Thread Pools in Java

Difficulty Level : Medium Last Updated : 30 Jul, 2020

Background

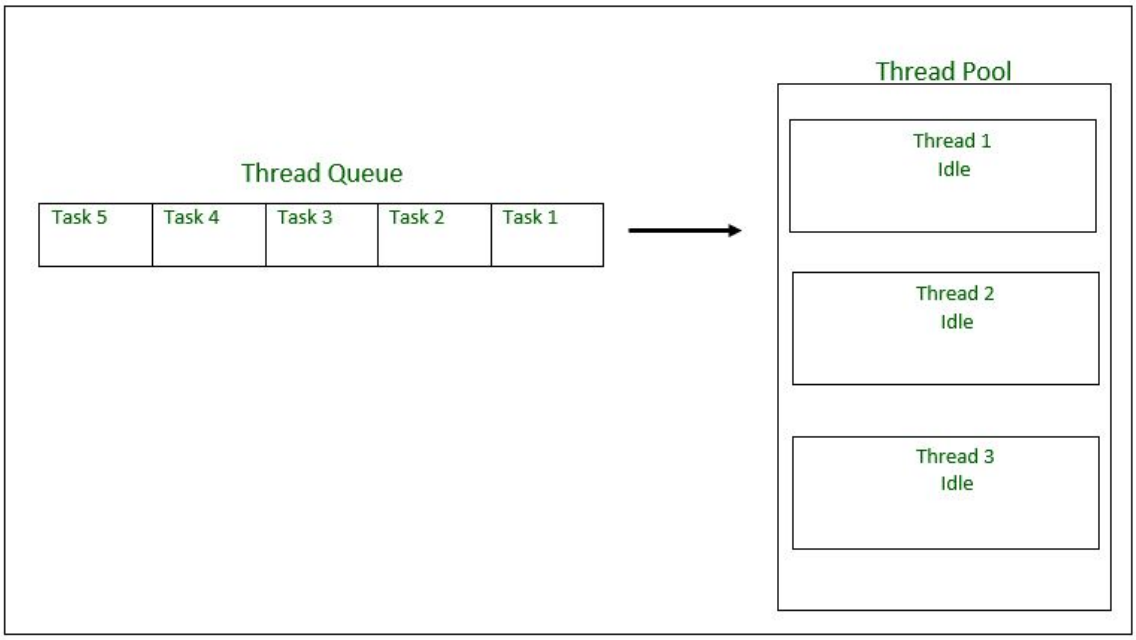
Server Programs such as database and web servers repeatedly execute requests from multiple clients and these are oriented around processing a large number of short tasks. An approach for building a server application would be to create a new thread each time a request arrives and service this new request in the newly created thread. While this approach seems simple to implement, it has significant disadvantages. A server that creates a new thread for every request would spend more time and consume more system resources in creating and destroying threads than processing actual requests.

Since active threads consume system resources, a JVM creating too many threads at the same time can cause the system to run out of memory. This necessitates the need to limit the number of threads being created.

What is ThreadPool in Java?

A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing. Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive.

- Java provides the Executor framework which is centered around the Executor interface, its sub-interface –**ExecutorService** and the class-**ThreadPoolExecutor**, which implements both of these interfaces. By using the executor, one only has to implement the Runnable objects and send them to the executor to execute.
- They allow you to take advantage of threading, but focus on the tasks that you want the thread to perform, instead of thread mechanics.
- To use thread pools, we first create a object of ExecutorService and pass a set of tasks to it. ThreadPoolExecutor class allows to set the core and maximum pool size.The runnables that are run by a particular thread are executed sequentially.



Thread Pool Initialization with size = 3 threads. Task Queue = 5 Runnable Objects

Executor Thread Pool Methods

Method	Description
newFixedThreadPool(int)	Creates a fixed size thread pool.
newCachedThreadPool()	Creates a thread pool that creates new threads as needed, but will reuse previou constructed threads when they are availal
newSingleThreadExecutor()	Creates a single thread.

In case of a fixed thread pool, if all threads are being currently run by the executor then the pending tasks are placed in a queue and are executed when a thread becomes idle.

Thread Pool Example

In the following tutorial, we will look at a basic example of thread pool executor- FixedThreadPool.

Steps to be followed

1. Create a task(Runnable Object) to execute
2. Create Executor Pool using Executors
3. Pass tasks to Executor Pool
4. Shutdown the Executor Pool

```
// Java program to illustrate
// ThreadPool
```

```
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

// Task class to be executed (Step 1)
class Task implements Runnable
{
    private String name;

    public Task(String s)
    {
        name = s;
    }

    // Prints task name and sleeps for 1s
    // This Whole process is repeated 5 times
    public void run()
    {
        try
        {
            for (int i = 0; i<=5; i++)
            {
                if (i==0)
                {
                    Date d = new Date();
                    SimpleDateFormat ft = new SimpleDateFormat("hh:mm:ss");
                    System.out.println("Initialization Time for"
                            + " task name - "+ name +" = " +ft.format(d));
                    //prints the initialization time for every task
                }
                else
                {
                    Date d = new Date();
                    SimpleDateFormat ft = new SimpleDateFormat("hh:mm:ss");
                    System.out.println("Executing Time for task name - "+
                            name +" = " +ft.format(d));
                    // prints the execution time for every task
                }
                Thread.sleep(1000);
            }
            System.out.println(name+" complete");
        }

        catch(InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
public class Test
{
     // Maximum number of threads in thread pool
    static final int MAX_T = 3;

    public static void main(String[] args)
```

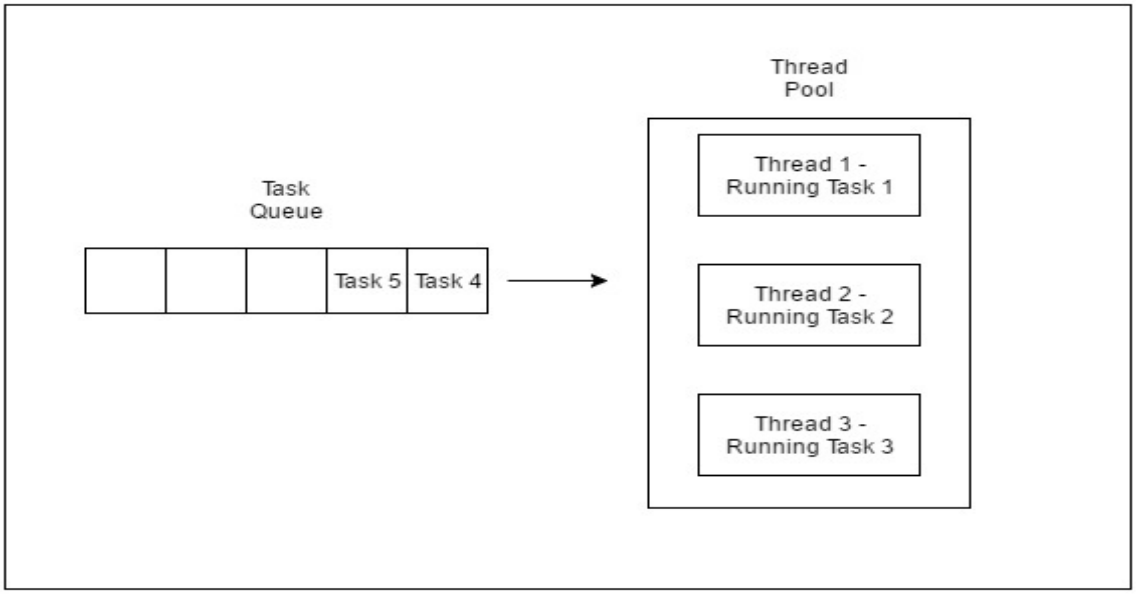
```
        {
            // creates five tasks
            Runnable r1 = new Task("task 1");
            Runnable r2 = new Task("task 2");
            Runnable r3 = new Task("task 3");
            Runnable r4 = new Task("task 4");
            Runnable r5 = new Task("task 5");

            // creates a thread pool with MAX_T no. of
            // threads as the fixed pool size(Step 2)
            ExecutorService pool = Executors.newFixedThreadPool(MAX_T);

            // passes the Task objects to the pool to execute (Step 3)
            pool.execute(r1);
            pool.execute(r2);
            pool.execute(r3);
            pool.execute(r4);
            pool.execute(r5);

            // pool shutdown ( Step 4)
            pool.shutdown();
        }
    }
```

Sample Execution

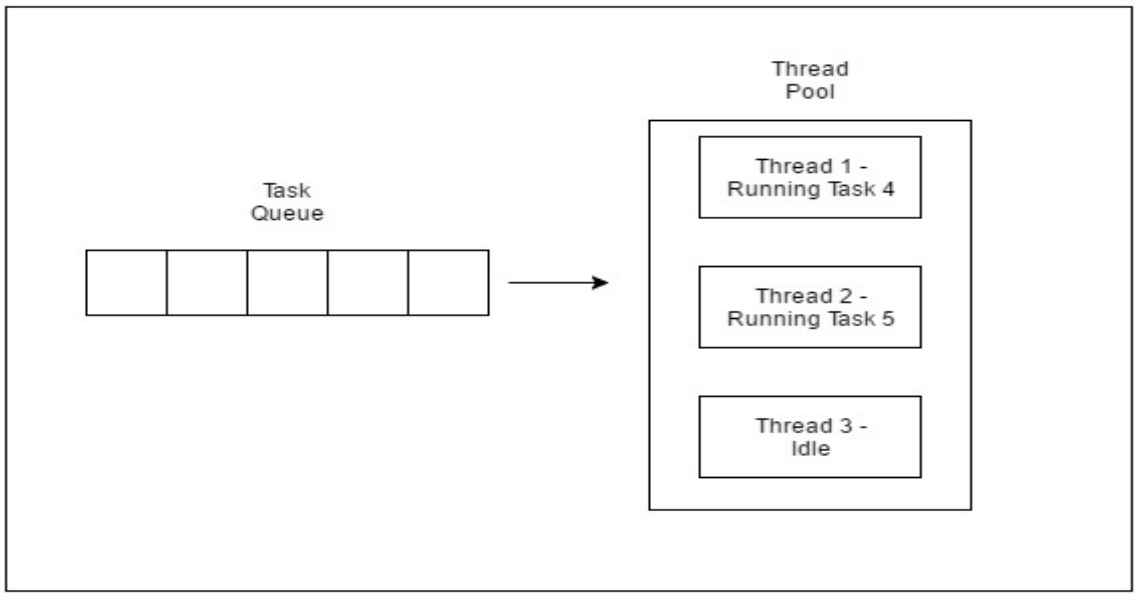
```
Output:
Initialization Time for task name - task 2 = 02:32:56
Initialization Time for task name - task 1 = 02:32:56
Initialization Time for task name - task 3 = 02:32:56
Executing Time for task name - task 1 = 02:32:57
Executing Time for task name - task 2 = 02:32:57
Executing Time for task name - task 3 = 02:32:57
Executing Time for task name - task 1 = 02:32:58
Executing Time for task name - task 2 = 02:32:58
Executing Time for task name - task 3 = 02:32:58
Executing Time for task name - task 1 = 02:32:59
Executing Time for task name - task 2 = 02:32:59
Executing Time for task name - task 3 = 02:32:59
Executing Time for task name - task 1 = 02:33:00
Executing Time for task name - task 3 = 02:33:00
Executing Time for task name - task 2 = 02:33:00
Executing Time for task name - task 2 = 02:33:01
Executing Time for task name - task 1 = 02:33:01
Executing Time for task name - task 3 = 02:33:01
```

```
task 2 complete
task 1 complete
task 3 complete
Initialization Time for task name - task 5 = 02:33:02
Initialization Time for task name - task 4 = 02:33:02
Executing Time for task name - task 4 = 02:33:03
Executing Time for task name - task 5 = 02:33:03
Executing Time for task name - task 5 = 02:33:04
Executing Time for task name - task 4 = 02:33:04
Executing Time for task name - task 4 = 02:33:05
Executing Time for task name - task 5 = 02:33:05
Executing Time for task name - task 5 = 02:33:06
Executing Time for task name - task 4 = 02:33:06
Executing Time for task name - task 5 = 02:33:07
Executing Time for task name - task 4 = 02:33:07
task 5 complete
task 4 complete
```

As seen in the execution of the program, the task 4 or task 5 are executed only when a thread in the pool becomes idle. Until then, the extra tasks are placed in a queue.



Thread Pool executing first three tasks



Thread Pool executing task 4 and 5

```
One of the main advantages of using this approach is when you want to proce
ThreadPool will create maximum of 10 threads to process 10 requests at a t
ThreadPool will internally allocate the 11th request to this Thread
and will keep on doing the same to all the remaining requests.
```

Risks in using Thread Pools

1. Deadlock **:** While deadlock can occur in any multi-threaded program, thread pools introduce another case of deadlock, one in which all the executing threads are waiting for the results from the blocked threads waiting in the queue due to the unavailability of threads for execution.
2. **Thread Leakage :**Thread Leakage occurs if a thread is removed from the pool to execute a task but not returned to it when the task completed. As an example, if the thread throws an exception and pool class does not catch this exception, then the thread will simply exit, reducing the size of the thread pool by one. If this repeats many times, then the pool would eventually become empty and no threads would be available to execute other requests.
3. **Resource Thrashing :**If the thread pool size is very large then time is wasted in context switching between threads. Having more threads than the optimal number may cause starvation problem leading to resource thrashing as explained.

Important Points

1. Don’t queue tasks that concurrently wait for results from other tasks. This can lead to a situation of deadlock as described above.
2. Be careful while using threads for a long lived operation. It might result in the thread waiting forever and would eventually lead to resource leakage.

3. The Thread Pool has to be ended explicitly at the end. If this is not done, then the program goes on executing and never ends. Call shutdown() on the pool to end the executor. If you try to send another task to the executor after shutdown, it will throw a RejectedExecutionException.
4. One needs to understand the tasks to effectively tune the thread pool. If the tasks are very contrasting then it makes sense to use different thread pools for different types of tasks so as to tune them properly.
5. You can restrict maximum number of threads that can run in JVM, reducing chances of JVM running out of memory.
6. If you need to implement your loop to create new threads for processing, using ThreadPool will help to process faster, as ThreadPool does not create new Threads after it reached it's max limit.
7. After completion of Thread Processing, ThreadPool can use the same Thread to do another process(so saving the time and resources to create another Thread.)

Tuning Thread Pool

- The optimum size of the thread pool depends on the number of processors available and the nature of the tasks. On a N processor system for a queue of only computation type processes, a maximum thread pool size of N or N+1 will achieve the maximum efficiency.But tasks may wait for I/O and in such a case we take into account the ratio of waiting time(W) and service time(S) for a request; resulting in a maximum pool size of N* (1+ W/S) for maximum efficiency.

The thread pool is a useful tool for organizing server applications. It is quite straightforward in concept, but there are several issues to watch for when implementing and using one, such as deadlock, resource thrashing. Use of executor service makes it easier to implement.

This article is contributed by **Abhishek**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.