L51

Friday, February 4, 2022 10:44 PM

Thread Synchronization

Q1. Threads in Java can be assigned priorities. The priority of a thread determines how it will behave with regards to other threads. For example if a low priority thread is hogging too much of CPU time and is not voluntarily surrendering CPU by sleeping or going into a wait, the system could pre-empt it and transfer the control to a thread with higher priority.

There can arise scenarios where multiple threads may want to access a shared resource. For example, let us consider two threads trying to add elements into an ArrayList. It is very likely that during the add operation call from a thread, while the ArrayList's internal data structure is being modified, the second thread can also invoke the add method. This could result in possible data corruption or even worse can mess up the internal state of the ArrayList.

Whenever a common resource is manipulated by multiple threads, it is important to endure that the manipulations happen in isolation of each other. Mean while one thread is performing the manipulation on the shared resource, the other threads are made to wait until the former completes. This is called **synchronization**.

Java allows us to write code which can build such synchronization between multiple threads by using the synchronized keyword.

In legacy collection classes like Vector, Stack and Hashtable all methods were **synchronized**.

A method is called synchronized when the synchronized keyword is used in the method declaration statement as shown below:

```
// Below code is from isEmpty() method in java.util.Vector class:public synchronized boolean isEmpty() {
        return elementCount == 0;
}
```

The **synchronized** keyword ensured that when one thread is calling this method it obtains the Vector object's **intrinsic lock** and this **intrinsic lock** is not released until the thread finishes execution of this method. And while the first thread is inside executing such a method, if another thread attempts to enter the method, it will have to wait till the initial thread which has obtained the lock comes out of the method and makes the lock available for the second thread to obtain it and enter.

This is how using an object's **intrinsic lock** (also called *monitor*) the synchronization is achieved.

In Java, every object has an intrinsic lock associated with it, meaning every object can be used for synchronization. The intrinsic lock acts as a gate-pass. Just like a visitor while entering into a secure building or a facility is given a visitor pass and is asked to surrender it back while leaving, similarly the thread entering into a synchronized block gains the intrinsic lock associated with the object on which it is synchronized and release it while exiting.

The above synchronized method can also be written as :

```
public boolean isEmpty() {
        synchronized (this) { // this is also called the synchronized statementreturn elementCount == 0;
        }
}
```

In both the above code snippets, the **intrinsic lock** is on the this reference.

In the second way, i.e, while using a **synchronized** statement synchronizing can be done on few lines of code which is usually called the critical block that should not be executed simultaneously by more than one thread.

Java also provides a keyword called volatile which when applied to fields the JVM will ensure that all threads see a consistent value for the fields while they are being updated by multiple threads.

See and retype the below code.

Note how the ArrayList instance which is not thread-safe by default is made thread-safe by using the utility method in Collections.synchronizedList() class. Similarly we can obtain a thread-safe instance of HashSet, LinkedHashSet, HashMap, LinkedHashMap and other non-thread-safe collection classes using corresponding methods in the Collections.synchronizedSet() class.

```java
package q11343;
import java.util.*;
public class SimpleThreadDemo3 {
    public static void main(String[] args) throws InterruptedException {
        List<String> entriesList = Collections.synchronizedList(new ArrayList<String>());
        Counter c1 = new Counter("Ganga", entriesList);
        Counter c2 = new Counter("Yamuna", entriesList);
        Thread t1 = new Thread(c1);
        Thread t2 = new Thread(c2);
        t1.start();
        t2.start();
        System.out.println("started t1 and t2 threads");
        t1.join();
        System.out.println("t1 has completed. t1.isAlive() = " + t1.isAlive());
        t2.join();
        System.out.println("t2 has completed. t2.isAlive() = " + t2.isAlive());
        System.out.println("At the end entriesList = " + entriesList);
    }
}
class Counter implements Runnable {
    private String name;
    private List<String> entriesList;
    public Counter(String name, List<String> entriesList) {
        this.name = name;
        this.entriesList = entriesList;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            entriesList.add(name + " : " + i);
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Q2. Given:

```
class SequenceGenerator {
        synchronized void sequence(long n) {
                for (int i = 1; i < 3; i++)
                        System.out.print(n + "-" + i + " ");
        }
}
public class Tester implements Runnable {
        static SequenceGenerator sg = new SequenceGenerator ();
public static void main(String[] args) {
                new Thread(new Tester()).start();
                new Thread(new Tester()).start();
        }
public void run() {
                sg.sequence(Thread.currentThread().getId());
        }
}
```

Which of the following statements are true?

- [ ] The output could be 5-1 6-1 6-2 5-2
- [x] The output could be 6-1 6-2 5-1 5-2
- [ ] The output could be 6-1 5-2 6-2 5-1
- [ ] The output could be 6-1 6-2 5-1 7-1

Q3. Let us consider a scenario where we have two threads, one thread downloads the files from the Internet and the other thread reads those downloaded files and extracts their content. In such a scenario, until the first thread has finished downloading the file completely from the Internet the second thread should not start reading it.

Java provides a way to solve such a problem by providing a means for the first thread to signal the second thread through **wait** and **notify** mechanism.

The Object class which is the root call of all classes in Java, has the below to facilitate **wait** and **notify** between threads:

1. wait() - it causes the current thread to wait until another thread invokes the notify() or notifyAll() on this object.
2. wait(long timeoutInMilliSeconds) - it causes the current thread to wait until another thread invokes the notify() or notifyAll() on this object or the specified timeout happend.
3. wait(long timeoutInMilliSeconds, int additionalNanoSeconds) - it causes the current thread to wait until another thread invokes the notify() or notifyAll() on this object or the specified timeout happend.
4. notify() - it causes one of the threads waiting on this object's intrinsic lock to wake up.
5. notifyAll() - it causes all the threads waiting on this object's intrinsic lock to wake up, while only one thread gets a chance to obtain the lock to proceed.

If any thread is waiting using any of the above mentioned **wait** methods, it will come out of its wait, if an Thread.interrupt() is called on that thread.

The **wait** or an **notify** call can be made on a object only after obtaining the **intrinsic lock** (also called **monitor**) on that object by synchronizing over it.

If we try to call **wait** or **notify** on a object without synchronizing on that object an IllegalMonitorStateException is thrown.

Note: We should never use the suspend() and resume() methods provided in the **Thread** class as they are depricated. Instead we should manually write their code.

See and retype the below code. In the below code thread t1 waits after it prints 5, until t2 signals t1 to resume using the notify.

Note how the **wait** and **notify** calls are made inside the **synchronized** statements over the monitor.

```java
package q11345;
public class WaitNotifyDemo {
    public static void main(String[] args) throws InterruptedException {
        Object sharedLock = new Object();
        Waiter waiter = new Waiter(sharedLock);
        Notifier notifier = new Notifier(sharedLock);
        Thread t1 = new Thread(waiter);
        Thread t2 = new Thread(notifier);
        t1.start();
        t2.start();
        t2.join();
        t1.join();
    }
}
class Waiter implements Runnable {
    private Object sharedLock;
    public Waiter(Object sharedLock) {
        this.sharedLock = sharedLock;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Waiter : " + i);
            if (i == 5) {
                System.out.println("Waiter will wait now until notified by Notifier");
                synchronized (sharedLock) {
                    try {
                        sharedLock.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println("Waiter has come out of wait.");
            }
        }
    }
}
class Notifier implements Runnable {
    private Object sharedLock;
    public Notifier(Object sharedLock) {
        this.sharedLock = sharedLock;
    }
    public void run() {
        System.out.println("Notifier is busy taking a nap to 7 secs.");
        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
```

```
47              e.printStackTrace();
48          }
49          System.out.println("Hurray! Notifier is awake and is about to call notify().");
50          synchronized (sharedLock) {
51              sharedLock.notify();
52          }
53      }
54  }
```

Q4. Read the code below:

```
void waitForSignal() throws InterruptedException {
    Object obj = new Object();
    synchronized (Thread.currentThread()) {
        obj.wait();
        obj.notify();
    }
}
```

Which statement is true?

- [ ] Executing the code results in InterruptedException.
- [x] The code throws IllegalMonitorStateException.
- [ ] The code throws a TimeoutException after ten minutes.
- [ ] Reversing the order of obj.wait() and obj.notify() might cause this method to complete normally.
- [ ] A call to notify() or notifyAll() from another thread might cause this method to complete normally.
- [ ] This code does NOT compile unless "obj.wait()" is replaced with "((Thread)obj).wait()".