Experiment No. 2

Title: Program on Multithreading using Python

Batch:A2 **Roll No:16010421063** **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:

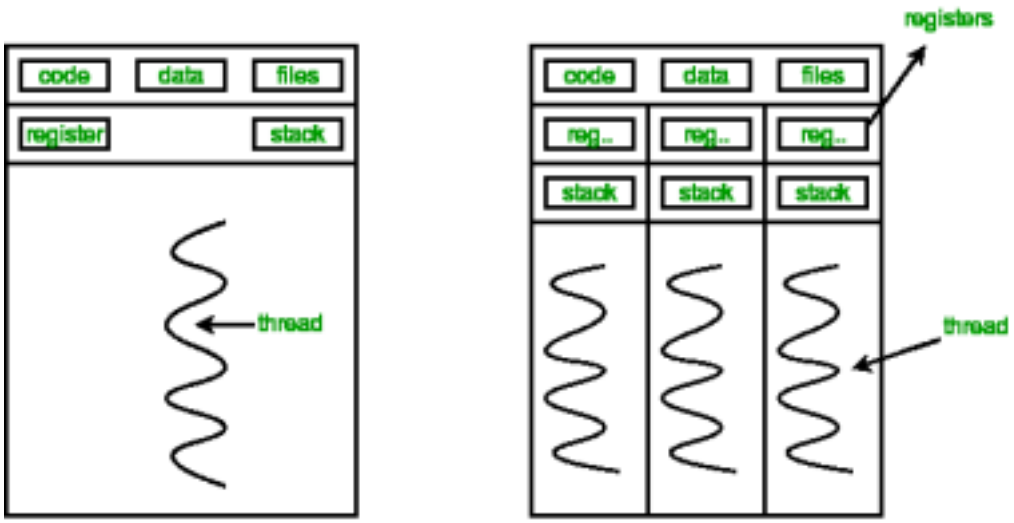
- An executable program.
- The associated data needed by the program (variables, work space, buffers, etc.)
- The execution context of the program (State of process)

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

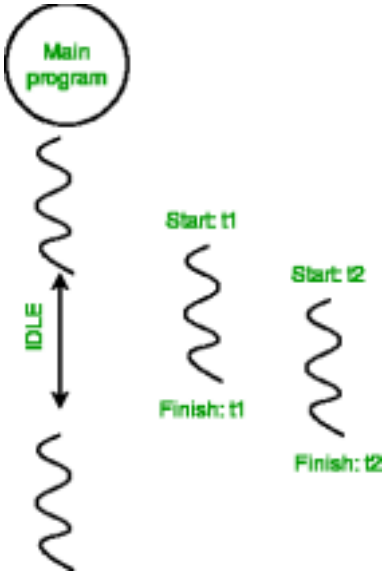
Once the threads start, the current program (you can think of it like a main thread) also keeps on executing. In order to stop execution of current program until a thread is complete, we use join method.

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 the current program will be 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)

```
import threading
import random
import time
class r_numThread(threading.Thread):
    def __init__(self):
        super().__init__()
    def random_number(self):
        global num
        num=random.randint(1,100)
        print(f"Number- {num}")
```

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


class squareThread(r_numThread):
    def __init__(self):
        super().__init__()

    def square(self,num):
        num_square = num**2
        print(f"Square-{num_square}\n")

    def run(self):
        self.square(num)


class cubeThread(r_numThread):
    def __init__(self):
        super().__init__()
    def cube(self,num):
        num_cube = num**3
        print(f"Cube- {num_cube}")
    def run(self):
        self.cube(num)

r_numthread = r_numThread()
square_thread = squareThread()
cube_thread = cubeThread()
r_numthread.start()
square_thread.start()
cube_thread.start()
print(threading.active_count())
print(threading.current_thread())
print(threading.enumerate())
print(time.perf_counter())
```

```
PS D:\SY> python -u "d:\SY\Python\test.py"
Number- 58
Square-3364
Cube- 195112

3
<_MainThread(MainThread, started 12168)>
[<_MainThread(MainThread, started 12168)>]
3253.9960213
PS D:\SY>
```

Outcomes: CO1. Understanding the usage of multithreading

Questions:

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

Function-based-

```
import threading

def test():

    print("test")

t1=threading.Thread(target=test)

t1.start()
```

```
PS D:\SY> python -u "d:\SY\Python\test.py"
test
PS D:\SY>
```

Class Based-

```
import threading



class TestThread(threading.Thread):

    def __init__(self):

        super().__init__()



    def test(self):

        print('Hello World')



    def run(self):

        self.test()

t1=TestThread()

t1.start()
```

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

thread?

Ans-

Wait()-This method releases the underlying lock, and then blocks until it is awakened by a notify() or notify_all() call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns

Notify()- This method wakes up at most n of the threads waiting for the condition variable; it is a no-op if no threads are waiting.

Conclusion: (Conclusion to be based on the objectives and outcomes achieved) :
Understood the concept of multi threading and implemented the code for the same using two methods 1) Function based 2) CLass based

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