Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/Java: More Multithreading/5b7186d917a1437e093bd506

TUTORIAL

Java: More Multithreading

Topics

1.6 Synchronized Statement

1.8 Wait and Notify

Synchronization

There are many situations in which multiple threads must share access to common objects. For example, the effects of multithreading are considered in following codes by having multiple executing threads write to the Java console, a common shared object. These examples do not require any coordination or synchronization in the way the threads access the console window: Whichever thread is currently executing is able to write to the console window. There are times when you might want to coordinate access to shared resources. For example, in a database system, you might not want one thread to be updating a database record while another thread is trying to read it. Java enables you to coordinate the actions of multiple threads using synchronized methods and synchronized statements. I'll cover synchronized methods first and then synchronized statements.

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 can be invoked for an object at a given point in time. This keeps synchronized methods in multiple threads from conflicting with each other.

All classes and objects are associated with a unique lock. The lock is used to control the way in which synchronized methods are allowed to access the class or object. When a synchronized method is

invoked for a given object, it tries to acquire the lock for that object. If it succeeds, no other synchronized method may be invoked for that object until the lock is released. A lock is automatically released when the method completes its execution and returns. A lock may also be released when a synchronized method executes certain methods, such as wait(). We'll cover the wait() method in a later section.

Note: Class Locks, Although we speak of a lock on a class, technically a class lock is a lock that is acquired on the class's Class object.

Note: Locks and States, When a thread is waiting for a lock on an object, the thread leaves the running state and enters the waiting state. When the thread acquires the lock on the object, it moves from the waiting state to the ready state.

The following example shows how synchronized methods and object locks are used to coordinate access to a common object by multiple threads. This example adapts the Thread1 program for use with synchronized methods, as shown in code below:

Java

```
class Thread3 extends Thread
{
  static String message[] = {"CODE","QUOTIENT","GET",
"BETTER,","@", "CODING,"};
  public Thread3(String id)
  {
    super(id);
  }
  public void run()
  {
    SynchronizedOutput.displayList(getName(),message);
  }
  void randomWait()
  {
    try {
      sleep((long)(300*Math.random()));
    }
    catch (InterruptedException x){
      System.out.println("Interrupted!");
```

```
    }
  }
}

class SynchronizedOutput
{
  public static synchronized void displayList(String name,String list[])
  {
    for(int i=0;i<list.length;++i)
    {
      Thread3 t = (Thread3) Thread.currentThread();
      t.randomWait();
      System.out.println(name+list[i]);
    }
  }
}

class Main
{
  public static void main(String args[])
  {
    Thread3 thread1 = new Thread3("thread1: ");
    Thread3 thread2 = new Thread3("thread2: ");
    thread1.start();
    thread2.start();
    boolean thread1IsAlive = true;
    boolean thread2IsAlive = true;
    do {
      if(thread1IsAlive && !thread1.isAlive())
      {
        thread1IsAlive = false;
        System.out.println("Thread 1 is dead.");
      }
      if(thread2IsAlive && !thread2.isAlive())
      {
        thread2IsAlive = false;
        System.out.println("Thread 2 is dead.");
      }
    }while(thread1IsAlive || thread2IsAlive);
```

```
  }
}
```

Now edit above file and delete the synchronized keyword in the declaration of the displayList() method of class SynchronizedOutput. Save the file, recompile it, and rerun it with the change in place. You may now get different output.

Java

```
class SynchronizedOutput
{
  public static void displayList(String name,String list[])
  {
    for(int i=0;i<list.length;++i)
    {
      Thread3 t = (Thread3) Thread.currentThread();
      t.randomWait();
      System.out.println(name+list[i]);
    }
  }
}

```

The difference in the program's output should give you a feel for the effects of synchronization upon multithreaded program execution. What difference does the fact that displayList() is synchronized have on the program's execution? When displayList() is not synchronized, it may be invoked by one thread (for example thread1), display some output, and wait while thread2 executes. When thread2 executes, it too invokes displayList() to display some output. Two separate invocations of displayList(), one for thread1 and the other for thread2, execute concurrently. This explains the mixed output display.

When the synchronized keyword is used, thread1 invokes displayList() and acquires a lock for the SynchronizedOutput class (because displayList() is a static method). After this, displayList() proceeds with the output display for thread1. Because thread1 acquired a lock for the SynchronizedOutput class, thread2 must wait until the lock is released before it is able to invoke displayList() to

display its output. This explains why one task's output is completed before the other's.

Synchronized Statement

The synchronized statement is similar to a synchronized method in that it is used to acquire a lock on an object before performing an action. The synchronized statement differs from a synchronized method in that it can be used with the lock of any object—the synchronized method can only be used with its object's (or class's) lock. It also differs in that it applies to a statement block, rather than an entire method. The syntax of the synchronized statement is as follows:

```
synchronized (object)
{
        statement(s)
}
```

The statements enclosed by the braces are only executed when the current thread acquires the lock for the object or class enclosed by parentheses. Code below shows how to use this statement:

Java

```
class Thread4 extends Thread
{
  static String message[] = {"CODE","QUOTIENT","GET",
"BETTER,","@", "CODING,"};
  public Thread4(String id)
  {
    super(id);
  }
  public void run()
  {
    synchronized(System.out)
    {
      for(int i=0;i<message.length;++i)
      {
        randomWait();
        System.out.println(getName()+message[i]);
      }
```

```
    }
  }
  void randomWait()
  {
    try
    {
      sleep((long)(300*Math.random()));
    }
    catch (InterruptedException x)
    {
      System.out.println("Interrupted!");
    }
  }
}

class Main
{
  public static void main(String args[])
  {
    Thread4 thread1 = new Thread4("thread1: ");
    Thread4 thread2 = new Thread4("thread2: ");
    thread1.start();
    thread2.start();
    boolean thread1IsAlive = true;
    boolean thread2IsAlive = true;
    do {
      if(thread1IsAlive && !thread1.isAlive())
      {
        thread1IsAlive = false;
        System.out.println("Thread 1 is dead.");
      }
      if(thread2IsAlive && !thread2.isAlive())
      {
        thread2IsAlive = false;
        System.out.println("Thread 2 is dead.");
      }
    }while(thread1IsAlive || thread2IsAlive);
  }
}
```

Wait and Notify

The wait(), notify(), and notifyAll() methods of the Object class are used to provide an efficient way for threads to wait for resources. They also provide additional synchronization support. These methods are used to implement a synchronization scheme where threads that have acquired an object's (or class's) lock, wait until they are notified that access to the object is permitted. The wait(), notify() and notifyAll() methods provide you with finer control over shared resources. They enable you to place threads in a waiting pool until resources become available to satisfy the threads' requests. A separate resource monitor or controller process can be used to notify waiting threads that they are able to execute.

When a thread has acquired an object's lock, either through a synchronized method or a synchronized statement, the thread invokes the object's wait() method. This causes the thread to lose the object's lock and enter the waiting state. It is possible for several threads to be waiting as the result of invoking an object's wait() method. These threads are referred to as the waiting pool. A thread typically invokes an object's wait() method by invoking a synchronized method of the object. The call to wait() is embedded in the synchronized method. For example, a serviceproviding object would provide a method of the following form:

```
public synchronized returnType requestService(parameters)
{
 while(resourceNotAvailable)
 {
        wait();
 }
 // Perform service
}
```

When another thread acquires the object's lock and invokes the object's notify() method, one of the waiting threads is returned to the ready state. (The notifyAll() method is used to return all of the threads in the waiting pool to the ready state.) When a waiting thread reenters the running state, it reacquires the lock on the object and can then continue its processing.

Code below shows how the wait() and notify() methods are used to control access to a shared resource. A Resource object is shared among a Controller object and several User objects. The Resource object provides access to the console display. The User objects execute as separate threads and use the Resource object to display their output. The Controller object also executes as a separate thread and controls access to the Resource object, making it available for output every 10 seconds.

The displayOutput() method of the Resource object displays a message on the console when okToSend is true. If a User thread invokes displayOutput() when okToSend is false, it enters the waiting pool. The allowOutput() method of Resource is periodically invoked by the Controller to allow a User thread to exit the waiting pool and display output via the Resource object. Note that the wait() method, like the sleep() method of the Thread class, throws the InterruptedException. This exception is thrown when a thread's interrupt() method is invoked while it is in the waiting state.

Java

```
class Main
{
  public static void main(String args[])
  {
    Resource resource = new Resource();
    Thread controller = new Thread(new
Controller(resource));
    Thread[] user = new Thread[3];
    for(int i=0;i<user.length;++i)
      user[i] = new Thread(new User(i,resource));
    controller.start();
    for(int i=0;i<user.length;++i)
      user[i].start();
    boolean alive;
    do {
      alive = false;
      for(int i=0;i<user.length;++i)
        alive = user[i].isAlive();
      Thread.currentThread().yield();
    } while(alive);
    controller.interrupt();
```

```
  }
}

class Resource
{
  boolean okToSend = false;
  public synchronized void displayOutput(int id,String[] message)
  {
    try
    {
      while(!okToSend)
        wait();
      okToSend = false;
      for(int i=0;i<message.length;++i)
        System.out.println(id+": "+message[i]);
    }
    catch(InterruptedException ex){ }
  }
  public synchronized void allowOutput()
  {
    okToSend = true;
    notify();
  }
}

class Controller implements Runnable
{
  Resource resource;
  public Controller(Resource resource)
  {
    this.resource = resource;
  }
  public void run()
  {
    try
    {
      while(true)
      {
        Thread.currentThread().sleep((long)10);
```

```
        resource.allowOutput();
      }
    }catch(InterruptedException ex) { }
  }
}

class User implements Runnable
{
  static String message[] = {"CODE","QUOTIENT","GET",
"BETTER,","@", "CODING,"};
  int id;
  Resource resource;
  public User(int id,Resource resource)
  {
    this.id = id;
    this.resource = resource;
  }
  public void run()
  {
    resource.displayOutput(id,message);
  }
}

```