Operating System Lab

Name : Avisek Shaw

Department : CSE

Roll Number : 16500118051

Semester : 5th

Registration Number : 181650110016 of 2018-2019

Paper Name : Operating Systems

Paper Code : PCC-CS592

No:	TOPIC	Date	Teacher's Signature
1	**Brief Introduction to UNIX**		
1.1	Operating System	07/10/2020	
1.2	UNIX Operating System	07/10/2020	
1.3	Kernel	07/10/2020	
1.4	Shell	07/10/2020	
1.5	Mode Of Operation Of UNIX	07/10/2020	
1.6	System Calls	07/10/2020	
1.7	Features Of Unix	07/10/2020	
2	**Shell programming**		
2.1	Basic Command Structures	14/10/2020	
2.2	VI Editor	14/10/2020	
2.3	Basic Shell Syntax	14/10/2020	
2.3.1	Variable Declaration	14/10/2020	
2.3.2	Condition Checking	14/10/2020	
2.3.3	Control Structure	14/10/2020	
2.3.4	Loop	14/10/2020	
2.3.5	Functions	14/10/2020	
2.3.6	Programming Examples	14/10/2020	
2.3.6.1	Add two numbers	21/10/2020	
2.3.6.2	Add two numbers using command line	21/10/2020	
2.3.6.3	Multiply two numbers	21/10/2020	
2.3.6.4	Swap two value without using 3 variables	21/10/2020	
2.3.6.5	Calculate the gross salary	28/10/2020	
2.3.6.6	Count total no of file in a directory	28/10/2020	
2.3.6.7	Display the bigger number between 3 numbers	28/10/2020	
2.3.6.8	Calculate average marks of a student & corresponding grade	28/10/2020	
2.3.6.9	Calculate given 3 digit number is palindrome or not	11/11/2020	
2.3.6.10	Calculate factorial of a particular number	11/11/2020	
2.3.6.11	Calculate Fibonacci series up to a specific range	11/11/2020	
2.3.6.12	Display the content of a file without using the 'cat' command	11/11/2020	
2.3.6.13	Copy the content of a file into another file	11/11/2020	
2.3.6.14	Count total no of lines in a file without using 'wc' command	18/11/2020	
2.3.6.15	Menu driven shell program to perform addition, subtraction, multiplication, division, modulus	18/11/2020	
2.3.6.16	Menu driven shell program to count total no of files, subdirectories under a directory, count no of lines in a file containing 'y' and display first & last lines of a file	18/11/2020	
2.3.6.17	Calculate factorial of a number using function	18/11/2020	
2.3.6.18	Reverse a particular string using function	18/11/2020	

3	**Process**	30/12/2021	
3.1	Definition	30/12/2021	
3.2	Different system calls related to process creation	30/12/2021	
3.3	Write a shell program which implement process and perform a task of printing word-“BYE”.	30/12/2021	
4	**Semaphore**	30/12/2021	
4.1	Introduction	30/12/2021	
4.2	Functions related to semaphore	30/12/2021	
4.3	To write a LINUX/UNIX C Program for the Implementation of Producer Consumer Algorithm using Semaphore	30/12/2021	
5	**Thread**	06/01/2021	
5.1	Introduction	06/01/2021	
5.2	Different thread functions	06/01/2021	
5.3	Write a Unix C program to demonstrate use of pthread basic functions	06/01/2021	
6	**Inter process Communication**	06/01/2021	
6.1	Introduction	06/01/2021	
6.2	Pipe	06/01/2021	
6.3.1	Count total no of file in a directory.	06/01/2021	
6.3.2	Count total no of user who are currently logged in.	06/01/2021	
6.3.3	Count total no of line in a file which contain word ‘Kolkata’.	06/01/2021	
6.3.4	Count total no of sub-directory under a specific directory.	06/01/2021	

1 Brief Introduction to UNIX

What is Operating System?

An operating system (OS) is system software that manages computer hardware and software resources and provides common services for computer programs. The operating system is a component of the system software in a computer system. Application programs usually require an operating system to function.

What is UNIX Operating System?

The UNIX operating system is a set of programs that act as a link between the computer and the user.

The computer programs that allocate the system resources and coordinate all the details of the computer's internals is called the operating system or kernel.

Users communicate with the kernel through a program known as the shell. The shell is a command line interpreter; it translates commands entered by the user and converts them into a language that is understood by the kernel.

It is popular *multi-user*, *multitasking* operating system developed at Bell Labs in the early 1970s. Created by just a handful of programmers, UNIX was designed to be a small, flexible system used exclusively by programmers.

UNIX was one of the first operating systems to be written in a high-level programming language, namely C. This meant that it could be installed on virtually any computer for which a C compiler existed. This natural portability combined with its low price made it a popular choice among universities. (It was inexpensive because antitrust regulations prohibited Bell Labs from marketing it as a full-scale product.)

Kernel:

Kernel: The kernel is the heart of the operating system. It interacts with hardware and most of the tasks like memory management, task scheduling and file management.

Shell:

The shell is the utility that processes your requests. When you type in a command at your terminal, the shell interprets the command and calls the program that you want. The shell uses standard syntax for all commands. C Shell, Bourne Shell and Korn Shell are most famous shells which are available with most of the UNIX variants.

Mode Of Operation Of UNIX:

User Mode:

User mode is a mode where all user programs execute. It does not have access to RAM and hardware. The reason for this is because if all programs ran in kernel mode, they would be able to overwrite each other's memory. If it needs to access any of these features – it makes a call to the underlying API. Each process started by windows except of system process runs in user mode.

In User mode, the executing code has no ability to directly access hardware or reference memory. Code running in user mode must delegate to system APIs to access hardware or memory. Due to the protection afforded by this sort of isolation, crashes in user mode are always recoverable. Most of the code running on your computer will execute in user mode.

Kernel Mode:

Kernel mode is a mode where all kernel programs execute (different drivers). It has access to every resource and underlying hardware. Any CPU instruction can be executed and every memory address can be accessed. This mode is reserved for drivers which operate on the lowest level

In Kernel mode, the executing code has complete and unrestricted access to the underlying hardware. It can execute any CPU instruction and reference any memory address. Kernel mode is generally reserved for the lowest-level, most trusted functions of the operating system. Crashes in kernel mode are catastrophic; they will halt the entire PC.

System Calls:

System calls are some predefined functions which transfer the control from user mode to kernel mode.

A system call is a request for the operating system to do something on behalf of the user's program. The system calls are functions used in the kernel itself.

A system call can also be defined as a programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. This may include hardware-related services (for example, accessing a hard disk drive), creation and execution of new processes, and communication with integral kernel services such as process scheduling. System calls provide an essential interface between a process and the operating system.

Feature Of Unix:

Multi user:

A multi-user operating system allows more than one user to share the same computer system at the same time. It does this by time-slicing the computer processor at regular intervals between the various users.

This switching between user programs is done by part of the kernel. To switch from one program to another requires:

- a regular timed interrupt event (provided by a clock)
- saving the interrupted programs state and data
- restoring the next programs state and data
- running that program till the next timed interrupt occurs

Multi-tasking:

Multi-tasking operating systems permit the use of more than one program to run at once. It does this in the same way as a multi-user system, by rapidly switching the processor between the various programs.

A multi-user system is also a multi-tasking system. This means that a user can run more than one program at once, using key selection to switch between them.

Multi-tasking systems support foreground and background tasks. A foreground task is one that the user interacts directly with using the keyboard and screen. A background task is one that runs in the background (it does not have access to the keyboard). Background tasks are usually used for printing or backups.

Protection & Security:

User Accounts

Every UNIX-like system includes a root account, which is the only account that may directly carry out administrative functions. All of the other accounts on the system are unprivileged. This means these accounts have no rights beyond access to files marked with appropriate permissions, and the ability to launch network services.

File Permissions

Every file and directory on a UNIX-style system is marked with three sets of file permissions that determine how it may be accessed, and by whom:

- The permissions for the owner, the specific account that is responsible for the file
- The permissions for the group that may use the file
- The permissions that apply to all other accounts

Each set may have none or more of the following permissions on the item:

- read
- write
- execute

A user may only run a program file if they belong to a set that has the execute permission. For directories, the execute permission indicates that users in the relevant set may see the files within it, although they may not actually read, write or execute any file unless the permissions of that file permit it. Executable files with the setUID property automatically run with the privileges of the file owner, rather than the account that activates them. Avoid setting the execute permission or setUID on any file or directory unless you specifically require it.

Encrypted Storage

Create one or more encrypted volumes for your sensitive files. Each volume is a single file which may enclose other files and directories of your choice. To access the contents of the volume, you must provide the correct decryption password to a utility, which then makes the volume available as if it were a directory or drive. The contents of an encrypted volume cannot be read when the volume is not mounted. You may store or copy encrypted volume files as you wish without affecting their security.

The System Firewall

The netfilter framework included in the Linux kernel restricts incoming and outgoing network connections according to a set of rules that have been defined by the administrator. Several Linux distributions configure firewall rules by default, and offer utilities for managing simple firewall configurations. You may also manage the firewall rules on any Linux system with the standard iptables and ip6tables command-line utilities, or with third-party utilities such as Fire starter. If you decide to use iptables, remember that it only configures restrictions for IP version 4 connections, and that you will need to use ip6tables to setup rules for IP version 6 as well.

Fedora, Red Hat, and SUSE automatically enable the firewall and supply their own graphical configuration utilities. You must manually configure and enable the firewall on Debian and Ubuntu systems. Current releases of Ubuntu include a command-line utility called ufw for firewall configuration.

Those Linux distributions that enable a firewall by default use a netfilter configuration that blocks connections from other systems. Any attempt by a remote system to access a service on a blocked port simply fails. This means that no other system may connect to an installed service, unless you specifically choose to unblock the relevant port.

Application Isolation

The most common UNIX-like operating systems provide several methods of limiting the ability of a program to affect either other running programs, or the host system itself.

A Note on Viruses and Malware

The security features of UNIX-like systems described above combine to form a strong defense against malware:

- Software is often supplied in the form of packages, rather than programs
- If you download a working program, it cannot run until you choose to mark the files as executable
- By default, applications such as the OpenOffice.org suite and the Evolution email client do not run programs embedded in emails or documents
- Web browsers require you to approve the installation of plug-ins
- Software vulnerabilities can be rapidly closed by vendors supplying updated packages to the repositories

Although a virus could be written for use against current UNIX-like systems, no effective malware is known to exist. It is likely that any future malware would need the consent of a user on the system in order to install itself, significantly reducing the possibility that any such software would be able to spread across networks.

2 Shell programming:

Basic Command Structures:

who

The who command prints information about all users who are currently logged in.

Syntax: #who

who am i

Only print information about the user and host associated with standard input (the terminal where the command was issued). This method adheres to the POSIX standard

Syntax: #who am i

touch

The **touch** command updates the access and modification times of each FILE to the current system time.
If you specify a FILE that does not already exist, **touch** creates an empty file with that name

Syntax: #touch abc.txt

cat

It reads data from files, and outputs their contents. It is the simplest way to display the contents of a file at the command line.

It can be used to:

- Display text files
- Copy text files into a new document
- Append the contents of a text file to the end of another text file, combining them

The simplest way to use cat is to simply give it the name of a text file. It will display the contents of the text file on the screen.

Syntax: #cat abc.txt

cat sends its output to stdout (standard output), which is usually the terminal screen. However, you can redirect this output to a file using the shell redirection symbol "**>**".

Syntax: #cat abc.txt > xyz.txt

Instead of overwriting another file, you can also append a source text file to another using the redirection operator "**>>**".

Syntax: #cat abc.txt >> xyz.txt

mkdir

Short for "make directory", mkdir is used to create directories on a file system.If the specified *DIRECTORY* does not already exist, mkdir creates it.

Syntax: #mkdir programs

cd

The cd command, which stands for "change directory", changes the shell's current working directory.

It allows you to change your working directory. You use it to move around within the hierarchy of your file system.

Syntax: #cd fifa15

rm

The **rm** command removes (deletes) files or directories.

Syntax: #rm abc.txt

ls

Lists the contents of a directory.

List information about the *FILE*s (the current directory by default). Sort entries alphabetically.

Syntax:

- ls -l

Lists the total files in the directory and subdirectories, the names of the files in the current directory, their permissions, the number of subdirectories in directories listed, the size of the file, and the date of last modification.

- ls -laxo

Lists files with permissions, shows hidden files, displays them in a column format, and suppresses group information.

- ls ~

List the contents of your home directory by adding a tilde after the **ls** command.

- ls /

List the contents of your root directory.

- ls ../

List the contents of the parent directory.

- ls */

List the contents of all subdirectories.

- ls -d */

Display a list of directories in the current directory.

- ls *.{htm, php, cgi}

List all files containing the file extension .htm, .php, or .cgi

- ls -ltr

List files sorted by the time they were last modified in reverse order (most recently modified files last).

bc

bc is a language that supports arbitrary-precision numbers, meaning that it delivers accurate results regardless of how large (or very very small) the numbers are.

bc starts by processing code from all the files listed on the command line in the order listed. After all files have been processed, bc reads from the standard input. All code is executed as it is read.

Syntax: # `echo 5*6|bc`

expr

expr evaluates arguments as an expression.

Syntax: # expr 5 + 6

head

head, by default, prints the first 10 lines of each FILE to standard output.

Syntax:

#head myfile.txt

Display the first ten lines of **myfile.txt**.

#head -15 myfile.txt

Display the first fifteen lines of **myfile.txt**.

tail

tail prints the last 10 lines of each FILE to standard output.

Syntax:

#tail myfile.txt

Outputs the last 10 lines of the file **myfile.txt**.

#tail myfile.txt -n 100

Outputs the last **100** lines of the file **myfile.txt**.

sort

sort is a simple and very useful command which will rearrange the lines in a text file so that they are sorted, numerically and alphabetically. By default, the rules for sorting are:

- lines starting with a number will appear before lines starting with a letter;
- lines starting with a letter that appears earlier in the alphabet will appear before lines starting with a letter that appears later in the alphabet;
- lines starting with a lowercase letter will appear before lines starting with the same letter in uppercase.

Syntax:

Let's say you have a file, **data.txt**, which contains the following ASCII text:

apples
oranges
pears
kiwis
bananas

To sort the lines in this file alphabetically, use the following command:

#sort data.txt

...which will produce the following output:

apples
bananas
kiwis
oranges
pears

grep

grep, which stands for "global regular expression print," processes text line by line and prints any lines which match a specified pattern.

Syntax: #grep –w "hope" abc.txt

Search the file abc.txt for lines containing the character “hope”. Only lines containing the distinct word "hope" will be matched. Lines in which "hope" is *part* of a word will *not* be matched.

cut

Remove or "cut out" sections of each line of a file or files.

Syntax:

Let's say you have a file named **data.txt** which contains the following text:

one two three four five
alpha beta gamma delta epsilon

In this example, each of these words is separated by a tab character, not spaces. The tab character is the default delimiter of cut, so it will by default consider a field to be anything delimited by a tab.

To "cut" only the third field of each line, use the command:

#cut -f 3 data.txt

...which will output the following:

three
gamma

VI Editor:

The default editor that comes with the UNIX operating system is called vi (visual editor). [Alternate editors for UNIX environments include pico and emacs, a product of GNU.]
The UNIX vi editor is a full screen editor and has two modes of operation:

Command mode commands which cause action to be taken on the file, and

Insert mode in which entered text is inserted into the file.

In the command mode, every character typed is a command that does something to the text file being edited; a character typed in the command mode may even cause the vi editor to enter the insert mode. In the insert mode, every character typed is added to the text in the file; pressing the <Esc> (Escape) key turns off the Insert mode.
While there are a number of vi commands, just a handful of these is usually sufficient for beginning vi users. To assist such users, this Web page contains a sampling of basic vi commands. The most basic and useful commands are marked with an asterisk (* or star) in the tables below. With practice, these commands should become automatic.
NOTE: Both UNIX and vi are case-sensitive. Be sure not to use a capital letter in place of a lowercase letter; the results will not be what you expect.

Basic Shell Syntax:

Variable Declaration:

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).

The following examples are valid variable names −

_ALI
TOKEN_A
VAR_1
VAR_2

Following are the examples of invalid variable names
− 2_VAR
-VARIABLE
VAR1-VAR2
VAR_A!

Defining Variables
Variables are defined as follows −
variable_name=variable_value

For example:
NAME="Zara Ali"

Above example defines the variable NAME and assigns it the value "Zara Ali". Variables of this type are called scalar variables. A scalar variable can hold only one value at a time.
The shell enables you to store any value you want in a variable. For example −
VAR1="Zara Ali"
VAR2=100

Condition Checking:

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not, if yes then condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	[$a -le $b] is true.
==	Equality - Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!=	Not Equality - Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [$a <= $b] is correct where as [$a <= $b] is incorrect.

Control Structure:

The *control flow* commands alter the order of execution of commands within a shell script. They include the **if...then, for... in, while, until,** and **case** statements. In addition, the **break** and **continue** statements work in conjunction with the control flow structures to alter the order of execution of commands within a script.

- **If-else if-fi:**

The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

Syntax

```
if [ expression 1 ]
then
   Statement(s) to be executed if expression 1 is
true elif [ expression 2 ]
then
   Statement(s) to be executed if expression 2 is
true elif [ expression 3 ]
then
   Statement(s) to be executed if expression 3 is
true else
   Statement(s) to be executed if no expression is true
fi
```

There is nothing special about this code. It is just a series of *if* statements, where each *if* is part of the *else* clause of the previous statement. Here statement(s) are executed based on the true condition, if non of the condition is true then *else* block is executed.

- **Switch case:**

You can use multiple if...elif statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Shell support case...esac statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.

Syntax

The basic syntax of the case...esac statement is to give an expression to evaluate and several different statements to execute based on the value of the expression. The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

```
case word in
  pattern1)
     Statement(s) to be executed if pattern1 matches
     ;;
  pattern2)
     Statement(s) to be executed if pattern2 matches
     ;;
  pattern3)
     Statement(s) to be executed if pattern3 matches
     ;;
Esac
```

Here the string word is compared against every pattern until a match is found. The statement(s) following the matching pattern executes. If no matches are found, the case statement exits without performing any action.
There is no maximum number of patterns, but the minimum is one.
When statement(s) part executes, the command ;; indicates that program flow should jump to the end of the entire case statement. This is similar to break in the C programming language.

Loop:

- **While loop:**

The while loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.

Syntax

```
while command
do
   Statement(s) to be executed if command is true
done
```

Here Shell *command* is evaluated. If the resulting value is *true*, given*statement(s)* are executed. If *command* is *false* then no statement would be not executed and program would jump to the next line after done statement.

Example

Here is a simple example that uses the while loop to display the numbers zero to nine −

```
#!/bin/sh
a=0
while [ $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

This will produce following result −

```
0
1
2
3
4
5
6
7
8
9
```

Each time this loop executes, the variable a is checked to see whether it has a value that is less than 10. If the value of a is less than 10, this test condition has an exit status of 0. In this case, the current value of a is displayed and then a is incremented by 1.

- **For loop:**

The for loop operate on lists of items. It repeats a set of commands for every item in a list.

Syntax

```
for var in word1 word2 ... wordN
do
   Statement(s) to be executed for every word.
done
```

Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

Example

Here is a simple example that uses for loop to span through the given list of numbers −

```
#!/bin/sh
for var in 0 1 2 3 4 5 6 7 8 9
do
   echo $var
done
```

This will produce following result −

```
0
1
2
3
4
5
6
7
8
9
```

Following is the example to display all the files starting with **.bash** and available in your home. I'm executing this script from my root −

```
#!/bin/sh

for FILE in $HOME/.bash*
do
   echo $FILE
done
```

This will produce following result −

```
/root/.bash_history
/root/.bash_logout
/root/.bash_profile
/root/.bashrc
```

- **Until Loop:**

The while loop is perfect for a situation where you need to execute a set of commands while some condition is true. Sometimes you need to execute a set of commands until a condition is true.

Syntax

```
until command
do
   Statement(s) to be executed until command is true
```

done

Here Shell *command* is evaluated. If the resulting value is *false*, given*statement(s)* are executed. If *command* is *true* then no statement would be not executed and program would jump to the next line after done statement.

Example

Here is a simple example that uses the until loop to display the numbers zero to nine −

```
#!/bin/sh

a=0

until [ ! $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

This will produce following result −

```
0
1
2
3
4
5
6
7
8
9
```

Functions:

Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual task when it is needed.

Using functions to perform repetitive tasks is an excellent way to create code reuse. Code reuse is an important part of modern object-oriented programming principles. Shell functions are similar to subroutines, procedures, and functions in other programming languages.

Creating Functions

To declare a function, simply use the following syntax −

```
function_name () {
   list of commands
}
```

The name of your function is function_name, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, which are followed by a list of commands enclosed within braces.

Example

Following is the simple example of using function −

```
#!/bin/sh

# Define your function here
Hello () {
   echo "Hello World"
}

# Invoke your function
Hello
```

When you would execute above script it would produce following result −

```
$./test.sh
Hello World
$
```

Pass Parameters to a Function

You can define a function which would accept parameters while calling those function. These parameters would be represented by $1, $2 and so on.

Following is an example where we pass two parameters *Zara* and *Ali* and then we capture and print these parameters in the function.

```
#!/bin/sh

# Define your function here
Hello () {
   echo "Hello World $1 $2"
}

# Invoke your function
Hello Zara Ali
```

This would produce following result −

```
$./test.sh
Hello World Zara Ali
$
```

Returning Values from Functions

If you execute an exit command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.

If you instead want to just terminate execution of the function, then there is way to come out of a defined function.

Based on the situation you can return any value from your function using the return command whose syntax is as follows −

return code

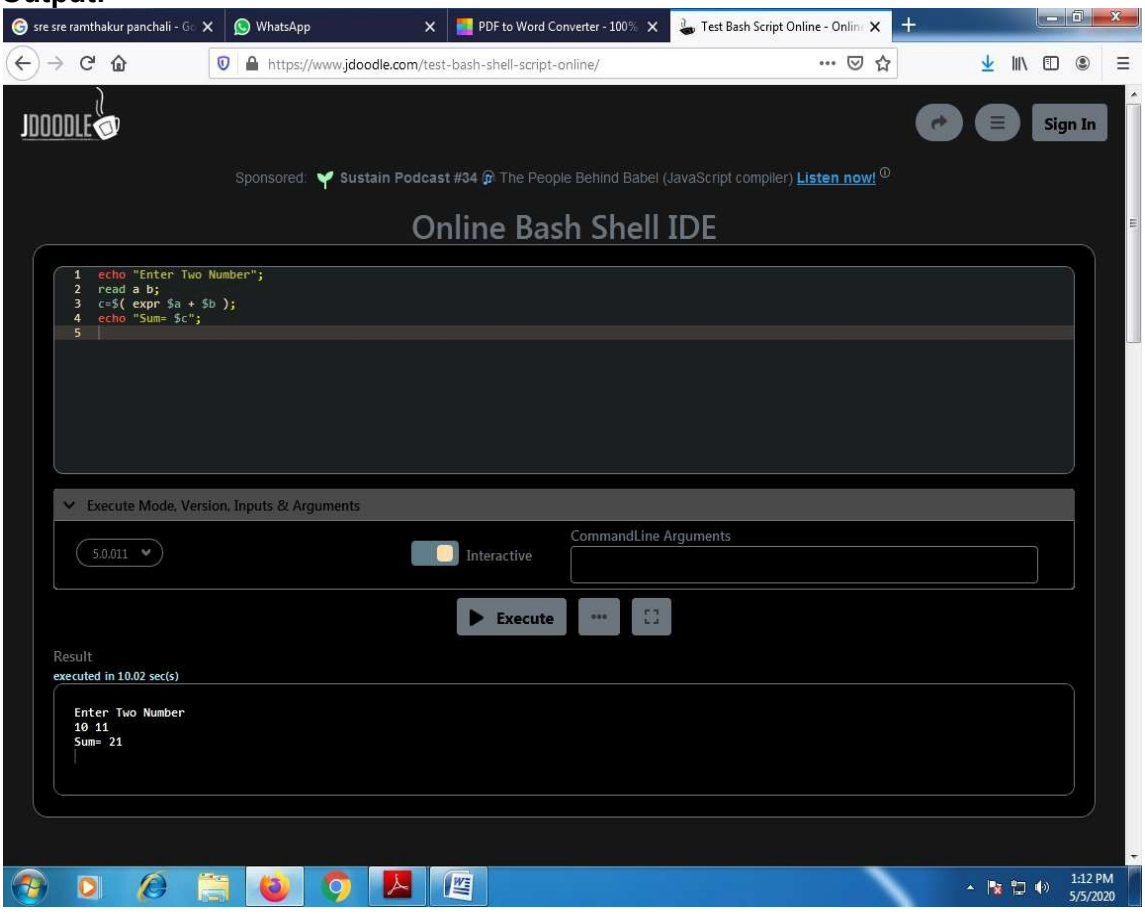
Here *code* can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

Programming Examples:

Write a shell program which add two numbers.

```
echo "Enter Two Number"
read a b
c=$( expr $a + $b )
echo "Sum= $c"
```

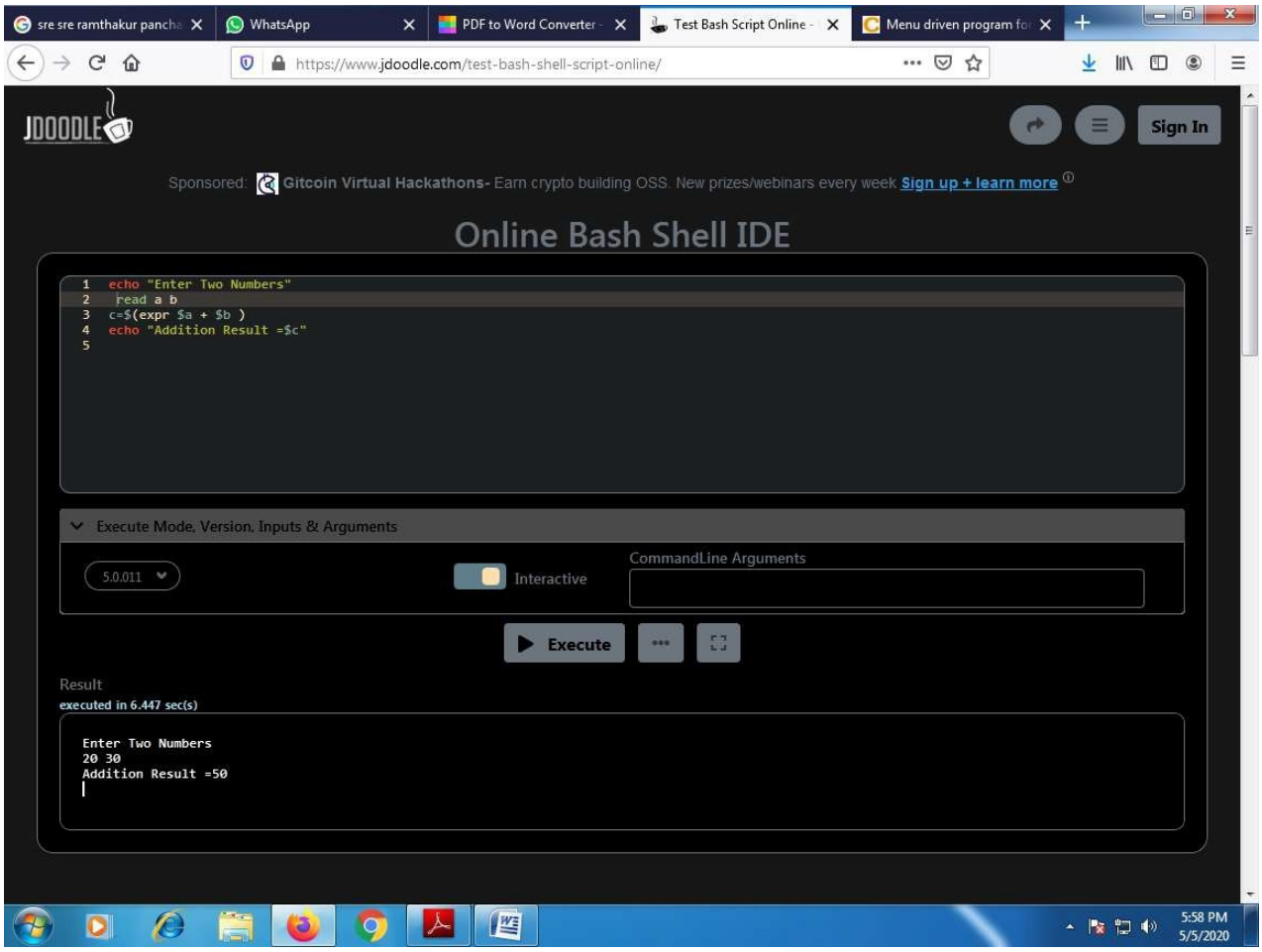
Output:



Write a shell program which add two numbers using command line.

```
c=$( expr $1 + $2 )
echo "Sum= $c"
```

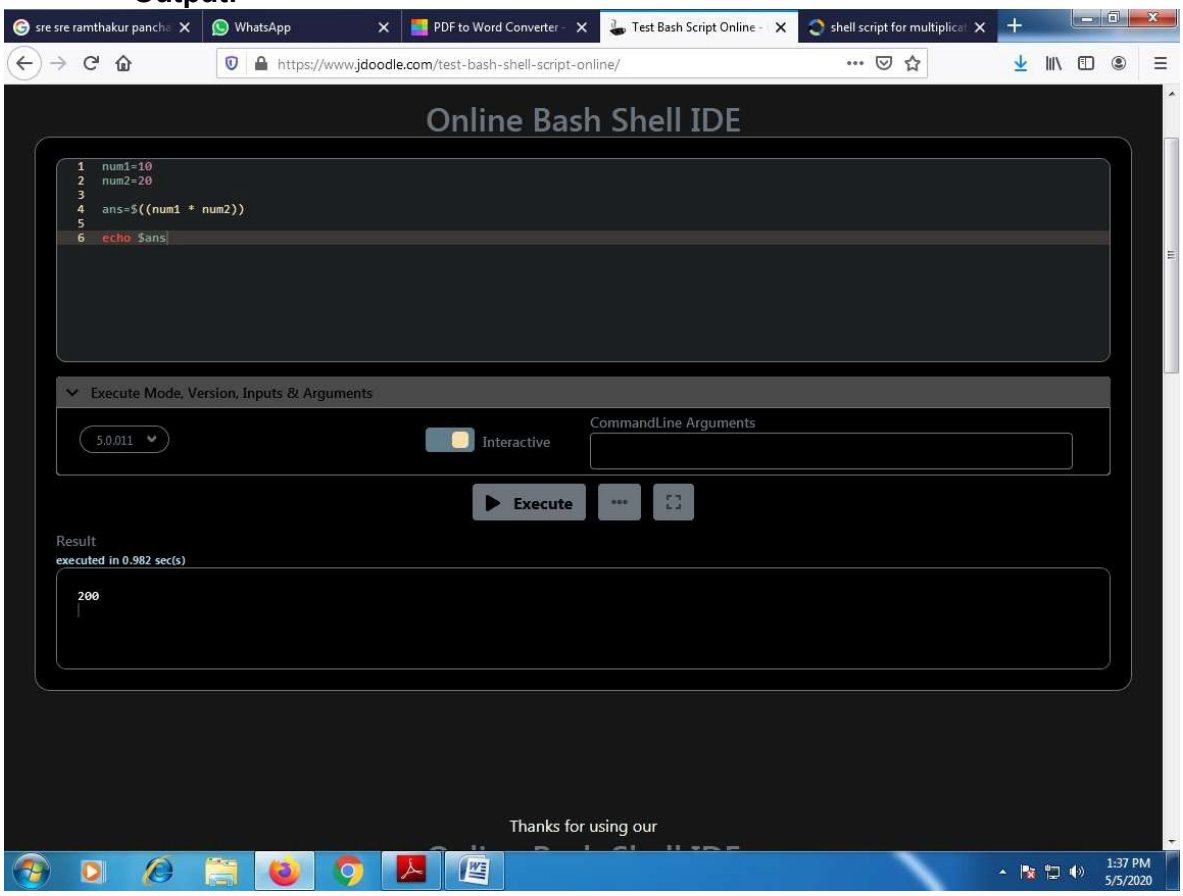
Output:



Write a shell program which multiply two numbers.

```
num1=
10
num2=
20
ans=$((num1 *
num2)) echo $ans
```

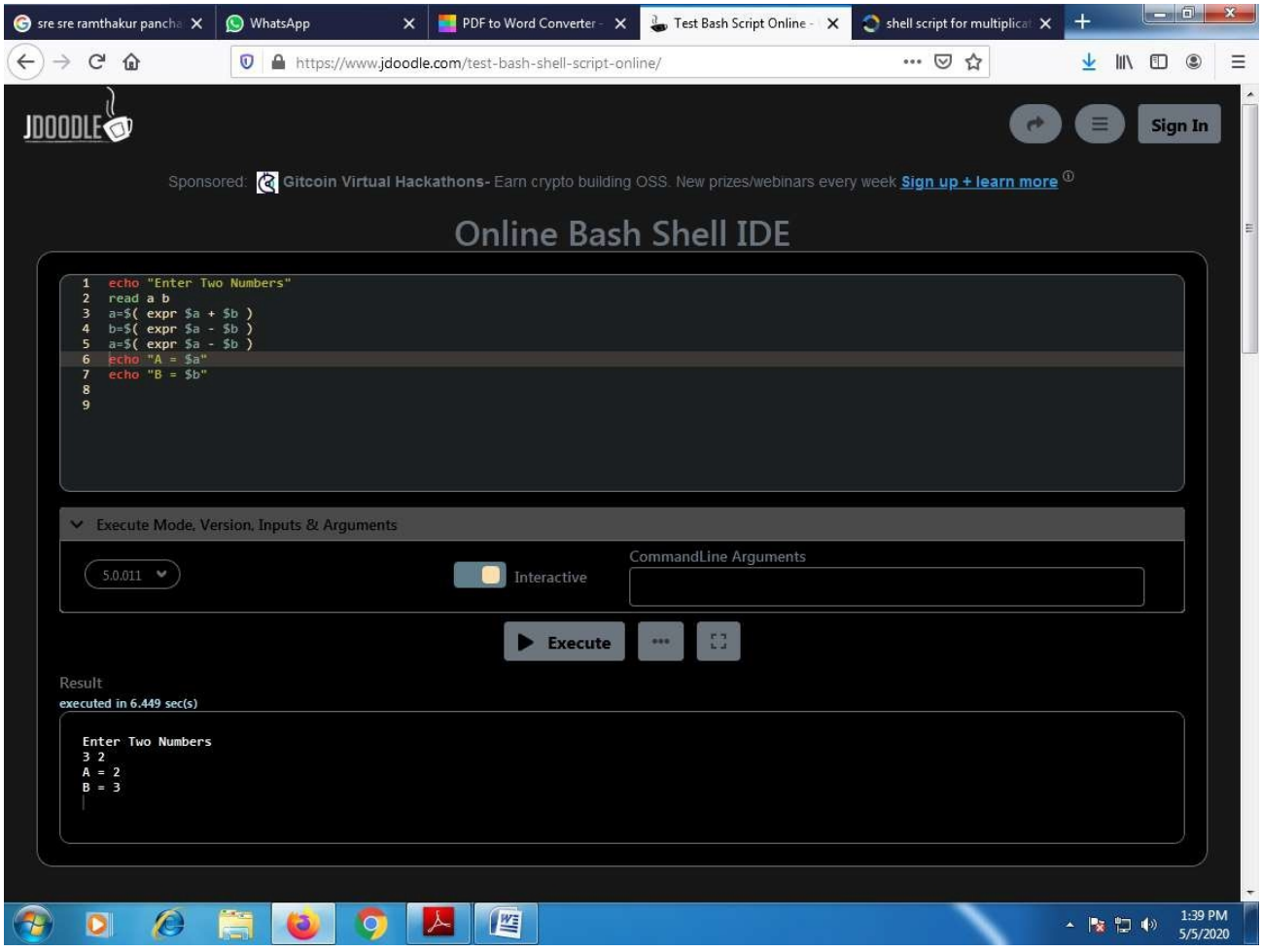
Output:



Write a shell program which swap two value without using 3 variables.

```
echo "Enter Two Numbers"
read a b
a=$( expr $a + $b )
b=$( expr $a - $b )
a=$( expr $a - $b )
echo "A = $a"
echo "B = $b"
```

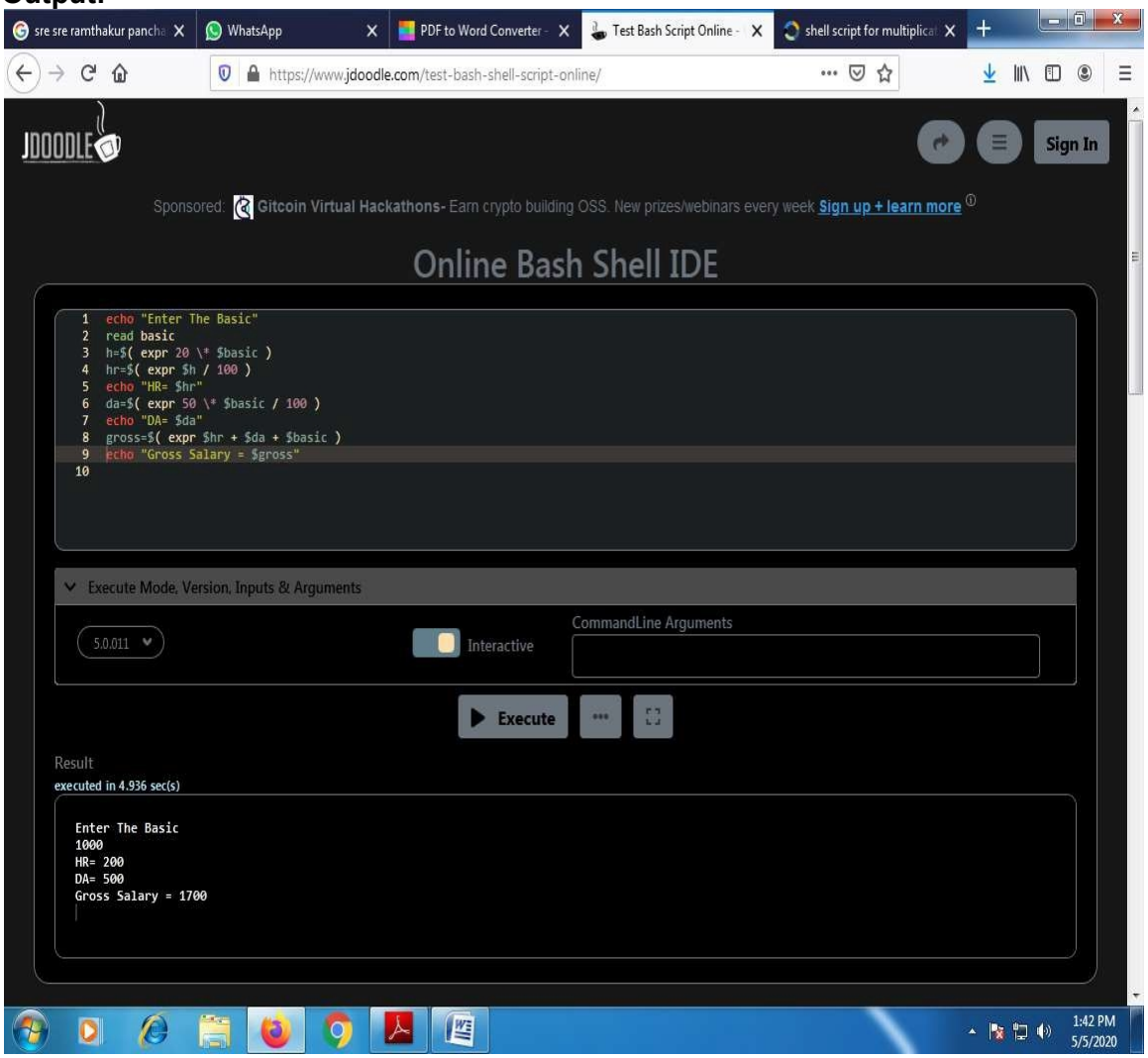
Output:



Write a shell program which calculate the gross salary of an employee where basic are provided through keyboard and
HR=20%
basic DA=50%
basic Gross Salary=(Basic+HR+DA)

```
echo "Enter The Basic"
read basic
h=$( expr 20 \* $basic )
hr=$( expr $h / 100 )
echo "HR= $hr"
da=$( expr 50 \* $basic / 100 )
echo "DA= $da"
gross=$( expr $hr + $da + $basic )
echo "Gross Salary = $gross"
```

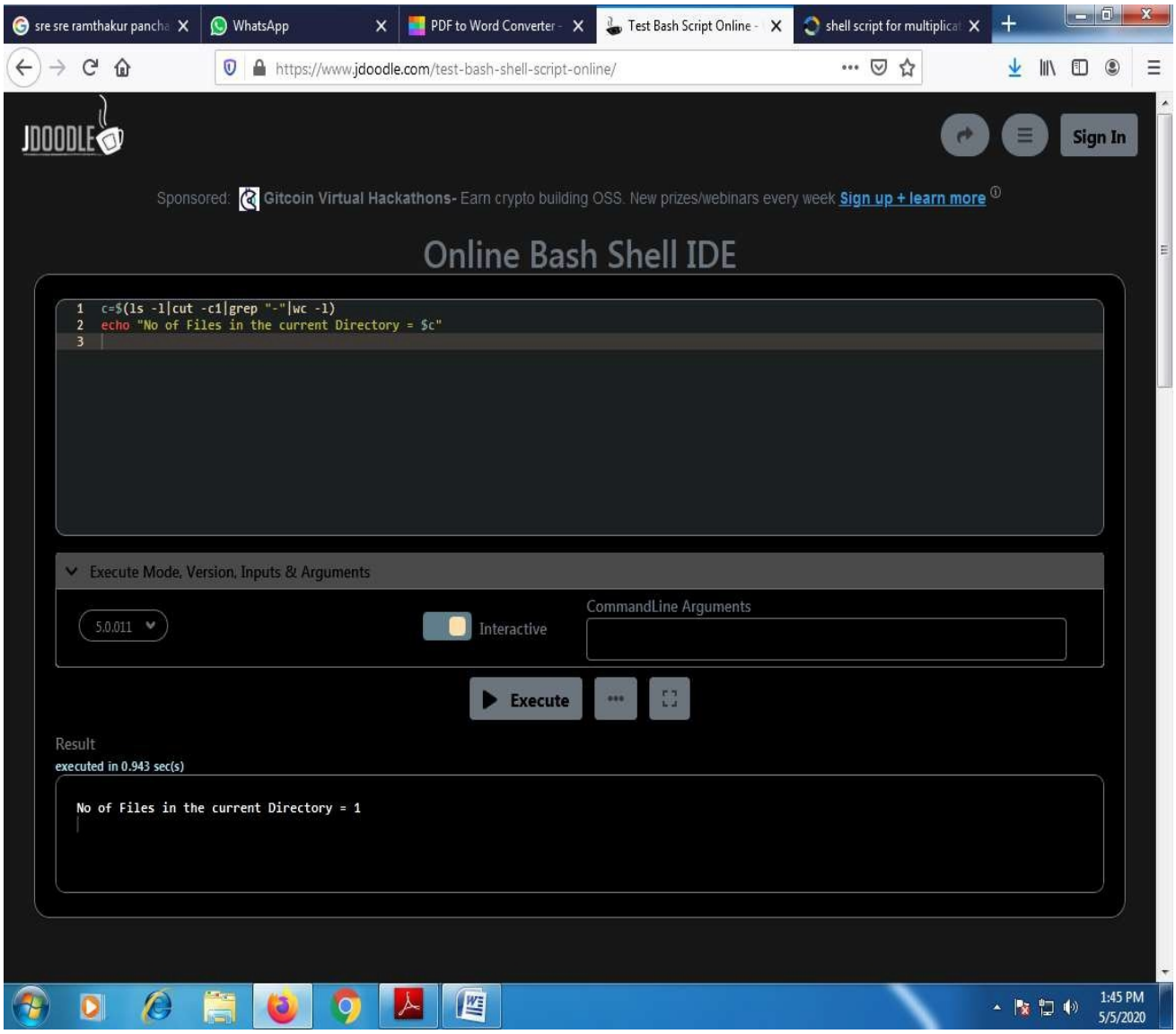
Output:



Write a shell program which count total no of file in a directory.

```
c=$(ls -l|cut -c1|grep "-"|wc -l)
echo "No of Files in the current Directory = $c"
```

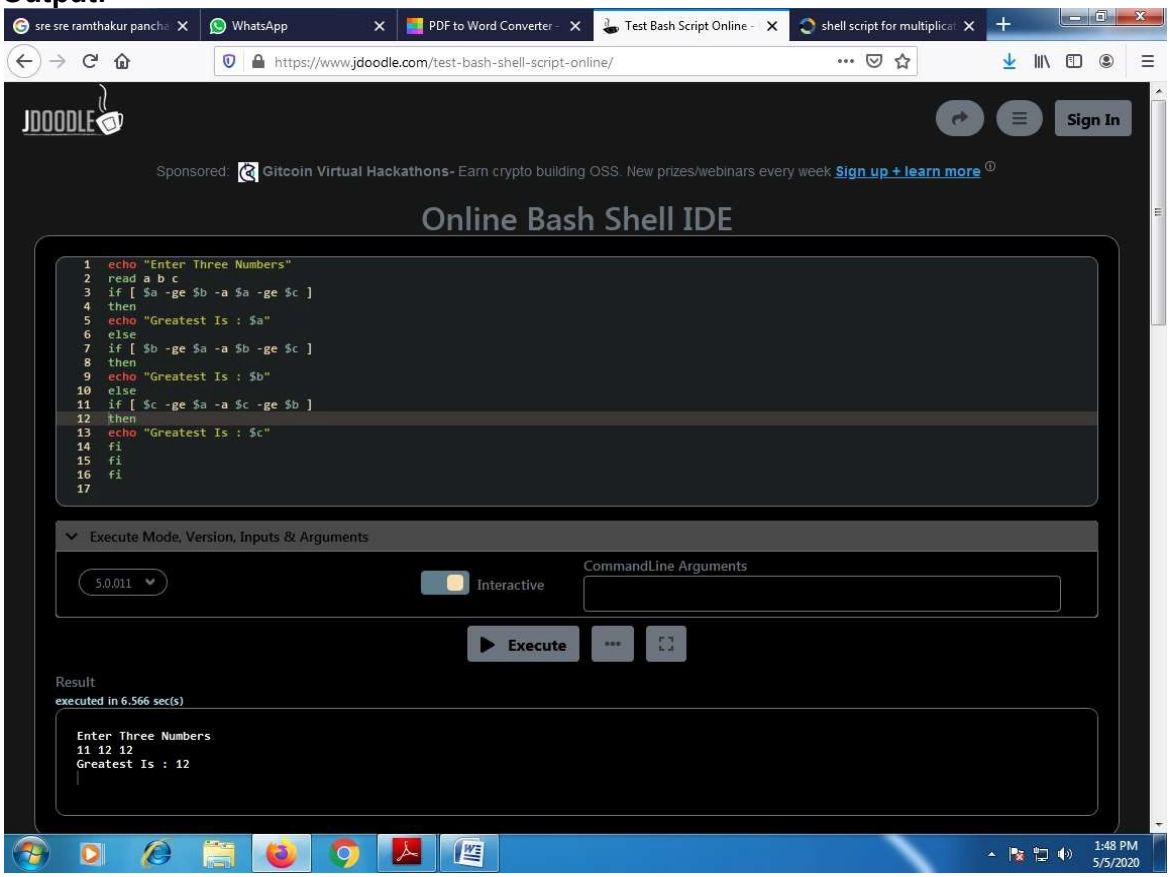
Output:

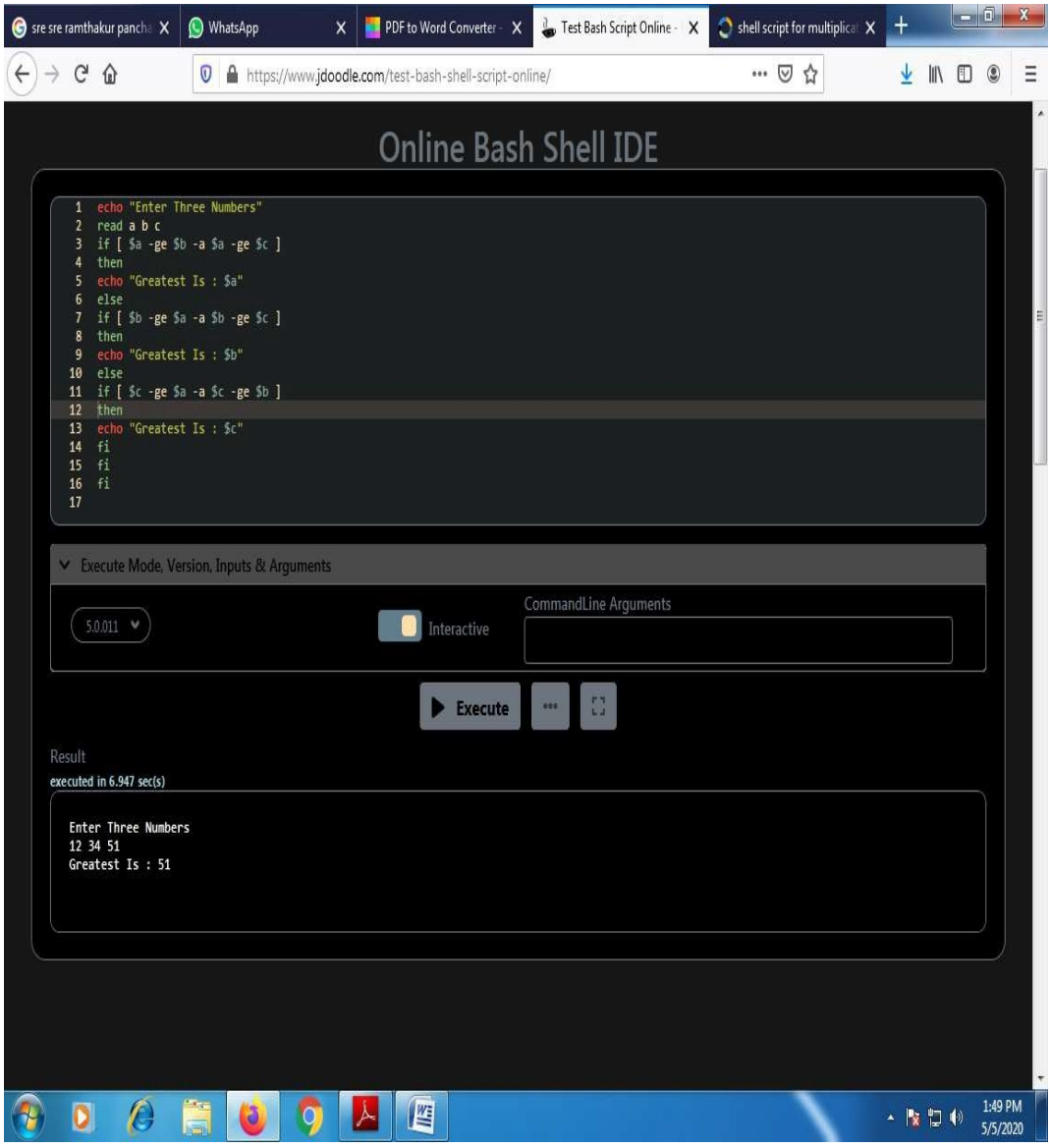


Write a shell program which display the bigger number between 3 numbers.

```
echo "Enter Three Numbers"
read a b c
if [ $a -ge $b -a $a -ge $c ]
then
    echo "Greatest Is : $a"
    else
       if [ $b -ge $a -a $b -ge $c ]
       then
          echo "Greatest Is : $b"
          else
             if [ $c -ge $a -a $c -ge $b ]
             then
                echo "Greatest Is : $c"
             fi
       fi
fi
```

Output:



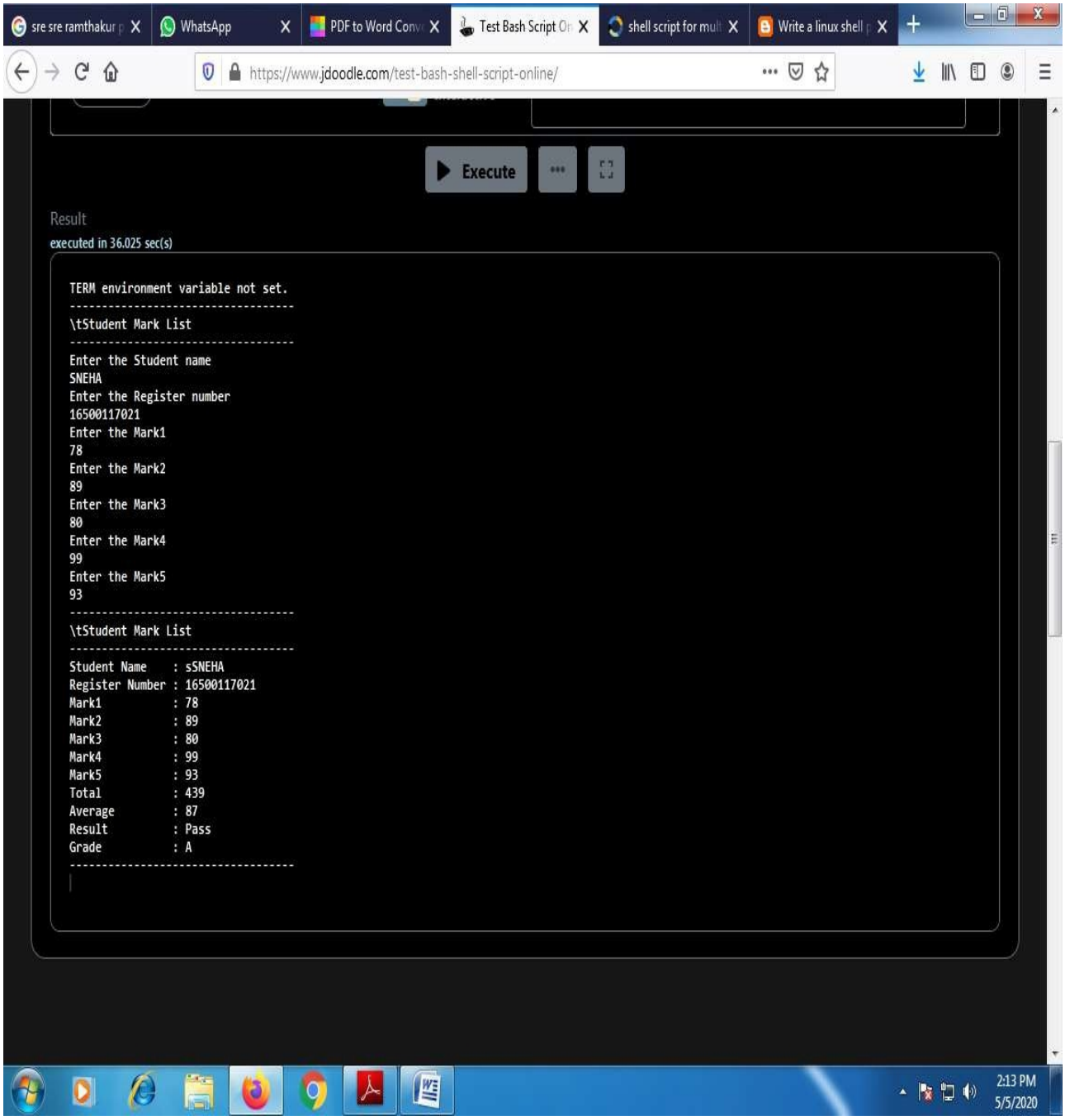

sre sre ramthakur pancha
WhatsApp
PDF to Word Converter
Test Bash Script Online
shell script for multiplica
https://www.jdoodle.com/test-bash-shell-script-online/
Online Bash Shell IDE
1 echo "Enter Three Numbers"
2 read a b c
3 if [$a -ge $b -a $a -ge $c]
4 then
5 echo "Greatest Is : $a"
6 else
7 if [$b -ge $a -a $b -ge $c]
8 then
9 echo "Greatest Is : $b"
10 else
11 if [$c -ge $a -a $c -ge $b]
12 then
13 echo "Greatest Is : $c"
14 fi
15 fi
16 fi
17
Execute Mode, Version, Inputs & Arguments
5.0.011
Interactive
CommandLine Arguments
Execute
Result
executed in 6.947 sec(s)
Enter Three Numbers
12 34 51
Greatest Is : 51
1:49 PM
5/5/2020

Write a shell program which calculate average marks of a student considering that no of subject is 5 and also calculate grade based in average marks.

```
clear
echo .....
echo '\tStudent Mark List'
echo.....
echo Enter the Student name
read name
echo Enter the Register number
read rno
echo Enter the Mark1
read m1
echo Enter the Mark2
read m2
echo Enter the Mark3
read m3
echo Enter the Mark4
read m4
echo Enter the Mark5
read m5
tot=$(expr $m1 + $m2 + $m3 + $m4 + $m5) avg=$
(expr $tot / 5)
echo .....
echo '\tStudent Mark List'
echo.....
echo  "Student  Name  :  $name"
echo "Register Number :  $rno"
echo "Mark1            : $m1"
echo "Mark2            : $m2"
echo "Mark3            : $m3"
echo "Mark4            : $m4"
echo "Mark5            : $m5"
echo "Total           : $tot"
echo "Average          : $avg"
if [ $m1 -ge 35 ]  && [ $m2 -ge 35 ] && [ $m3 -ge 35 ] && [ $m4 -ge 35 ] && [
```

```
$m5 -ge 35 ]
then
 echo "Result          : Pass"
 if [ $avg -ge 90 ]
then
echo "Grade           : S"
 elif [ $avg -ge 80 ]
 then
echo "Grade           : A"
 elif [ $avg -ge 70 ]
 then
 echo "Grade           : B"
elif [ $avg -ge 60 ]
then
 echo "Grade           : C"
elif [ $avg -ge 50 ]
then
echo "Grade           : D"
elif [ $avg -ge 35 ]
 then
echo "Grade           : E"
 fi
else
echo "Result          : Fail"
fi
echo .....
```

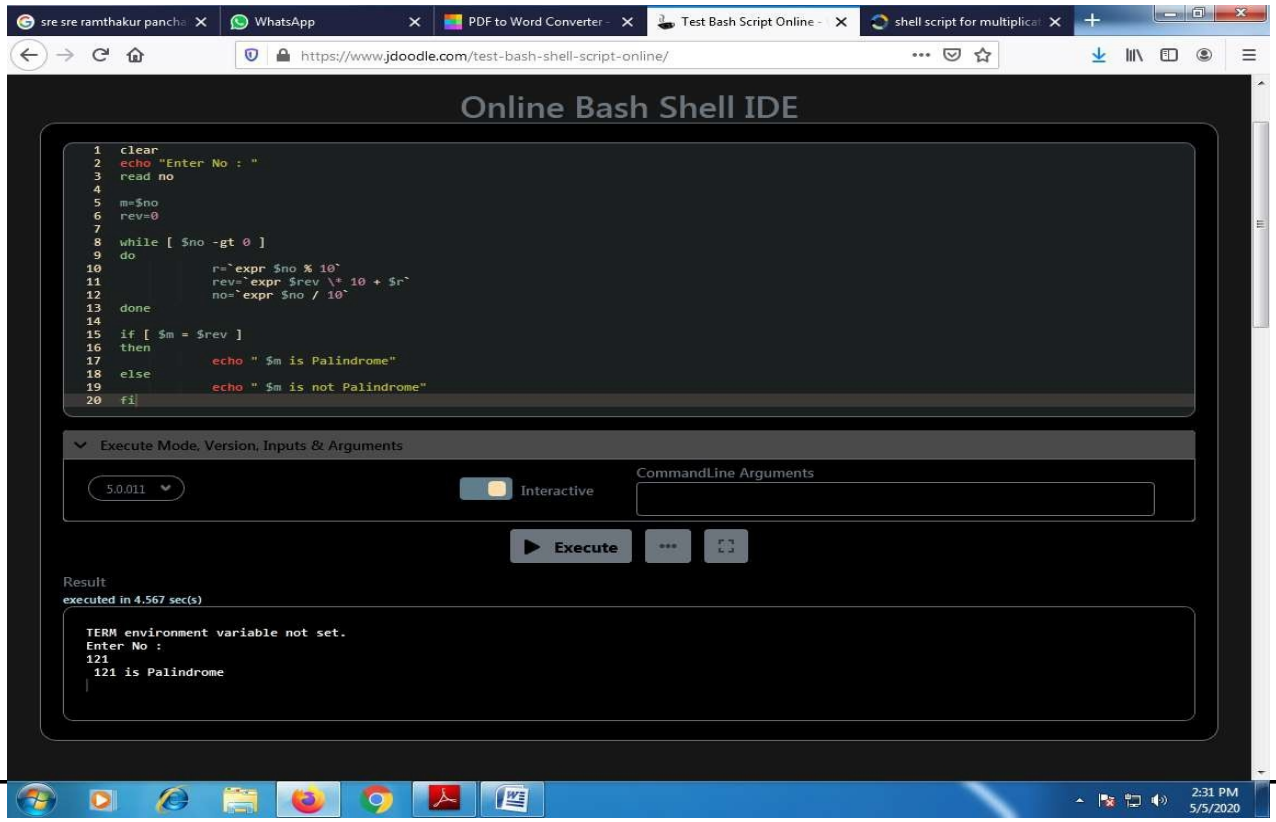
OUTPUT:


sre sre ramthakur p
WhatsApp
PDF to Word Conv
Test Bash Script On
shell script for mult
Write a linux shell p
https://www.jdoodle.com/test-bash-shell-script-online/
Execute
Result
executed in 36.025 sec(s)
TERM environment variable not set.
\tStudent Mark List
Enter the Student name
SNEHA
Enter the Register number
16500117021
Enter the Mark1
78
Enter the Mark2
89
Enter the Mark3
80
Enter the Mark4
99
Enter the Mark5
93
\tStudent Mark List
Student Name : sSNEHA
Register Number : 16500117021
Mark1 : 78
Mark2 : 89
Mark3 : 80
Mark4 : 99
Mark5 : 93
Total : 439
Average : 87
Result : Pass
Grade : A
2:13 PM
5/5/2020

Write a shell program which calculate whether a given 3 digit number is palindrome or not.

```
clear
echo "Enter No :
" read no
m=$
no
rev=
0
while [ $no -gt
0 ] do
        r=`expr $no % 10`
        rev=`expr $rev \* 10
        + $r` no=`expr
        $no / 10`
done
if [ $m =
$rev ] then
        echo " $m is
 else
        Palindrome" echo " $m
fi
        is not Palindrome"
```

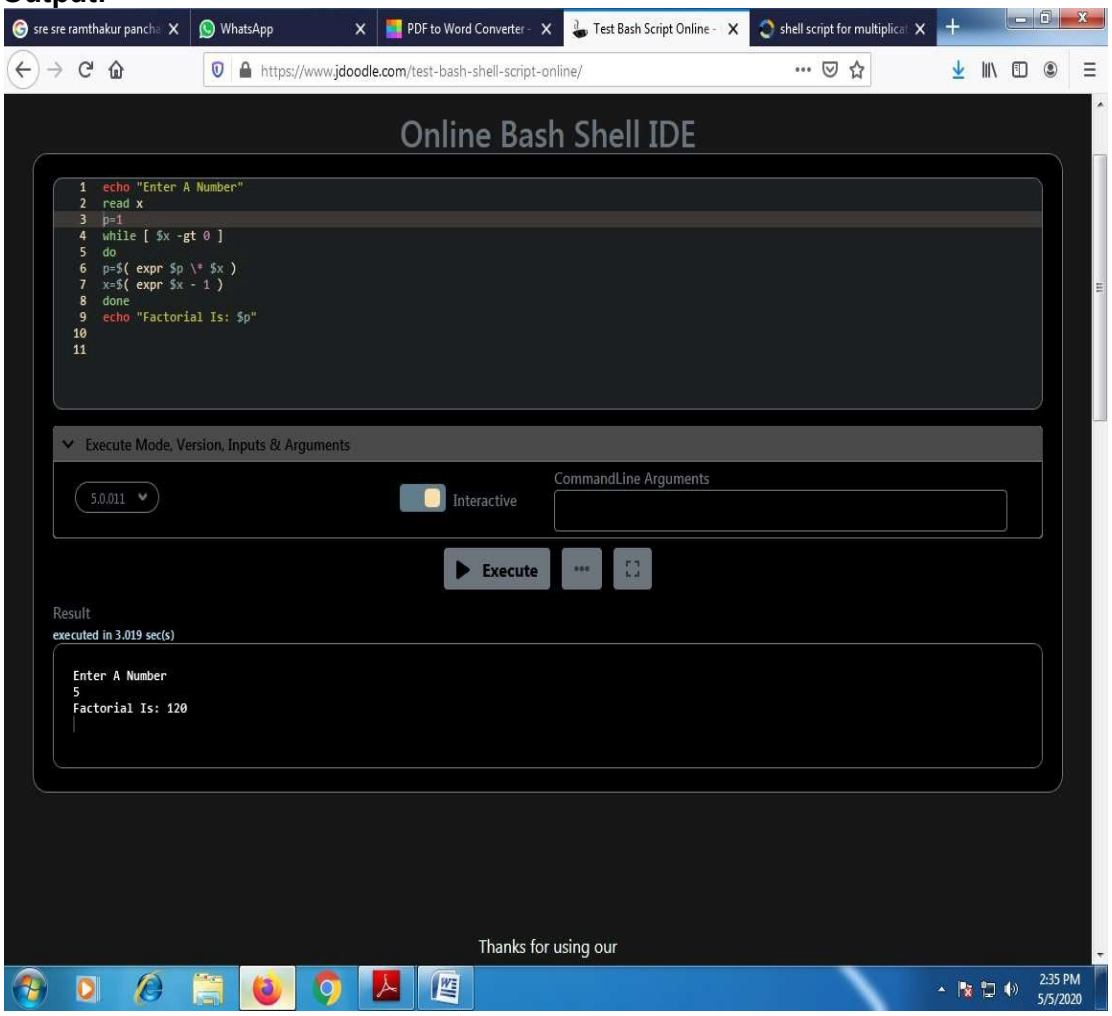
Output:



Write a shell program which calculate factorial of a particular number.

```
echo "Enter A Number"
read x
p=1
while [ $x -gt 0 ]
do
p=$( expr $p \* $x )
x=$( expr $x - 1 )
done
echo "Factorial Is: $p"
```

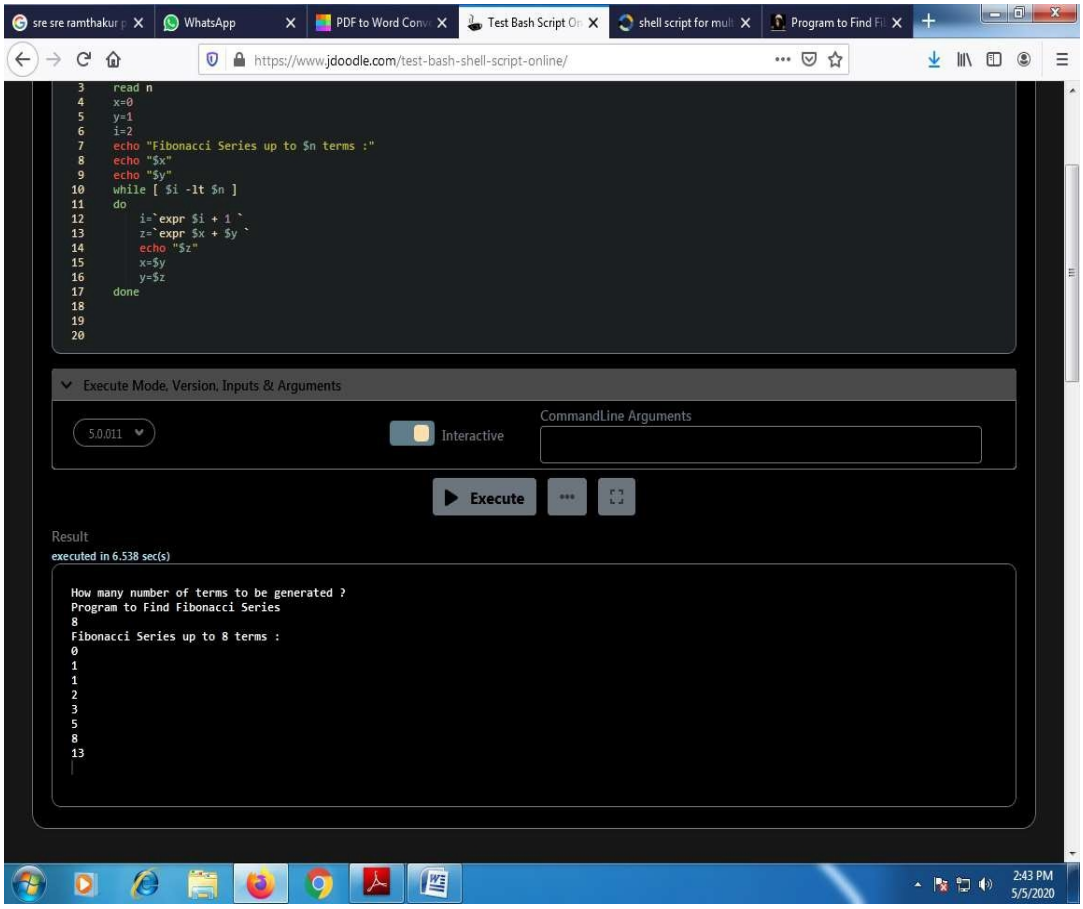
Output:



Write a shell program which calculate Fibonacci series up to a specific range.

```
echo "Enter The Range"
read x
n=$x
a=0
b=1
c=0
while [ $c -le $n ]
do
echo "$c"
a=$b
b=$c
c=$( expr $a + $b )
done
```

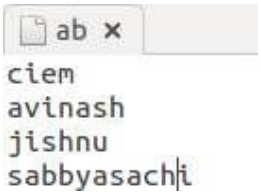
Output:


sre sre ramthakur
WhatsApp
PDF to Word Conv
Test Bash Script On
shell script for mult
Program to Find Fi
https://www.jdoodle.com/test-bash-shell-script-online/
read n
x=0
y=1
i=2
echo "Fibonacci Series up to $n terms :"
echo "$x"
echo "$y"
while [$i -lt $n]
do
i=`expr $i + 1 `
z=`expr $x + $y `
echo "$z"
x=$y
y=$z
done
Execute Mode, Version, Inputs & Arguments
5.0.011
Interactive
CommandLine Arguments
Execute
Result
executed in 6.538 sec(s)
How many number of terms to be generated ?
Program to Find Fibonacci Series
8
Fibonacci Series up to 8 terms :
0
1
1
2
3
5
8
13
2:43 PM
5/5/2020

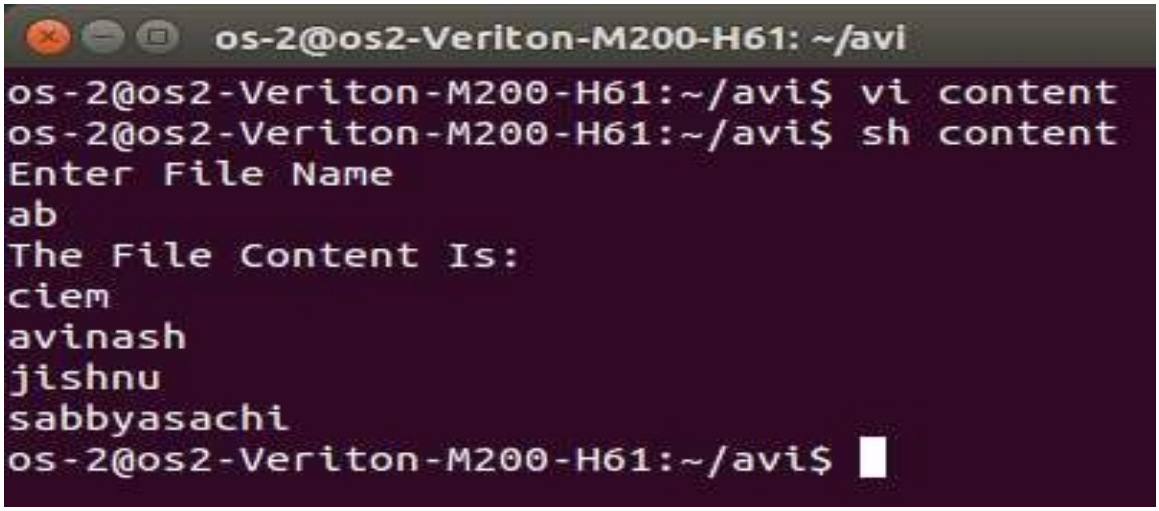
Write a shell program which display the content of a file without using the ‘cat’ command.

```
echo "Enter File Name"
read f
exec<$f
echo “The File Content Is:”
while read x
do
echo "$x"
done
```

Output:



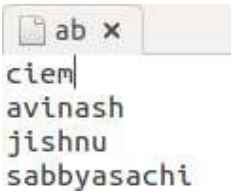
File whose content is to showed



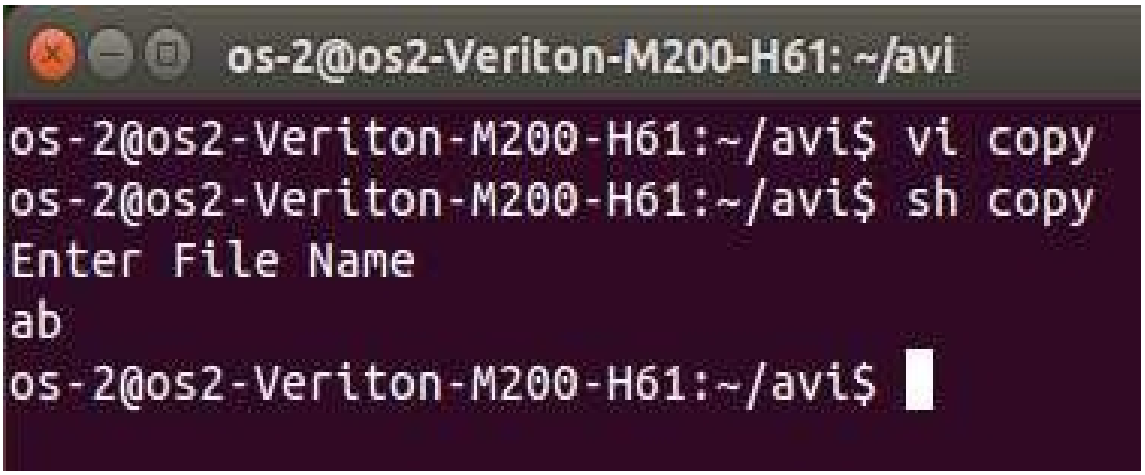
Write a shell program which copy the content of a file into another file.

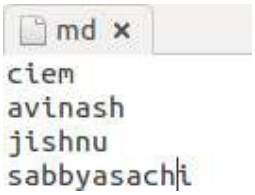
```
echo "Enter File Name"
read f
exec<$f
exec>md
while read x
do
echo "$x"
done
exec>
echo “Operation Copy Successful”
```

Output:



Original File Content





File in which the content is copied

Write a shell program to count total no of lines in a file without using 'wc' command.

```
echo "Enter File Name"
read f
exec<$f
c=0
while read x
do
c=$( expr $c + 1 )
done
echo "Number Of lines In the File Is: $c"
```

Output:

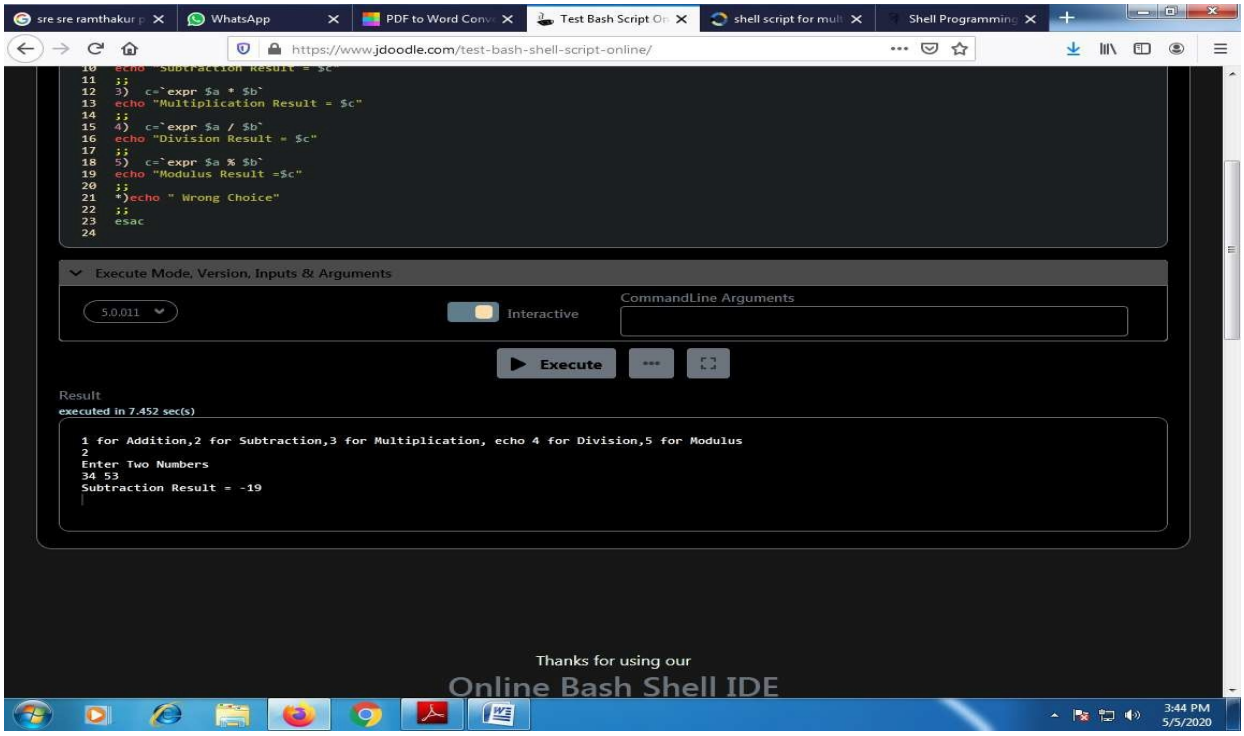
Write a menu driven shell program which perform the following:

- **Addition**
- **Subtraction**
- **Multiplication**
- **Division**
- **Modulus**

```
echo "1 for Addition,2 for Subtraction,3 for Multiplication,"
echo "4 for Division,5 for Modulus"
read ch
echo "Enter Two Numbers"
```

```
read a b
case $ch in
1 c=`echo $a+$b|bc`
  echo "Addition Result =$c"
          ;;
2 c=`echo $a-$b|bc`
  echo "Subtraction Result = $c"
  ;;
3 c=`echo $a*$b|bc`
  echo "Multiplication Result = $c"
  ;;
4 c=`echo $a/$b|bc`
  echo "Division Result = $c"
  ;;
5 c=`echo $a%$b|bc`
  echo "Modulus Result =$c"
  ;;
*)echo " Wrong Choice"
  ;;
esac
```

Output:

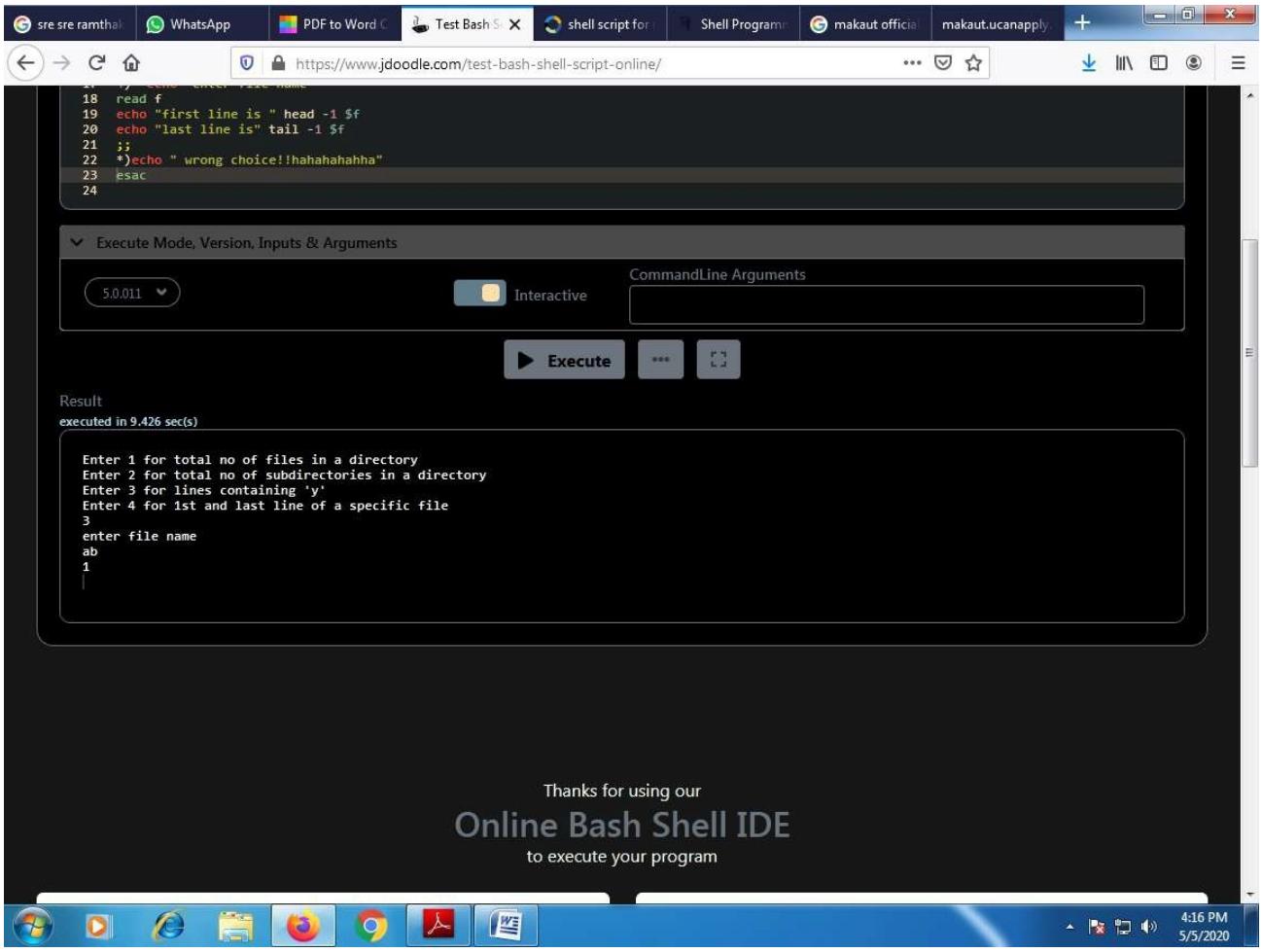


Write a menu driven shell program which perform the following:

- **Count total no of files under a directory.**
- **Count total no of subdirectory under a directory.**
- **Count total no of line(s) which contains a specific 'Y'.**
- **Display first and last line of a specific file.**

```
echo "Enter 1 for total no of files in a directory"
echo "Enter 2 for total no of subdirectories in a directory"
echo "Enter 3 for lines containing 'y'"
echo "Enter 4 for 1st and last line of a specific file"
read ch
case $ch in
1c=$(ls -l|cut -c1|grep "-"|wc -l)
  echo "No of files= $c"
  ;;
2c=$(ls -l|cut -c1|grep "d"|wc -l)
  echo "No Of  subdirectories=$c"
  ;;
3echo "enter file name"
  read f
echo "No of lines containing 'n'"
grep n $f|wc -l
  ;;
4  echo "enter file name"
read f
echo "first line is "
head -1 $f
echo "last line is"
tail -1 $f
 ;;
*)echo " wrong choice!!hahahahahha"
esac
```

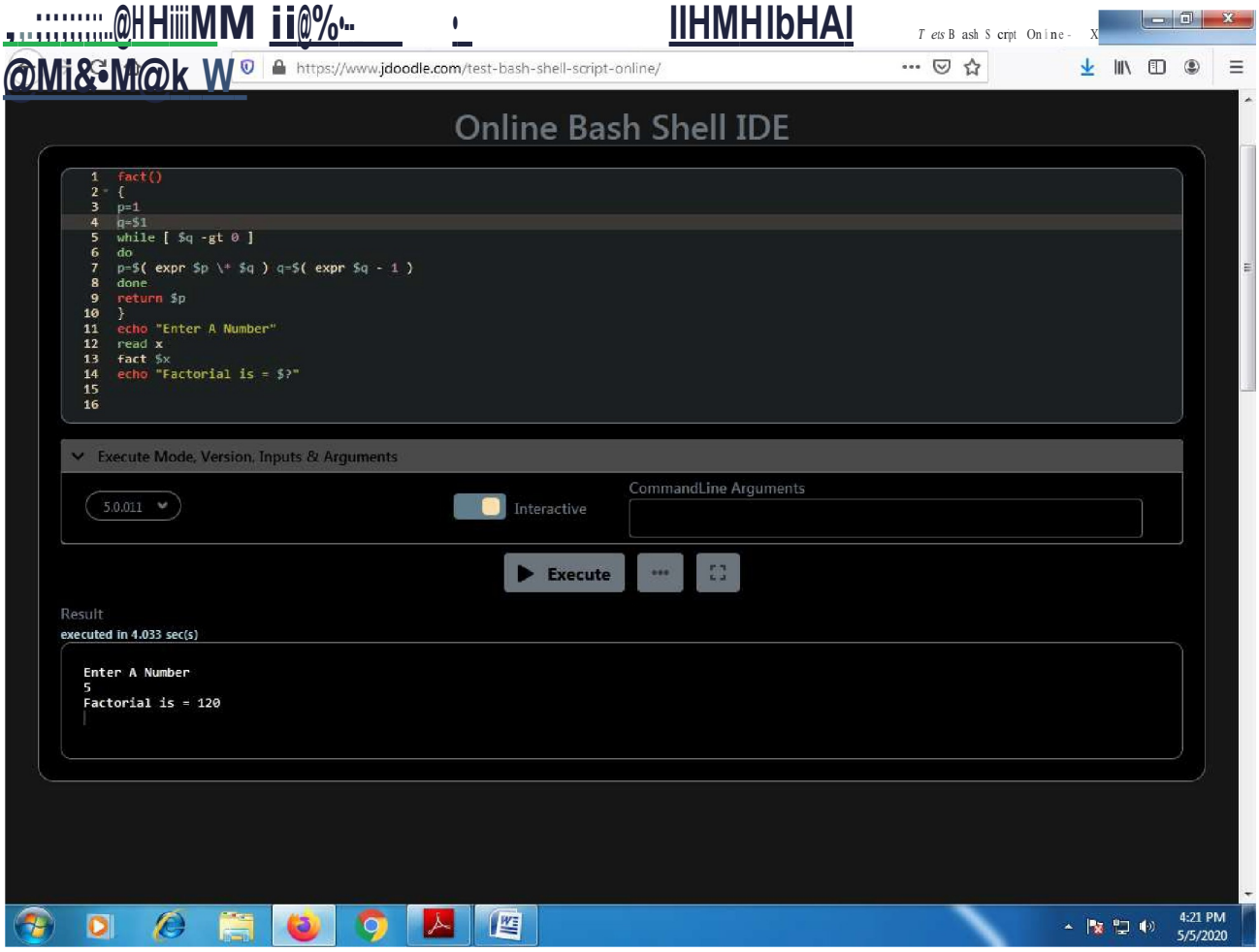
Output:


https://www.jdoodle.com/test-bash-shell-script-online/
18 read f
19 echo "first line is " head -1 $f
20 echo "last line is" tail -1 $f
21 ;;
22 *)echo " wrong choice!!hahahahahha"
23 esac
24
Execute Mode, Version, Inputs & Arguments
5.0.011
Interactive
CommandLine Arguments
Execute
Result
executed in 9.426 sec(s)
Enter 1 for total no of files in a directory
Enter 2 for total no of subdirectories in a directory
Enter 3 for lines containing 'y'
Enter 4 for 1st and last line of a specific file
3
enter file name
ab
1
Thanks for using our
Online Bash Shell IDE
to execute your program
4:16 PM
5/5/2020

Write a shell program which calculate factorial of a number using function.

```
fact()
{
p=1
q=$1
while [ $q -gt 0 ]
do
p=$( expr $p \* $q )
q=$( expr $q - 1 )
done
return $p
}
echo "Enter A Number"
read x
fact $x
echo "Factorial is = $?"
```

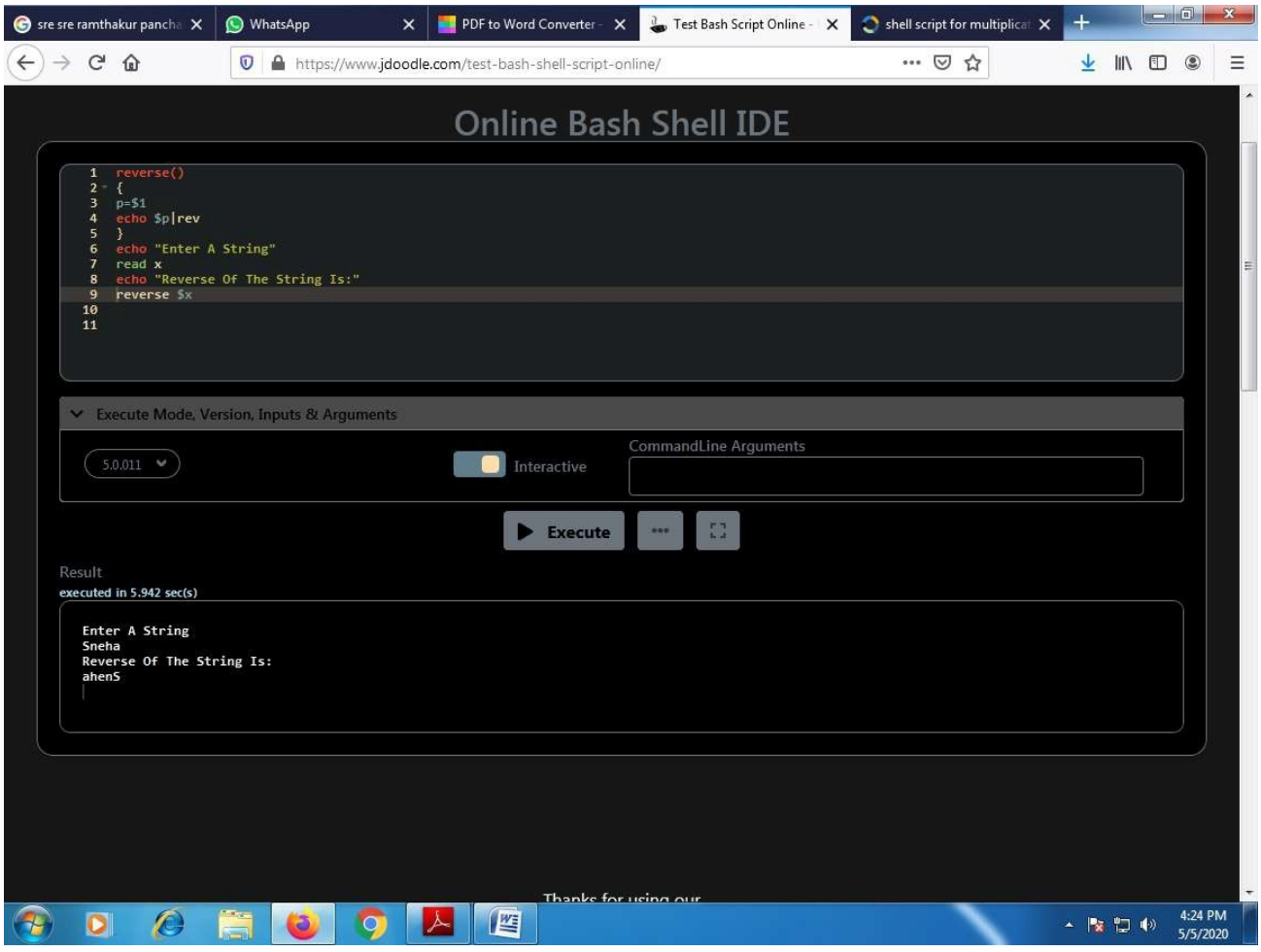
Output:


https://www.jdoodle.com/test-bash-shell-script-online/
Online Bash Shell IDE
1 fact()
2 {
3 p=1
4 q=$1
5 while [$q -gt 0]
6 do
7 p=$(expr $p * $q) q=$(expr $q - 1)
8 done
9 return $p
10 }
11 echo "Enter A Number"
12 read x
13 fact $x
14 echo "Factorial is = $?"
15
16
Execute Mode, Version, Inputs & Arguments
5.0.011
Interactive
CommandLine Arguments
Execute
Result
executed in 4.033 sec(s)
Enter A Number
5
Factorial is = 120
4:21 PM
5/5/2020

Write a shell program which reverse a particular string using function.

```
reverse()
{
p=$1
echo $p|rev
}
echo "Enter A String"
read x
echo "Reverse Of The String Is:"
reverse $x
```

Output:



3 Process

Definition: A **process** is a program in execution in memory or in other words, an instance of a program in memory. Any program executed creates a process. A program can be a command, a shell script, or any binary executable or any application. However, not all commands end up in creating process, there are some exceptions. Similar to how a file created has properties associated with it, a process also has lots of properties associated to it.

Process attributes:

A process has some properties associated to it:

PID: Process-Id. Every process created in Unix/Linux has an identification number associated to it which is called the process-id. This process id is used by the kernel to identify the process similar to how the inode number is used for file identification. The PID is unique for a process at any given point of time. However, it gets recycled.

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

TTY: Terminal to which the process is associated to. Every command is run from a terminal which is associated to the process. However, not all processes are associated to a terminal. There are some processes which do not belong to any terminal. These are called daemons.

UID: User Id- The user to whom the process belongs to. And the user who is the owner of the process can only kill the process (Of course, root user can kill any process). When a process tries to access files, the accessibility depends on the permissions the process owner has on those files.

Process States

New: When a process is created, then it is in new state.

Ready State: Process is in ready to run.

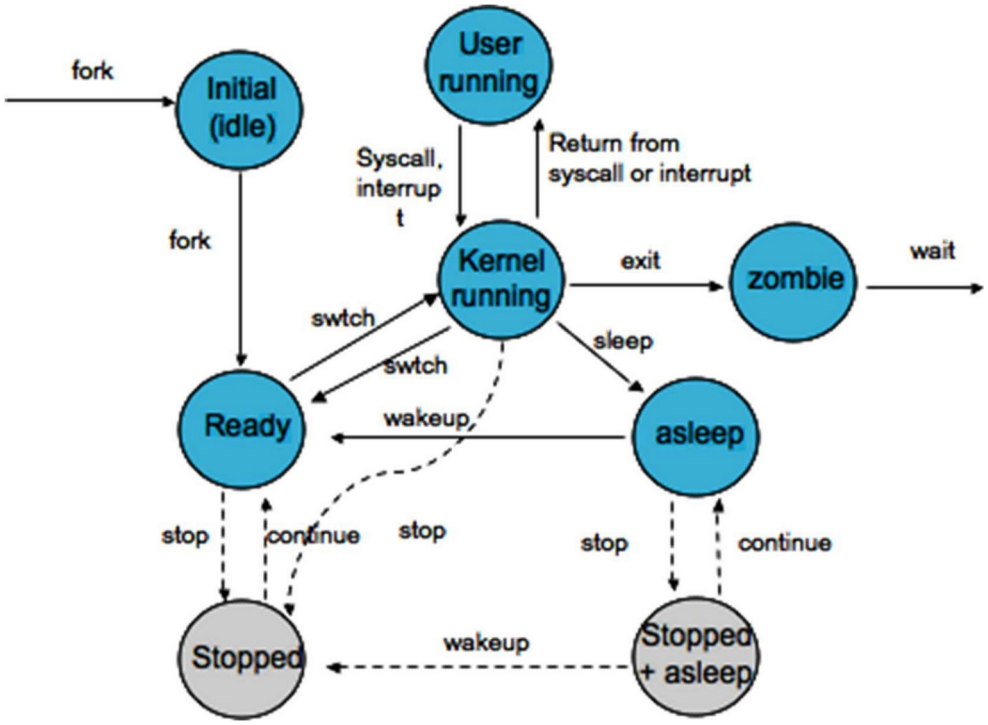
Running: Process is running.

Interruptible: A blocked state of a process and waiting for an event or signal from another process

Uninterruptible: A blocked state. Process waits for a hardware condition and cannot handle any signal

Stopped: Process is stopped or halted and can be restarted by some other process

Zombie: Process terminated, but information is still there in the process table.



Internal Functionality:

When you execute a program on your UNIX system, the system creates a special environment for that program. This environment contains everything needed for the system to run the program as if no other program were running on the system.

Whenever you issue a command in UNIX, it creates, or starts, a new process. When you tried out the **ls** command to list directory contents, you started a process. A process, in simple terms, is an instance of a running program.

The operating system tracks processes through a five digit ID number known as the **pid** or process ID. Each process in the system has a unique pid.

Pids eventually repeat because all the possible numbers are used up and the next pid rolls or starts over. At any one time, no two processes with the same pid exist in the system because it is the pid that UNIX uses to track each process.

Starting a Process:

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

i Foreground Processes

ii Background Processes

Foreground Processes

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

You can see this happen with the ls command. If I want to list all the files in my current directory, I can use the following command −

$ls ch*.doc

This would display all the files whose name start with ch and ends with .doc −

```
ch01-1.doc   ch010.doc   ch02.doc   ch03-2.doc
ch04-1.doc   ch040.doc   ch05.doc   ch06-2.doc
ch01-2.doc  ch02-1.doc
```

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

While a program is running in foreground and taking much time, we cannot run any other commands (start any other processes) because prompt would not be available until program finishes its processing and comes out.

Background Processes

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

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

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

```
$ls ch*.doc &
```

This would also display all the files whose name start with ch and ends with .doc −

```
ch01-1.doc   ch010.doc   ch02.doc   ch03-2.doc
ch04-1.doc   ch040.doc   ch05.doc   ch06-2.doc
ch01-2.doc  ch02-1.doc
```

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

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

If you press the Enter key now, you see the following −

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

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

Different system calls related to process creation:

Fork()

NAME
fork - create a child process
SYNOPSIS
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);

DESCRIPTION
fork() creates a child process that differs from the parent process only in its PID and PPID, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited.
Under Linux, **fork**() is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

RETURN VALUE
On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, a -1 will be returned in the parent's context, no child process will be created, and *errno* will be set appropriately.

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

```
$ps
PID      TTY     TIME       CMD
18358    ttyp3   00:00:00   sh
18361    ttyp3   00:01:31   abiword
18789    ttyp3   00:00:00   ps
```

One of the most commonly used flags for ps is the **-f** (f for full) option, which provides more information as shown in the following example −

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

Here is the description of all the fields displayed by ps -f command −

Column	Description
UID	User ID that this process belongs to (the person running it).
PID	Process ID.
PPID	Parent process ID (the ID of the process that started it).
C	CPU utilization of process.
STIME	Process start time.
TTY	Terminal type associated with the process
TIME	CPU time taken by the process.
CMD	The command that started this process.

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

Option	Description
-a	Shows information about all users
-x	Shows information about processes without terminals.
-u	Shows additional information like -f option.
-e	Display extended information.

Batch processing

Batch processing is done by using **batch** keyword.

batch read commands from standard input or a specified file which are to be executed at a later time.

batch, commands are read from standard input or the file specified with the -f option and executed.

Stopping Processes

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

If a process is running in background mode then first you would need to get its Job ID using **ps** command and after that you can use **kill** command to kill the process as follows −

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

Killing process:

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

```
$kill -9 6738
Terminated
```

3.2.5) Types Of Processes:

Parent and Child Processes

Each unix process has two ID numbers assigned to it: Process ID (pid) and Parent process ID (ppid). Each user process in the system has a parent process.

Most of the commands that you run have the shell as their parent. Check ps -f example where this command listed both process ID and parent process ID.

Zombie and Orphan Processes

Normally, when a child process is killed, the parent process is told via a SIGCHLD signal. Then the parent can do some other task or restart a new child as needed. However, sometimes the parent process is killed before its child is killed. In this case, the "parent of all processes," **init** process, becomes the new PPID (parent process ID). Sometime these processes are called orphan process.

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

Daemon Processes

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

A daemon process has no controlling terminal. It cannot open /dev/tty. If you do a "ps -ef" and look at the tty field, all daemons will have a ? for the tty.

More clearly, a daemon is just a process that runs in the background, usually waiting for something to happen that it is capable of working with, like a printer daemon is waiting for print commands.

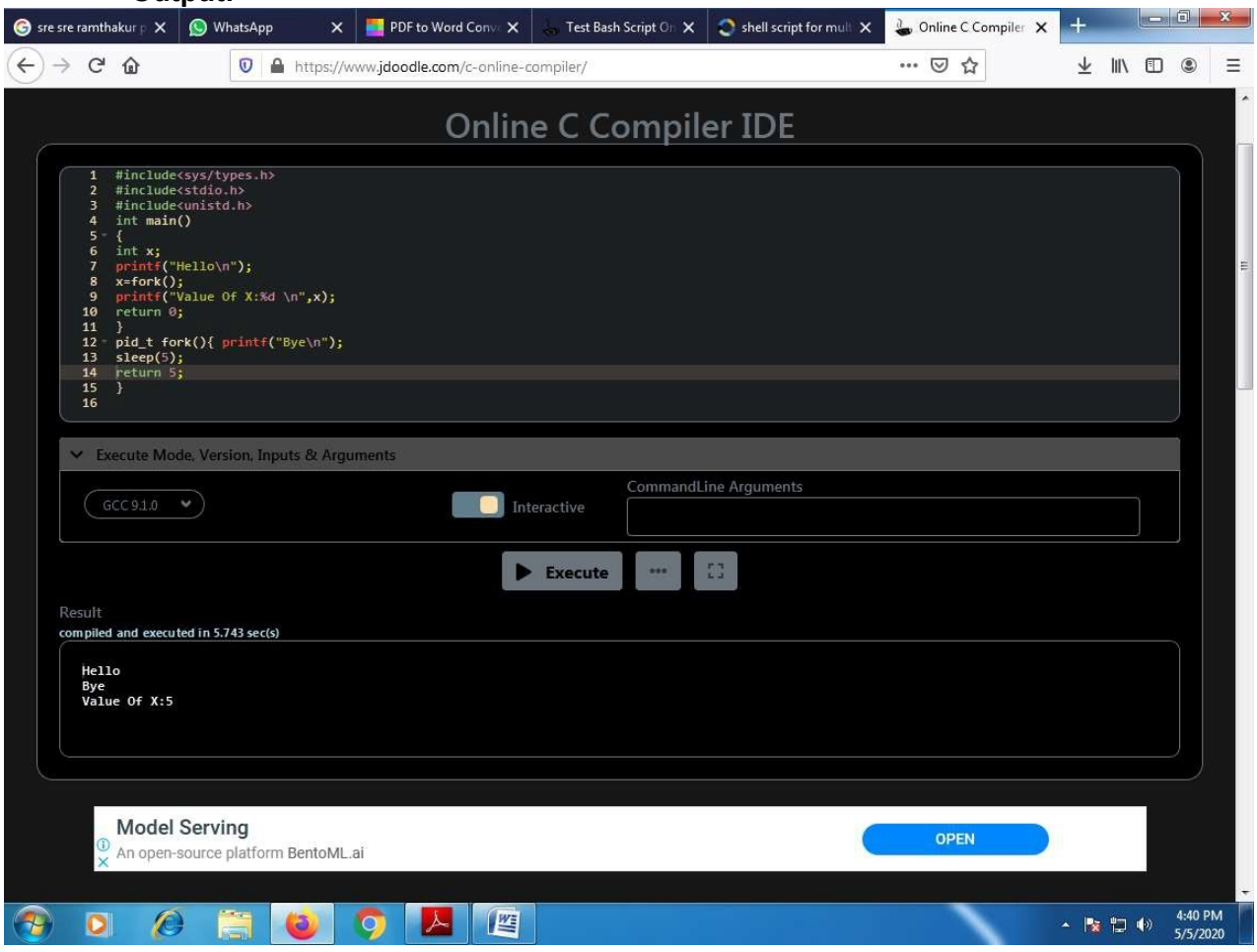
If you have a program which needs to do long processing then its worth to make it a daemon and run it in background.

Program:

Write a shell program which implement process and perform a task of printing word-“BYE”.

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
int main()
{
int x; printf("Hello\
n"); x=fork();
printf("Value Of X:%d \n",x);
return 0;
}
pid_t fork()
{ printf("Bye\
n"); sleep(5);
return 5;
}
```

Output:



4 Semaphore

Introduction:

A semaphore is a variable or abstract data type that is used for controlling access, by multiple processes, to a common resource in a concurrent system such as a multiprogramming operating system.

A useful way to think of a semaphore as used in the real-world systems is as a record of how many units of a particular resource are available, coupled with operations to safely (i.e., without race conditions) adjust that record as units are required or become free, and, if necessary, wait until a unit of the resource becomes available. Semaphores are a useful tool in the prevention of race conditions; however, their use is by no means a guarantee that a program is free from these problems. Semaphores which allow an arbitrary resource count are called counting semaphores, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores.

Important observations:

When used to control access to a pool of resources, a semaphore tracks only how many resources are free, it does not keep track of which of the resources are free. Some other mechanism (possibly involving more semaphores) may be required to select a particular free resource.

The paradigm is especially powerful because the semaphore count may serve as a useful trigger for a number of different actions. The librarian above may turn the lights off in the study hall when there are no students remaining, or may place a sign that says the rooms are very busy when most of the rooms are occupied.

The success of the protocol requires applications follow it correctly. Fairness and safety are likely to be compromised (which practically means a program may behave slowly, act erratically, hang or crash) if even a single process acts incorrectly. This includes:

i Requesting a resource and forgetting to release it;
ii Releasing a resource that was never requested;
iii Holding a resource for a long time without needing it;
iv Using a resource without requesting it first (or after releasing it).

Even if all processes follow these rules, multi-resource deadlock may still occur when there are different resources managed by different semaphores and when processes need to use more than one resource at a time, as illustrated by the dining philosophers problem.

Functions related to semaphore:

semget()

The **semget()** function creates a new semaphore or obtains the semaphore key of an existing semaphore. The first parameter, *key*, is an integer used to allow unrelated processes to access the same semaphore! There is a special semaphore *key* value **IPC_PRIVATE**, that only the semaphore creator can access it.

The *nsems* parameter is the number of semaphores that the semaphore set should contain.

The *flag* is a set of flages. The lower nine bits are the permissions for the semaphore, which behave like file permissions. If *flag* specifies both IPC_CREAT and IPC_EXCL and a semaphore set already exists for *key*, then semget() fails with errno set to EEXIST. (This is analogous to the effect of the combination O_CREAT | O_EXCL for open(2).) Upon creation, the least significant 9 bits of the argument semflg define the permissions (for owner, group and others) for the semaphore set. These bits have the same format, and the same meaning, as the mode argument of open(2)

semop()

A semaphore is represented by an anonymous structure including the following members:

```
unsigned short semval; // semaphore value
unsigned short semzcnt; // # waiting for zero
unsigned short semncnt; // # waiting for
increase pid_t sempid; // process that did last
op
```

The **semop()** function is used for changing the value of a semaphore. The first parameter is the semaphore identifier, as returned from **semget()**. The second parameter *array* is a pointer to an array of structures, each of which has at least the following members:

```
struct sembuf {
  short  sem_num;  //semaphore  number
  short  sem_op;  //semaphore  operation
  short sem_flg; //operation flags
}
```

semctl()

The function **semctl()** performs the control operation specified by *cmd* on the semaphore set identified by *semid*, or on the *semnum*-th semaphore of that set. (Semaphores are numbered starting at 0.) This function has three or four arguments. When there are four, the call is **semctl(semid,semnum,cmd,arg);** where the fourth argument *arg* has a type union semun defined as follows:

```
union semun {
  int val;               /* value for SETVAL */
  struct semid_ds *buf;    /* buffer for IPC_STAT, IPC_SET
  */ unsigned short *array;       /* array for GETALL, SETALL
  */
                        /* Linux specific part: */
  struct seminfo *__buf;    /* buffer for IPC_INFO */
};
```

SEMAPHORE_P()

Purpose

semaphore_p issues a P on a semaphore.

Synopsis

#include <types.h>
#include <stub.h>
int *semaphore_p*(Capability ***semaphore**);

Parameters

semaphore: a capability for the semaphore to apply P.

Description

semaphore_p issues a P operation on the semaphore designated by **semaphore**. The P operation decrements the semaphore's value and, if it becomes to 0, blocks the calling thread.
A thread blocked on a semaphore is releases when the semaphore's value returns to a positive value because some thread issued a V operation on it.

Return Values

On success, *semaphore_p* returns 0 and semaphore's value is decremented.
On fail, *semaphore_p* returns a negative value indicating one of the following errors:
Invalid capability: the capability in **semaphore** is not valid;
Insufficient rights: the capability in **semaphore** does not allow P.

SEMAPHORE_V()

Purpose

semaphore_v issues a V on a semaphore.

Synopsis

#include <types.h>
#include <stub.h>
int *semaphore_v*(Capability ***semaphore**);

Parameters

semaphore: a capability for the semaphore to apply V.

Description

semaphore_v issues a V operation on the semaphore designated by **semaphore**. The V operation increments the semaphore's value, If there are thread block on the specified semaphore, the first blocked is released.

Return Values

On success, *semaphore_v* returns 0 and semaphore's value is incremented.
On fail, *semaphore_v* returns a negative value indicating one of the following errors:
Invalid capability: the capability in **semaphore** is not valid;
Insufficient rights: the capability in **semaphore** does not allow V.

set_semvalue()

The function set_semvalue initializes the semaphore using the SETVAL command in a semctl call.You need to do this before you can use the semaphore:

```
static int set_semvalue (int sem_id)
{
                        union semun sem_union;
                        sem_union.val = 1;
                        if (semctl (sem_id, 0, SETVAL, sem_union.val) == -1)
(0);
                        return return (1);
}
```

del_semvalue()

After the critical section,you call semaphore_v,setting the semaphore as available, before going through the for loop again after a random wait. After the loop,the call to del_semvalue is made to clean up the code:

```
static void del_semvalue (int sem_id)
{
                        union semun sem_union;
                        if (semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
                                fprintf (stderr, "Failed to delete semaphore\n");
}
```

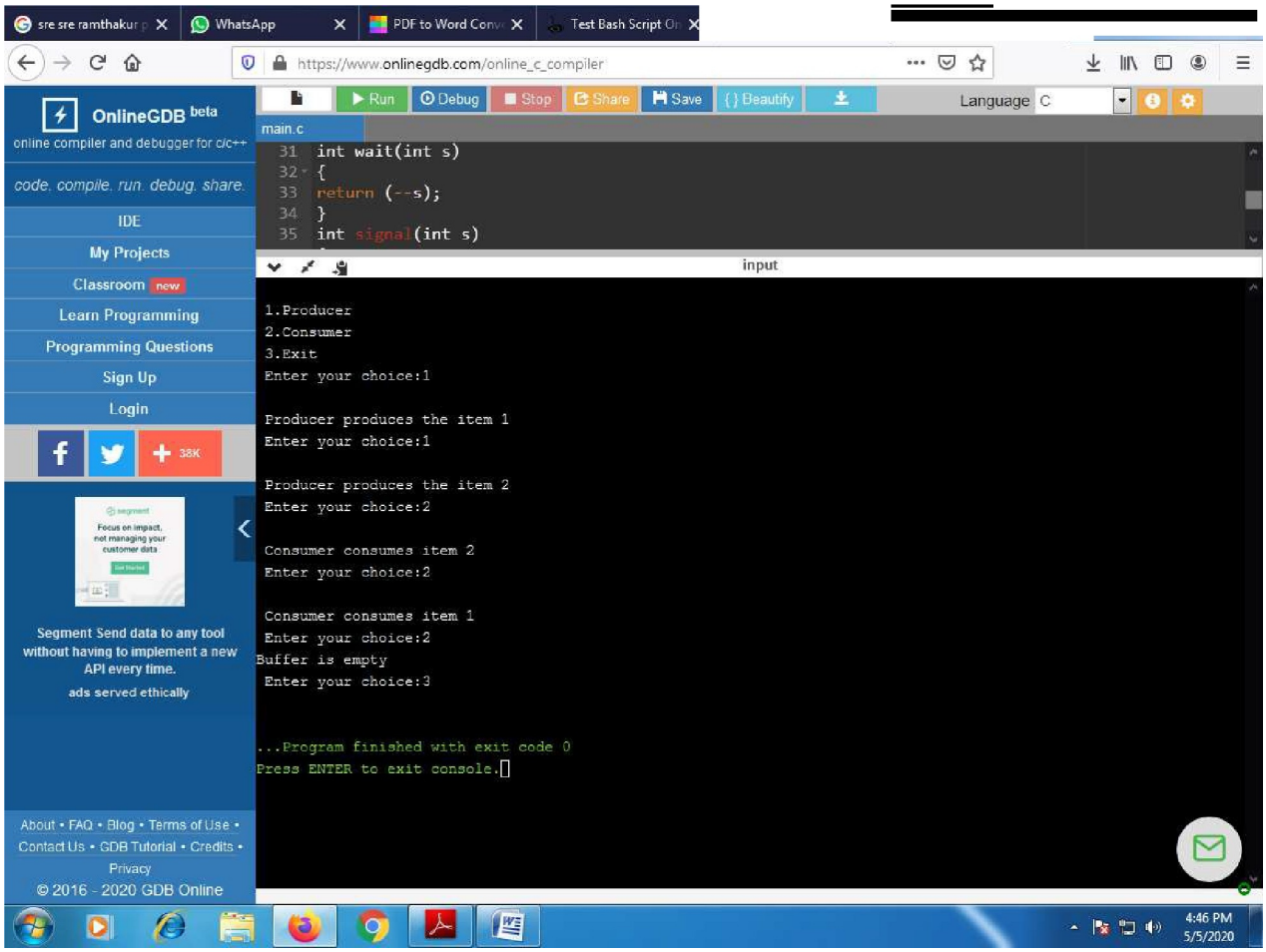
Program:

To write a LINUX/UNIX C Program for the Implementation of Producer Consumer Algorithm using Semaphore.

```
#include<stdio.h>
#include<stdlib.h>
int mutex=1,full=0,empty=3,x=0;
main()
{
  int n;
  void producer();
  void consumer();
  int wait(int);
  int signal(int);
  printf("\n 1.Producer \n 2.Consumer \n
  3.Exit"); while(1)
  {
    printf("\n Enter your choice:");
    scanf("%d",&n);
    switch(n)
    {
      case 1:
            if((mutex==1)&&(empty!=0))
              producer();
            else
              printf("Buffer is full");
   break;
      case 2:
         if((mutex==1)&&(full!=0))
  consumer();
  else
     printf("Buffer is empty");
    break;
      case 3:
  exit(0);
  break;
    }
  }
}
int wait(int s)
{
  return (--s);
}
int signal(int s)
{
```

```
  return(++s);
}
void producer()
{
  mutex=wait(mutex);
  full=signal(full);
  empty=wait(empty);
  x++;
  printf("\n Producer produces the item
  %d",x); mutex=signal(mutex);
}
void consumer()
{
  mutex=wait(mutex);
  full=wait(full);
  empty=signal(empty);
  printf("\n Consumer consumes item
  %d",x); x--;
  mutex=signal(mutex);
}
```

Output:

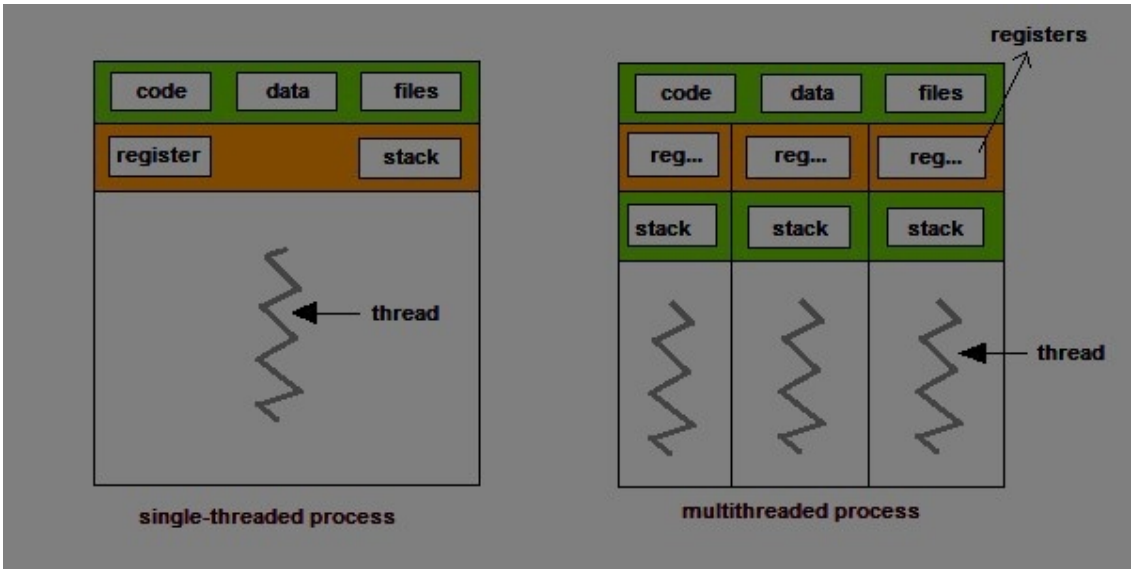

sre sre ramthakur
WhatsApp
PDF to Word Conv
Test Bash Script On
https://www.onlinegdb.com/online_c_compiler
OnlineGDB beta
online compiler and debugger for c/c++
code. compile. run. debug. share.
IDE
My Projects
Classroom new
Learn Programming
Programming Questions
Sign Up
Login
38K
Segment Send data to any tool without having to implement a new API every time.
ads served ethically
About • FAQ • Blog • Terms of Use • Contact Us • GDB Tutorial • Credits • Privacy
© 2016 - 2020 GDB Online
Run
Debug
Stop
Share
Save
{ } Beautify
Language C
main.c
31 int wait(int s)
32 {
33 return (--s);
34 }
35 int signal(int s)
input
1.Producer
2.Consumer
3.Exit
Enter your choice:1
Producer produces the item 1
Enter your choice:1
Producer produces the item 2
Enter your choice:2
Consumer consumes item 2
Enter your choice:2
Consumer consumes item 1
Enter your choice:2
Buffer is empty
Enter your choice:3
...Program finished with exit code 0
Press ENTER to exit console.
4:46 PM
5/5/2020

5 Thread

Introduction:

Thread is an execution unit which consists of its own program counter, a stack, and a set of registers. Threads are also known as Lightweight processes. Threads are popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel.

As each thread has its own independent resource for process execution, multpile processes can be executed parallely by increasing number of threads.



Types of thread:

There are two types of threads:

User Threads

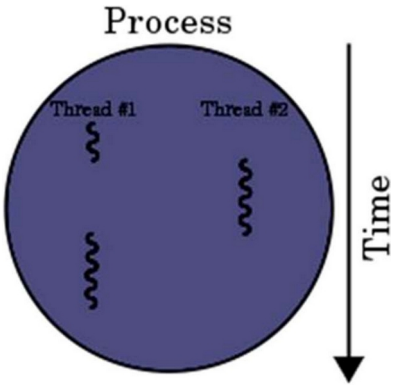
Kernel Threads

User threads, are above the kernel and without kernel support. These are the threads that application programmers use in their programs.

Kernel threads are supported within the kernel of the OS itself. All modern OSs support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

In computer science, a *thread* of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.[1] The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently (one starting before others finish) and share resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its instructions (executable code) and its context (the values of its variables at any given time).

On one processor, multithreading is generally implemented by time slicing (as in multitasking), and the central processing unit (CPU) switches between different *software threads*. This context switching generally happens often enough that users perceive the threads or tasks as running at the same time (in parallel). On a multiprocessor or multi-core system, multiple threads can be executed in parallel (at the same instant), with every processor or core executing a separate thread simultaneously; on a processor or core with *hardware threads*, separate software threads can also be executed concurrently by separate hardware threads.



Different thread functions:

pthread_create():

NAME

pthread_create - thread creation

SYNOPSIS

#include <pthread.h>

int pthread_create(pthread_t *_thread_, const pthread_attr_t *_attr_,
void *(*_start_routine_)(void*), void *_arg_);

DESCRIPTION

The *pthread_create()* function is used to create a new thread, with attributes specified by *attr*, within a process. If *attr* is NULL, the default attributes are used. If the attributes specified by *attr* are modified later, the thread's attributes are not

affected. Upon successful completion, *pthread_create()* stores the ID of the created thread in the location referenced by *thread*.
The thread is created executing *start_routine* with *arg* as its sole argument. If the *start_routine* returns, the effect is as if there was an implicit call to *pthread_exit()* using the return value of *start_routine* as the exit status. Note that the thread in which *main()* was originally invoked differs from this. When it returns from *main()*, the effect is as if there was an implicit call to *exit()* using the return value of *main()* as the exit status.
The signal state of the new thread is initialised as follows:
The signal mask is inherited from the creating thread.
The set of signals pending for the new thread is empty.

If *pthread_create()* fails, no new thread is created and the contents of the location referenced by *thread* are undefined.

RETURN VALUE
If successful, the *pthread_create()* function returns zero. Otherwise, an error number is returned to indicate the error.

pthread_join():

NAME
pthread_join - wait for thread termination

SYNOPSIS
#include <pthread.h>
int pthread_join(pthread_t *thread*, void ***value_ptr*);

DESCRIPTION
The *pthread_join*() function shall suspend execution of the calling thread until the target *thread* terminates, unless the target *thread* has already terminated. On return from a successful *pthread_join*() call with a non-NULL *value_ptr* argument, the value passed to *pthread_exit*() by the terminating thread shall be made available in the location referenced by *value_ptr*. When a *pthread_join*() returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to *pthread_join*() specifying the same target thread are undefined. If the thread calling *pthread_join*() is canceled, then the target thread shall not be detached.
It is unspecified whether a thread that has exited but remains unjoined counts against {PTHREAD_THREADS_MAX}.
The behavior is undefined if the value specified by the *thread* argument to *pthread_join*() does not refer to a joinable thread.
The behavior is undefined if the value specified by the *thread* argument to *pthread_join*() refers to the calling thread.

RETURN VALUE
If successful, the *pthread_join*() function shall return zero; otherwise, an error number shall be returned to indicate the error.

pthread_exit():

NAME
pthread_exit - thread termination

SYNOPSIS

#include <pthread.h>
void pthread_exit(void **value_ptr*);

DESCRIPTION

The *pthread_exit()* function terminates the calling thread and makes the value *value_ptr* available to any successful join with the terminating thread. Any cancellation cleanup handlers that have been pushed and not yet popped are popped in the reverse order that they were pushed and then executed. After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data, appropriate destructor functions will be called in an unspecified order. Thread termination does not release any application visible process resources, including, but not limited to, mutexes and file descriptors, nor does it perform any process level cleanup actions, including, but not limited to, calling any *atexit()* routines that may exist.
An implicit call to *pthread_exit()* is made when a thread other than the thread in which *main()* was first invoked returns from the start routine that was used to create it. The function's return value serves as the thread's exit status.
The behaviour of *pthread_exit()* is undefined if called from a cancellation cleanup handler or destructor function that was invoked as a result of either an implicit or explicit call to *pthread_exit()*.
After a thread has terminated, the result of access to local (auto) variables of the thread is undefined. Thus, references to local variables of the exiting thread should not be used for the *pthread_exit() value_ptr* parameter value.
The process exits with an exit status of 0 after the last thread has been terminated. The behaviour is as if the implementation called *exit()* with a zero argument at thread termination time.

RETURN VALUE

The *pthread_exit()* function cannot return to its caller.

Program:

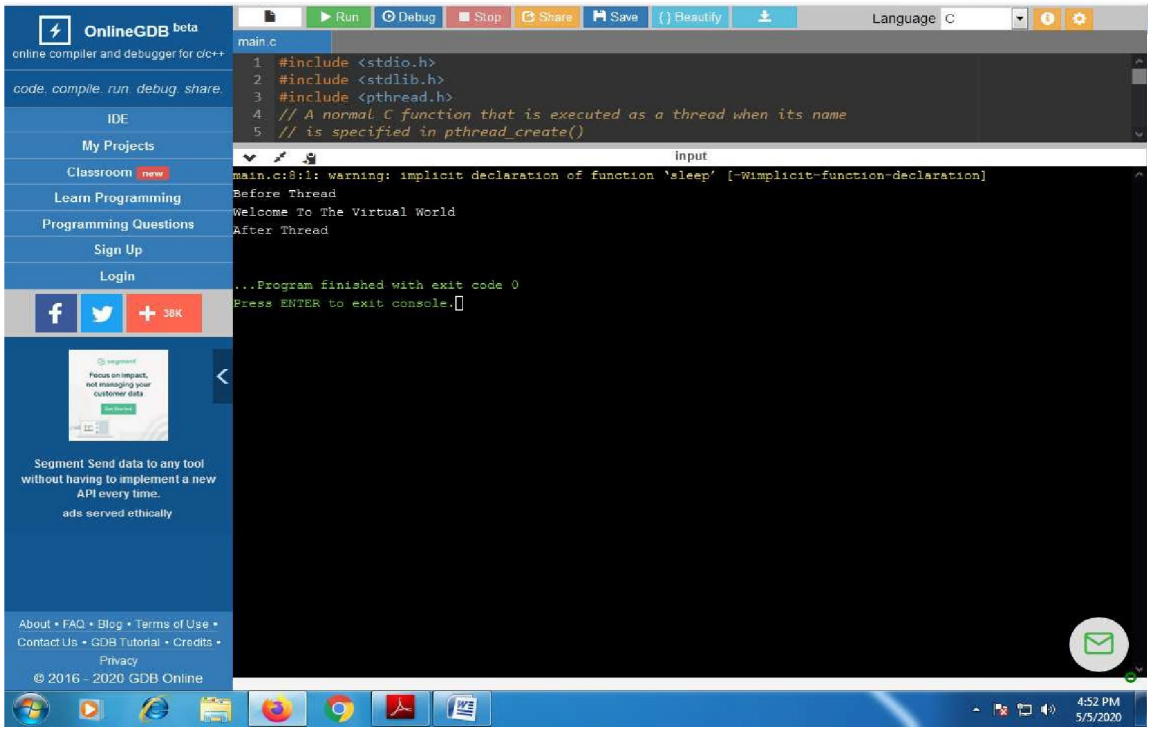
Write a Unix C program to demonstrate use of pthread basic functions.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// A normal C function that is executed as a thread when its name
// is specified in pthread_create()
void *myThreadFun(void *vargp)
{
    sleep(1);
    printf("Welcome To The Virtual World \n");
    return NULL;
}
int main()
{
    pthread_t tid;
    printf("Before  Thread\n");
    pthread_create(&tid, NULL, myThreadFun, NULL);
    pthread_join(tid, NULL);
    printf("After Thread\n");
    exit(0);
}
```

Output:

( ) C' 1'.at (D ii ht1ps :// www .on linegdb .com / online_c_compiler 0 ::kIIIN:D®



6 Inter process Communication:

Introduction:

Interprocess communication (IPC) is a set of programming interfaces that allow a programmer to coordinate activities among different program processes that can run concurrently in an operating system. This allows a program to handle many user requests at the same time. Since even a single user request may result in multiple processes running in the operating system on the user's behalf, the processes need to communicate with each other. The IPC interfaces make this possible. Each IPC method has its own advantages and limitations so it is not unusual for a single program to use all of the IPC methods.

IPC methods include pipes and named pipes; message queueing; semaphores; shared memory; and sockets.

Interprocess Communication Protocol:

Interprocess Communication Protocol is a set of rules which creates an environment where process can co-operate with each other.

Interprocess Communication Protocol can be achieved by 3 different ways:

i Via shared resource
ii Via message passing
iii Via monitor

Pipe

A pipe is a technique for passing information from one program process to another. Unlike other forms of interprocess communication (IPC), a pipe is one-way communication only. Basically, a pipe passes a parameter such as the output of one process to another process which accepts it as input. The system temporarily holds the piped information until it is read by the receiving process.

Pipe generally transfer an output of a command as a input of a second command. A pipe is specified in a command line as a simple vertical bar (|) between two command sequences. The output or result of the first command sequence is used as the input to the second command sequence.

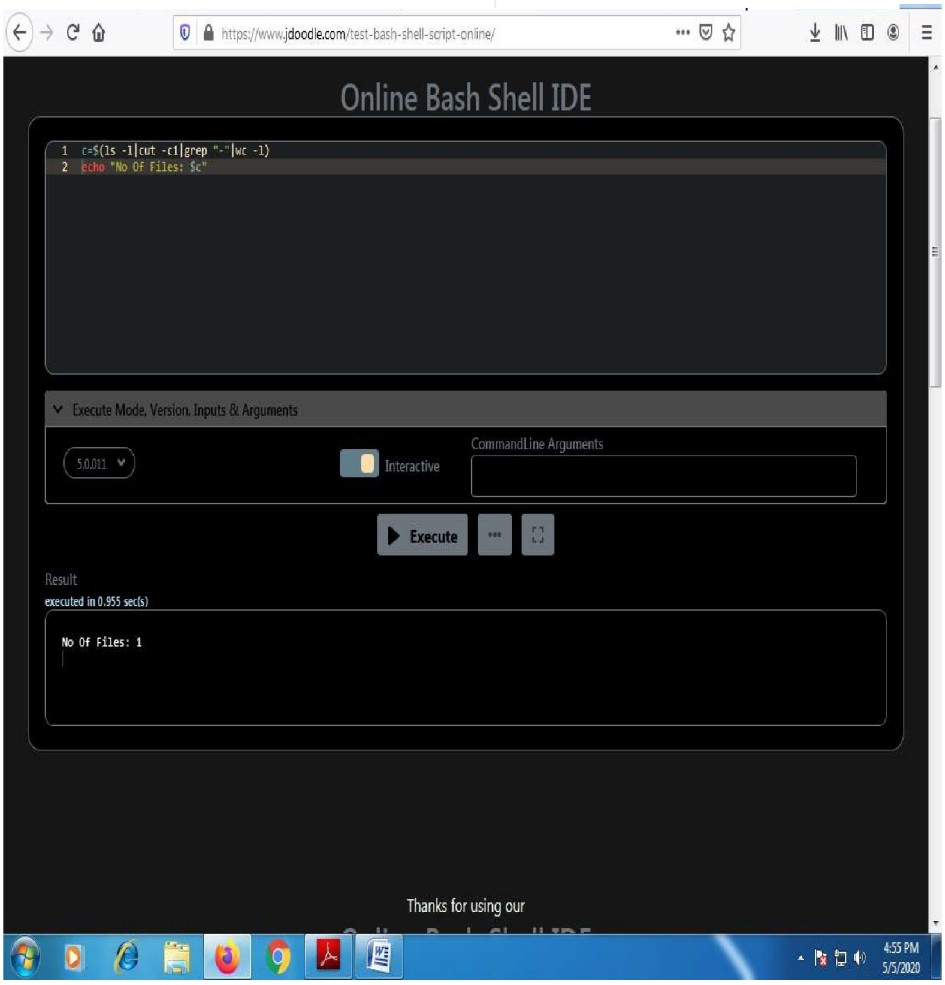
```
#who|wc –l
```

Program:

Write a shell program which counts total no of file in a directory.

```
c=$(ls -l|cut -c1|grep "-"|wc -l)
echo "No Of Files: $c"
```

Output:


https://www.jdoodle.com/test-bash-shell-script-online/
Online Bash Shell IDE
1 c=$(ls -1|cut -c1|grep "-"|wc -l)
2 echo "No Of Files: $c"
Execute Mode, Version, Inputs & Arguments
5.0.011
Interactive
CommandLine Arguments
Execute
Result
executed in 0.955 sec(s)
No Of Files: 1
Thanks for using our
4:55 PM
5/5/2020