Experiment -3 (OS Lab)
Name: Veeransh Shah
Reg Id: 221070063

Aim:
Implement semaphore for the process synchronisation.

Theory:

Semaphore:
A Semaphore is a synchronisation primitive used to control access to a common resource by multiple processes or threads in a concurrent system. It maintains a count, which represents the number of units of the resource available.

Operations:
A semaphore supports two fundamental operations”

1. Acquire:
    - When a process/thread wants to use the shared resource, it must first acquire the semaphore.
    - If the semaphore count is greater than zero, the process/thread decrements the count and proceeds.
    - If the count is zero, the process/thread may block(wait) until the count becomes greater than zero, indicating that the resource available

2. Release:
    - When a process/thread finishes using the shared resource, it must release the semaphore.
    - This operation increments the semaphore count, indicating that the resource is now available for use by another process/thread.

Example:

Consider a scenario where multiple threads need access to a shared resource, but the resource can only be accessed by a limited number of threads simultaneously. We can use a semaphore to control access to this resource.

In the provided Python code:

- The `Semaphore` class encapsulates the semaphore functionality.
- The `acquire()` method is called when a thread wants to access the shared resource. It blocks the thread until the semaphore count becomes positive, and then decrements the count.
- The `release()` method is called when a thread finishes using the shared resource. It increments the semaphore count, potentially allowing another waiting thread to acquire the resource.

- In the `worker` function, threads attempt to acquire the semaphore before performing their work and release it when finished.
- By initialising the semaphore with a count of 1 (`Semaphore(1)`), we ensure that only one thread can access the shared resource at a time, effectively providing mutual exclusion.

Benefits:

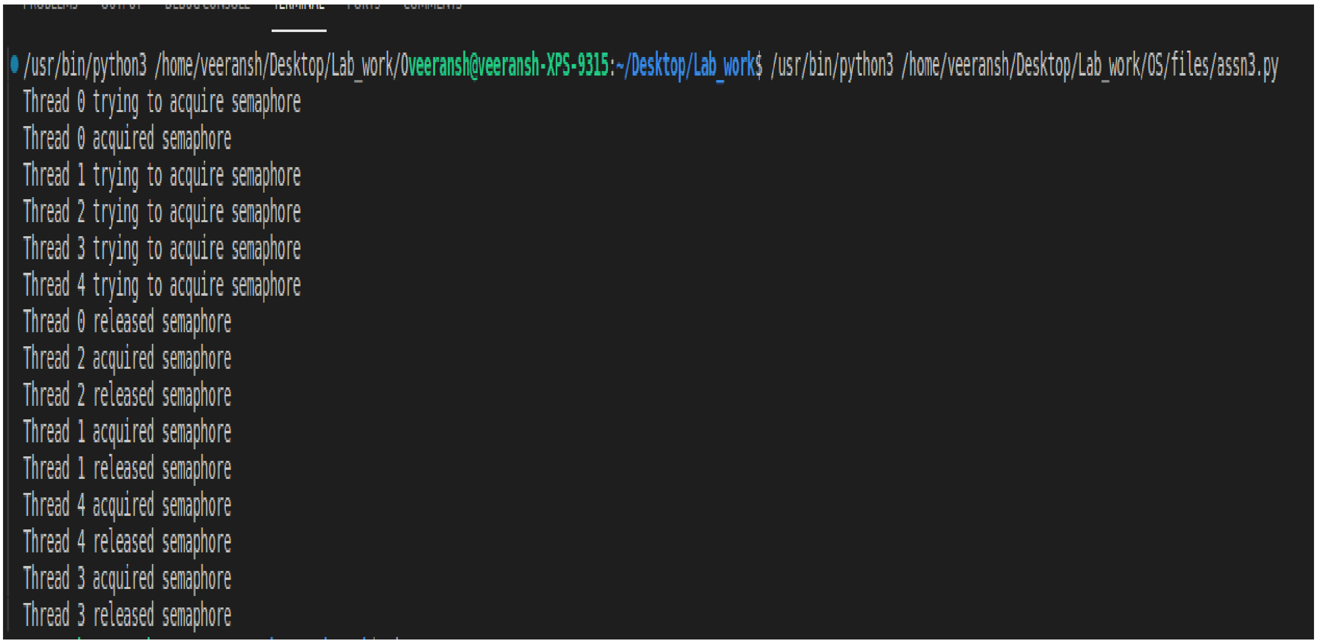
- Semaphores facilitate coordination and synchronisation between concurrent processes/threads.
- They help prevent race conditions and ensure orderly access to shared resources.
- Semaphores can be used to solve various synchronisation problems, such as producer-consumer, readers-writers, and dining philosophers.

In summary, semaphores are a powerful synchronisation mechanism that helps manage access to shared resources in concurrent systems, promoting thread safety and preventing race conditions. The provided Python code demonstrates a simple implementation of semaphores and their usage in a multi-threaded environment.

Code:

```
import threading

class Semaphore:
    def __init__(self, initial=1):
        self.lock = threading.Lock()
        self.value = initial

    def acquire(self):
        with self.lock:
            while self.value == 0:
                self.lock.release()
                self.lock.acquire()
            self.value -= 1

    def release(self):
        with self.lock:
            self.value += 1

# Example usage:
import time

def worker(semaphore, id):
    print(f"Thread {id} trying to acquire semaphore")
    semaphore.acquire()
    print(f"Thread {id} acquired semaphore")
    time.sleep(2)   # Simulate some work being done
    semaphore.release()
    print(f"Thread {id} released semaphore")

semaphore = Semaphore()   # Initialize semaphore
threads = []

for i in range(5):
    t = threading.Thread(target=worker, args=(semaphore, i))
    threads.append(t)
    t.start()

for t in threads:
    t.join()
```

Output:



```
● /usr/bin/python3 /home/veeransh/Desktop/Lab_work/0veeransh@veeransh-XPS-9315:~/Desktop/Lab_work$ /usr/bin/python3 /home/veeransh/Desktop/Lab_work/OS/files/assn3.py
  Thread 0 trying to acquire semaphore
  Thread 0 acquired semaphore
  Thread 1 trying to acquire semaphore
  Thread 2 trying to acquire semaphore
  Thread 3 trying to acquire semaphore
  Thread 4 trying to acquire semaphore
  Thread 0 released semaphore
  Thread 2 acquired semaphore
  Thread 2 released semaphore
  Thread 1 acquired semaphore
  Thread 1 released semaphore
  Thread 4 acquired semaphore
  Thread 4 released semaphore
  Thread 3 acquired semaphore
  Thread 3 released semaphore
```

Conclusion:
In conclusion, semaphores are a fundamental synchronisation primitive used in concurrent programming to control access to shared resources among multiple processes or threads. They operate by maintaining a count that represents the availability of the resource, and processes or threads must acquire and release the semaphore to access the resource.