**Thread concept: clone, threads of java**

**Subject - Unix Operating System**

**Name – Hemant Sharma**

**PRN – 22610001 Class – TYIT**

**Assignment No – 4(c)**

**Title-** Write program to synchronize threads using construct –monitor/serialize/semaphore of Java (In java only).

**Objectives:**

1. To learn about threading in Linux/Unix and Java and difference between them.
2. Use of system call/library to write effective programs.

**Theory:**

A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource.

Thus, to access the resource, a thread must be granted a permit from the semaphore.

Working of semaphore: In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit.

* If the semaphore’s count is greater than zero, then the thread acquires a permit, which causes the semaphore’s count to be decremented.
* Otherwise, the thread will be blocked until a permit can be acquired.
* When the thread no longer needs an access to the shared resource, it releases the permit, which causes the semaphore’s count to be incremented.
* If there is another thread waiting for a permit, then that thread will acquire a permit at that time. Java provide Semaphore class in java.util.concurrent package that implements this mechanism, so you don’t have to implement your own semaphores.

**Program:**

import java.util.concurrent.Semaphore;

// Shared resource class

class SharedResource {

    private int count = 0;

    // Monitor-based synchronization using synchronized keyword

    public synchronized void increment() {

        count++;

        System.out.println(Thread.currentThread().getName() + " - Count: " + count);

    }

    // Getter method (for checking final count)

    public int getCount() {

        return count;

    }

}

// Thread class using Monitor (synchronized)

class MonitorThread extends Thread {

    private final SharedResource resource;

    public MonitorThread(SharedResource resource) {

        this.resource = resource;

    }

    public void run() {

        for (int i = 0; i < 5; i++) {

            resource.increment();

            try {

                Thread.sleep(100);

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

}

// Semaphore-based synchronization

class SemaphoreThread extends Thread {

    private final Semaphore semaphore;

    private final SharedResource resource;

    public SemaphoreThread(Semaphore semaphore, SharedResource resource) {

        this.semaphore = semaphore;

        this.resource = resource;

    }

    public void run() {

        try {

            semaphore.acquire(); // Acquire semaphore lock

            for (int i = 0; i < 5; i++) {

                resource.increment();

                Thread.sleep(100);

            }

            semaphore.release(); // Release semaphore lock

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

}

// Main class

public class ThreadSynchronizationDemo {

    public static void main(String[] args) {

        SharedResource sharedResource = new SharedResource();

        Semaphore semaphore = new Semaphore(1); // Allows 1 thread at a time

        // Using synchronized (Monitor)

        System.out.println("Using Monitor (synchronized):");

        MonitorThread t1 = new MonitorThread(sharedResource);

        MonitorThread t2 = new MonitorThread(sharedResource);

        t1.start();

        t2.start();

        // Wait for monitor-based threads to finish

        try {

            t1.join();

            t2.join();

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

        // Using Semaphore

        System.out.println("\nUsing Semaphore:");

        SemaphoreThread t3 = new SemaphoreThread(semaphore, sharedResource);

        SemaphoreThread t4 = new SemaphoreThread(semaphore, sharedResource);

        t3.start();

        t4.start();

        // Wait for semaphore-based threads to finish

        try {

            t3.join();

            t4.join();

        } catch (InterruptedException e) {

            e.printStackTrace();

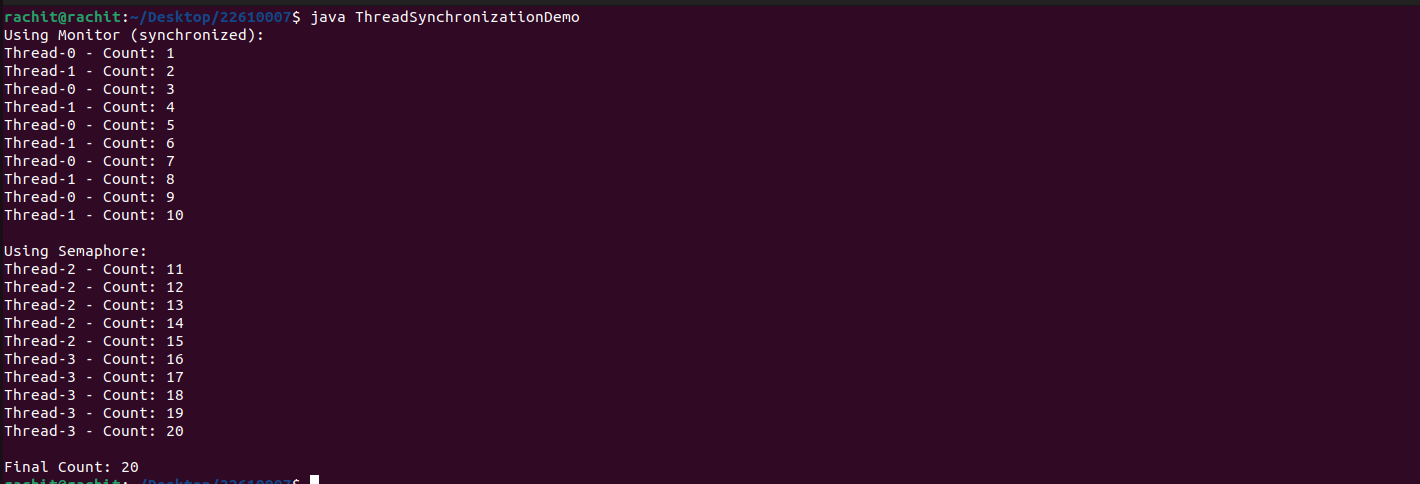
        }

        System.out.println("\nFinal Count: " + sharedResource.getCount());

    }

}

**Output:**

****

**Conclusion:**

Synchronization of multiple threads using semaphore to let threads work synchronously to produce desirable outputs learned and implemented in Java.