**Assignment 7 (Problem 2.2)**

**Rasika Mohod**

rmohod@gmu.edu

G01044774

**a)**

**Java thread pool**

Java thread pool manages the pool of worker threads, it contains a queue that keeps tasks waiting to get executed. We can use *ThreadPoolExecutor* to create thread pool in java. Java thread pool manages the collection of Runnable threads and worker threads execute Runnable from the queue.

Suppose for an **example** in a program, we create fixed size thread pool of 5 worker threads. And then we submit 10 jobs to this pool. Now since the pool size is 5, it will start working on 5 jobs and other jobs will be in ‘wait’ state. As soon as one of the job is finished, another job from the wait queue will be picked up by worker thread and will get executed.

**ThreadPoolExecutor**

The java.util.concurrent.ThreadPoolExecutor is an implementation of the ExecutorService interface. The ThreadPoolExecutor executes the given task (Callable or Runnable) using one of its internally pooled threads. The thread pool contained inside the ThreadPoolExecutor can contain a varying amount of threads. The number of threads in the pool is determined by the variables: corePoolSize & maximumPoolSize.

* If less than corePoolSize threads are created in the thread pool when a task is delegated to the thread pool, then a new thread is created, even if idle threads exist in the pool.
* If the internal queue of tasks is full, and corePoolSize threads or more are running, but less than maximumPoolSize threads are running, then a new thread is created to execute the task.

**ExecutorService**

The java.util.concurrent.ExecutorService interface represents an asynchronous execution mechanism which is capable of executing tasks in the background. An ExecutorService is thus very similar to a thread pool. In fact, the implementation of ExecutorService present in the java.util.concurrent package is a thread pool implementation. Once the thread has delegated the task to the ExecutorService, the thread continues its own execution independent of the execution of that task.

Since ExecutorService is an interface, you need to initialize its implementations in order to make any use of it. The ExecutorService has the following implementation in the java.util.concurrent package:

* ThreadPoolExecutor
* ScheduledThreadPoolExecutor

There are a few different ways to delegate tasks for execution to an ExecutorService:

* execute(Runnable)
* submit(Runnable)
* submit(Callable)
* invokeAny(...)
* invokeAll(...)

**b)**

**Output 1:**

Input Number of Tasks = 5

Input factorial (n) = 5

Input Thread Pool Size = 20

**Execution 1:**

Input Number of Tasks = 5

Input factorial (n) = 5

Input Thread Pool Size = 20

Starting task Task 4 at 15: 17: 53: 667

Starting task Task 0 at 15: 17: 53: 667

Starting task Task 1 at 15: 17: 53: 667

Starting task Task 2 at 15: 17: 53: 667

Ending task Task 1 at 15: 17: 53: 667 after 0 milliseconds

Starting task Task 3 at 15: 17: 53: 667

Ending task Task 2 at 15: 17: 53: 667 after 0 milliseconds

Ending task Task 0 at 15: 17: 53: 667 after 0 milliseconds

Ending task Task 4 at 15: 17: 53: 667 after 0 milliseconds

Ending task Task 3 at 15: 17: 53: 667 after 0 milliseconds

**Execution 2:**

Input Number of Tasks = 5

Input factorial (n) = 5

Input Thread Pool Size = 20

Starting task Task 4 at 15: 19: 09: 112

Starting task Task 1 at 15: 19: 09: 112

Ending task Task 4 at 15: 19: 09: 112 after 0 milliseconds

Starting task Task 3 at 15: 19: 09: 112

Starting task Task 0 at 15: 19: 09: 112

Starting task Task 2 at 15: 19: 09: 112

Ending task Task 0 at 15: 19: 09: 126 after 14 milliseconds

Ending task Task 3 at 15: 19: 09: 126 after 14 milliseconds

Ending task Task 1 at 15: 19: 09: 112 after 0 milliseconds

Ending task Task 2 at 15: 19: 09: 126 after 14 milliseconds

**Execution 3:**

Input Number of Tasks = 5

Input factorial (n) = 5

Input Thread Pool Size = 20

Starting task Task 2 at 15: 19: 52: 718

Starting task Task 1 at 15: 19: 52: 718

Starting task Task 3 at 15: 19: 52: 718

Starting task Task 4 at 15: 19: 52: 718

Ending task Task 3 at 15: 19: 52: 718 after 0 milliseconds

Starting task Task 0 at 15: 19: 52: 718

Ending task Task 4 at 15: 19: 52: 718 after 0 milliseconds

Ending task Task 1 at 15: 19: 52: 718 after 0 milliseconds

Ending task Task 2 at 15: 19: 52: 718 after 0 milliseconds

Ending task Task 0 at 15: 19: 52: 730 after 12 milliseconds

**Output 2:**

Input Number of Tasks = 15

Input factorial (n) = 8

Input Thread Pool Size = 9

**Execution 1:**

Input Number of Tasks = 15

Input factorial (n) = 8

Input Thread Pool Size = 9

Starting task Task 7 at 12: 46: 32: 856

Starting task Task 1 at 12: 46: 32: 856

Ending task Task 1 at 12: 46: 32: 856 after 0 milliseconds

Starting task Task 4 at 12: 46: 32: 856

Starting task Task 2 at 12: 46: 32: 856

Starting task Task 0 at 12: 46: 32: 856

Starting task Task 6 at 12: 46: 32: 856

Starting task Task 5 at 12: 46: 32: 856

Starting task Task 8 at 12: 46: 32: 856

Ending task Task 5 at 12: 46: 32: 860 after 4 milliseconds

Ending task Task 6 at 12: 46: 32: 860 after 4 milliseconds

Ending task Task 0 at 12: 46: 32: 860 after 4 milliseconds

Ending task Task 2 at 12: 46: 32: 860 after 4 milliseconds

Starting task Task 11 at 12: 46: 32: 860

Starting task Task 12 at 12: 46: 32: 860

Starting task Task 9 at 12: 46: 32: 860

Starting task Task 13 at 12: 46: 32: 860

Ending task Task 9 at 12: 46: 32: 860 after 0 milliseconds

Ending task Task 4 at 12: 46: 32: 856 after 0 milliseconds

Starting task Task 3 at 12: 46: 32: 856

Ending task Task 7 at 12: 46: 32: 856 after 0 milliseconds

Starting task Task 14 at 12: 46: 32: 861

Ending task Task 3 at 12: 46: 32: 861 after 5 milliseconds

Ending task Task 13 at 12: 46: 32: 861 after 1 milliseconds

Ending task Task 12 at 12: 46: 32: 860 after 0 milliseconds

Starting task Task 10 at 12: 46: 32: 860

Ending task Task 11 at 12: 46: 32: 860 after 0 milliseconds

Ending task Task 8 at 12: 46: 32: 860 after 4 milliseconds

Ending task Task 10 at 12: 46: 32: 861 after 1 milliseconds

Ending task Task 14 at 12: 46: 32: 861 after 0 milliseconds

**Execution 2:**

Input Number of Tasks = 15

Input factorial (n) = 8

Input Thread Pool Size = 9

Starting task Task 3 at 12: 47: 14: 315

Starting task Task 5 at 12: 47: 14: 315

Starting task Task 6 at 12: 47: 14: 315

Starting task Task 0 at 12: 47: 14: 327

Ending task Task 6 at 12: 47: 14: 327 after 12 milliseconds

Starting task Task 8 at 12: 47: 14: 315

Ending task Task 0 at 12: 47: 14: 327 after 0 milliseconds

Ending task Task 8 at 12: 47: 14: 327 after 12 milliseconds

Starting task Task 4 at 12: 47: 14: 315

Starting task Task 9 at 12: 47: 14: 327

Starting task Task 7 at 12: 47: 14: 315

Ending task Task 9 at 12: 47: 14: 327 after 0 milliseconds

Ending task Task 7 at 12: 47: 14: 327 after 12 milliseconds

Starting task Task 2 at 12: 47: 14: 315

Ending task Task 5 at 12: 47: 14: 327 after 12 milliseconds

Ending task Task 3 at 12: 47: 14: 315 after 0 milliseconds

Starting task Task 13 at 12: 47: 14: 327

Ending task Task 2 at 12: 47: 14: 327 after 12 milliseconds

Starting task Task 14 at 12: 47: 14: 327

Starting task Task 12 at 12: 47: 14: 327

Starting task Task 11 at 12: 47: 14: 327

Ending task Task 4 at 12: 47: 14: 327 after 12 milliseconds

Ending task Task 11 at 12: 47: 14: 328 after 1 milliseconds

Starting task Task 10 at 12: 47: 14: 327

Starting task Task 1 at 12: 47: 14: 327

Ending task Task 10 at 12: 47: 14: 328 after 1 milliseconds

Ending task Task 12 at 12: 47: 14: 328 after 1 milliseconds

Ending task Task 14 at 12: 47: 14: 328 after 1 milliseconds

Ending task Task 13 at 12: 47: 14: 327 after 0 milliseconds

Ending task Task 1 at 12: 47: 14: 328 after 1 milliseconds

**Execution 3:**

Input Number of Tasks = 15

Input factorial (n) = 8

Input Thread Pool Size = 9

Starting task Task 4 at 12: 47: 35: 015

Ending task Task 4 at 12: 47: 35: 015 after 0 milliseconds

Starting task Task 5 at 12: 47: 35: 015

Ending task Task 5 at 12: 47: 35: 026 after 11 milliseconds

Starting task Task 1 at 12: 47: 35: 015

Starting task Task 10 at 12: 47: 35: 026

Starting task Task 7 at 12: 47: 35: 015

Starting task Task 6 at 12: 47: 35: 015

Ending task Task 7 at 12: 47: 35: 026 after 11 milliseconds

Starting task Task 8 at 12: 47: 35: 026

Ending task Task 8 at 12: 47: 35: 026 after 0 milliseconds

Starting task Task 9 at 12: 47: 35: 026

Starting task Task 11 at 12: 47: 35: 026

Ending task Task 9 at 12: 47: 35: 026 after 0 milliseconds

Starting task Task 12 at 12: 47: 35: 026

Starting task Task 3 at 12: 47: 35: 026

Ending task Task 12 at 12: 47: 35: 026 after 0 milliseconds

Starting task Task 0 at 12: 47: 35: 026

Ending task Task 6 at 12: 47: 35: 026 after 11 milliseconds

Ending task Task 0 at 12: 47: 35: 026 after 0 milliseconds

Starting task Task 2 at 12: 47: 35: 026

Ending task Task 10 at 12: 47: 35: 026 after 0 milliseconds

Ending task Task 2 at 12: 47: 35: 027 after 1 milliseconds

Ending task Task 1 at 12: 47: 35: 026 after 11 milliseconds

Starting task Task 14 at 12: 47: 35: 026

Ending task Task 3 at 12: 47: 35: 026 after 0 milliseconds

Starting task Task 13 at 12: 47: 35: 026

Ending task Task 11 at 12: 47: 35: 026 after 0 milliseconds

Ending task Task 13 at 12: 47: 35: 027 after 1 milliseconds

Ending task Task 14 at 12: 47: 35: 027 after 1 milliseconds

**Output 3:**

Input Number of Tasks = 20

Input factorial (n) = 8

Input Thread Pool Size = 15

**Execution 1:**

Input Number of Tasks = 20

Input factorial (n) = 8

Input Thread Pool Size = 15

Starting task Task 5 at 15: 26: 29: 492

Ending task Task 5 at 15: 26: 29: 492 after 0 milliseconds

Starting task Task 0 at 15: 26: 29: 492

Starting task Task 15 at 15: 26: 29: 501

Starting task Task 6 at 15: 26: 29: 492

Ending task Task 15 at 15: 26: 29: 501 after 0 milliseconds

Ending task Task 6 at 15: 26: 29: 501 after 9 milliseconds

Starting task Task 9 at 15: 26: 29: 501

Ending task Task 9 at 15: 26: 29: 501 after 0 milliseconds

Starting task Task 17 at 15: 26: 29: 501

Starting task Task 1 at 15: 26: 29: 492

Ending task Task 1 at 15: 26: 29: 501 after 9 milliseconds

Starting task Task 13 at 15: 26: 29: 492

Starting task Task 8 at 15: 26: 29: 492

Ending task Task 13 at 15: 26: 29: 501 after 9 milliseconds

Starting task Task 11 at 15: 26: 29: 492

Ending task Task 8 at 15: 26: 29: 501 after 9 milliseconds

Starting task Task 4 at 15: 26: 29: 492

Starting task Task 19 at 15: 26: 29: 501

Starting task Task 3 at 15: 26: 29: 501

Starting task Task 18 at 15: 26: 29: 501

Ending task Task 17 at 15: 26: 29: 501 after 0 milliseconds

Ending task Task 18 at 15: 26: 29: 502 after 1 milliseconds

Starting task Task 12 at 15: 26: 29: 492

Starting task Task 16 at 15: 26: 29: 501

Starting task Task 7 at 15: 26: 29: 492

Starting task Task 2 at 15: 26: 29: 492

Ending task Task 0 at 15: 26: 29: 501 after 9 milliseconds

Starting task Task 14 at 15: 26: 29: 492

Starting task Task 10 at 15: 26: 29: 492

Ending task Task 14 at 15: 26: 29: 502 after 10 milliseconds

Ending task Task 2 at 15: 26: 29: 502 after 10 milliseconds

Ending task Task 7 at 15: 26: 29: 502 after 10 milliseconds

Ending task Task 16 at 15: 26: 29: 502 after 1 milliseconds

Ending task Task 12 at 15: 26: 29: 502 after 10 milliseconds

Ending task Task 3 at 15: 26: 29: 502 after 1 milliseconds

Ending task Task 19 at 15: 26: 29: 502 after 1 milliseconds

Ending task Task 4 at 15: 26: 29: 502 after 10 milliseconds

Ending task Task 11 at 15: 26: 29: 501 after 9 milliseconds

Ending task Task 10 at 15: 26: 29: 502 after 10 milliseconds

**Execution 2:**

Input Number of Tasks = 20

Input factorial (n) = 8

Input Thread Pool Size = 15

Starting task Task 8 at 15: 27: 53: 245

Starting task Task 9 at 15: 27: 53: 245

Starting task Task 6 at 15: 27: 53: 245

Ending task Task 9 at 15: 27: 53: 245 after 0 milliseconds

Starting task Task 13 at 15: 27: 53: 245

Ending task Task 13 at 15: 27: 53: 245 after 0 milliseconds

Starting task Task 15 at 15: 27: 53: 245

Starting task Task 16 at 15: 27: 53: 245

Starting task Task 11 at 15: 27: 53: 245

Ending task Task 11 at 15: 27: 53: 245 after 0 milliseconds

Starting task Task 1 at 15: 27: 53: 245

Starting task Task 7 at 15: 27: 53: 245

Starting task Task 3 at 15: 27: 53: 245

Ending task Task 7 at 15: 27: 53: 245 after 0 milliseconds

Ending task Task 3 at 15: 27: 53: 245 after 0 milliseconds

Starting task Task 5 at 15: 27: 53: 245

Starting task Task 19 at 15: 27: 53: 245

Starting task Task 4 at 15: 27: 53: 245

Ending task Task 19 at 15: 27: 53: 245 after 0 milliseconds

Ending task Task 4 at 15: 27: 53: 245 after 0 milliseconds

Ending task Task 1 at 15: 27: 53: 245 after 0 milliseconds

Starting task Task 12 at 15: 27: 53: 245

Starting task Task 17 at 15: 27: 53: 245

Starting task Task 2 at 15: 27: 53: 245

Starting task Task 0 at 15: 27: 53: 245

Ending task Task 2 at 15: 27: 53: 245 after 0 milliseconds

Ending task Task 16 at 15: 27: 53: 245 after 0 milliseconds

Starting task Task 10 at 15: 27: 53: 245

Ending task Task 15 at 15: 27: 53: 245 after 0 milliseconds

Ending task Task 10 at 15: 27: 53: 245 after 0 milliseconds

Ending task Task 6 at 15: 27: 53: 245 after 0 milliseconds

Starting task Task 14 at 15: 27: 53: 245

Ending task Task 8 at 15: 27: 53: 245 after 0 milliseconds

Ending task Task 14 at 15: 27: 53: 245 after 0 milliseconds

Ending task Task 0 at 15: 27: 53: 245 after 0 milliseconds

Ending task Task 17 at 15: 27: 53: 245 after 0 milliseconds

Ending task Task 12 at 15: 27: 53: 245 after 0 milliseconds

Ending task Task 5 at 15: 27: 53: 245 after 0 milliseconds

Starting task Task 18 at 15: 27: 53: 245

Ending task Task 18 at 15: 27: 53: 245 after 0 milliseconds

**Execution 3:**

Input Number of Tasks = 20

Input factorial (n) = 8

Input Thread Pool Size = 15

Starting task Task 0 at 15: 28: 57: 595

Starting task Task 8 at 15: 28: 57: 595

Starting task Task 9 at 15: 28: 57: 595

Starting task Task 12 at 15: 28: 57: 595

Starting task Task 3 at 15: 28: 57: 595

Starting task Task 13 at 15: 28: 57: 595

Ending task Task 3 at 15: 28: 57: 605 after 10 milliseconds

Ending task Task 13 at 15: 28: 57: 605 after 10 milliseconds

Starting task Task 1 at 15: 28: 57: 605

Ending task Task 1 at 15: 28: 57: 605 after 0 milliseconds

Starting task Task 15 at 15: 28: 57: 605

Starting task Task 10 at 15: 28: 57: 595

Ending task Task 15 at 15: 28: 57: 605 after 0 milliseconds

Ending task Task 10 at 15: 28: 57: 605 after 10 milliseconds

Starting task Task 14 at 15: 28: 57: 595

Ending task Task 14 at 15: 28: 57: 605 after 10 milliseconds

Starting task Task 6 at 15: 28: 57: 595

Starting task Task 4 at 15: 28: 57: 595

Ending task Task 6 at 15: 28: 57: 605 after 10 milliseconds

Starting task Task 5 at 15: 28: 57: 595

Starting task Task 7 at 15: 28: 57: 605

Starting task Task 2 at 15: 28: 57: 595

Starting task Task 19 at 15: 28: 57: 605

Starting task Task 18 at 15: 28: 57: 605

Starting task Task 17 at 15: 28: 57: 605

Starting task Task 16 at 15: 28: 57: 605

Ending task Task 12 at 15: 28: 57: 605 after 10 milliseconds

Ending task Task 9 at 15: 28: 57: 605 after 10 milliseconds

Ending task Task 8 at 15: 28: 57: 595 after 0 milliseconds

Starting task Task 11 at 15: 28: 57: 595

Ending task Task 0 at 15: 28: 57: 595 after 0 milliseconds

Ending task Task 11 at 15: 28: 57: 606 after 11 milliseconds

Ending task Task 16 at 15: 28: 57: 606 after 1 milliseconds

Ending task Task 17 at 15: 28: 57: 606 after 1 milliseconds

Ending task Task 18 at 15: 28: 57: 606 after 1 milliseconds

Ending task Task 19 at 15: 28: 57: 606 after 1 milliseconds

Ending task Task 2 at 15: 28: 57: 606 after 11 milliseconds

Ending task Task 7 at 15: 28: 57: 606 after 1 milliseconds

Ending task Task 5 at 15: 28: 57: 606 after 11 milliseconds

Ending task Task 4 at 15: 28: 57: 606 after 11 milliseconds

**c)**

**Analysis:**

The code logic here is written to perform the task of finding the nth Fibonacci series number and not the Factorial of a given number. This task is performed with different threads and also the time taken by each thread from its start to its completion is calculated.

Here the ThreadPoolExecutor is used. The number of tasks to be performed is taken as input from user; the number, n: the term to be found in Fibonacci series, is obtained as input from user and also the allowed thread pool size is taken as input from user.

The code creates a ThreadPoolExecutor object - **ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue)** with corePoolSize and maximumPoolSize set as the thread pool size given by user, the timeunit as milliseconds and LinkedBlockingQueue.

Later a tasks array is created of size as number of tasks given as input from user and each task is initialized and executed one after the other. Here we observe that a pool is a threshold which can contain the number of task. Even if the number of tasks is more, only the limited amount of tasks are allowed to be executed at a time. If one of the tasks in the pool finishes execution, the next random task enters the pool to start its execution. This pattern goes on till all the tasks are exhausted.

**d)**

Output 1:

Here the thread pool size is set way too large than the tasks. Hence, the tasks for their execution don’t have to wait for other tasks to finish. All the tasks (obviously) enter the pool at the same time, and randomly execute themselves. In this case the priority was also not implemented, so in true sense, the tasks were implemented in random order.

Output 2:

Here the pool size is less than the no. of threads/tasks. So at a time 9 tasks are randomly selected. They start the execution at the same time. Even if one task finishes, another task (waiting in the line), enters the pool and starts its execution. This happens till all the 15 tasks are completed.

Output 3:

The number of tasks is almost comparable to the thread pool size as input. In this case, even though the allowable pool size is 15, the processes that enter the pool would not have to wait till the pool fills up. Because the number of tasks is 20, as the some of the tasks execute at the same time the pool will be empty enough to occupy new tasks to try to keep the pool filled up, thus the pool will never fill up.

As observed, the combination of inputs given in Output 2 case is an optimum case to understand the concept of pooling. It shows the full utilization of the resources as the pool is always full. However, the Output 1 case does not utilize the concept of a thread pooling as most of it is always empty and unnecessarily resources are reserved, while the Output 3 case partially utilizes the pool concept.