Experiment No. 12
Demonstrate the concept of Multi-threading
Date of Performance:
Date of Submission:

Experiment No. 12

Title: Demonstrate the concept of Multi-threading

Aim: To study and implement the concept of Multi-threading

Objective: To introduce the concept of Multi-threading in python

Theory:

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. 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)**:

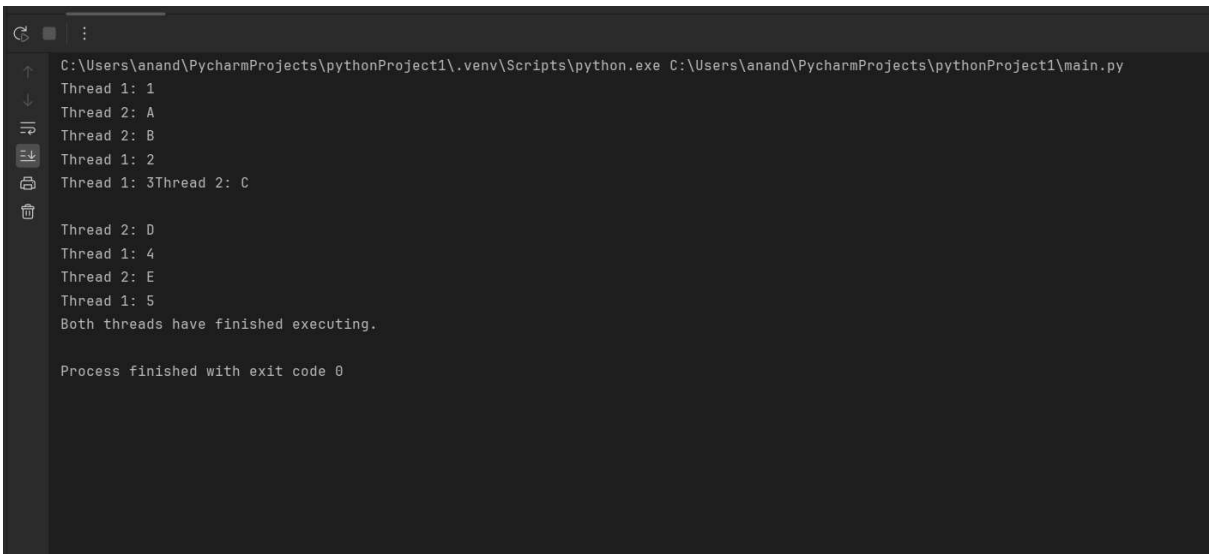
- **Thread Identifier:** Unique id (TID) is assigned to every new thread
- **Stack pointer:** Points to thread's stack in the process. Stack contains the local variables under thread's scope.
- **Program counter:** a register which stores the address of the instruction currently being executed by thread.
- **Thread state:** can be running, ready, waiting, start 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.

CODE:

```
import threading

import time

def print_numbers():

    for i in range(1, 6):
```

```
        print(f"Thread 1: {i}")
        time.sleep(1)
def print_letters():
    for char in ['A', 'B', 'C', 'D', 'E']:
        print(f"Thread 2: {char}")
        time.sleep(1)
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print("Both threads have finished executing.")
```

OUTPUT:



Conclusion: In conclusion, the implementation of multithreading in Python demonstrates its capability to execute concurrent tasks efficiently, thereby enhancing application performance. By leveraging the `threading` module, developers can create multiple threads that run simultaneously, facilitating parallel execution of tasks. Multithreading allows for improved resource utilization and responsiveness, especially in scenarios with I/O-bound or CPU-bound operations. Through this experiment, we've witnessed how multithreading can effectively manage concurrent tasks, enabling smoother execution and faster completion of operations. Embracing multithreading in Python opens doors to developing more responsive and scalable applications, empowering developers to tackle complex tasks with greater agility. With careful design and synchronization, multithreading proves to be a valuable tool for optimizing computational workflows and enhancing user experience. Overall, this experiment underscores the significance of multithreading in Python programming, offering a powerful mechanism for harnessing the full potential of modern computing architectures.