Java Concurrency in Practice

Chapter-8:
Applying Thread Pools

Upcode Software
Engineer Team

-2023-

CONTENT

1. What is thread pool ?

- A thread pool is a collection of threads that can be used to execute tasks concurrently.
- When a new task needs to be executed, the thread pool assigns an available thread to execute it.
- Once the task is completed, the thread is returned to the pool and made available for future tasks.

2. Why Do Developers Use Thread Pools ?

- creating new threads has overhead associated with it, which can lead to increased memory usage and slower execution when dealing with large amounts of data
- each time you switch between threads (also known as *context switching*), there is a performance penalty due to context switches, CPU cache flushes, and several other things going on behind the scenes, such as loading different stacks into memory, and so forth.
- If context switches are frequent and/or these caches get flushed too frequently, then performance will suffer significantly because there will be more time spent waiting in those caches than actually doing useful work on them
- Thread pools can help to improve the performance of your applications by reusing threads and avoiding the overhead of creating new threads each time a task is executed.

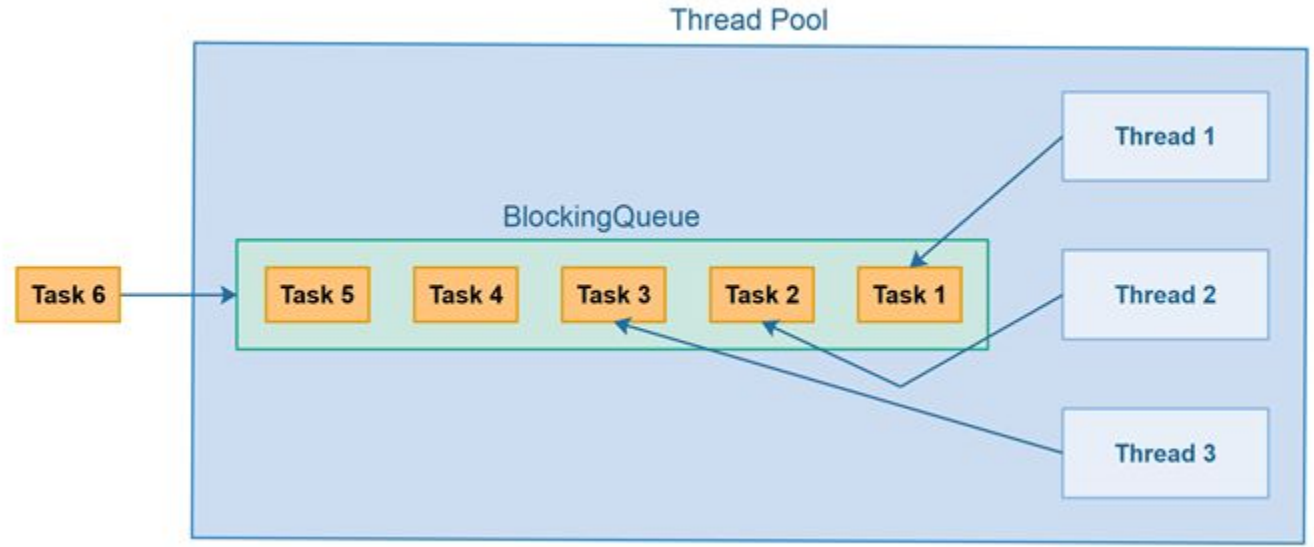
2. Why Do Developers Use Thread Pools ? 2/1

- Additionally, using a thread pool can make it easier to control how many threads are active at a time.
- Each thread consumes a certain amount of computer resources, such as memory (RAM), so if you have too many threads active at the same time, the total amount of resources (e.g. RAM) that is consumed may cause the computer to slow down - e.g.
- if so much RAM is consumed that the operating system (OS) starts swapping RAM out to disk.

3. How a Thread Pool Works ? 3/1

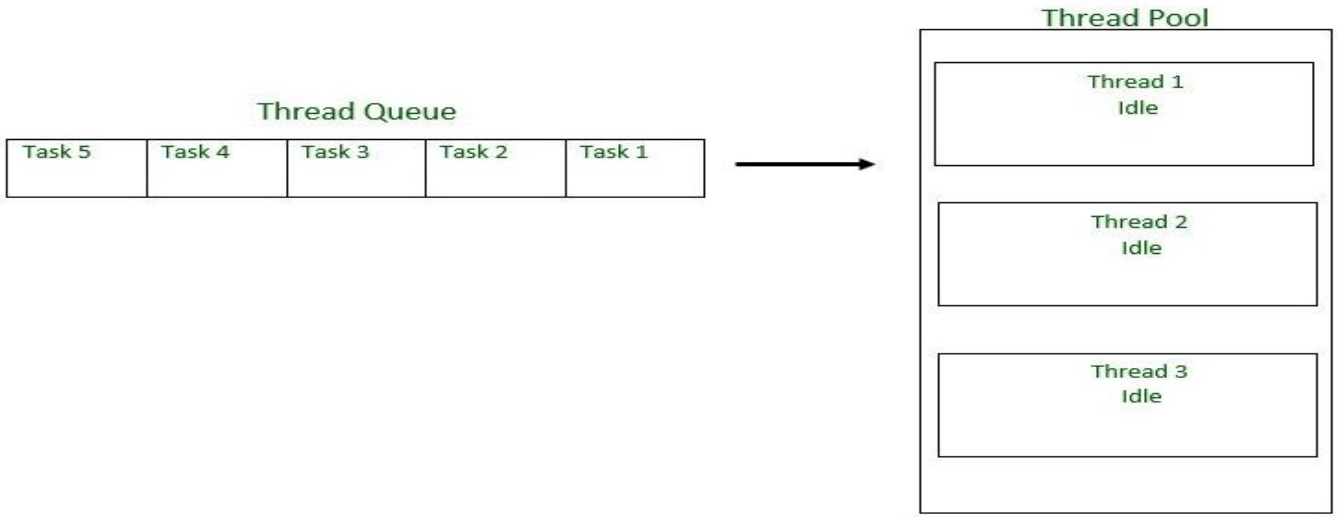
- Instead of starting a new thread for every task to execute concurrently, the task can be passed to a thread pool.
- As soon as the pool has any idle threads the task is assigned to one of them and executed.
- Internally the tasks are inserted into a Blocking Queue which the threads in the pool are dequeuing from.
- When a new task is inserted into the queue one of the idle threads will dequeue it successfully and execute it.
- The rest of the idle threads in the pool will be blocked waiting to dequeue tasks.

3. How a Thread Pool Works ? 3/2



3. How a Thread Pool Works ? 3/3

- 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.



4. How to Create a Thread Pool in Java ?

- The **java.util.concurrent** package provides several classes that can be used for this purpose, including the **Executors** class and the **ThreadPoolExecutor** class.
- The ***Executor*** interface has a single *execute* method to submit ***Runnable*** instances for execution.

```
Executor executor = Executors.newSingleThreadExecutor();
executor.execute(() -> System.out.println("Hello World"));
```

4. How to Create a Thread Pool in Java ? 4/2

- The ***ExecutorService*** interface contains a large number of methods to control the progress of the tasks and manage the termination of the service.

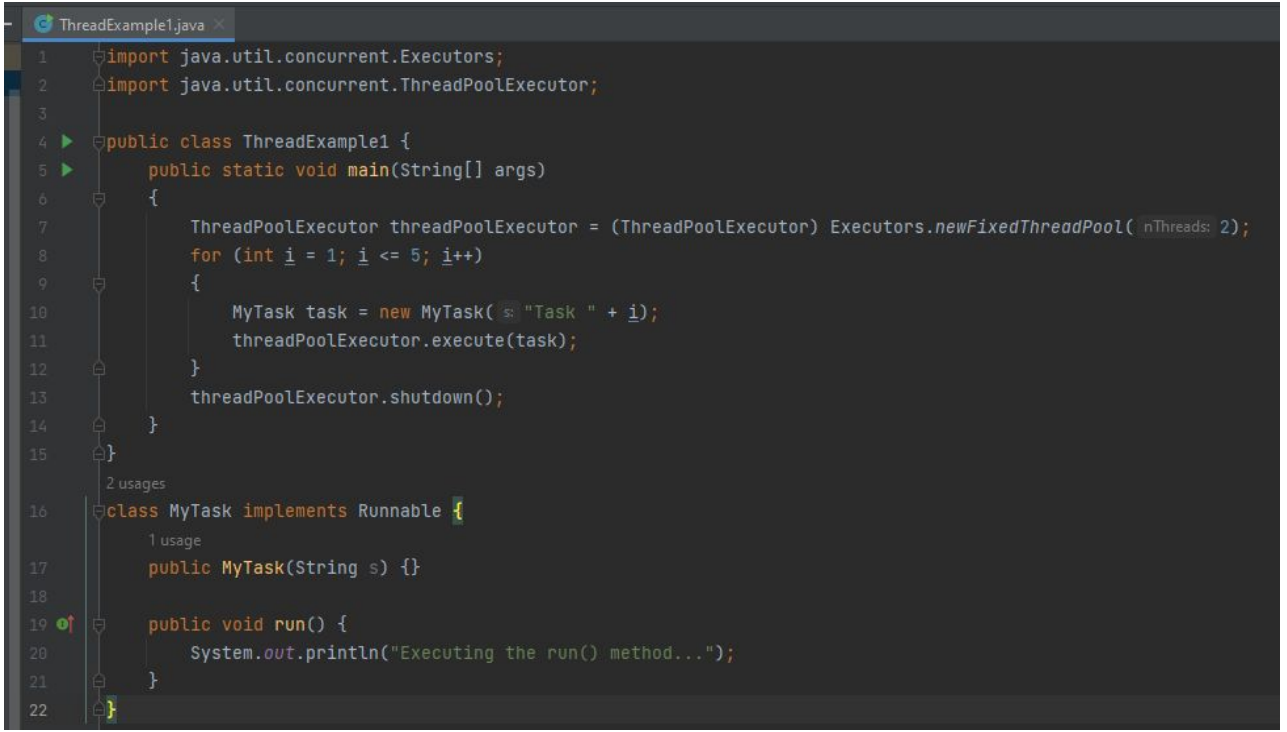
```
ExecutorService executorService = Executors.newFixedThreadPool(10);
Future<String> future = executorService.submit(() -> "Hello World");
String result = future.get();
```

- Using the **Executor** class is the simplest way to create a thread pool, but the **ThreadPoolExecutor** class provides more flexibility and control.

4. How to Create a Thread Pool in Java ? 4/3

```
import java.util.concurrent.*;
public class MyThreadPool {
public static void main(String[] args) {
// Create a fixed-size thread pool with three threads
ExecutorService executorService = Executors.newFixedThreadPool(3);
// Submit a task to the executor
executorService.submit(new Runnable() {
public void run() {
// Write your custom code here...
System.out.println("Inside run method...");
}
});
executorService.shutdown(); // Shut down the executor service
}
}
```

4. How to Create a Thread Pool in Java ? 4/4



5. Sizing Thread Pool 5/1

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

5. Sizing Thread pool 5/2

- Each new task will create a new thread if all existing threads are busy.
- In the case of high load, at best we will get a thread "starvation" situation, at worst OutOfMemoryError.
- It is better to maintain control and not allow clients to "DDoS/throttle" our service.

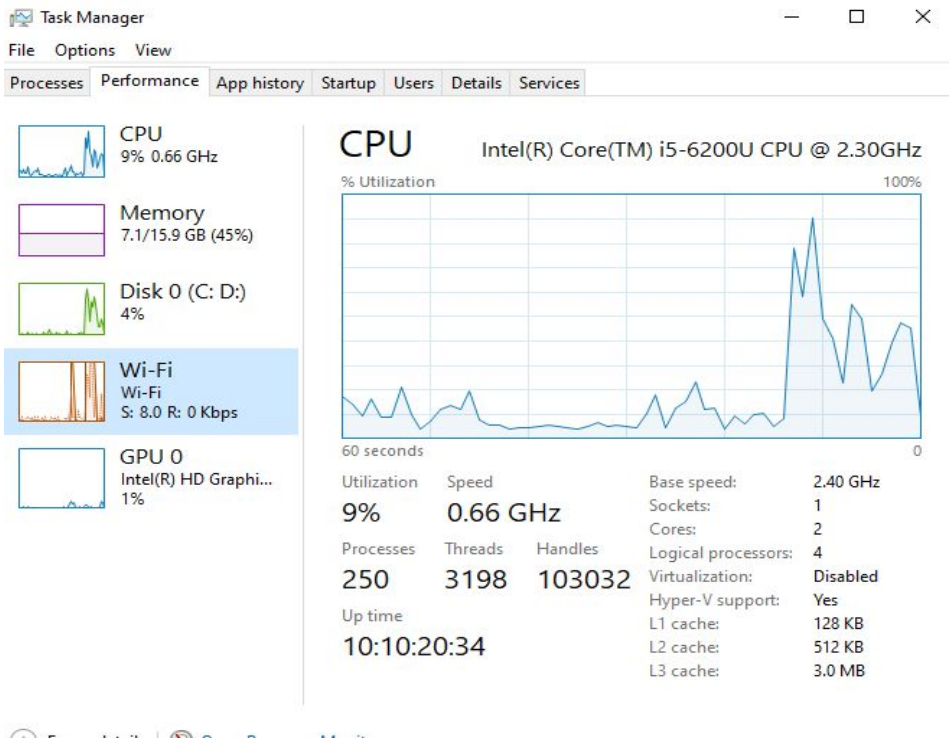
```
/** Thread Pool constructor */
public ThreadPoolExecutor(int corePoolSize,
              int maximumPoolSize,
              long keepAliveTime,
              TimeUnit unit,
              BlockingQueue workQueue) {...}

/** Cached Thread Pool */
public static ExecutorService newCachedThreadPool() {
              return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                            60L, TimeUnit.SECONDS,
                                            new SynchronousQueue());
}
```

5. Sizing Thread pool 5/3

CPU Cores – It is a physical CPU capable of doing independent processing. In case of Hyper Threading, a core can do two or more tasks simultaneously. Its leads to logical processing.

Logical Processors – It is the processor seen by the operating system. But actually it does not exist physically.

Logical Processor = *num. of core * num. of thread in each core = 4 * 2 = 8* where **No. of Core** = 4 (ie. physical processor unit)



5. Sizing Thread pool 5/4

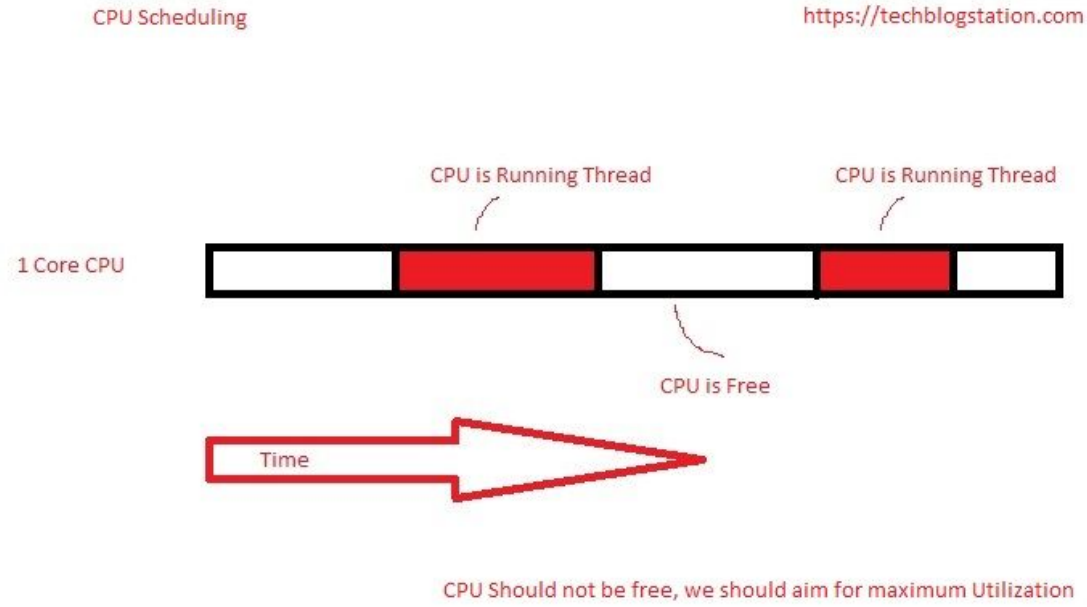
There are two type of tasks:

- CPU Bound
- IO Bound

CPU Bound Tasks which involve mathematical calculations and data marshaling etc. are known as CPU bound tasks.

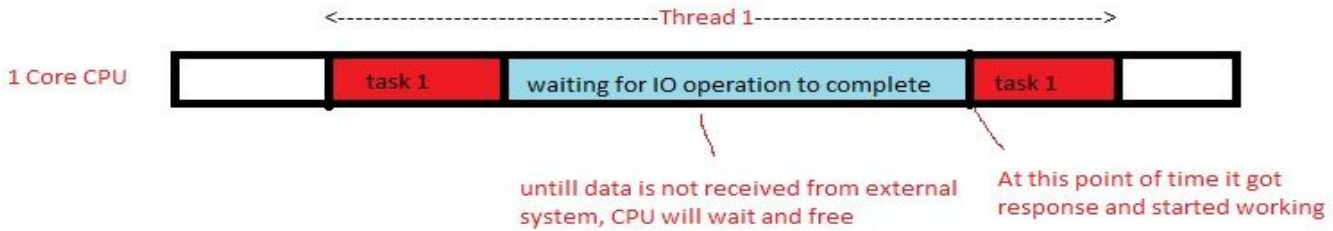
I/O Bound Tasks which involve communication to other application through network calls like database, web services, micro services, outside caches etc.

5. Sizing Thread pool 5/5

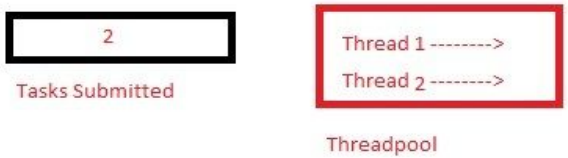


5. Sizing Thread pool

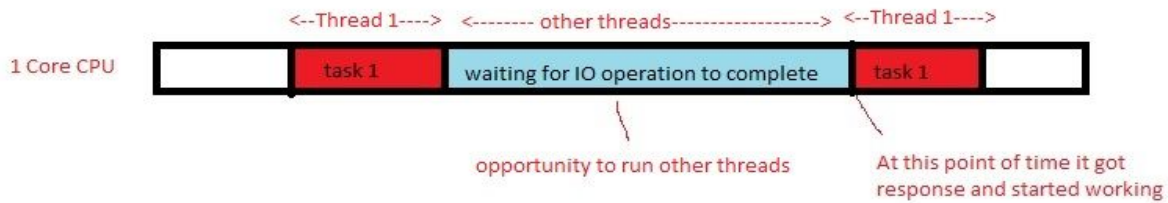
https://techblogstation.com



5. Sizing Thread pool



https://techblogstation.com



5. Sizing Thread pool 5/8

The formula to calculate the number of thread count for maximum utilization of CPU.

```
Number of threads = Number of Available Cores * (1 + Wait time / Service time)
```

- **Waiting time -** is the time spent waiting for IO bound tasks to complete, say waiting for HTTP response from remote service
- **Service time -** is the time spent being busy, say processing the HTTP response, marshaling/unmarshaling, any other transformations etc.

Wait time / Service time - this ratio is often called ***blocking coefficient*.**

5. Sizing Thread pool 5/9

We can get the total number of CPUs that we have as follows:

```
int numOfCores = Runtime.getRuntime().availableProcessors();
```

Brian Goetz in his famous book "Java Concurrency in Practice" recommends the following formula:

```
Number of threads = Number of Available Cores * (1 + Wait time / Service time)
```

For example:
*8 * (1 + 50 / 5) = 88*

6. Thread starvation deadlock

Deadlock can occur when two (or more) threads are blocking each other's progress — for example, *thread_1* is waiting for the release of *lock_2* held by *thread_2* upon which it will release its lock — *lock_1*, but the same is true for *thread_2* — it will release *lock_2* only upon the release of *lock_1* by *thread_1*.

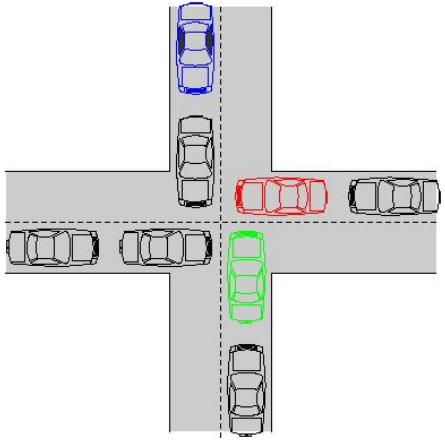


Figure - 1

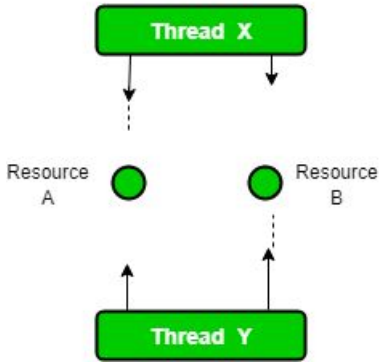
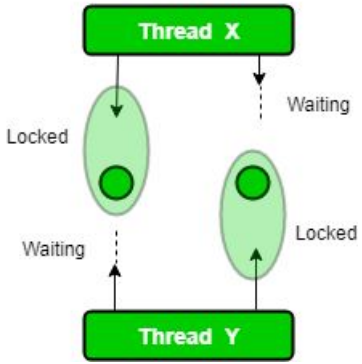


Figure - 2



7. Configuring ThreadPoolExecutor 7/1

- One of the added Advantage of using ThreadPoolTaskExecutor of Spring is that it is well suited for management and monitoring
- **The default configuration of core pool size is 1, max pool size and queue capacity as 2,147,483,647.**
- If you are using Java Annotation to define the bean, you can setup like below:

```
@Bean
public TaskExecutor threadPoolTaskExecutor() {
    ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
    executor.setCorePoolSize(5);
    executor.setMaxPoolSize(10);
    executor.initialize();
    return executor;
}
```

7. Configuring ThreadPoolExecutor 7/2

- **CorePoolSize** is the minimum number of threads that remain active at any given point of time. If you don't provide a value explicitly then it will have **default value as 1.**
- The TaskExecutor delegates the value to the underlying class ThreadPoolExecutor.
- **MaxPoolSize** is the maximum number of threads that can be created. The TaskExecutor delegates the value to the underlying ThreadPoolExecutor.
- The maxPoolSize relies on **queueCapacity because ThreadPoolTaskExecutor** creates a new thread only if the number of items in the queue exceeds queue capacity.

7. Configuring ThreadPoolExecutor 7/3

```
public void createTasks(ThreadPoolTaskExecutor taskExecutor, int numTasks) {
    for (int i=0; i {
            try {
                long sleepTime = ThreadLocalRandom.current().nextLong(1, 10) * 100;
                Thread.sleep(sleepTime);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
    }
}
```

7. Configuring ThreadPoolExecutor 7/4

```
public static void main(String[] args) {
    // Creating ThreadPoolTaskExecutor
    ThreadPoolTaskExecutor taskExecutor
         = new ThreadPoolTaskExecutor();
    taskExecutor.afterPropertiesSet();
    // Creating Tasks within ThreadPoolTaskExecutor
    createTasks(taskExecutor, 6);
    // Getting PoolSize
    System.out.println("Properties " +
        "- corePoolSize: " + taskExecutor.getCorePoolSize() +
        ", maxPoolSize: " + taskExecutor.getMaxPoolSize() +
        ", poolSize: " + taskExecutor.getPoolSize() +
        ", activeCount: " + taskExecutor.getActiveCount());
    // Shutting Down ThreadPoolTaskExecutor
    taskExecutor.setWaitForTasksToCompleteOnShutdown(true);
    taskExecutor.shutdown();
}
```

7. Configuring ThreadPoolExecutor 7/5

```
17:37:21.696 [main] INFO org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor - Initializing ExecutorService
Properties - corePoolSize: 1, maxPoolSize: 2147483647, poolSize: 1, activeCount: 1
17:37:21.747 [main] INFO org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor - Shutting down ExecutorService
```

```
ThreadPoolTaskExecutor taskExecutor = new ThreadPoolTaskExecutor();
taskExecutor.setCorePoolSize(1);
taskExecutor.setMaxPoolSize(4);
taskExecutor.setQueueCapacity(2);
taskExecutor.afterPropertiesSet();
```

```
17:47:29.986 [main] INFO org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor - Initializing ExecutorService
Properties - corePoolSize: 1, maxPoolSize: 4, poolSize: 4, activeCount: 4
17:47:30.036 [main] INFO org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor - Shutting down ExecutorService
```

7. Configuring ThreadPoolExecutor 7/6

- TaskExecutor creates a new thread only if the number of items in the queue exceeds queue capacity. As a result, you will observe that pool size increases.
- To conclude, it’s always a good practice to define the core pool size, max pool size and queue capacity for the TaskExecutor explicitly from our end instead of leaving it to use the default values.

7. Configuring ThreadPoolExecutor 7/7

- In Java, the implementation class of a thread pool is ThreadPoolExecutor, and the constructor is listed below:

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit timeUnit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

7. Configuring ThreadPoolExecutor 7/8

- **corePoolSize**
  In the constructor, corePoolSize specifies the number of kernel threads in a thread pool. The kernel threads are always alive by default, but when allowCoreThreadTimeout is set to true, the kernel threads are also recycled if they are timeout.
- **maximumPoolSize**
  In the constructor, maximumPoolSize specifies the maximum number of threads that a thread pool can hold.
- **keepAliveTime**
  in the constructor, keepAliveTime specifies the duration during which threads may remain idle before being terminated. If threads are idle for longer than that duration, the non-kernel threads will be recycled. If you set allowCoreThreadTimeout to true, the kernel threads will also be recycled in this case.

7. Configuring ThreadPoolExecutor 7/9

- **timeUnit**
  In the constructor, timeUnit specifies the time unit of the duration during which threads may remain idle. Common timeUnits are TimeUnit.MILLISECONDS, TimeUnit.SECONDS, and TimeUnit.MINUTES.
- **blockingQueue**
  In the constructor, blockingQueue represents the task queue. Common implementation classes of thread pool task queues are listed below:
  - ArrayBlockingQueue
  - LinkedBlockingQueue
  - PriorityBlockingQueue
  - DelayQueue
  - SynchronousQueue
  - LinkedTransferQueue
  - LinkedBlockingDeque

8.Managing queued tasks 8/1

- **ArrayBlockingQueue**: A bounded blocking queue backed by an array. This queue orders elements first-in-first-out (FIFO) and supports fair queue accesses for threads.
- **LinkedBlockingQueue**: An optionally-bounded blocking queue based on linked nodes. The capacity, if unspecified, is equal to Integer.MAX_VALUE. This queue orders elements FIFO.
- **PriorityBlockingQueue**: An unbounded blocking queue. It sorts elements in a natural order by default. It can also specify a Comparator.
- **DelayQueue**: An unbounded blocking queue of Delayed elements in which an element can only be taken when its delay has expired. It is often used in cache system design and scheduled task scheduling.

8.Managing queued tasks 8/2

- **SynchronousQueue**: A blocking queue that does not store elements. In SynchronousQueue, each insert operation must wait for a corresponding remove operation by another thread and vice versa.
- **LinkedTransferQueue**: An unbounded TransferQueue based on linked nodes. Compared with LinkedBlockingQueue, it has transfer() and tryTranfer() methods, which immediately transfer the element to the consumer waiting to receive it.
- **LinkedBlockingDeque**: An optionally-bounded blocking deque based on linked nodes. It allows elements to be inserted and removed from both the head and the tail of the queue.

9.Saturation policies 9/1

- When all threads are busy, and the internal queue fills up, the executor becomes saturated.
- Executors can perform predefined actions once they hit saturation. These actions are known as Saturation Policies.
- We can modify the saturation policy of an executor by passing an instance of *RejectedExecutionHandler* to its constructor.
  - Abort Policy
  - Caller-Runs Policy
  - Discard Policy
  - Discard-Oldest Policy
  - Custom Policy
  - Shutdown

9.Saturation policies 9/2

The default policy is the abort policy. Abort policy causes the executor to throw a ***RejectedExecutionException***:

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(1, 1, 0, MILLISECONDS,
  new SynchronousQueue<>(),
  new ThreadPoolExecutor.AbortPolicy());

executor.execute(() -> waitFor(250));

assertThatThrownBy(() -> executor.execute(() -> System.out.println("Will be rejected")))
  .isInstanceOf(RejectedExecutionException.class);
```

Since the first task takes a long time to execute, the executor rejects the second task.

9.Saturation policies 9/3

Instead of running a task asynchronously in another thread, **this policy makes the caller thread execute the task**:

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(1, 1, 0, MILLISECONDS,
  new SynchronousQueue<>(),
  new ThreadPoolExecutor.CallerRunsPolicy());

executor.execute(() -> waitFor(250));

long startTime = System.currentTimeMillis();
executor.execute(() -> waitFor(500));
long blockedDuration = System.currentTimeMillis() - startTime;

assertThat(blockedDuration).isGreaterThanOrEqualTo(500);
```

After submitting the first task, the executor can't accept any more new tasks. Therefore, the caller thread blocks until the second task returns. The **caller-runs policy** makes it easy to implement a simple form of throttling. That is, a slow consumer can slow down a fast producer to control the task submission flow.

9.Saturation policies 9/4

The **discard policy** silently discards the new task when it fails to submit it:

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(1, 1, 0, MILLISECONDS,
  new SynchronousQueue<>(),
  new ThreadPoolExecutor.DiscardPolicy());

executor.execute(() -> waitFor(100));

BlockingQueue<String> queue = new LinkedBlockingDeque<>();
executor.execute(() -> queue.offer("Discarded Result"));

assertThat(queue.poll(200, MILLISECONDS)).isNull();
```

Here, the second task publishes a simple message to a queue. Since it never gets a chance to execute, the queue remains empty, even though we're blocking on it for some time.

9.Saturation policies 9/5

The discard-oldest policy **first removes a task from the head of the queue, then re-submits the new task**:

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(1, 1, 0, MILLISECONDS,
  new ArrayBlockingQueue<>(2),
  new ThreadPoolExecutor.DiscardOldestPolicy());

executor.execute(() -> waitFor(100));

BlockingQueue<String> queue = new LinkedBlockingDeque<>();
executor.execute(() -> queue.offer("First"));
executor.execute(() -> queue.offer("Second"));
executor.execute(() -> queue.offer("Third"));
waitFor(150);

List<String> results = new ArrayList<>();
queue.drainTo(results);

assertThat(results).containsExactlyInAnyOrder("Second", "Third");
```

9.Saturation policies 9/6

This time, we're using a bounded queue that can hold just two tasks. Here's what happens when we submit these four tasks:

- The first tasks hogs the single thread for 100 milliseconds
- The executor queues the second and third tasks successfully
- When the fourth task arrives, the discard-oldest policy removes the oldest task to make room for this new one

The discard-oldest policy and priority queues don't play well together. Because the head of a priority queue has the highest priority, **we may simply lose the most important task**.

9.Saturation policies 9/7

It’s also possible to provide a custom saturation policy just by implementing the ***RejectedExecutionHandler*** interface:

```
class GrowPolicy implements RejectedExecutionHandler {

    private final Lock lock = new ReentrantLock();

    @Override
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
        lock.lock();
        try {
            executor.setMaximumPoolSize(executor.getMaximumPoolSize() + 1);
        } finally {
            lock.unlock();
        }

        executor.submit(r);
    }
}
```

9.Saturation policies 9/8

In this example, when the executor becomes saturated, we increment the max pool size by one and then re-submit the same task:

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(1, 1, 0, MILLISECONDS,
  new ArrayBlockingQueue<>(2),
  new GrowPolicy());

executor.execute(() -> waitFor(100));

BlockingQueue<String> queue = new LinkedBlockingDeque<>();
executor.execute(() -> queue.offer("First"));
executor.execute(() -> queue.offer("Second"));
executor.execute(() -> queue.offer("Third"));
waitFor(150);

List<String> results = new ArrayList<>();
queue.drainTo(results);

assertThat(results).contains("First", "Second", "Third");
```

9.Saturation policies 9/9

In addition to overloaded executors, **saturation policies also apply to all executors that have been shut down**:

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(1, 1, 0, MILLISECONDS, new LinkedBlockingQueue<>());
executor.shutdownNow();

assertThatThrownBy(() -> executor.execute(() -> {}))
  .isInstanceOf(RejectedExecutionException.class);
```

The same is true for all executors that are in the middle of a shutdown:

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(1, 1, 0, MILLISECONDS, new LinkedBlockingQueue<>());
executor.execute(() -> waitFor(100));
executor.shutdown();

assertThatThrownBy(() -> executor.execute(() -> {}))
  .isInstanceOf(RejectedExecutionException.class);
```

10.Thread Factory 10/1

- The ThreadFactory interface is part of Java's java.util.concurrent package.
- It's designed to create new threads, and it's typically used in conjunction with executor services and thread pools, where creating threads on-the-fly is a common requirement.
  ThreadFactory interface in action.

10.Thread Factory 10/2

```
class SimpleThreadFactory implements ThreadFactory {
   private final String namePrefix;

   public SimpleThreadFactory(String namePrefix) {
      this.namePrefix = namePrefix;
   }

   @Override
   public Thread newThread(Runnable r) {
      Thread t = new Thread(r);
      t.setName(namePrefix + "-" + t.getId());
      return t;
   }
}
```

10.Thread Factory 10/3

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.ThreadFactory;

public class Main {
    public static void main(String[] args) {
        SimpleThreadFactory factory = new SimpleThreadFactory("Test");
        ExecutorService executor = Executors.newFixedThreadPool(5, factory);

        for (int i = 0; i < 10; i++) {
            Runnable worker = new WorkerThread("" + i);
            executor.execute(worker);
        }

        executor.shutdown();
        while (!executor.isTerminated()) {
        }

        System.out.println("All threads finished");
    }
}
```

10.Thread Factory 10/4

Edit & Run

```
class WorkerThread implements Runnable {
   private String command;

   public WorkerThread(String s) {
       this.command = s;
   }

   @Override
   public void run() {
     System.out.println(Thread.currentThread().getName() + " Start. Command = " + command);
       processCommand();
     System.out.println(Thread.currentThread().getName() + " End.");
   }

   private void processCommand() {
     try {
         Thread.sleep(5000);
     } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}
```

Reference

1. Java-thread-pool
2. An Introduction to Thread Pools in Java
3. Thread Pools
4. How to Determine Java Thread Pool Size: A Comprehensive Guide
5. Java Thread Pool | Executor, ExecutorService, Example
6. Java - Thread Pools
7. Deadlock in Java Multithreading
8. Configuring ThreadPoolExecutor
9. What is Executor Framework and how it works internally
10. ThreadFactory Interface in Java with Examples

Thank you!

Presented by

Sanjar Suyunov

(sanjarsuyunov1304@gmail.com)