**TOPICS**

Synchronization

Data (Competitive) Synchronization

Event (Co-operative) Synchronization

Synchronized method

Synchronized block

Wait

Notify

NotifyAll

**DISCUSSION**

Competitive Synchronization.

When two or more threads write to the same instance variables within an object, their accesses to the variables need to be synchronized.  This is called competitive synchronization.  Without proper competitive synchronization, the values of instance variables that are accessed competitively by multiple threads would become inconsistent.

In Java, each object with one or more synchronized blocks (block implies a synchronized peace of code or a whole synchronized method) has a wait queue associated with it.  After a thread enters any one of its synchronized blocks, other threads attempting to enter this or any other synchronized blocks are put on the wait queue.  Only after the executing thread exits the synchronized block, another waiting thread is allowed to execute and enter a synchronized block. This scheme serializes the execution of all synchronized blocks associated with a given object.

In multithreading, in order to serialize accesses to an instance variable of an object, the programmer should synchronize code blocks that may concurrently try to write to that variable.

Co-operative Synchronization

At times, multiple threads co-operatively synchronize their executions.  For example, when a thread notices that another thread is performing a certain task (such as writing to a buffer), it voluntarily puts itself into a wait state.  When the executing thread completes the task, it notifies the waiting thread that he is done so that it may now run.  This type of synchronization is called cooperative synchronization.  The consumer/producer problem is a good example of that.

In the ticker assignment, even when moving is false, the thread keeps running and checking the value of moving during each pass of its loop.  An alternative solution would be that on detecting that moving is false, the thread would put itself in wait state.  Then when, the user presses the start button, the start event thread would issue a notify which would awaken the waiting thread.

In Java, the class Object provides wait and notify methods. Consequently, each object has wait and notify methods.

Java requires that a method which invokes wait or notify be synchronized.  This ensures that notify and wait would execute serially.  In the assignment, since the method run cannot be synchronized (a Java requirement), we can call the method myRun from within it and can synchronize that method. Then we can call wait from within myRun method. Similarly we can synchronize the setMoving method from which we can call notify.

Similarly, the methods that call wait and notify should be within the same object.  This allows the system to use the object�s wait queue.  In the assignment, the method myRun and setMoving are in the same object and use that object�s wait queue.

In the assignment, when a thread, executing the object�s myRun method,  issues wait, it (the thread) is put on the wait queue of that object.  Subsequently, when another thread executing the setMoving method of the same object invokes notify, the waiting thread is removed from the wait queue and starts executing.  The two threads synchronize cooperatively.

**DESCRIPTION**

Enhance the assignment TickerSimple (assignment with a single ticker) to provide the following:

Synchronize accesses to shared variables (e.g. moving).

Synchronize threads by using wait and notify

**SAMPLE CODE FOR WAIT/NOTIFY**

 private int x, y;

 public void run()

  {

    while(true)

    {

      myRun();

      try

      {

          Thread.sleep(100);

      }

      catch (InterruptedException ex)

      {

      }

    }

  }

  public synchronized void myRun ()

  {

    if(!moving)

    {

      try

      {

          wait();

      }

      catch (InterruptedException ex)

      {

      }

    }

    else

    {

      if (x >= this.getWidth)

          x = 10;

      else

          x += 10;

      this.repaint();

    }

  }

  public synchronized void setMoving(boolean moving)

  {

    this.moving = moving;

    notify();

  }