Name : Ajay Shitkar
Roll No: 58
Experiment No.9
Implement Distributed shared Memory
Date of Performance: 01/04/24
Date of Submission: 05/04/24

Aim: To implement distributed shared memory

Objective: Implement distributed shared memory using Semaphores

Theory:

A semaphore lock on the shared memory buffer reference allows processes accessing the shared memory to prevent a daemon from writing to the memory segment currently being accessed.

Semaphores are a synchronization mechanism used to coordinate the activities of multiple processes in a computer system. They are used to enforce mutual exclusion, avoid race conditions and implement synchronization between processes.

Semaphores provide two operations: wait (P) and signal (V). The wait operation decrements the value of the semaphore, and the signal operation increments the value of the semaphore.

The algorithm is as follows:

```
                    semaphore s = 1;

        Process 1 :                          Process 2 :
while (True)                         while (True)
{   nc1: /* non critical             {   nc2: /* non critical
           section */ ;                        section */
         wait(s);                            wait(s);
  crit1: /* critical section */ ;     crit2: /* critical section */
         signal(s);                          signal(s)
}                                    }
```

Code and output:

Code :

```
// java program to demonstrate
// use of semaphores Locks
import java.util.concurrent.*;
//A shared resource/class. class
Shared

{
static int count = 0;

}
```

```
class MyThread extends Thread

{

        Semaphore sem; String threadName; public
        MyThread(Semaphore sem, String threadName)
        { super(threadName); this.sem = sem;
                this.threadName = threadName;
        }


        @Override public
        void run() {


                // run by thread A

                if(this.getName().equals("A"))

                {

                        System.out.println("Starting " + threadName);

                        try

                        {

                                // First, get a permit.

                                System.out.println(threadName + " is waiting for a permit.");



                                // acquiring the lock
                                sem.acquire();


                                System.out.println(threadName + " gets a permit.");



                                // Now, accessing the shared resource.

                                // other waiting threads will wait, until this
```

```
                    // thread release the lock
                    for(int i=0; i < 5; i++)
                    {
                            Shared.count++;
                            System.out.println(threadName + ": " + Shared.count);

                            // Now, allowing a context switch -- if possible.
                            // for thread B to execute
                            Thread.sleep(10);
                    }
            } catch (InterruptedException exc) {
                            System.out.println(exc);
                    }

                    // Release the permit.
                    System.out.println(threadName + " releases the permit.");
                    sem.release();
    }

    // run by thread B
    else
    {
            System.out.println("Starting " + threadName);
            try
            {
                    // First, get a permit.
                    System.out.println(threadName + " is waiting for a permit.");
```

```
                    // acquiring the lock
                    sem.acquire();

                    System.out.println(threadName + " gets a permit.");

                    // Now, accessing the shared resource.
                    // other waiting threads will wait, until this
                    // thread release the lock
                    for(int i=0; i < 5; i++)
                    {
                        Shared.count--;
                        System.out.println(threadName + ": " + Shared.count);

                        // Now, allowing a context switch -- if possible.
                        // for thread A to execute
                        Thread.sleep(10);
                    }
                } catch (InterruptedException exc) {
                        System.out.println(exc);
                    }
                    // Release the permit.
                    System.out.println(threadName + " releases the permit.");
                    sem.release();
            }
        }
}
```

```
// Driver class public class
SemaphoreDemo { public
static void main(String
args[]) throws
InterruptedException
        {
                // creating a Semaphore object

                // with number of permits 1

                Semaphore sem = new Semaphore(1);



                // creating two threads with name A and B

                // Note that thread A will increment the count

                // and thread B will decrement the count

                MyThread mt1 = new MyThread(sem, "A");

                MyThread mt2 = new MyThread(sem, "B");



                // stating threads A and B

                mt1.start();
                mt2.start();


                // waiting for threads A and B

                mt1.join();
                mt2.join();


                // count will always remain 0 after

                // both threads will complete their execution

                System.out.println("count: " + Shared.count);

        }

}
```

Output :

Starting A

Starting B A is waiting

for a permit.

B is waiting for a permit.

A gets a permit.

A: 1

A: 2

A: 3

A: 4

A: 5

A releases the permit.

B gets a permit.

B: 4

B: 3

B: 2

B: 1

B: 0

B releases the permit.

count: 0

Conclusion: The experiment showcases a distributed shared memory implementation employing semaphores to uphold mutual exclusion efficiently. Semaphores play a pivotal role in synchronizing thread access to shared resources, permitting only one thread at a time. Before accessing shared memory, threads acquire semaphore permits, thereby preventing simultaneous modifications and upholding data consistency. Through semaphore operations, exclusive access to shared variables is guaranteed, mitigating race conditions effectively. Semaphores emerge as robust mechanisms for enforcing mutual exclusion and coordinating resource access in concurrent programming scenarios. This methodology effectively safeguards data integrity by confining shared memory access to a single thread at any given moment. In summary, the experiment underscores how semaphores reliably ensure mutual exclusion, a vital element for coherence in multi-threaded environments, within a succinct and efficient framework.