Unit 2: Multithreaded Programming-II

2

Unit Structure

2.1 LEARNING OBJECTIVE

- To explain concurrency issues in multithreading and its solutions
- To understand inter thread communication

2.2 OUTCOMES

After learning the contents of this chapter, the reader must be able to :

- Understand the importance of concurrency
- Use the concept of synchronization in programming, and
- Use inter-thread communication in programs.

2.3 INTRODUCTION

Due to multiple threaded in a program, an asynchronous behavior introduces in your program. Therefore, synchronization is necessary when a program needs.

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. For example, in a banking system, you would not want one thread to credit some amount to user account balance while another thread is trying to debit some amount from same account balance; in such situations, you need some way to ensure that they don't conflict with each other.

2.4 SYNCHRONIZATION

Synchronization provides a simple monitor facility that can be used to provide mutual-exclusion between Java threads.

Java implements an elegant model of interprocess synchronization: "The monitor" (also called a semaphore). The monitor is a control mechanism. You can assume that the monitor is a very small box that can allow only one thread to stay in it. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

Synchronized keyword in Java is used to provide mutually exclusive access to a shared resource with multiple threads in Java. Synchronization in Javaguarantees that no two threads can execute a synchronized method which requires the same lock simultaneously or concurrently.

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

You can synchronize your code in either of two ways. Both involve the use of the synchronizedkeyword, and both are examined here.

There are two ways to synchronized your code

1. using synchronized methods
2. synchronized statements

2.4.1 Using synchronized methods

When you divide your program into separate threads, you need to define how they will communicate with each other. Synchronized methods are used to coordinate access to objects that are shared among multiple threads. These methods are declared with the synchronized keyword. Only one synchronized method at a time can be invoked for an object at a given point of time. When a synchronized method is invoked for a given object, it acquires the monitor for that object. In this case no other synchronized method may be invoked for that object until the monitor is released. This keeps synchronized methods in multiple threads without any conflict with each other.

To understand the need for synchronization, let's begin with a simple example that does not use it—but should. The following program has three simple classes. The first one, MultiplicationTable, has a single method named printMulTable( ). The printMulTable ( ) method takes an int parameter. This method print multiplication value. It calls Thread.sleep(250), which pauses the current thread for 250 millisecond. The constructor of the next class, MThread, takes a reference to an instance of the MultiplicationTable class and an int, which are stored in t and n respectively. The constructor also creates a new thread that will call this object's

run( ) method. The thread is started immediately. The run( ) method of MThread calls the printMulTable ( ) method on the t instance of MultiplicationTable, passing in the n int. Finally, the synchronized class starts by creating a single instance of MultiplicationTable, and two instances of MThread, each with a unique int value. The same instance of MultiplicationTable is passed to each MThread.

// Program-6

```
//example of java synchronized method

class MultiplicationTable{
void printMulTable(int n){   //nonsynchronized method
  for(int i=1;i<=5;i++){
    System.out.println(n*i);
    try{
     Thread.sleep(250);
    }catch(Exception e){System.out.println(e);}
  }

 }
}

class MThread extends Thread{
        MultiplicationTable t;
        int n;
        MThread(MultiplicationTable t, int n ){
                this.t=t;
                this.n=n;
        }
        public void run(){
                t.printMulTable(n);
        }
}

public class ThreadSynchronizationDemo{
        public static void main(String args[]){
```

```
        MultiplicationTable obj = new MultiplicationTable();//only one object

        MThread t1=new MThread(obj, 5);
        MThread t2=new MThread(obj, 100);

        t1.start();
        t2.start();
    }
}
```

Output:

```
5
100
10
200
15
300
20
400
25
500
```

As you can see, by calling **sleep( )**, the **printMulTable( )** method allows execution to switch to another thread. This results in the mixed-up output of the two threads. In this program, nothing exists to stop two threads from calling the same method, on the same object, at the same time. This is known as a race condition, because the two threads are racing each other to complete the method. This example used **sleep( )** to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur. This can cause a program to run right one time and wrong the next.

To fix the preceding program, you must serialize access to **printMulTable ( )**. That is, you must restrict its access to only one thread at a time. To do this, you

simply need to precede **printMulTable ( )**'s definition with the keyword **synchronized**, as shown here:

```
synchronized void printMulTable(int n){    //synchronized method
  for(int i=1;i<=5;i++){
    System.out.println(n*i);
    try{
     Thread.sleep(250);
    }catch(Exception e){System.out.println(e);}
  }
 }
```

This prevents other threads from entering printMulTable **( )** while another thread is using it. After **synchronized** has been added to printMulTable **( )**, the output of the program is as follows:

Output:

```
5
10
15
20
25
100
200
300
400
500
```

In such type of situation you should use the **synchronized** keyword to prevent the state from race conditions.

2.4.2 The synchronized Statement/block

An effective and easy way of synchronization is to create synchronized methods within classes. But it will not work in all cases, for example, if you want to synchronize access to objects of a class that was not designed for multithreaded programming or the class does not use synchronized methods. Further, this class

was not created by you, but by a third party and you do not have access to the source code. In such situation, the synchronized statement block is a solution. Synchronized statement block are similar to synchronized methods. It is used to acquire a lock on an object before performing an action.

The syntax of Synchronized statement block:

```
Synchronized (obj) {
        // statement block
}
```

Here, **obj** is the object to be locked. If you desire to protect instance data, you should lock against that object. If you desire to protect class data, you should lock the appropriate **Class** object.

```
public void run() {
        synchronized (t) {
                t.printMulTable(n);
        }
}
```

2.5 DEADLOCK

Deadlock in java is a part of multithreading/multitasking. Deadlock can occur in a situation when two or more threads wait indefinitely for each other to relinquish locks. In simple words, a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock. Deadlock situations can also arise that involve more than two threads.

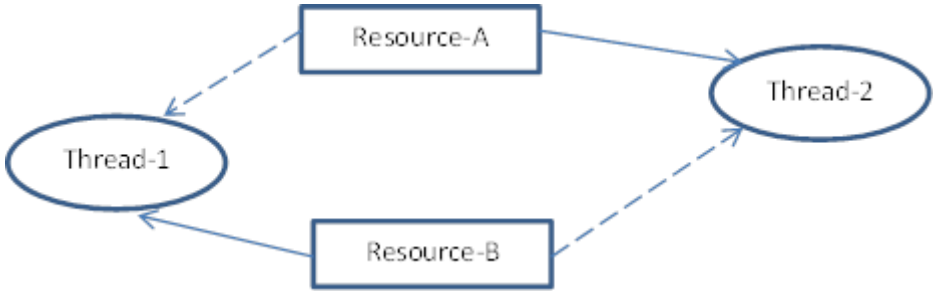

Figure-96 Deadlock Scenario

Thread-1 has resource-B and is requesting Resource-A
Thread-2 has resource-A and is requesting Resource-B

- **How to avoid deadlock**

The solution to any problem lies in identifying the cause of the problem. There many different situations and solution of the deadlock state. In above situation, it is the pattern of accessing the resources A and B, is main issue. So, to resolve it, we will simply re-order the statements where the code is accessing shared resources.

2.6 INTER-THREAD COMMUNICATION

In previous section you learn about how deadlock can occur if a thread obtain a lock and does not relinquish it. Now, in this section you will see that how threads can cooperate with each other, a thread can temporarily release a lock so the threads can get opportunity to execute a synchronized method or statement block. The lock can be acquired then after.

To avoid wastage of precious time of CPU, or to avoid polling, Java includes an interthread communication mechanism via the wait( ), notify( ), and notifyAll( ) methods. These methods are implemented as final methods in Object class, so all classes have them. These three methods can be called only from within a synchronized method or statement block.

The Object class contains three final methods that allow threads to communicate with each other. These methods are declared as:

public final void wait( ) throws Interrupted Exception
public final void wait(long milisec) throws Interrupted Exception
public final void wait(long milisec, int nanosec) throws Interrupted Exception

public final void notify( )
public final void notifyAll( )

- **wait( )** methodtells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ) or notifyAll().
- **notify( )** method **w**akes up a single thread that is waiting on this object's monitor.
- **notifyAll( )** method **w**akes up all threads that are waiting on this object's monitor, the highest priority Thread will be run first.

Let us see the following program written to control access of resource using wait() and notify () methods.

// Program-7

```
class WaitNotify implements Runnable
{
        WaitNotify ()
        {
                Thread th = new Thread (this);
                th.start();
        }
        synchronized void notifyThat ()
        {
                System.out.println ("Notify the threads waiting");
                this.notify();
        }
        synchronized public void run()
        {
        try
        {
                System.out.println("Thead is waiting....");
                this.wait ();
        }
        catch (InterruptedException e){}
                System.out.println ("Waiting thread notified");
        }
}
Class RunWaitNotify
{
        public static void main (String args[])
        {
                WaitNotify wait_not = new WaitNotify();
```

```
                Thread.yield ();
                wait_not.notifyThat();
        }
}
```

Output:

Thead is waiting....

Notify the threads waiting

Waiting thread notified

2.7 SUSPENDING, RESUMING, AND STOPPING THREADS

Prior to Java 2 the suspend( ), resume( ), and stop( ) methods defined by Thread seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs.

Java 2 onward these methods were deprecated. Here's why. The suspend( ) method of the Thread class is deprecated in Java 2. This was done because suspend( ) can sometimes cause serious system failures. Assume that a thread has obtained locks on critical data structures. If that thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.

The resume( ) method is also deprecated. It does not cause problems, but cannot be used without the suspend( ) method as its counterpart. The stop( ) method of the Thread class, too, is deprecated in Java 2. This was done because the similar to suspend() method.

The task of suspend(), resume() and stop() methods is accomplished by forming a flag variable that indicates the execution state of the thread. As long as this flag is set to "running," the run( ) method must continue to let the thread execute. If this variable is set to "suspend," the thread must pause. If it is set to "stop," the thread must terminate. Of course, a variety of ways exist in which to write such code, but the central theme will be the same for all programs.

Check Your Progress 1

1) Which is more preferred – Synchronized method or synchronized block?
2) What is deadlock?
3) How does thread communicate with each other?

2.8 LET US SUM UP

This chapter explains various issue and solutions in concurrency. This chapter explains concept of synchronization, creating synchronous methods and inter thread communication. It is also explained how object locks are used to control access to shared resources. It is also explain deadlock.

2.9 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

Check Your Progress 1

1) Synchronized block is more preferred way because it doesn't lock the Object, synchronized methods lock the Object and if there are multiple synchronization blocks in the class, even though they are not related, it will stop them from execution and put them in wait state to get the lock on Object.
2) Deadlock is a situation when two or more threads wait indefinitely for each other to relinquish locks.
3) When threads want to share resources, communication between Threads is important to coordinate their activity. Object class contains wait(), notify() and notifyAll() methods allows threads to communicate.