Java Concurrency and Multithreading

TS. Nguyễn Văn Sơn

BM Công Nghệ Phần Mềm – FIT UET

Bài giảng được tài trợ bởi
Trung tâm Nghiên cứu và Phát triển Samsung Việt Nam (SRV)

Thông tin giảng viên

- Liên lạc: Phòng 321, E3, sonnguyen@vnu.edu.vn
- Nghiên cứu: AI for Software Engineering, Automated AI Engineering, Quality Assurance AI-enabled Systems
- Một số bài toán điển hình:
  - Tự động gợi ý/sinh mã nguồn
  - Phát hiện lỗ hổng bảo mật phần mềm
  - Tự động hóa quá trình phát triển giải pháp dựa trên AI
  - Đảm bảo chất lượng hệ thống xe tự hành
- Nhóm NC: Intelligent Software Engineering (iSE)

Tôi, năm 2022

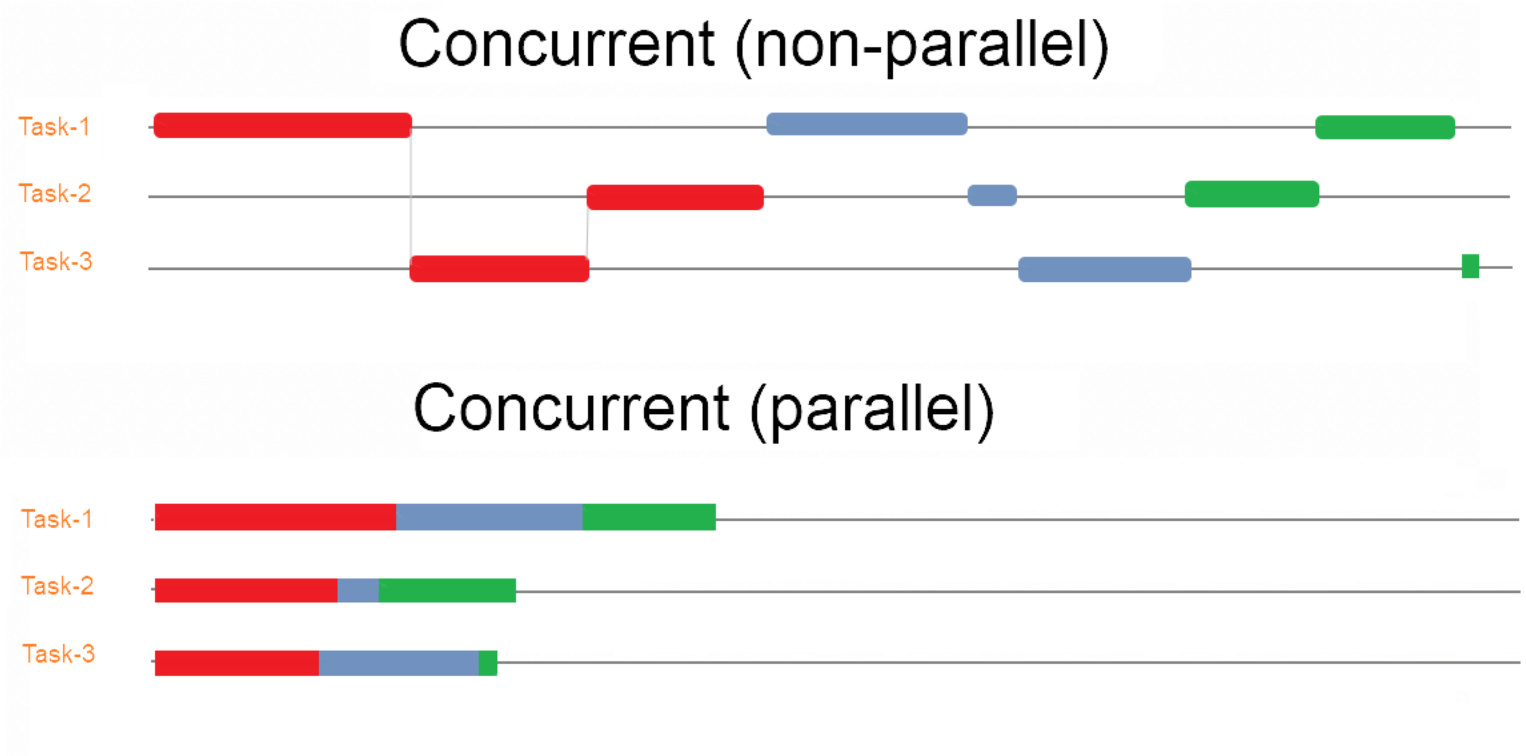
Contents

Core concepts

Concurrency

- **Concurrency**: run several programs or several parts of a program in parallel
- A task can be performed asynchronously or in parallel
  - → Improves the throughput and the interactivity of the program.

Concurrency vs. Parallelism

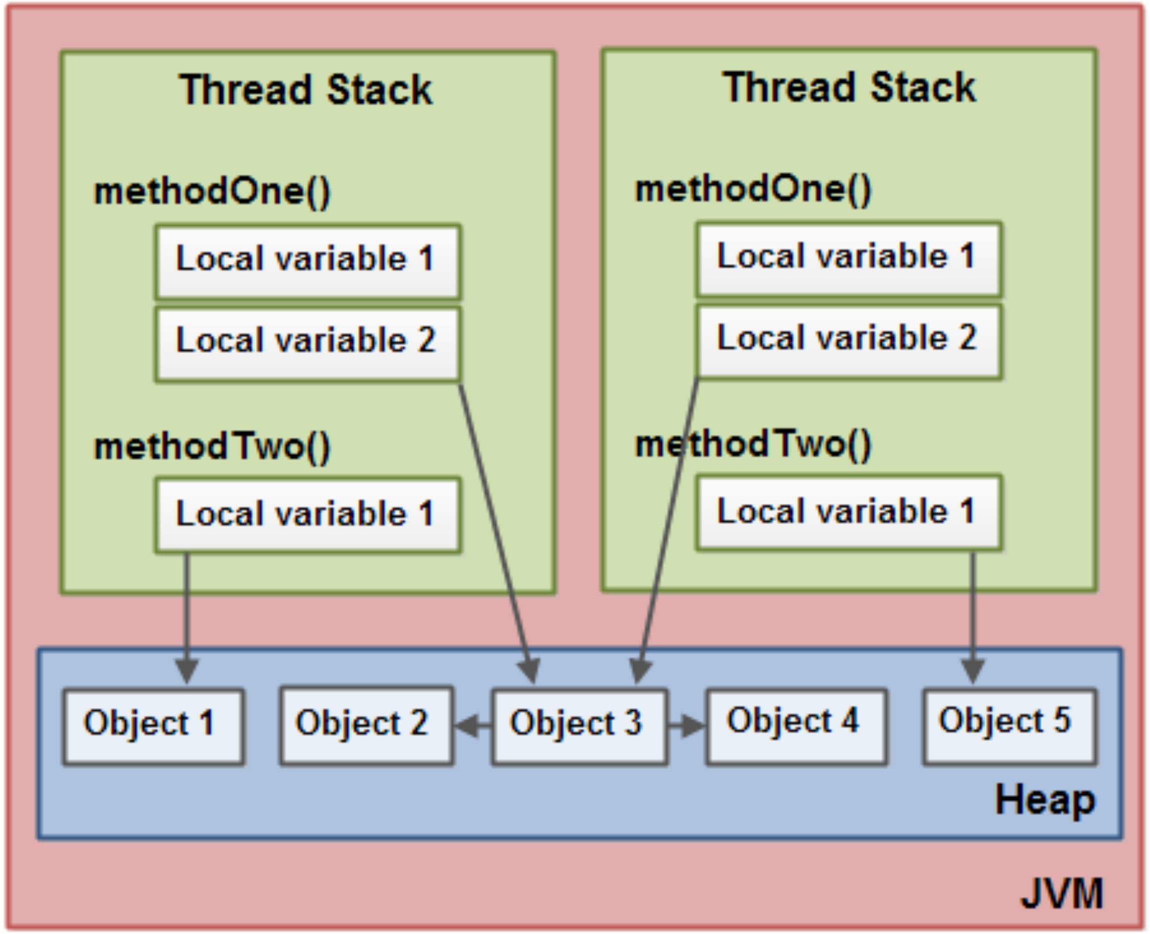


Unit of Concurrency

- **Multi-processing -** Multiple Processors/CPUs executing concurrently (Unit: CPU)
- **Multi-tasking -** Multiple tasks/processes running concurrently on a single CPU.
  - OS executes these tasks by switching between them very frequently (Unit: Process)
- **Multi-threading -** Multiple parts of the same program running concurrently.
  - Dividing the same program into multiple parts/threads and run those threads concurrently (Unit: Thread)

Processes and Threads

- A **process** is a program in execution running independently and isolated from others
- A **thread** is a path of execution within a process
  - It has its own call stack but can access shared data of other threads in the same process.
- A Java application runs by default in one process
- Within a Java application might have several threads to achieve parallel processing or asynchronous behavior


Thread Stack
methodOne()
Local variable 1
Local variable 2
methodTwo()
Local variable 1
Thread Stack
methodOne()
Local variable 1
Local variable 2
methodTwo()
Local variable 1
Object 1
Object 2
Object 3
Object 4
Object 5
Heap
JVM

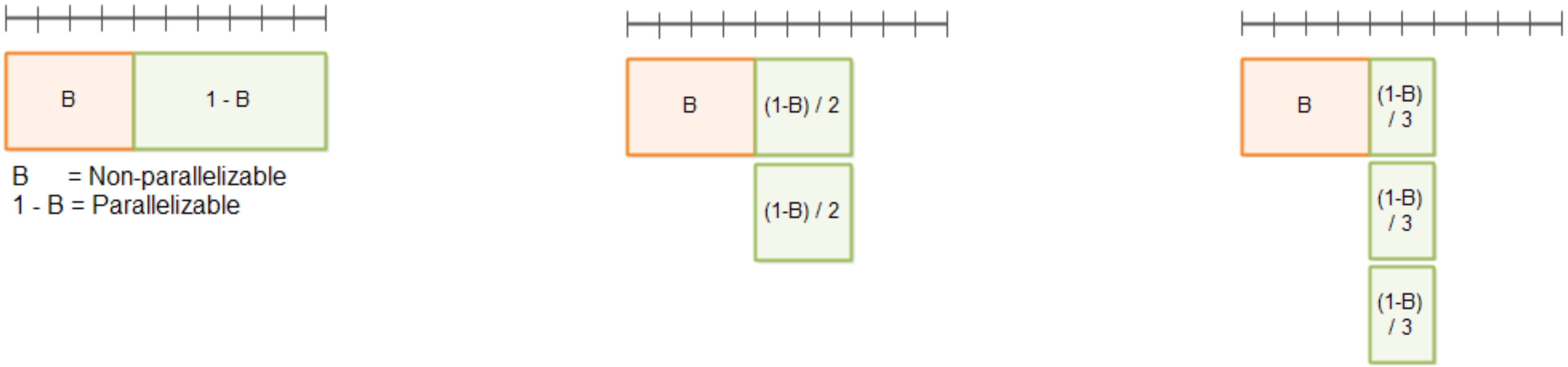
Improvements with concurrency

- Concurrency promises to perform certain tasks faster
  - A task = Several subtasks,
  - These subtasks can be executed in parallel

  → Save time (Better CPU Utilization)
- The theoretical possible performance gain can be calculated by *Amdahl's Law*

Amdahl’s Law Illustrated

- If **B** is the percentage of the program which can not run in parallel and **N** is the number of processes, then the maximum performance gain is 1 / (B + ((1-B)/N)).



Saving time

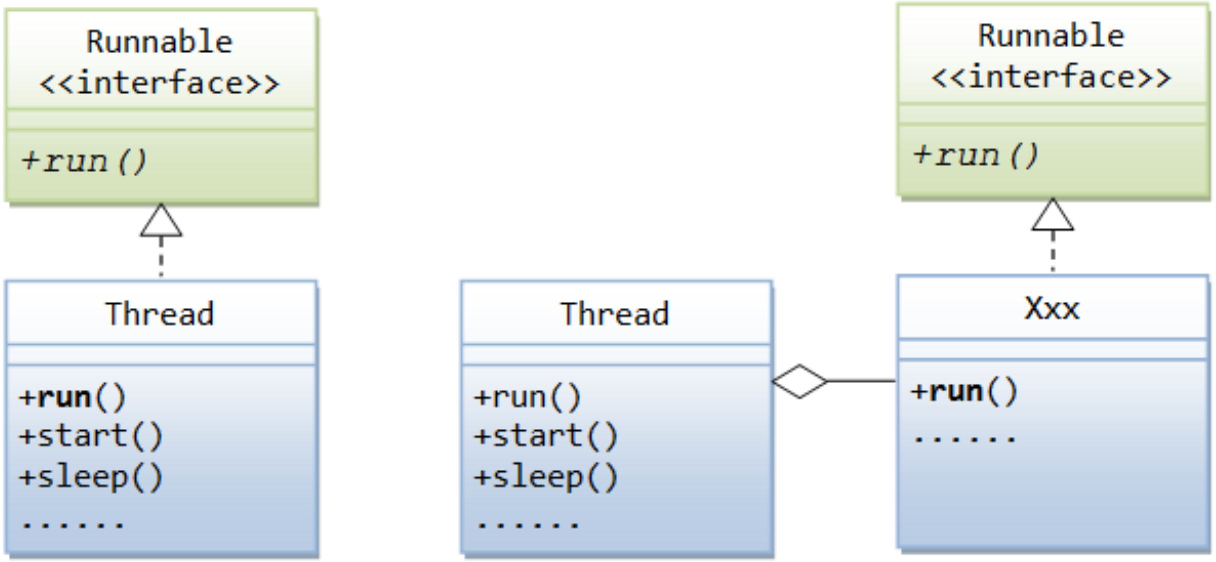
```
5 seconds reading file A
2 seconds processing file A
5 seconds reading file B
2 seconds processing file B
----------------------
14 seconds total
```

```
5 seconds reading file A
5 seconds reading file B + 2 seconds processing file A
2 seconds processing file B
----------------------
12 seconds total
```

Java Thread

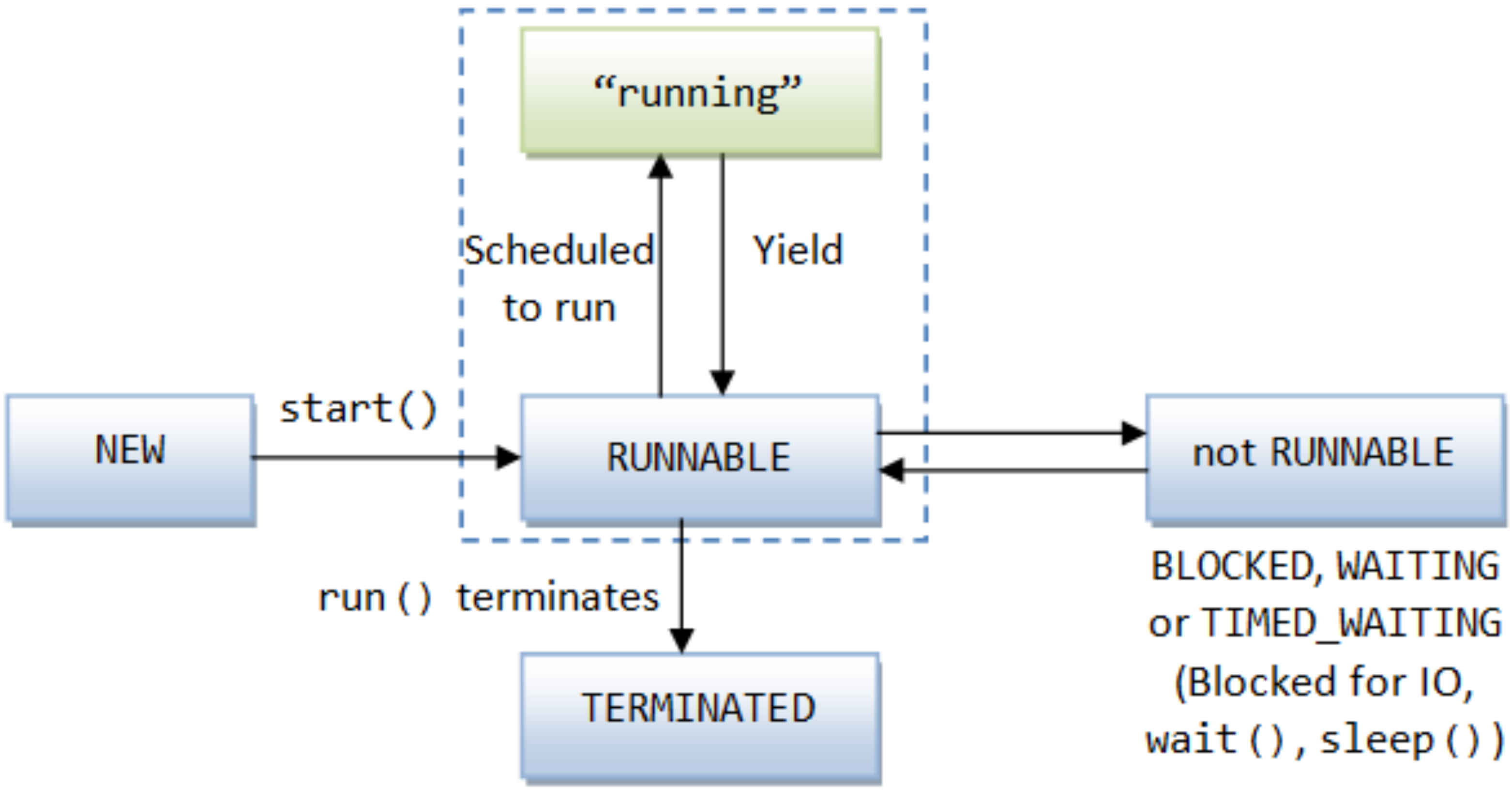
Creating and Starting a Thread

- By extending `Thread` class
- By providing a `Runnable` object
  - `Runnable` Anonymous Class
  - `Runnable` Lambda Expression



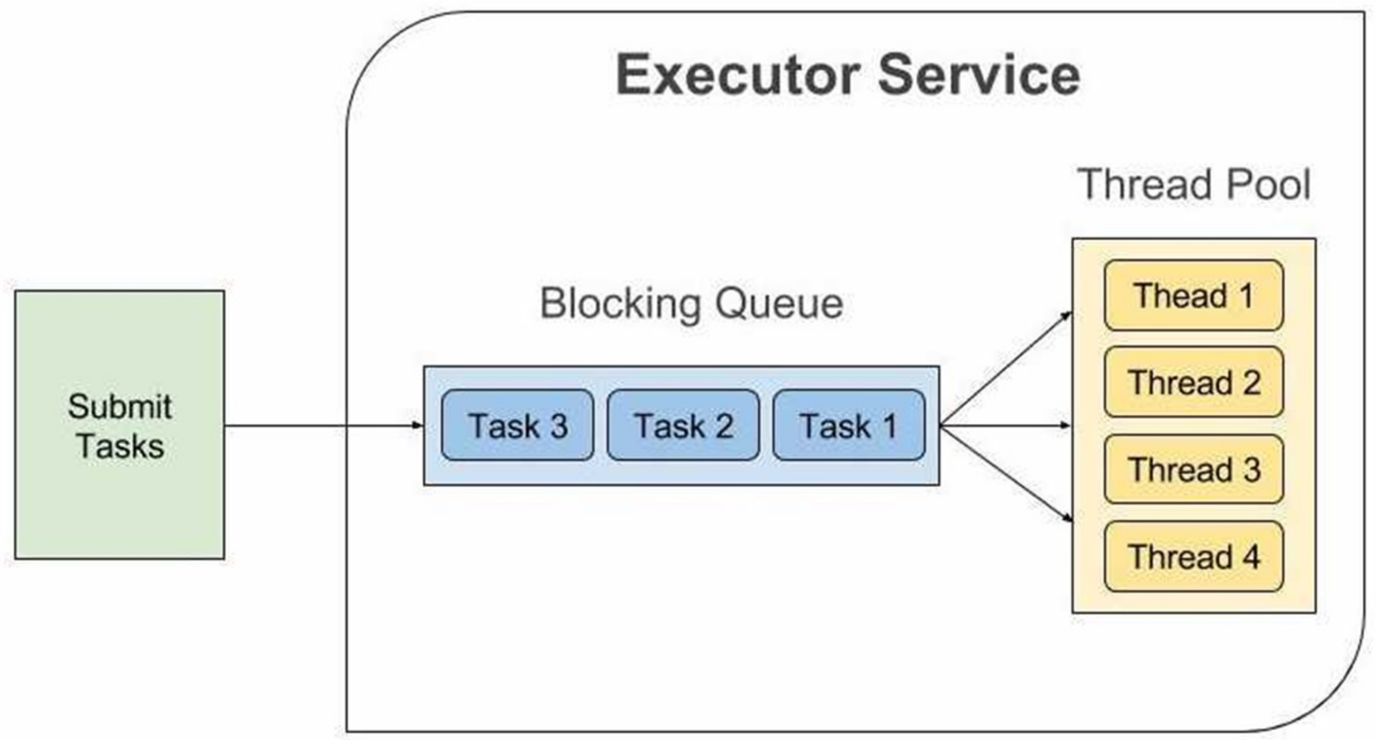
`Runnable` or `Thread`, which one to use?

The Life Cycle of a Thread



Executor Framework

Executor Framework



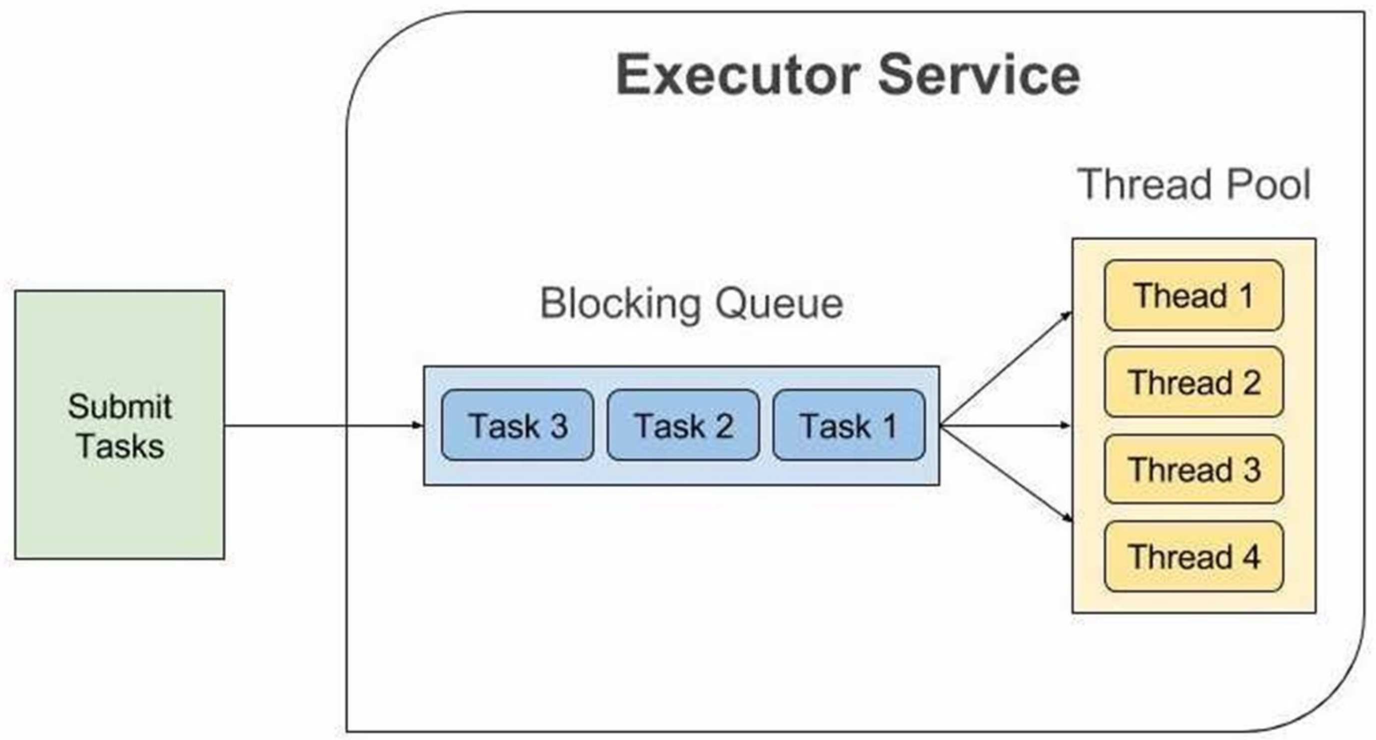
Executor Framework - Functionalities

1. **Thread Creation**: Methods for creating threads, a pool of threads, the application can use to run tasks concurrently.
2. **Thread Management**: Managing the life cycle of the threads in the thread pool.
   - You don't need to worry about whether the threads in the thread pool are active or busy or dead before submitting a task for execution
3. **Task submission and execution**: Methods for submitting tasks for execution in the thread pool and deciding when the tasks will be executed

Java executor interfaces

- **`Executor`** - A simple interface that contains a method called `execute()` to launch a task specified by a `Runnable` object.
- **`ExecutorService`** - A sub-interface of `Executor` that adds functionality to manage the lifecycle of the tasks. It also provides a `submit()` method whose overloaded versions can accept a `Runnable` and a `Callable` (discussed later).
- **`ScheduledExecutorService`** - A sub-interface of `ExecutorService`. It adds functionality to schedule the execution of the tasks.

Thread Pool


Executor Service
Thread Pool
Blocking Queue
Submit
Tasks
Task 3
Task 2
Task 1
Thead 1
Thread 2
Thread 3
Thread 4

Callable

`Callable`

- `Runnable` object to define the tasks that are executed inside a thread

→ **What if you want to return a result from your tasks?**

→ Java provides a `Callable` interface

- A `Callable` is similar to `Runnable`, except it can return a result and throw a checked exception
- `Callable` interface has a single method `call()` to contain the code executed by a thread

Callable examples

```java
Callable<String> callable = new Callable<String>() {
    @Override
    public String call() throws Exception {
        // Perform some computation
        Thread.sleep(2000);
        return "Return some result";
    }
};
```

```java
//a lambda expression
Callable<String> callable = () -> {
    // Perform some computation
    Thread.sleep(2000);
    return "Return some result";
};
```

Executing `Callable` tasks

- A `Callable` can be submitted to an executor service for execution
  - What about the `Callable`’s result?
  - How do you access it?
  - When the result of the submitted task will be available
- `Future` can be used to fetch the result of the task when it is available

Cancelling a `Future`

- Cancelling a future by using `Future.cancel()` method
- The `cancel()` method accepts `mayInterruptIfRunning`.
  - `True`: the thread that is currently executing the task will be interrupted
  - `Fase`: in-progress tasks will be allowed to complete.
- `isCancelled():` check if a task is canceled or not.
- After the cancellation of the task, `isDone()` will always be true.

`invokeAll` & `invokeAny`

Submit multiple tasks and wait for all of them to complete

- Executing multiple tasks by passing a collection of Callables to the `invokeAll()` method
  - `invokeAll()` returns a list of Futures
  - Any call to `future.get()` will be blocked until all the Futures are complete

Submit multiple tasks and wait for any one of them to complete

- The `invokeAny()` method accepts a collection of Callables and returns the result of the fastest `Callable`.

Synchronization

Issues with concurrency

- Threads have their call stack but can also access shared data
- **Access problem:** if several threads access and change the same shared data at the same time
  - Safety failure (inconsistent data)
  - Thread Interference Errors (Race conditions)
- **Visibility problem**: if thread A reads shared data, which is later changed by thread B and thread A is unaware of this change
  - Liveness failure (e.g., deadlocks)
  - Memory Consistency Errors

Synchronization

How do we avoid those problems?

1. Only one thread can read and write a shared variable at a time.
    - When one thread is accessing a shared variable, other threads should wait until the first thread is done.
    - This guarantees that the access to a shared variable is `Atomic`, and that multiple threads do not interfere.
2. Whenever any thread modifies a shared variable, it automatically establishes a *happens-before* relationship with subsequent reads and writes of the shared variable by other threads.
    - This guarantees that changes done by one thread are visible to others.

Synchronized Methods

- The `synchronized` keyword makes sure that only one thread can enter the sync methods at one time

Synchronized Blocks

- Java internally uses a so-called *intrinsic lock or monitor lock* to manage thread synchronization. Every object has an intrinsic lock associated with it.

Volatile Keyword

- `Volatile` keyword is used to avoid memory consistency errors in multithreaded programs.
  - It tells the compiler to avoid doing any optimizations to the variable.
  - If you mark a variable as `volatile`, the compiler won’t optimize or reorder instructions around that variable.

Locks

- `ReentrantLock` is a mutually exclusive lock with the same behavior as the intrinsic/implicit lock accessed via the synchronized keyword.
  - Thread that currently owns the lock can acquire it more than once without any problem.
- The `ReentrantLock` also provides various methods for more fine-grained control
- The `tryLock()` method tries to acquire the lock without pausing the thread.
  - If the thread couldn't acquire the lock because it was held by some other thread, then it returns immediately instead of waiting for the lock to be released.

ReadWriteLock

- `ReadWriteLock` consists of a pair of locks - one for read access and one for write access.
  - The read lock may be held by multiple threads simultaneously as long as the write lock is not held by any thread.
- `ReadWriteLock` allows for an increased level of concurrency.
  - It performs better compared to other locks in applications where there are fewer writes than reads.

`Atomic` Variables

- Java’s concurrency API defines several classes in `java.util.concurrent.atomic` package that support Atomic operations on single variables.
- Atomic classes internally use compare-and-swap instructions supported by modern CPUs to achieve synchronization. These instructions are generally much faster than locks.
- The `AtomicInteger.incrementAndGet()` method is atomic,
  - Safely call it from several threads simultaneously and ensure the access to the count variable will be synchronized.

Java Virtual Thread

Java (normal) Thread

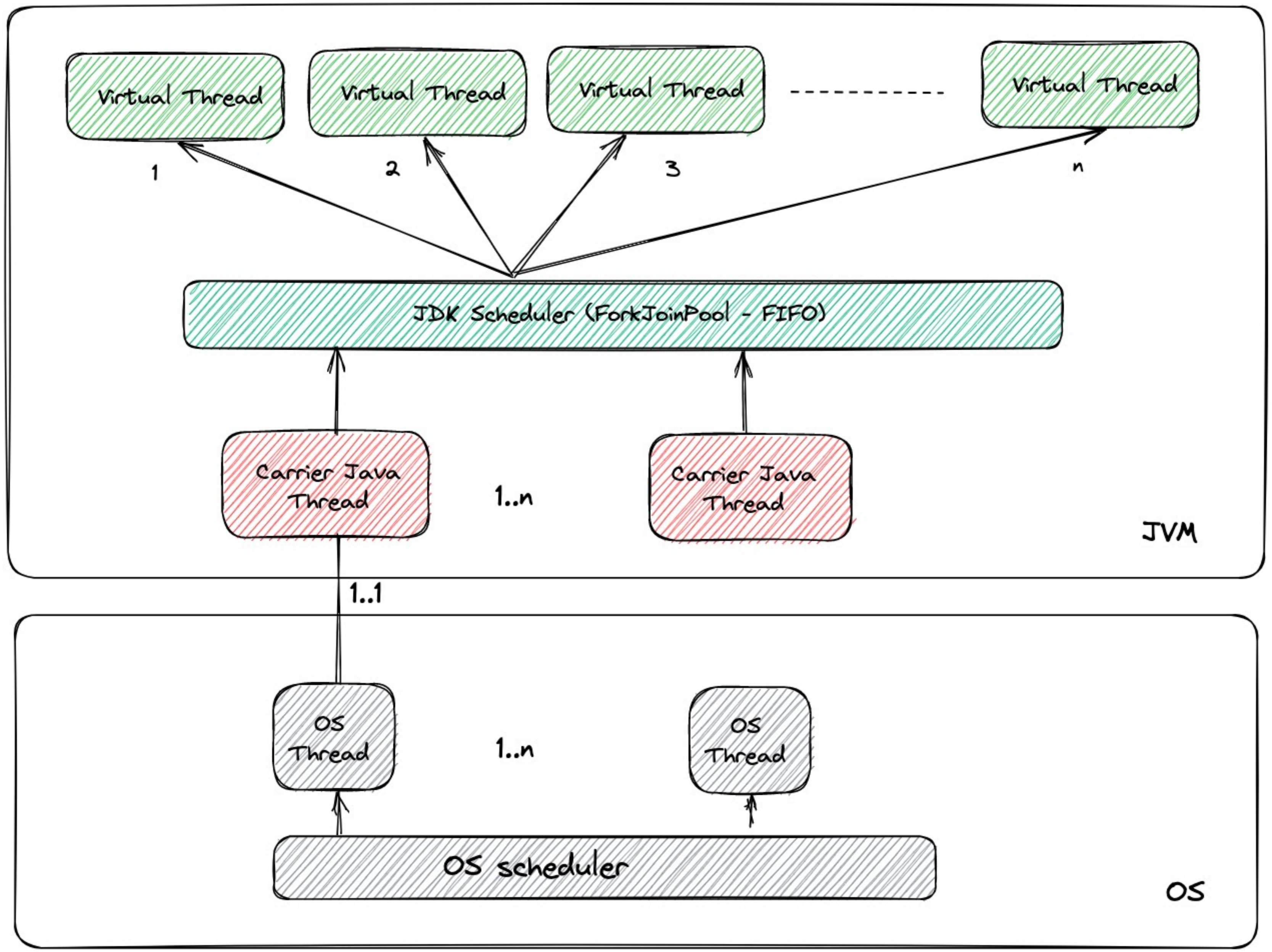
- To create a new kernel thread, we must do a system call, and that’s a costly operation
  - → Using thread pools instead of reallocating and deallocating threads as needed
- Scaling up application by adding more threads
  - → Context switching + Memory footprint
  - → Cost of thread maintaining may be significant and affect the processing time

Java Virtual Thread

- Virtual threads are managed by the JVM
- Their allocation doesn’t require a system call
- They’re free of OS’s context switch
  - Virtual threads run on the *carrier kernel thread* used under-the-hood

  →More virtual threads…

  →Managing virtual threads is much cheaper


Virtual Thread
Virtual Thread
Virtual Thread
Virtual Thread
1
2
3
n
JDK Scheduler (ForkJoinPool - FIFO)
Carrier Java Thread
1..n
Carrier Java Thread
JVM
1..1
OS Thread
1..n
OS Thread
OS scheduler
OS

API is the same!!!

```java
Runnable printThread = () -> System.out.println(Thread.currentThread());

ThreadFactory virtualThreadFactory = Thread.builder().virtual().factory();
ThreadFactory kernelThreadFactory = Thread.builder().factory();

Thread virtualThread = virtualThreadFactory.newThread(printThread);
Thread kernelThread = kernelThreadFactory.newThread(printThread);

virtualThread.start();
kernelThread.start();
```