Luxoft
A DXC Technology Company

Project Loom - Scalable Concurrency with Virtual Java Threads

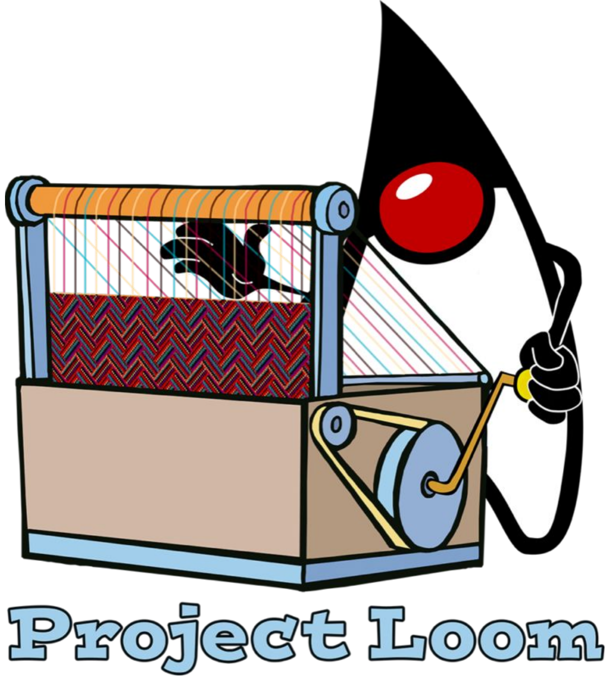
Rahman Usta
rahman.usta@dxc.com

/ustarahman

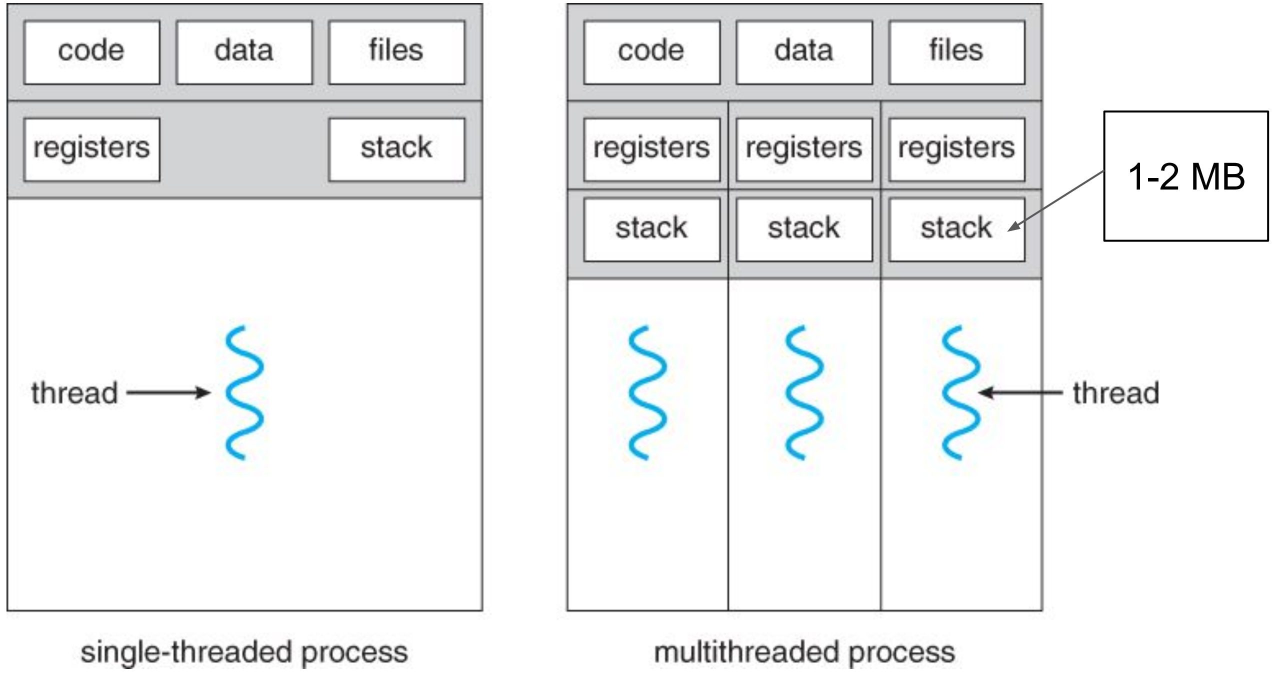
/rahmanusta

Project Loom - Agenda

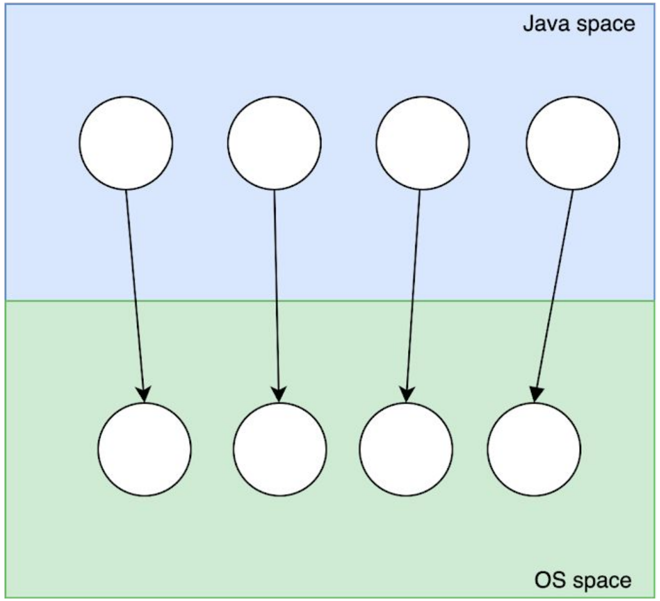
- What is Thread/Virtual Thread?
- MultiThreading Models
- How to create 1.000.000 Virtual threads
- Executor API changes
- Scalability and Structured Concurrency
- Async APIs vs Project Loom
- Demos



What is Thread ?



One-to-One Multithreading Model



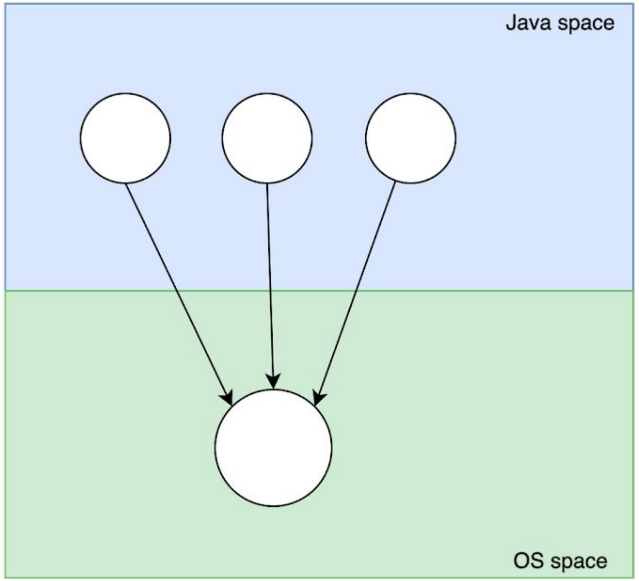
One Java thread is mapped to One Native thread

Issues with One-to-One Model

- Responsibilities on Developer
    - How many threads ?
        - Each thread has memory costs
    - How many thread pool ?
        - Size ? Type ?
- OS Threads are limited and precious resources
- Reactive programming?
        - To use or not to use ?
        - What library to use?
- Context switches are expensive

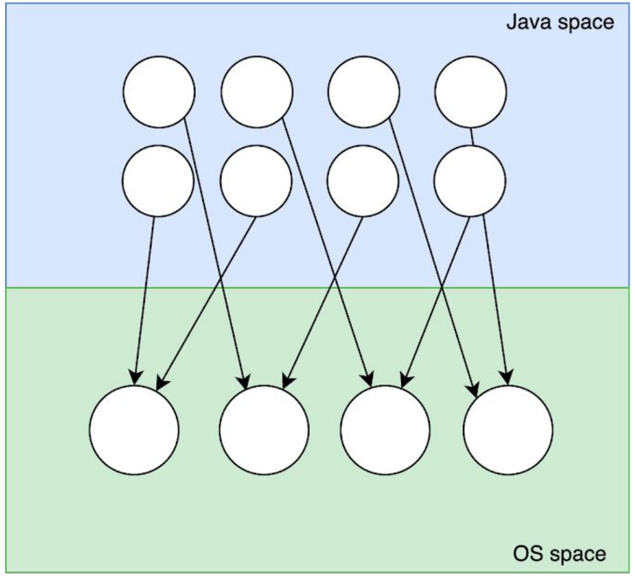
Can you create a thread without fear ?

Many-To-One Multithreading Model



Green threads are one of the example to the Many2One Model

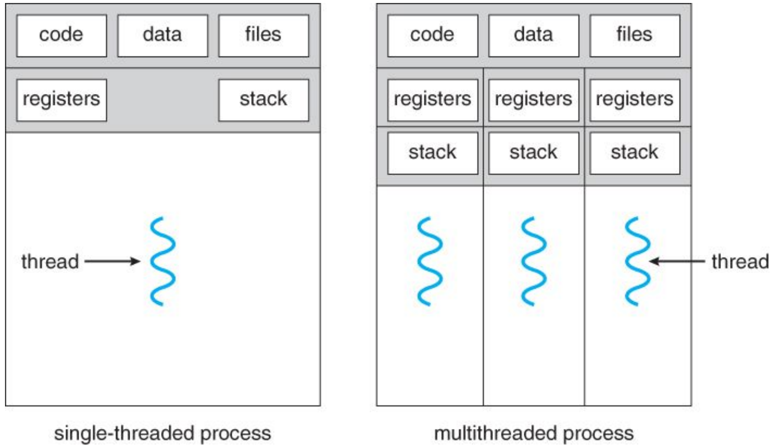
Many-to-Many Multithreading Model



Java on Solaris or Virtual Threads

OS Thread vs Virtual Thread

- OS Threads are expensive
  - 1-2 MB stack size
  - 1-10 microseconds for context switching
- Virtual Threads are cheap
  - Create as many you can
  - Virtual Threads are just Java objects
  - 200-300 B stack size for per vThread
  - Context switch is just around 200 nanoseconds



How to create Virtual threads

```
Thread virtualThread1 = Thread.startVirtualThread(() -> {

    System.out.println("Hello world");

});

Thread virtualThread2 = Thread.builder().virtual().task(() -> {

   System.out.println("Hello world");

}).build();

virtualThread2.start();
```

Demo

Create 1_000_000 threads and see how it works with Regular and Virtual Java threads.

Scalability and Synchronous programming

- Easy to understand
- Easy to debug
- Error handling is easy
- Not so scalable

```
requestService("Service 1");
String response = requestService("Service 2");
requestService(response + " > Service 2.1");
requestService("Service 3");
```

Scalability and Asynchronous programming

- Scalable
- Not so easy to understand
- Not easy to debug
- Error handling is hard

```
CompletableFuture
        .allOf(
                CompletableFuture.runAsync(() -> requestService("Service 1")),
                CompletableFuture.supplyAsync(() -> requestService("Service 2"))
                        .thenAcceptAsync(s ->  requestService(s + " > Service 
2.1")),
                CompletableFuture.runAsync(() -> requestService("Service 3"))
        ).join();
```

Scalability and Structured Concurrency

- Structured concurrency allows developers to write concurrent code in a visible code block.
- Code looks ***synchronous*** but runs as ***asynchronous***
- All tasks are done after leaving code block

```java
try (ExecutorService executor =
            Executors.newVirtualThreadExecutor()) {

   executor.submit(() -> System.out.println("Hello"));
   executor.submit(() -> System.out.println("World"));
}
```

Scalability and Thread Pools

Thread Pools were developed to scale Threads which are precious resources

- Scalable for **CPU** bound operations
  - Calculations/Operations on CPU
  - CPU bound operations doesn’t block
- NOT scalable for **I/O** bound operations
  - File, Socket, Thread.sleep etc.
  - I/O bound operations may block

Demo

Test and see how **Virtual Thread Pools** handle blocking operations efficiently.

Pinning issues

Carrier thread is pinned if synchronized used in virtual thread. It causes carrier thread not to be scheduled to another virtual thread.

It is not a bug but a point to improve

java.util.lock.* is safe to use

Debugging

Thank you!