Created By: Hassan Sardar

SUBJECT: PARALLEL AND DISTRIBUTED COMPUTING

COMSATS UNIVERSITY ISLAMABAD

Lab Manual: Threading in Python with Output Visualization

Objective The objective of this lab is to learn how to create and manage threads in Python, and visualize the output of threaded programs to understand how multiple threads execute concurrently.

Prerequisites Basic knowledge of Python programming. Understanding of functions and loops in Python. Familiarity with the threading module in Python.

Tools Required Python 3.x installed on your machine. A code editor (VS Code, PyCharm, etc.) Matplotlib for visualization (use pip install matplotlib)

Introduction to Threading Threading is a concept that allows multiple threads to be executed in parallel, providing a way to multitask or run tasks concurrently. Python provides the threading module to create and manage threads. Threading is especially useful when tasks are IO-bound (e.g., reading from a file or waiting for network requests).

Lab Tasks Task 1: Basic Thread Creation In this task, we will create two threads and visualize their execution using print statements.

```
import threading  # Import the threading module to work with threads
import time       # Import the time module to add delays in thread execution

# Define a function 'task' that takes a thread name and a delay as arguments
def task(name, delay):
    # Loop 5 times
    for i in range(5):
        # Print the current task name and iteration count
      print(f'Task {name} - Count {i}')
        # Pause the thread execution for 'delay' seconds
        time.sleep(delay)

# Creating two thread objects, targeting the 'task' function
# 'args' is used to pass arguments to the 'task' function
thread1 =  threading.Thread(target=task, args=("Thread-1", 1))  # Create thread1 with 1-second delay
thread2 =threading.Thread(target=task, args=("Thread-2", 1.5))  # Create thread2 with 1.5-second delay

# Starting both threads, which begin execution in parallel
 thread1.start()  # Start thread1
thread2.start()  # Start thread2

# Wait for both threads to complete their execution
 thread1.join()  # Wait until thread1 completes
thread2.join()  # Wait until thread2 completes

# Print a message indicating both threads have finished executing
print("Both threads have finished executing")
```

```
Task Thread-1 - Count 0
Task Thread-2 - Count 0
Task Thread-1 - Count 1
Task Thread-2 - Count 1
Task Thread-1 - Count 2
Task Thread-1 - Count 3Task Thread-2 - Count 2

Task Thread-1 - Count 4
Task Thread-2 - Count 3
Task Thread-2 - Count 4
Both threads have finished executing
```

Explanation: We define a function task that prints a message 5 times with a delay. Two threads are created and started using threading.Thread. Both threads run concurrently and print messages. The join() function is used to wait until both threads finish execution.

Output Visualization: You will observe the printed output in the console, where both threads print their messages simultaneously, but with different delays.

Task 2: Thread Synchronization When multiple threads modify shared data, it may lead to race conditions. In this task, we'll use a thread lock to synchronize the threads.

Double-click (or enter) to edit

```
import threading  # Import the threading module to work with threads

# Shared variable
counter = 0  # Initialize a shared counter variable
lock = threading.Lock()  # Create a Lock object to control access to shared resources

# Define a function 'increment_counter' to increment the shared counter
def increment_counter(name):
    print("the name is "+name)
    global counter  # Declare 'counter' as a global variable to modify it inside the function
    # Loop 1000 times to increment the counter
    for  in range(1000):
        # Acquire the lock to ensure only one thread can access the shared counter at a time
        with lock:
            counter += 1  # Increment the counter
    # Print a message indicating the thread has finished updating the counter
    print(f'{name} finished updating counter.')

# Creating two threads targeting the 'increment_counter' function
# 'args' is used to pass the thread name as an argument to the function
thread1 =threading.Thread(target=increment_counter, args=("Thread-1",))  # Create thread1
thread2 = threading.Thread(target=increment_counter, args=("Thread-2",))  # Create thread2

# Starting both threads, which will run in parallel
  thread1.start()  # Start thread1
  thread2.start()  # Start thread2

# Wait for both threads to complete their execution
thread1.join()  # Wait until thread1 finishes
thread2.join()  # Wait until thread2 finishes

# Print the final value of the shared counter after both threads have finished executing
print(f"Final counter value: {counter}")
```

```
the name is Thread-1
Thread-1 finished updating counter.
the name is Thread-2
Thread-2 finished updating counter.
Final counter value: 2000
```

Explanation: Two threads increment a shared variable counter. The lock ensures that only one thread can access the shared data at a time, preventing race conditions.

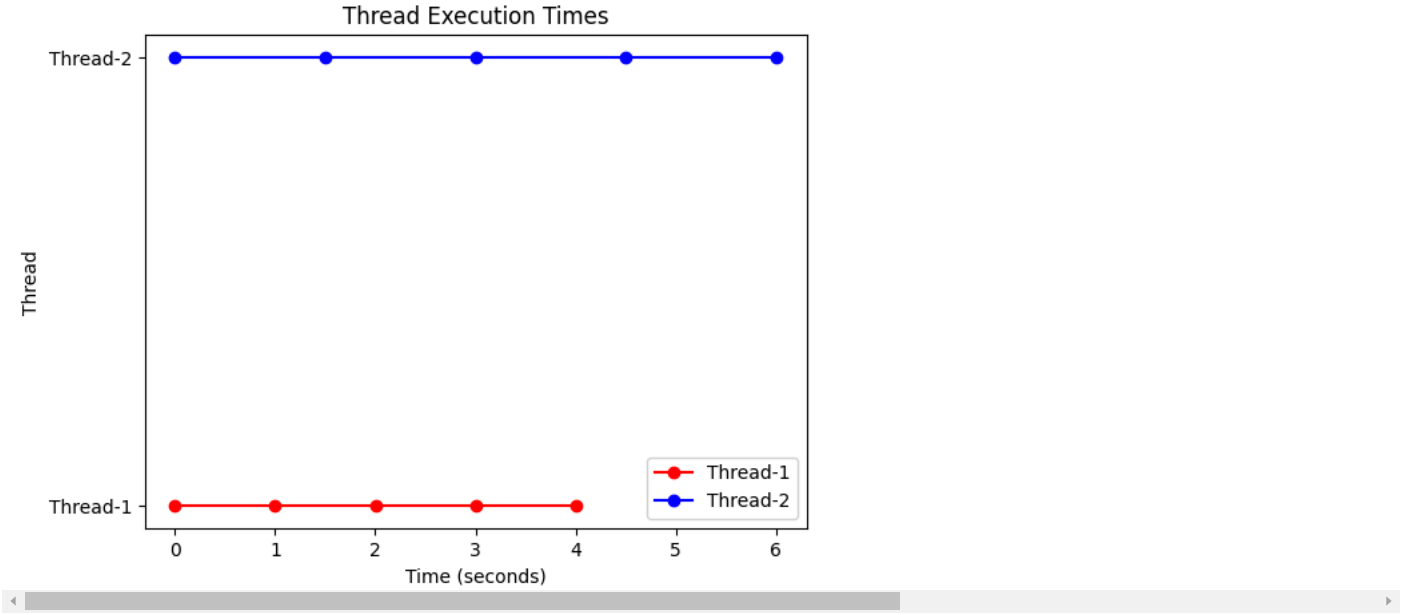
Output Visualization: Without locking, you may get unexpected results due to race conditions. With locking, the final counter value will be 2000, as both threads increment it properly without interference.

Task 3: Visualizing Thread Execution Using Matplotlib In this task, we will visualize the execution time of two threads using Matplotlib.

```
import threading  # Import the threading module to handle multi-threading
import time  # Import the time module for measuring execution time and delays
import matplotlib.pyplot as plt  # Import Matplotlib for plotting the execution times

# Lists to store execution times for plotting
thread1_times = []  # This list will store the execution times for thread1
thread2_times = []  # This list will store the execution times for thread2

# Function to simulate a task and record the time at each step
task_visualize(name, delay, times_list):
    start_time = time.time()  # Record the start time
    for i in range(5):  # Loop 5 times
        print(f'Task {name} - Count {i}')  # Print the current task name and count
```

```
        times_list.append(time.time() - start_time)  # Record the elapsed time for this iteration
        time.sleep(delay)  # Pause the thread for 'delay' seconds before the next iteration

# Creating two threads, each running the 'task_visualize' function
# Passing different delays (1 second and 1.5 seconds) to simulate varying thread execution speeds
thread1 = threading.Thread(target=task_visualize, args=("Thread-1", 1, thread1_times))  # Thread 1 with 1 second delay
thread2 = threading.Thread(target=task_visualize, args=("Thread-2", 1.5, thread2_times))  # Thread 2 with 1.5 seconds delay

# Starting the threads, so they begin executing concurrently
thread1.start()  # Start thread1
thread2.start()  # Start thread2

# Waiting for both threads to complete their execution before proceeding
thread1.join()  # Wait until thread1 finishes
thread2.join()  # Wait until thread2 finishes

# Plotting the results using Matplotlib
plt.plot(thread1_times, [1]*len(thread1_times), 'ro-', label="Thread-1")  # Plot thread1 execution times as red dots
plt.plot(thread2_times, [2]*len(thread2_times), 'bo-', label="Thread-2")  # Plot thread2 execution times as blue dots
plt.xlabel("Time (seconds)")  # Set the label for the x-axis
plt.ylabel("Thread")  # Set the label for the y-axis
plt.title("Thread Execution Times")  # Set the title of the plot
plt.yticks([1, 2], ['Thread-1', 'Thread-2'])  # Set y-ticks for the threads (1 for thread1, 2 for thread2)
plt.legend()  # Display the legend to differentiate between thread1 and thread2
plt.show()  # Display the plot
```

```
Task Thread-1 - Count 0
Task Thread-2 - Count 0
Task Thread-1 - Count 1
Task Thread-2 - Count 1
Task Thread-1 - Count 2
Task Thread-1 - Count 3
Task Thread-2 - Count 2
Task Thread-1 - Count 4
Task Thread-2 - Count 3
Task Thread-2 - Count 4
```



Explanation: Two threads are executed and their time is recorded after every print statement. The execution times are plotted using Matplotlib. The graph visualizes how each thread runs at different intervals due to different delays.

Output Visualization: A plot is generated, showing how Thread-1 and Thread-2 execute over time. Thread-1 runs more frequently (every 1 second), while Thread-2 has a longer delay (1.5 seconds).

Task 4: Using a Thread Pool In this task, we will use the ThreadPoolExecutor from the concurrent.futures module to run multiple threads efficiently.

```
from concurrent.futures import ThreadPoolExecutor  # Import ThreadPoolExecutor to manage a pool of threads
import time  # Import the time module for adding delays in task execution
```

```
# Define the task function to be executed by each thread in the pool
def task_pool(name):
print(f"Starting task {name}")  # Print a message indicating the start of the task
time.sleep(2)  # Simulate a task by sleeping (pausing execution) for 2 seconds
print(f"Finished task {name}")  # Print a message indicating the completion of the task

# Creating a thread pool with a maximum of 3 worker threads
with ThreadPoolExecutor(max_workers=3) as executor:
tasks = ["Task-1", "Task-2", "Task-3", "Task-4", "Task-5"]  # A list of tasks to be executed
executor.map(task_pool, tasks)  # Assign the tasks to the thread pool using the map() function, which distributes tasks to available threads

# Print message after all tasks have completed execution
print("All tasks are completed")
```

```
Starting task Task-1
Starting task Task-2
Starting task Task-3
Finished task Task-1
Starting task Task-4
Finished task Task-2
Starting task Task-5
Finished task Task-3
Finished task Task-4
Finished task Task-5
All tasks are completed
```

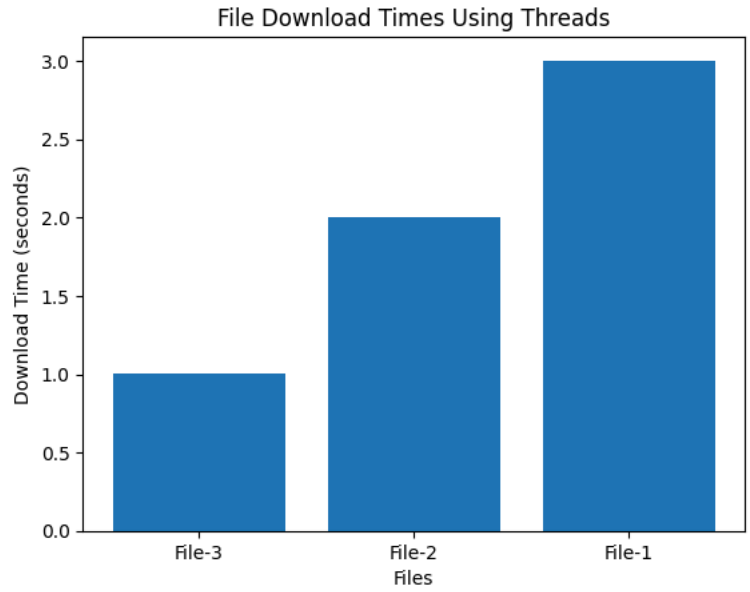
Explanation: A ThreadPoolExecutor is used to create a pool of threads. The executor.map() method runs the function on each item in the task list, distributing tasks across the threads.

Output Visualization: You will see tasks starting and finishing in parallel, with a maximum of 3 threads running at the same time.

Task 5: Visualizing Concurrent Downloading with Threads In this task, we will simulate concurrent downloading using threads and visualize the completion of each download.

```
import threading  # Import the threading module to handle multi-threading
import time  # Import the time module to simulate delays and record execution times
import matplotlib.pyplot as plt  # Import Matplotlib for plotting the download times

download_times = []  # List to store the download times for each file

# Function to simulate downloading a file
download_file(name, delay, times_list):
    start_time = time.time()  # Record the start time of the download
    print(f"Downloading {name}...")  # Print the name of the file being downloaded
    time.sleep(delay)  # Simulate the download time by pausing execution for the given delay
    end_time = time.time()  # Record the end time of the download
    times_list.append((name, end_time - start_time))  # Append the file name and download duration to the list
    print(f"{name} download completed.")  # Print a message when the download is complete

# Creating a list of files to download with corresponding delays (in seconds)
files = [("File-1", 3), ("File-2", 2), ("File-3", 1)]
threads = []  # List to hold the thread objects

# Loop through each file and create a thread for each download
for file, delay in files:
    # Create a new thread for each file, passing the file name, delay, and the list to store times
    threads = threading.Thread(target=download_file, args=(file, delay, download_times))
    threads.append(thread)  # Add the thread to the threads list
    thread.start()  # Start the thread (this begins the download)

# Wait for all threads to finish their execution
for threads in threads:
    thread.join()  # Join ensures that the main thread waits for each thread to complete

# Extract the file names and times from the download_times list
files, times = zip(*download_times)

# Plotting the results using a bar chart
plt.bar(files, times)  # Create a bar chart with file names on the x-axis and download times on the y-axis
plt.xlabel("Files")  # Label for the x-axis
plt.ylabel("Download Time (seconds)")  # Label for the y-axis
plt.title("File Download Times Using Threads")  # Title for the chart
plt.show()  # Display the plot
```

```
Downloading File-1...
Downloading File-2...
Downloading File-3...
File-3 download completed.
File-2 download completed.
File-1 download completed.
```



Explanation: Each thread simulates downloading a file by sleeping for a given time. The times for each download are visualized in a bar chart, showing how threads can download files concurrently.

Output Visualization: A bar chart is generated that displays the download time for each file, simulating how threads handle concurrent downloads.

Conclusion In this lab, you have learned how to create, manage, and synchronize threads in Python. You also visualized the execution of threaded tasks using Matplotlib. The skills gained here will help you in building more efficient, concurrent programs.