Basic Thread Synchronization

Programación Concurrente

2017

Ing. Ventre, Luis O.

Controlling concurrent access to multiple copies of a resource

- En el último ej. aprendimos las bases del uso de semáforos.
  - Éste implementaba un semáforo binario, útil para proteger el acceso a un único recurso, o a una sección crítica.
- Los semáforos también pueden ser utilizados para proteger varios recursos.

- Se verá un ej. con una cola de impresión que puede imprimir en 3 impresoras diferentes.

Controlling concurrent access to multiple copies of a resource

- Se modificará el ejercicio de PrintQueue con semáforos.
- Clases:
  - PrintQueue:
    - Agregar la declaración de un arreglo de boolean llamado freeprinters.
    - Declarar un objeto Lock
    - Modificar el constructor para inicializar los nuevos objetos declarados. El array de impresoras libres debe inicializarse todos en true.
    - Declare un semáforo será inicializado en 3.

Controlling concurrent access to multiple copies of a resource

- Modificar el método PrintJob. Este recibe un objeto como parámetro.
  - Primero debe ejecutar acquire() para tomar el semáforo.
  - Luego averigua la impresora que nos asignaron con el método getprinter().
  - Simula impresión.
  - Liberar la impresora (marcar en arreglo como liberada).
  - Libera con release() el semáforo.
- Implementar el método getprinter().
  - Tomo el lock para acceder a una impresora.
  - Luego busca el primer valor true del array freeprinters.
  - Finalmente libera el lock y devuelve el índice de la freeprinter.

Controlling concurrent access to multiple copies of a resource

- La clase main y job no tienen diferencia.

- La clave de éste ejercicio es la clase PrintQueue.
  - El semáforo creado, es inicializado en 3, limitando el acceso a los primeros 3 hilos que llamen el método acquire(). A través de esto los hilos accederán a la sección crítica.
  - En esta sección crítica los hilos obtienen un índice de impresora a utilizar. Esta parte esta implementada para mayor realismo pero no útiliza semáforos.

Controlling concurrent access to multiple copies of a resource

- La clase main y job no tienen diferencia.
- Clase PrintQueue

Main.java | PrintQueue.java | Job.java

```
1  package com.packtpub.java7.concurrency.chapter3.recipe2.task;
2  import java.util.concurrent.Semaphore;
6
8   * This class implements a PrintQueue that have access to three printers.
16 public class PrintQueue {
17
19     * Semaphore to control the access to the printers
21    private Semaphore semaphore;
23     * Array to control what printer is free
25    private boolean freePrinters[];
26
28     * Lock to control the access to the freePrinters array
30    private Lock lockPrinters;
31
33     * Constructor of the class. It initializes the three objects
35    public PrintQueue(){
36        semaphore=new Semaphore(3);
37        freePrinters=new boolean[3];
38        for (int i=0; i<3; i++){
39            freePrinters[i]=true;
40        }
41        lockPrinters=new ReentrantLock();
42    }
43
```

Controlling concurrent access to multiple copies of a resource

- Clase PrintQueue método printJob

```
44    public void printJob (Object document){
45        try {
46            // Get access to the semaphore. If there is one or more printers free,
47            // it will get the access to one of the printers
48            semaphore.acquire();
49
50            // Get the number of the free printer
51            int assignedPrinter=getPrinter();
52
53            Long duration=(long)(Math.random()*10);
54            System.out.printf("%s: PrintQueue: Printing a Job in Printer %d during %d seconds\n",
55                    Thread.currentThread().getName(),assignedPrinter,duration);
56            TimeUnit.SECONDS.sleep(duration);
57
58            // Free the printer
59            freePrinters[assignedPrinter]=true;
60        } catch (InterruptedException e) {
61            e.printStackTrace();
62        } finally {
63            // Free the semaphore
64            semaphore.release();
65        }
66    }
```

Controlling concurrent access to multiple copies of a resource

- Clase PrintQueue método getPrinter

```
68      private int getPrinter() {
69          int ret=-1;
70
71          try {
72              // Get the access to the array
73              lockPrinters.lock();
74              // Look for the first free printer
75              for (int i=0; i<freePrinters.length; i++) {
76                  if (freePrinters[i]){
77                      ret=i;
78                      freePrinters[i]=false;
79                      break;
80                  }
81              }
82          } catch (Exception e) {
83              e.printStackTrace();
84          } finally {
85              // Free the access to the array
86              lockPrinters.unlock();
87          }
88          return ret;
89      }
90
91  }
92
```

Controlling concurrent access to multiple copies of a resource

- La salida:

```
Thread 3: Going to print a job
Thread 0: Going to print a job
Thread 4: Going to print a job
Thread 2: Going to print a job
Thread 9: PrintQueue: Printing a Job in Printer 2 during 5 seconds
Thread 8: PrintQueue: Printing a Job in Printer 1 during 3 seconds
Thread 8: The document has been printed
Thread 11: PrintQueue: Printing a Job in Printer 1 during 0 seconds
Thread 11: The document has been printed
Thread 5: PrintQueue: Printing a Job in Printer 1 during 4 seconds
Thread 1: The document has been printed
Thread 10: PrintQueue: Printing a Job in Printer 0 during 6 seconds
Thread 9: The document has been printed
Thread 7: PrintQueue: Printing a Job in Printer 2 during 1 seconds
Thread 7: The document has been printed
```

- Un detalle interesante.
  - Los métodos acquire(), acquireUninterruptibly(), tryacquire() y release() tienen otra versión sobrecargada que acepta cómo parámetro la cantidad de permisos a tomar o devolver del contador interno.

Waiting for multiple concurrent events

- En java existe una clase llamada CountDownLatch, que permite a uno o más hilos esperar, hasta que un grupo de operaciones sea realizada.
  - Esta clase se inicializa con un valor (int) que representa el número de acciones que el/los hilo/s va/n a esperar para continuar.
  - Cuando un hilo debe esperar la ejecución de éstas acciones ejecuta el método await(). Este método lo duerme hasta la completitud de todas las acciones.

Waiting for multiple concurrent events

- Cuando una hilo, finaliza una acción ejecuta el método countDown() para decrementar el contador interno.
- Cuando el contador llega a 0, la clase despierta a todos los hilos dormidos en el método await().

- Se verá un ejemplo que modela un sistema de videoconferencia.
  - El sistema de videoconferencia deberá esperar el arribo de todos los participantes antes de comenzar.

Waiting for multiple concurrent events

- Clase: VideoConference (runnable)
  - Declarar un objeto de la clase CountDownLatch
  - Implementar el constructor. (inicializar el objeto de la clase CountDownLatch, recordar que el sistema esperará el arribo del número de participantes recibido como parámetro).
  - Implementar el método arrive(). Este método será llamado cada vez que arribe un participante.
  - Implementar el método run. Primero se ejecuta getCount, para imprimir cantidad de participantes. Luego llama el método await() para esperarlos. Y finalmente imprime mensaje de arribo de todos.

Waiting for multiple concurrent events

- Clase:

Video

Conference

Main.java | Participant.java | Videoconference.java

```
1  package com.packtpub.java7.concurrency.chapter3.recipe3.task;
2  import java.util.concurrent.CountDownLatch;
3
5  * This class implements the controller of the Videoconference
11 public class Videoconference implements Runnable{
13     * This class uses a CountDownLatch to control the arrivel of all
16     private final CountDownLatch controller;
17
19     * Constructor of the class. Initializes the CountDownLatch
22     public Videoconference(int number) {
23         controller=new CountDownLatch(number);
24     }
26     * This method is called by every participant when he incorporates to the VideoConference
29     public void arrive(String name){
30         System.out.printf("%s has arrived.\n",name);
31         // This method uses the countDown method to decrement the internal counter of the
32         // CountDownLatch
33         controller.countDown();
34
35         System.out.printf("VideoConference: Waiting for %d participants.\n",controller.getCount());
36     }
38     * This is the main method of the Controller of the VideoConference. It waits for all
41     @Override
42     public void run() {
43         System.out.printf("VideoConference: Initialization: %d participants.\n",controller.getCount());
44         try {
45             // Wait for all the participants
46             controller.await();
47             // Starts the conference
48             System.out.printf("VideoConference: All the participants have come\n");
49             System.out.printf("VideoConference: Let's start...\n");
50         } catch (InterruptedException e) {
51             e.printStackTrace();
52         }
53     }
```

Waiting for multiple concurrent events

- Clase: Participante(runnable)
  - Declara un atributo privado de la clase VideoConference.
  - Declara un atributo privado string.
  - Implementa el constructor
  - Implementa el run.
    - Se duerme por un tiempo randómico.
    - Luego ejecuta el método arrive.

Waiting for multiple concurrent events

- Clase: Participant

Main.java | Participant.java | Videoconference.java

```
1  package com.packtpub.java7.concurrency.chapter3.recipe3.task;
2
3  import java.util.concurrent.TimeUnit;
4
6  * This class implements a participant in the VideoConference
9  public class Participant implements Runnable {
10
12     * VideoConference in which this participant will take part off
14     private Videoconference conference;
15
17     * Name of the participant. For log purposes only
19     private String name;
20
22     * Constructor of the class. Initialize its attributes
26     public Participant(Videoconference conference, String name) {
27         this.conference=conference;
28         this.name=name;
29     }
30
32     * Core method of the participant. Waits a random time and joins
34     @Override
35     public void run() {
36         Long duration=(long)(Math.random()*10);
37         try {
38             TimeUnit.SECONDS.sleep(duration);
39         } catch (InterruptedException e) {
40             e.printStackTrace();
41         }
42         conference.arrive(name);
43         System.out.printf("%s Ha ejecutado arrive \n",name);
44
45     }
46 }
47
```

Waiting for multiple concurrent events

- Clase: Main
  - Crear un objeto de la clase VideoConference inicializado en 10.
  - Crear un hilo para este objeto y lanzarlo.
  - Crear 10 objetos Participantes y 10 hilos y lanzarlos.

Waiting for multiple concurrent events

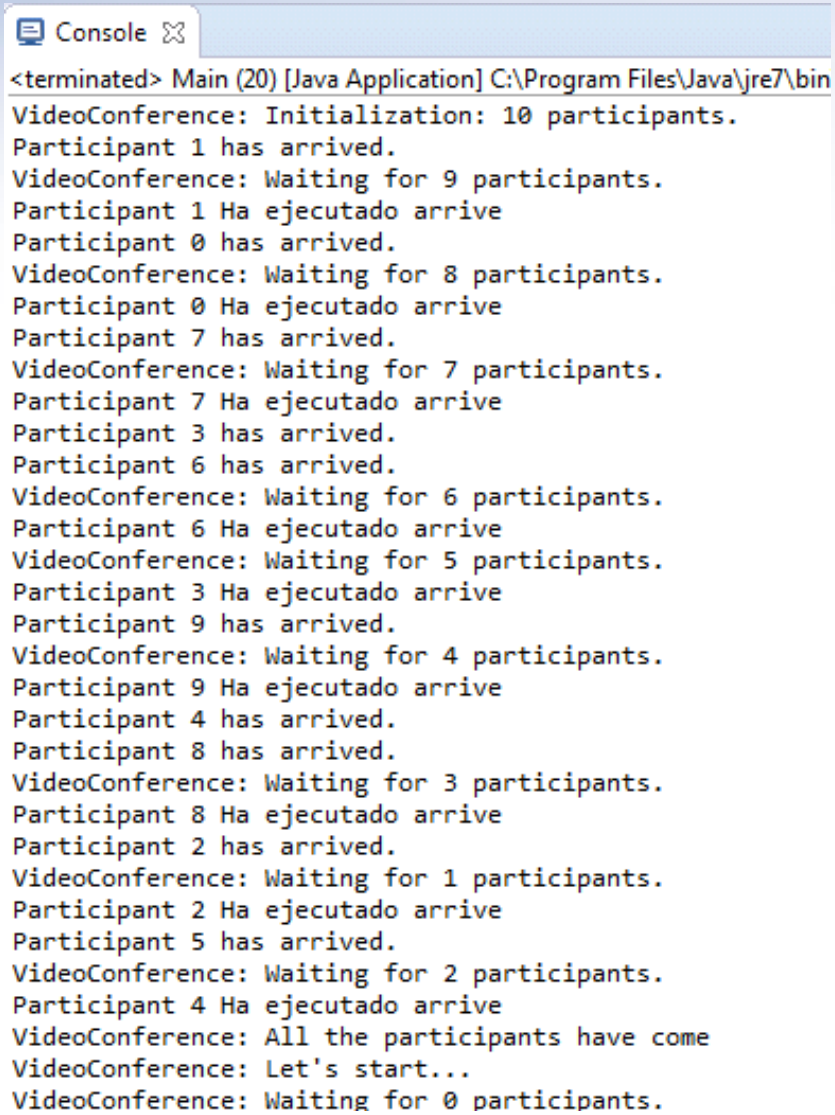
- Clase: Main

Main.java | Participant.java | Videoconference.java

```
1  package com.packtpub.java7.concurrency.chapter3.recipe3.core;
2
3  import com.packtpub.java7.concurrency.chapter3.recipe3.task.Participant;
5
7   * Main class of the example. Create, initialize and execute all the objects
11 public class Main {
12
14      * Main method of the example
17     public static void main(String[] args) {
18
19         // Creates a VideoConference with 10 participants.
20         Videoconference conference=new Videoconference(10);
21         // Creates a thread to run the VideoConference and start it.
22         Thread threadConference=new Thread(conference);
23         threadConference.start();
24
25         // Creates ten participants, a thread for each one and starts them
26         for (int i=0; i<10; i++){
27             Participant p=new Participant(conference, "Participant "+i);
28             Thread t=new Thread(p);
29             t.start();
30         }
31
32     }
33
34 }
```

Waiting for multiple concurrent events

- Ejecución



Waiting for multiple concurrent events

- La clase CountDownLatch tiene 3 elementos básicos:
  - El valor de inicialización que determina cuántos eventos debe esperar.
  - El método await(), llamado por los threads que quieren esperar la finalización de todas las acciones.
  - El método countDown(), llamado por los eventos cuando finalizan la acción que sincronizará.

Waiting for multiple concurrent events

- Detalles a considerar:
  - No hay ninguna forma de resetear el valor del contador interno del countDownLatch. Ni modificarlo.
  - Una vez inicializado en su valor, solo se puede decrementar con el método countDown().
  - Cuando el contador llega a 0, todas las llamadas al método await() son devueltas inmediatamente y el countDown() no tiene efecto.

Waiting for multiple concurrent events

- Detalles a considerar:
  - Esta primitiva tiene diferencias con otras:
    - No es utilizada para proteger un recurso compartido ni una sección crítica.
    - Es utilizado para sincronizar uno o más hilos con la ejecución de una o más tareas.
    - Solo admite un uso…se debe crear otro objeto si queremos hacer la sincronización nuevamente.
  - Existe una sobrecarga del método await() que puede recibir hasta dos parámetros de tiempo (long, timeunit). El thread se duerme hasta que llega a 0 o pasa ese tiempo y es interrumpido.

Synchronizing tasks in a common point

- Java provee una clase llamada CyclicBarrier, la cual nos permite la sincronización de dos o más hilos en un determinado punto.

- Esta clase comparte algunos objetivos con la anterior vista, pero es más potente.

- La clase CyclicBarrier es inicializada con un número entero. El cual es el número de hilos que serán sincronizados en un determinado punto.

Synchronizing tasks in a common point

- Cuando un hilo llega al punto determinado ejecuta el método await() para esperar el arribo de los demás hilos.

- Cuando el último de los hilos ejecuta el método await(), la clase CyclicBarrier despierta a todos para que prosigan su ejecución.

- Una ventaja interesante es la posibilidad de enviarle al CyclicBarrier un objeto runnable que se ejecutará una vez que todos los hilos ejecutaron el await().

Synchronizing tasks in a common point

- Se verá un ejemplo de como sincronizar un grupo de hilos en un determinado punto.
- Se enviará un objeto runnable también como parámetro, que deberá ser ejecutado luego de que todos los hilos lleguen al punto de sincronización.
- Se buscará un número en una matriz de números. La matriz sera dividida en submatrices. Por lo que cada hilo buscará por el número en una sub-matriz.

Synchronizing tasks in a common point

- Una vez que todos los hilos hayan concluído la búsqueda se ejecutará un reporte final para unificar los resultados.

- Clases: MatrixMock (esta clase generará la matrix con números randómicos)
  - Implementar el constructor cuyos parámetros serán:
    - Filas
    - Columnas
    - Número a buscar.

Synchronizing tasks in a common point

- Luego se debe llenar la matriz con números random y cada vez que el numero creado es igual al que se busca se debe incrementar un contador.

- Finalmente se imprime en pantalla el número de veces que esta en la matriz el número buscado. Finalidad comparar los resultados.

- Implementar el método getRow(). Recibe como parámetro un número de fila y devuelve esa fila.

Synchronizing tasks in a common point

- Clase: Results.
  - Esta clase almacenará en un arreglo el número de ocurrencias del número buscado en cada fila.
  - Implemente el constructor.
  - Implementar el método setData(), que recibe como parámetro una posición en el array y un valor y lo setea.
  - Implementar el método getData(), devuelve el array con los resultados.

Synchronizing tasks in a common point

- Clase: Searcher (runnable)
- Esta clase buscará un número en una determinada parte de la matriz.
- Declara varios atributos, entre ellos un objeto CyclicBarrier.
- Implementar el constructor.
- Implementar el método run.
  - Procesar las filas contando el número de encuentros.
  - Imprimir pantalla resultado y llamar método await()

Synchronizing tasks in a common point

- Clase: Grouper. Runnable.
- Encargada de unir los resultados de cada submatriz.
  - Implementar el constructor.
  - Implementar el método run().
  - Imprimir en pantalla el resultado.
- Clase: Main.
  - Declarar e inicializar tamaño de matriz y valor a buscar.
  - Crear un Grouper. Crear un Results. Crear un MatrixMock.

Synchronizing tasks in a common point

- Crear un CyclicBarrier.
- Crear 5 objetos Searchers y 5 hilos para ejecutarlos y lanzarlos.
- Ejecución.

Console

<terminated> Main (21) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe

```
Mock: There are 997310 ocurrences of number in generated data.
Thread-0: Processing lines from 0 to 2000.
Thread-4: Processing lines from 8000 to 10000.
Thread-3: Processing lines from 6000 to 8000.
Thread-2: Processing lines from 4000 to 6000.
Thread-1: Processing lines from 2000 to 4000.
Thread-2: Lines processed.
Thread-1: Lines processed.
Main: The main thread has finished.
Thread-4: Lines processed.
Thread-3: Lines processed.
Thread-0: Lines processed.
Grouper: Processing results...
Grouper: Total result: 997310.
```

Synchronizing tasks in a common point

- Conclusion:
- El problema resuelto es simple, una matriz grande de números enteros, se busca saber la cantidad de veces que se repite un número en la misma.
- Para mejorar performance, usamos técnica de divide y conquistarás.
- Con 5 hilos que buscan en 5 submatrices.
- Usamos un CyclicBarrier para sincronizar los 5 hilos y para ejecutar un runnable GROUPER que reúna la información e imprima en pantalla.

Synchronizing tasks in a common point

- La clase CyclicBarrier tiene un contador interno para controlar cuantos hilos deben arribar a ese punto a sincronizarse.
- Cada vez que uno llega ejecuta el método await() y la clase pone ese hilo a dormir.
- Una vez que llegaron todos los hilos, se los despierta a todos para que continúen. Y opcionalmente se crea un nuevo hilo para ejecutar el parámetro runnable enviado.

Synchronizing tasks in a common point

- La clase CyclicBarrier tiene otro método sobrecargado await() el cual puede recibir hasta dos parametros de tiempo (long, timeUnit).
- El hilo estará dormido en la barrera cíclica hasta que todos lleguen o se transcurra éste determinado tiempo.
- La clase CyclicBarrier también provee el método getNumberWaiting() devuelve el número de threads dormidos en la barrera.
- El método getParties() indica el número de tareas que serán sincronizadas con la barrera.

Synchronizing tasks in a common point

- La clase CyclicBarrier tiene una ventaja frente a los CountDownLatch, la misma puede ser RESETEADA a su valor de inicialización. (método reset())

- Si se ejecuta este método, todos los hilos dormidos en el await() reciben una excepción de BrokenBarrier. Debe ver como se maneja esta excepción.

Synchronizing tasks in a common point

- Finalmente cuando hay varios hilos esperando en el método await() y uno de ellos es interrumpido; este hilo recibe un interruptedException, pero los otros hilos que estaban esperando reciben un BrokenBarrierException, y el objeto CyclicBarrier es pasado a estado Broken.

- Existe un método para consultar éste estado y es isBroken(). Puede devolver true or false.