JAVA 21 VIRTUAL THREADS

Index

- Intro

- The role of a ***virtual thread*** is to avoid blocking a *platform thread* by unmounting itself.

- You will learn why they have been added to the JDK; how you can create them; how you can use them; what performances you may expect from them; and how they are working under the hood.

- Why Virtual threads have been added?

- The effort to add them was huge, it took years, the pull request on the JDK was massive, so there must be a good reason.

- Virtual Threads are there to solve a problem that has to do with asynchronous programming.

- Suppose you have to query a remote database for a User object. You have the name of the user, and you need to fetch the full User object from your database. You are probably going to write this kind of code:

```
Json request =
   buildUserRequest(name);

String userJson =
   userServer.userByName(request);
```

- First, you need to create a request, with all the information needed, probably in the form of a Json object. And then you need to launch this request on the network. After a while you get a response, probably in the form of another Json object, and then you unmarshal this object to recreate your User object. This is the kind of code you end up with, simple to write, simple to read, simple to debug, simple to test, and simple to maintain.

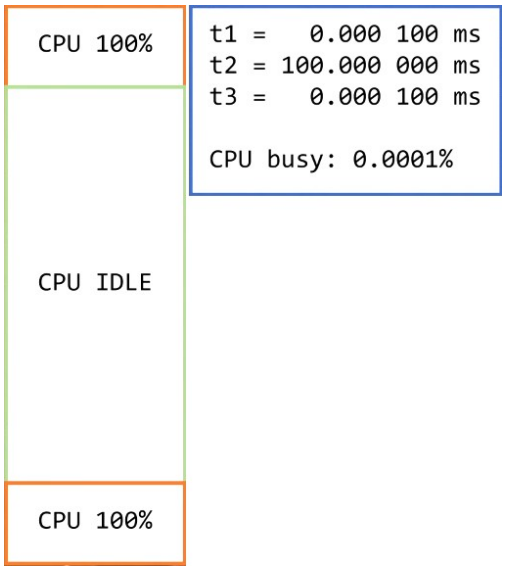
- How busy could your CPU be when it is performing this request?
  - The first step is pretty simple, it's just in-memory computations, so you can assume that it will run in the order of several tens of nanoseconds.

  - The second step is a request over the network. This one takes much longer, probably in the order of the hundred of milliseconds. And your CPU is not doing anything during this time, waiting for the response to come from the server.

  - The third and last step is also in-memory computation: the unmarshaling of a Json object, so that's another tens of nanoseconds.

- On the overall, this code executes in about a hundred milliseconds. So to compute how busy your CPU is, you just need to divide the time it takes to prepare the request and unmarshal the Json response, that is several tens of nanoseconds, and the time your CPU is waiting for a response from the server. And that gives you

0,0001%, that's the order of magnitude. In a nutshell, your CPU is idle during this request.



- Why the *one request per thread* model cannot work

- The first solution you may be thinking about is to launch more than one request in parallel. And the classical model is to have one request per thread, and launch as many threads as you can. This has been working for years, years ago. The real question is: can you keep your CPU busy 100% of the time with such an approach? So let let us do the math. How many threads would you need to keep your CPU busy 100% of the time. Well, the math is actually pretty simple, with 10 threads you have a CPU usage of 0,001%, 100 threads will give you 0,01%. With 1000 threads you can expect 0,1% CPU usage, and with 10,000 threads, 1%. To have a CPU busy 100% of the time, you need a million threads with this model.

- It turns out that a thread in Java is actually a thin wrapper on an operating system thread, sometimes called a ***kernel thread***, or a ***platform thread***. And this thread consumes quite some resources.

→ About “a Java thread as a thin wrapper on an OS thread” (form GPT)

- When we say that *a thread in Java is a thin wrapper on an operating system thread*, we mean that each *Java thread* is associated with an *operating system thread*.

- The JVM abstracts the details of managing these *OS threads*, making it easier to work with threads in Java.

- The JVM provides a consistent way to create, manage, and control threads within a Java application.

- By serving as a wrapper, the JVM provides a layer of abstraction that shields the developer from the complexities of working directly with the operating system's threads. This allows developers to focus on writing application logic without needing to delve deeply into low-level operating system details.

- The JVM manages the mapping between Java threads and operating system threads, enabling the efficient execution of concurrent tasks within a Java application.

- Typically, creating a thread takes 2MB of memory, upfront. So creating a million of them would require 2TB of memory. Which is a lot. But there's worse: creating a thread also takes time, in the order of the millisecond. So creating a million of them would actually require a thousand seconds, which is a little more than 15mn.

- Actually, if you try to create threads in a loop, depending on your operating system, you'll probably hit a limit long before that, at about a few thousand threads. So if you use this approach you will quickly reach that limit, and your CPU usage will still be very low, in the order 1%.

- Unfortunately, the solution that consists in creating one thread for each request is not a viable solution to increase your number of requests per second. This has been known for years, and this is why other approaches have been explored, with success.

- Solving this problem: less expensive threads or asynchronous code?

- You can see that you have actually two solutions to improve this situation. On the one hand, you want to increase the number of requests per second. To achieve that, you need to launch your requests in parallel. Launching one request per kernel thread does not lead to a viable solution, because kernel threads are way too expensive. So at this point you have indeed two solutions:
    - Either you create new type of threads that are less expensive than the current kernel, or platform threads.
    - Or you launch more than one request per platform thread.
        - Launching several requests per threads means that you cannot write your code using the imperative approach anymore.
        - What you need to do is to split this code into small atomic operations. Each operation gets inputs, is doing one thing, and produces a result. So what you need to do is to write all these operations as lambdas, and then you need to wire them together using an asynchronous framework.

        - In a nutshell, the job of that framework is to call your lambdas with the right inputs, to get the outputs, and to transmit these outputs as inputs to the next lambda you defined.

        - What you are creating are pipelines of operations, that process your data, and that are executed by the framework you are using.

        - The responsibility of this framework is to correctly execute your operations in the right order, and distribute them among the threads it has, to keep your CPU busy, and your number of requests per second optimal.

        - At some point, one of your lambda will launch a request on the network. It should then immediately return, to make sure that it does not block the thread that is running it.

        - Avoid writing blocking lambdas. If your lambda is done, then the thread has nothing more to execute, it is free to do something else.

        - Your request has been launched, and there is a handler somewhere that will trigger a signal when the response is there. Your framework knows this handler, it knows when the data from the response is available, then it is its job to run your next lambda, the one you defined that will react on this data. This lambda is invoked, once again, by your framework, with this data as an input. Because these two operations are two different lambdas, the framework thread can do something else in the meantime, like launching more requests, or running other lambdas.

        - There is obviously an overhead in doing that, nothing is ever free, but it is still much cheaper and much more efficient than blocking a thread.

    - This second solution has been thoroughly explored and optimized over the last 10 years, and this is what asynchronous frameworks are doing, with great success. Using this technique, you can dramatically increase your number of requests per second, and you have many frameworks to choose from to do that.

    - What does it require from you, as a developer? You need to change the way you write your code. Remember, you need to create atomic, non-blocking lambda operations.

- Writing an imperative, blocking online shopping example

- I would like to take you through an example, that I borrowed from Tomasz Nurkiewicz, a famous author and speaker, and a renowned expert in asynchronous programming.

- The problem you need to solve is the following, and it is a very classical problem. A user is doing some online shopping, and what you want to do is make sure that everything is saved in a database, and that you send this user an email with a receipt at the end of this transaction.
    - The first step consists in making sure that the user is saved in your database.

- Then you fetch the shopping cart.
- And then you loop over the items that person chose, to compute the total price, pretty simple.
- And then you invoke the payment service, and record the transaction id.
- And with all these elements, you send the email with all the details of the transaction.

```
User user = userService.findUserByName(name);
if (!repo.contains(user)) {
    repo.save(user);
}
var cart = cartService.loadCartFor(user);
var total =
        cart.items().stream()
                .mapToInt(Item::price)
                .sum();
var transactionId =
   paymentService.pay(user, total);
emailService.send(user, cart, transactionId);
```

- All this is pretty easy, and the code is simple enough to understand. Actually, anybody can easily check it, to make sure that the business process has been correctly implemented, because it's just a simple piece of code. If you need to debug this code for some reason, executing it step by step is easy. Writing unit tests is also super easy. More than that, if there is business requirement that tells you: "oh, if a person spends more than 100 on our site, we would like to give that person a coupon", you know exactly what to do.

- Making this code asynchronous the CompletableFuture API

- The only problem is that, as you saw, it doesn't scale, and what you need to do is to adapt this code to the asynchronous framework your application is using.

- Adapting the previous code to the *CompletableFuture API* (that's part of the JDK) looks like this:

```
var future =
supplyAsync(
    () -> userService.findUserByName(name))
  .thenCompose(
    user -> allOf(
      supplyAsync(
        () -> !repo.contains(user))
          .thenAccept(
            doesNotContain -> {
              if (doesNotContain)
                repo.save(user);
              }
            ),
            supplyAsync(
              () -> cartService.loadCartFor(user))
                .thenApply(
                  cart ->
                    supplyAsync(
                      () -> cart.items().stream().mapToInt(Item::price).sum())
                        .thenApply(
                          total -> paymentService.pay(user, total))
                        .thenAccept(
                          transactionId -> emailService.send(user, cart, transactionId)
                        )
                )
    )
);
```

- The imperative version of this code was easy to follow, and thus, it was easy to spot in the code if a mistake was made in implementing the business process. In this mess, how can you check that?.

- Let me just highlight the portions of the business code in it.

```
var future =
supplyAsync(
    () -> userService.findUserByName(name))
  .thenCompose(
    user -> allOf(
      supplyAsync(
        () -> !repo.contains(user))
          .thenAccept(
            doesNotContain -> {
              if (doesNotContain)
                repo.save(user);
              }
            ),
            supplyAsync(
              () -> cartService.loadCartFor(user))
                .thenApply(
                  cart ->
                    supplyAsync(
                      () -> cart.items().stream().mapToInt(Item::price).sum())
                        .thenApply(
                          total -> paymentService.pay(user, total))
                        .thenAccept(
                          transactionId -> emailService.send(user, cart, transactionId)
                        )
                )
    )
);
```

- What you see is that they are drowned in a mess of technical code, making it super hard to understand how these pieces of code actually work together.

- Even the **types returned** by these elements of business code are hard to spot, because they are not there. What is the return type of this call to *paymentService.pay()*? How can you check that you are returning the right thing?

- Suppose that you need to implement the new business requirement we just talked about a minute ago: "If a person spends more than 100 on our site, we need to give that person a coupon". Where are you going to add this code? And how can you add it in this mess? But that's not the only issue, unfortunately.

- Issues with the asynchronous code: debugging, testing, stack trace, exception handling, timeout handling

- Debugging such a code, running it step by step, is next to impossible.

- Writing unit tests is also super hard. All you can test is the process as a whole, which is not really a unit test anymore.

- If something goes wrong in the imperative version of the code, the ***stack trace*** will give you the exact *context* in which your code is running.

- All the lambdas you wrote in the asynchronous version are executed by your framework and they are executed asynchronously, which means **out of the context** of your business process, since the asynchronous code is not part of the stack trace.

- The element of your application that triggered this processing, was just there to declare your processing pipeline, and it is now gone, doing something else. It is not waiting for any response anymore. Meaning that it is not in the *stack trace* anymore.

- The real element that is running all these lambdas, getting the results and forwarding the results to the other lambdas is your framework. So if you examine the *stack trace*, you do not see your ***application context***, all you see is your framework calling your lambdas.

- It also makes exception handling super hard. If you are lucky and an exception is thrown within your lambda code, then the stack trace will take you there, and you can do something. But if one of your lambda returns a null value for instance, that triggers a NullPointerException **in the framework code**, then you are in trouble because the stack trace does not tell you where this null value came from.

- Suppose one of your lambda doesn't return, because it is waiting for a response that is not coming. If you are lucky you have a timeout somewhere that was set up in your code. But then again, the stack trace tells you that a timeout exception was thrown, with no information about what lambda actually triggered the timeout. And if no timeout handling was set up, then you have a stranded lambda, that is preventing a process from producing a result. This lambda is lost somewhere, in some event queue, never to be activated again. And next to impossible to spot.

- Not having a meaningful stack trace has a cost: it makes your code hard to debug, and for the same reasons, impossible to profile.

- Asynchronous programming can give great performances, but the cost of maintenance is actually super high.

- The asynchronous approach can dramatically improve your requests per second performances, which is what you need, but you also understand that it comes with a **maintenance cost**.

- Making threads less expensive: by how much?

- Now you are left with the first solution we had: making threads less expensive.

- We need a new type of thread that is a thousand times less expensive than the current model, to be able to launch a million of them. If this can be achieved, then writing your code in an imperative, blocking way becomes compatible with the number of requests per second that you need. And you will not need to write asynchronous code anymore.

- Would it work if the gain is only a hundred times less expensive instead of a thousand? Well, no, because in that case your CPU usage would be only 10%. And if the gain is 500, then the CPU usage would be 50%. So to reach max throughput,

you would still need to rely on asynchronous code. And this new type of thread would actually be useless. So gaining a factor of a thousand, is actually the minimal gain you want to achieve.

# Threads	% CPU usage
1	0.000 1%
10	0.001%
100	0.01%
1 000	0.1%
10 000	1%
100 000	10%
1 000 000	100%

- Creating and using virtual threads

- Virtual threads from the JDK 21 are much lighter than platform threads, by a factor of more than a thousand.
- You can easily launch a million of them, even on a small machine.
- Virtual threads are so lightweight, that you do not need to pool them anymore. When you need one, just create it and let it die when you do not need it anymore.
- Underneath you still have platform threads, because this is the tool operating systems are using to execute several tasks in parallel, and you cannot avoid using them. So the trick is still about running several tasks per platform thread.
- A virtual thread has to run on top of a platform thread, because there is no other solution. And if you want to have a million of them, it means that they need to share these platform threads.
- Let us first create a first virtual thread an run it. For that, let us go back to our previous example, and let us make the imperative code we wrote a *Runnable*, so that we can run it in a thread we create. And then you need to create a virtual thread, which is very easy too.

```
Runnable task = () -> {
  User user = userService.findUserByName(name);
  if (!repo.contains(user)) {
      repo.save(user);
  }
  var cart = cartService.loadCartFor(user);
  var total =
          cart.items().stream()
                  .mapToInt(Item::price)
                  .sum();
  var transactionId =
     paymentService.pay(user, total);
  emailService.send(user, cart, transactionId);
};
```

- You now have two factory methods added to the Thread class: *ofVirtual()*, to create virtual threads, and *ofPlatform()*, to create platform threads.

```
Thread virtualThread =
   Thread.ofVirtual()
         .unstarted(task);

virtualThread.start();
// do something else
virtualThread.join();
```

- Of course you can still use the old pattern, that is calling new Thread() directly. You can then use one of the methods available on the builder you get to create exactly what you need: you can set the name, and you can allow or disallow the use of thread local variables.

- There is also a factory method that has beed added to the *Executors* factory class, called *newVirtualThreadPerTaskExecutor()*. The executor service you get actually creates a virtual thread on demand, everytime you submit a task to it. It does not pool anything.

```
var service =
  Executors.newVirtualThreadPerTaskExecutor();
```

- This executor service may be very useful if you have a concurrent application for instance, built on such an executor service, and you want to check what kind of gain virtual threads can give you.

- Because a virtual thread is a thread, you can do with it everything you can do with a regular thread: you can start it, you cannot stop it, or suspend it, or you can join it for instance.

- There is one thing that you cannot do with a virtual thread, which is you cannot make it a non-daemon thread.

- A virtual thread is always a daemon thread. So having live virtual threads in your application will not prevent it from exiting. It also means that all the problems you may come across with platform threads are still there. Race conditions, deadlocks, visibility, tearability, all this good concurrent programming stuff, is still there with virtual threads. So be aware of that.

- The good news is that it also means that all the solutions that you know to ensure the correctness of your code and the safety are also the same.

- When a virtual thread is executing a *Runnable*, internally, there is a special pool of platform threads, which is by the way a modified fork join pool, that is there to run your virtual threads.

- Your virtual thread is mounted on a platform thread from this pool, and your task is executed by this platform thread, through your virtual thread.

- If you print a virtual thread, you will get this result:

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


Thread.ofVirtual()
      .start(task)
      .join();

> VirtualThread[#22]/runnable@ForkJoinPool-1-worker-1
```

- It shows you that your virtual thread is mounted on the worker 1 of the Fork Join Pool 1.

- There is one platform thread for each core of your CPU.

- At some point in our task, the code is making a request on your database. It is calling repo.contains(user). This is a blocking code, and you know that you want to avoid blocking a platform thread.

- The role of a virtual thread is to avoid blocking a platform thread.

- Your virtual thread detects that it is actually executing a blocking I/O operation. And instead of blocking the platform thread it is running on, it unmounts itself from this platform thread, moving its own context, that is its stack to somewhere else in the memory, namely, in the ***heap*** memory.

- The virtual thread is executed by a special object called a *Continuation*. This object is in the non-public part of the API, so you should not use it in production.

- This *Continuation* has a *run()* method that executes your virtual thread, and that in turn execute your task.

- When a blocking call is issued in the Java NIO API for instance, a call to *Contination.yield()* is made to unmount your virtual thread from the platform thread, and copies the stack of your virtual thread to the heap memory.

- *Contination.yield()* is only invoked if this blocking call is being executed in the context of a virtual thread. It works like your virtual thread was saying: "Hey, I won't be needing the CPU for a while, because I am waiting for a network response, so please, remove me from this platform thread".

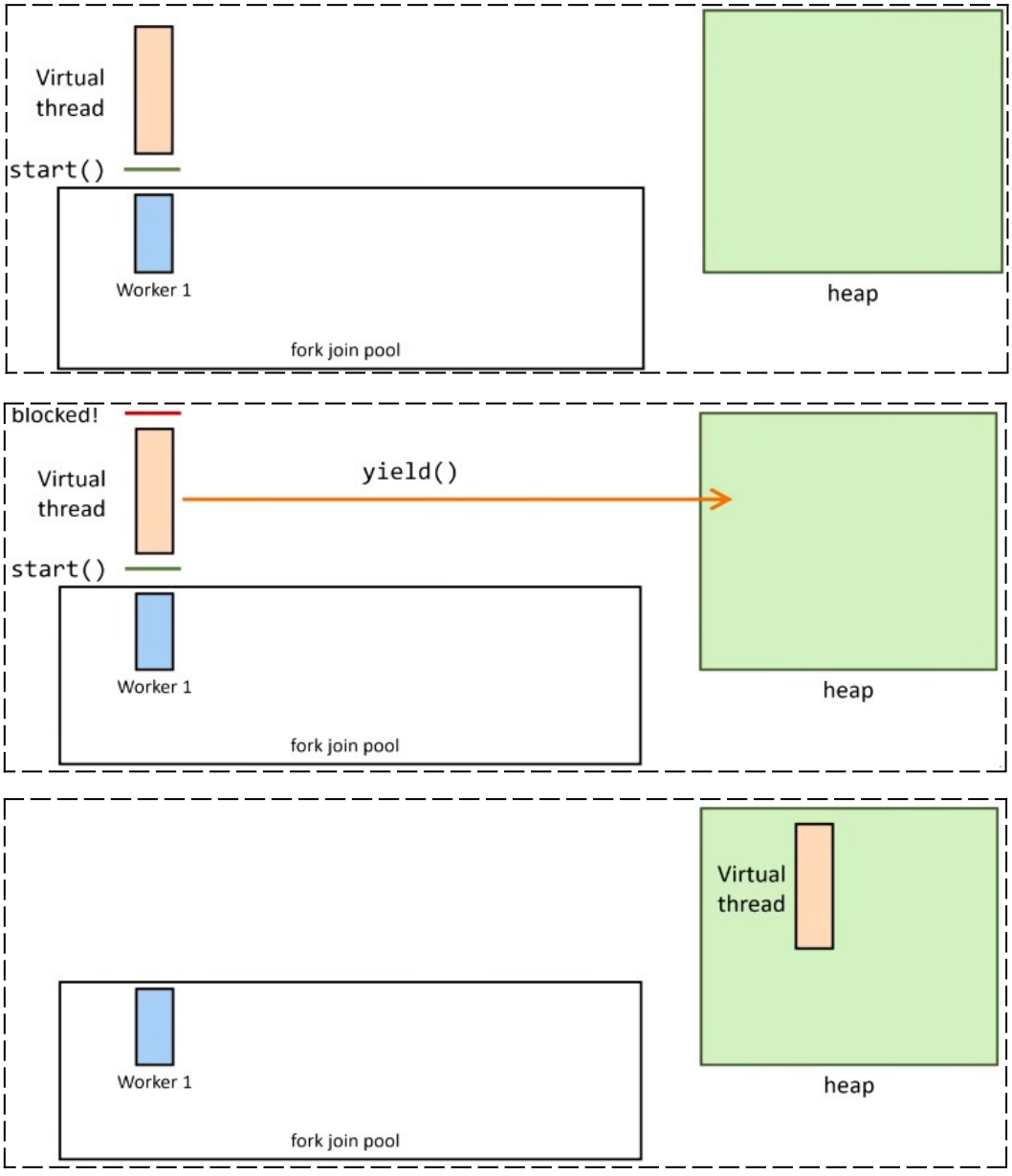
- Blocking calls are not found only in Java NIO, actually all the blocking calls you can think of in the JDK have been refactored to call this *Continuation.yield()*. And that also includes the synchronization objects from java.util.concurrent.

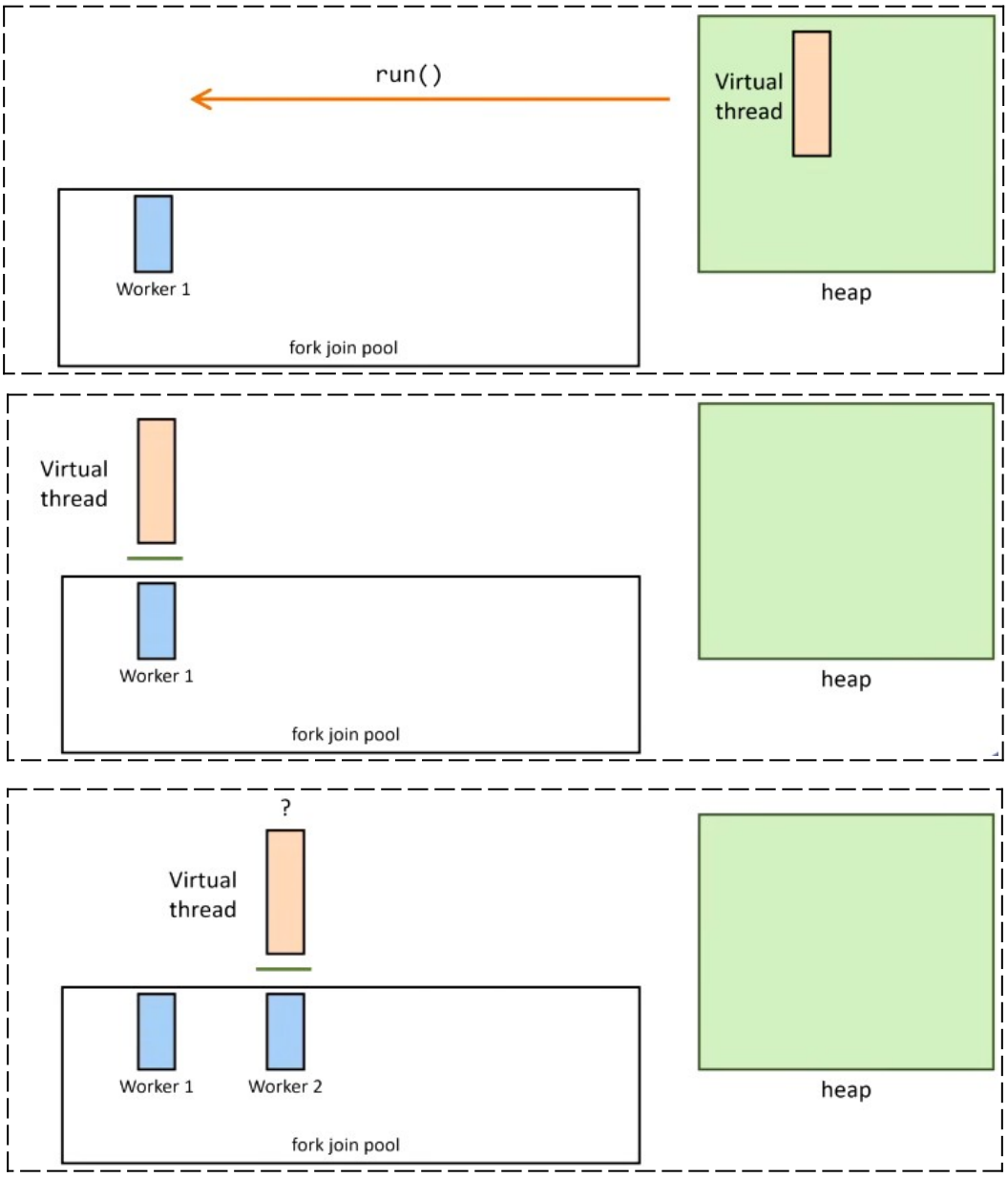
- Then at some point the data from the response is there. The operating system handler that monitors this data then triggers a signal, that calls *continuation.run()*. And this call to *run()* takes the stack of your virtual thread from the heap memory

where it has been saved previously, and puts it in the **wait list of the platform thread** it was mounted on in this fork join pool.

- Because this is the way the fork join pool is working, if this thread is busy, and another thread is available, then this other thread will steal this task from the first thread.

- You may observe that your task actually jumped from one platform thread to the other while it was blocked. This is how virtual threads are working under the hood. As you can see: no magic, just very smart code, actually doing wonders.





- Running blocking code in a virtual thread

- The cost of running a task in a virtual thread is the cost of running it in a platform thread, plus an overhead. So it is more expensive to run a task in a virtual thread than running it in a platform thread. But the cost of blocking a virtual thread is tremendously lower than blocking a platform thread.
- The cost of blocking a virtual thread is the cost of moving the stack of your virtual thread, that is several tens of kilobytes, from the stack to the heap memory and back.
- Running a task in a virtual thread is interesting only if it is a blocking task, because there you will have this gain.
- Running a task that is doing only in-memory computations in a virtual thread is useless, don't do it, this is not why virtual threads have been made.

- Don't run your parallel streams in virtual threads, you'll actually degrade your performances. You are not doing I/O in your parallel streams.

- If you check how the JVM is working, you will see that the Just In Time compiler is still running in a platform thread, and the same goes for the Garbage collectors for instance. These are in-memory computations, they do not run in virtual thread.

- The caveat you need to know has to do with native code calls and synchronization.

- Handling native code and synchronized blocks: avoid pinning virtual threads

- Moving this stack around is ok as long as your code is not playing with addresses on this stack. You cannot get an address on the stack in Java, but you can in C or C++. And your Java code may call some C code or some C++ code. So if the JVM detects that you have native calls in the stack of your virtual thread, it will pin it to its platform thread to prevent it from being moved to somewhere else.

- A problem you may have heard before is that the synchronized block behaves in that way too. If you have a synchronized block in your stack, then your virtual thread is pinned on your platform thread. What can you do if you are in this situation?. Your first reflex should be to do nothing, and assess your code precisely.

- A performance problem may occur if you pin a virtual thread on a platform for a long time. If your native code, or your synchronized block is only doing some in-memory computations that will take in the order of the hundreds of nanoseconds to execute, then odds are that you will not see any performance loss. It's the case if your synchronized block is guarding some in-memory data structure to ensure its correctness, safety and visibility.

- On the other hand, if you have some I/O operations in this synchronized block that can pin your thread for hundreds of milliseconds, then maybe it will be a good idea to do something about it.

- The good news is that you can replace your synchronized block with a ***reentrant lock***, that will do the same without pinning your virtual thread. But remember, you don't have to do that, there are still many situations where pinning a virtual thread on its platform thread will not affect your performances. As usual when it comes to performances: measure, don't guess.

```
synchronized(key) {
   // do something
}
```

```
Lock lock = new ReentrantLock();
try {
   lock.lock();
 // do something
} finally {
   lock.unlock();
}
```

- Wrapping up: virtual threads are cheap, blocking them is fine

- Virtual threads are a new kind of threads in the Java space.
- Virtual threads are:
  - Cheap to create: you can have million of them.
  - Cheap to block: you do not need to rely on non-blocking asynchronous code anymore, running some blocking code in a virtual thread is just fine.
- You do not need to pool them. Create them on demand, and let them die at the end, this is what the JDK is doing.
- You should only run blocking, typically I/O code, in a virtual thread.
- Running in-memory computations is useless, don't do that.
- When you run some native code or when you do some synchronization, your virtual thread is pinned on its platform thread. That may be an issue if you are doing I/O operations in your synchronized block.
- Odds that you will not work directly with virtual threads in your application. But don't worry, most of the great framework you know of are already supporting, or working on supporting virtual threads, including Jetty, Vertx, Tomcat, Spring, Quarkus, Spring Boot, or Helidon.