Prácticas Concurrencia y Distribución (17/18)

Arno Formella, Anália García Lourenço, Lorena Otero Cerdeira, David Olivieri

semana 9 – 15 abril

8. Semana 8 (09/04–15/04): *Semaphores* y *Thread Executors*

Objetivos: Implementar un semáforo y estudiar su comportamiento dentro del modelo de consumidor/productor.

1. **(P4: para entregar en grupo de práctica)**: Semáforos.

   Un semáforo es un objeto que controla el acceso a un recurso común. Antes de acceder al recurso, un hilo debe adquirir un permiso del semáforo. Después de terminar con el recurso, el hilo debe devolver el permiso al semáforo.

   a) Dado el siguiente código, completa el código necesario para los métodos `down()` y `up()`. Usa esta implementación en el código del productor/consumidor de la semana pasada.

   ```
   public class MySemaphore {
     public MySemaphore(int initialValue) {
       // POR HACER: Implementar esto
     }
     // Initializa un semaforo con el valor zero.
     public MySemaphore() {
       this(0);
     }

     public void down() {
       // POR HACER: Implementar esto
     }
     public void up() {
       // POR HACER: Implementar esto
     }
   }
   ```

   b) Vuelve a escribir el código del productor/consumidor (del código de la semana pasada), ahora con el tipo de datos `MySemaphore` para controlar la cola de bloqueo.

   c) La API de Java viene con una implementación de un semáforo. Reemplaza `MySemaphore` con la implementación de `Semaphore` de Java. Compara los resultados de cada una de las implementaciones. Explica el resultado ¿Se comporta como se esperaba?

2. **(P3: para entregar dentro de una semana)**: Estudio de dividir tareas concurrentes: dividir tareas en una matriz utilizando `Executor` y `ThreadPool`

   Cuando trabajamos con matrices (Semana 3), el diseño parecía ideal. El programa era altamente paralelo, y los hilos ni siquiera tenían que sincronizarse. En la práctica, sin embargo, aunque este diseño podría funcionar bien para matrices pequeñas, tendría un rendimiento muy bajo para matrices lo suficientemente

grandes como para ser interesante. La razón es que los hilos requieren memoria para el *stack* y otra información. Crear, programar y destruir hilos es costoso. Crear muchos hilos de corta duración es una forma ineficaz de organizar un cálculo multiproceso. Una forma más efectiva es con `ThreadPool`s.

Java tiene servicios especializados e interfaces para ejecutar grupos de hilos de manera eficiente. Consulta la documentación provista en Faitic o de libros sobre este tema.

a) Escribe una rutina principal para usar el código asociado en este ejemplo, que implemente la adición y multiplicación de la matriz con el servicio de `Executor`.
   - Explica cómo se dividen las tareas en este código. ¿Qué hace el objeto `Future` en este código?
   - Haz una gráfica del tiempo de ejecución como una función de la dimensión de la matriz. Compara esta gráfica con la de un programa secuencial para la multiplicación de matrices.
   - ¿Cuál es la limitación del método de división proporcionado? ¿Cómo podría mejorarse y generalizarse?

b) Modifica el código de la semana 3, problema 2, para usar el servicio de `Executor` de forma similar al que se usa en el código proporcionado para la suma y la multiplicación.

La clase `Matrix`:

```
import java.util.Random;

public class Matrix {
  double[][] data;
  int dim;
  int rowDisplace;
  int colDisplace;

  public Matrix(int d) {
    dim=d;
    data=new double[dim][dim];
  }
  private Matrix(double[][] matrix, int x, int y, int d) {
    data=matrix;
    dim=d;
    rowDisplace=x;
    colDisplace=y;
  }

  public void initialize() {
    Random rand=new Random();
    for(int i=0; i<dim; ++i){
      for(int j=0; j<dim; ++j){
        data[i][j]=(double)rand.nextInt(10);
      }
    }
  }

  public double get(int row, int col) {
    return data[row+rowDisplace][col+colDisplace];
  }
  public void set(int row, int col, double value) {
    data[row+rowDisplace][col+colDisplace]=value;
```

```
  }
  public int getDim() {
    return dim;
  }

  Matrix[][] split() {
    Matrix[][] result=new Matrix[2][2];
    final int newDim=dim/2;
    result[0][0]=
      new Matrix(data,rowDisplace,colDisplace,newDim);
    result[0][1]=
      new Matrix(data,rowDisplace,colDisplace+newDim,newDim);
    result[1][0]=
      new Matrix(data,rowDisplace+newDim,colDisplace,newDim);
    result[1][1]=
      new Matrix(data,rowDisplace+newDim,colDisplace+newDim,newDim);
    return result;
  }

  public String toString(){
    String ret="";
    for(int i=0; i<dim; ++i){
      for(int j=0; j<dim; ++j)
      ret+=data[i][j]+" ";
      ret+="\n";
    }
    return ret;
  }
}
```

La clase `MatrixTask`:

```
import java.util.concurrent.atomic.AtomicIntegerArray;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

public class MatrixTask {
  static ExecutorService exec=Executors.newCachedThreadPool();

  static Matrix add(Matrix a, Matrix b) throws ExecutionException {
    final int n=a.getDim();
    final Matrix c=new Matrix(n);
    Future<?> future=exec.submit(new AddTask(a,b,c));
    try {
      future.get();
    }
    catch(InterruptedException e) {
      e.printStackTrace();
    }
```

```
    catch(ExecutionException e) {
      e.printStackTrace();
    }

    // Finish the executor
    exec.shutdown();
    System.out.println("Exec: No more tasks");
    try {
      TimeUnit.SECONDS.sleep(1);
    }
    catch(InterruptedException e) {
      e.printStackTrace();
    }
    // The example finish
    System.out.println("Exec: return value");
    return c;
  }

  static class AddTask implements Runnable {
    Matrix a;
    Matrix b;
    Matrix c;

    public AddTask(Matrix a, Matrix b, Matrix c) {
      this.a=a;
      this.b=b;
      this.c=c;
    }
    public void run() {
      try {
        final int n=a.getDim();
        if(n==1) {
          c.set(0,0,a.get(0,0)+b.get(0,0));
        }
        else {
          final Matrix[][] aa=a.split();
          final Matrix[][] bb=b.split();
          final Matrix[][] cc=c.split();
          final Future<?>[][] future=(Future<?>[][]) new Future[2][2];
          for(int i=0; i<2; ++i)
            for(int j=0; j<2; ++j)
              future[i][j] =
                exec.submit(new AddTask(aa[i][j],bb[i][j],cc[i][j]));
          for(int i=0; i<2; ++i)
            for(int j=0; j<2; ++j)
              future[i][j].get();
        }
      }
      catch(Exception E) {
        E.printStackTrace();
      }
    }
  }
```

```
  static class MulTask implements Runnable {
    Matrix a;
    Matrix b;
    Matrix c;
    Matrix lhs;
    Matrix rhs;
    public MulTask(Matrix a, Matrix b, Matrix c) {
      this.a=a;
      this.b=b;
      this.c=c;
      this.lhs=new Matrix(a.getDim());
      this.rhs=new Matrix(a.getDim());
    }
    public void run() {
      try {
        if(a.getDim()==1) {
          c.set(0,0,a.get(0,0)*b.get(0,0));
        }
        else {
          final Matrix[][] aa=a.split();
          final Matrix[][] bb=b.split();
          final Matrix[][] cc=c.split();
          final Matrix[][] ll=lhs.split();
          final Matrix[][] rr=rhs.split();
          final Future<?>[][][] future=(Future<?>[][][])new Future[2][2][2];
          for(int i=0; i<2; ++i)
            for(int j=0; j<2; ++j) {
              future[i][j][0]=
                exec.submit(new MulTask(aa[i][0],bb[0][i],ll[i][j]));
              future[i][j][1]=
                exec.submit(new MulTask(aa[1][i],bb[i][1],rr[i][j]));
            }
          for(int i=0; i<2; ++i)
            for(int j=0; j<2; ++j)
              for(int k=0; k<2; ++k)
                future[i][j][k].get();
                Future<?> done=exec.submit(new AddTask(lhs,rhs,c));
                done.get();
        }
      }
      catch(Exception E) {
        E.printStackTrace();
      }
    }
  }
}
```