NAME: Meet Raut

DIV: S2-1

ROLL NO: 2201084

EXPERIMENT – 10:

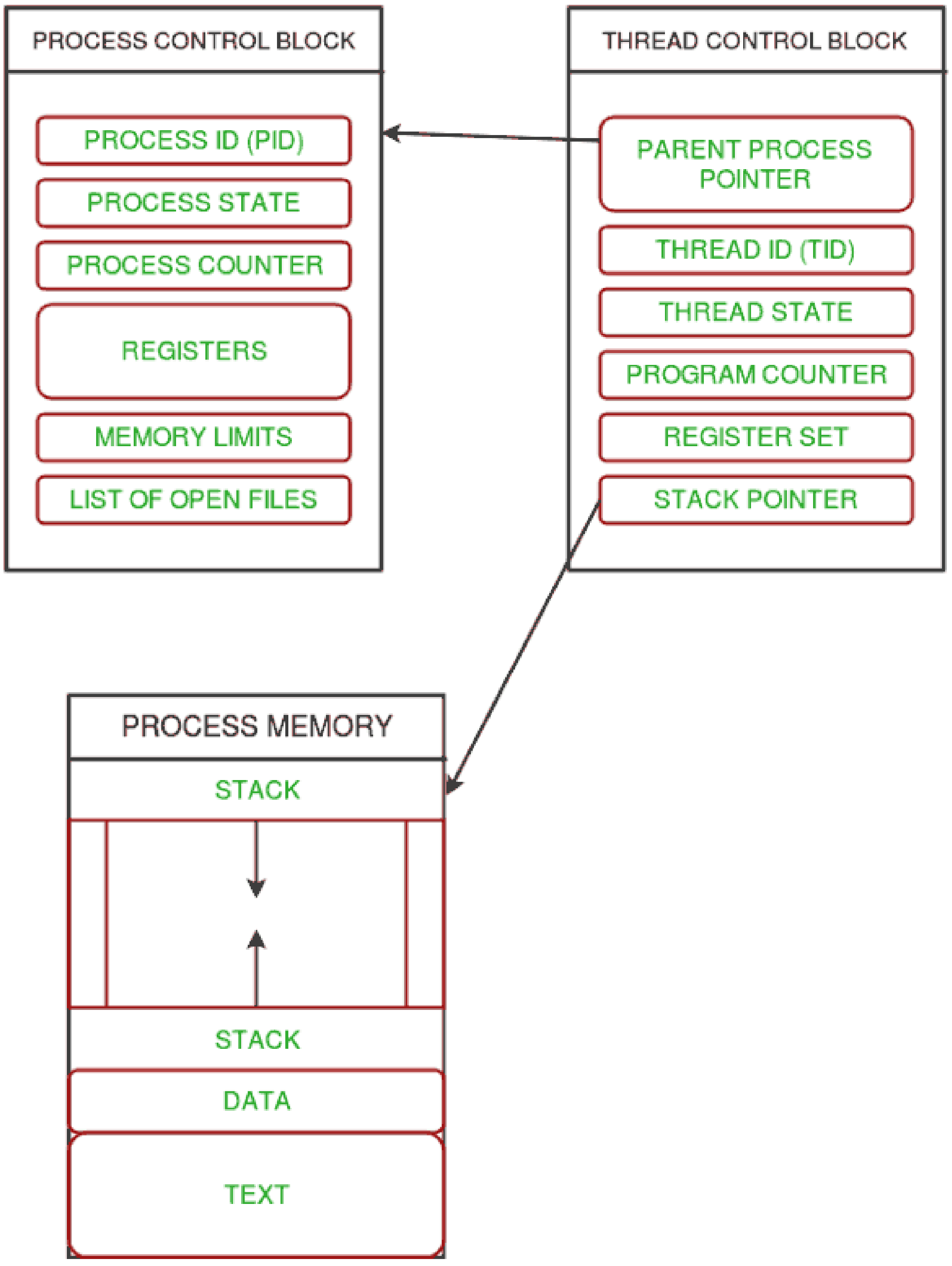
- ***AIM:*** **To study and implement programs on Threading using python.**

- ***THEORY:***

An Intro to Python Threading:

A **thread** is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS (Operating System). In simple words, a thread is a sequence of such instructions within a program that can be executed independently of other code. For simplicity, you can assume that a thread is simply a subset of a process! A thread contains all this information in a **Thread Control Block (TCB)**:

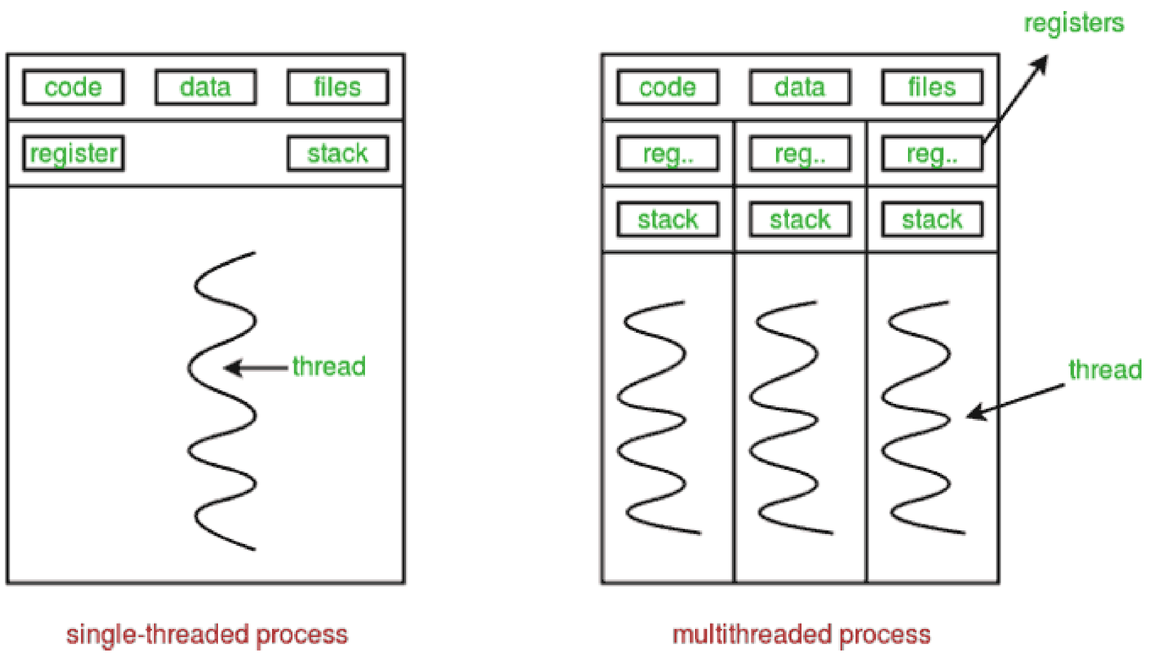
- **Thread Identifier:** Unique id (TID) is assigned to every new thread
- **Stack pointer:** Points to the thread's stack in the process. The stack contains the local variables under the thread's scope.
- **Program counter:** a register that stores the address of the instruction currently being executed by a thread.
- **Thread state:** can be running, ready, waiting, starting, or done.
- **Thread's register set:** registers assigned to thread for computations.
- **Parent process Pointer:** A pointer to the Process control block (PCB) of the process that the thread lives on.



Multiple threads can exist within one process where:

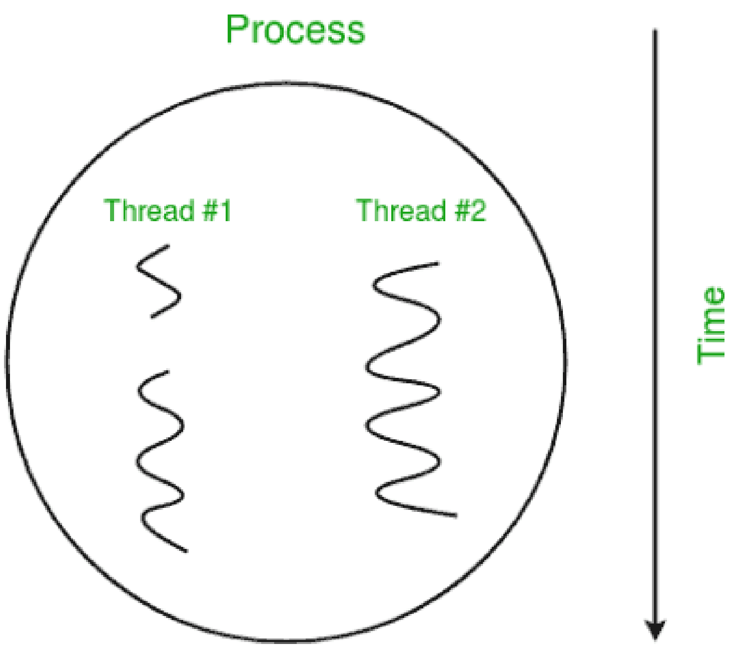
- Each thread contains its own **register set** and **local variables (stored in the stack)**.
- All threads of a process share **global variables (stored in heap)** and the **program code**.

Consider the diagram below to understand how multiple threads exist in memory:



Python Multithreading:

Multithreading is a stringing procedure in Python programming to run various strings simultaneously by quickly exchanging between strings with a central processor help (called setting exchanging). In addition, it makes it possible to share its data space with the primary threads of a process, making it easier than with individual processes to share information and communicate with other threads. The goal of multithreading is to complete multiple tasks at the same time, which improves application rendering and performance.



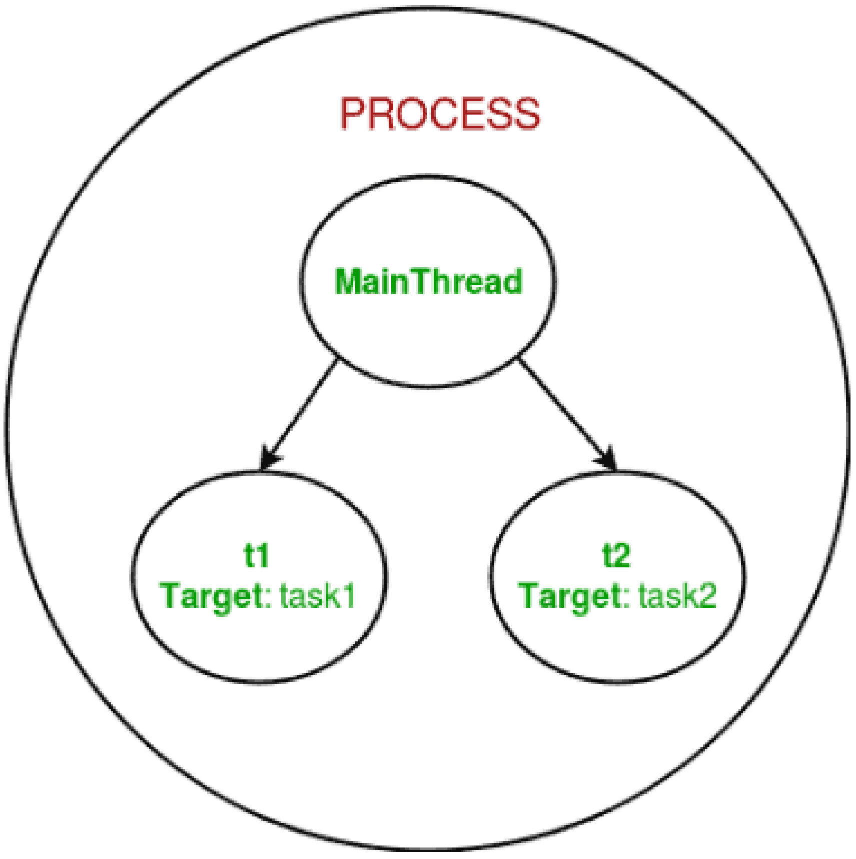
Benefits of Using Python for Multithreading:

The following are the advantages of using Python for multithreading:

1. It guarantees powerful usage of PC framework assets.
2. Applications with multiple threads respond faster.
3. It is more cost-effective because it shares resources and its state with sub-threads (child).
4. It makes the multiprocessor engineering more viable because of closeness.
5. By running multiple threads simultaneously, it cuts down on time.
6. To store multiple threads, the system does not require a lot of memory.

When to use Multithreading in Python?

It is an exceptionally valuable strategy for efficient and working on the presentation of an application. Programmers can run multiple subtasks of an application at the same time by using multithreading. It lets threads talk to the same processor and share resources like files, data, and memory. In addition, it makes it easier for the user to continue running a program even when a portion of it is blocked or too long.



How to achieve multithreading in Python?

There are two main modules of multithreading used to handle threads in Python.

1. The thread module
2. The threading module

Thread modules

It is started with Python 3, designated as obsolete, and can only be accessed with _thread that supports backward compatibility.

Syntax:

```
thread.start_new_thread ( function_name, args[, kwargs] )
```

To implement the thread module in Python, we need to import a **thread** module and then define a function that performs some action by setting the target with a variable.

Threading Modules

The threading module is a high-level implementation of multithreading used to deploy an application in Python. To use multithreading, we need to import the threading module in Python Program.

Thread Class Methods

Methods	Description
start()	A start() method is used to initiate the activity of a thread. And it calls only once for each thread so that the execution of the thread can begin.
run()	A run() method is used to define a thread's activity and can be overridden by a class that extends the threads class.
join()	A join() method is used to block the execution of another code until the thread terminates.

Follow the given below steps to implement the threading module in Python Multithreading:

1. Import the threading module

Create a new thread by importing the **threading** module, as shown.

Syntax:

```
import threading
```

A **threading** module is made up of a **Thread** class, which is instantiated to create a Python thread.

2. Declaration of the thread parameters: It contains the target function, argument, and **kwargs** as the parameter in the **Thread()** class.

- **Target:** It defines the function name that is executed by the thread.
- **Args:** It defines the arguments that are passed to the target function name.

For example:

```
import threading
def print_hello(n):
print("Hello, how old are you ", n)
t1 = threading.Thread( target = print_hello, args =(18, ))
```

In the above code, we invoked the **print_hello()** function as the target parameter. The **print_hello()** contains one parameter **n**, which passed to the **args** parameter.

3. Start a new thread: To start a thread in Python multithreading, call the thread class's object. The start() method can be called once for each thread object; otherwise, it throws an exception error.

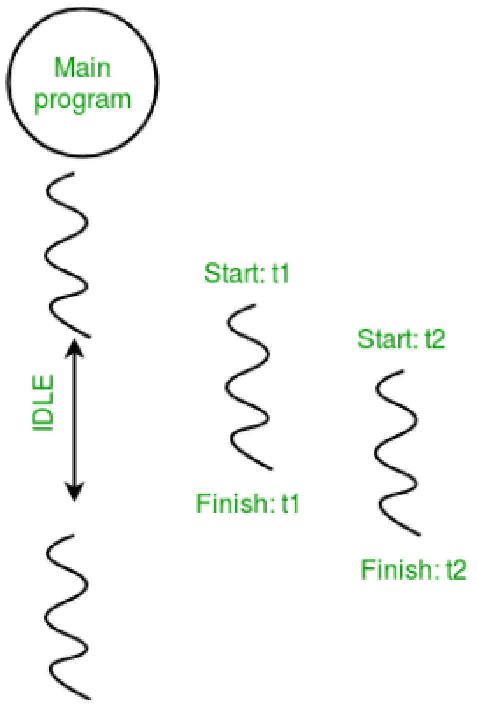
Syntax:

```
t1.start()
t2.start()
```

4. Join method: It is a join() method used in the thread class to halt the main thread's execution and waits till the complete execution of the thread object. When the thread object is completed, it starts the execution of the main thread in Python. The join() method stops the main thread from running when the above program is run and waits until the thread t1 has finished running. The main thread begins its execution once the t1 has completed successfully.

5. Synchronizing Threads in Python

It is a thread synchronization mechanism that makes sure that no two threads can run the same part of the program at the same time to access shared resources. Critical sections could be used to describe the situation. To avoid the critical section condition, in which two threads cannot simultaneously access resources, we employ a race condition.



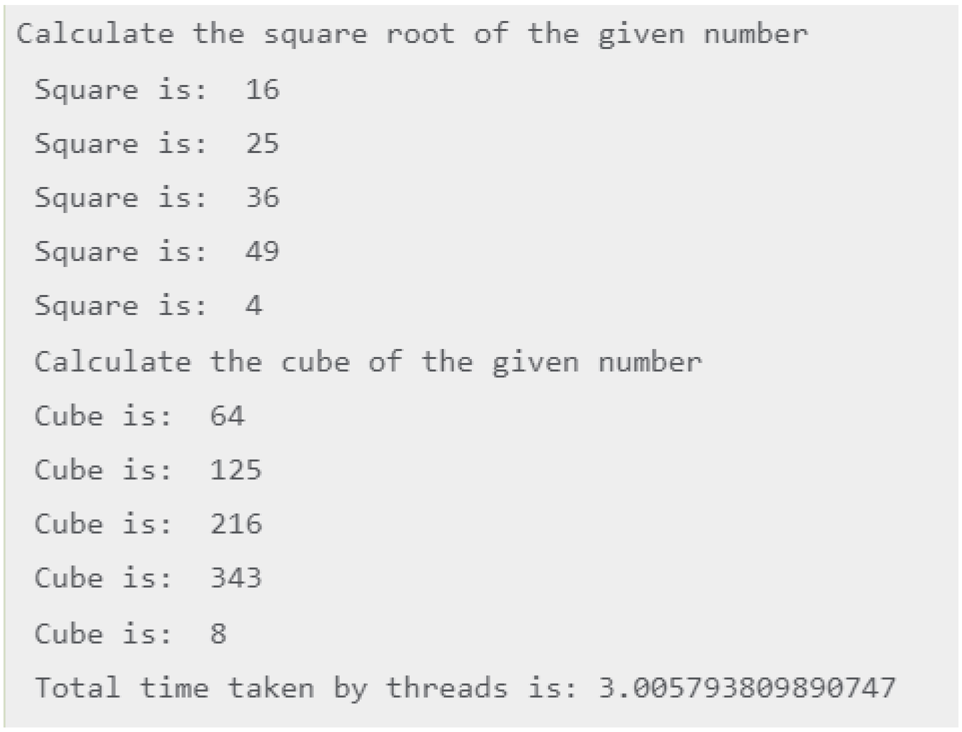
Python ThreadPool:

A thread pool is a collection of threads that are created in advance and can be reused to execute multiple tasks. The concurrent.futures module in Python provides a ThreadPoolExecutor class that makes it easy to create and manage a thread pool.
In this example, we define a function worker that will run in a thread. We create a ThreadPoolExecutor with a maximum of 2 worker threads. We then submit two tasks to the pool using the submit method. The pool manages the execution of the tasks in its worker threads. We use the shutdown method to wait for all tasks to complete before the main thread continues.
Multithreading can help you make your programs more efficient and responsive. However, it's important to be careful when working with threads to avoid issues such as race conditions and deadlocks.
This code uses a thread pool created with **concurrent.futures.ThreadPoolExecutor** to run two worker tasks concurrently. The main thread waits for the worker threads to finish using **pool.shutdown(wait=True)**. This allows for efficient parallel processing of tasks in a multi-threaded environment.

- **PROGRAM 1:**

```
import thread # import the thread module
import time # import time module

def cal_sqre(num): # define the cal_sqre function
    print(" Calculate the square root of the given number")
    for n in num:
        time.sleep(0.3) # at each iteration it waits for 0.3 time
        print(' Square is : ', n * n)

def cal_cube(num): # define the cal_cube() function
    print(" Calculate the cube of  the given number")
    for n in num:
        time.sleep(0.3) # at each iteration it waits for 0.3 time
        print(" Cube is : ", n * n *n)

arr = [4, 5, 6, 7, 2] # given array

t1 = time.time() # get total time to execute the functions
cal_sqre(arr) # call cal_sqre() function
cal_cube(arr) # call cal_cube() function

print(" Total time taken by threads is :", time.time() - t1) # print the total time
```

- **OUTPUT:**

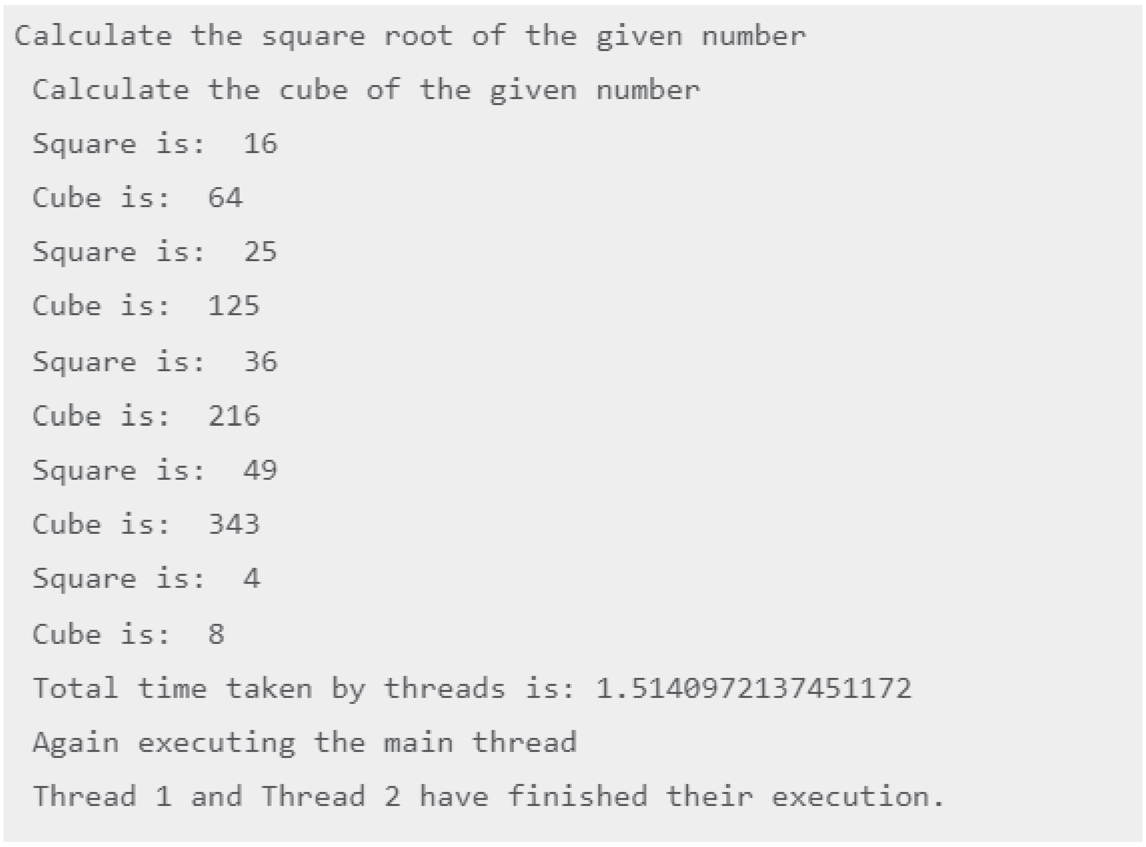

```
Calculate the square root of the given number
 Square is:  16
 Square is:  25
 Square is:  36
 Square is:  49
 Square is:  4
 Calculate the cube of the given number
 Cube is:  64
 Cube is:  125
 Cube is:  216
 Cube is:  343
 Cube is:  8
 Total time taken by threads is: 3.005793809890747
```

- **PROGRAM 2:**

```
import time # import time module
import threading
from threading import *
def cal_sqre(num): # define a square calculating function
    print(" Calculate the square root of the given number")
    for n in num: # Use for loop
        time.sleep(0.3) # at each iteration it waits for 0.3 time
        print(' Square is : ', n * n)

def cal_cube(num): # define a cube calculating function
    print(" Calculate the cube of  the given number")
    for n in num: # for loop
        time.sleep(0.3) # at each iteration it waits for 0.3 time
        print(" Cube is : ", n * n *n)
```

```
ar = [4, 5, 6, 7, 2] # given array

t = time.time() # get total time to execute the functions
#cal_cube(ar)
#cal_sqre(ar)
th1 = threading.Thread(target=cal_sqre, args=(ar, ))
th2 = threading.Thread(target=cal_cube, args=(ar, ))
th1.start()
th2.start()
th1.join()
th2.join()
print(" Total time taking by threads is :", time.time() - t) # print the total time
print(" Again executing the main thread")
print(" Thread 1 and Thread 2 have finished their execution.")
```

- **OUTPUT:**

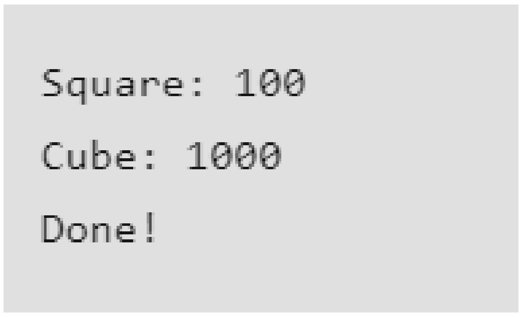


```
Calculate the square root of the given number
 Calculate the cube of the given number
 Square is:  16
 Cube is:  64
 Square is:  25
 Cube is:  125
 Square is:  36
 Cube is:  216
 Square is:  49
 Cube is:  343
 Square is:  4
 Cube is:  8
 Total time taken by threads is: 1.5140972137451172
 Again executing the main thread
 Thread 1 and Thread 2 have finished their execution.
```

- **<u>PROGRAM 3:</u>**

```
import threading

def print_cube(num):
        print("Cube: {}" .format(num * num * num))

def print_square(num):
        print("Square: {}" .format(num * num))

if __name__ =="__main__":
        t1 = threading.Thread(target=print_square, args=(10,))
        t2 = threading.Thread(target=print_cube, args=(10,))

        t1.start()
        t2.start()

        t1.join()
        t2.join()
        print("Done!")
```

- **OUTPUT:**



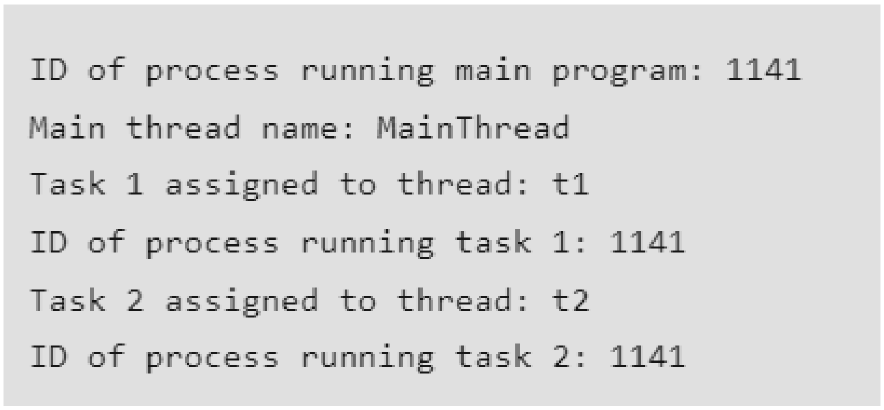
```
Square: 100
Cube: 1000
Done!
```

➢ **PROGRAM 4:**

```
import threading
import os

def task1():
        print("Task 1 assigned to thread:
{}".format(threading.current_thread().name))
        print("ID of process running task 1: {}".format(os.getpid()))

def task2():
        print("Task 2 assigned to thread:
{}".format(threading.current_thread().name))
        print("ID of process running task 2: {}".format(os.getpid()))

if __name__ == "__main__":

        print("ID of process running main program: {}".format(os.getpid()))

        print("Main thread name: {}".format(threading.current_thread().name))

        t1 = threading.Thread(target=task1, name='t1')
        t2 = threading.Thread(target=task2, name='t2')

        t1.start()
```

```
    t2.start()

    t1.join()
    t2.join()
```

- **OUTPUT:**

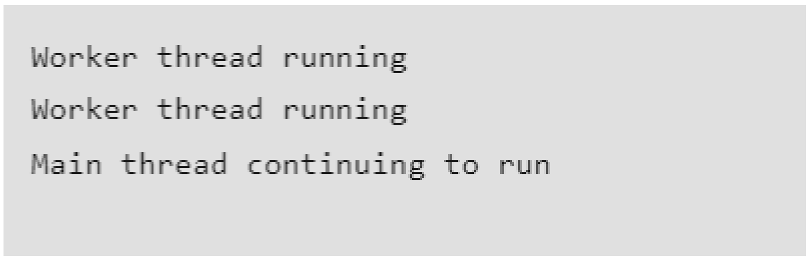


```
ID of process running main program: 1141
Main thread name: MainThread
Task 1 assigned to thread: t1
ID of process running task 1: 1141
Task 2 assigned to thread: t2
ID of process running task 2: 1141
```

- **PROGRAM 5:**

```
import concurrent.futures

def worker():
    print("Worker thread running")

pool = concurrent.futures.ThreadPoolExecutor(max_workers=2)

pool.submit(worker)
pool.submit(worker)

pool.shutdown(wait=True)

print("Main thread continuing to run")
```

- **OUTPUT:**

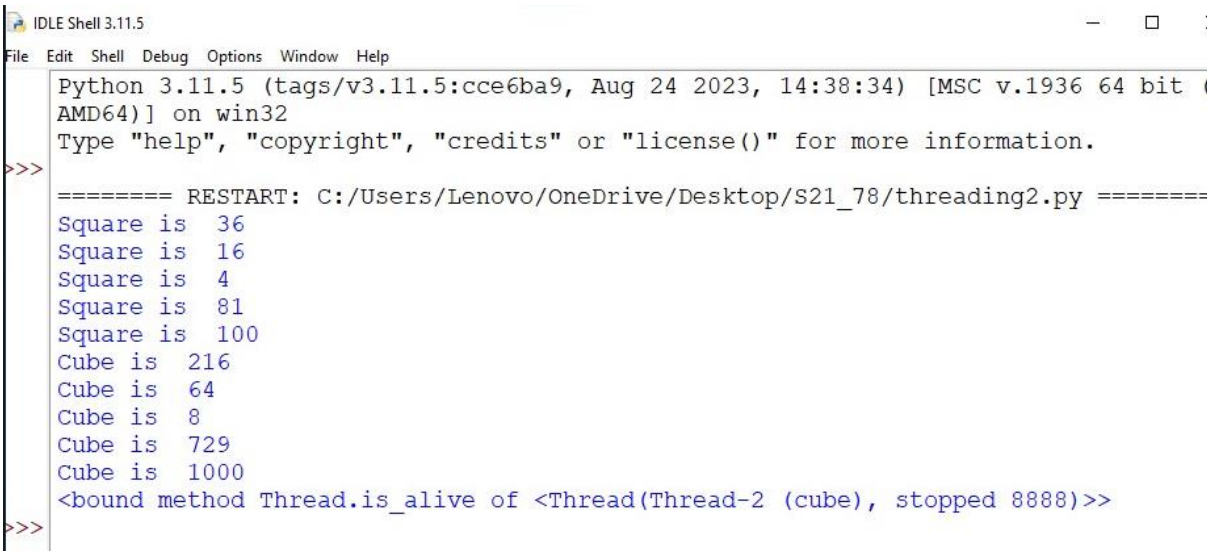


```
Worker thread running
Worker thread running
Main thread continuing to run
```

- **PROGRAM 6:**

```
import threading
import thread
import time

def squares(num):
    for n in num:
        time.sleep(0.3)
        print("Square is ",n*n)

def cube(num):
    time.sleep(0.3)
    print ("Cube is ",n*n*n)

arr=[6,4,2,9,10]

t1=threading.Thread(target=sqares, args=(arr, ))
t2=threading.Thread(target=cube, args=(arr, ))
t1.start()
t1.join()
t2.join()
print(t2.is_alive)
t1._stop()
```

- **OUTPUT:**



```
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======== RESTART: C:/Users/Lenovo/OneDrive/Desktop/S21_78/threading2.py ========
Square is  36
Square is  16
Square is  4
Square is  81
Square is  100
Cube is  216
Cube is  64
Cube is  8
Cube is  729
Cube is  1000
<bound method Thread.is_alive of <Thread(Thread-2 (cube), stopped 8888)>>
>>>
```

- ***CONCLUSION:*** **Hence, we have successfully implemented program on Threading using python; LO 5.**