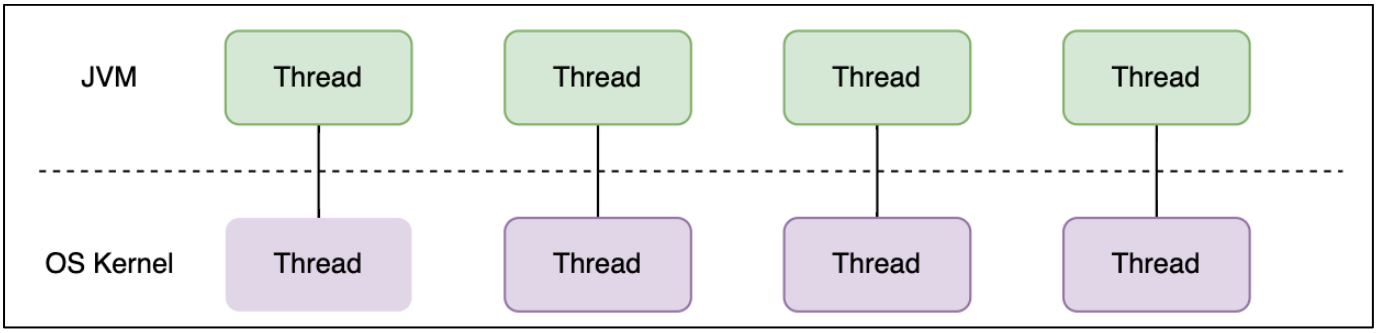
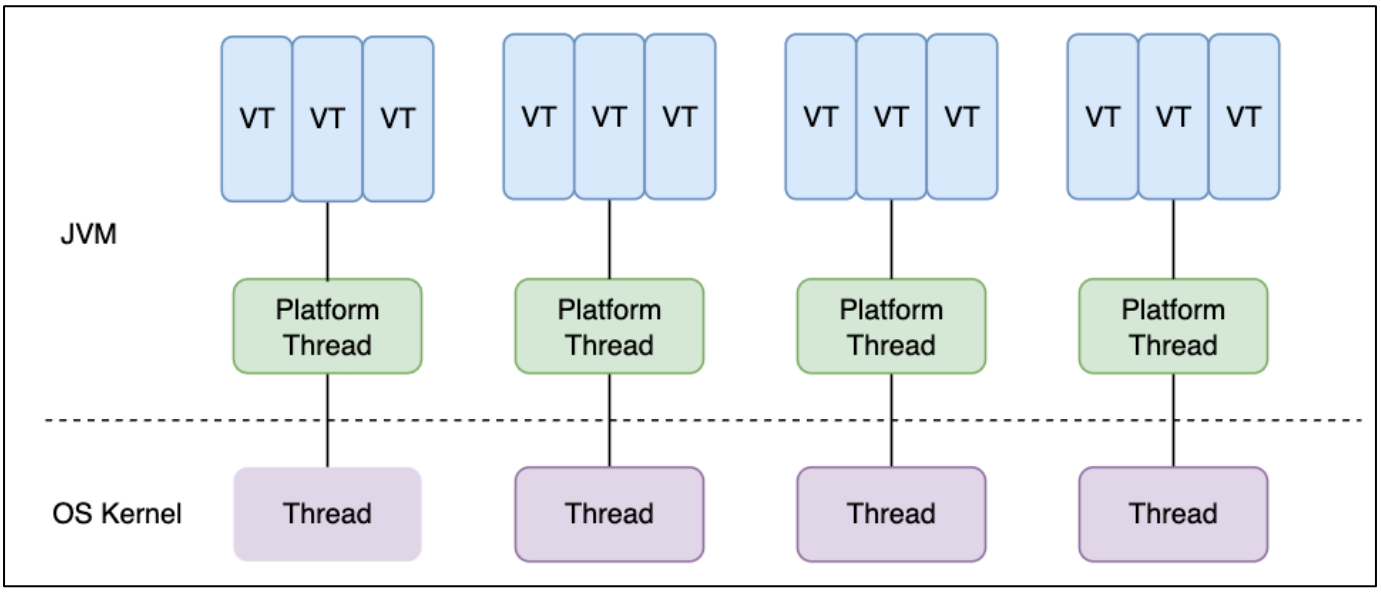
Java threads vs. virtual threads

Our common Java threads are one-to-one with the system kernel threads, and the system kernel thread scheduler is responsible for scheduling Java threads. In order to increase the performance of the application, we will add more and more Java threads, and obviously the system will take up a lot of resources to handle thread context switching when scheduling Java threads.



In recent decades, we have relied on the multithreaded model described above to solve the problems of concurrent programming in Java. To increase the throughput of the system, we have to keep increasing the number of threads, but the threads of the machine are expensive and the number of available threads is limited. Even though we use various thread pools to maximize the cost effectiveness of threads, threads often become bottlenecks in the performance of our applications before CPU, network, or memory resources are exhausted, not unlocking the maximum performance that the hardware should have.



To solve this problem Java19 introduces Virtual Thread. In Java19, the threads we used to use are called platform threads, which still correspond one-to-one with the system kernel threads. A large number (M) of virtual threads run on a smaller number (N) of platform threads (one-to-one correspondence with OS threads) (M:N scheduling). Multiple virtual threads are scheduled by the JVM to execute on a particular platform thread, and only one virtual thread is executed at a time on a platform thread.

Create Java virtual threads

New thread-related APIs

`Thread.ofVirtual()` and `Thread.ofPlatform()` are new APIs for creating virtual and platform threads.

```
12345678//output thread ID including virtual threads and system threads Thread.getId() deprecated from
        jdk19Runnable runnable = () -> System.out.println(Thread.currentThread().threadId());// Create
        virtual threadsThread thread =
        Thread.ofVirtual().name("testVT").unstarted(runnable);testVT.start();// Create virtual platform
        threadsThread testPT = Thread.ofPlatform().name("testPT").unstarted(runnable);testPT.start();
```

Use `Thread.startVirtualThread(Runnable)` to quickly create a virtual thread and start it.

```
123// Output thread IDs including virtual threads and system threadsRunnable runnable =
   () -> System.out.println(Thread.currentThread().threadId());Thread thread =
   Thread.startVirtualThread(runnable);
```

Determine if a thread is virtual with `Thread.isVirtual()`.

```
12Runnable runnable = () -> System.out.println(Thread.currentThread().isVirtual());Thread
  thread = Thread.startVirtualThread(runnable);
```

Use `Thread.join` to wait for the virtual thread to finish, use `Thread.sleep` to make the virtual thread sleep.

```
1234Runnable runnable = () -> System.out.println(Thread.sleep(10));Thread thread =
    Thread.startVirtualThread(runnable);// Wait for the virtual thread to
    finishthread.join();
```

Use `Executors.newVirtualThreadPerTaskExecutor()` to create an `ExecutorService` that creates a new virtual thread for each task.

```
123try (var executor = Executors.newVirtualThreadPerTaskExecutor())
   {  executor.submit(() -> System.out.println("hello"));}
```

Support interchange and migration with existing code using thread pools and ExecutorService.

> **Because virtual threads are a preview feature in Java19, the code that appears in this article needs to be run as follows**.

- Compile the program using `javac --release 19 --enable-preview Main.java` and run it using `java --enable-preview Main`.
- Or run the program using `java --source 19 --enable-preview Main.java`.

Performance of platform threads vs. virtual threads

Since we are trying to solve the problem of platform threads, we will directly test the performance of platform threads vs. virtual threads.

The test is simple: execute 10,000 tasks of one second of sleep in parallel and compare the total execution time and the number of system threads used.

To monitor the number of system threads used for the test, write the following code.

```
ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(1);scheduledExecutorService.scheduleAtFixedRate(() ->
{  ThreadMXBean threadBean = ManagementFactory.getThreadMXBean();  ThreadInfo[] threadInfo =
threadBean.dumpAllThreads(false, false);  System.out.println(threadInfo.length + " os
thread");}, 1, 1, TimeUnit.SECONDS);
```

The scheduling thread pool fetches and prints the number of system threads every second, making it easy to observe the number of threads.

```
public static void main(String[] args) {  //Record the number of system threads
ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(1);
scheduledExecutorService.scheduleAtFixedRate(() -> {    ThreadMXBean threadBean
= ManagementFactory.getThreadMXBean();    ThreadInfo[] threadInfo =
threadBean.dumpAllThreads(false, false);    System.out.println(threadInfo.length
+ " os thread");  }, 1, 1, TimeUnit.SECONDS);  long l =
System.currentTimeMillis();  try(var executor = Executors.newCachedThreadPool())
{    IntStream.range(0, 10000).forEach(i -> {      executor.submit(() ->
{        Thread.sleep(Duration.ofSeconds(1));        System.out.println(i);
return i;      });    });  }  System.out.printf("Time spent: %d ms",
System.currentTimeMillis() - l);}
```

First we use `Executors.newCachedThreadPool()` to execute 10000 tasks, because the maximum number of threads in `newCachedThreadPool` is Integer.MAX_VALUE, so theoretically at least a few thousand system threads will be created to execute.

The output is as follows (redundant output has been omitted)

```
//output171423914 os thread  Exception in thread "main" java.lang.OutOfMemoryError: unable to
create native thread: possibly out of memory or process/resource limits reached        at
java.base/java.lang.Thread.start0(Native Method)        at
java.base/java.lang.Thread.start(Thread.java:1560)        at
java.base/java.lang.System$2.start(System.java:2526)
```

As you can see from the above output, the maximum number of system threads created is 3914 and then an exception is thrown when the threads continue to be created and the program terminates. It is not realistic to try to improve the performance of the system by having a large number of system threads, because threads are expensive and resources are limited.

Now we use a thread pool with a fixed size of 200 to solve the problem of not being able to request too many system threads.

```
public static void main(String[] args) {  //Record the number of system
threads  ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(1);
scheduledExecutorService.scheduleAtFixedRate(() -> {    ThreadMXBean
threadBean = ManagementFactory.getThreadMXBean();    ThreadInfo[] threadInfo =
threadBean.dumpAllThreads(false, false);
System.out.println(threadInfo.length + " os thread");  }, 1, 1,
TimeUnit.SECONDS);  long l = System.currentTimeMillis();  try(var executor =
Executors.newFixedThreadPool(200)) {    IntStream.range(0, 10000).forEach(i ->
{      executor.submit(() -> {        Thread.sleep(Duration.ofSeconds(1));
System.out.println(i);        return i;      });    });  }
```

```
                System.out.printf("Time spent: %dms\n", System.currentTimeMillis() - l);}
```

The output is as follows.

```
123456//output199879998207 os threadTime spent: 50436ms
```

With the fixed size thread pool, there is no problem of creating a large number of system threads causing failure, and the task can be run normally, with a maximum of 207 system threads created, taking a total of 50436ms.

Let's take a look at the results of using virtual threads.

```
 1 2 3 4 5 6 7 8
9101112131415161718192021
```

```
public static void main(String[] args) {  ScheduledExecutorService
scheduledExecutorService = Executors.newScheduledThreadPool(1);
scheduledExecutorService.scheduleAtFixedRate(() -> {    ThreadMXBean threadBean
= ManagementFactory.getThreadMXBean();    ThreadInfo[] threadInfo =
threadBean.dumpAllThreads(false, false);    System.out.println(threadInfo.length
+ " os thread");  }, 10, 10, TimeUnit.MILLISECONDS);  long l =
System.currentTimeMillis();  try(var executor =
Executors.newVirtualThreadPerTaskExecutor()) {    IntStream.range(0,
10000).forEach(i -> {      executor.submit(() ->
{        Thread.sleep(Duration.ofSeconds(1));        System.out.println(i);
return i;      });    });  }  System.out.printf("Time spent: %dms\n",
System.currentTimeMillis() - l);}
```

The only difference between the code that uses virtual threads and the one that uses fixed size is the word `Executors.newFixedThreadPool(200)` replaced by `Executors.newVirtualThreadPerTaskExecutor()`.

The output is as follows.

```
12345//output1989015 os threadTime spent: 1582ms
```

As can be seen from the output, the total execution time is 1582 ms and the maximum number of system threads used is 15. The conclusion is clear that using virtual threads is much faster than platform threads and uses less resources from the system threads.

If we replace the task in this test program with one that performs a one-second computation (e.g., sorting a huge array), rather than just sleep for 1 second, there is no significant performance gain even if we increase the number of virtual or platform threads to much larger than the number of processor cores. Because virtual threads are not faster threads, they have no advantage over platform threads in terms of how fast they can run code. Virtual threads exist to provide higher throughput, not speed (lower latency).

The use of virtual threads can significantly increase program throughput if your application meets the following two characteristics.

- The program has a high number of concurrent tasks.
- IO-intensive, workload-independent CPU constraints.

Virtual threads can help increase the throughput of server-side applications because such applications have a large number of concurrent tasks, and these tasks usually have a large number of IO waits.