**re$LIST OF EXPERIMENTS**

1. Develop programs for demonstrating shell programs.

* + - * Finding whether the given number is an Armstrong number
      * To print the Fibonacci series
      * To find the largest of n numbers
      * To compare 2 strings
      * To do string search
      * Creation of process and child process
      * Demonstration of inter-process communication
      * Creation of Zombie and Orphan process
      * Creation of threads

2. Demonstration of shared memory concept

3. Simulation of the CPU scheduling algorithms

4. Demonstration of Semaphores

5. Implementation of Producer-Consumer problem using semaphores

6. Simulation of Bankers algorithm for deadlock avoidance

7. Simulation of First fit, best fit, worst fit memory allocation methods.

8. Basic Shell commands

9. Creation of virtual machine in a hypervisor (Can’t be implemented)

**STRUCTURE OF THE EXPERIMENTS:**

* + 1. **AIM**
    2. **ALGORITHM**
    3. **PROGRAM CODE**
    4. **INPUT AND OUTPUT(SCREENSHOT)**
    5. **RESULT**

**EXPERIMENT 1.1:**

**AIM:**

To find whether the given number is an Armstrong number using shell programming.

**ALGORITHM:**

1. Start.
2. read number
3. set sum=0 and temp=number
4. reminder=number%10
5. sum=sum+(reminder\*reminder\*reminder)
6. number=number/10
7. repeat steps 4 to 6 until number > 0
8. if sum = temp
9. display number is Armstrong
10. else
11. display number is not Armstrong
12. End.

**PROGRAM CODE:**

echo "Enter the number"

read n

function ams

{

t=$n

s=0

b=0

c=10

while [ $n -gt $b ]

do

r=$((n % c))

i=$((r \* r \* r))

s=$((s + i))

n=$((n / c))

done

echo $s

if [ $s == $t ]

then

echo "Amstrong number"

else

echo "Not an Armstrong number"

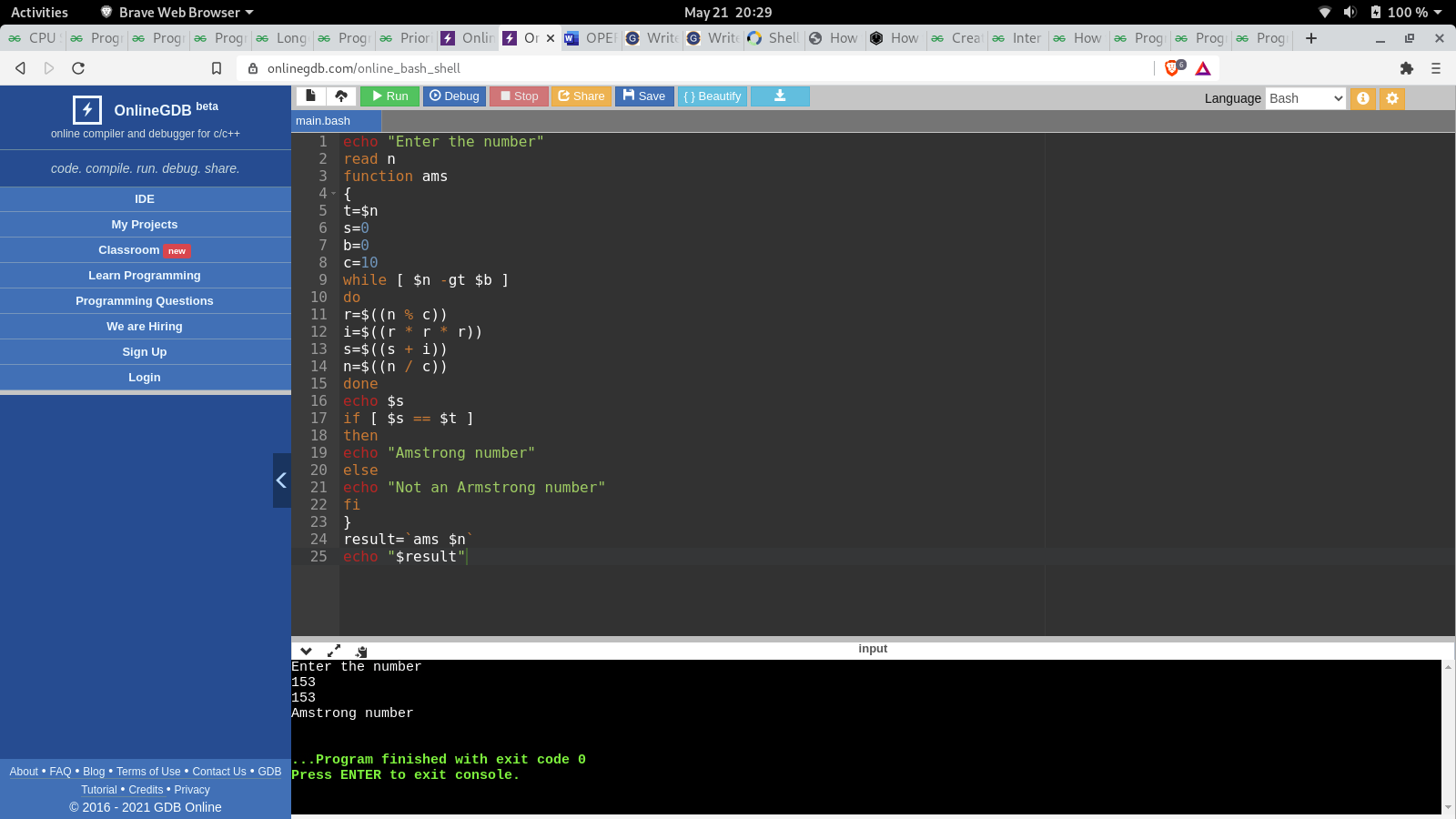
fi

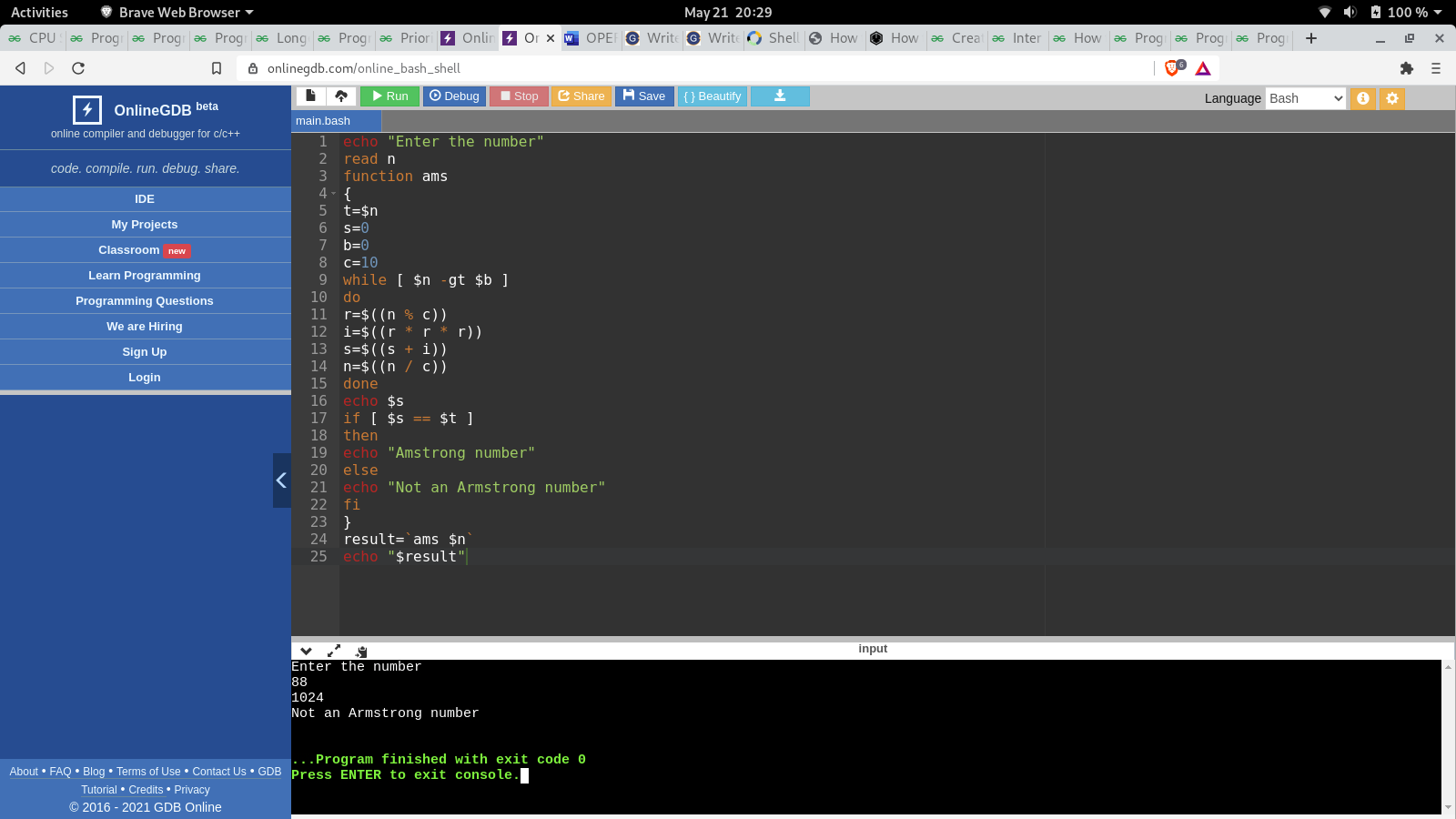
}

result=`ams $n`

echo "$result"

**INPUT AND OUTPUT:**





**RESULT:**

Hence the program for finding Armstrong number has been done using shell programming.

**EXPERIMENT 1.2:**

**AIM:**

To print the Fibonacci series using shell programming.

**ALGORITHM:**

1. Start
2. Declare variables i, f1, f2, result
3. Initialize the variables, f1=0, f2=1, and result =0
4. Enter the number of terms of Fibonacci series to be printed
5. Print First two terms of series
6. Use loop for the following steps
7. result = f1+f2
8. f1 =f2
9. f2= result
10. increase value of i each time by 1
11. print the value of show
12. End

**PROGRAM CODE:**

echo "How many number of terms to be generated ?"

read n

function fib

{

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

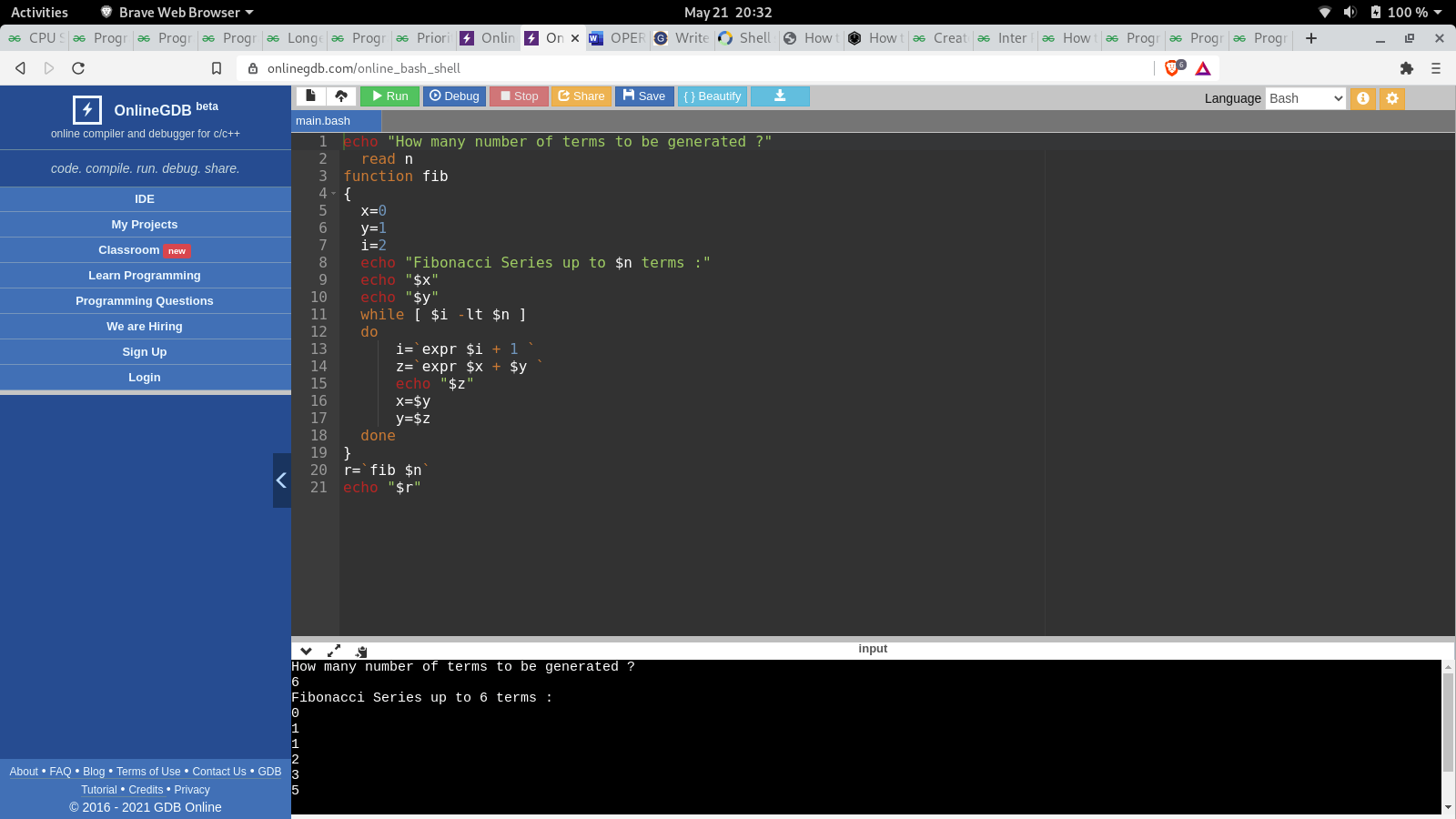
done

}

r=`fib $n`

echo "$r"

**INPUT AND OUTPUT:**



**RESULT:**

Hence the Fibonacci series has been printed successfully.

**EXPERIMENT 1.3:**

**AIM:**

To find the largest of n numbers using shell programming.

**ALGORITHM:**

1. Start.
2. Get N.
3. Read N numbers using loop.
4. Set first number as max.
5. From number 2 onwards update the max value if the number > max.
6. Display the result.
7. End.

**PROGRAM CODE:**

echo "Enter Size(N)"

read N

i=1

max=0

echo "Enter Numbers"

while [ $i -le $N ]

do

read num

if [ $i -eq 1 ]

then

max=$num

else

if [ $num -gt $max ]

then

max=$num

fi

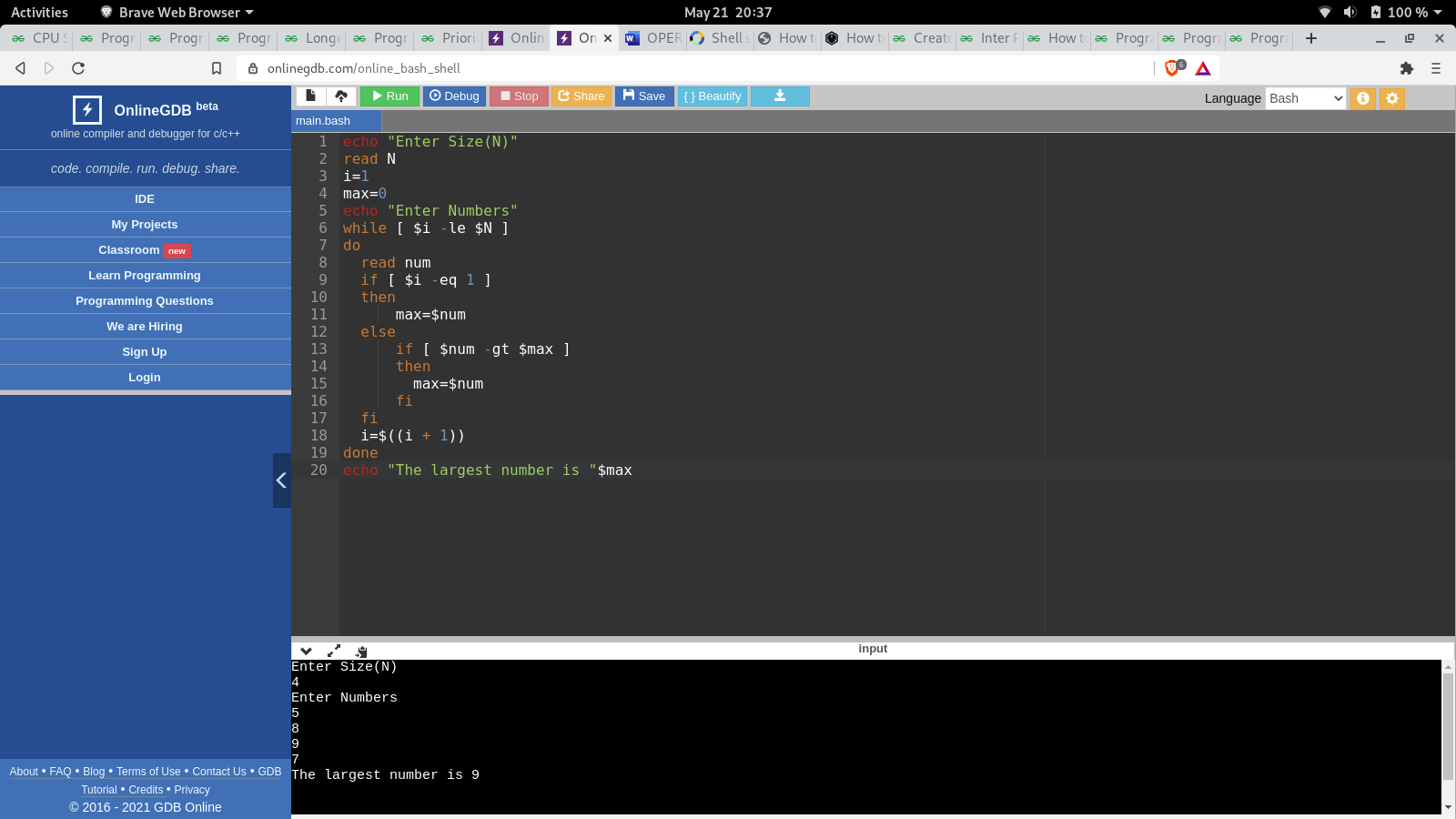
fi

i=$((i + 1))

done

echo "The largest number is "$max

**INPUT AND OUTPUT:**



**RESULT:**

Hence the largest of n numbers using the shell programming has been completed.

**EXPERIMENT 1.4:**

**AIM:**

To compare two strings using shell programming.

**ALGORITHM:**

1. Start.
2. Input the two strings as str1 and str2.
3. Check for the two strings equality.
4. If both are equal print the strings are equal.
5. If not equal then print the strings are not equal.
6. End.

**PROGRAM CODE:**

echo "Enter string 1"

read -a string1

echo "Enter string 2"

read -a string2

if [ $string1 == $string2 ];

then

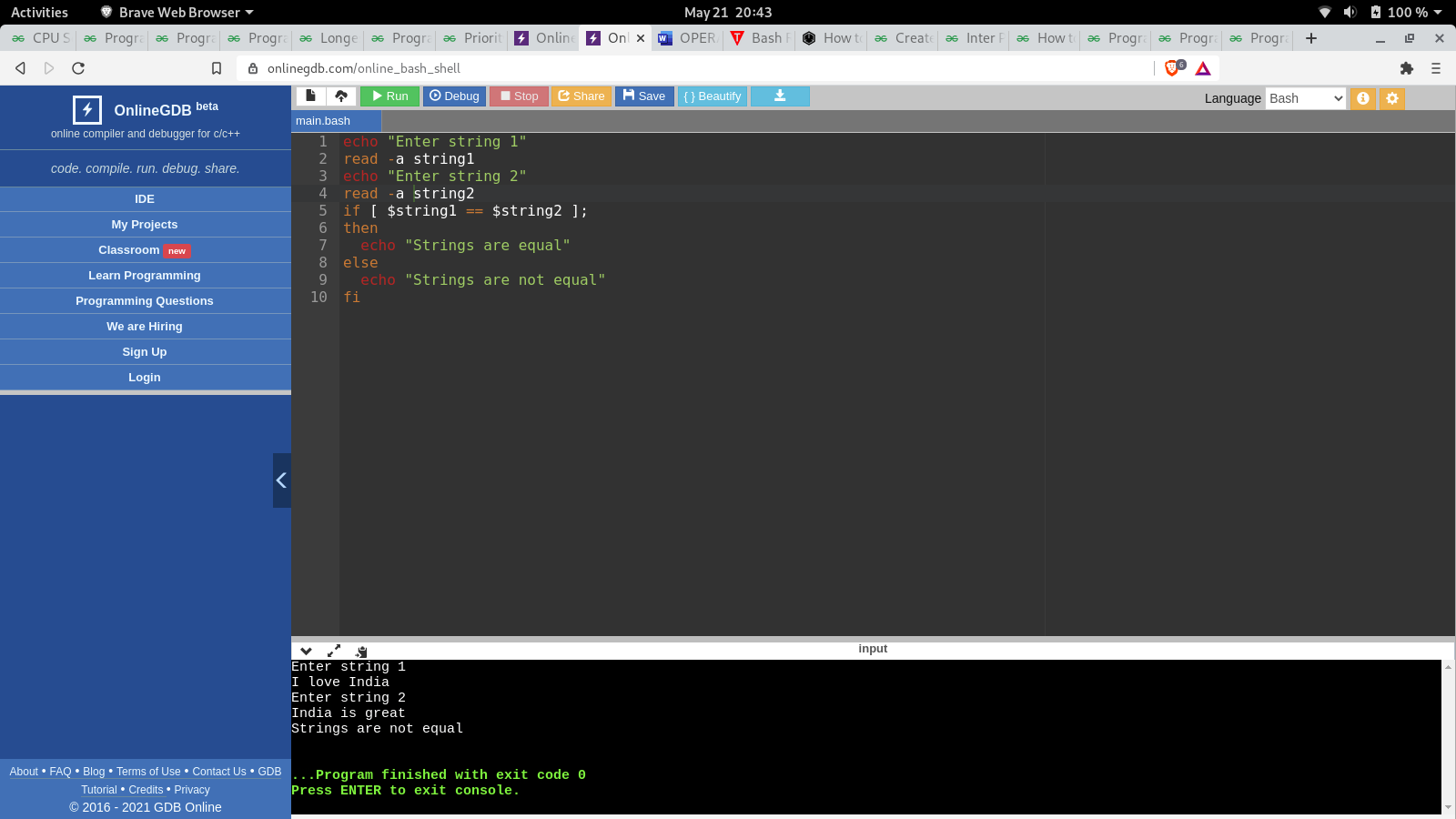
echo "Strings are equal"

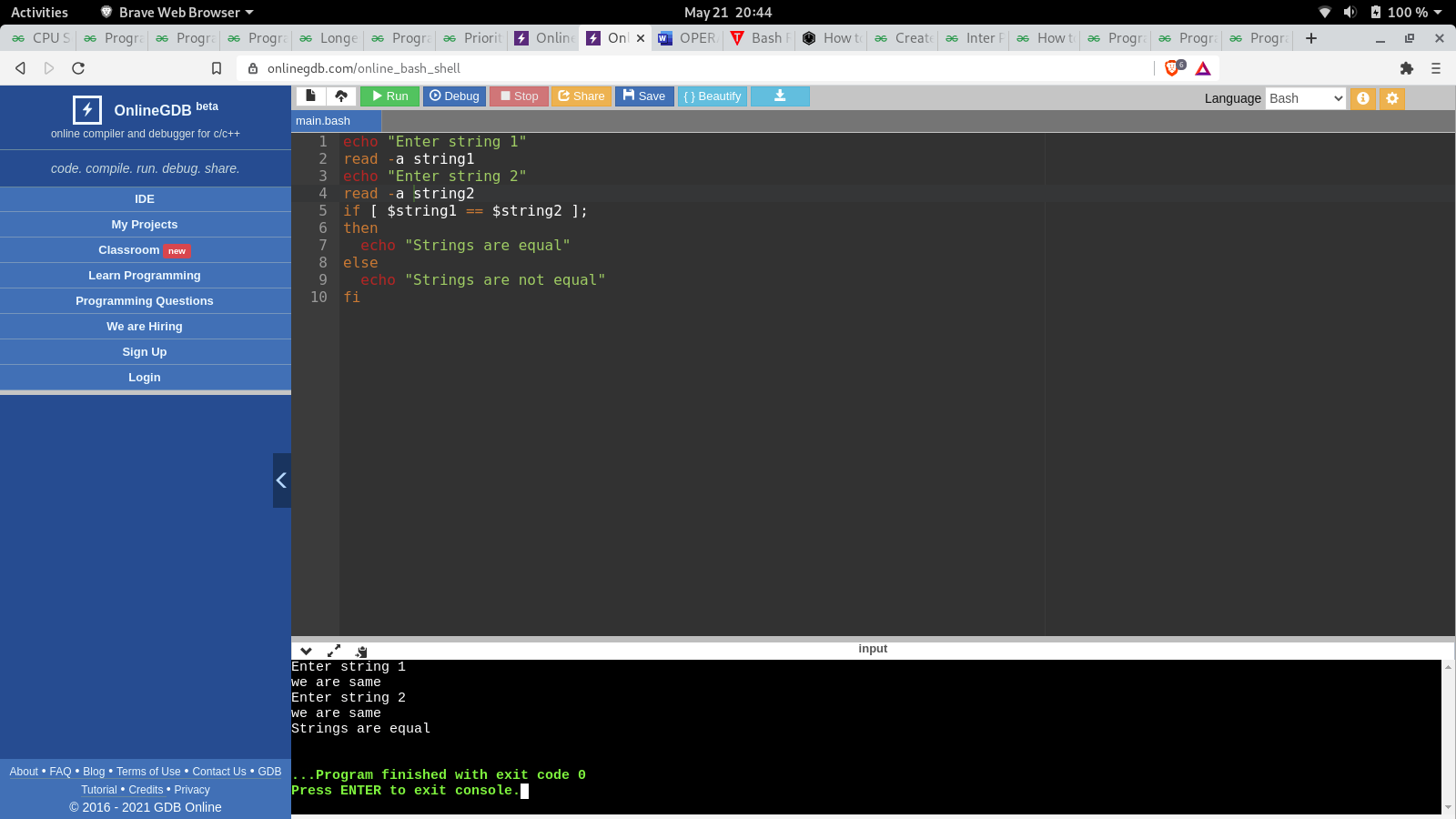
else

echo "Strings are not equal"

fi

**INPUT AND OUTPUT:**





**RESULT:**

Hence the shell program for comparing the two strings has been completed.

**EXPERIMENT 1.5:**

**AIM:**

To do a string search using shell programming.

**ALGORITHM:**

1. Start.
2. Input the two strings as str1 and str2.
3. Check for the two strings equality.
4. If the sub string is present in the string are equal.
5. If not equal then print the sub string is not equal.
6. End.

**PROGRAM CODE:**

STR='Linux is an operating system'

SUB='Linux'

if [[ "$STR" == \*"$SUB"\* ]]; then

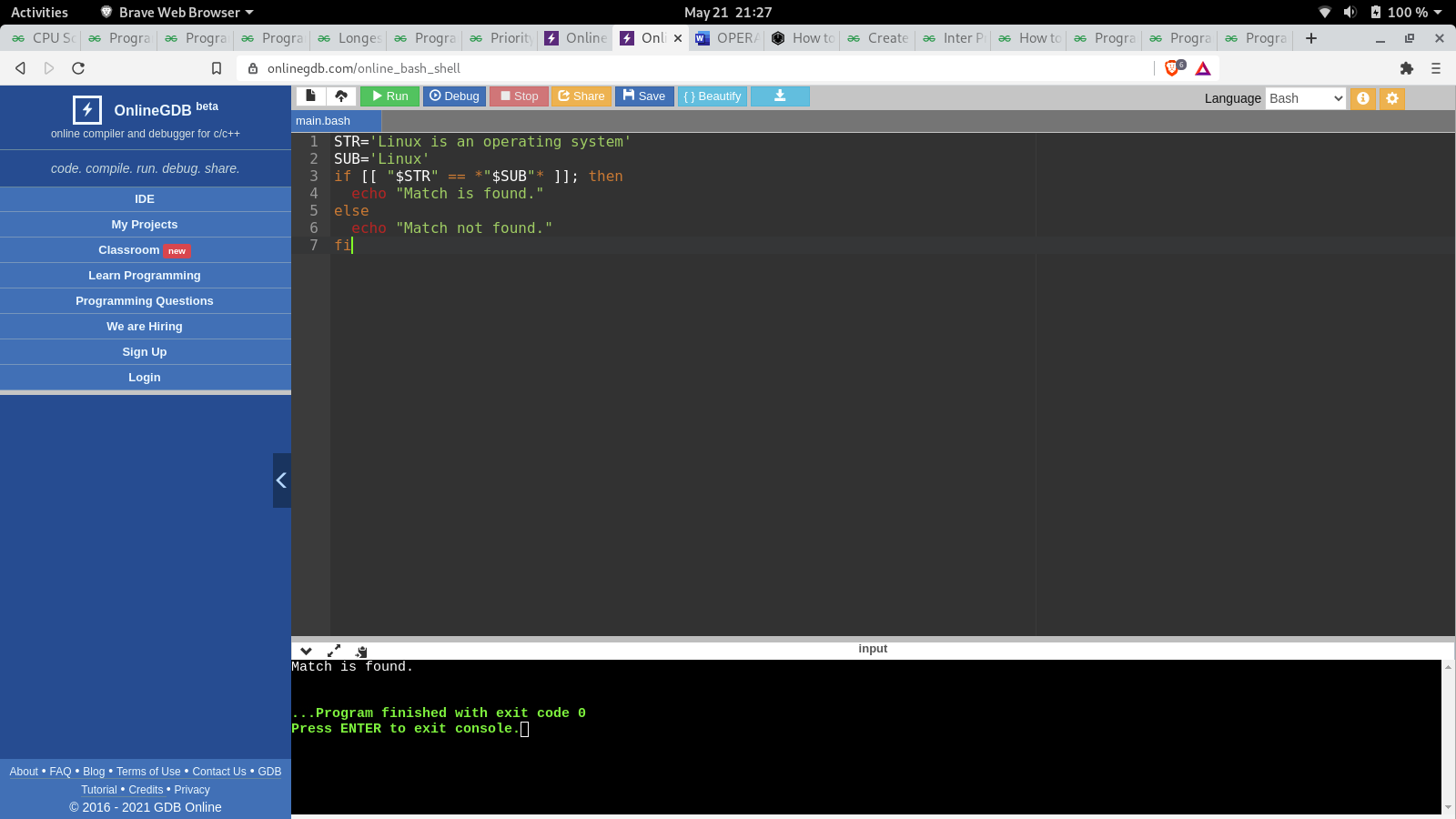
echo "Match is found."

else

echo "Match not found."

fi

**INPUT AND OUTPUT:**



**RESULT:**

Hence the string search using shell programming has been completed.

**EXPERIMENT 1.6:**

**AIM:**

To create a parent process and child process.

**ALGORITHM:**

1. Start.
2. Start a function.
3. Call a fork function (It divides the process into parent and child process).
4. Infer the difference between the process id’s .
5. End.

**PROGRAM CODE:**

#include<stdio.h>

int main()

{

for(int i=0;i<5;i++) // loop will run n times (n=5)

{

if(fork() == 0)

{

printf("[child] pid %d from [parent] pid %d\n",getpid(),getppid());

exit(0);

}

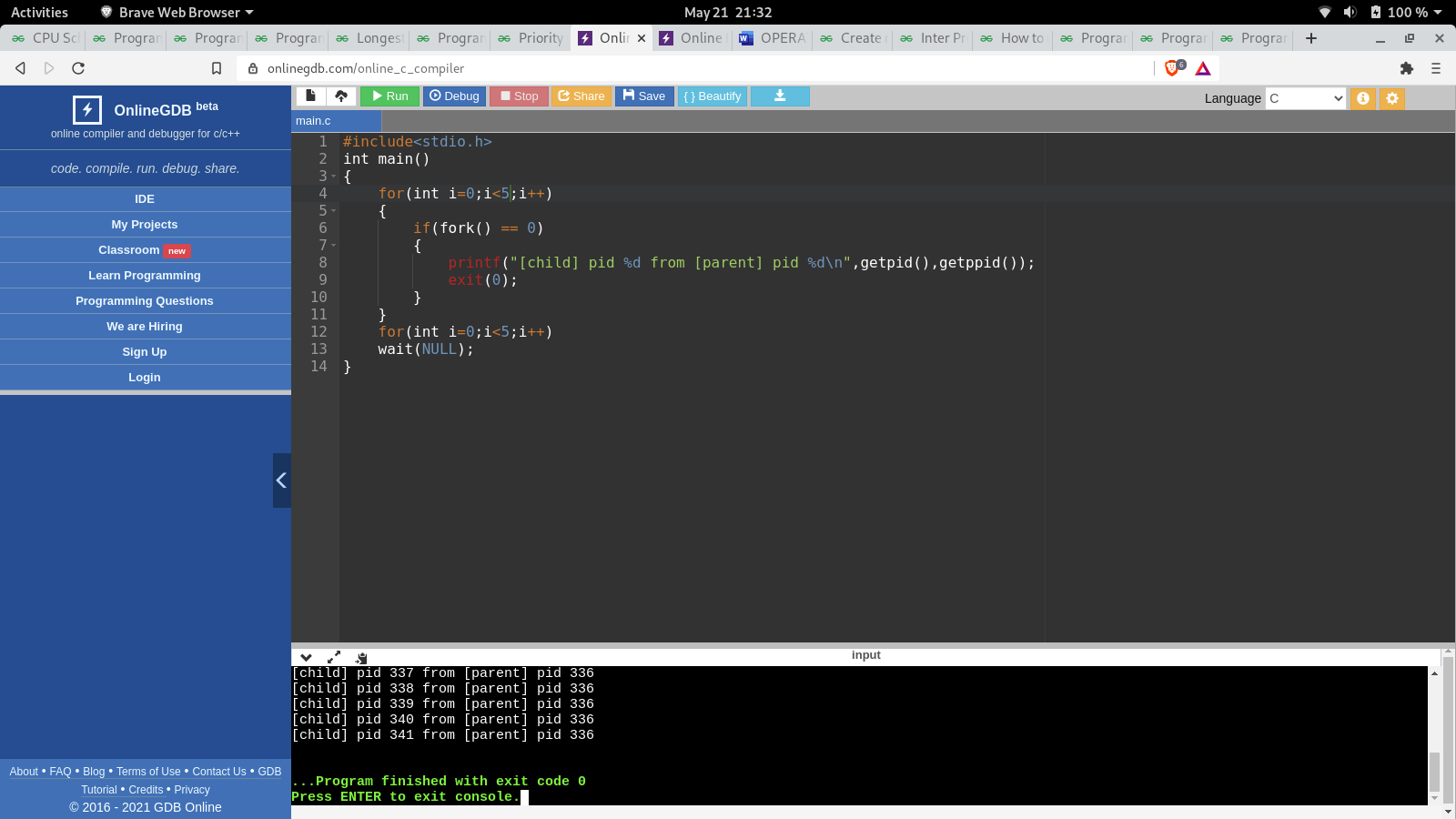
}

for(int i=0;i<5;i++) // loop will run n times (n=5)

wait(NULL);

}

**INPUT AND OUTPUT:**



**RESULT:**

The creation of a process and child process are executed successfully.

**EXPERIMENT 1.7:**

**AIM:**

To demonstrate inter-process communication in the operating system using producer consumer problem (Message passing).

**ALGORITHM:**

1. Start
2. Create a producer process code
3. Sharing the message with consumer.
4. Create a consumer process code.
5. Sharing the message with the producer.
6. End.

**PROGRAM CODE:**

void Producer(void){

int item;

Message m;

while(1){

receive(Consumer, &m);

item = produce();

build\_message(&m , item ) ;

send(Consumer, &m);

}

}

void Consumer(void){

int item;

Message m;

while(1){

receive(Producer, &m);

item = extracted\_item();

send(Producer, &m);

consume\_item(item);

}

}

**RESULT:**

Inter-process thread communication has been demonstrated successfully.

**EXPERIMENT 1.8:**

**AIM:**

To Create zombie and orphan processes in operating systems.

**ALGORITHM:**

1. Start.
2. Start a function.
3. Call a fork function (It divides the process into parent and child process).
4. Create a zombie and orphan process.
5. End.

**PROGRAM CODE:**

1. Zombie process

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

pid\_t child\_pid = fork();

// Parent process

if (child\_pid > 0)

sleep(50);

else

exit(0);

return 0;

}

2. Orphan process

#include<stdio.h>

#include<unistd.h>

int main()

{

pid\_t p;

p=fork();

if(p==0)

sleep(10);

printf("The child process pid is %d parent pid %d\n", getpid(), getppid());

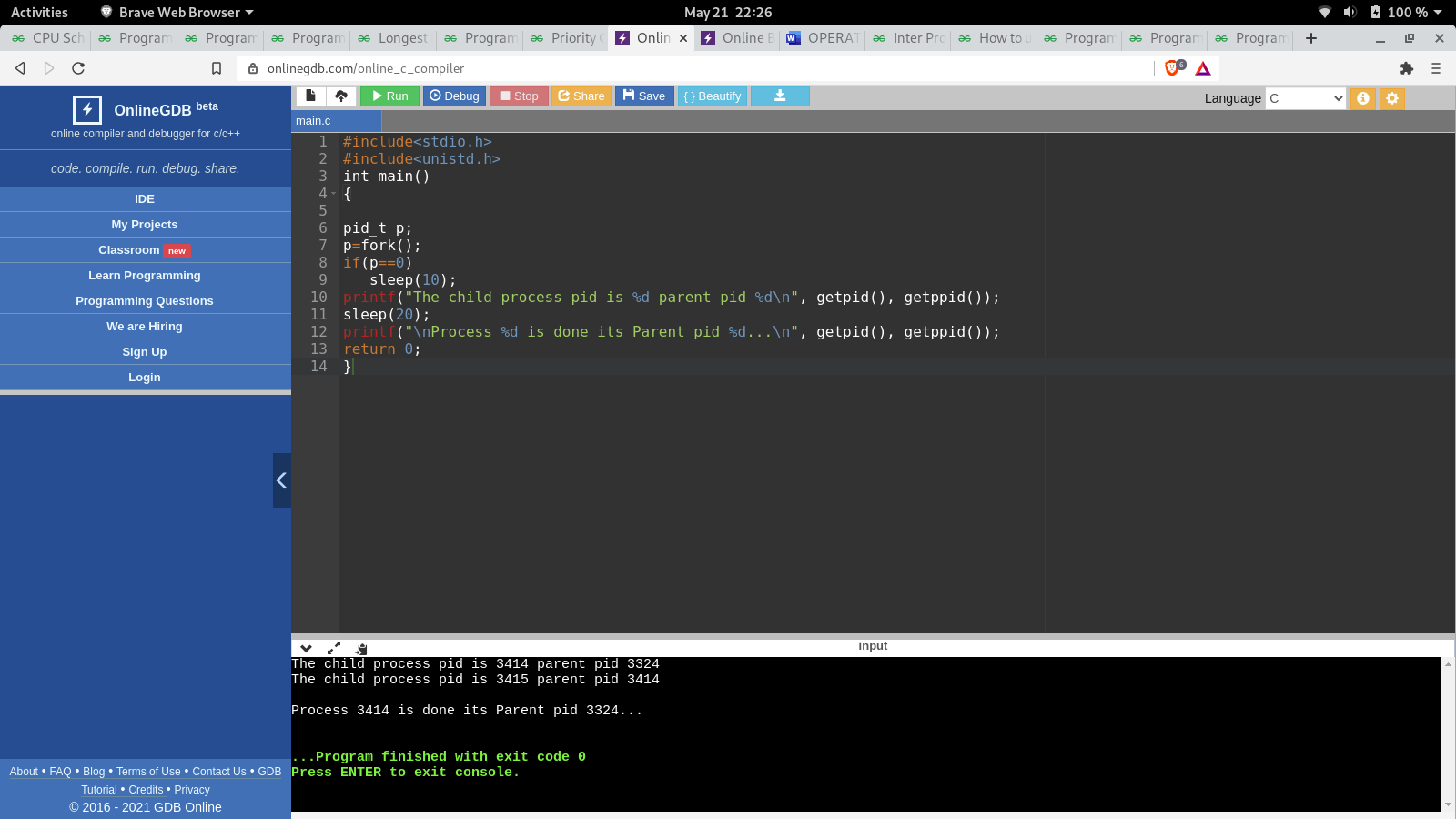
sleep(20);

printf("\nProcess %d is done its Parent pid %d...\n", getpid(), getppid());

return 0;

}

**INPUT AND OUTPUT:**



**RESULT:**

The creation of zombie and orphan processes have been created successfully.

**EXPERIMENT 1.9:**

**AIM:**

To create threads in operating system.

**ALGORITHM:**

1. Start.
2. Create a thread using pthread
3. Display sample texts.
4. End.

**PROGRAM CODE:**

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

#include <pthread.h>

#include <unistd.h>

void worker(void \*);

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

pthread\_t t1;

int thread\_id = 1;

if ( (pthread\_create(&t1, NULL, (void \*)&worker, (void \*)&thread\_id)) != 0) {

printf("Error creating thread\n");

exit(1);

}

pthread\_join(t1, NULL);

return 0;

}

void worker(void \*a) {

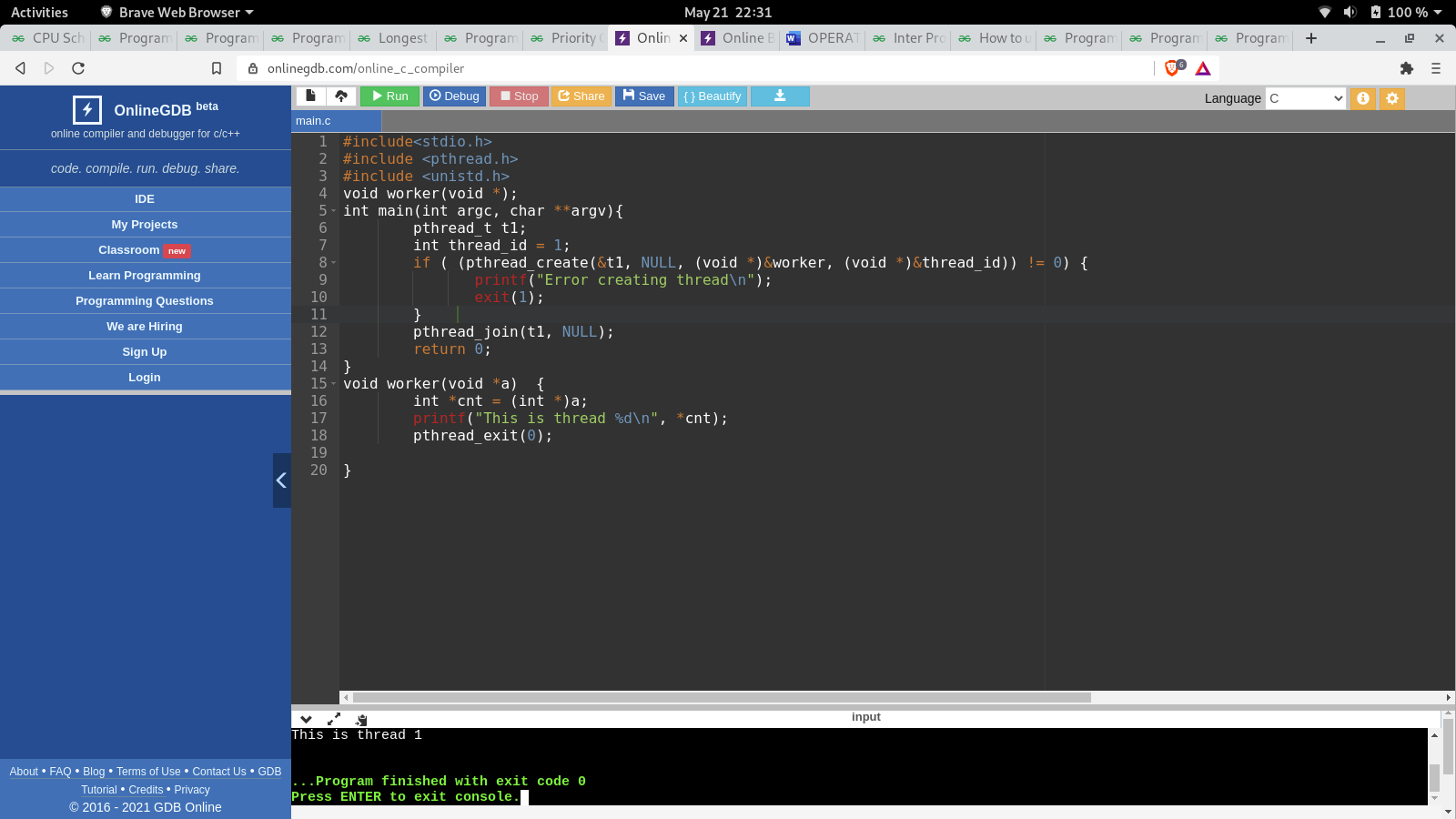
int \*cnt = (int \*)a;

printf("This is thread %d\n", \*cnt);

pthread\_exit(0);

}

**INPUT AND OUTPUT:**



**RESULT:**

Thread creation in operating systems in successful.

**EXPERIMENT 2:**

**AIM:**

To demonstrate shared memory concept in operating systems.

**ALGORITHM:**

1. Start.
2. Create a shared memory for the producer and consumer.
3. Create producer process
4. Create a consumer process
5. Communicate between producer and consumer
6. End.

**PROGRAM CODE:**

#define buff\_max 25

#define mod %

struct item{

// different member of the produced data

// or consumed data

---------

}

// An array is needed for holding the items.

// This is the shared place which will be

// access by both process

// item shared\_buff [ buff\_max ];

// Two variables which will keep track of

// the indexes of the items produced by producer

// and consumer The free index points to

// the next free index. The full index points to

// the first full index.

int free\_index = 0;

int full\_index = 0;

item nextProduced;

while(1){

// check if there is no space

// for production.

// if so keep waiting.

while((free\_index+1) mod buff\_max == full\_index);

shared\_buff[free\_index] = nextProduced;

free\_index = (free\_index + 1) mod buff\_max;

}

item nextConsumed;

while(1){

// check if there is an available

// item for consumption.

// if not keep on waiting for

// get them produced.

while((free\_index == full\_index);

nextConsumed = shared\_buff[full\_index];

full\_index = (full\_index + 1) mod buff\_max;

}

**RESULT:**

The demonstration of shared memory concepts in operation systems have been done successfully.