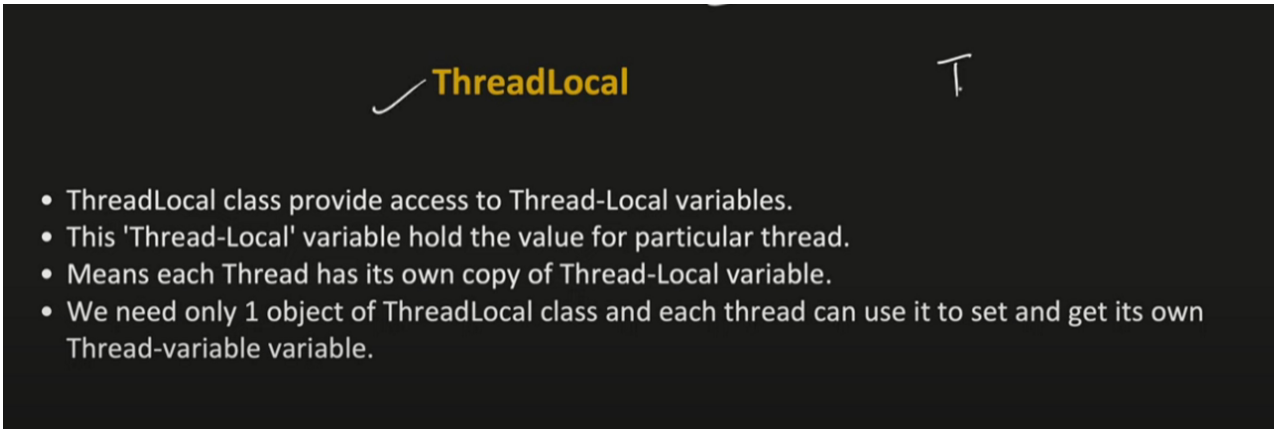
Lesson 10 : ThreadLocal VirtualThread

Sunday, May 26, 2024 11:30 AM

ThreadLocal in Java provides thread-local variables. Each thread accessing such a variable has its own, independently initialized copy of the variable. ThreadLocal is useful when you want to ensure that a variable is not shared between threads, which can avoid issues with concurrent access and modifications.



.set() already sets values to that particular Thread variables as inside the set() implmentation, it fetches the current thread.

Only one time you need to initiate ThreadLocal class object, and you can use the same to set for each particular value that you want.

REMEMBER TO CLEANUP IF RESUING THE THREAD.

As we have only one instance of ThreadLocal object here so after setting it for one thread, remember to cleanup for next thread. Otherwise we will encounter problem as below. The value is set for first thread and when that thread again gets chance to execute, the same old value is reflected which was set previously.

Therefore when your task gets completed, always do threadLocalObj.remove().

Code example of ThreadLocal

```
public class ThreadLocalExample {

  // Define a ThreadLocal variable to hold the user ID
  private static final ThreadLocal<Integer> threadLocalUserId = new ThreadLocal<Integer>() {
    @Override
    protected Integer initialValue() {
      return null; // Initial value is null
    }
  };

  // Method to set the user ID for the current thread
  public static void setUserId(Integer userId) {
    threadLocalUserId.set(userId);
  }

  // Method to get the user ID for the current thread
  public static Integer getUserId() {
    return threadLocalUserId.get();
  }

  // Method to clear the user ID for the current thread
  public static void clear() {
    threadLocalUserId.remove();
  }

  // Main method to demonstrate usage
  public static void main(String[] args) {
    Runnable task1 = () -> {
      setUserId(101);
      System.out.println("Task1 User ID: " + getUserId());
      clear();
    };

    Runnable task2 = () -> {
      setUserId(102);
      System.out.println("Task2 User ID: " + getUserId());
      clear();
    };

    Thread thread1 = new Thread(task1);
    Thread thread2 = new Thread(task2);

    thread1.start();
    thread2.start();
  }
```

}

VIRTUAL THREAD vs PLATOFRM (NORMAL) THREAD

Higher throughput means number of tasks completed per sec.
Latency is the delay between the submission of a task and its completion.

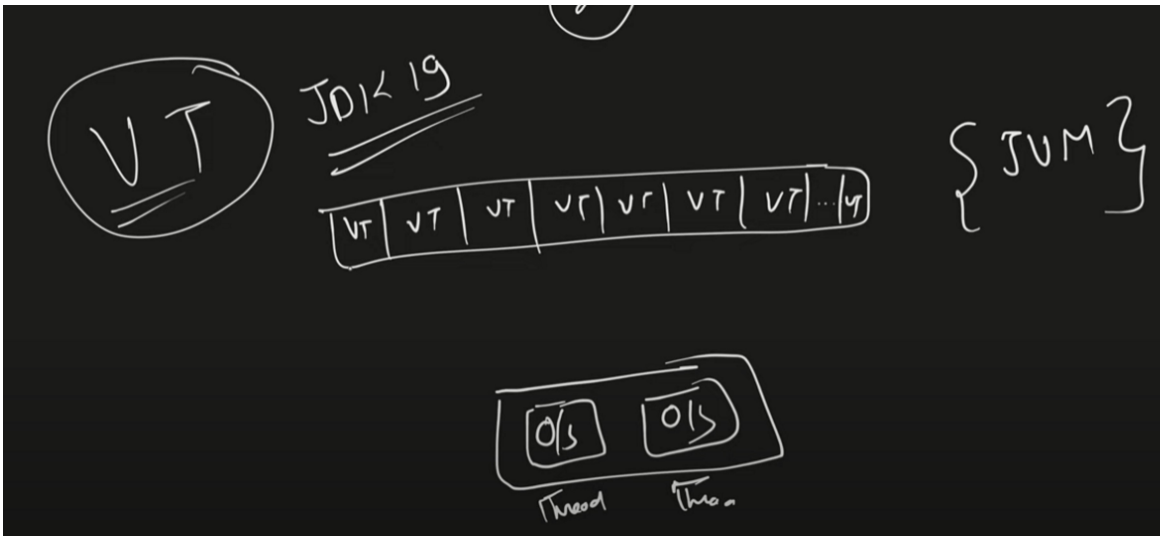
Normal Thread : If we are creating 10 normal threads then it means we are creating 10 OS threads.
So basically platform thread is just a **wrapper** provided by JVM. That is why we have all the properties of OS thread like stack, counter, register etc. So these threads are one to one attached with OS threads.

Disadvantages : It is slow. Thread creation takes time. JVM makes system calls and these system calls take time.
That is why we have use thread pool executor so it creates thread all at once and we do not have to create thread one by one which is time consuming. But here also we need to create new threads if min threshold exceeds so it does not completely solve our problem.

Another disadvantage is since normal threads are one to one mapped with OS threads, suppose there is IO (DB call) call taking 4 secs. So this thread will be in blocked state this blocking the OS thread as well

VIRTUAL THREADS

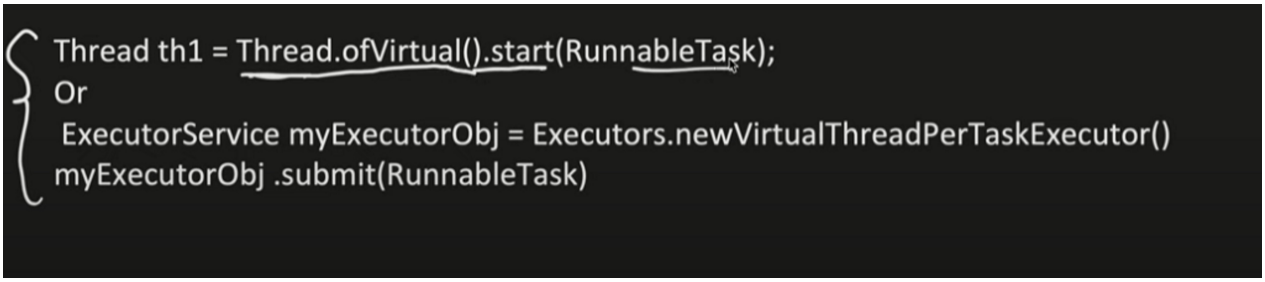
It has been made available since JDK 19.



As the above image indicates, even if there are only 2 OS threads, we can create n number of virtual threads and it is in full control of JVM.

So if any virtual thread is in blocked state because of IO, then JVM detaches that virtual thread from the attached OS thread and attaches the one that requires task to get executed. So there is no one to one mapping b/w OS thread and virtual thread. And all functionalities are backward compatible so it can be used just like normal threads.

Ways to create vitual thread



Traditional Threads

When you call Thread.sleep on a traditional thread:

Blocking the OS Thread: The thread is put into a sleep state for the specified duration. During this time, the underlying OS thread is effectively idle and not doing any useful work.

Resource Utilization: The OS thread continues to consume system resources (like memory) while it is sleeping. Since OS threads are heavyweight, creating and managing a large number of sleeping threads can lead to significant resource overhead.

Context Switching: When the sleep duration expires, the thread needs to be rescheduled by the operating system, which involves context switching. Context switching is relatively expensive in terms of CPU cycles and can affect the overall performance of the application.

Virtual Threads

When you call Thread.sleep on a virtual thread:

Non-blocking Sleep: The virtual thread is put into a sleep state, but unlike traditional threads, it doesn't block the underlying carrier thread (the OS thread on which it is currently running). Instead, the virtual thread is suspended, and the carrier thread is free to execute other virtual threads.

Efficient Resource Utilization: Since virtual threads are lightweight and managed by the Java runtime rather than the OS, they consume significantly fewer resources. Suspending a virtual thread doesn't involve the same resource overhead as suspending an OS thread.

Reduced Context Switching: The Java runtime manages virtual threads more efficiently, reducing the need for expensive context switches. When the sleep duration expires, the virtual thread is resumed without the overhead associated with rescheduling an OS thread.
Key Differences

Blocking vs. Non-blocking: Thread.sleep on a traditional thread blocks the OS thread, while on a virtual thread, it suspends the virtual thread without blocking the carrier thread.

Resource Efficiency: Virtual threads are much more resource-efficient, allowing a higher number of concurrent sleeping threads without significant overhead.

Context Switching: Virtual threads minimize the performance impact of context switching compared to traditional threads.