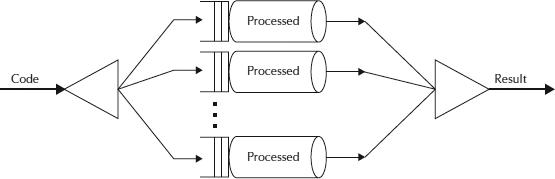
**Fork/Join Framework Tutorial: ForkJoinPool Example**

The effective use of parallel cores in a Java program has always been a challenge. There were few home-grown frameworks that would distribute the work across multiple cores and then join them to return the result set. Java 7 has incorporated this feature as a [**Fork and Join framework**](https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html).

Basically the **Fork-Join breaks the task at hand into mini-tasks** until the mini-task is simple enough that it can be solved without further breakups. It’s **like a**[**divide-and-conquer algorithm**](https://en.wikipedia.org/wiki/Divide_and_conquer_algorithms). One important concept to note in this framework is that **ideally no worker thread is idle**. They implement a **work-stealing algorithm** in that idle workers steal the work from those workers who are busy.

Fork Join Framework

It’s based on the work of Doug Lea, a thought leader on Java concurrency. Fork/Join deals with the threading hassles; you just indicate to the framework which portions of the work can be broken apart and handled recursively. It employs pseudocode (as taken from Doug Lea’s paper on the subject):

Result solve(Problem problem) {

if (problem is small)

directly solve problem

else {

split problem into independent parts

fork new subtasks to solve each part

join all subtasks

compose result from subresults

}

}

**Discussion Points**

1) [Core Classes used in Fork/Join Framework](https://howtodoinjava.com/java7/forkjoin-framework-tutorial-forkjoinpool-example/#Core_Classes)

i) [ForkJoinPool](https://howtodoinjava.com/java7/forkjoin-framework-tutorial-forkjoinpool-example/#ForkJoinPool)

ii) [ForkJoinTask](https://howtodoinjava.com/java7/forkjoin-framework-tutorial-forkjoinpool-example/#ForkJoinTask)

2) [Example Implementations of Fork/Join Pool Framework](https://howtodoinjava.com/java7/forkjoin-framework-tutorial-forkjoinpool-example/#Example_Impl)

i) [Implementation Sourcecode](https://howtodoinjava.com/java7/forkjoin-framework-tutorial-forkjoinpool-example/#Sourcecode)

ii) [How it works?](https://howtodoinjava.com/java7/forkjoin-framework-tutorial-forkjoinpool-example/#How_it_works)

3) [Difference between Fork/Join Framework And ExecutorService](https://howtodoinjava.com/java7/forkjoin-framework-tutorial-forkjoinpool-example/#Diff_ForkJoin_ExecutorService)

4) [Existing Implementations in JDK](https://howtodoinjava.com/java7/forkjoin-framework-tutorial-forkjoinpool-example/#Existing_Implementations)

5) [Conclusion](https://howtodoinjava.com/java7/forkjoin-framework-tutorial-forkjoinpool-example/#Conclusion)

**Core Classes used in Fork/Join Framework**

The core classes supporting the Fork-Join mechanism are [ForkJoinPool](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.html" \o "ForkJoinPool" \t "_blank) and [ForkJoinTask](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinTask.html" \o "ForkJoinTask" \t "_blank).

Let’s learn about their roles in detail.

**ForkJoinPool**

The ForkJoinPool is basically a specialized implementation of ExecutorService implementing the work-stealing algorithm we talked about above. We create an instance of ForkJoinPool by providing the target parallelism level i.e. the **number of processors** as shown below:

ForkJoinPool pool = new ForkJoinPool(numberOfProcessors);

Where numberOfProcessors = Runtime.getRunTime().availableProcessors();

If you use a no-argument constructor, by default, it creates a pool of size that equals the number of available processors obtained using above technique.

Although you specify any initial pool size, the **pool adjusts its size dynamically in an attempt to maintain enough active threads** at any given point in time. Another important difference compared to other ExecutorService's is that this pool need not be explicitly shutdown upon program exit because all its threads are in daemon mode.

There are three different ways of submitting a task to the ForkJoinPool.

**1) execute() method** //Desired asynchronous execution; call its fork method to split the work between multiple threads.  
**2) invoke() method**: //Await to obtain the result; call the invoke method on the pool.  
**3) submit() method**: //Returns a Future object that you can use for checking status and obtaining the result on its completion.

**ForkJoinTask**

This is an abstract class for creating tasks that run within a ForkJoinPool. The Recursiveaction and RecursiveTask are the only two direct, known subclasses of ForkJoinTask. The only difference between these two classes is that the RecursiveAction does not return a value while RecursiveTask does have a return value and returns an object of specified type.

In both cases, you would need to implement the compute method in your subclass that performs the main computation desired by the task.

The ForkJoinTask class provides several methods for checking the execution status of a task. The **isDone()** method returns true if a task completes in any way. The **isCompletedNormally()** method returns true if a task completes without cancellation or encountering an exception, and **isCancelled()** returns true if the task was cancelled. Lastly, **isCompletedabnormally()** returns true if the task was either cancelled or encountered an exception.

**Example Implementations of Fork/Join Pool Framework**

In this example, you will learn how to use the asynchronous methods provided by the ForkJoinPool and ForkJoinTask classes for the management of tasks. You are going to implement a **program that will search for files with a determined extension inside a folder and its subfolders**. The ForkJoinTask class you’re going to implement will process the content of a folder. For each subfolder inside that folder, it will send a new task to the ForkJoinPool class in an asynchronous way. For each file inside that folder, the task will check the extension of the file and add it to the result list if it proceeds.

The solution to above problem is implemented in FolderProcessor class, which is given below:

**Implementation Sourcecode**

**FolderProcessor.java**

package forkJoinDemoAsyncExample;

import java.io.File;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.RecursiveTask;

public class FolderProcessor extends RecursiveTask<List<String>>

{

private static final long serialVersionUID = 1L;

//This attribute will store the full path of the folder this task is going to process.

private final String path;

//This attribute will store the name of the extension of the files this task is going to look for.

private final String extension;

//Implement the constructor of the class to initialize its attributes

public FolderProcessor(String path, String extension)

{

this.path = path;

this.extension = extension;

}

//Implement the compute() method. As you parameterized the RecursiveTask class with the List<String> type,

//this method has to return an object of that type.

@Override

protected List<String> compute()

{

//List to store the names of the files stored in the folder.

List<String> list = new ArrayList<String>();

//FolderProcessor tasks to store the subtasks that are going to process the subfolders stored in the folder

List<FolderProcessor> tasks = new ArrayList<FolderProcessor>();

//Get the content of the folder.

File file = new File(path);

File content[] = file.listFiles();

//For each element in the folder, if there is a subfolder, create a new FolderProcessor object

//and execute it asynchronously using the fork() method.

if (content != null)

{

for (int i = 0; i < content.length; i++)

{

if (content[i].isDirectory())

{

FolderProcessor task = new FolderProcessor(content[i].getAbsolutePath(), extension);

task.fork();

tasks.add(task);

}

//Otherwise, compare the extension of the file with the extension you are looking for using the checkFile() method

//and, if they are equal, store the full path of the file in the list of strings declared earlier.

else

{

if (checkFile(content[i].getName()))

{

list.add(content[i].getAbsolutePath());

}

}

}

}

//If the list of the FolderProcessor subtasks has more than 50 elements,

//write a message to the console to indicate this circumstance.

if (tasks.size() > 50)

{

System.out.printf("%s: %d tasks ran.\n", file.getAbsolutePath(), tasks.size());

}

//add to the list of files the results returned by the subtasks launched by this task.

addResultsFromTasks(list, tasks);

//Return the list of strings

return list;

}

//For each task stored in the list of tasks, call the join() method that will wait for its finalization and then will return the result of the task.

//Add that result to the list of strings using the addAll() method.

private void addResultsFromTasks(List<String> list, List<FolderProcessor> tasks)

{

for (FolderProcessor item : tasks)

{

list.addAll(item.join());

}

}

//This method compares if the name of a file passed as a parameter ends with the extension you are looking for.

private boolean checkFile(String name)

{

return name.endsWith(extension);

}

}

And to use above FolderProcessor, follow below code:

**Main.java**

package forkJoinDemoAsyncExample;

import java.util.List;

import java.util.concurrent.ForkJoinPool;

import java.util.concurrent.TimeUnit;

public class Main

{

public static void main(String[] args)

{

//Create ForkJoinPool using the default constructor.

ForkJoinPool pool = new ForkJoinPool();

//Create three FolderProcessor tasks. Initialize each one with a different folder path.

FolderProcessor system = new FolderProcessor("C:\\Windows", "log");

FolderProcessor apps = new FolderProcessor("C:\\Program Files", "log");

FolderProcessor documents = new FolderProcessor("C:\\Documents And Settings", "log");

//Execute the three tasks in the pool using the execute() method.

pool.execute(system);

pool.execute(apps);

pool.execute(documents);

//Write to the console information about the status of the pool every second

//until the three tasks have finished their execution.

do

{

System.out.printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

System.out.printf("Main: Parallelism: %d\n", pool.getParallelism());

System.out.printf("Main: Active Threads: %d\n", pool.getActiveThreadCount());

System.out.printf("Main: Task Count: %d\n", pool.getQueuedTaskCount());

System.out.printf("Main: Steal Count: %d\n", pool.getStealCount());

System.out.printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

try

{

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e)

{

e.printStackTrace();

}

} while ((!system.isDone()) || (!apps.isDone()) || (!documents.isDone()));

//Shut down ForkJoinPool using the shutdown() method.

pool.shutdown();

//Write the number of results generated by each task to the console.

List<String> results;

results = system.join();

System.out.printf("System: %d files found.\n", results.size());

results = apps.join();

System.out.printf("Apps: %d files found.\n", results.size());

results = documents.join();

System.out.printf("Documents: %d files found.\n", results.size());

}

}

Output of above program will look like this:

Main: Parallelism: 2

Main: Active Threads: 3

Main: Task Count: 1403

Main: Steal Count: 5551

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Main: Parallelism: 2

Main: Active Threads: 3

Main: Task Count: 586

Main: Steal Count: 5551

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**System: 337 files found.**

**Apps: 10 files found.**

**Documents: 0 files found.**

**How it works?**

In the FolderProcessor class, Each task processes the content of a folder. As you know, this content has the following two kinds of elements:

* Files
* Other folders

**If the task finds a folder, it creates another Task object to process that folder and sends it to the pool using the fork() method**. This method sends the task to the pool that will execute it if it has a free worker-thread or it can create a new one. The **method returns immediately, so the task can continue processing** the content of the folder. For every file, a task compares its extension with the one it’s looking for and, if they are equal, adds the name of the file to the list of results.

**Once the task has processed all the content of the assigned folder, it waits for the finalization of all the tasks it sent to the pool using the join() method**. This method called in a task waits for the finalization of its execution and returns the value returned by the **compute()** method. The task groups the results of all the tasks it sent with its own results and returns that list as a return value of the compute() method.

**Difference between Fork/Join Framework And ExecutorService**

The **main difference between the Fork/Join and the Executor frameworks is the work-stealing algorithm**. Unlike the Executor framework, when a task is waiting for the finalization of the sub-tasks it has created using the join operation, the thread that is executing that task (called worker thread ) looks for other tasks that have not been executed yet and begins its execution. By this way, the threads take full advantage of their running time, thereby improving the performance of the application.

**Existing Implementations in JDK**

There are some generally useful features in Java SE which are already implemented using the fork/join framework.

**1)** One such implementation, introduced in Java SE 8, is used by the **java.util.Arrays class for its parallelSort() methods**. These methods are similar to sort(), but leverage concurrency via the fork/join framework. Parallel sorting of large arrays is faster than sequential sorting when run on multiprocessor systems.

**2)** Parallelism used in **Stream.parallel()**. Read more about this [parallel stream operation in java 8](https://howtodoinjava.com/java8/java-8-tutorial-streams-by-examples/).

**Conclusion**

Designing good multi-threaded algorithms is hard, and **fork/join doesn’t work in every circumstance**. It’s very useful within its own domain of applicability, but in the end, you have to decide whether your problem fits within the framework, and if not, **you must be prepared to develop your own solution**, building on the superb tools provided by java.util.concurrent package.