ORDINA
a Sopra Steria company

RICK HUTTEN

SOFTWARE ENGINEER @ ORDINA

Simplifying Concurrency with Project Loom:
A Journey with Pizza-chef Fabio

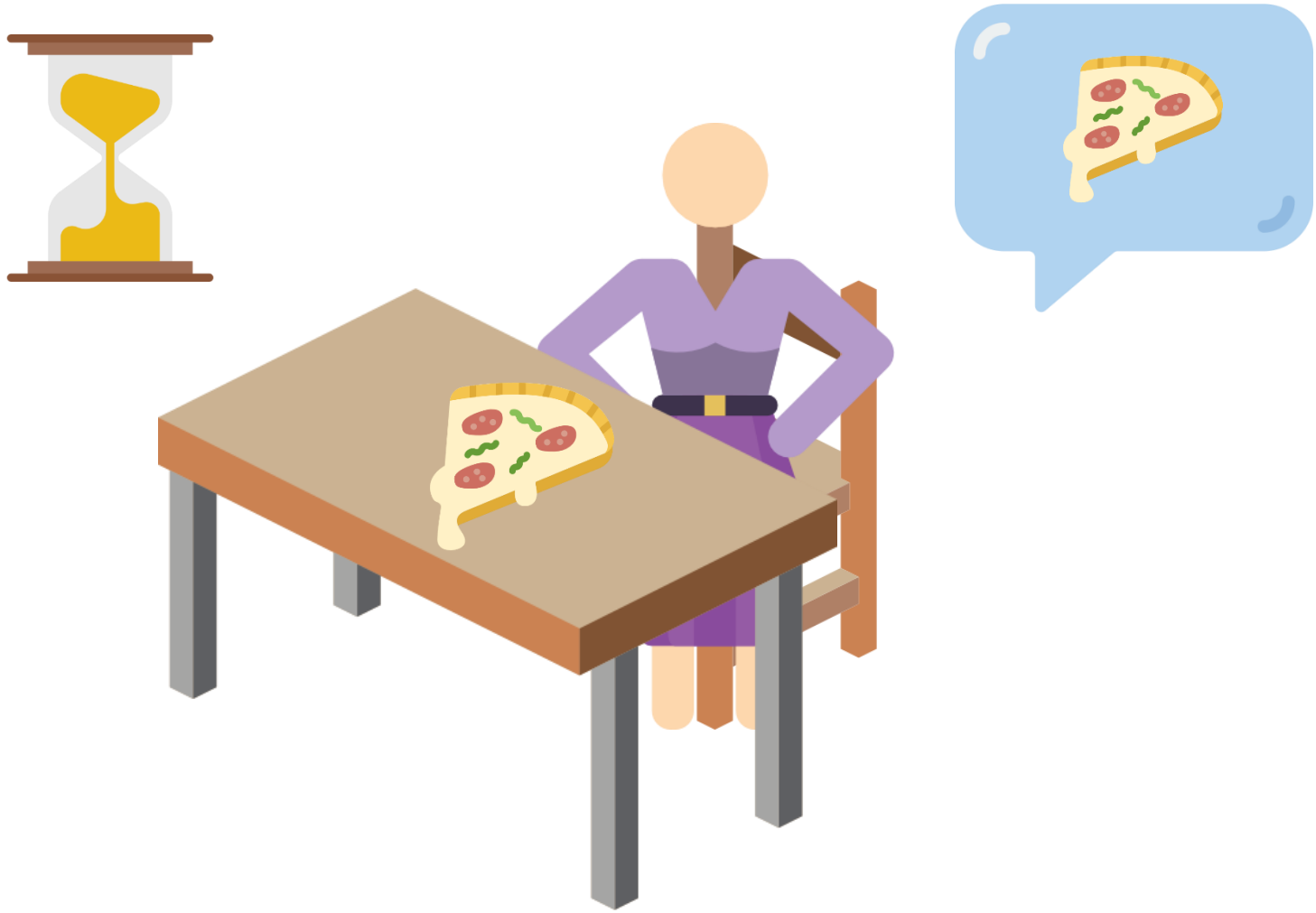
PROJECT LOOM

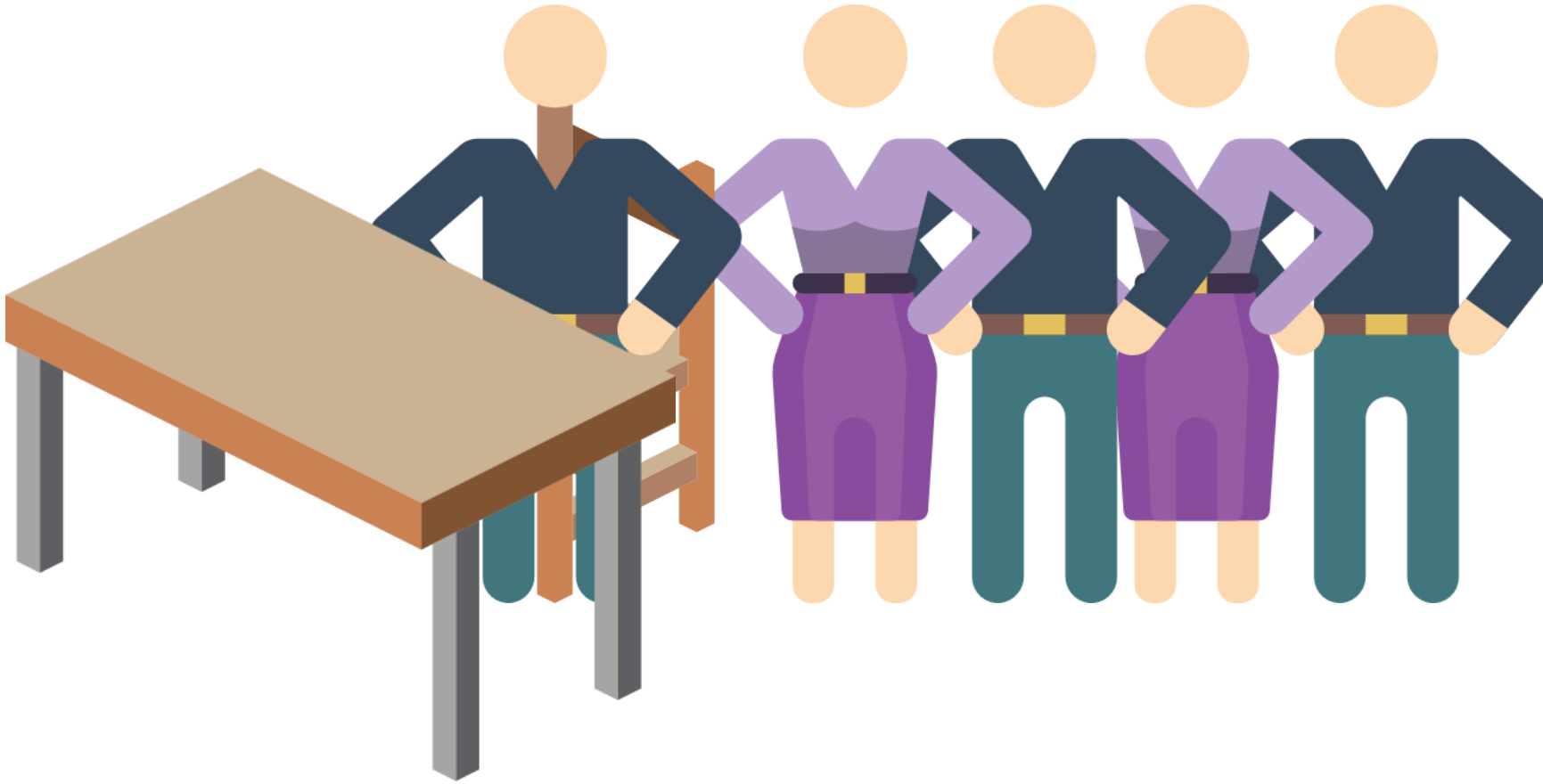
JDK 21

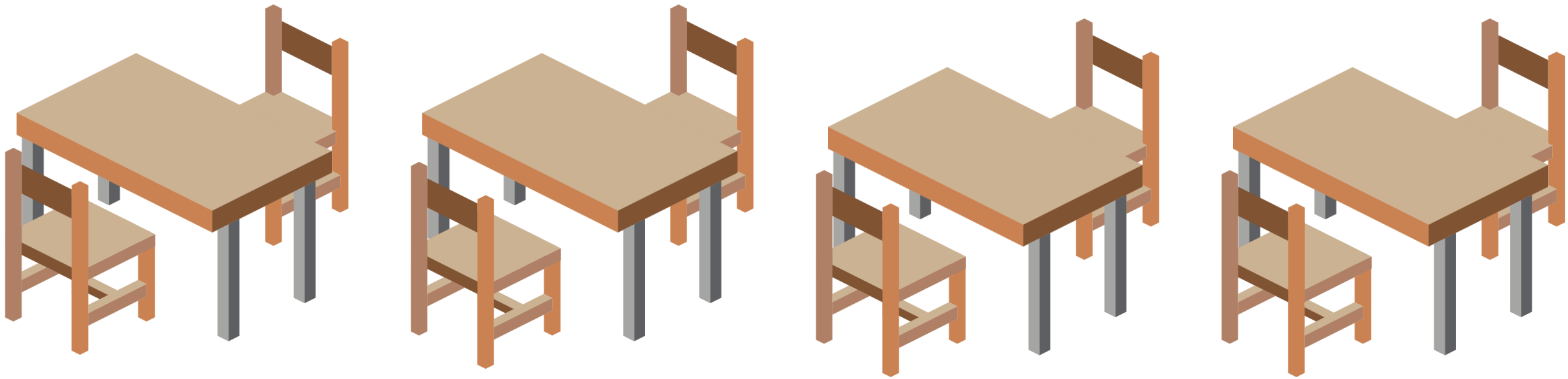
- **JEP 444: Virtual Threads**
- **JEP 453: Structured Concurrency (Preview)**

JDK 22

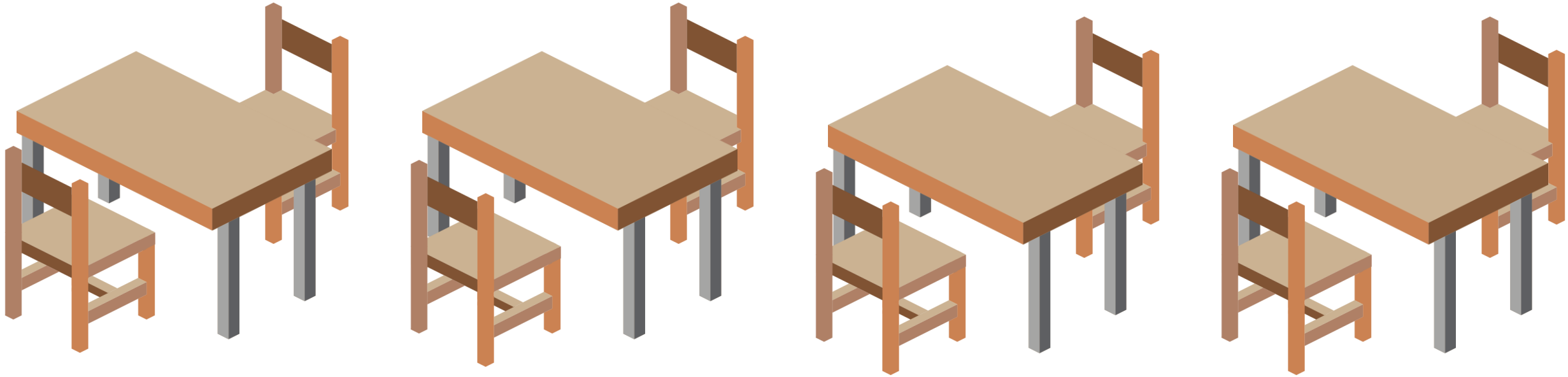
- **JEP 462: Structured Concurrency (Second Preview)**



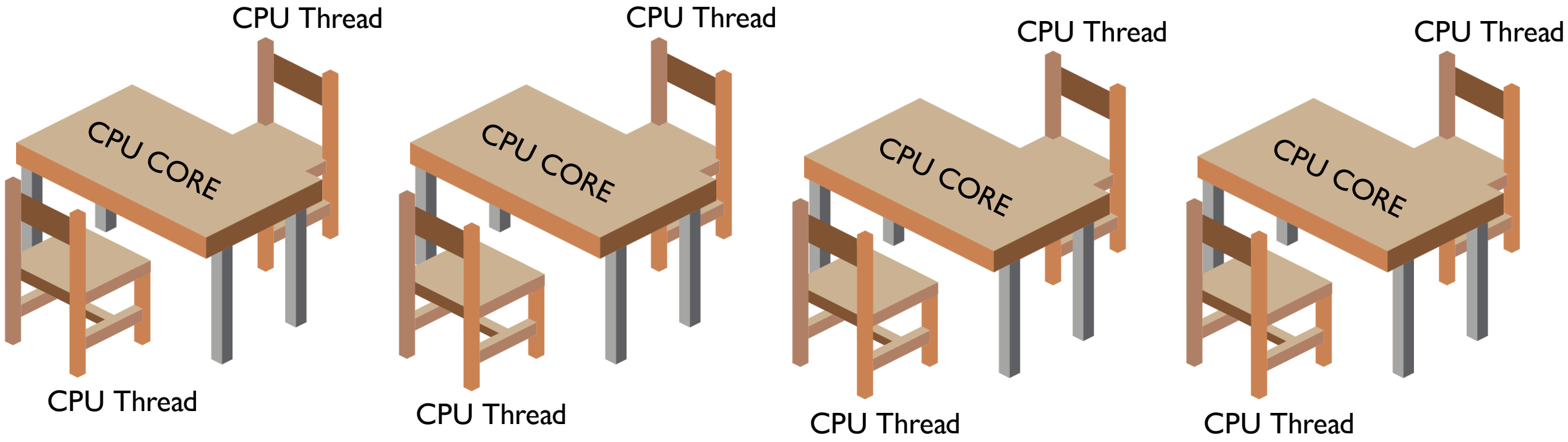




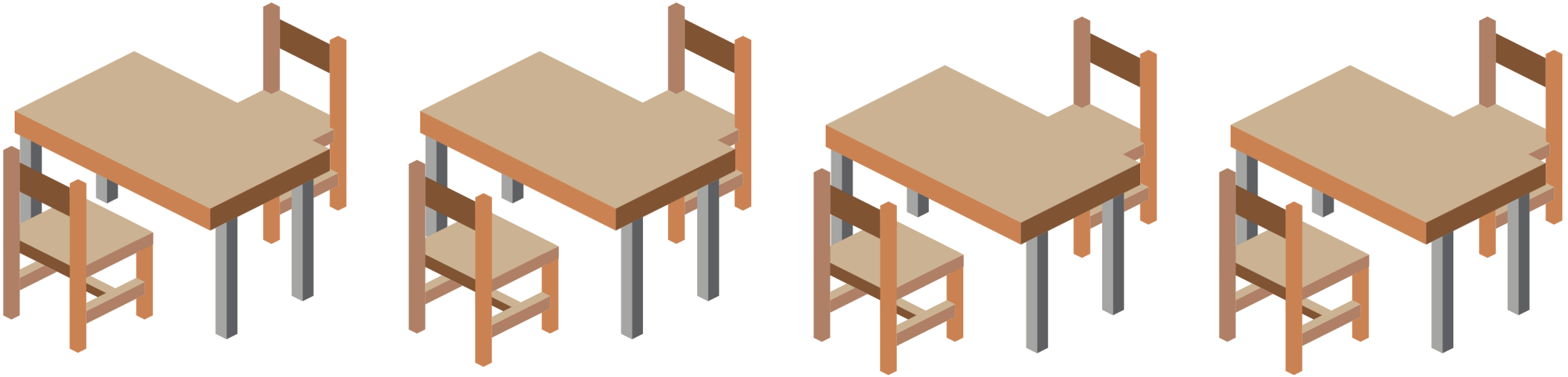
4 TABLES
8 CHAIRS
8 CUSTOMERS

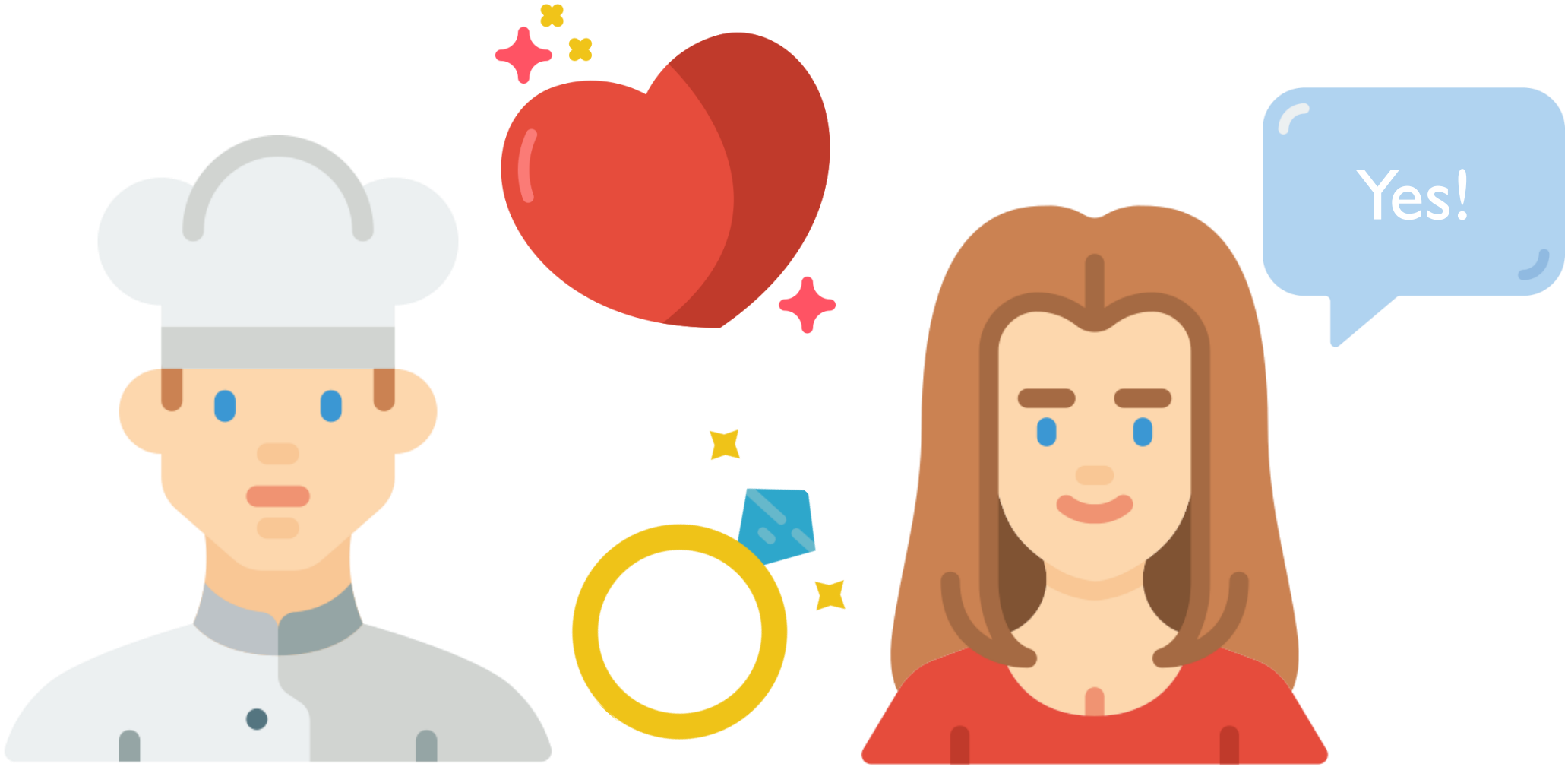


4 TABLES → 4 CORES
8 CHAIRS → 8 CPU THREADS
8 CUSTOMERS → 8 TASKS



MULTITHREADING




Yes!

HOW TO MAKE MORE MONEY

HOW TO MAKE MORE MONEY

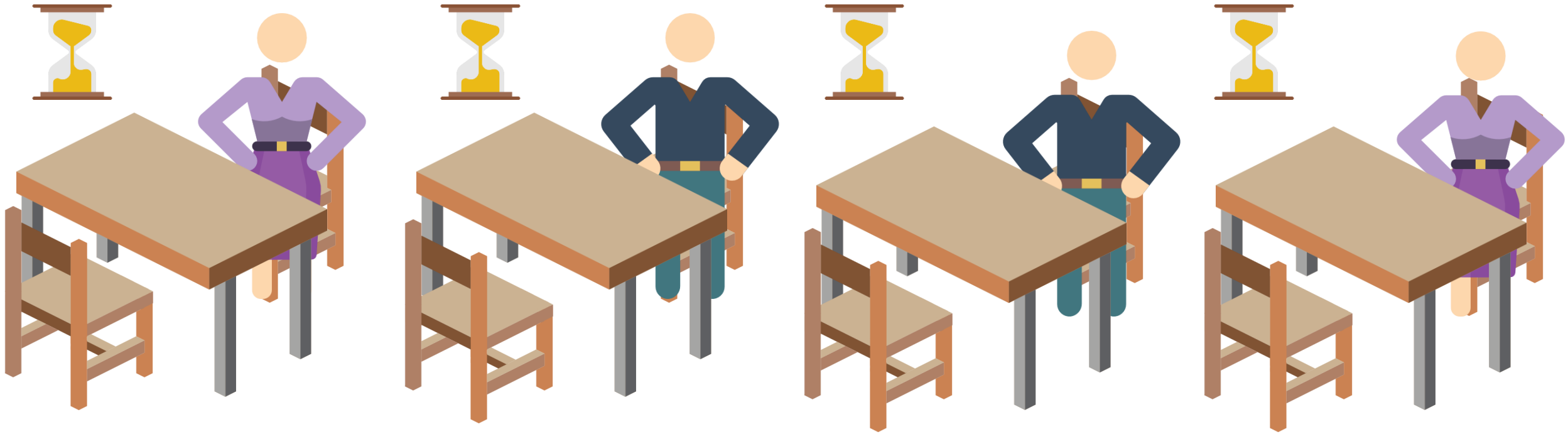
1. Need to serve more people!

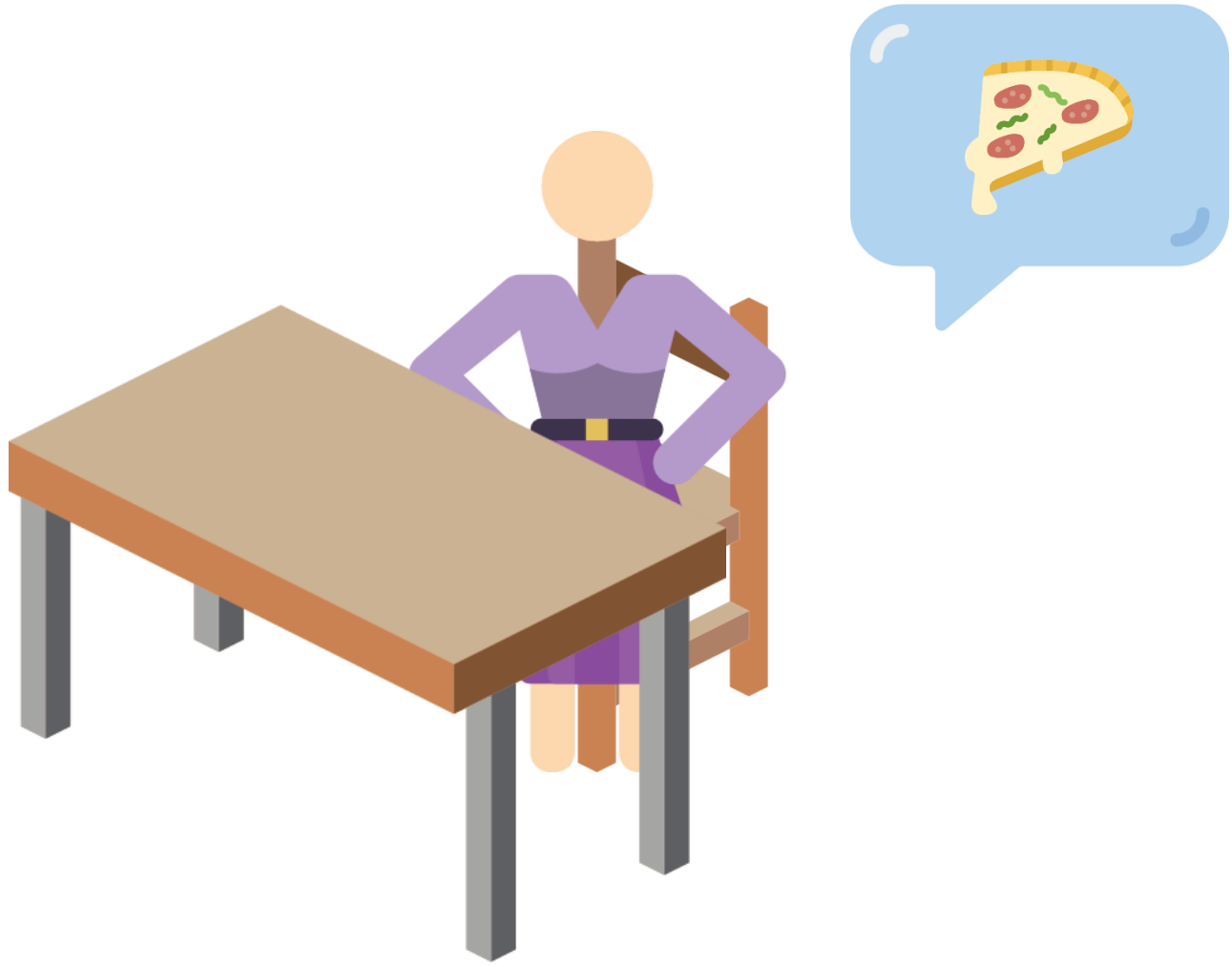
PROCESS

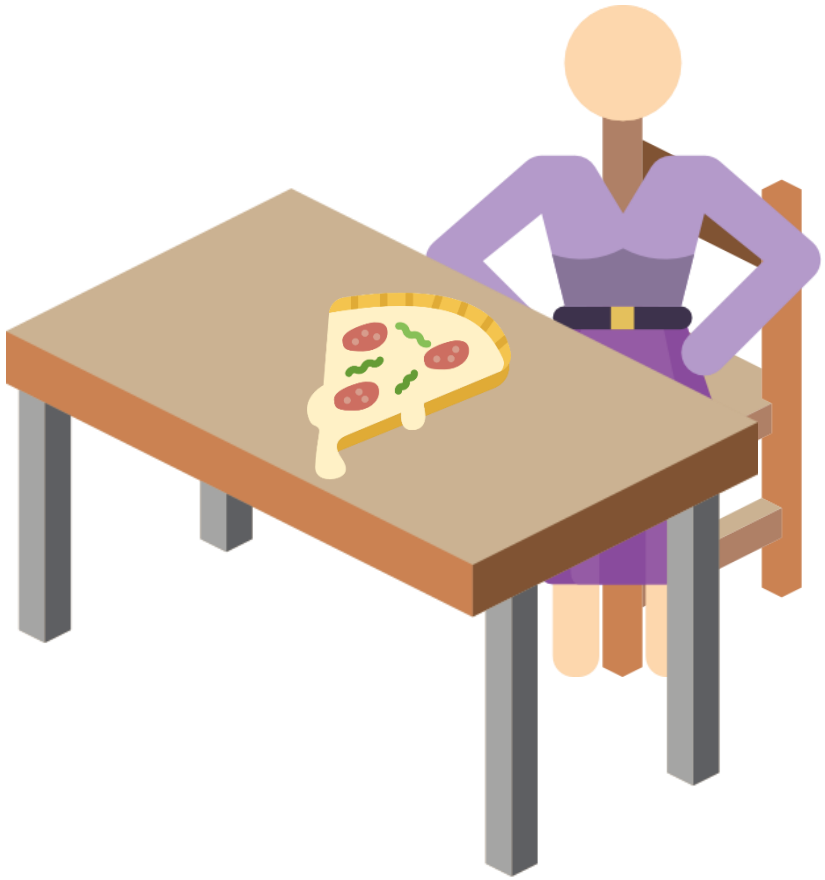
• Customer comes in and orders a pizza	1 minute
• Customer waits for the chef to make the pizza	30 minutes
• The customer eats the pizza	10 minutes

PROCESS

• Customer comes in and orders a pizza	1 minute
• Customer waits for the chef to make the pizza	30 minutes
• The customer eats the pizza	10 minutes







SWITCHING CUSTOMERS

• Customer comes in and orders a pizza	1 minute
• Customer waits for the chef to make the pizza	30 minutes
• The customer eats the pizza	10 minutes

SWITCHING CUSTOMERS

• Customer comes in	1 minute
• Customer orders a pizza	1 minute
• Customer leaves the restaurant	1 minute
• Customer waits for the chef to make the pizza	30 minutes
• Customer comes back into the restaurant	1 minute
• The customer eats the pizza	10 minutes

HOW MANY THREADS

In the limit for network I/O:

- 10-100 ns preparing request and processing results
- 10-100 ms waiting for response

When a task runs on a thread, it either:

- Finishes
- An exception is thrown

HOW MANY THREADS

In the limit for network I/O:

- 10-100 ns preparing request and processing results
- 10-100 ms waiting for response

When a task runs on a thread, it either:

- Finishes
- An exception is thrown

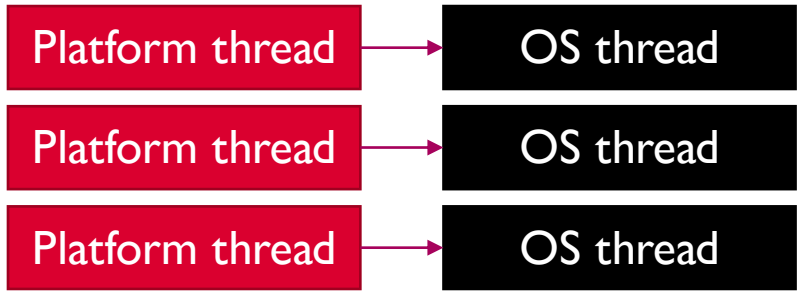
Need a million threads for 100% CPU usage

THREADS ARE EXPENSIVE

- Creating threads is expensive → 1 ms + 2 MB
- Context switching is expensive → 0.1 ms
- Introduction of: **VIRTUAL THREADS**
  - Thread pools e.g. ExecutorService
  - Async programming e.g. CompletableFuture

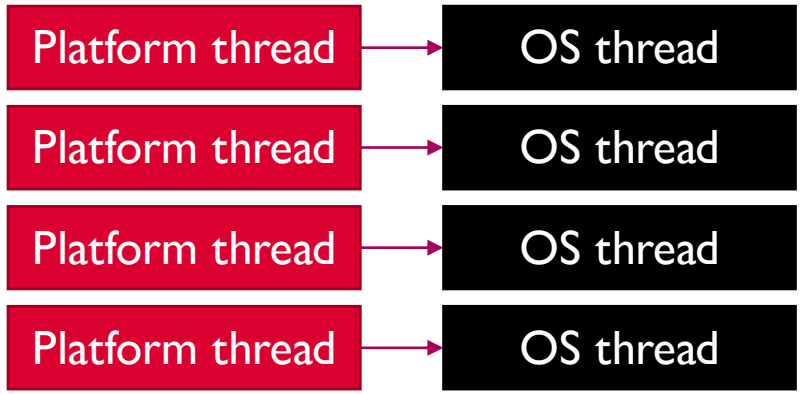
VIRTUAL THREADS

Platform threads

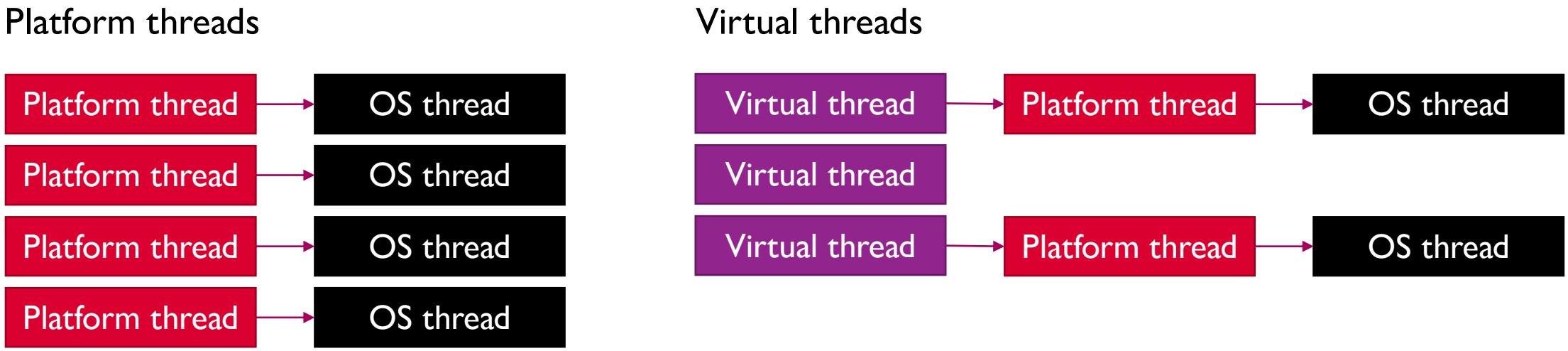


VIRTUAL THREADS

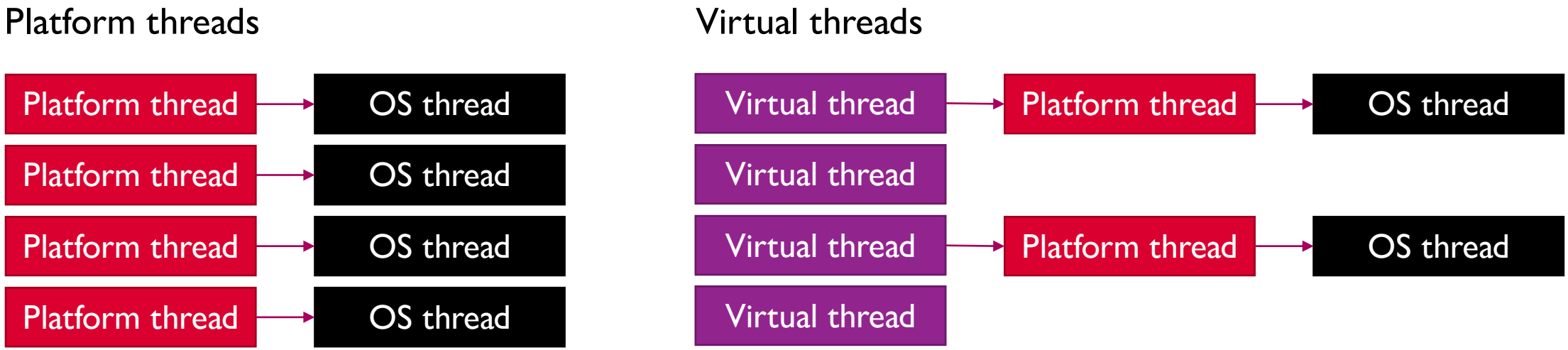
Platform threads



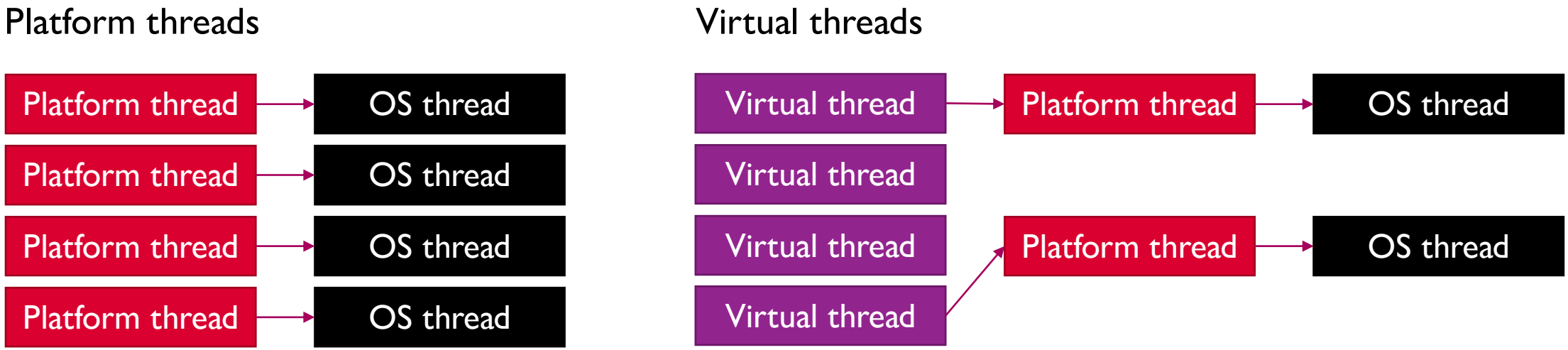
VIRTUAL THREADS



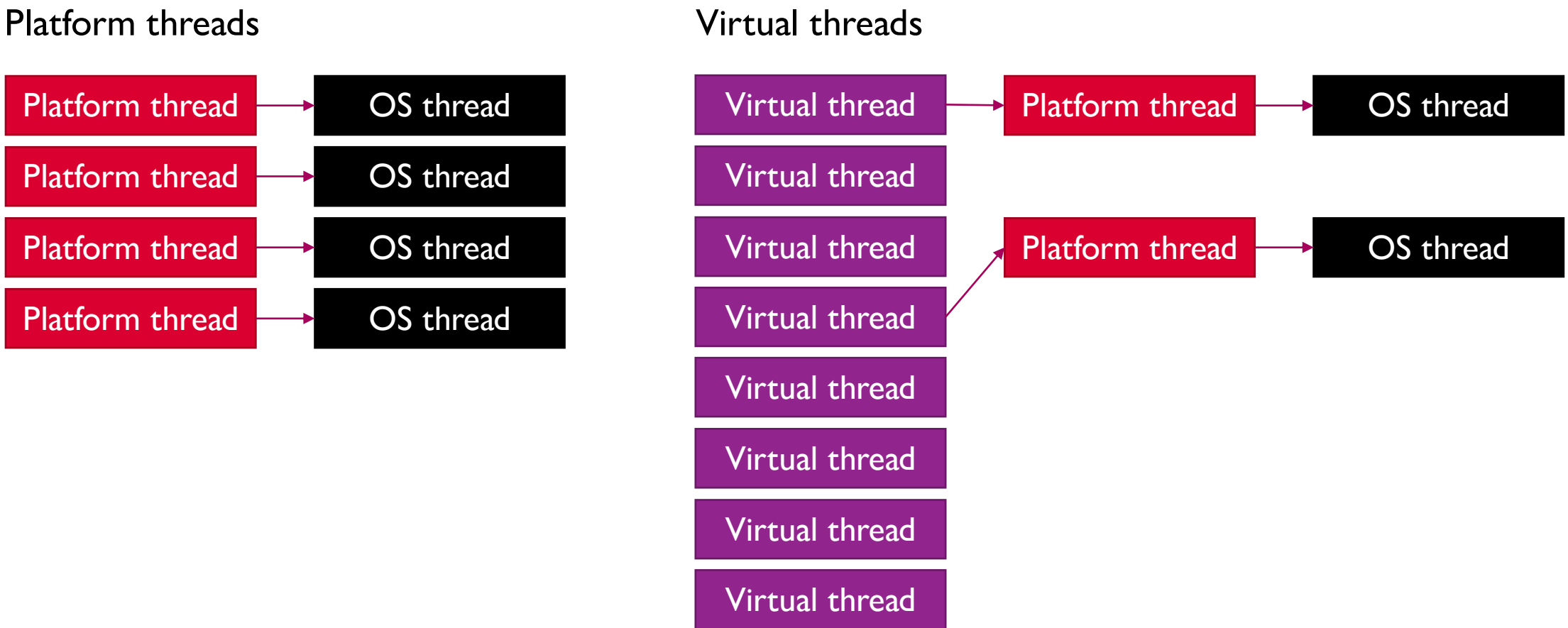
VIRTUAL THREADS



VIRTUAL THREADS



VIRTUAL THREADS



VIRTUAL THREADS

- Layer on top of platform threads, subclass of java.lang.Thread
- Creating virtual threads and context switching is faster and uses less memory

VIRTUAL THREADS

- Usage is similar to java.lang.Thread

```
var thread = new Thread(this::doTask,  name: "thread");
thread.start();
thread.join();

Thread.ofPlatform() Thread.Builder.OfPlatform
        .name("platform")
        .start(this::doTask) Thread
        .join();

Thread.ofVirtual() Thread.Builder.OfVirtual
        .name("virtual")
        .start(this::doTask) Thread
        .join();
```

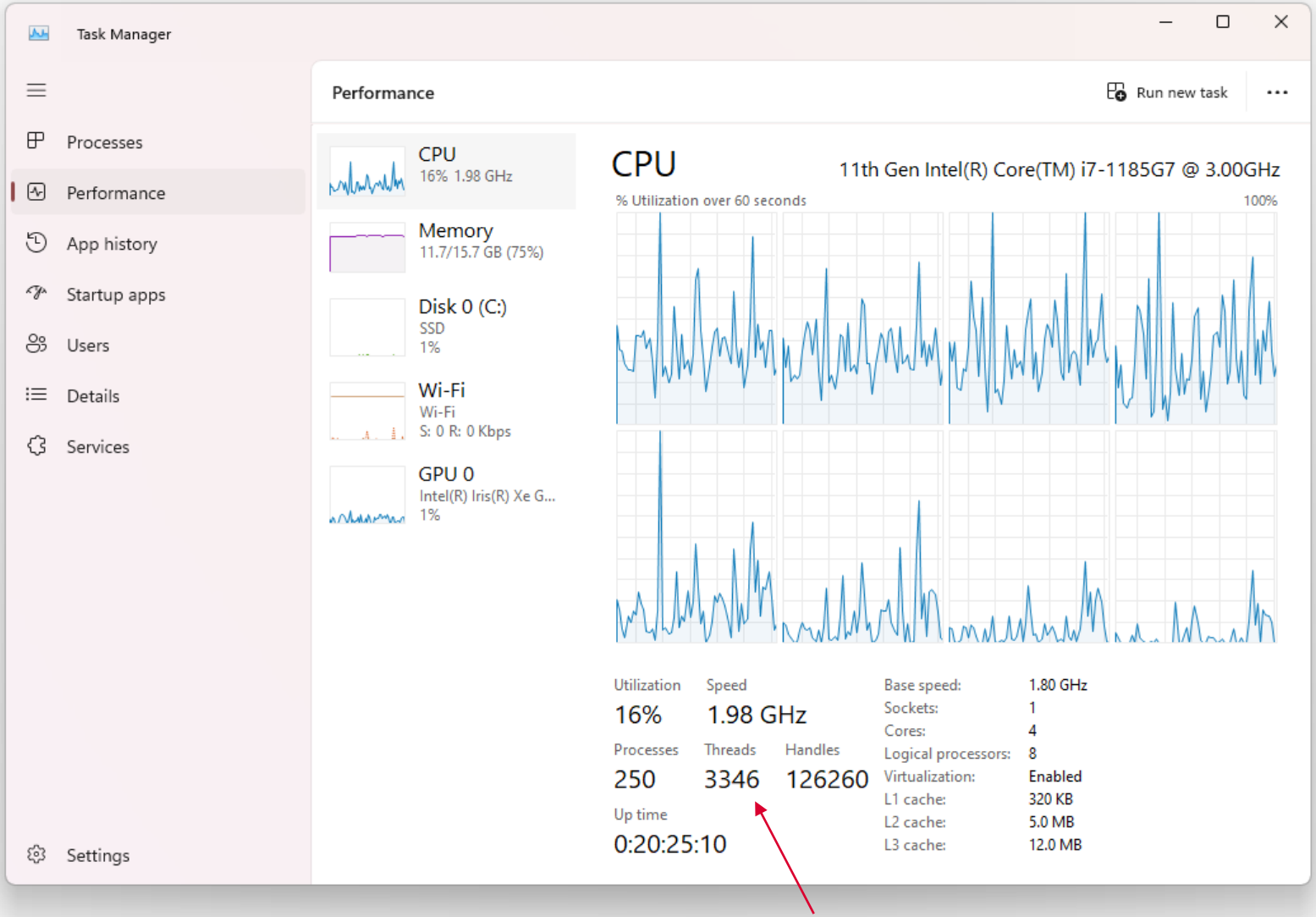
VIRTUAL THREADS

- Usage is similar to java.lang.Thread
- Replacing platform threads with virtual threads **should** work *

* famous last words

BLOCKING OPERATIONS

- Platform threads are blocked
    - Scheduled by OS


Task Manager
Processes
Performance
App history
Startup apps
Users
Details
Services
Settings
Performance
Run new task
CPU
16% 1.98 GHz
Memory
11.7/15.7 GB (75%)
Disk 0 (C:)
SSD
1%
Wi-Fi
Wi-Fi
S: 0 R: 0 Kbps
GPU 0
Intel(R) Iris(R) Xe G...
1%
CPU
11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz
% Utilization over 60 seconds
100%
Utilization
16%
Speed
1.98 GHz
Processes
250
Threads
3346
Handles
126260
Up time
0:20:25:10
Base speed: 1.80 GHz
Sockets: 1
Cores: 4
Logical processors: 8
Virtualization: Enabled
L1 cache: 320 KB
L2 cache: 5.0 MB
L3 cache: 12.0 MB

BLOCKING OPERATIONS

- Platform threads are blocked
  - Scheduled by OS
  - Many blocking calls → many OS threads → wasted resources

BLOCKING OPERATIONS

- Platform threads are blocked
  - Scheduled by OS
  - Many blocking calls → many OS threads → wasted resources
- Virtual theads run on platform thread
  - On blocking call:
    - Park itself for a duration
    - Inform JVM to unmount virtual thread
    - Yield (pause) current operation
  - Runs tasks in a Continuation

CONTINUATIONS

```python
def continuation():
    print("Doing work part 1/2")
    yield
    print("Doing work part 2/2")
    yield


con = continuation()
print("Let's start")
next(con)
print("And the next one")
next(con)
print("Finished")
```



```
Let's start
Doing work part 1/2
And the next one
Doing work part 2/2
Finished
```

```java
private static void sleepNanos(long nanos) throws InterruptedException {
    ThreadSleepEvent event = beforeSleep(nanos);
    try {
        if (currentThread() instanceof VirtualThread vthread) {
            vthread.sleepNanos(nanos);
        } else {
            sleepNanos0(nanos);
        }
    } finally {
        afterSleep(event);
    }
}
```

```java
void sleepNanos(long nanos) throws InterruptedException {  1 usage
    assert Thread.currentThread() == this && nanos >= 0;
    if (getAndClearInterrupt())
        throw new InterruptedException();
    if (nanos == 0) {
        tryYield();
    } else {
        // park for the sleep time
        try {
            long remainingNanos = nanos;
            long startNanos = System.nanoTime();
            while (remainingNanos > 0) {
                parkNanos(remainingNanos);
                if (getAndClearInterrupt()) {
                    throw new InterruptedException();
                }
                remainingNanos = nanos - (System.nanoTime() - startNanos);
            }
        } finally {
            // may have been unparked while sleeping
            setParkPermit(true);
        }
    }
}
```

```
void parkNanos(long nanos) {
    assert Thread.currentThread() == this;

    // complete immediately if parking permit available or interrupted
    if (getAndSetParkPermit( newValue: false) || interrupted)
        return;

    // park the thread for the waiting time
    if (nanos > 0) {
        long startTime = System.nanoTime();

        boolean yielded = false;
        Future<?> unparker = scheduleUnpark(nanos);  // may throw OOME
        setState(TIMED_PARKING);
        try {
            yielded = yieldContinuation();  // may throw
        } finally {
            assert (Thread.currentThread() == this) && (yielded == (state() == RUNNING));
            if (!yielded) {
                assert state() == TIMED_PARKING;
                setState(RUNNING);
            }
            cancel(unparker);
        }

        // park on carrier thread for remaining time when pinned
        if (!yielded) {
            long remainingNanos = nanos - (System.nanoTime() - startTime);
            parkOnCarrierThread( timed: true, remainingNanos);
        }
    }
}
```

```java
private boolean yieldContinuation() {
    notifyJvmtiUnmount(/*hide*/true);
    try {
        return Continuation.yield(VTHREAD_SCOPE);
    } finally {
        notifyJvmtiMount(/*hide*/false);
    }
}
```

VIRTUAL THREADS

- Used for blocking calls
- Yield operation on blocking calls to allow other virtual threads to run on CPU
- Copy the stack to the heap and vice versa

VIRTUAL THREADS

- How do we create virtual threads?

```
var virtualThread1 : Thread = Thread.ofVirtual().start(this::doTask);
var virtualThread2 : Thread = Thread.ofVirtual().start(this::doTask);

virtualThread1.join();
virtualThread2.join();
```

STRUCTURED CONCURRENCY

- A new way to organise virtual threads
- Create a scope (StructuredTaskScope) in which to create virtual threads
- All threads are bound to this scope, exiting scope will stop threads

STRUCTURED CONCURRENCY

StructuredTaskScope

```
try(var scope = new StructuredTaskScope<Type>()) {
    // Create threads
    var subtask1 :Subtask<Type>  = scope.fork(this::doTask);
    var subtask2 :Subtask<Type>  = scope.fork(this::doTask);

    // Join all threads
    scope.join();

    // Get the results
    var result1 :Type  = subtask1.get();
    var result2 :Type  = subtask2.get();
}
```

STRUCTURED CONCURRENCY

StructuredTaskScope.ShutdownOnSuccess

```
try(var scope = new StructuredTaskScope.ShutdownOnSuccess<Type>()) {
    scope.fork(this::doTask);
    scope.fork(this::doTask);

    scope.join();
    var result :Type = scope.result();
}
```

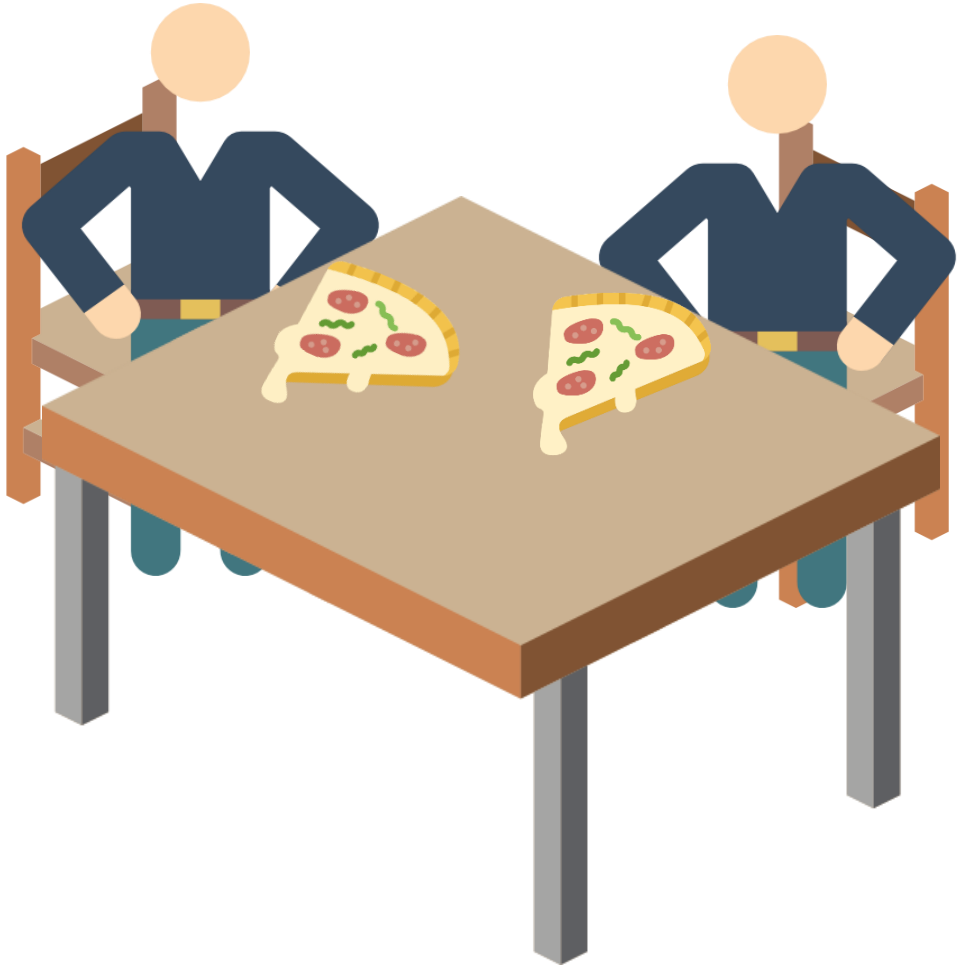
STRUCTURED CONCURRENCY

StructuredTaskScope.ShutdownOnFailure

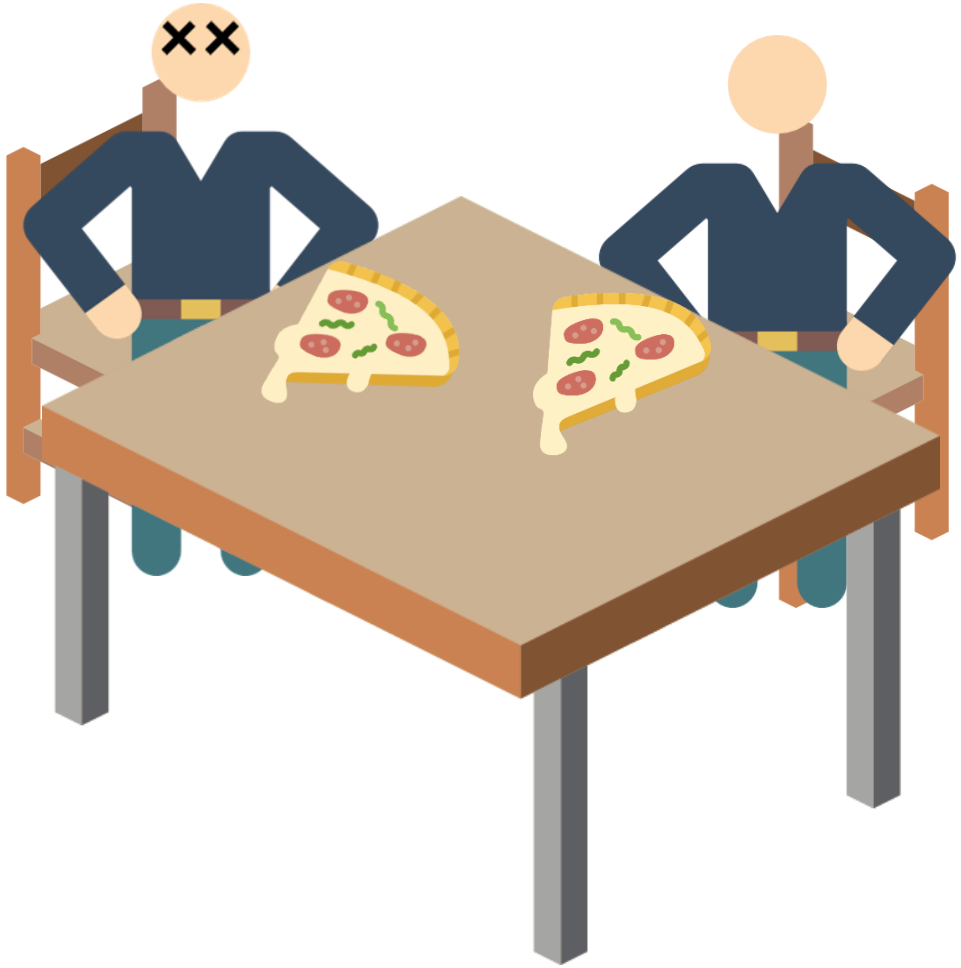
```
try(var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    var subtask1 : Subtask<Type>  = scope.fork(this::doTask);
    var subtask2 : Subtask<Type>  = scope.fork(this::doTask);

    scope.join();
    var error : Optional<Throwable>  = scope.exception();
    var result1 : Type  = subtask1.get();
    var result2 : Type  = subtask2.get();
}
```

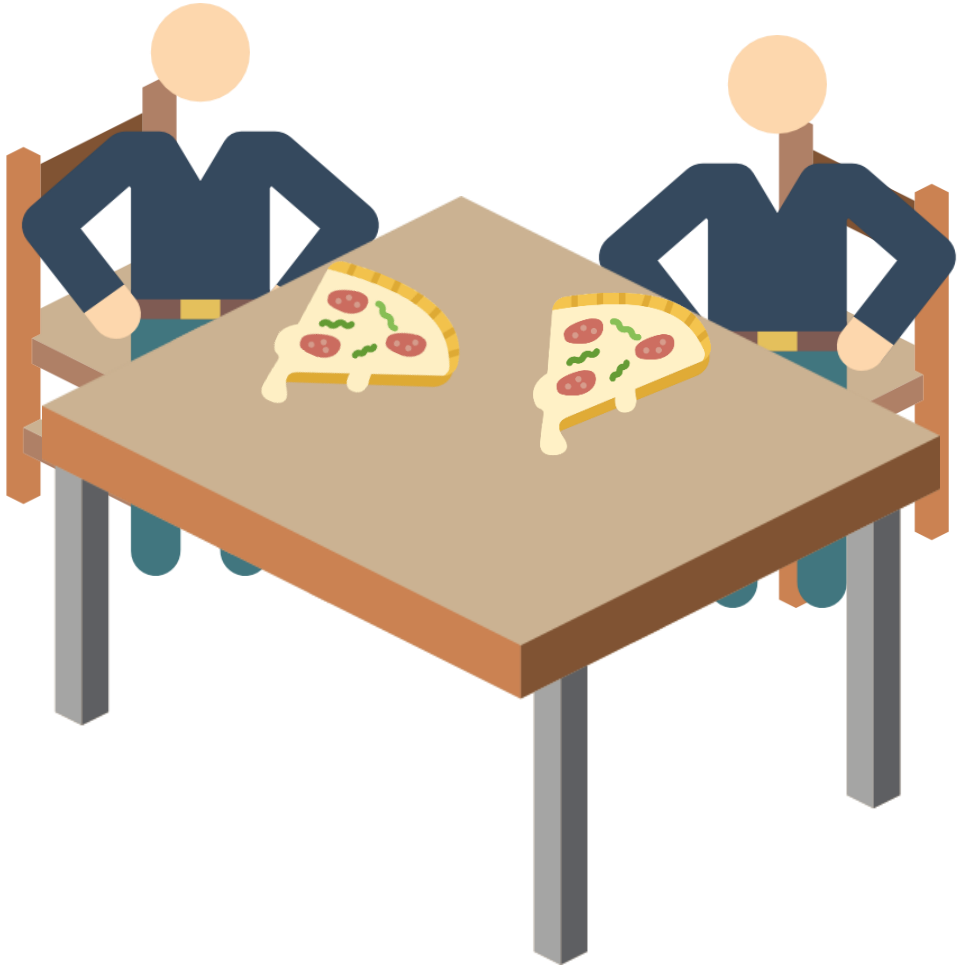
OLD SITUATION



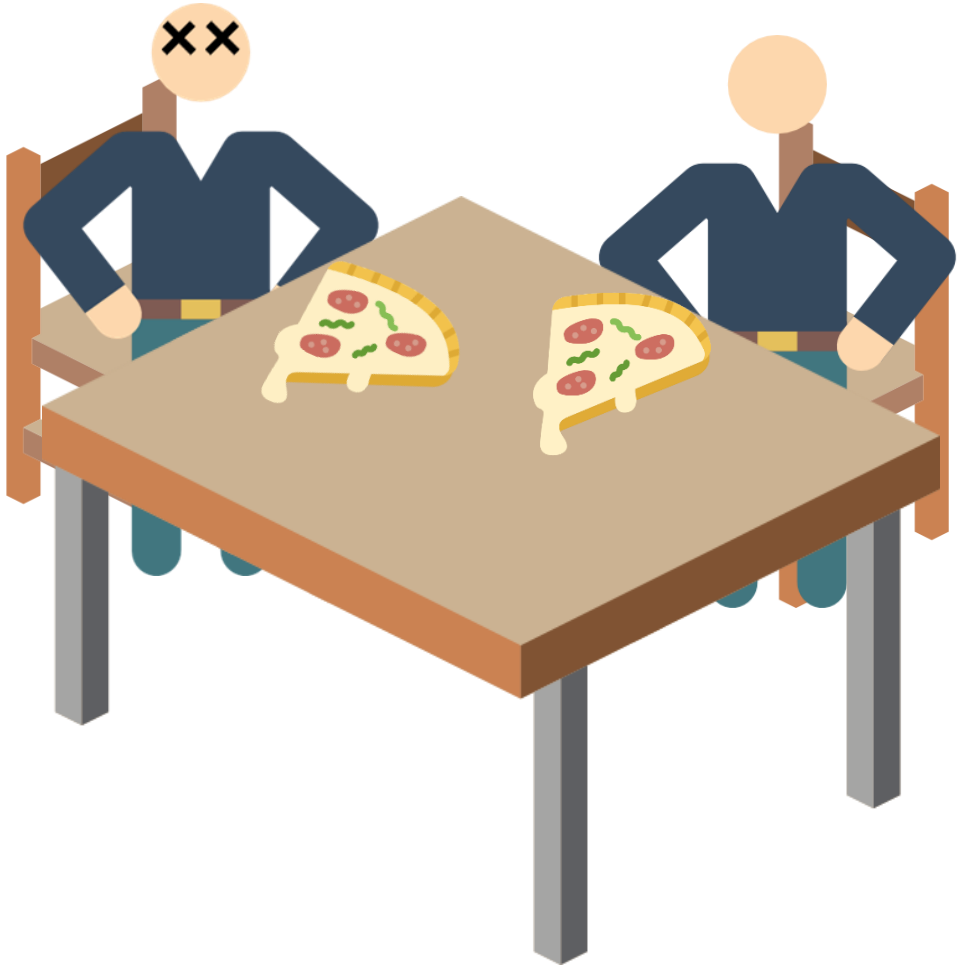
OLD SITUATION



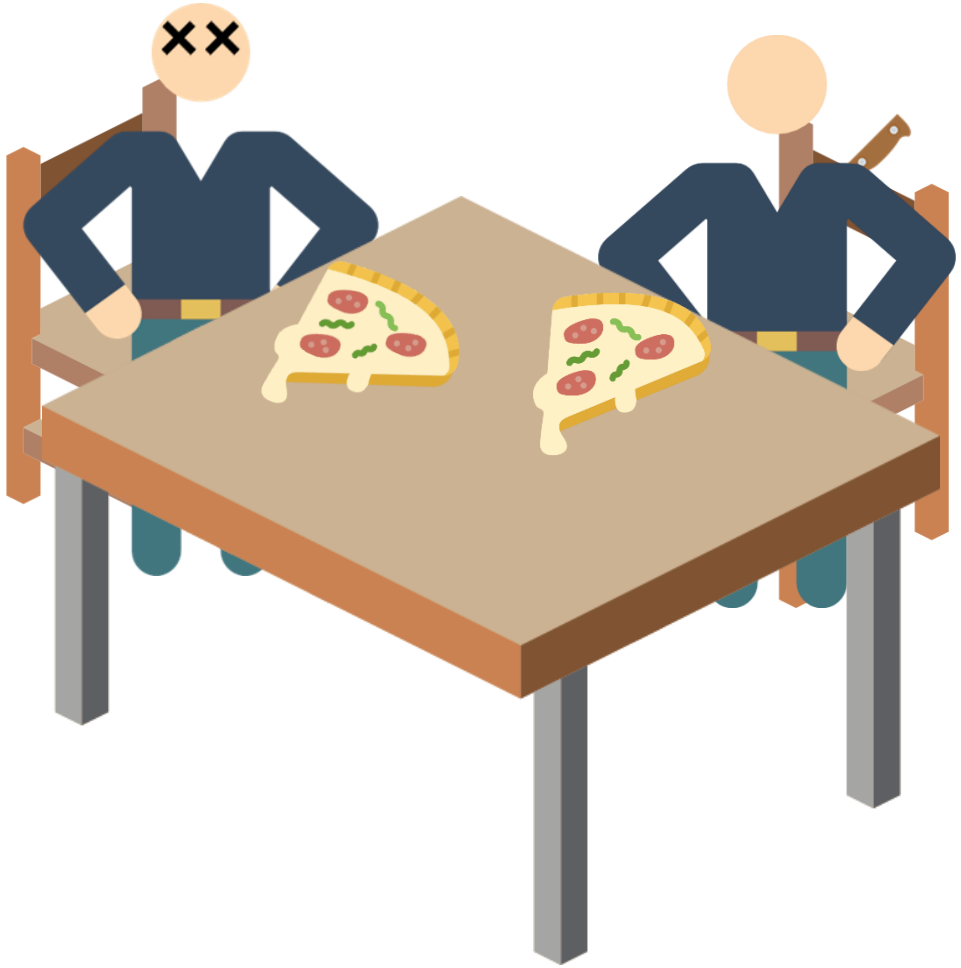
NEW SITUATION



NEW SITUATION



NEW SITUATION



STRUCTURED CONCURRENCY

StructuredTaskScope

- Meant to be extended

```java
public class AnyPizzaScope extends StructuredTaskScope<Pizza> {
    private final AtomicReference<Pizza> pizzaAtomic = new AtomicReference<>();  2 usages

    @Override  5 usages
    protected void handleComplete(Subtask<? extends Pizza> subtask) {
        // This function is run in virtual thread, be wary of race conditions
        switch (subtask.state()) {
            case UNAVAILABLE -> { /* Do nothing, thread cancelled */ }
            case SUCCESS -> onFirstPizza(subtask.get());
            case FAILED -> { /* Do nothing, couldn't get pizza */ }
        }
    }

    private void onFirstPizza(Pizza pizza) {  1 usage
        pizzaAtomic.set(pizza);
        this.shutdown(); // Shutdown all other pizza orders
    }

    public Optional<Pizza> getPizza() {  1 usage
        return Optional.ofNullable(pizzaAtomic.get());
    }
}
```

```java
try(var pizzaScope = new AnyPizzaScope()) {
    pizzaScope.fork(this::orderSalami);
    pizzaScope.fork(this::orderQuattroFormaggi);
    pizzaScope.fork(this::orderCapricciosa);

    pizzaScope.join();

    var pizza : Optional<Pizza> = pizzaScope.getPizza();
}
```

STRUCTURED CONCURRENCY

StructuredTaskScope

- Meant to be extended
- Can be created everywhere, unlike ExecutorService

VIRTUAL THREADS

Should we replace all threads with virtual threads?

VIRTUAL THREADS

Should we replace all threads with virtual threads?

- No, there are downsides

VIRTUAL THREADS DOWNSIDES

- Copy the stack to the heap when they unmount
- When the stack is copied back from heap:
  - Different location on stack → pointers can break

VIRTUAL THREADS DOWNSIDES

Virtual threads can't unmount (pinning) when:

- Code is executing inside 'synchronize' blocks → use locks instead
- It has a native method or foreign function in its stack
- No blocking operations are made

VIRTUAL THREADS DOWNSIDES

ThreadLocal has two common use cases:

1. Object pool for expensive resources
   - Leads to huge memory footprint in virtual threads
2. Storage exclusive for thread
   - Not immutable → ScopedValues (In preview as of JDK 22)

IN SUMMARY

1. Virtual threads should be used for blocking operations while not wasting many resources
2. Virtual threads are not faster than platform threads, performance is slightly lower. For CPU tasks, keep using platform threads with e.g. ExecutorService
3. Use StructuredTaskScope to coordinate multiple virtual threads like a single unit of work

… and Fabio?

… and they lived happily every after

ORDINA
a Sopra Steria company

THANK YOU

RICK HUTTEN

SOFTWARE ENGINEER @ ORDINA

LINKEDIN.COM/IN/RICK-HUTTEN-56182312A/