CONTEXT

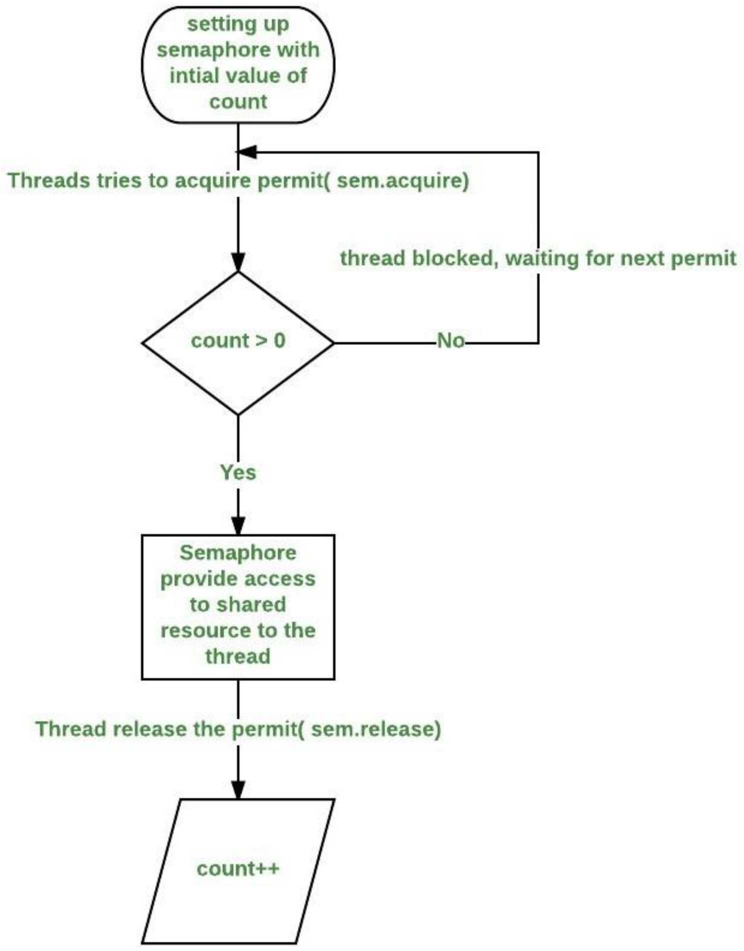
INTRODUCTION:

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:

In general, to use a semaphore, the thread that wants access to the shared

resource tries to acquire a permit.


setting up semaphore with intial value of count
Threads tries to acquire permit(sem.acquire)
thread blocked, waiting for next permit
count > 0
No
Yes
Semaphore provide access to shared resource to the thread
Thread release the permit(sem.release)
count++

- 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.

DESCRIPTION:

Semaphores in Process Synchronization

Semaphore was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a simple integer value, which is known as a semaphore. Semaphore is simply a variable which is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

Semaphores are of two types:

1. **Binary Semaphore –** This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.
2. **Counting Semaphore –** Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Some point regarding P and V operation

1. P operation is also called wait, sleep or down operation and V operation is also called signal, wake-up or up operation.
2. Both operations are atomic and semaphore(s) is always initialized to one.Here atomic means that variable on which read, modify and update happens at the same time/moment with no pre-emption i.e. in between read, modify and update no other operation is performed that may change the variable.
3. A critical section is surrounded by both operations to implement process synchronization.See below image.critical section of Process P is in between P and V operation.

ADVANTAGES:

- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

DISADVANTAGES:

- Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
- Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

IMPLEMENTATION:

Implementation of binary semaphores:

```
struct semaphore {
    enum value(0, 1);

    // q contains all Process Control Blocks (PCBs)
    // corresponding to processes got blocked
    // while performing down operation.
    Queue<process> q;

} P(semaphore s)
{
    if (s.value == 1) {
        s.value = 0;
    }
    else {
        // add the process to the waiting queue
        q.push(P)
            sleep();
    }
}
V(Semaphore s)
{
    if (s.q is empty) {
        s.value = 1;
    }
    else {

        // select a process from waiting queue
        q.pop();
        wakeup();
    }
}
```

The description above is for binary semaphore which can take only two values 0 and 1 and ensure the mutual exclusion. There is one other type of semaphore called counting semaphore which can take values greater than one.

Limitations

1. One of the biggest limitation of semaphore is priority inversion.
2. Deadlock, suppose a process is trying to wake up another process which is not in sleep state.Therefore a deadlock may block indefinitely.
3. The operating system has to keep track of all calls to wait and to signal the semaphore.

Problem in this implementation of semaphore

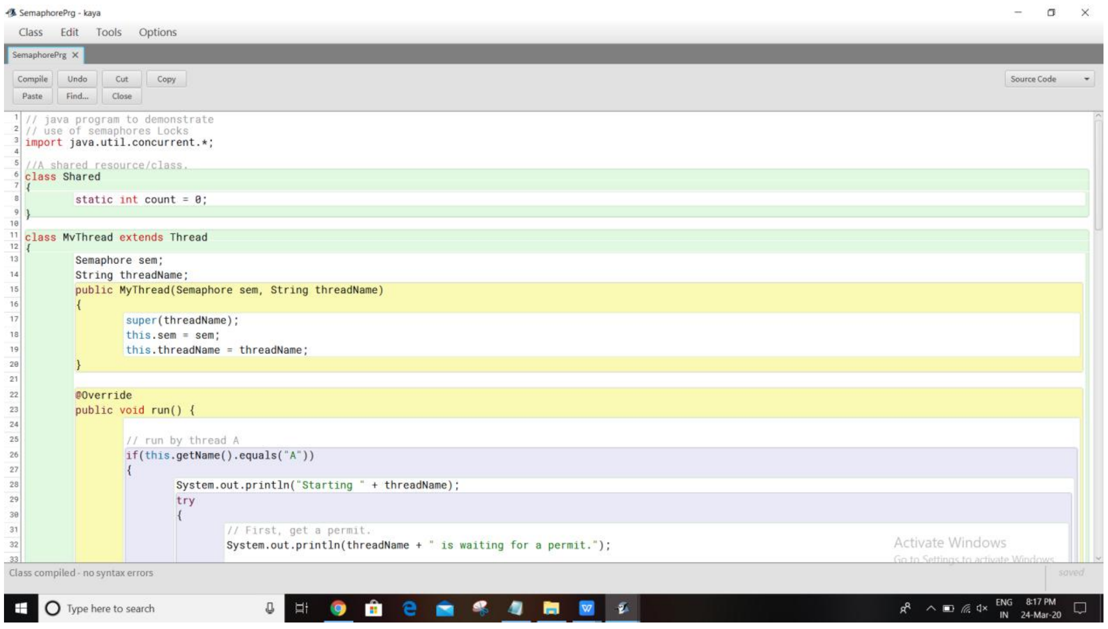
Whenever any process waits then it continuously checks for semaphore value (look at this line while (s==0); in P operation) and waste CPU cycle. To avoid this another implementation is provided below.

Implementation of counting semaphore

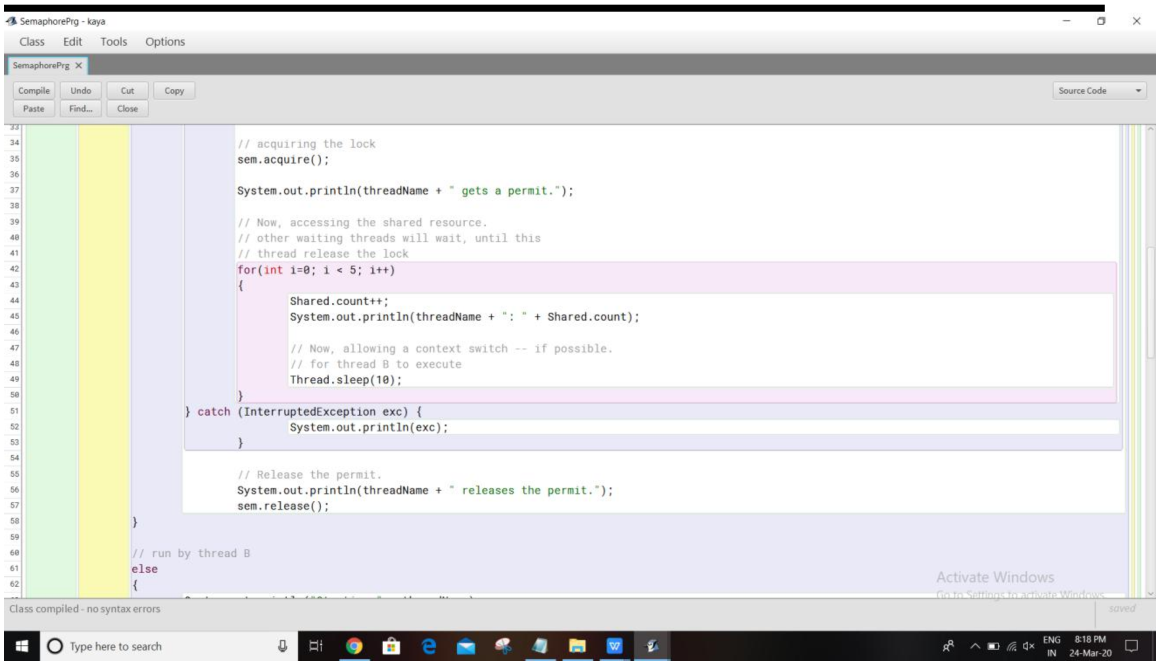
```
struct Semaphore {
    int value;

    // q contains all Process Control Blocks(PCBs)
    // corresponding to processes got blocked
    // while performing down operation.
    Queue<process> q;

} P(Semaphore s)
{
    s.value = s.value - 1;
    if (s.value < 0) {

        // add process to queue
        // here p is a process which is currently executing
        q.push(p);
        block();
    }
    else
        return;
}

V(Semaphore s)
{
    s.value = s.value + 1;
    if (s.value <= 0) {

        // remove process p from queue
        q.pop();
        wakeup(p);
    }
    else
        return;
}
```

In this implementation whenever process waits it is added to a waiting queue of processes associated with that semaphore. This is done through system call block() on that process. When a process is completed it calls signal function and one process in the queue is resumed. It uses wakeup() system call.

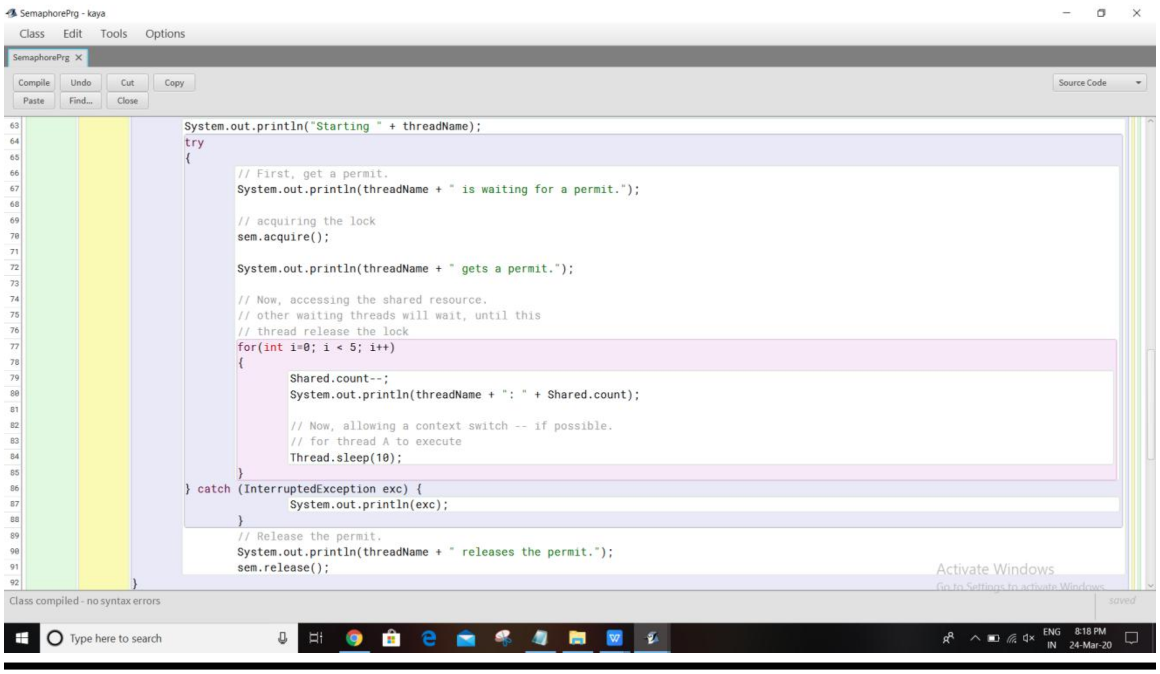
CODE:



```
// 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.");
```

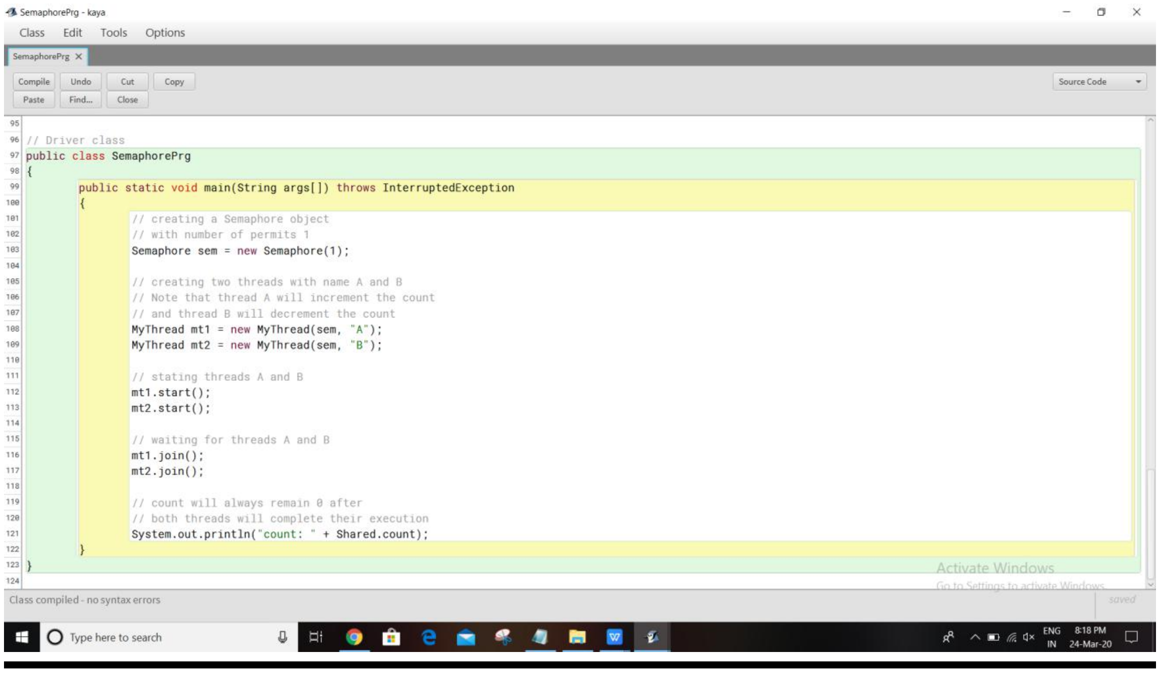


```
                                // acquiring the lock
                                sem.acquire();

                                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();
        }
```

Class compiled - no syntax errors



```
// Driver class
public class SemaphorePrg
{
        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);
        }
}
```

Class compiled - no syntax errors

REPOSITORY:

CONCLUSION: