Experiment No. 3

Title: Program on Multithreading using Python

Batch: B1 **Roll No: 16010420133** **Experiment No 3**

Aim: Program on implementation of multithreading in Python

Resources needed: Python IDE

Theory:

What is thread?

In computing, a process is an instance of a computer program that is being executed. Any process has 3 basic components:

- An executable program.
- The associated data needed by the program (variables, work space, buffers, etc.)
- The execution context of the program (State of process)

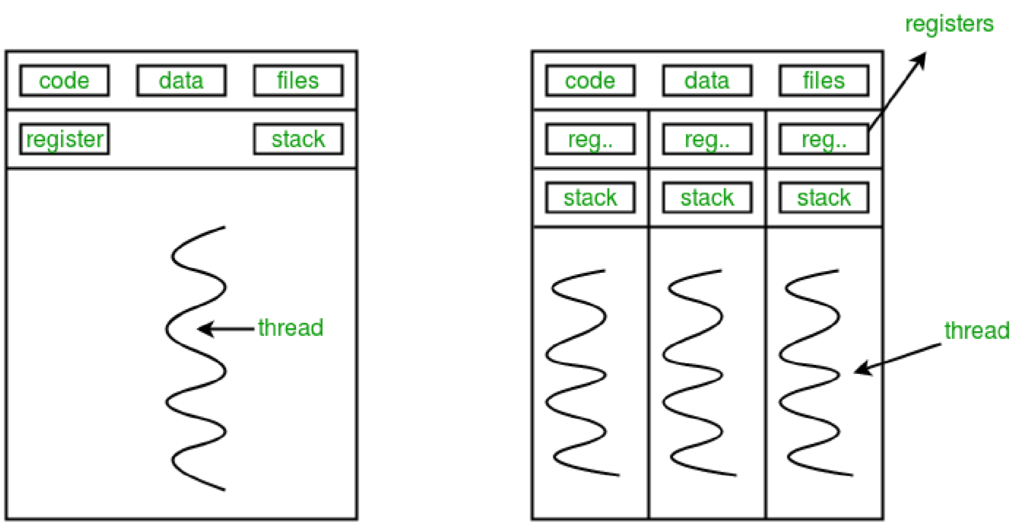
A thread is an entity within a process that can be scheduled for execution independently. Also, it is the smallest unit of processing that can be performed in an OS (Operating System).

What is Multithreading?

Multiple threads can exist within one process where:

- Each thread contains its own **register set** and **local variables (stored in stack)**.
- All thread 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:



single-threaded process multithreaded process

Multithreading is defined as the ability of a processor to execute multiple threads concurrently.
Multithreading in Python
In Python, the **threading** module provides a very simple and intuitive API for spawning multiple threads in a program.

```
# Python program to illustrate the concept of threading
# importing the threading module

import threading

def print_cube(num):
        """
        function to print cube of given num
        """

if__name__== "_main__":
        # creating thread
        t1 = threading.Thread(target=print_square, args=(10,))
                # starting thread 1
        t1.start()

        # wait until thread 1 is completely executed
        t1.join()

        # both threads completely executed
        print("Done!")
```

Creating a new thread and related methods: Using object of Thread class from threading module.

To create a new thread, we create an object of Thread class. It takes following arguments:

target: the function to be executed by thread

args: the arguments to be passed to the target function

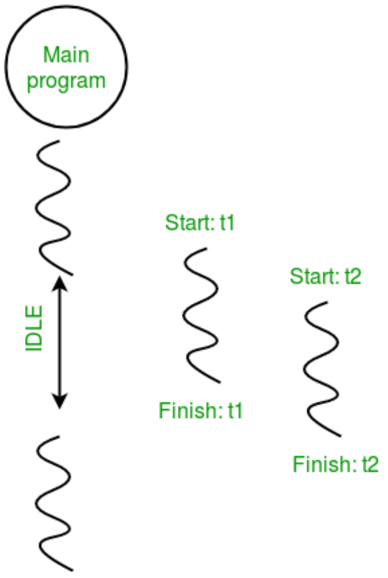
Once the threads start, the current program (you can think of it like a main thread) also keeps on executing. In order to stop execution of current program until a thread is complete, we use join method.

t1.join()

t2.join()

As a result, the current program will first wait for the completion of t1 and then t2. Once, they are finished, the remaining statements of current program are executed..

Diagram below depicts actual process of execution.



Threading.current_thread().name this will print current thread's name and threading.main_thread().name will print main thread's name. **os.getpid()** function to get ID of current process.

Thread synchronization is defined as a mechanism which ensures that two or more concurrent threads do not simultaneously execute some particular program segment. Concurrent accesses to shared resource can lead to race condition.

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. As a result, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes.

For example two threads trying to increment shared variable value may read same initial values and produce wrong result.

Hence there is requirement of acquiring lock on shared resource before use. **threading** module provides a **Lock** class to deal with the race conditions.

Lock class provides following methods:

acquire([blocking]) : To acquire a lock. A lock can be blocking or non-blocking.

When invoked with the blocking argument set to True (the default), thread execution is blocked until the lock is unlocked, then lock is set to locked and return True.

When invoked with the blocking argument set to False, thread execution is not blocked. If lock is unlocked, then set it to locked and return True else return False immediately.

release() : To release a lock.

When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

If lock is already unlocked, a ThreadError is raised.

Firstly, a Lock object is created using:

```
lock = threading.Lock()
```

Then, lock is passed as target function argument:

```
t1 = threading.Thread(target=thread_task, args=(lock,))

t2 = threading.Thread(target=thread_task, args=(lock,))
```

In the critical section of target function, we apply lock using lock.acquire() method. As soon as a lock is acquired, no other thread can access the critical section (here, increment function) until the lock is released using lock.release() method.

Activities:

a) Write a program to create three threads. Thread 1 will generate random number. Thread two will compute square of this number and thread 3 will compute cube of this number

Result: (script and output)

Code:

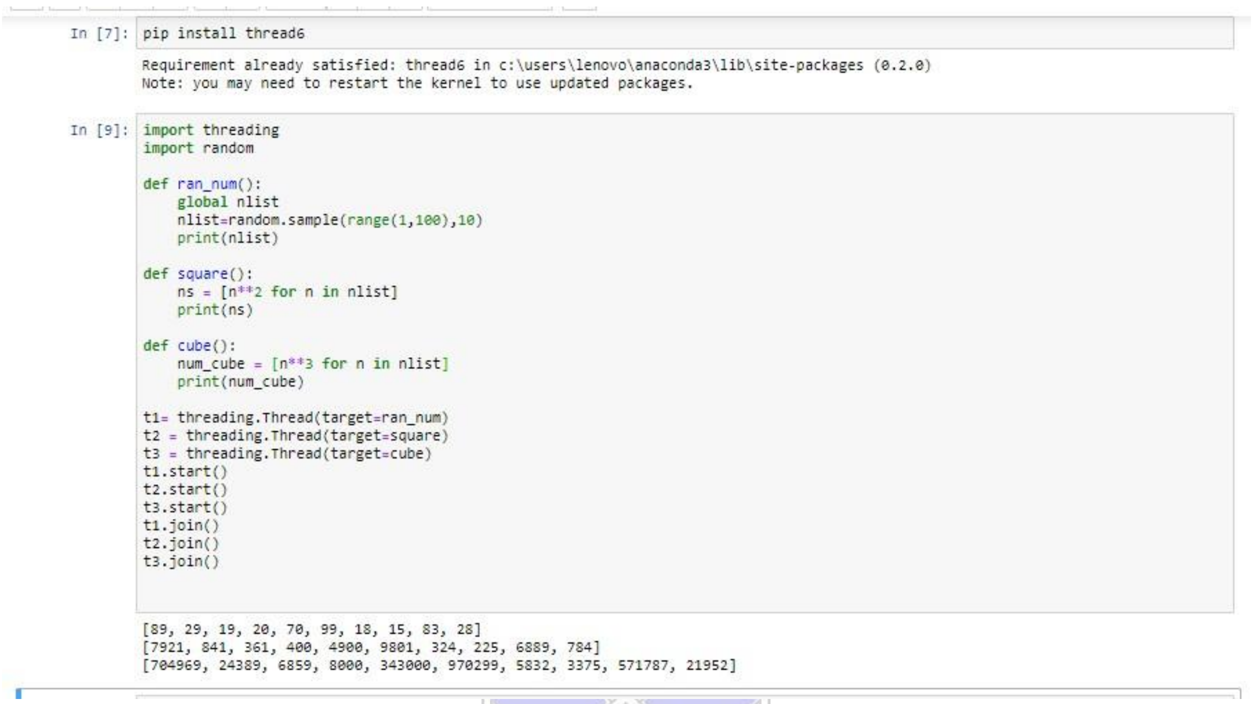
```
import threading
import random

def ran_num():
    global nlist
    nlist=random.sample(range(1,100),10)
    print(nlist)

def square():
    ns = [n**2 for n in nlist]
    print(ns)

def cube():
    num_cube = [n**3 for n in nlist]
    print(num_cube)

t1= threading.Thread(target=ran_num)
t2 = threading.Thread(target=square)
t3 = threading.Thread(target=cube)
t1.start()
t2.start()
t3.start()
t1.join()
t2.join()
t3.join()
```



```
In [7]: pip install thread6

Requirement already satisfied: thread6 in c:\users\lenovo\anaconda3\lib\site-packages (0.2.0)
Note: you may need to restart the kernel to use updated packages.

In [9]: import threading
        import random

        def ran_num():
            global nlist
            nlist=random.sample(range(1,100),10)
            print(nlist)

        def square():
            ns = [n**2 for n in nlist]
            print(ns)

        def cube():
            num_cube = [n**3 for n in nlist]
            print(num_cube)

        t1= threading.Thread(target=ran_num)
        t2 = threading.Thread(target=square)
        t3 = threading.Thread(target=cube)
        t1.start()
        t2.start()
        t3.start()
        t1.join()
        t2.join()
        t3.join()

[89, 29, 19, 20, 70, 99, 18, 15, 83, 28]
[7921, 841, 361, 400, 4900, 9801, 324, 225, 6889, 784]
[704969, 24389, 6859, 8000, 343000, 970299, 5832, 3375, 571787, 21952]
```

Outcomes:
CO1: Multithreading in Python

Questions:

a) **What are other ways to create threads in python? Give examples.**

Ans: 1.) Creation of thread by extending Thread Class:

Example:

```
from threading import *
class func_1(Thread):
    def run(self):
        for i in range(5):
            print('Thread_1')
t = func_1()
t.start()
for i in range(5):
    print('Main Thread')
```

2.) Creation of thread without extending Thread Class :

```
from threading import *
class thr:
    def func_1(self):
        for i in range(5):
            print('Thread_1')
obj = thr()
t1_obj = Thread(target=obj.func_1)
t1_obj.start()
for i in range(5):
    print('Main Thread')
```

b) How wait() and notify() methods can be used with lock in thread?

Ans: The wait() method releases the lock, and then blocks until another thread awakens it by calling notify() .Once awakened, wait() re-acquires the lock and returns.
The notify() method wakes up one of the threads waiting for the condition variable, if any are waiting.

Conclusion: I have understood the various concepts involved in multithreading and have implemented them.

References:

1. Daniel Arbuckle, Learning Python Testing, Packt Publishing, 1st Edition, 2014
2. Wesly J Chun, Core Python Applications Programming, O'Reilly, 3rd Edition, 2015
3. Wes McKinney, Python for Data Analysis, O'Reilly, 1st Edition, 2017
4. Albert Lukaszewsk, MySQL for Python, Packt Publishing, 1st Edition, 2010
5. Eric Chou, Mastering Python Networking, Packt Publishing, 2nd Edition, 2017