Report
Assignment 5: Concurrency Control

Name: Vivek Sapkal   Roll No.: B22AI066

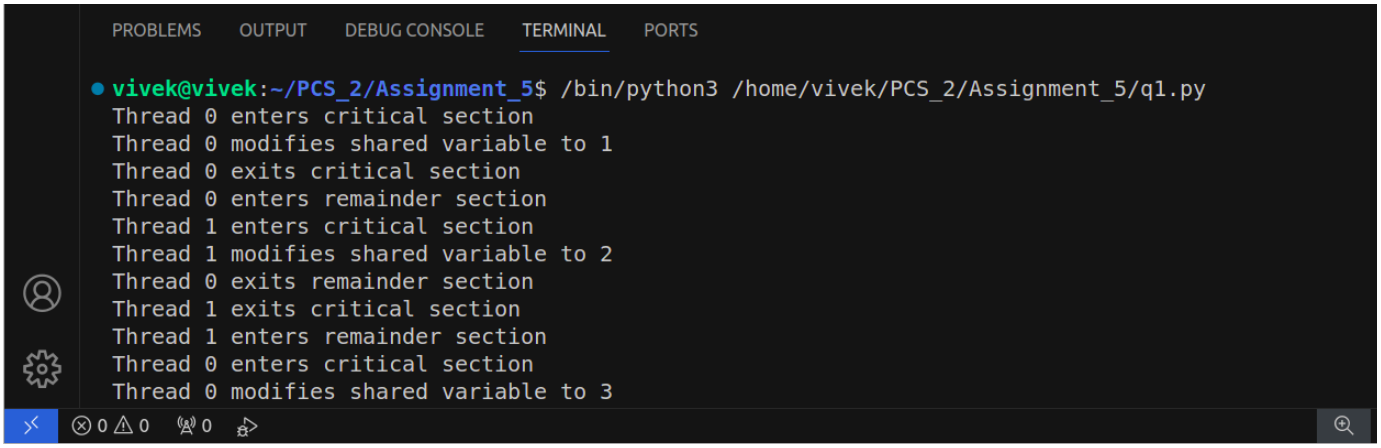
Q. 1)

➢ **Code:**

```
import threading
import time

NUM_THREADS = 2
NUM_EXECUTIONS = 5

flag = [False] * NUM_THREADS
turn = 0

# Global variable representing the shared resource
shared_variable = 0

def critical_section(thread_id):
   global shared_variable

   other_thread = 1 - thread_id

   for _ in range(NUM_EXECUTIONS):
       flag[thread_id] = True
       turn = other_thread

       # Busy wait until it's this thread's turn and the other thread
isn't ready
       while flag[other_thread] and turn == other_thread:
           pass
```

```
        # Critical section: accessing and modifying the shared resource
(here the global variable)
        print(f"Thread {thread_id} enters critical section")
        shared_variable += 1
        print(f"Thread {thread_id} modifies shared variable to
{shared_variable}")

        # Simulating some work in the critical section
        time.sleep(1)

        # exiting the critical section
        print(f"Thread {thread_id} exits critical section")
        flag[thread_id] = False

        # Remainder section
        print(f"Thread {thread_id} enters remainder section")
        # Simulating some work in the remainder section
        time.sleep(0.5)
        print(f"Thread {thread_id} exits remainder section")

if __name__ == "__main__":
    # Creating threads
    threads = []
    for i in range(NUM_THREADS):
        t = threading.Thread(target=critical_section, args=(i,))
        threads.append(t)

    # Starting threads
    for t in threads:
        t.start()

    # Joining threads
    for t in threads:
        t.join()

    print("Final value of shared variable:", shared_variable)
```

➢ Output:



➢ Readme:

- Overview
    - This Python program demonstrates the implementation of a concurrent program using Peterson's algorithm for mutual exclusion. The code simulates multiple threads (workers) accessing and modifying a shared resource while ensuring exclusive access to critical sections.

- Implementation Details
- Global Variables
    - NUM_THREADS: Number of worker threads in the simulation (set to 2 in this example).
    - flag: An array indicating whether a thread is ready to enter its critical section.
    - turn: Indicates whose turn it is to enter the critical section.
    - shared_variable: A global variable representing the shared resource.

- Functions
- critical_section(thread_id)
    - Each thread executes a critical section that uses Peterson's algorithm for mutual exclusion.
    - The critical section involves accessing and modifying the shared resource (shared_variable).
    - After the critical section, the thread enters a remainder section to simulate additional non-critical work.

- Running the Program
    - The program creates two threads and starts their execution.
    - The threads execute critical and remainder sections a certain number of times.
    - Threads use Peterson's algorithm to ensure mutual exclusion in the critical section.
    - The final value of the shared variable is printed after all threads complete their execution.

Q. 2)

➢ **Code:**

```
import threading
import time
import random

# Define a semaphore with an initial count of 3
semaphore = threading.Semaphore(3)

# Shared resource
shared_resource = []

# Function to simulate accessing a shared resource
def access_shared_resource(thread_id):
   global shared_resource
```

```
    # Simulate some work before accessing the resource
    time.sleep(random.uniform(0.5, 1.5))

    # Acquire the semaphore
    semaphore.acquire()

    print(f"Thread {thread_id}: Acquired the semaphore")

    # Simulate accessing the shared resource (e.g., adding data to a
shared_resource)
    data = f"Data from Thread {thread_id}"
    print(f"Thread {thread_id}: Accessing shared resource (Adding {data} to
shared_resource)")
    shared_resource.append(data)

    # Simulate some work while accessing the resource
    time.sleep(random.uniform(1, 2))

    # Release the semaphore
    print(f"Thread {thread_id}: Releasing the semaphore")
    semaphore.release()

# Create multiple threads
threads = []
for i in range(10):
    thread = threading.Thread(target=access_shared_resource, args=(i,))
    threads.append(thread)
    thread.start()

# Wait for all threads to complete
for thread in threads:
    thread.join()

print("All threads have finished execution")
print("Final shared_resource contents:", shared_resource)
```

➢ Output:

```
vivek@vivek:~/PCS_2/Assignment_5$ /bin/python3 /home/vivek/PCS_2/Assignment_5/q2.py
Thread 3: Acquired the semaphore
Thread 3: Accessing shared resource (Adding Data from Thread 3 to shared_resource)
Thread 2: Acquired the semaphore
Thread 2: Accessing shared resource (Adding Data from Thread 2 to shared_resource)
Thread 7: Acquired the semaphore
Thread 7: Accessing shared resource (Adding Data from Thread 7 to shared_resource)
Thread 3: Releasing the semaphore
Thread 0: Acquired the semaphore
Thread 0: Accessing shared resource (Adding Data from Thread 0 to shared_resource)
Thread 7: Releasing the semaphore
Thread 8: Acquired the semaphore
Thread 8: Accessing shared resource (Adding Data from Thread 8 to shared_resource)
Thread 2: Releasing the semaphore
Thread 4: Acquired the semaphore
Thread 4: Accessing shared resource (Adding Data from Thread 4 to shared_resource)
Thread 0: Releasing the semaphore
Thread 5: Acquired the semaphore
Thread 5: Accessing shared resource (Adding Data from Thread 5 to shared_resource)
Thread 8: Releasing the semaphore
Thread 9: Acquired the semaphore
Thread 9: Accessing shared resource (Adding Data from Thread 9 to shared_resource)
Thread 4: Releasing the semaphore
Thread 6: Acquired the semaphore
Thread 6: Accessing shared resource (Adding Data from Thread 6 to shared_resource)
Thread 9: Releasing the semaphore
Thread 1: Acquired the semaphore
Thread 1: Accessing shared resource (Adding Data from Thread 1 to shared_resource)
Thread 5: Releasing the semaphore
Thread 6: Releasing the semaphore
Thread 1: Releasing the semaphore
All threads have finished execution
Final shared_resource contents: ['Data from Thread 3', 'Data from Thread 2', 'Data from Thread 7', 'Data from Thread 0', 'Data from Thread 8',
'Data from Thread 4', 'Data from Thread 5', 'Data from Thread 9', 'Data from Thread 6', 'Data from Thread 1']
vivek@vivek:~/PCS_2/Assignment_5$
0 ⚠ 0   0        Ln 15, Col 55   Spaces: 4   UTF-8   LF   {} Python   3.8.10 64-bit
```

➢ Readme:

- # Introduction

- This Python program demonstrates concurrent access to a shared resource by multiple threads using semaphores. Semaphores are synchronization primitives that help control access to shared resources in concurrent programming.

- # Program Description

- The program simulates a scenario where multiple threads access a shared resource, which in this case is represented by a simulated database. Each thread performs the following steps:

- Performs initial work before accessing the resource.
- Acquires a semaphore to control access to the shared resource.
- Simulates accessing the shared resource (e.g., adding data to a database).
- Releases the semaphore to allow other threads to access the resource.

Implementation Details

- The program is implemented in Python using the threading module for creating and managing threads.
- A counting semaphore is used to control access to the shared resource. The semaphore is initialized with an initial count of 3, allowing up to three threads to access the resource simultaneously.
- Threads are created to simulate concurrent access to the shared resource. Each thread executes a function (access_shared_resource) that performs the specified steps.
- Random delays are introduced to simulate real-world scenarios where operations take varying amounts of time.