Experiment No. 2

Title: Program on Multithreading using Python

Batch: B-1 **Roll No: 16010422234** **Name: Chandana Ramesh Galgali**

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)

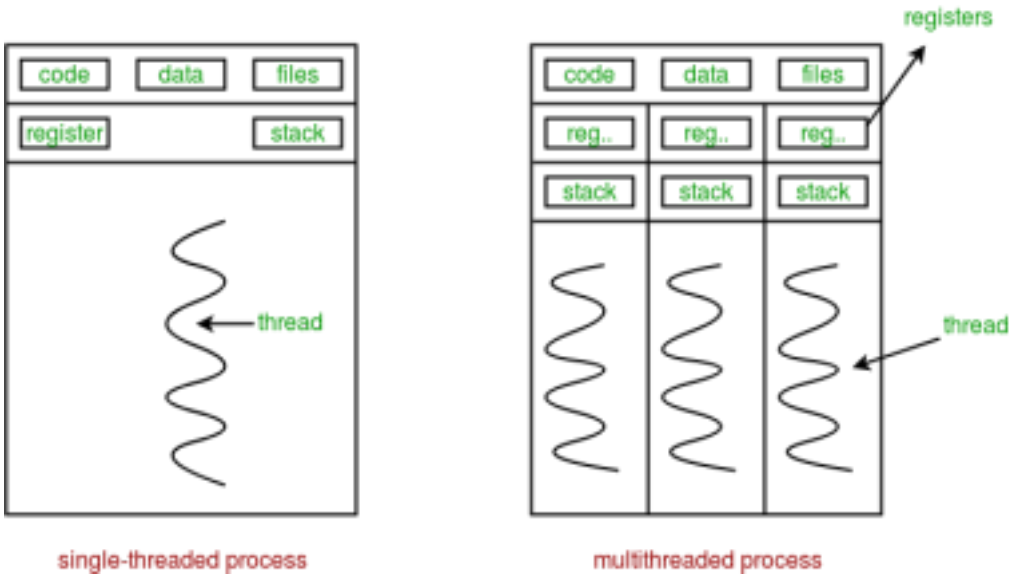
A thread is an entity within a process that can be scheduled for execution independently. Also, it is the smallest unit of processing that can be performed in an OS (Operating System).

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 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:



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 the Thread class from the threading module.

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

target: the function to be executed by thread

args: the arguments to be passed to the target function

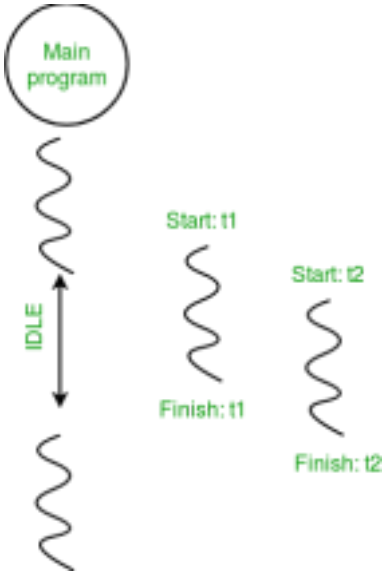
Once the threads start, the current program (you can think of it like a main thread) also keeps on executing. In order to stop execution of the current program until a thread is complete, we use the join method.

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 are executed.

Diagram below depicts the 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 access to shared resources can lead to race conditions.

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 the same initial values and produce the wrong result.

Hence there is a requirement of acquiring locks on shared resources 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 returns 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 unlock, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

If the 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 the 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 a random number. Thread two will compute the square of this number and thread 3 will compute the cube of this number.

Result: (script and output)

```
import random

import threading

def generate_random():

    x=random.randint(0,100)

    print("The number randomly generated is: ",x)

    return x

def square(num):

    print("Square of the number is: ",num*num)

def cube(num):
```

```
    print("Cube of the number is: ",num*num*num)

if __name__ == "__main__":

    x=generate_random()

    lock = threading.Lock()

    t1 = threading.Thread(target=generate_random, args=(lock,))

    t2 = threading.Thread(target=square, args=(x,))

    t3 = threading.Thread(target=cube, args=(x,))

    t2.start()

    t3.start()

    t2.join()

    t3.join()

    print("Done!")
```

Outcomes:

```
The number randomly generated is:  4
Square of the number is:  16
Cube of the number is:  64
Done!
```

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

Ans: In Python, there are multiple ways to create threads. Here are a few examples:

1)Using the threading module:

Python

```
import threading

def my_function():
  # Code to be executed in the thread

# Create a new thread
thread = threading.Thread(target=my_function)

# Start the thread
thread.start()
```

2)Subclassing the Thread class:

Python

```
import threading

class MyThread(threading.Thread):
  def run(self):
    # Code to be executed in the thread

# Create a new thread
thread = MyThread()

# Start the thread
thread.start()
```

3)Using the ThreadPoolExecutor from the concurrent.futures module:

```python
from concurrent.futures import ThreadPoolExecutor

def my_function():
  # Code to be executed in the thread

# Create a thread pool executor
executor = ThreadPoolExecutor()

# Submit the function to the executor
future = executor.submit(my_function)
```

4)Using the multiprocessing module for creating separate processes (which can run in parallel):

```python
import multiprocessing

def my_function():
  # Code to be executed in the process

# Create a new process
process = multiprocessing.Process(target=my_function)

# Start the process
process.start()
```

These are just a few examples of how you can create threads in Python. Each method has its own advantages and use cases, so you can choose the one that best fits your requirements.

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

Ans: In Python, the wait() and notify() methods are typically used in conjunction with a lock to implement thread synchronization and communication. These methods are part of the Condition class, which is a higher-level synchronization primitive provided by the threading module.

Here's an example of how wait() and notify() can be used with a lock in a thread:

Python

```python
import threading

# Create a lock and a condition variable
lock = threading.Lock()
condition = threading.Condition(lock)

# Shared data
data = []

# Consumer thread function
def consumer():
  with condition:
    # Wait until there is data available
    while not data:
      condition.wait()

    # Consume the data
    item = data.pop(0)
    print("Consumed:", item)

# Producer thread function
def producer():
  with condition:
    # Produce some data
    item = "Data"
    data.append(item)
    print("Produced:", item)

    # Notify the consumer that data is available
    condition.notify()
```

```
# Create and start the consumer thread
consumer_thread = threading.Thread(target=consumer)
consumer_thread.start()

# Create and start the producer thread
producer_thread = threading.Thread(target=producer)
producer_thread.start()
```

In this example, the consumer thread waits until there is data available in the data list. It uses the wait() method to release the lock and wait for a notification from the producer thread. The producer thread acquires the lock, produces some data, and then uses the notify() method to notify the consumer thread that data is available.

By using wait() and notify() with a lock and a condition variable, you can achieve thread synchronization and communication, allowing threads to coordinate their actions and share data safely.

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

The experiment on the implementation of multithreading in Python proved to be a valuable learning experience, equipping developers with the knowledge and tools to harness the potential of multithreading for building robust and efficient applications.

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