Java7 - Fork és Join

A Java 5 hozott néhány újdonságot a párhuzamos programozás terén, a Java 7 további könnyítéseket tartalmaz, amelyek közül az egyiket Fork/Join néven találjuk meg a dokumentációt böngészve, s a megoldás használatához csak három osztályt kell megismernünk:

- RecursiveAction – visszatérési érték nélküli feladat
- RecursiveTask – visszatérési értéket adó feladat
- ForkJoinPool – a szálakat kezel osztály (thread pool)

Az elnevezésekbl láthatjuk, hogy alapveten hosszabb ideig tartó rekurzív feladatokra találták ki ezt a technológiát, mint a fájlrendszer felderítése vagy egy weboldal letöltése, de remekül használható matematikai és/vagy fizikai szimulációkhoz is. A RecursiveTask dokumentációs oldalán egy tipikus rekurzív feladat: a Fibonacci számsor kiszámolása a példa, nézzük meg közelebbrl.

FibonacciTask.java

```
public class Fibonacci extends RecursiveTask<BigInteger>
{
  final Integer number;

  public Fibonacci(final Integer number)
  {
    this.number = number;
  }

  @Override
  public BigInteger compute()
  {
    if (number <= 1)
    {
      return new BigInteger("" + number);
    }

    final Fibonacci f1 = new Fibonacci(number - 1);
    f1.fork();

    final Fibonacci f2 = new Fibonacci(number - 2);
    f2.fork();

    return f2.join().add(f1.join());
  }
}
```

Két fontos dolgot kell észrevennünk:

- Az osztály a *RecursiveTask* osztályból származik és implementálja a *compute* metódust az átadott *BigInteger* típus szerint (mivelhogy nagy számokkal dolgozunk).
- Létrehozunk két példányt, amelyeket a *fork* használatával elindításra jelölünk, majd a *join* hívással várjuk meg az eredményeket.

A használatához kell egy új osztály, amely létrehozza a ForkJoinPool példányt, majd elindítja a folyamatot, ezt alább látjuk:

App.java

```
public class App
{
  public BigInteger computeFibonacci(Integer number)
  {
    final Fibonacci fibonacci = new Fibonacci(number);
    final ForkJoinPool fjPool = new ForkJoinPool(5);
    return fjPool.invoke(fibonacci);
  }
  public static void main(String[] args)
  {
    final Integer number = Integer.parseInt(args[0]);
    final App app = new App();
    System.out.println(app.computeFibonacci(number));
  }
}
```

Ez esetben a lényeg a *computeFibonacci* metódusban van, ahol létrehozunk egy új *Fibonacci* osztályt, egy új *ForkJoinPool* osztályt maximum öt szállal, majd egyszeren az *invoke* használatával elindítjuk a többszálú folyamatot. A szálak számáról és a *ForkJoinPool* gondoskodik, a szálak indításáról és lezárásáról pedig a *RecursiveTask* sosztályban megvalósított *fork* és *join* metódusok: a fejlesztnek egyszeren csak a feladatra kell koncentrálnia, minden mást megold a Java 7.

Megfelelen nagy számmal indítva – az 40 már bven sok – láthatjuk, hogy egy négymagos (nyolcutas) i7 processzort szépen kiterhel öt szálon az adott java processz (495%):

```
PID  USER      PR  NI  VIRT  RES  SHR S   %CPU %MEM    TIME+  COMMAND
7268 auth.gab  20   0 1191m 391m 7024 S    495  5.0   8:26.22 java
```

Ha megnöveljük a használt szálak számát, akkor egészen a processzor végs határig tudjuk növelni a párhuzamosságot (736%):

```
PID  USER      PR  NI  VIRT  RES  SHR S   %CPU %MEM    TIME+  COMMAND
7442 auth.gab  20   0 1192m 369m 7008 S    736  4.7   1:52.95 java
```

A kapott eredményt leellenrizhetjük a maths.surrey.ac.uk adott oldalán; 40 esetén ez 102334155 kell legyen.