Experiment No. 2

Title: Program on Multithreading using Python

Batch: A2 **Roll No: 16010421052** **Experiment No.:2**

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)

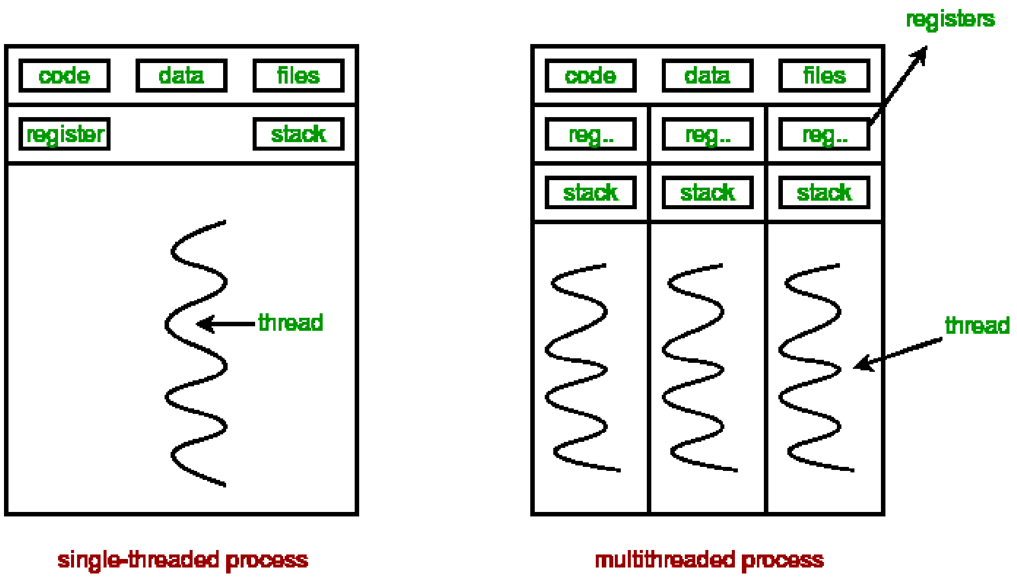
A thread is an entity within a process that can be scheduled for execution independently. Also, it is the smallest unit of processing that can be performed in an OS (Operating System).

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:



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

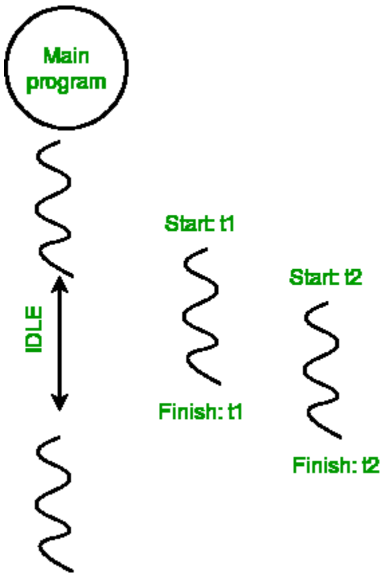
Once the threads start, the current program (you can think of it like a main thread) also keeps on executing. In order to stop execution of current program until a thread is complete, we use join method.

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

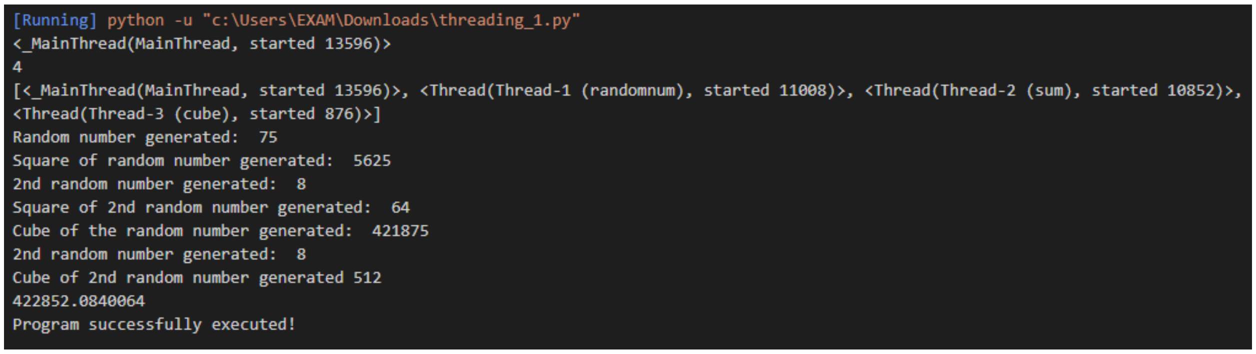
Result: (script and output)

Code 1:

```
import threading
import time
import random

def randomnum():

    time.sleep(2)
    global p
    p = random.randint(0, 100)

    print("Random number generated: ", p)

def square(x):

    time.sleep(4)
    n = p**2

    print("Square of random number generated: ", n)
    print("2nd random number generated: ", x)

    m = x**2

    print("Square of 2nd random number generated: ", m)

def cube(x):

    time.sleep(6)
    c = p**3

    print("Cube of the random number generated: ", c)
    print("2nd random number generated: ", x)

    m = x**3

    print("Cube of 2nd random number generated", m)

t1 = threading.Thread(target = randomnum)
x = random.randint(0, 100)
t2 = threading.Thread(target = square, args = (x,))
t3 = threading.Thread(target = cube, args = (x,))

t1.start()
t2.start()
t3.start()
```

```
print(threading.current_thread())
print(threading.active_count())
print(threading.enumerate())

t1.join()
t2.join()
t3.join()

print(time.perf_counter())
print("Program successfully executed!")
```

Output 1:



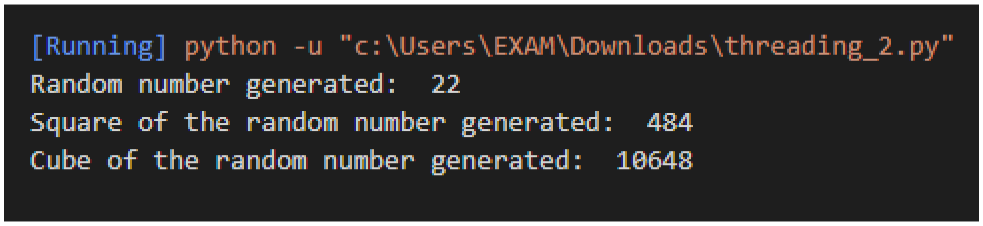
```
[Running] python -u "c:\Users\EXAM\Downloads\threading_1.py"
<_MainThread(MainThread, started 13596)>
4
[<_MainThread(MainThread, started 13596)>, <Thread(Thread-1 (randomnum), started 11008)>, <Thread(Thread-2 (sum), started 10852)>,
<Thread(Thread-3 (cube), started 876)>]
Random number generated:  75
Square of random number generated:  5625
2nd random number generated:  8
Square of 2nd random number generated:  64
Cube of the random number generated:  421875
2nd random number generated:  8
Cube of 2nd random number generated 512
422852.0840064
Program successfully executed!
```

Code 2:

```
import threading
import time
import random

class Thread1(threading.Thread):

    def __init__(self):
        super().__init__()

    def randomnum(self):

        time.sleep(2)
        global p
        p = random.randint(0, 100)

        print("Random number generated: ", p)

    def run(self):
        self.randomnum()
```

```python
class Thread2(threading.Thread):

    def __init__(self):
        super().__init__()

    def square(self):

        time.sleep(4)
        n = p**2

        print("Square of the random number generated: ", n)

    def run(self):
        self.square()

class Thread3(threading.Thread):

    def __init__(self):
        super().__init__()

    def cube(self):
        time.sleep(6)
        c = p**3

        print("Cube of the random number generated: ", c)

    def run(self):
        self.cube()

t1=Thread1()
t2=Thread2()
t3=Thread3()

t1.start()
t2.start()
t3.start()

t1.join()
t2.join()
t3.join()
```

Output 2:



```
[Running] python -u "c:\Users\EXAM\Downloads\threading_2.py"
Random number generated:  22
Square of the random number generated:  484
Cube of the random number generated:  10648
```

Outcomes:

CO1: Understanding the usage of multithreading.

Questions:

a) What are other ways to create threads in python? Give examples.
Ans:
A thread is a set of operations that are set for execution by a computer. Threading speeds up program execution by allowing us to run parts of a program concurrently. So threading is a way that we can execute multiple pieces of code at the same time.
There are two ways of creating threads in Python and those are; using a class or using a function.

1) Create a Thread without using an Explicit function:
By importing the module and creating the Thread class object separately we can easily create a thread. It is a function-oriented way of creating a thread.
Example-

```
# Import required modules
from threading import *
# Explicit function
def display() :
for i in range(10) :
   print("Child Thread")
# Driver Code
# Create object of thread class
Thread_obj = Thread(target=display)
# Executing child thread
Thread_obj.start()
# Executing main thread
for i in range(10):
print('Main Thread')
```

2) Create Thread by extending Thread Class :

In this method, we will extend the thread class from the threading module. This approach of creating a thread is also known as Object-Oriented Way.

Example-

```
# Import required module
from threading import *
# Extending Thread class
class Mythread(Thread):
    # Target function for thread
    def run(self):
        for i in range(10):
                print('Child Thread')
# Driver Code
# Creating thread class object
t = Mythread()
# Execution of target function
t.start()

# Executed by main thread
for i in range(10):
    print('Main Thread')
```

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

Ans:

The wait and notify methods are called on objects that are being used as locks. The lock is a shared communication point**:**

- When a thread that has a lock calls notifyAll on it, the other threads waiting on that same lock get notified. When a thread that has a lock calls notify on it, one of the threads waiting on that same lock gets notified.
- When a thread that has a lock calls wait on it, the thread releases the lock and goes dormant until either a) it receives a notification, or b) it just wakes up arbitrarily (the "spurious wakeup"); the waiting thread remains stuck in the call to wait until it wakes up due to one of these 2 reasons, then the thread has to re-acquire the lock before it can exit the wait method.

Conclusion: (Conclusion to be based on the objectives and outcomes achieved)

In this experiment I learnt the concepts of various threading methods and coded for the same. We learnt how threads can be created by functions as well as classes. We learnt about time and threading modules and their various functions. I learnt about locks and thread synchronization too.

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