(Video : 35)

3rd Feb & 5th Feb

Future, Callable, Executor.submit().

ThreadPoolExecutor executor = new ThreadPoolExecutor(x, y, z, ...);

executor.submit(____);

⇒ Inside this You provide what needs to be done.

1) Runnable.

⇒ We know the Functional Interface Runnable.

```
@FunctionalInterface
public interface Runnable {
    @ public void run();
}
```

In executor.submit(Runnable); ⇒ Can be submitted.

⇒ returns Nothing

```
executor.submit( () -> {
        System.out.println( ____ );
});
```

2. Callable –

```
@FunctionalInterface
public interface Callable<V> {

    public V call();
}
```

↳ has a return type

executor.submit(Callable) → that's what we are submitting

```
Future<Integer> f = executor.submit(
    () -> {
        return 10;
    });

try {
    Integer g = f.get();
}
catch (Ex e){
}
```

3) (Runnable, T)

executor.submit(Runnable, T);

↳ this type basically will run the Runnable method and whatever you pass in T will return the T. If you don't do anything with it, same T will get returned.

If You do something in the run() method it will respond accordingly.

```
MyClass extends Runnable {

    List<Integer> objList;

    public MyClass( List<Integer> objList){

        this.objList = objList;
    }

    @ public void run( ){

        if (objList != null){

            objlist.add(10);
        }
    }
}
```

```
List<Integer> objList = new ArrayList();

MyRunnable runnableObj = new MyRunnable(objList);
```

```
Future<List<Integer>> fList = executor.submit(
        runnableObj, objList);
```

(executes the run method and however well or whatever it does with T, it replaces it.)

```
try{
    fList.get();
}
catch(Exception e){}
```

I know, it looks quite redundant, if Callable can do the same

→ Callable doing same thing.

```
Future<List<Integer>> fList = executor.submit(
    () -> {
        List<Integer> l = new ArrayList<>();
        l.add(10);
        return l;
    });
```

Video 35 → CompletableFuture, I need more hands on to understand properly.

Watching Video 36:

We studied about ThreadPool Executor already.

Few more executors to study

Below are ExecutorService ⇒ here we can only customize parameters Not like PoolTaskExecutor.

Fixed Thread Pool Executor.

Min/Max is same size. Executors.newFixedThreadPool(5)

Queue Size → Unbounded.

Even when idle threads will be Alive.

Cached Thread Pool ⇒ Executors.newCachedThreadPool();

min → 0

max → Integer.MAX_VALUE,

No queue → Request comes and thread is created dynamically.

Used for concurrent bursts of short lived task.

~~Q) Click button~~ to

Thread Alive time → 60 seconds

Krishan

I asked chatGPT to give me an example for new CachedThreadPool where it can be used.

Chatgpt →

Logging → [writing to database / file]

When spike is coming, new threads are being spawned. And mind you new threads are being created parallelly on the fly to spread

Logging is done.

These threads can be reused for future load coming to the system, if it has not been idle for 60 seconds

Krishan → I asked by thread creation takes time won't it consume time.

Chatgpt → (Parallel Creation of Threads)

Instead of one-by-one creation, new threads are spawned in parallel, so many threads are created.

Q) When does thread creation become a bottleneck?

A) 1) If suddenly 1000+ new threads are required.

2) If tasks are too short lived, creation/destroying will take a lot of time like thrashing.

3). CPU already heavily loaded and spawning new threads will put more load.

Single ThreadExecutor

(min : 1
maxThread : 1)

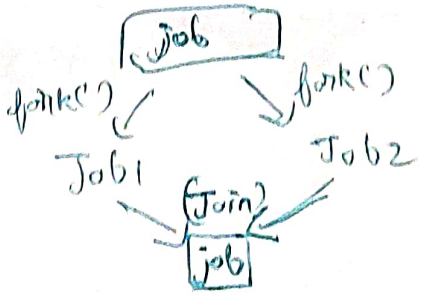
Queue Size : Infinite.

No Concurrency.

Fork Join Pool.

Work Stealing Pool

Job of this pool breaks tasks into subtask and evaluate in parallel.



We will implement & write code for ForkJoinPool.
Later, with a very good understanding.

Implement using (RecursiveTask)

Video 37:
- ThreadPoolExecutor. Shutdown / awaitTermination / ShutdownNow

Q1) Shutdown.

→ When you call `poolExecutor.shutdown()`.

It does not accepts new task, finish the tasks already in queue and thats it.

It shutsdown after that

Rejected Exception, if new task is submitted.

2) Await Termination → Simple true / false it returns.

```
poolExecutor.shutdown()
try{
    boolean isOff = poolExecutor.awaitTermination(2, TimeUnit.SECONDS);
                    ⇒ returns true/false if shutdown is completed or not.
}
catch(Exception e){
}
```

3) ShutDown Now → Halts all processing. Shuts down the executor & returns the list of Executing tasks.

ScheduledThreadPoolExecutor → Various flavours are there of it

1) You can add Delay → tell it to run of ~~x~~ seconds.

2) You can add interval → tell it to run after every ~~x~~ seconds.

Mostly polling can be done using this.

2) Await Termination → Simple true/false it returns.

```
poolExecutor.shutdown()
try{
    boolean isOff = poolExecutor.awaitTermination(2,
                                    TimeUnit.SECONDS);
                ⇒ returns true/false
}                   if shutdown is completed
catch(Exception e){     not.
}
```

3) ShotDown Now → Halts all processing. Shuts down the executor & returns the list of Executing tasks.

ScheduledThreadPoolExecutor → Various flavours are there of it

1) You can add Delay → tell it to run of ~~x~~ seconds.

2) You can add interval → tell it to run after every ~~x~~ seconds. Mostly polling can be done using this.

(Video 38) : Normal Vs Virtual Threads / (ThreadLocal)

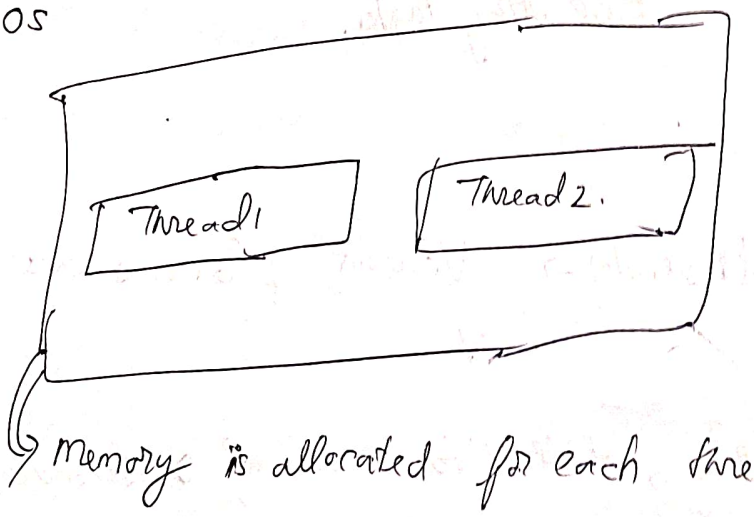
So Reality of today is ThreadLocal went over my head:-

I am yet to understand ThreadLocal properly.

Normal V/S Virtual Threads:

We know about Normal Threads.

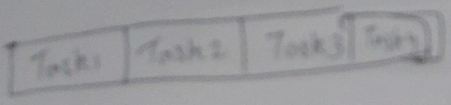
Lightweight Process. We want to create a subtask to do some work.



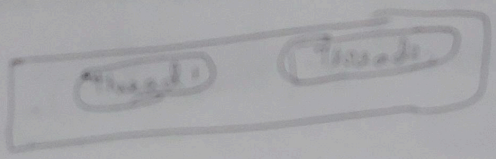
Memory is allocated for each thread.

And thread creation also takes time.

Now let say



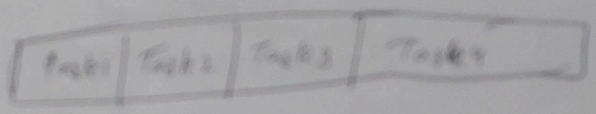
→ waiting Tasks



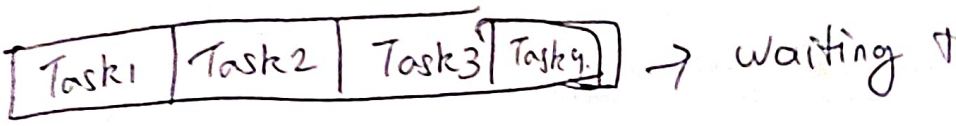
Thread1 → Already executing some task

Thread2 → Already executing some task

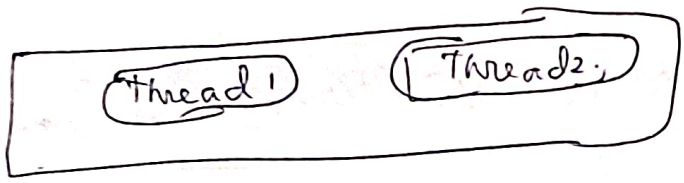
Now while executing, Suppose some I/O call, DB call happens. Thread goes into Blocked state, until then Thread is there idle and still it is being not utilized to pick any of the waiting Tasks from the queue



Now Let's say-



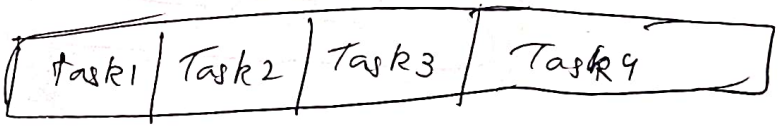
→ waiting Tasks



Thread1 → Already executing some task.

Thread2 → Already executing some task.

Now while executing, suppose some I/O call, DB call happens. Thread goes into blocked state, untill then thread is there idle and still it is being not utilised to pick any of the waiting Tasks from the queue.



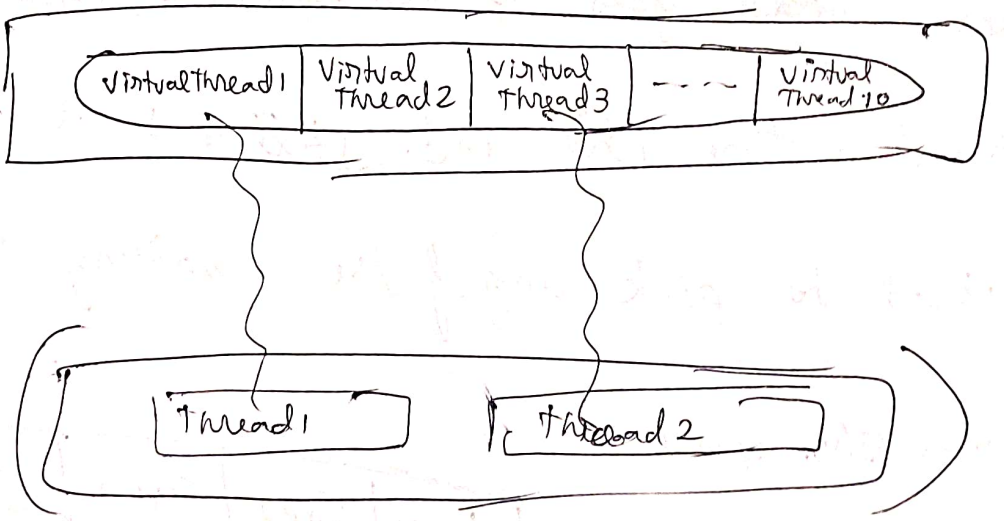
What do Virtual Threads do better?

JVM creates a lot of virtual threads. Which are not real threads created at OS (Kernel level).

These threads are lighter than actual threads.

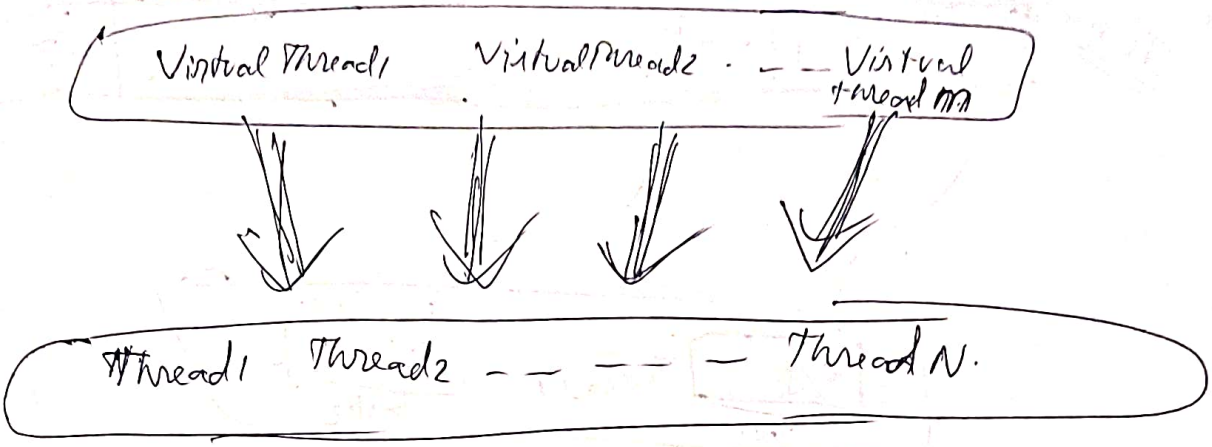
(Lets suppose.)

JVM



⇒ In OS has memory

At any time a particular Virtual thread is mapped to any one of the Normal thread

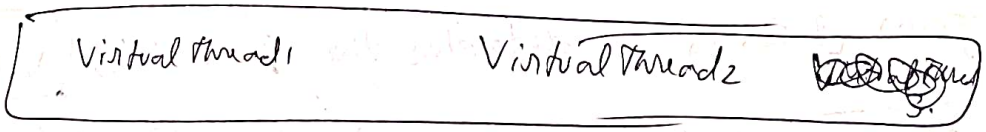


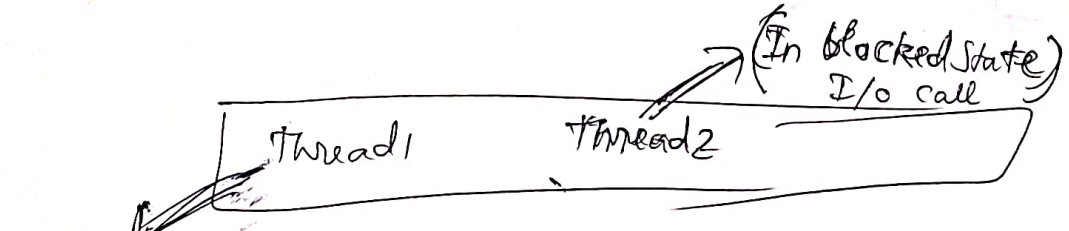
(M Virtual Threads) are multiplexed.

into (N Normal Threads)

Lets take the previous example.

TaskQueue → | Task1 | Task2 | -- | TaskN |

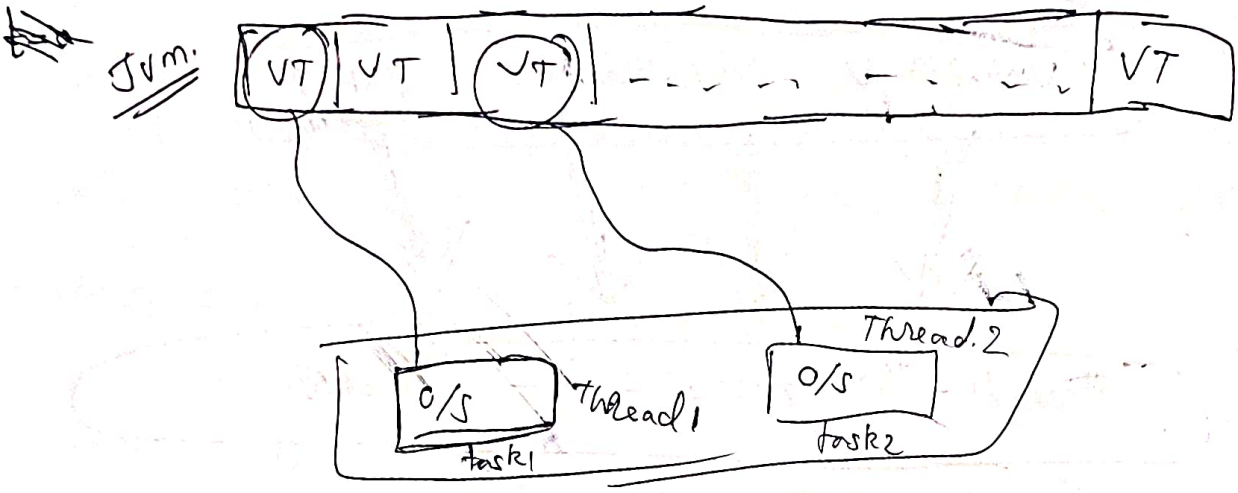




Now instead of Not Utilizing Thread2 and letting it be idle.

While I/O is happening.

Virtual Thread will pick one task and assign it to the idle Thread.



** There is no tight coupling b/w Virtual thread & Real thread.

Lets say Task 2, went for DB call which takes 4 seconds.

So until that O/S Thread 2 will sit idle for 4 seconds.

** What JVM does is, it detaches the link b/w [Virtual Thread] & [O/S Thread] - And gives a new Virtual Thread to execute its Task.