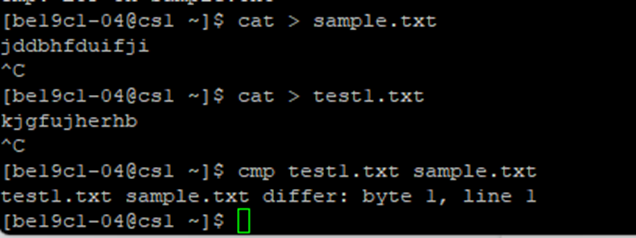


1. **Cmp Command:**

**DESCRIPTION:** It compares two files, and if they differ, tells the first byte and line number where they differ.

**SYNTAX:** cmp [option …] fromfile tofile

**OUTPUT:**

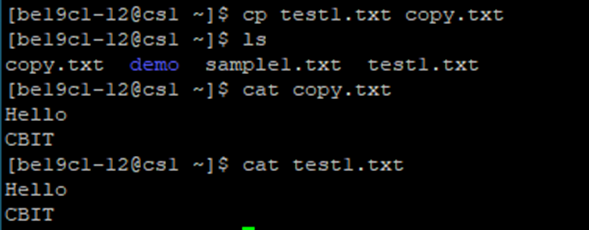


1. **Cp Command:**

**DESCRIPTION:** It copies the files

**SYNTAX:** cp [option] source destination

**OUTPUT:**



1. **Echo Command:**

**DESCRIPTION:** It is used for displaying a line of string/text that is passed as the arguments

**SYNTAX:** echo [option] [string]

**OUTPUT:**

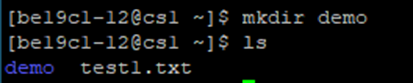


1. **Mkdir Command:**

**DESCRIPTION:** It created new directory

**SYNTAX:** mkdir [options] DirectoryNames

**OUTPUT:**

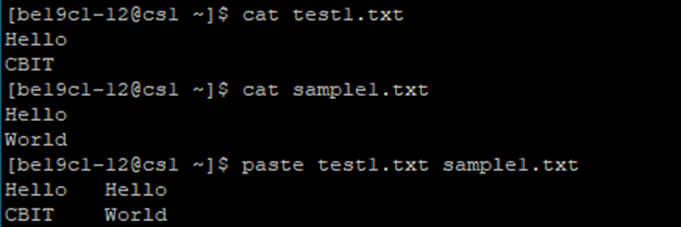


1. **Paste Command:**

**DESCRIPTION:** It merges the lines from multiple files. It sequentially writes the corresponding lines from each file separated by a TAB delimiter on the UNIX terminal

**SYNTAX:** paste [options] files-list

**OUTPUT:**

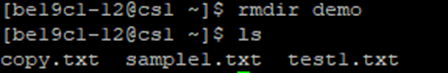


1. **Rmdir Command:**

**DESCRIPTION:** It removes the directories

**SYNTAX:** rmdir [options] DirectoryNames

**OUTPUT:**

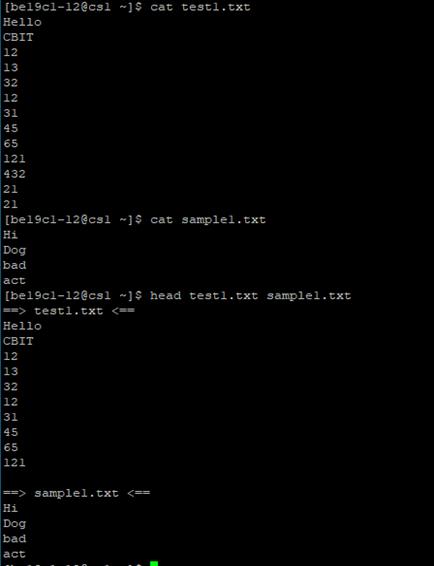


1. **Head Command:**

**DESCRIPTION:** It displays the beginning of a text file or piped data. It prints the first 10 lines of the specified files.

**SYNTAX:** head [option] [files]

**OUTPUT:**

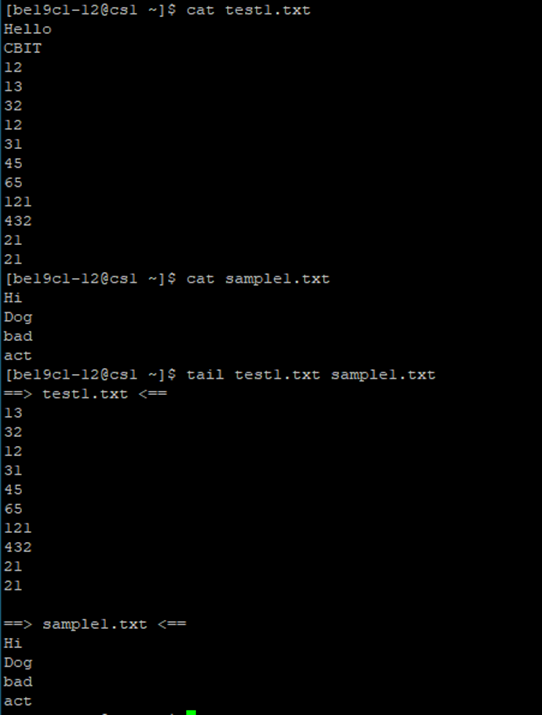


1. **Tail Command:**

**DESCRIPTION:** It displays the tail end of a text file or piped data. It prints the last 10 line sof the specified file.

**SYNTAX:** tail [option] [files]

**OUTPUT:**



1. **Date Command:**

**DESCRIPTION:** It is used to display date, time, time zone, etc. It is also used to set the date and time of the System. It is used to display the date in different formats and calculate dates over time.

**SYNTAX:** date [option] [+format]

**OUTPUT:**

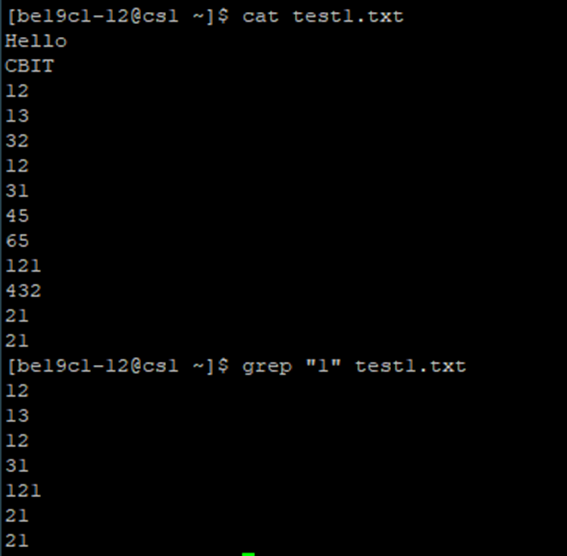


1. **Grep Command:**

**DESCRIPTION:** It prints lines matching a pattern

**SYNTAX:** grep [options] PATTERN [file]

**OUTPUT:**

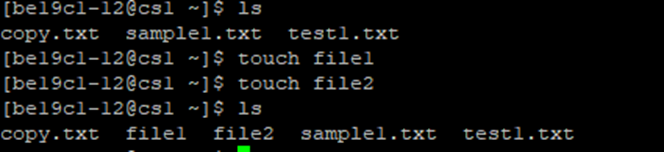


1. **Touch Command:**

**DESCRIPTION:** It creates a new file or update its timestamp

**SYNTAX:** touch [option …] [File …]

**OUTPUT:**



1. **Chmod Command:**

**DESCRIPTION:** It is used to change the access permissions, change mode

**SYNTAX:** chmod [options] Mode File

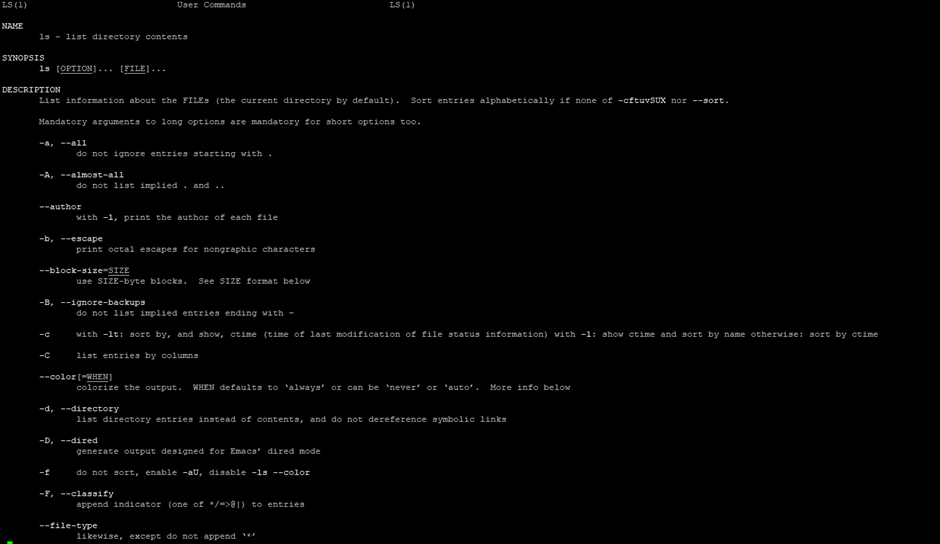
**OUTPUT:**Giving access of editing to anyone



1. **Man Command:**

**DESCRIPTION:** It is the interface to view the system’s reference manuals.

**SYNTAX:** man ls

**OUTPUT:**

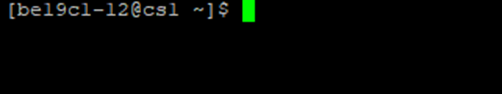
1. **Clear Command:**

**DESCRIPTION:** It clears the terminal Screen

**SYNTAX:** clear

**OUTPUT:**

After entering the clear command entire terminal will be cleared

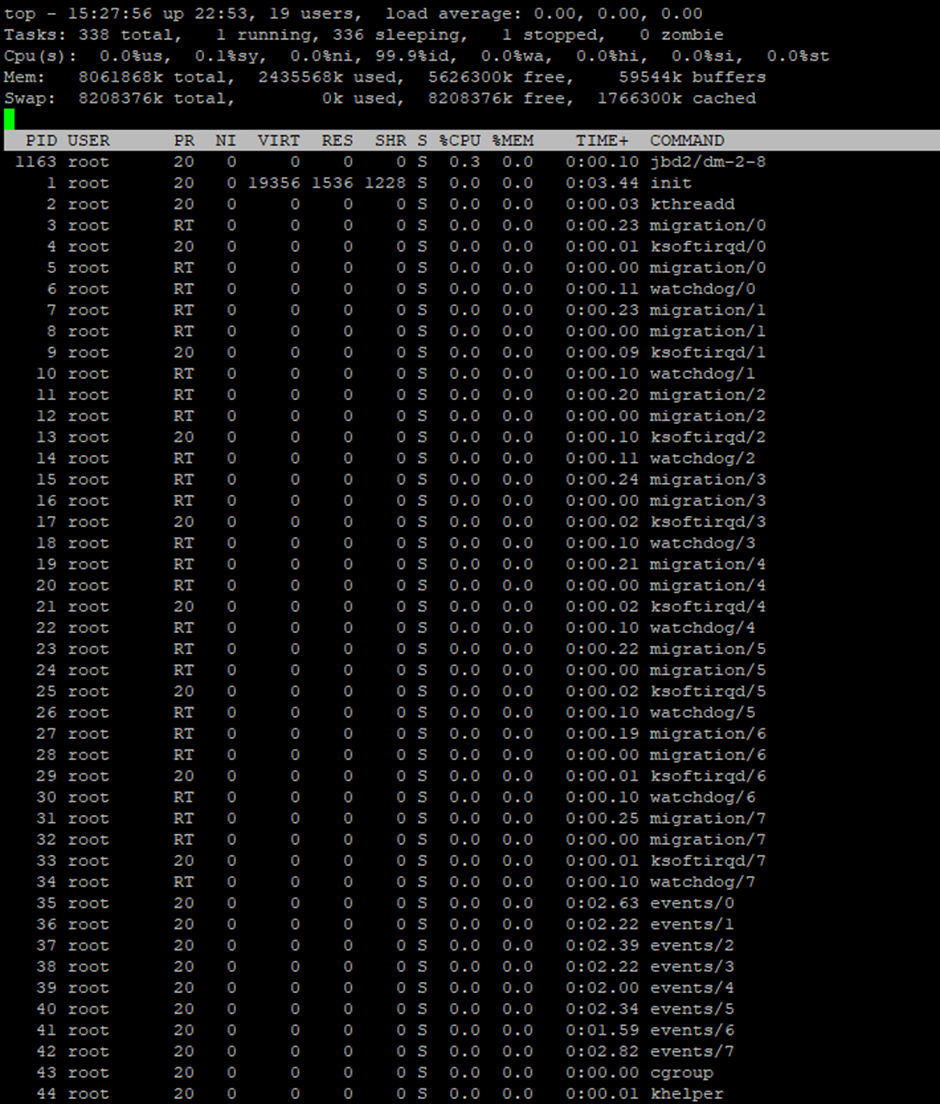


**PROCESS RELATED COMMANDS:**

1. **Top Command:**

**DESCRIPTION:** To track the running processes on our machine

**SYNTAX:** top

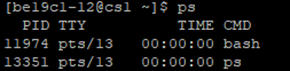
**OUTPUT:**

1. **Ps Command:**

**DESCRIPTION:** ps is short for Process status. It displays the currently-running processes.

**SYNTAX:** ps

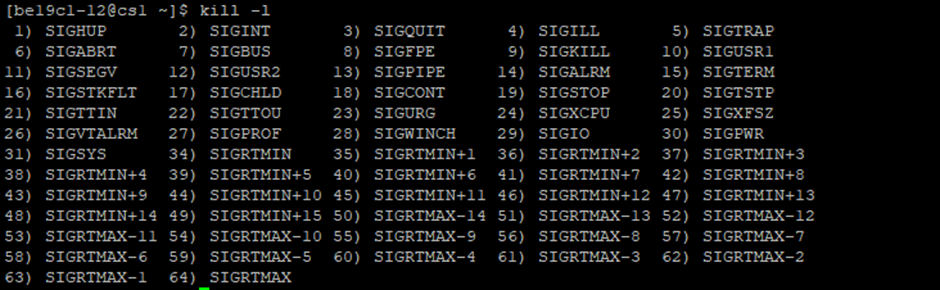
**OUTPUT:**



1. **Kill Command:**

**DESCRIPTION:** It sends a signal to the process. There are different types of signals that we can send

**SYNTAX:** kill [number]

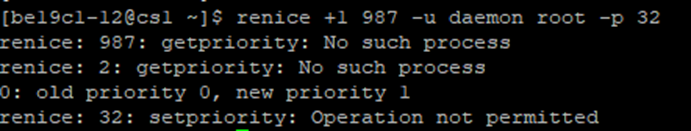
**OUTPUT:**  
List of different types of signals

1. **Nice Command:**

**DESCRIPTION:** The Nice command configures the priority of a Linux process before it is started. The renice command sets the priority of an already running process.

**SYNTAX:** nice -nice\_value command-arguments; renice -n nice\_value –p pid\_of\_the\_process

**OUTPUT:**

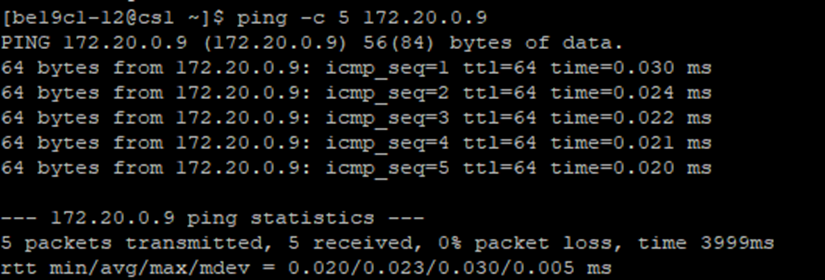


**NETWORK RELATED COMMANDS:**

1. **Ping Command:**

**DESCRIPTION:** It allows us to test the reachability of a device on a network

**SYNTAX:** ping [host]

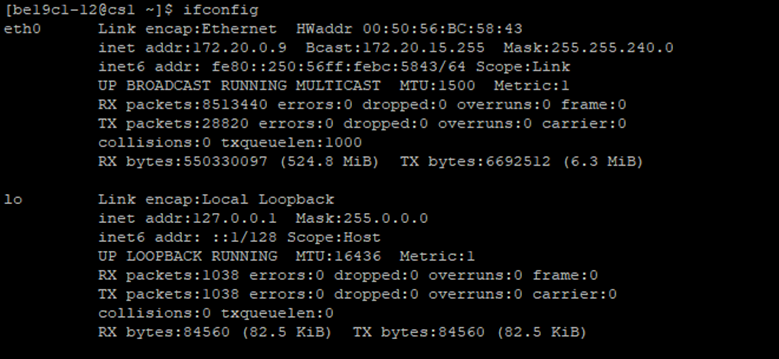
**OUTPUT:**

1. **Ifconfig Command:**

**DESCRIPTION:** The command ifconfig stands for interface configurator. This command enables us to initialize an interface, assign IP address, enable or disable an interface. It displays route and network interface.

**SYNTAX:** ifconfig

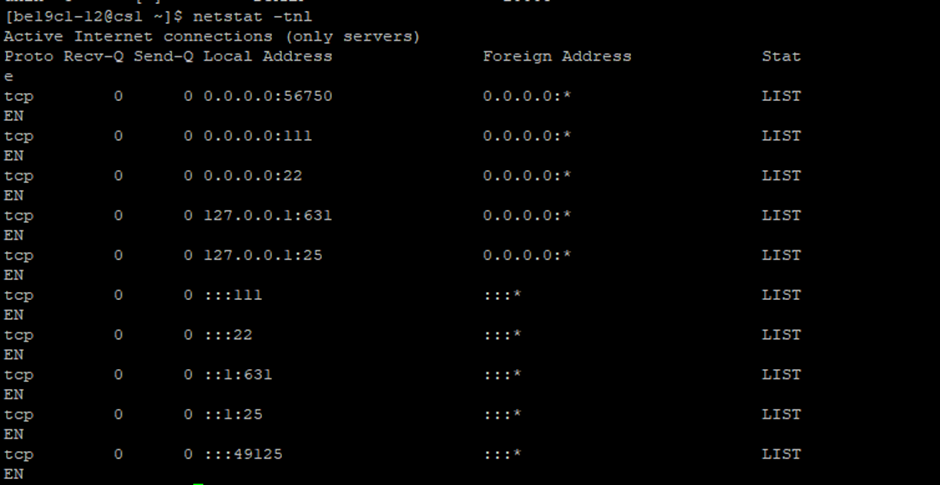
**OUTPUT:**



1. **Netstat Command:**

**DESCRIPTION:** The Netstat command displays active TCP connections, ports on which the computer is listening. The information this command provides can be useful in pinpointing problems in our network connections.

**SYNTAX:** netstat

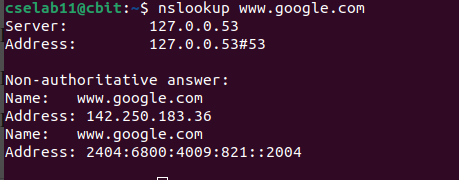
**OUTPUT:**

1. **Nslookup Command:**

**DESCRIPTION:** NsLookUp command is used to find all the IP addresses for given domain name.

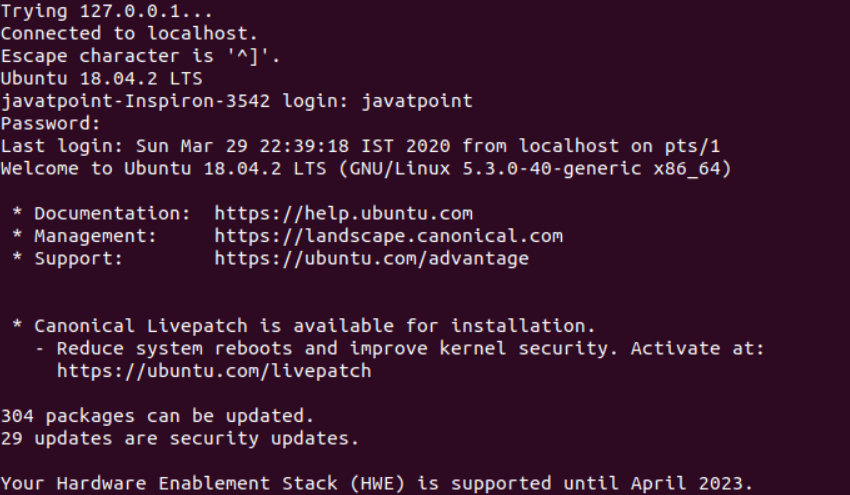
**SYNTAX:** nslookup host

**OUTPUT:**



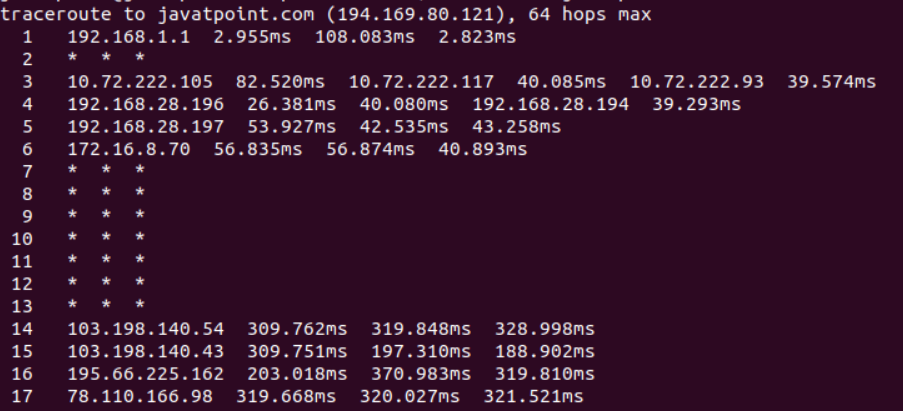
1. **Telnet Command:**

**DESCRIPTION:** It connects destination via the telnet protocol, if telnet connection establishes on any port means connectivity between two hosts is working fine.  
**SYNTAX:** telnet hostname/IP address

**OUTPUT:**

1. **Traceroute Command:**

**DESCRIPTION:** A handy utility to view the number of hops and response time to get to a remote system or website is a traceroute.  
**SYNTAX:** traceroute [OPTION...] HOST

**OUTPUT:**

**CONCLUSION:**

By executing the above commands, we have understood the commands.

**To Study about UNIX vi Editor and its features:**

* The default editor that comes with the UNIX operating system is vi editor (visual editor).
* Using vi editor, we can edit an existing file or create a new file from scratch.
* We can also use this editor to just read a text file
* Syntax: vi filename

**Modes of Operation in vi editor:**

* There are three modes of operation in vi:
* **Command Mode:**

1. When vi starts up, it is in Command Mode.
2. This mode is where vi interprets any characters we type as commands and thus does not display them in the window.
3. This mode allows us to move through a file, and to delete, copy, or paste a piece of text.
4. To enter into Command Mode from any other mode, it requires pressing the **[Esc]** key.
5. If we press [Esc] when we are already in Command Mode, then vi will beep or flash the screen

* **Insert Mode:**

1. This mode enables you to insert text into the file.
2. Everything that’s typed in this mode is interpreted as input and finally, it is put in the file.
3. The vi always starts in command mode.
4. To enter text, you must be in insert mode. To come in insert mode you simply type i.
5. To get out of insert mode, press the Esc key, which will put you back into command mode.

* **Last Line Mode (Escape Mode):**

1. Line Mode is invoked by typing a colon [:], while vi is in Command Mode.
2. The cursor will jump to the last line of the screen and vi will wait for a command.
3. This mode enables you to perform tasks such as saving files, executing commands.

**Vi Commands:**

* **To switch from command to insert mode:**
* i – Start typing before the current character
* l – Start typing at the start of current line
* a – Start typing after the current character
* A – Start typing at the end of current line
* o – Start typing on a new line after the current line
* O – Start typing on a new line before the current line
* **To move around a file:**
* j – To move down
* k – To move up
* h – To move left
* l – To move right
* **To delete:**
* x – Delete the current character
* X – Delete the character before the cursor
* r – Replace the current character
* xp – Switch two characters
* dd – Delete the current line
* D – Delete the current line from current character to the end of the line
* dG – Delete from the current line to the end of the file
* **To repeat or undo:**
* u – Undo the last command
* . – repeat the last command
* **To save and quit:**
* :wq – Save and quit
* :w – Save
* :q – Quit
* :w fname – Save as fname
* ZZ – Save and quit
* :q! – Quit discarding changes made
* :w! – Save (and write to non-writable file)
* **Command to cut, copy and paste:**
* dd – delete a line
* yy – (yank yank) copy a line
* p – Paste after the current line
* P – Paste before the current line
* <n>dd – Delete the specified n number of lines
* <n>yy – Copy the specified n number of lines
* **Start and end of line:**
* 0 – Bring at the start of the current line
* ^ - Bring at the start of the current line
* $ - Bring at the end of the current line
* d0 – Delete till start of a line
* d$ - Delete till end of a line
* **Joining lines:**
* J – Join two lines
* yyp – Repeat the current line
* ddp – Swap two lines
* **Move forward or backward:**
* w – Move one word forward
* b – Move one word backward
* <n>w – Move specified number of words forward
* dw – Delete one word
* yw – Copy one word
* <n>dw – Delete specified number of words

**SHELL SCRIPTING**

**SHELL:**

* Shell is a user program or its environment provided for user interaction.
* It is a command language interpreter that executes commands.
* It reads from the input from the user and executes programs based on the input
* It displays the program output when the program finishes the execution
* Shell is not part of system kernel, but uses the system kernel to execute programs, create files, etc.

**SHELL SCRIPT:**

* A shell script is a list of commands in a computer program that is run by the UNIX shell which is a command line interpreter.
* It usually has comments that describes the steps
* Operations performed by shell scripts are program execution, File manipulation and text printing

**VARIABLES:**

* The shell enables us to create, assign and delete variables.
* Variable names must contain only letters, numbers or the underscore
* Defining Variables:

Variable\_name = variable\_value

* Variable can be accessed by using a dollar sign ($) as a prefix for variable\_name
* After a variable is marked read-only, its value cannot be changed
* Unsetting a variable directs the shell to remove the variable from the list of the variables.
* Unset command cannot be used on the variables which are marked read-only

**READ STATEMENT:**

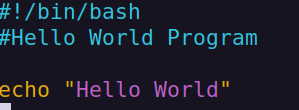
* The Read statement is used to get input from user and store the data to variable
* Syntax: read variable1, variable2, …., variable

**ECHO STATEMENT:**

* The Echo statement is used to display text or value of variables on the screen.
* Syntax: echo [options] [string, variables…]

**BASIC SHELL SCRIPT:**

* touch HelloWorld.sh ---> creates a file
* nano HelloWorld.sh ----> opens the file in text editor



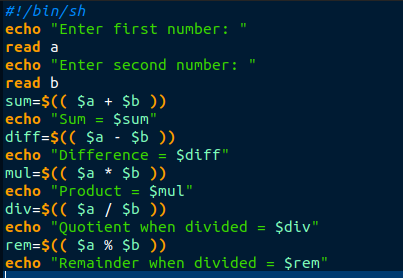
* chmod +x HelloWorld.sh ----> gives permission to execute the script
* ./HelloWorld.sh runs the script

**Output:**

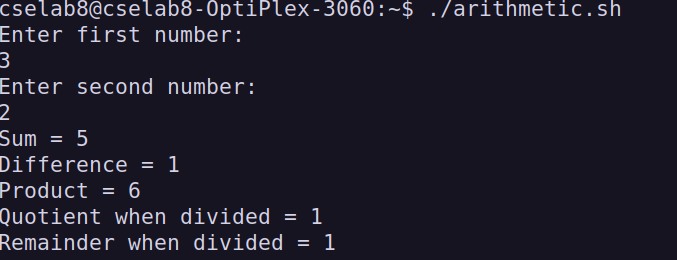
**ARITHMETIC OPERTIONS:**

* Demonstration for basic Arithmetic operations:

**Code:**



**Output:**



**DECISION MAKING STATEMENTS**

* There are two types of decision-making statements within shell scripting. They are:
* If-else statement
* Case-esac statement

**IF-ELSE STATEMENT:**

* It is a conditional statement.
* There are couple of varieties present within the if-else statement. They are:
* If-fi
* If-else-fi
* If-elif-else-fi
* Nested if-else
* **Syntax:**
* **If-fi**

if [ expression ]; then

statements

fi

* **If-else-fi**

if [ expression ]; then

statement1

else

statement2

fi

* **If-elif-else-fi**

if [ expression ]; then

statement1

elif [ expression ]

then

statement2

else

statement3

fi

* **Nested if-else**

if [ expression ]

then

statement1

if [ expression ]

then

statement

else

statement

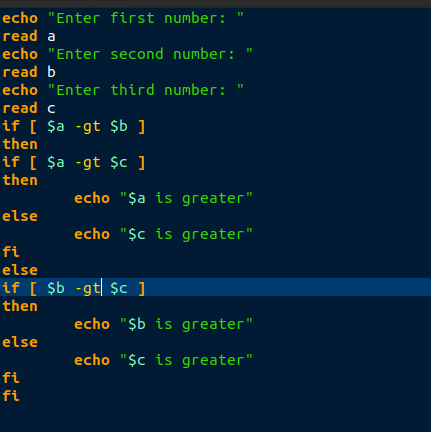
fi

else

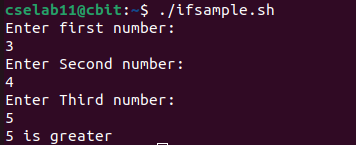
statement

fi

* **Demo Program for If-else Statement:**



**Output:**



**THE CASE-ESAC STATEMENT:**

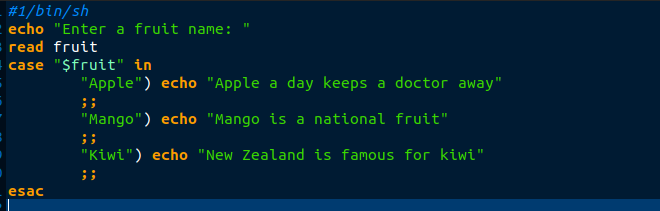
* Case-esac is basically working the same as switch statement in programming.
* **Syntax:**

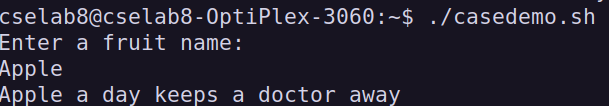
case $var in

Pattern 1) Statement 1;;

Pattern n) Statement n;;

esac

* **Demo for case-esac statement:**

**Output:**

**LOOPS:**

* There are 3 looping statements which can be used in bash programming
* While statement
* For statement
* Until statement

**FOR STATEMENT:**

* The for loop moves through a specified list of values until the list is exhausted.
* Syntax:

for varname in list

do

statement

done

**WHILE STATEMENT:**

* Linux scripting while loop is similar to C language while loop
* There is a condition in while and commands are executed till the condition is valid. Once condition becomes false, loop terminates.
* Syntax:

while [condition]

do

statement

done

**UNTIL STATEMENT:**

* The only difference is that until statement executes its code block while its conditional expression is false, and while statement executes its code block while its conditional expression is true
* Until loop always executes at least once.
* Loop while executes till it returns a zero value and until loop executes till it returns non-zero value.
* Syntax:

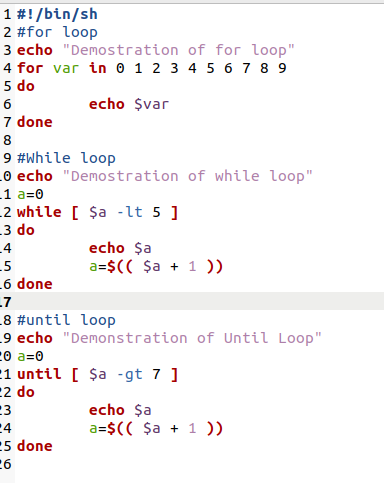
until [ condition ]

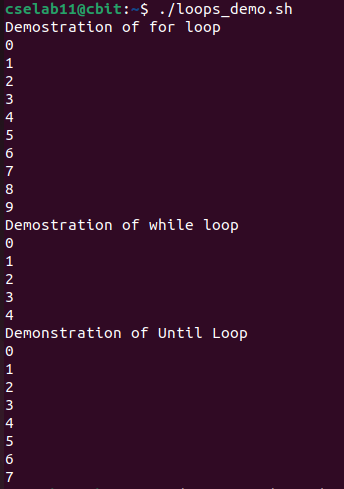
do

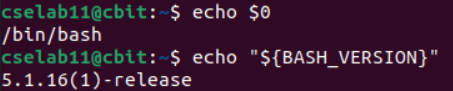
commands

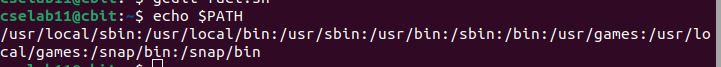
done

* **Demo program for all the loops:**



 **Output:**

* To display Shell: echo $0
* To find version of the shell: echo “${BASH\_VERSION}”
* To find path: echo $PATH



**PROGRAMS:**

**Program-1: Write the Shell script to find Factorial of a number.**

**AIM:** To find the factorial of a number.

**DESCRIPTION:** In this program, we try to read a number from the user and find factorial for that number

**ALGORITHM:**Step-1: Start

Step-2: Read the number

Step-3: a->1

Step-4: product->1

Step-5: While a<=n

product->product\*a

a->a+1

Step-6: Print product

Step-7: End

**PROGRAM:**

#!/bin/sh

echo "Enter a number: "

read n

a=1

product=1

while [ $a -le $n ]

do

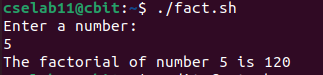
product=$(( $product \* $a ))

a=$(( $a + 1 ))

done

echo "The factorial of number $n is $product"

**OUTPUT:**



**CONCLUSION:**

By executing the above shell script, we have successfully found factorial for the number given by user.

**Program-2: Write the Shell Script to check if the given number is even or odd.**

**AIM:** To check if the given number is even or odd.

**DESCRIPTION:** In this program, we try to read a number from the user and check whether the given number is even or odd.

**ALGORITHM:**

Step-1: Start

Step-2: Read a number

Step-3: rem->n%2

Step-4: if rem=0

Print(“Even”)

Step-5: else

Print(“Odd”)

Step-6: End

**PROGRAM:**

#!/bin/sh

echo "Enter a number: "

read n

rem=$(( $n % 2 ))

if [ $rem -eq 0 ]

then

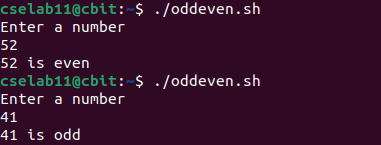
echo "$n is even"

else

echo "$n is odd"

fi

**OUTPUT:**



**CONCLUSION:**

By executing the above shell script, we have successfully checked whether the given number is even or odd.

**Program-3: Write a shell script to find power of a number.**

**AIM:** To find power of a number.

**DESCRIPTION:** In this program, we try to read the base and power of a exponent and find the value of that exponent.

**ALGORITHM:**

Step-1: Start

Step-2: Read base and power as a, b

Step-3: i->1

Step-4: res->1

Step-5: while i<=b

res->res\*a

i->i+1

Step-6: Print res

Step-7: End

**PROGRAM:**

#!/bin/sh

echo "Enter a number for base: "

read a

echo "Enter a number for power: "

read b

i=1

res=1

while [ $i -le $b ]

do

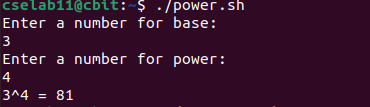
res=$(( $res \* $a ))

i=$(( $i + 1 ))

done

echo "$a^$b = $res"

**OUTPUT:**



**CONCLUSION:**

By executing the above shell script, we have successfully found power of a number.

**Program-4: Write a Shell script to check whether the given number is palindrome or not.**

**AIM:** To check whether the given number is palindrome or not.

**DESCRIPTION**: In this program, we try to read the number from the user and check whether the given number is palindrome or not.

**ALGORITHM:**Step-1: Start

Step-2: Read a number n

Step-3: temp->n

Step-4: re->0

Step-5: while n!=0

r=n%10

re=re\*10+r

n=n/10

Step-6: if re=temp

Print(“Palindrome”)

Step-7: else

Print(“Not a Palindrome”)

Step-8: End

**PROGRAM:**

#!/bin/sh

echo "Enter a number: "

read n

temp=$n

re=0

while [ $n != 0 ]

do

r=$(( $n % 10 ))

re=$(( $re \* 10 + $r ))

n=$(( $n / 10 ))

done

if [ $re -eq $temp ]

then

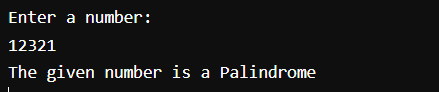
echo "The given number is a Palindrome"

else

echo "The given number is not a Palindrome"

fi

**OUTPUT:**



**CONCLUSION:**

By executing the above shell script, we have successfully checked whether the given number is palindrome or not

**Program-5:** **Write a Shell Script to print the Fibonacci Series.**

**AIM:** To print the Fibonacci Series.

**DESCRIPTION:** In this program, we try to print the Fibonacci Series

**ALGORITHM:**

Step-1: Start

Step-2: Read a number n

Step-3: a->0

Step-4: b->1

Step-5: Print a, b

Step-6: i->3

Step-7: while i<=n

c=a+b

print c

a->b

b->c

i=i+1

Step-8: End

**PROGRAM:**

#!/bin/sh

echo "Enter a number: "

read n

a=0

b=1

echo "Fibonacci Series: "

echo $a

echo $b

i=3

while [ $i -le $n ]

do

c=$(( $a + $b ))

echo $c

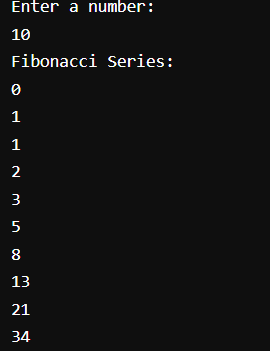
a=$b

b=$c

i=$(( $i + 1 ))

done

**OUTPUT:**



**CONCLUSION:**

By executing the above shell script, we have successfully printed the Fibonacci Series

**Program-6: Write a Shell Script to find whether a given number is prime or not.**

**AIM:** To find whether a given number is prime or not

**DESCRIPTION:** In this program, we try to read a number from the user and check whether the given number is prime or not

**ALGORITHM:**  
Step-1: Start

Step-2: Read a number n

Step-3: flag->0

Step-4: if n<=1

flag->1

Step-5: else

i->2

a->n/2

while i<=a

rem=n%i

if rem=0

flag=1

i=i+1

Step-6: if flag==0

Print “Prime”

Step-7: else  
 Print “not a prime”

Step-8: End

**PROGRAM:**

#!/bin/sh

echo "Enter a number: "

read n

flag=0

if [ $n -le 1 ]

then

flag=1

else

i=2

a=$(( $n / 2 ))

while [ $i -le $a ]

do

rem=$(( $n % $i ))

if [ $rem -eq 0 ]

then

flag=1;

break;

fi

i=$(( $i + 1 ))

done

fi

if [ $flag -eq 0 ]

then

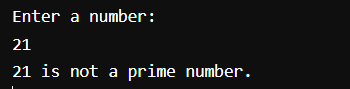
echo "$n is a prime number"

else

echo "$n is not a prime number."

fi

**OUTPUT:**



**CONCLUSION:**

By executing the above shell script, we have successfully found whether the number given by the user is prime or not

**Program-7: Write a shell script to check if the given number is an Armstrong number or not.**

**AIM:** To find whether given number is an Armstrong number or not.

**DESCRIPTION:** In this program, we try to read a number from the user and check whether the number is an Armstrong number or not.

**ALGORITHM:**  
Step-1: Start

Step-2: Read a number n

Step-3: res->0; num->0; temp->n

Step-4: while temp!=0

num->num+1

temp->temp/10

Step-5: temp->n

Step-6: while temp!=0

r->temp%10

re=1

i=1

while i<=num

re->re\*r

i->i+1

res->res+re

temp->temp/10

Step-7: if res=n

Print “Armstrong Number”

Step-8: else

Print “not a Armstrong number”

Step-9: End

**PROGRAM:**

#!/bin/sh

echo "Enter a number: "

read n

res=0

num=0

temp=$n

while [ $temp -ne 0 ]

do

num=$(( $num + 1 ))

temp=$(( $temp / 10 ))

done

temp=$n

while [ $temp -ne 0 ]

do

r=$(( $temp % 10 ))

re=1

i=1

while [ $i -le $num ]

do

re=$(( $re \* $r ))

i=$(( $i + 1 ))

done

res=$(( $res + $re ))

temp=$(( $temp / 10 ))

done

if [ $res -eq $n ]

then

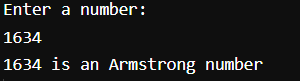
echo "$n is an Armstrong number"

else

echo "$n is not an Armstrong number"

fi

**OUTPUT:**



**CONCLUSION:**

By executing the above shell script, we have successfully checked whether the given number is Armstrong number or not.

**Program-8: Write a Shell Script to demonstrate Mini Calculator.**

**AIM:** To demonstrate Mini Calculator

**DESCRIPTION:** In this program, we try to perform all basic arithmetic operations by demonstrating a mini calculator.

**ALGORITHM:**

Step-1: Start

Step-2: Read two numbers a, b

Step-3: Print “Operations to be Performed:

1. Add
2. Sub
3. Multiply
4. Division“

Step-4: Read Choice c

Step-5: while c<=4

case c in

“1” sum=a+b; Print sum

“2” diff=a-b; Print diff

“3” mul=a\*b; Print mul

“4” div=a/b; Print div

Read c

Step-6: End

**PROGRAM:**

#!/bin/sh

echo "Enter two numbers: "

read a

read b

echo "Operations to be Performed: "

echo "1. Add"

echo "2. Sub"

echo "3. Multiply"

echo "4. Division"

echo "Enter your Choice: "

read c

while [ $c -le 4 ]

do

case "$c" in

"1") sum=$(( $a + $b ))

echo "$a + $b = $sum"

;;

"2") diff=$(( $a - $b ))

echo "$a - $b = $diff"

;;

"3") mul=$(( $a \* $b))

echo "$a \* $b = $mul"

;;

"4") div=$(( $a / $b ))

echo "$a / $b = $div"

;;

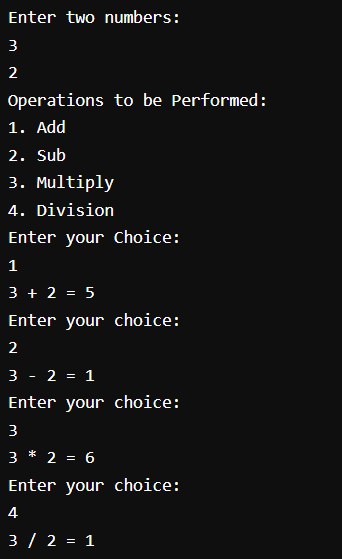
esac

echo "Enter your choice: "

read c

done

**OUTPUT:**



**CONCLUSION:**

By executing the above shell script, we have successfully demonstrated the Mini Calculator

**PROCESS RELATED SYSTEM CALLS**

**1. Go to the manual pages and write in brief about fork(), getpid(), getppid(), exec() and wait() system call. Write syntax, header files required and explanation**

1. **Fork():**

**DESCRIPTION:** fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

**HEADER FILES:**

#include <sys/types.h>

#include <unistd.h>

**SYNTAX:**

pid\_t fork(void);

**RETURN VALUE:** On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

1. **Getpid():**

**DESCRIPTION:** This command is used to get process identification. This is often used by routines that generate unique temporary filenames. This function is always successful.

**HEADER FILES:**

#include <sys/types.h>

#include <unistd.h>

**SYNTAX:**

pid\_t getpid(void);

**RETURN VALUE:** getpid() returns the process ID (PID) of the calling process.

1. **Getppid():**

**DESCRIPTION:** This command is used to get process identification. This function is always successful.

**HEADER FILES:**

#include <sys/types.h>

#include <unistd.h>

**SYNTAX:**

pid\_t getppid(void);

**RETURN VALUE:** getppid() returns the process ID of the parent of the calling process. This will be either the ID of the process that created this process using fork() or if that process has already terminated, the ID of the process to which this process has been reparented.

1. **Exec():**

**DESCRIPTION:** The exec() family of the functions are used to execute a file. They replace the current process image with a new process image. Functions are execl, execlp, execle, execv, execvp, execvpe.

**HEADER FILES:**

#include <unistd.h>

**SYNTAX:**

int execl ( const char \*pathname, const char \*arg, …… /\* (char \*) NULL \*/);

int execlp ( const char \*file, const char \*arg, …… /\* (char \*) NULL \*/);

int execle ( const char \*pathname, const char \*arg, …… /\*, (char \*) NULL, char \*const envp[] \*/);

int execv ( const char \*pathname, char \*const argv[]);

int execvp ( const char \*file, char \*const argv[]);

int execvpe ( const char \*file, char \*const argv[], char \*const envp[]);

**RETURN VALUE:** The exec() functions return only if an error has occurred. The return value is -1, and errno is set to indicate the error.

1. **Wait():**

**DESCRIPTION:** The wait() system call suspends execution of the calling thread until one of its children terminates.

**HEADER FILES:**

#include <sys/types.h>

#include <sys/wait.h>  
**SYNTAX:**

pid\_t wait(int \* wstatus);

**RETURN VALUE:** On success, return the process ID of the terminated child; on error, -1 is returned.

**2. Execute the following program and write the output and explain it.**

**AIM:** To demonstrate the fork system call

**DESCRIPTION:**

fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

**ALGORITHM:**

Step-1: Start

Step-2: a->2

Step-3: pid\_t pid

Step-4: pid->fork()

Step-5: Print pid

Step-6: if pid<0

Print “Fork Failed”

Step-7: else if pid=0

Print “Child Process”

Print a

Print ++a

Step-8: else

Print “Parent Process”

Print a

Print --a

Step-9: Print a

Step-10: End

**PROGRAM:**

#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

int main()

{

int a=2;

pid\_t pid;

pid=fork();

printf("%d\n",pid);

if(pid<0)

{

printf("fork failed");

}

else if(pid==0)

{

printf("child process \t a is : ");

printf("%d\n",++a);

}

else

{

printf("parent process \t a is : ");

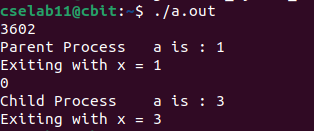
printf("%d\n",--a);

}

printf("exiting with x=%d\n",a);

}

**OUTPUT:**



**CONCLUSION:**

Here, when fork() function is called, then a child process is created with variable a = 2 and the existing process is called parent process with variable a = 2. The variables in child and parent process are independent of each other. Process ID displayed is the parent’s process id and in that section a value has been decremented so the parent process exits with a = 1. Now, the Child’s process ID is displayed and in that section a value is incremented so the child process exits with a = 3.

By executing the above program, we have successfully demonstrated the fork() System call

**3. Execute the following program and write the output. You need to execute the following program twice once without using wait() and other one using wait() system call. Explain your Results.**

**AIM:**  To demonstrate the getpid(), getppid() and wait System calls.

**DESCRIPTION:**

getpid() command is used to get process identification. This is often used by routines that generate unique temporary filenames. This function is always successful.

getppid() command is used to get process identification. This function is always successful.

The wait() system call suspends execution of the calling thread until one of its children terminates.

**ALGORITHM:**Step-1: Start

Step-2: a->2

Step-3: pid\_t pid

Step-4: pid->fork()

Step-5: Print pid

Step-6: if pid<0

Print “Error”

Step-7: else if pid=0

Print “Child Process”

Print ++a

Print getpid()

Print getppid()

Step-8: else

Print “Parent Process”

Print –a

Print getpid()

Print pid()

Step-9: Print a

Step-10: End

**PROGRAM:**

#include<stdio.h>

#include<unistd.h>

int main()

{

int a=2;

pid\_t pid;

pid=fork();

printf("%d\n",pid);

if(pid<0)

{

printf("Error");

}

else if(pid==0)

{

printf("child process");

printf("%d\n",++a);

printf("I am the child and my process id is %d\n",getpid());

printf("I am the child and my parent process id is %d\n",getppid());

}

else

{

wait();

printf("parent process");

printf("%d\n",--a);

printf("I am the parent and my process id is %d\n",getpid());

printf("I am the parent and my child process id is %d\n",pid);

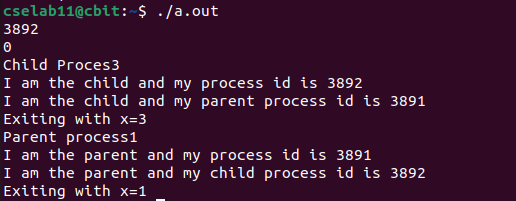
}

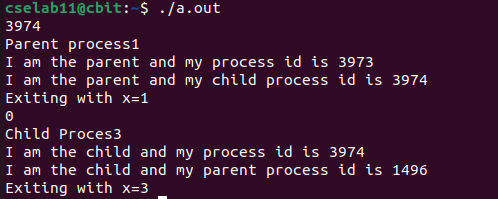
printf("exiting with x=%d\n",a);

}

**OUTPUT:**

**Using wait() system call:**



**Without using System call:**

**CONCLUSION:**

Here, in this program when wait() system call is used then the parent process waits until the child process is completed. Getpid() system call returns the process id of the that process and getppid() returns the process id of the parent process. When wait() is not used then first parent process will be completed then child process will come for the execution at that time the parent process is already completed then the child process will be taken by the another process depending upon the scheduler

**4. Write a program to demonstrate the concept of orphan process and Zombie process**

**AIM:** To demonstrate the demonstrate the concept of orphan process and Zombie process

**DESCRIPTION:**

**Orphan Process:**  
A child process that remains running even after its parent process is terminated or completed without waiting for the child process execution is called an orphan. A process becomes an orphan unintentionally. Sometime intentionally becomes orphans due to long-running time to complete the assigned task without user attention. The orphan process has controlling terminals.

**Zombie Process:**  
A Zombie is a process that has completed its task but still, it shows an entry in a process table. The zombie process usually occurred in the child process. Very short time the process is a zombie. After the process has completed all of its tasks it reports the parent process that it has about to terminate. Zombie is unable to terminate itself because it is treated as a dead process. So parent process needs to execute to terminate the command to terminate the child.

**ALGORITHM:**

**Zombie Process:**

Step-1: Start

Step-2: pid\_t child\_pid = fork()

Step-3: if child\_pid>0

sleep(10)

Print “Hello”

Step-4: else

exit(0)

Step-5: End

**Orphan Process:**

Step-1: Start

Step-2: pid = fork()

Step-3: if pid=0

Print “Child Process”

Print getpid()

Print getppid()

sleep(5)

Print getpid()

Print getppid()

Step-4: else

sleep(10)

Print getpid()

Print getppid()

Print “Parent Terminates”

Step-5: End

**PROGRAM:**

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

pid\_t child\_pid = fork();

if (child\_pid>0)

{

sleep(10);

printf("Hello\n");

}

else

exit(0);

return 0;

}

**OUTPUT:**





**PROGRAM:**#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

int pid;

pid = fork();

if(pid == 0)

{

printf("I am the child, my process ID is %d\n",getpid());

printf("My parent's process ID is %d\n",getppid());

sleep(5);

printf("\nAfter sleep\nI am the child, my process ID is %d\n",getpid());

printf("My parent's process ID is %d\n",getppid());

exit(0);

}

else

{

sleep(10);

printf("I am the parent, my process ID is %d\n",getpid());

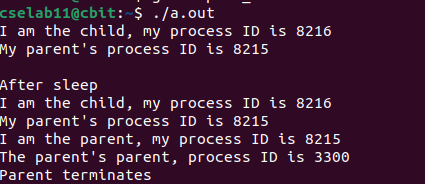
printf("The parent's parent, process ID is %d\n",getppid());

printf("Parent terminates\n");

}

return 0;

**OUTPUT:**



**CONCLUSION:**By executing the above program, we have successfully demonstrated the orphan and zombie process

**5.** **Write a C program to create a child process and allow the parent to display “parent” and the child to display “child” on the screen.?**

**AIM:** To create a child process and allow the parent to display “parent” and the child to display “child” on the screen

**DESCRIPTION:**

Here in this program, we try to create a child process and allow the parent to display “parent” and the child to display “child” on the screen

**ALGORITHM:**Step-1: Start

Step-2: pid = fork()

Step-3: if pid>0

Print “Parent”

Print getpid()

Step-4: else if pid=0

Print “Child”

Print getpid()

Step-5: else

Print “Error”

Step-6: End

**PROGRAM:**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

int pid = fork();

if (pid > 0)

{

printf("I am Parent Process\n");

printf("ID: %d\n", getpid());

}

else if (pid == 0)

{

printf("I am Child Process\n");

printf("ID: %d\n", getpid());

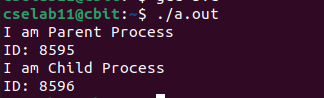
}

else

printf("Failed to create Child Process\n");

return 0;

}

**OUTPUT:**

**CONCLUSION:**

By executing the above program, we have successfully created a child process and allowed child to print “child” and parent to print “parent”

**EXEC() SYSTEM CALL**

**1. Goto the manual pages and write in brief about exec () system call.**

**Write syntax, header files required and explanation**

**DESCRIPTION:** The exec() system call is used to make the processes. When the exec() function is used, the currently running process is terminated and replaced with the newly formed process. In other words, only the new process persists after calling exec(). The parent process is shut down. This system call also substitutes the parent process's text segment, address space, and data segment with the child process.

**HEADER FILES:**

#include <unistd.h>

**SYNTAX:**

extern char \*\*environ;

    int execl(const char \*pathname, const char \*arg, ...

                    /\* (char  \*) NULL \*/);

    int execlp(const char \*file, const char \*arg, ...

                    /\* (char  \*) NULL \*/);

    int execle(const char \*pathname, const char \*arg, ...

                    /\*, (char \*) NULL, char \*const envp[] \*/);

    int execv(const char \*pathname, char \*const argv[]);

    int execvp(const char \*file, char \*const argv[]);

    int execvpe(const char \*file, char \*const argv[],

                    char \*const envp[]);

**2. Write the output of the following program**

**AIM:** To demonstrate exec system call

**DESCRIPTION:**

The exec() system call is used to make the processes. When the exec() function is used, the currently running process is terminated and replaced with the newly formed process. In other words, only the new process persists after calling exec(). The parent process is shut down. This system call also substitutes the parent process's text segment, address space, and data segment with the child process.

**ALGORITHM:**

Step-1: Start

Step-2: pid=fork()

Step-3: if pid<0

Print “Fork Failed”

Step-4: else if pid=0

execl(“/bin/ls”, “ls”, NULL);

exit(0)

Step-5: else

wait(NULL)

Print “Child process complete”

Step-6: End

**PROGRAM:**

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<string.h>

#include<errno.h>

int main(int argc,char \*argv[])

{

int pid,childpid,status;

pid=fork();

if(pid<0)

{

printf("fork failed");

}

else if(pid==0)

{

execl("/bin/ls","ls",NULL);

exit(0);

}

else

{

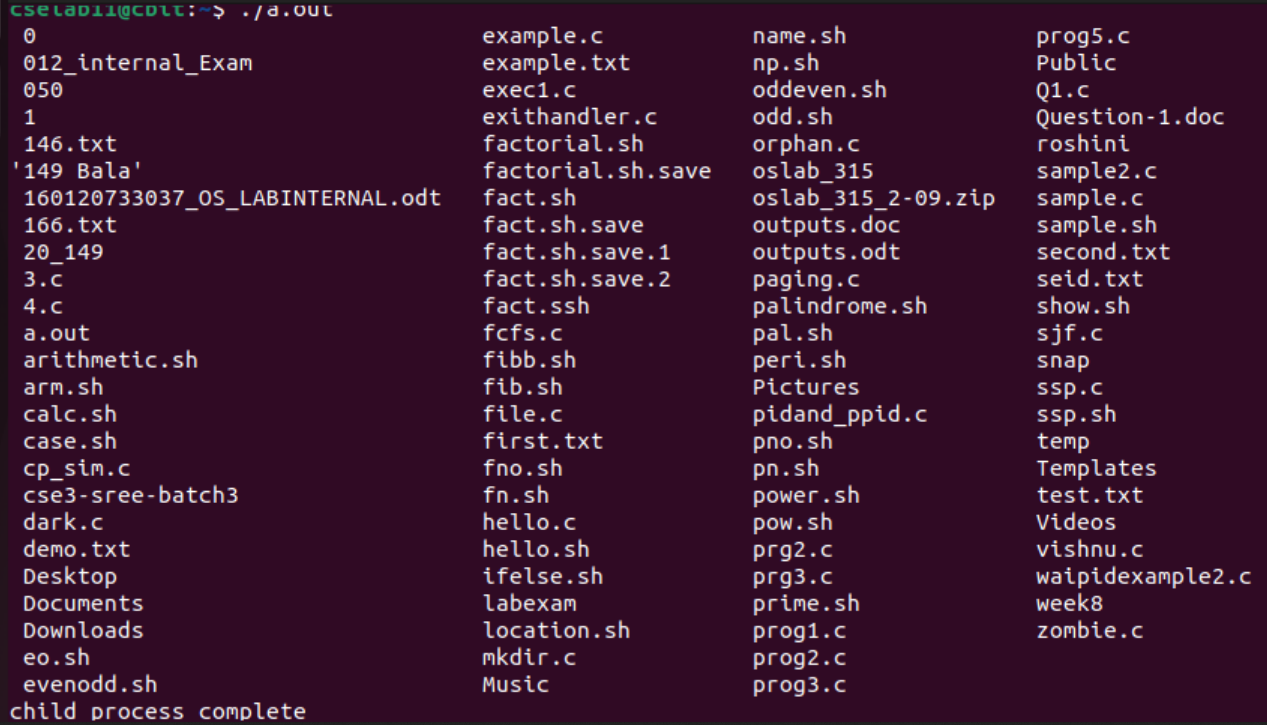
wait(NULL);

printf("child process complete\n");

}

return 0;

}

**OUTPUT:**

**CONCLUSION:**

By executing the above program, we have successfully demonstrated the exec system call

**3. Write a program using the fork() system call that computes the factorial of a given integer in the child process. The integer whose factorial is to be computed will be provided in the command line. For example, if 5 is provided, the child will compute 5! and output 120. Because the parent and child processes have their own copies of the data it will be necessary for the child to output the factorial. Have the parent invoke the wait () call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a non-negative number is passed on the command line.**

**AIM:** To create a child process and allow the child process to calculate factorial of a number

**DESCRIPTION:**Here in this program, we have to create a child process and we have to allow the child process to calculate factorial of a number and parent have to invoke the wait() call to wait for the child process to complete before exiting the program

**ALGORITHM:**

Step-1: Start

Step-2: pid\_t ret

Step-3: ret = fork()

Step-4: fact=1

Step-5: for i->1;i<argc;i++

num=atoi(argv[1])

Step-6: if ret=0 and num>0

for i->1;i<=num;i++

fact->fact\*1

Print getpid()

Print fact

Step-7: else

if num>0

print “Invalid input”

wait(NULL)

Print getpid()

Step-8: End

**PROGRAM:**  
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main (int argc, char \* argv[])

{

pid\_t ret;

int i, num,sum=0,fact=1;

ret=fork();

for(i=1;i<argc; i++){

     num=atoi(argv[1]);

}

if (ret == 0 && num > 0) {

   for(i=1;i<=num; i++){

     fact\*=i;

}printf("\n [ ID = %d] Factorial of %d is %d\n",getpid(),num,fact);

}

else{

if(num < 0)

printf("Invalid input");

wait(NULL);

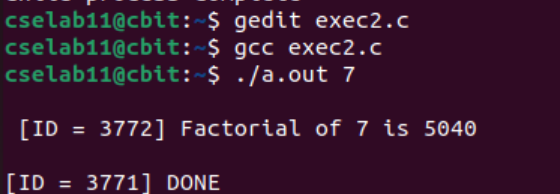
printf("\n[ID = %d] DONE\n",getpid());

}

return 0;

}

**OUTPUT:**



**CONCLUSION:**

By executing the above program, we have successfully created a child process and allowed the child factorial to calculate the factorial of a number.

**4. Write a program that creates a child process and the child process checks whether a given number is palindrome or not. Make use of exec() system call for finding palindrome.**

**AIM:** To create a child process and the child process checks whether a given number is palindrome or not

**DESCRIPTION:**Here in this program, we have to create a child process and the child process checks whether a given number is palindrome or not. Here exec() system call should be used for finding the palindrome

**ALGORITHM:**Step-1: Start

Create a function palindrome()

Step-2: Read a number n

Step-3: temp->n

Step-4: sum->0

Step-5: while n>0

r->n%10

sum->sum\*10+r

n->n/10

Step-6: if temp=sum

Print “Palindrome”

Step-7: else

Print “Not a Palindrome”

Main function:

Step-8: pid->fork()

Step-9: if pid=0

Print “child”

palindrome()

Step-10: else

wait()

Print “Parent”

Step-11: Create a new file with name exec.c

Step-12: Print pid of exec.c file

Step-13: execv (“./pal”, args)

Step-14: End

**PROGRAM:**

**exec.c**

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

int main(int argc, char \*argv[])

{

    printf("PID of execute.c = %d\n", getpid());

    char \*args[] = {"pal", NULL};

    execv("./pal", args);

    printf("The control never comes back to this line #unreachable");

    return 0;

}

**Pal.c**

#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>

void palindrome()

{

int n,r,sum=0,temp;

printf("Checking palindrome");

printf("\nEnter the number: ");

scanf("%d",&n);

temp=n;

while(n>0)

{

r=n%10;

sum=(sum\*10)+r;

n=n/10;

}

if (temp==sum)

printf("Palindrome number ");

else

printf("Not palindrome");

}

int main()

{

int pid;

pid = fork();

if(pid==0){

printf("I am the child process");

palindrome();

}

else{

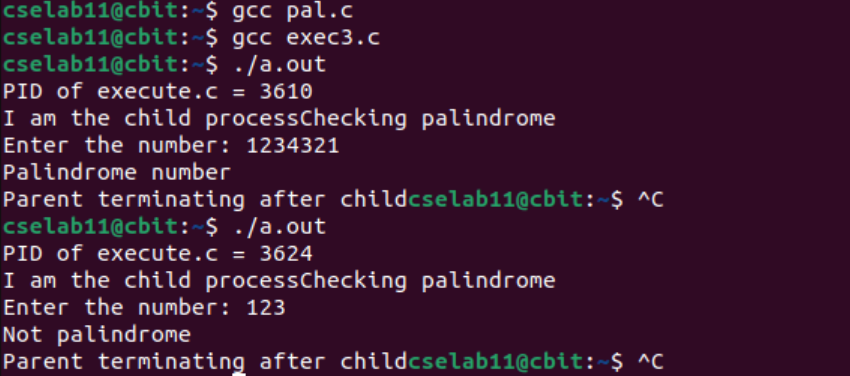
wait(NULL);

printf("\nParent terminating after child");

}

}

**OUTPUT:**



**CONCLUSION:**

By executing the above program, we have successfully created the child process and allowed the child process to check whether the given number is a palindrome or not

**DEMONSTRATION OF LINUX/UNIX PROCESS RELATED SYSTEM CALLS**

**AIM:** To demonstrate the Linux/Unix Process Related System Calls – getuid, setuid, nice

**DESCRIPTION:**

**1. Getuid:**

**DESCRIPTION:** This system call is used to get user identity

**HEADER FILES:**

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

**SYNTAX:**

uid\_t getuid(void)

**RETURN VALUE:**

getuid() returns the real user ID of the current process

**2. Setuid:**

**DESCRIPTION:** This system call is used to set user identity. It sets the effective user ID of the calling process

**HEADER FILES:**

#include <unistd.h>

**SYNTAX:**

int setuid(uid\_t uid)  
**RETURN VALUE:**

On success, zero is returned. On error, -1 is returned, and errno is set to indicate the error

**3. Nice:**

**DESCRIPTION:** The Nice command configures the priority of a Linux process before it is started. The renice command sets the priority of an already running process.

**SYNTAX:** nice -nice\_value command-arguments; renice -n nice\_value –p pid\_of\_the\_process

**ALGORITHM:  
getuid():**

Step-1: Start

Step-2: Print “The Real user ID is “

Step-3: Print getuid()

Step-4: Print “The Effective user ID is”

Step-5: Print setuid()

Step-6: End

**Setuid():**

Step-1: Start

Step-2: Print getuid(), geteuid()

Step-3: if (setuid(25) != 0)

perror “setuid() error”

Step-4: else

Print getuid(), geteuid()

Step-5: End

**Nice():**

Step-1: Start

Step-2: Define a function my\_nice(int incr):

prio -> getpriority(PRIO\_PROCESS, 0)

if (setpriority(PRIO\_PROCESS, 0, prio+incr)=-1)

return -1

prio -> getpriority(PRIO\_PROCESS, 0)

return prio

Step-3: prio -> getpriority(PRIO\_PROCESS, 0)

Step-4: Print prio

Step-5: my\_nice(5)

Step-6: prio -> getpriority(PRIO\_PROCESS, 0)

Step-7: Print prio

Step-8: my\_nice(-7)

Step-9: prio -> getpriority(PRIO\_PROCESS, 0)

Step-10: Print prio

Step-11: End

**PROGRAM:**

**GETUID():**

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

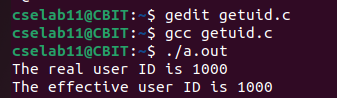
int main(void){

     printf("The real user ID is %d\n", getuid());

     printf("The effective user ID is %d\n", geteuid());

return 0;

}

**OUTPUT:**

**SETUID():**

#define \_POSIX\_SOURCE

#include <sys/types.h>

#include <stdio.h>

#include <unistd.h>

void main() {

  printf("prior to setuid(), uid=%d, effective uid=%d\n",

      (int) getuid(), (int) geteuid());

  if (setuid(25) != 0)

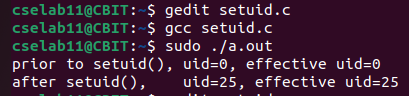
perror("setuid() error");

  else

printf("after setuid(), uid=%d, effective uid=%d\n",

        (int) getuid(), (int) geteuid());

}

**OUTPUT:**

**NICE():**

#include <stdio.h>

#include <sys/resource.h>

int my\_nice(int incr)

{

    int prio = getpriority(PRIO\_PROCESS, 0);

    if (setpriority(PRIO\_PROCESS, 0, prio + incr) == -1)

    return -1;

    prio = getpriority(PRIO\_PROCESS, 0);

    return prio;

}

int main(void)

{

    int prio = getpriority(PRIO\_PROCESS, 0);

    printf("Current priority = %d\n", prio);

    printf("\nAdding +5 to the priority\n");

    my\_nice(5);

    prio = getpriority(PRIO\_PROCESS, 0);

    printf("Current priority = %d\n", prio);

    printf("\nAdding -7 to the priority\n");

    my\_nice(-7);

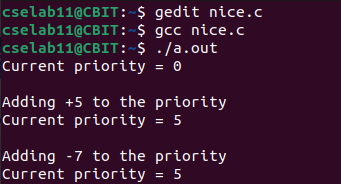
    prio = getpriority(PRIO\_PROCESS, 0);

    printf("Current priority = %d\n", prio);

    return 0;

}

**OUTPUT:**



#include <stdio.h>

#include <unistd.h>

#include <sys/resource.h>

#include <sys/time.h>

int main()

{

    int incr;

    printf("Current priority of the process %d\n",getpriority(PRIO\_PROCESS, 0));

    printf("Enter incr value\n");

    scanf("%d", &incr);

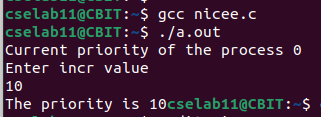
    int ret;

    ret = nice(incr);

    printf("The priority is %d", ret);

}

**OUTPUT:**



**CONCLUSION:**

By executing the above program, we have successfully demonstrated the process related system calls – getuid(), setuid(), nice()

**FILE RELATED SYSTEM CALLS**

**1. Go to manual pages and write in brief about creat(), open(), read(), write(), close(), lseek(), stat() system call. Write syntax, header files required and explanation**

1. **Creat():**

**DESCRIPTION:** This command is used to create a file

**HEADER FILES:**

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>  
**SYNTAX:**

int creat(const char \*path, mode\_t mod);  
**RETURN VALUE:** The function returns the file descriptor or in case of an error -1

1. **Open():**

**DESCRIPTION:** This command is used to open or create a file

**HEADER FILES:**

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>  
**SYNTAX:**

int open(const char \*path, int flags,... /\* mode\_t mod \*/);  
**RETURN VALUE:** This function returns the file descriptor or in case of an error -1

1. **Read():**

**DESCRIPTION:** This command is used to read from file descriptor

**HEADER FILES:**#include <unistd.h>  
**SYNTAX:**

ssize\_t read(int fd, void\* buf, size\_t noct);  
**RETURN VALUE:** The function returns the number of bytes read, 0 for end of file (EOF) and -1 in case an error occurred.

1. **Write():**

**DESCRIPTION:** This command is used to send a message to another user.

**HEADER FILES:**  
#include <unistd.h>  
**SYNTAX:**

ssize\_t write(int fd, const void\* buf, size\_t noct);  
**RETURN VALUE:** The function returns the number of bytes written and the value -1 in case of an error.

1. **Close():**

**DESCRIPTION:** This command is used to close a file descriptor

**HEADER FILES:**

#include <unistd.h>  
**SYNTAX:**

int close(int fd);  
**RETURN VALUE:** The function returns 0 in case of success and -1 in case of an error. At the termination of a process an open file is closed anyway.

1. **Lseek():**

**DESCRIPTION:** This command is used for reposition read/write file offset

**HEADER FILES:**

#include <sys/types.h>

#include <unistd.h>  
**SYNTAX:**

off\_t lseek(int fd, off\_t offset, int ref);  
**RETURN VALUE:** The function returns the displacement of the new current position from the beginning of the file or -1 in case of an error.

1. **Stat():**

**DESCRIPTION:** This command is to display file or file system status.

**HEADER FILES:**

#include <sys/types.h>

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

int stat(const char\* path, struct stat\* buf);  
**RETURN VALUE:** The function returns 0 in case of success and -1 in case of an error.

**2. Write a program to create a file.**

**AIM:** To demonstrate create system call

**DESCRIPTION:**

This command is used to create a file. Here in this program, we try to create a file using creat system call

**ALGORITHM:**

Step-1: Start

Step-2: fd -> creat(“first.txt”,S\_IREAD|S\_IWRITE)

Step-3: fd1 -> creat(“second.txt”,S\_IREAD|S\_IWRITE)

Step-4: Print fd

Step-5: Print fd1

Step-6: if fd=-1

Print “Error”

Step-7: else

Print “Success”

Step-8: close(fd)

Step-9: close(fd1)

Step-10: End

**PROGRAM:**

#include<stdio.h> /\*header file for main function\*/

#include<sys/types.h>

#include<sys/stat.h> /\*header files for creat() system call\*/

#include<fcntl.h>

int main()

{

int fd; /\*creating 2 file descriptors\*/

int fd1;

fd=creat("first.txt",S\_IREAD|S\_IWRITE); /\*creating 2 files which \*/

fd1=creat("second.txt",S\_IREAD|S\_IWRITE); //returns file descriptors

printf("%d\n",fd);

printf("%d\n",fd1);

if(fd==-1) /\*checking whether file descriptor is negative or not\*/

printf("ERROR\n");

else

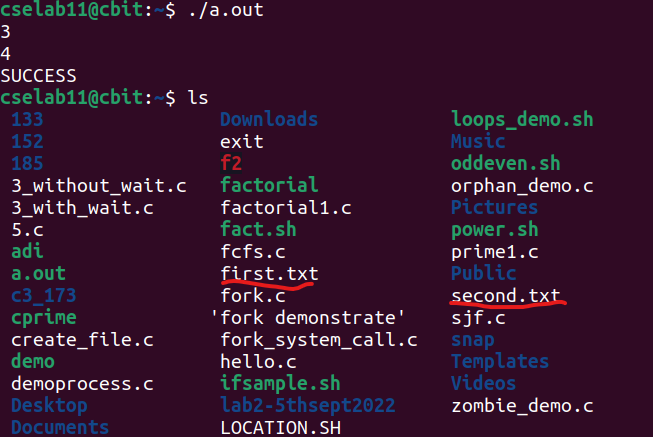
printf("SUCCESS\n");

close(fd); /\*closing the file descriptors\*/

close(fd1);

}

**OUTPUT:**



**CONCLUSION:**

By executing the above program, we have successfully created the files using Creat system call.

**3. Program to write contents from file to console**

**AIM:** To demonstrate write system call

**DESCRIPTION:**

This command is used to send a message to another user. Here in this program, we try to write contents from file to console

**ALGORITHM:**

Step-1: Start

Step-2: fd -> open(argv[1],O\_RDONLY)

Step-3: if ffd=-1

exit(-1)

Step-4: while (n\_char=read(fd, buffer, 1))!=0

write(1,buffer,n\_char)

Step-5: End

**PROGRAM:**  
#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

int main(int argc,char \*argv[])

{

int fd;

int n\_char=0;

char buffer[1];

fd=open(argv[1],O\_RDONLY);

if(fd==-1)

{

exit(-1);

}

while((n\_char=read(fd,buffer,1))!=0)

{

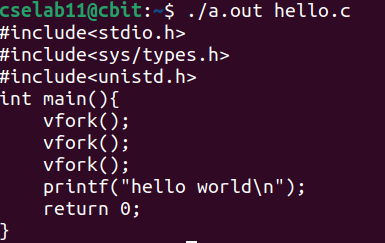
write(1,buffer,n\_char);

}

return 0;

}

**OUTPUT:**



**CONCLUSION:**

By executing the above program, we have successfully copied the contents from file to console

**4. Program to read from one file and write to another file**

**AIM:** To copy the contents of one file to another file

**DESCRIPTION:**

Here in program, we try to read the contents from one file and write that content into another file

**ALGORITHM:**

Step-1: Start

Step-2: fd1->open(“first.txt”,O\_RDONLY)

Step-3: Print fd1

Step-4: fd2->creat(“second.txt”,S\_IREAD|S\_IWRITE)

Step-5: Print fd2

Step-6: if fd1<0 or fd2<0

Print “Error”

exit(1)

Step-7: while read(fd1,ch,1)>0

write(fd2,ch,1)

Print ch

Step-8: close fd1

Step-9: close fd2

Step-10: End

**PROGRAM:**

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

int main()

{

int fd1,fd2;

char ch[1];

fd1=open("first.txt",O\_RDONLY);

printf("%d\n",fd1);

fd2=creat("second.txt",S\_IREAD|S\_IWRITE);

printf("%d\n",fd2);

if(fd1<0||fd2<0)

{

printf("Error");

exit(-1);

}

while((read(fd1,ch,1))>0)

{

write(fd2,ch,1);

printf("%c",ch[0]);

}

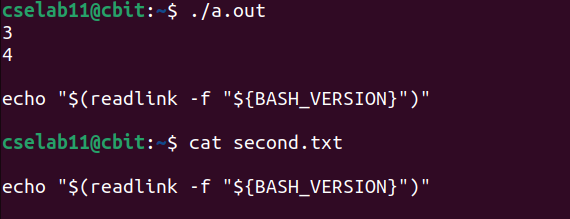
close(fd1);

close(fd2);

return 0;

}

**OUTPUT:**



**CONCLUSION:**

By executing the above program, we have successfully copied the contents of file to another file

**5. Program to show the working of the lseek function**

**AIM:** To demonstrate the working of the lseek function

**DESCRIPTION:**

This command is used for reposition read/write file offset. Here in this program, we try to demonstrate the lseek system call

**ALGORITHM:**

Step-1: Start

Step-2: fd1->open(“1.txt”,O\_RDWR)

Step-3: buffer[0]->’1’

Step-4: do

Read buffer[0]

if buffer[0]!=’#’

write(fd1,buffer,1)

while (buffer[0]!=’#’)

Step-5: close(fd1)

Step-6: fd2->open(“1.txt”,O\_RDWR)

Step-7: lseek(fd2,2\*sizeof(char),0)

Step-8: do

k=read(fd2,&buffer[0],1)

if k!=0

Print buffer[0]

while k!=0

Step-9: End

**PROGRAM:**

#include<unistd.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include<stdio.h>

#include<stdlib.h>

int main()

{

int fd1=open("1.txt",O\_RDWR);

printf("%d",fd1);

char buffer[1];

buffer[0]='1';

printf("enter the data : (press # to exit)");

do

{

scanf("%c",&buffer[0]);

if(buffer[0]!='#')

write(fd1,buffer,1);

}while(buffer[0]!='#');

close(fd1);

int fd2=open("1.txt",O\_RDWR);

lseek(fd2,2\*sizeof(char),0);int k;

do

{

k=read(fd2,&buffer[0],1);

if(k!=0)

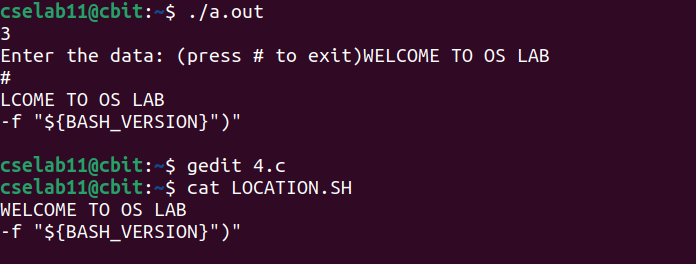
printf("%c",buffer[0]);

}while(k!=0);

return 0;

}

**OUTPUT:**



**CONCLUSION:**

By executing the above program, we have successfully demonstrated the lseek function

**Task-1: Write a C program using file related system call to take your college name as input from the user, write it to the file. Again, read from the file and display on the screen**

**AIM:** To write the user given input in to a file and read the contents of that file and display on the screen.

**DESCRIPTION:** In this program, we try to use write system call and read system calls.

**ALGORITHM:**

Step-1: Start

Step-2: Read the data to be entered into a file

Step-3: Print ch

Step-4: res->open(“create.txt”,O\_RDWR)

Step-5: write(res,ch,strlen(ch))  
Step-6: close(res)

Step-7: res->open(“create.txt”,O\_RDONLY,0)

Step-8: read(res,&ch,12)

Step-9: Print ch

Step-10: End

**PROGRAM:**

#include<stdio.h>

#include<string.h>

#include<fcntl.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<unistd.h>

void main()

{

int res;

char ch[12];

printf("To write the content entered by the user in to a file: \n\n");

printf("Enter data you want to enter into file: ");

scanf("%s",ch);

printf("%s\n",ch);

res=open("create.txt",O\_RDWR);

write(res,ch,strlen(ch));

close(res);

printf("Successfully updated the contents into the file\n");

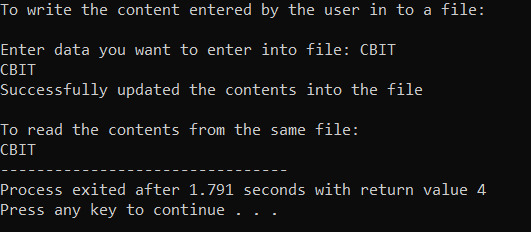
printf("\nTo read the contents from the same file: \n");

res=open("create.txt",O\_RDONLY,0);

read(res,&ch,12);

printf("%s",ch);

}

**OUTPUT:**

**CONCLUSION:**

By executing the above program, we have successfully taken the input from user and written in to the file and again read from the file and displayed it on the screen.

**Task-2: Program for simulation of cp command using file related system calls.**

**AIM:** To simulate cp command using file related system calls

**DESCRIPTION:** The command cp is used to copy a file into another file. In this program, we try to simulate the working of the cp command

**ALGORITHM:**

Step-1: Start

Step-2: Define a createf function

Step-3: for i->0;i<2;i++

Read n

fptr->fopen(n,’w’)

if fptr=NULL

Print “Unable to create file”

Read the data to be entered in to the file

Write the data into the file

fclose(fptr)

Step-4: Define a copyfun function

Step-5: Read the name of the source file

Step-6: source->fopen(source\_file,”r”)

Step-7: if source=NULL

Print “Press any key to exit”

exit(EXIT\_FAILURE)

Step-8: Read the name of the target file

Step-9: target->fopen(target\_file,”w”)

Step-10: if target=NULL

fclose(source)

print “Press any key to exit”

exit(EXIT\_FAILURE)

Step-11: while (ch->fgetc(source))!=EOF

fputc(ch,target)

Step-12: fclose(source)  
Step-13: fclose(target)

Step-14: End

**PROGRAM:**

#include<stdio.h>

#include<stdlib.h>

#include<dirent.h>

#define DATA\_SIZE 1000

void createf()

{

char data[DATA\_SIZE];

char n[100];

FILE \* fPtr;

int i;

printf("Create 2 files \nfile1: with data \nfile2: without data for copying\n");

for ( i=0;i<2;i++)

{

printf("Enter a file name:");

gets(n);

fPtr = fopen(n,"w");

if(fPtr == NULL)

{

printf("Unable to create file.\n");

exit(EXIT\_FAILURE);

}

printf("Enter contents to store in file: ");

fgets(data, DATA\_SIZE, stdin);

fputs(data, fPtr);

fclose(fPtr);

printf("File created and saved successfully.\n");

}

}

void copyfun()

{

char ch, source\_file[20], target\_file[20];

FILE \*source, \*target;

printf("Enter name of file to copy\n");

gets(source\_file);

source = fopen(source\_file, "r");

if (source == NULL)

{

printf("Press any key to exit...\n");

exit(EXIT\_FAILURE);

}

printf("Enter name of target file\n");

gets(target\_file);

target = fopen(target\_file, "w");

if (target == NULL)

{

fclose(source);

printf("Press any key to exit...\n");

exit(EXIT\_FAILURE);

}

while ((ch = fgetc(source)) != EOF)

fputc(ch, target);

printf("File copied successfully.\n");

fclose(source);

fclose(target);

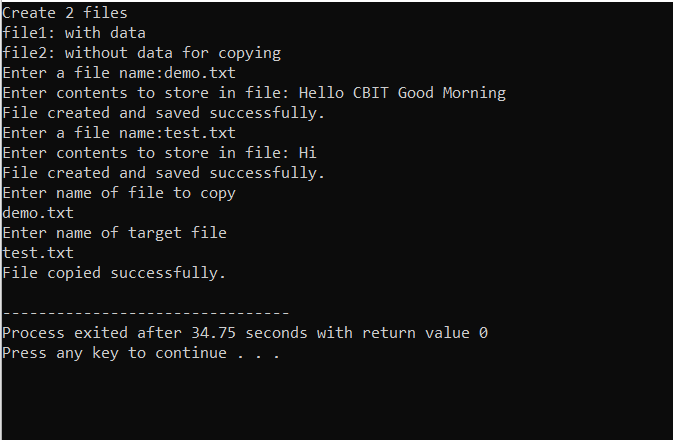
}

int main(){

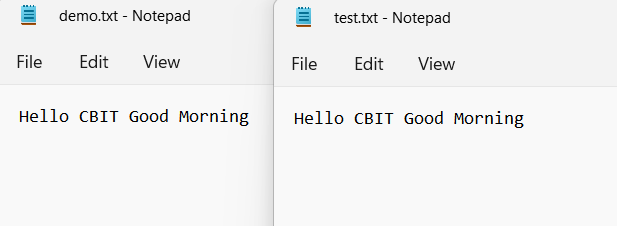
createf();

copyfun();

}

**OUTPUT:**

**Contents of demo.txt file and test.txt file after executing the above program:**



**CONCLUSION:**

By executing the above program, we have successfully simulated the cp command using file related system calls

**Task-3: Program for simulation of grep command using file related system calls.**

**AIM:** To simulate grep command using file related system calls.

**DESCRIPTION:** The command grep is used to print the lines matching a pattern. In this program, we try to simulate the grep command.

**ALGORITHM:**

Step-1: Start

Step-2: Read the file name

Step-3: Read the pattern to be searched

Step-4: fp->fopen(fn,”r”)

Step-5: while !feof(fp)

fgets(temp,1000,fp)

if strstr(temp,pat)

Print temp

Step-6: fclose(fp)

Step-7: End

**PROGRAM:**

#include<stdio.h>

#include<string.h>

void main()

{

char fn[10],pat[10],temp[200];

FILE \*fp;

printf("Enter file name: ");

scanf("%s",fn);

printf("Enter pattern to be searched: ");

scanf("%s",pat);

fp=fopen(fn,"r");

while(!feof(fp))

{

fgets(temp,1000,fp);

if(strstr(temp,pat))

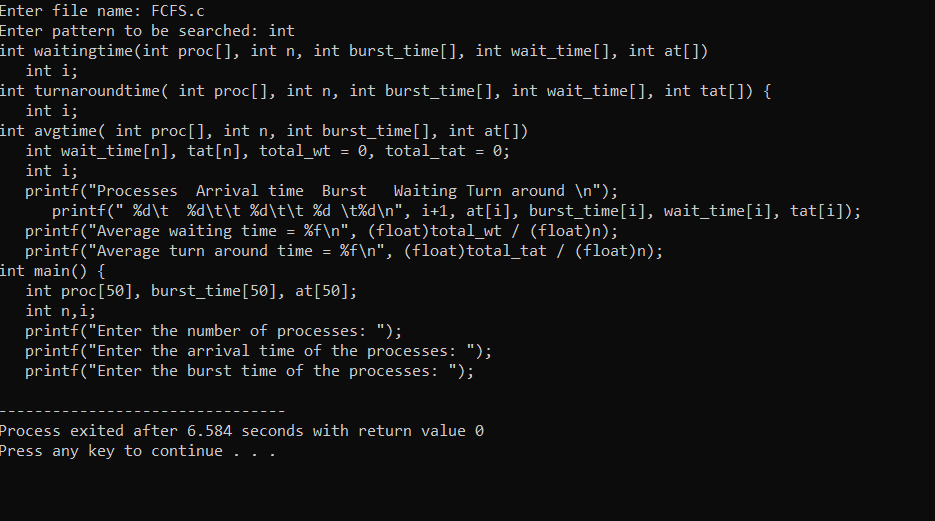
printf("%s",temp);

}

fclose(fp);

}

**OUTPUT:**



**CONCLUSION:**

By executing the above program, we have successfully simulated the grep command using file related system calls.

**STAT SYSTEM CALL**

**AIM:** To demonstrate the Stat System Call

**DESCRIPTION:**

Stat System call is a system call in Linux to check the status of a file such as to check when the file was accessed. The stat() system call actually returns file attributes. The file attributes of an inode are basically returned by Stat() function. An inode contains the metadata of the file. An inode contains: the type of the file, when the file was accessed (modified, deleted) that is timestamps, and the parh of the file, the user ID and the group ID, links of the file, and physical address of file content.

**SYNTAX:**  
int stat(const char \*path, struct stat \*buf)

**ALGORITHM:**

Step-1: Start

Step-2: struct stat sfile

Step-3: stat(“file.c”, &sfile)

Step-4: Print sfile.st\_mode

Step-5: Print sfile.st\_uid

Step-6: Print sfile.st\_size

Step-7: Print sfile.st\_gid

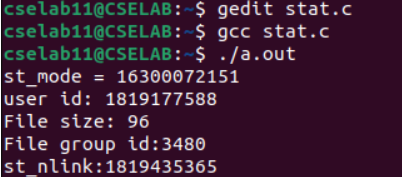
Step-8: Print sfile.st\_nlink

Step-9: End

**PROGRAM:**

#include <stdio.h>  
#include <sys/stat.h>  
int main()  
{  
 struct stat sfile;  
 stat("file.c", &sfile);  
 printf("st\_mode = %o\n", sfile.st\_mode);  
 printf("user id: %d\n", sfile.st\_uid);  
 printf("File size: %ld\n", sfile.st\_size);  
 printf("File group id:%d\n", sfile.st\_gid);  
 printf("st\_nlink:%u\n",(unsigned int)sfile.st\_nlink);  
 return 0;  
}

**OUTPUT:**



**CONCLUSION:**

By executing the above program, we have successfully demonstrated the Stat System Call.

**DEMONSTRATION OF LINUX/UNIX FILE RELATED SYSTEM CALLS**

**AIM:** To demonstrate Linux/Unix File Related System Calls – mkdir, chmod, chown

**DESCRIPTION:**

**1. mkdir():**

**Description:** The mkdir() function shall create a new directory with name path  
**Header Files:**

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

int mkir(const char \*path, mode\_t mode);

**Return Value:**

Upon successful completion, *mkdir*() shall return 0. Otherwise, -1 shall be returned, no directory shall be created, and *errno* shall be set to indicate the error.

**2. chmod():**

**Description:** The chmod() and fchmod() system calls change a file’s mode bits. (the mode consists of the file permission bits plus the set-user-ID, set-group-ID, and sticky bits)

**Header files:**

#include <sys/types.h>

#include <sys/stat.h>

**Syntax:**

int chmod(const char \*path, mode\_t mode);

int fchmod(int fildes, mode\_t mode);

**Return Value:**

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

**3. chown():**

**Description:** The chown() function sets the owner ID and group ID of the file that pathname specifies.

**Header files:**

#include <sys/types.h>

#include <unistd.h>

**Syntax:**

int chown(const char \*pathname, uid\_t owner, gid\_t group);

int fchown(int fildes, uid\_t owner, gid\_t group);

int lchown(const char \*pathname, uid\_t owner, gid\_t group);

**Return Value:**

If successful, chown(), fchown(), and lchown() return zero. On failure, they return -1, make no changes to the owner or group of the file, and set errno

**ALGORITHM:**

**Mkdir():**  
Step-1: Start

Step-2: Read the name of the directory to be created

Step-3: status -> mkdir(dir\_name, 0755)

Step-4: if (status = 0):

Print “Directory creation successful”

Step-5: else:

Print “Directory not created”

Step-6: End

**Chmod():**

Step-1: Start

Step-2: filename -> argv[1]

Step-3: r -> stat(filename, &fs)

Step-4: if r=-1:

Print stderr

Print “Error Reading”

Print filename

Step-5: r -> chmod( filename, fs.st\_mode|S\_IWGRP+S\_IWOTH)

Step-6: if r!=0:

Print stderr

Print “Unable to reset permissions”  
 Print filename

Step-7: stat(filename, &fs)

Step-8: End

**Chown():**  
Step-1: Start

Step-2: ecode -> 0

Step-3: for (i->1;i<argc;i++)

if (chown(argv[i],1000,24)=0)

perror(argv[i])

ecode++

Step-4: exit(ecode)

Step-5: End

**PROGRAM:**

**Mkdir():**  
#include<sys/stat.h>

#include<sys/types.h>

#include<stdio.h>

void main(){

int status;

char dir\_name[10];

scanf("%s", dir\_name);

status = mkdir(dir\_name, 0755);

if(status == 0)

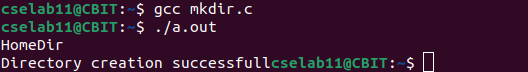
    printf("Directory creation successfull");

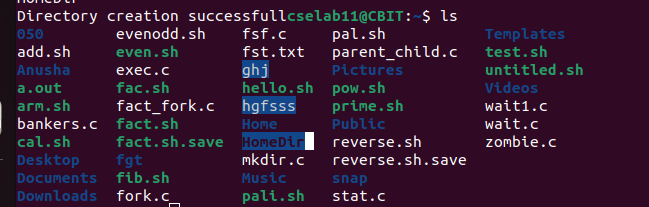
else

    printf("Directory not created");

}

**OUTPUT:**





**Chmod():**

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

int main(int argc, char \*argv[])

{

const char \*filename;

struct stat fs;

int r;

filename = argv[1];

r = stat(filename,&fs);

if( r==-1)

{

     fprintf(stderr,"Error reading '%s'\n",filename);

     exit(1);

}

r = chmod( filename, fs.st\_mode | S\_IWGRP+S\_IWOTH );

if( r!=0)

{

     fprintf(stderr,"Unable to reset permissions on '%s'\n",filename);

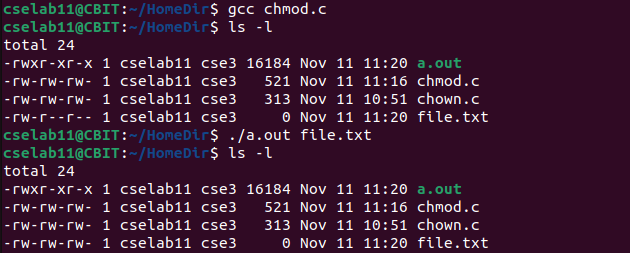
     exit(1);

}

stat(filename,&fs);

return(0);

}

**OUTPUT:**

**Chown():**

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main( int argc, char\*\* argv )

  {

int i;

int ecode = 0;

for( i = 1; i < argc; i++ )

{

   if( chown( argv[i], 1000, 24 ) == 0 )

{

     perror( argv[i] );

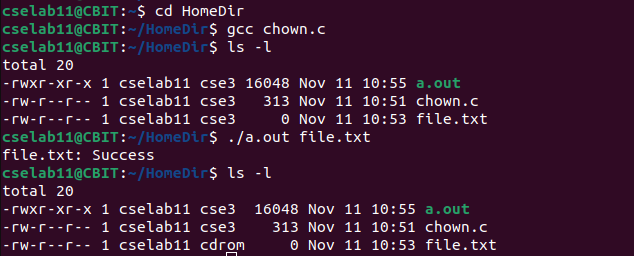
     ecode++;

   }

}

exit( ecode );

  }

**OUTPUT:**

**CONCLUSION:**

By executing the above program, we have successfully demonstrated the File Related System Calls – mkdir, chmod, chown.

**DEMONSTRATION OF CPU SCHEDULING**

**1. First Come First Serve Scheduling (FCFS)**

**AIM:** To demonstrate the First Come First Serve CPU Scheduling

**DESCRIPTION:**

First Come, First Served (FCFS) also known as First In, First Out (FIFO) is the CPU scheduling algorithm in which the CPU is allocated to the processes in the order they are queued in the ready queue. FCFS follows non-pre-emptive scheduling which mean once the CPU is allocated to a process it does not leave the CPU until the process will not get terminated or may get halted due to some I/O interrupt.

**ALGORITHM:**  
Step-1: Start

Step-2: In function int waitingtime (int proc[], int n, int burst\_time[], int wait\_time[], int at[])

Set wait\_time[0] = 0

Loop For i = 1 and i < n and i++

Set wait\_time[i] = burst\_time[i-1] + wait\_time[i-1] – (at[i] – a[i-1])

End For

Step-3: In function int turnaroundtime( int proc[], int n, int burst\_time[], int wait\_time[], int tat[])

Loop For i = 0 and i < n and i++

Set tat[i] = burst\_time[i] + wait\_time[i]

End For

Step-4: In function int avgtime( int proc[], int n, int burst\_time[], int at[])

Declare and initialize wait\_time[n], tat[n], total\_wt=0, total\_tat = 0

Call waitingtime(proc, n, burst\_time, wait\_time, tat)

Call turnaroundtime(proc, n, burst\_time, wait\_time, tat)

Loop For i=0 and i<n and i++

Set total\_wt = total\_wt + wait\_time[i]

Set total\_tat = total\_tat + tat[i]

Print process number, arrival time, burst time, wait time and turnaround time

End For

Print Average waiting time = total\_wt/n

Print Average turnaround time = total\_tat/n

Step-5: In int main()

Read the input from user n, burst\_time[], at[]

Call avgtime(proc, n, burst\_time, at)

Step-6: End

**PROGRAM:**

#include <stdio.h>

//Function to calculate waiting time

int waitingtime(int proc[], int n, int burst\_time[], int wait\_time[], int at[])

{

int i;

wait\_time[0] = 0;

for (i = 1; i < n ; i++)

wait\_time[i] = burst\_time[i-1] + wait\_time[i-1] - (at[i]-at[i-1]);

return 0;

}

// Function to calculate turn around time

int turnaroundtime( int proc[], int n, int burst\_time[], int wait\_time[], int tat[]) {

int i;

for ( i = 0; i < n ; i++)

tat[i] = burst\_time[i] + wait\_time[i];

return 0;

}

//Function to Calculation Average waiting and Turn around time

int avgtime( int proc[], int n, int burst\_time[], int at[])

{

int wait\_time[n], tat[n], total\_wt = 0, total\_tat = 0;

int i;

waitingtime(proc, n, burst\_time, wait\_time, at);

turnaroundtime(proc, n, burst\_time, wait\_time, tat);

printf("Processes Arrival time Burst Waiting Turn around \n");

for ( i=0; i<n; i++) {

total\_wt = total\_wt + wait\_time[i];

total\_tat = total\_tat + tat[i];

printf(" %d\t %d\t\t %d\t\t %d \t%d\n", i+1, at[i], burst\_time[i], wait\_time[i], tat[i]);

}

printf("Average waiting time = %f\n", (float)total\_wt / (float)n);

printf("Average turn around time = %f\n", (float)total\_tat / (float)n);

return 0;

}

// main function

int main() {

//process id's

int proc[50], burst\_time[50], at[50];

int n,i;

printf("Enter the number of processes: ");

scanf("%d", &n);

printf("Enter the arrival time of the processes: ");

for(i=0;i<n;i++)

{

scanf("%d", &at[i]);

}

printf("Enter the burst time of the processes: ");

for(i=0;i<n;i++)

{

scanf("%d", &burst\_time[i]);

}

for(i=0;i<n;i++)

{

proc[i]=i+1;

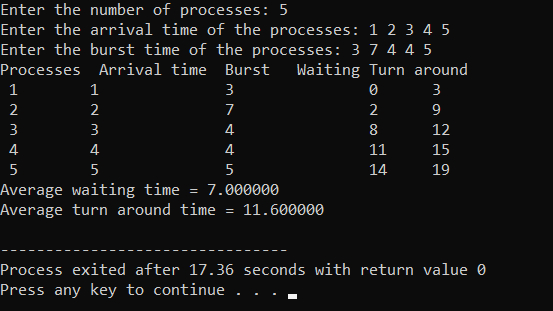
}

avgtime(proc, n, burst\_time, at);

return 0;

}

**OUTPUT:**



**CONCLUSION:**

By executing the above program, we have successfully demonstrated the FCFS CPU Scheduling

**2. Shortest Job First Scheduling:**  
**AIM:** To demonstrate Shortest Job First CPU Scheduling

**DESCRIPTION:**

SJF scheduling algorithm, schedules the process according to their burst time. In SJF scheduling, the process with the lowest burst time, among the list of available processes in the ready queue, is going to be scheduled next

**ALGORITHM:**  
Step-1: Start

Step-2: Take process, arrival time, burst time input from the user

Step-3: Sort the process according to arrival time and if the process has the same arrival time then sort them having less burst time

Step-4: Swap the process one above one in the order of execution

Step-5: Find the turnaround time (tat) and waiting time (wt)

Step-6: Find average tat and average wt

Step-7: End

**PROGRAM:**

#include <stdio.h>

int main()

{

int arrival\_time[10], burst\_time[10], temp[10];

int i, smallest, count = 0, time, limit;

int wt, tat;

double wait\_time = 0, turnaround\_time = 0, end;

float average\_waiting\_time, average\_turnaround\_time;

printf("Enter the Number of Processes: ");

scanf("%d", &limit);

printf("Enter Arrival Time: ");

for(i = 0; i < limit; i++)

{

scanf("%d", &arrival\_time[i]);

}

printf("Enter Burst Time: ");

for(i = 0; i < limit; i++)

{

scanf("%d", &burst\_time[i]);

temp[i] = burst\_time[i];

}

burst\_time[9] = 9999;

printf("Processes Arrival time Burst Waiting Turn around \n");

for(time = 0; count != limit; time++)

{

smallest = 9;

for(i = 0; i < limit; i++)

{

if(arrival\_time[i] <= time && burst\_time[i] < burst\_time[smallest] && burst\_time[i] > 0)

{

smallest = i;

}

}

burst\_time[smallest]--;

if(burst\_time[smallest] == 0)

{

count++;

end = time + 1;

wt = end - arrival\_time[smallest] - temp[smallest];

tat = end - arrival\_time[smallest];

wait\_time = wait\_time + wt;

turnaround\_time = turnaround\_time + tat;

printf(" %d\t %d\t\t %d\t\t %d \t%d\n", smallest+1, arrival\_time[smallest], temp[smallest], wt, tat);

}

}

average\_waiting\_time = wait\_time / limit;

average\_turnaround\_time = turnaround\_time / limit;

printf("Average Waiting Time: %lf\n", average\_waiting\_time);

printf("Average Turnaround Time: %lf\n", average\_turnaround\_time);

return 0;

}

**OUTPUT:**

Text

Description automatically generated

**CONCLUSION:**

By executing the above program, we have successfully demonstrated the Shortest Job First CPU Scheduling.

**BANKER’S ALGORITHM**

**AIM:** To execute the banker’s algorithm and find whether a safe sequence exists or not.

**DESCRIPTION:**

Banker’s algorithm is used for resources which are of multiple instance type. It generally consists of

some data structures like:

1. Available: a vector of length m which keeps a track of the available resources

2. Max: a n\*m matrix which keeps a track of maximum allocated resources for a process

3. Allocation: a n\*m matrix which keeps a track of the currently allocated resources to a particular

process

4. Need: a n\*m matrix which keeps a track of additionally required resources for a particular process

apart from the allocated resources.

If the Banker’s algorithm generates a safe sequence, then there exists no deadlock else there exists

a deadlock.

**ALGORITHM:**

1. Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.

Initialize: Work = Available

Finish[i] = false; for i=1, 2, 3, 4….n

2. Find an i such that both

a) Finish[i] = false

b) Need[i] &lt;= Work

if no such i exists goto step (4)

3. Work = Work + Allocation[i]

Finish[i] = true

goto step (2)

4. if Finish [i] = true for all i then the system is in a safe state

**PROGRAM:**  
n=int(input("Enter the number of processes: "))

m=int(input("Enter the number of resources: "))

print("Enter the Allocated matrix: ")

alloc=[]

for i in range(n):

l=list(map(int, input().strip().split()))

alloc.append(l)

print("Enter the Max matrix: ")

max=[]

for i in range(n):

l=list(map(int, input().strip().split()))

max.append(l)

print("Enter the Avail matrix: ")

avail=list(map(int, input().strip().split()))

print("Allocated Matrix: ", alloc, end="\n")

print("Max Matrix: ", max, end="\n")

print("Avail Matrix: ", avail, end="\n")

f=[0]\*n

ans=[0]\*n

ind=0;

for k in range(n):

f[k] = 0

need=[[ 0 for i in range(m)] for i in range(n)]

for i in range(n):

for j in range(m):

need[i][j] = max[i][j] - alloc[i][j]

y=0

for k in range(n):

for i in range(n):

if (f[i] == 0):

flag = 0

for j in range(m):

if (need[i][j] > avail[j]):

flag=1

break

if (flag == 0):

ans[ind] = i

ind += 1

for y in range(m):

avail[y] += alloc[i][y]

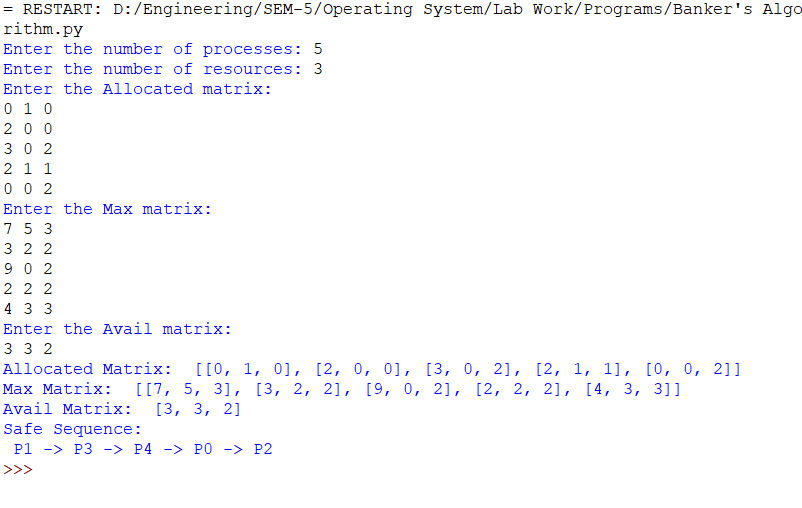
f[i]=1

print("Safe Sequence: ")

for i in range (n-1):

print(" P", ans[i], " ->", sep="", end="")

print(" P", ans[n-1], sep="")

**OUTPUT:**

**CONCLUSION:**

By executing the program, we have successfully the banker’s Algorithm and found whether a safe sequence exists or not.

**PAGING MEMORY MANAGEMENT TECHNIQUE**

**AIM:** To implement paging Memory Management Technique

**DESCRIPTION:**

Paging is the memory management technique in which secondary memory is divided into fixed-size blocks called pages, and main memory is divided into fixed-size blocks called frames. The Frame has the same size as that of a Page. The processes are initially in secondary memory, from where the processes are shifted to main memory (RAM) when there is a requirement. Each process is mainly divided into parts where the size of each part is the same as the page size. One page of a process is mainly stored in one of the memory frames. Paging follows no contiguous memory allocation. That means pages in the main memory can be stored at different locations in the memory.

**ALGORITHM:**Step-1: Start

Step-2: Read the memory size

Step-3: Read the Page size

Step-4: Calculate the number of pages

Step-5: Read the number of Frames

Step-6: Print the total size of the memory

Step-7: Read the frame number for each page and store them in page table

Step-8: Print the page table

Step-9: Read the logical address i.e., the page number and offset

Step-10: Search for the input page number if found go to step-12

Step-11: If not found print page not found go to step 15

Step-12: Read the base register

Step-13: Calculate Physical Address i.e., phy\_add = base\_reg+inp\_fra\*page\_size+inp\_off

Step-14: Print physical address

Step-15: End

**PROGRAM:**  
mem\_size = int(input('Enter memory size: '))

page\_size = int(input('Enter page size: '))

no\_of\_pages = mem\_size//page\_size

print('No of pages available in memory:',no\_of\_pages)

no\_of\_frames = int(input('Enter number of frames: '))

print('Total size of memory: ',page\_size\*no\_of\_frames)

page\_table = list()

for i in range(no\_of\_pages):

print('Enter frame number for page',i)

fno = int(input())

page\_table.append([i,fno])

print('Page Table:')

print(page\_table)

inp\_pno, inp\_off = map(int,input('Enter logical address: ').split())

inp\_fra = None

for i in page\_table:

if(i[0]==inp\_pno):

inp\_fra = i[1]

if(inp\_fra==None):

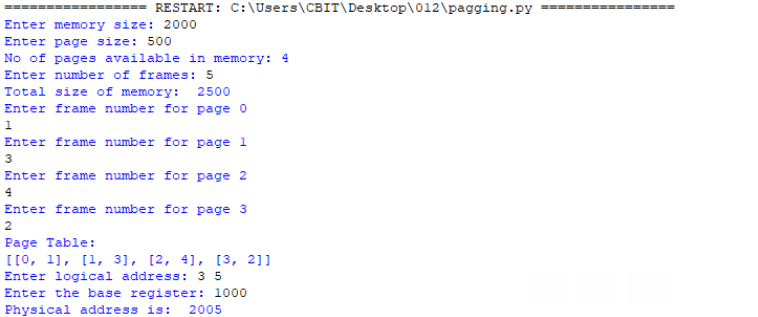
print('Page not found')

else:

base\_reg = int(input('Enter the base register: '))

phy\_add = base\_reg + inp\_fra\*page\_size+inp\_off

print('Physical address is: ',phy\_add)

**OUTPUT:**  
  
**CONCLUSION:**  
By executing the above program, we have successfully implemented the paging memory management technique.

**SEGMENTATION MEMORY MANAGEMENT TECHNIQUES**

**AIM:** To implement Segmentation Memory Management Techniques

**DESCRIPTION:**

Segmentation divides the user program and the secondary memory into uneven-sized blocks known as Segments. Segmentation can be divided into two types namely - Virtual Memory Segmentation and Simple Segmentation.

A Segment Table is used to store the information of all segments of the currently executing process. The swapping of the segments of the process results in the breaking of the free memory space into small pieces. This breaking of free space into pieces is called External Fragmentation.

**ALGORITHM:**  
Step-1: Start

Step-2: Read number of segments no\_of\_seg

Step-3: seg\_table -> []

Step-4: for i in range (no\_of\_seg):

Read the base and limit of segment

Seg\_table.append([base, limit])

Step-5: for i in seg\_table

Print i

Step-6: Read the logical address [Segment number, Offset]

Step-7: if (offset < seg\_table[inp\_seg\_no][1]):  
 phy\_add = seg\_table[inp\_seg\_no][0] + offset

Print phy\_add

Step-8: else

Print “offset is greater than limit

**PROGRAM:**

no\_of\_seg = int(input("Enter number of segments: "))

seg\_table = []

for i in range (no\_of\_seg):

    print("Enter base and limit of segment ", i)

    base, limit = map(int, input().split())

    seg\_table.append([base, limit])

print("SEGMENT TABLE")

for i in seg\_table:

    print(i)

print("Enter Logical Address [Segment number, Offset]")

inp\_seg\_no, offset = map(int, input().split())

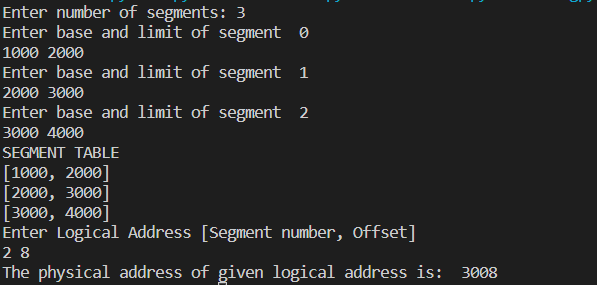
if(offset<seg\_table[inp\_seg\_no][1]):

    phy\_add = seg\_table[inp\_seg\_no][0] + offset

    print("The physical address of given logical address is: ", phy\_add)

else:

    print("Offset is greater than limit")

**OUTPUT:**

**CONCLUSION:**  
By executing the above program, we have successfully implemented the Segmentation Memory Management Technique

**IMPLEMENTATION OF FILE ALLOCATION METHODS**

**1. Contiguous File Allocation**

**AIM:** To implement the Contiguous File Allocation Method

**DESCRIPTION:**

In Contiguous File Allocation Method, each file occupies a contiguous set of blocks on the disk. For example, if a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be: b, b+1, b+2, …… b+n-1. This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file.

**ALGORITHM:**

Step 1: Start the program.

Step 2: Get the number of memory partition and their sizes.

Step 3: Get the number of processes and values of block size for each process.

Step 4: First fit algorithm searches all the entire memory block until a hole which is big enough is encountered. It allocates that memory block for the requesting process.

Step 5: Best-fit algorithm searches the memory blocks for the smallest hole which can be allocated to requesting process and allocates if.

Step 6: Worst fit algorithm searches the memory blocks for the largest hole and allocates it to the process.

Step 7: Analyses all the three memory management techniques and display the best algorithm which utilizes the memory resources effectively and efficiently.

Step 8: Stop the program.

**PROGRAM:**  
#include <stdio.h>

#include <conio.h>

main()

{

int n,i,j,b[20],sb[20],t[20],x,c[20][20];

printf("Enter no.of files: ");

scanf("%d",&n);

for(i=0;i<n;i++)

{

printf("Enter no. of blocks occupied by file-%d: ",i+1);

scanf("%d",&b[i]);

printf("Enter the starting block of file-%d: ",i+1);

scanf("%d",&sb[i]);

t[i]=sb[i];

for(j=0;j<b[i];j++)

c[i][j]=sb[i]++;

}

printf("Filename\tStart block\tlength\n");

for(i=0;i<n;i++)

printf("%d\t\t %d \t\t%d\n",i+1,t[i],b[i]);

printf("Enter file name:");

scanf("%d",&x);

printf("File name is:%d\n",x);

printf("length is:%d\n",b[x-1]);

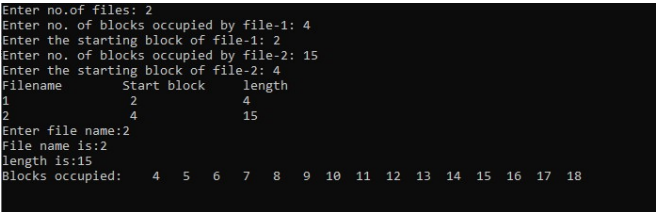
printf("Blocks occupied: ");

for(i=0;i<b[x-1];i++)

printf("%4d",c[x-1][i]);

getch();

}

**OUTPUT:**

**CONCLUSION:**

By executing the above program, we have successfully implemented the Contiguous File Allocation Method

**2. Indexed File Allocation**

**AIM:** To implement the Indexed File Allocation Method

**DESCRIPTION:**  
In Indexed File Allocation Method, a special block known as the Index block contains the pointers to all the blocks occupied by a file. Each file has its own index block. The ith entry in the index block contains the disk address of the ith file block.

**ALGORITHM:**

Step 1: Start.

Step 2: Let n be the size of the buffer

Step 3: check if there are any producer

Step 4: if yes check whether the buffer is full

Step 5: If no the producer item is stored in the buffer

Step 6: If the buffer is full the producer has to wait

Step 7: Check there is any consumer. If yes check whether the buffer is empty

Step 8: If no the consumer consumes them from the buffer

Step 9: If the buffer is empty, the consumer has to wait.

Step 10: Repeat checking for the producer and consumer till required

Step 11: Terminate the process.

**PROGRAM:**  
#include <stdio.h>

#include <conio.h>

main()

{

int n,m[20],i,j,sb[20],s[20],b[20][20],x;

printf("Enter no. of files: ");

scanf("%d",&n);

for(i=0;i<n;i++)

{

printf("Enter starting block and size of file-%d: ",i+1);

scanf("%d%d",&sb[i],&s[i]);

printf("Enter blocks occupied by file-%d: ",i+1);

scanf("%d",&m[i]);

printf("enter blocks of file-%d: ",i+1);

for(j=0;j<m[i];j++)

scanf("%d",&b[i][j]);

}

printf("\nFile\t index\tlength\n");

for(i=0;i<n;i++)

{

printf("%d\t%d\t%d\n",i+1,sb[i],m[i]);

}

printf("\nEnter file name: ");

scanf("%d",&x);

printf("File name is:%d\n",x);

i=x-1;

printf("Index is:%d\n",sb[i]);

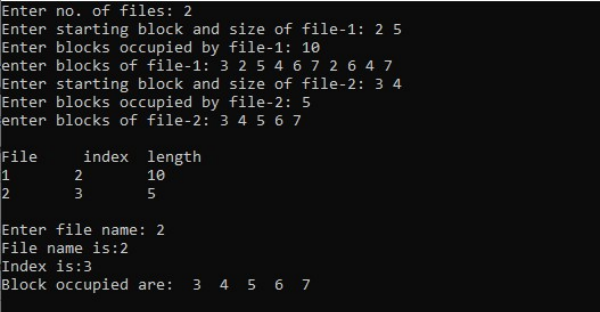
printf("Block occupied are:");

for(j=0;j<m[i];j++)

printf("%3d",b[i][j]);

getch();

}

**OUTPUT:**

**CONCLUSION:**

By executing the above program, we have successfully implemented the Indexed File Allocation Method.

**3. Linked File Allocation**

**AIM:** To implement the Linked File Allocation Method

**DESCRIPTION:**

In Linked File Allocation Method, each file is a linked list of disk blocks which need not becontiguous. The disk blocks can be scattered anywhere on the disk. The directory entry contains a pointer to the starting and the ending file block. Each block contains a pointer to the next block occupied by the file.

**ALGORITHM:**

Step 1: Create a queue to hold all pages in memory

Step 2: When the page is required replace the page at the head of the queue

Step 3: Now the new page is inserted at the tail of the queue

Step 4: Create a stack

Step 5: When the page fault occurs replace page present at the bottom of the stack

Step 6: Stop the allocation.

**PROGRAM:**  
#include <stdio.h>

#include <conio.h>

struct file

{

char fname[10];

int start,size,block[10];

}f[10];

main()

{

int i,j,n;

printf("Enter no. of files:");

scanf("%d",&n);

for(i=0;i<n;i++)

{

printf("Enter file name:");

scanf("%s",&f[i].fname);

printf("Enter starting block:");

scanf("%d",&f[i].start);

f[i].block[0]=f[i].start;

printf("Enter no.of blocks:");

scanf("%d",&f[i].size);

printf("Enter block numbers:");

for(j=1;j<=f[i].size;j++)

{

scanf("%d",&f[i].block[j]);

}

}

printf("File\tstart\tsize\tblock\n");

for(i=0;i<n;i++)

{

printf("%s\t%d\t%d\t",f[i].fname,f[i].start,f[i].size);

for(j=1;j<=f[i].size-1;j++)

printf("%d--->",f[i].block[j]);

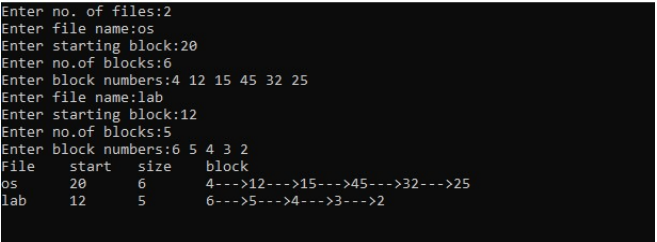
printf("%d",f[i].block[j]);

printf("\n");

}

getch();

}

**OUTPUT:**

**CONCLUSION:**

By executing the above program, we have successfully implemented the Linked File Allocation Method

**SOCKET PROGRAMMING**

**AIM:** To implement echo server using Socket Programming

**DESCRIPTION:**

**Sockets:** Provide a standard interface between the network and application. Allow communication between processes on the same machine or different machines. A socket is a communication end point. It basically contains an IP address and Port number. Socket in Unix/Linux domain provide communication between the processes in the same machine.

**Types of sockets**: There are four types of sockets, they are

1. **Stream Sockets**: Provide reliable byte stream transport service. These sockets guarantee the delivery of packets and the order in a network environment. Uses TCP (Transmission Control Protocol). Data records do not have any boundaries.
2. **Datagram Sockets**: Delivery in a network environment is not guaranteed. Uses UDP (User Datagram Protocol)

**Client Process**: Typically, a process which makes a request for information. After getting the response, a client process may terminate or may do some other processing. Ex. Web browser

**Server Process**: Is a process which takes a request from the clients and serves it. After getting a request from the client, this process will perform the required processing, gather the requested information and send it to the client. Once done, it becomes ready to serve another client. Server processes are always alert and ready to serve incoming requests. Ex: HTTP server, SMTP server, DNS, mail server etc.

**Types of Servers**: Servers can be:

1. **Iterative Server**: Serves its clients one after other. They cannot serve clients simultaneously and may be implemented when the service time s finite and short time then they can be implemented as iterative servers. Ex: Time server, DNS etc.
2. **Concurrent Server**: Is a server that can serve multiple client requests simultaneously. Simplest way to create concurrent servers, we use *fork ()* system call.

**Socket system calls**: In a client server environment, both the client and server have to create sockets for communication. The standard API for network programming in C is Berkely Sockets. This API was first introduced in 4.3BSD UNIX and now available on all Unix-like platforms including Linux, MacOS X, Free BSD and Solaris. A very similar network API is available on Windows.

**Server side**: socket(), bind(), listen(), accept(), close()

**Client side:** socket(), connect() and close()

**Both sides:** recv()/read(), send()/write()

**ALGORITHM:**

Step-1: Start

Step-2: sfd = socket(AF\_INET, SOCK\_STREAM, 0)

Step-3: Print sfd, strerror(errno)

Step-4: struct sockaddr\_in addr

Step-5: addr.sin\_family = AF\_INET

Step-6: addr.sin\_port = htons(8090)

Step-7: status = inet\_pton(AF\_INET, “127.0.1”, &addr.sin\_addr)  
Step-8: Print Sattus, strerror(errno)

Step-9: cfd = connect(sfd, (struct sockaddr\*)&addr, addrlen)

Step-10: Print cfd, strerror(errno)

Step-11: assert(cfd != -1)

Step-12: Read a buffer Message from the user

Step-13: status = send(sfd, buf, strlen(buf), 0)

Step-14: Print status, strerror(errno)

Step-15: assert(status != -1)

Step-16: status = read(sfd, buf, 1024)

Step-17: Print status, strerror(errno)

Step-18: assert(status!=-1)

Step-19: Print buf

Step-20: close(cfd)

Step-21: End

**PROGRAM:**  
**client.c:**

#include<assert.h>

#include<errno.h>

#include<stdio.h>

#include<string.h>

#include<arpa/inet.h>

#include<sys/socket.h>

#include<sys/types.h>

int main() {

int status;

int sfd = socket(AF\_INET, SOCK\_STREAM, 0);

printf("sfd: %d, %s\n", sfd, strerror(errno));

assert(sfd != -1);

struct sockaddr\_in addr;

socklen\_t addrlen = sizeof(addr);

addr.sin\_family = AF\_INET;

addr.sin\_port = htons(8090);

status = inet\_pton(AF\_INET, "127.0.0.1", &addr.sin\_addr);

printf("ip convert status: %d, %s\n", status, strerror(errno));

assert(status != -1);

int cfd = connect(sfd, (struct sockaddr\*)&addr, addrlen);

printf("cfd: %d, %s\n", cfd, strerror(errno));

assert(cfd != -1);

char buf[1024];

printf("Enter message: ");

scanf("%[^\n]", buf);

status = send(sfd, buf, strlen(buf), 0);

printf("send status: %d, %s\n", status, strerror(errno));

assert(status != -1);

status = read(sfd, buf, 1024);

printf("server status: %d, %s\n", status, strerror(errno));

assert(status != -1);

printf("Got: %s\n", buf);

close(cfd);

return 0;

}

**server.c:**

#include<assert.h>

#include<errno.h>

#include<stdio.h>

#include<string.h>

#include<arpa/inet.h>

#include<sys/socket.h>

#include<sys/types.h>

int main() {

int status;

int sfd = socket(AF\_INET, SOCK\_STREAM, 0);

printf("sfd: %d, %s\n", sfd, strerror(errno));

assert(sfd != -1);

struct sockaddr\_in addr;

socklen\_t addrlen = sizeof(addr);

addr.sin\_family = AF\_INET;

addr.sin\_port = htons(8090);

addr.sin\_addr.s\_addr = INADDR\_ANY;

status = bind(sfd, (struct sockaddr \*)&addr, addrlen);

printf("status: %d, %s\n", status, strerror(errno));

assert(status != -1);

status = listen(sfd, 5);

printf("status: %d, %s\n", status, strerror(errno));

assert(status != -1);

int nsfd = accept(sfd, (struct sockaddr \*)&addr, &addrlen);

printf("nsfd: %d, %s\n", nsfd, strerror(errno));

assert(nsfd != -1);

char buf[1024];

status = read(nsfd, buf, 1024);

printf("status: %d, %s\n", status, strerror(errno));

assert(status != -1);

printf("Got: %s\n", buf);

ssize\_t send\_status = send (nsfd, buf, strlen(buf), 0);

printf("send: %lu, %s\n", send\_status, strerror(errno));

assert(status != -1);

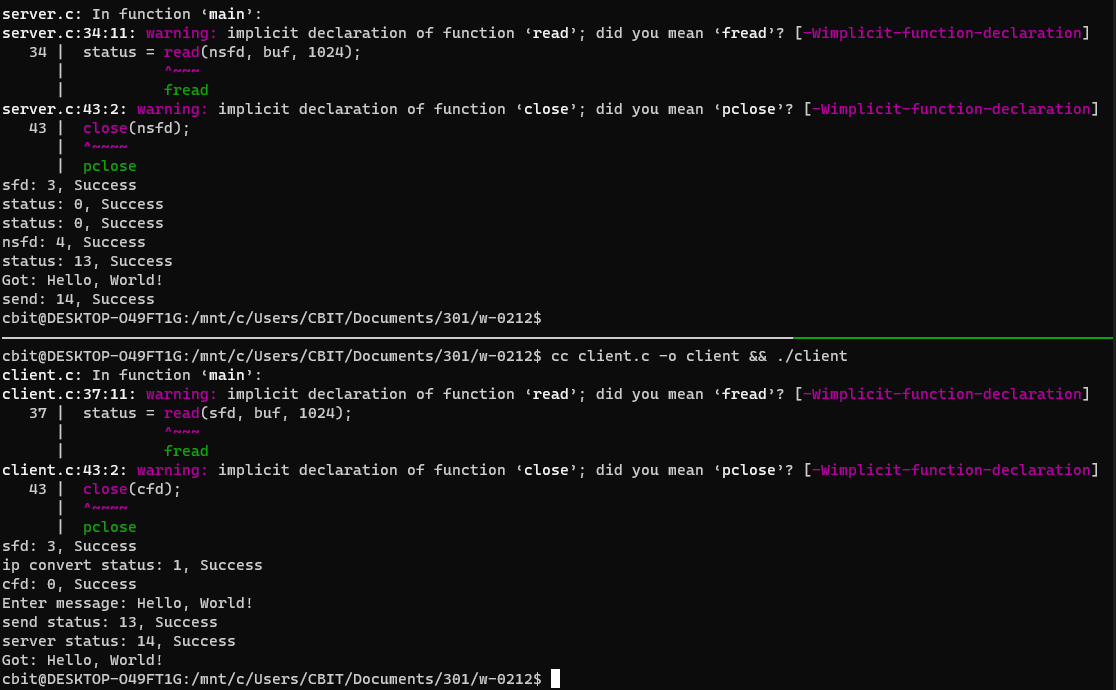
close(nsfd);

shutdown(sfd, SHUT\_RDWR);

return 0;

}

**OUTPUT:**



**CONCLUSION:**

By executing the above program, we have successfully implemented the echo-server using Socket Programming.