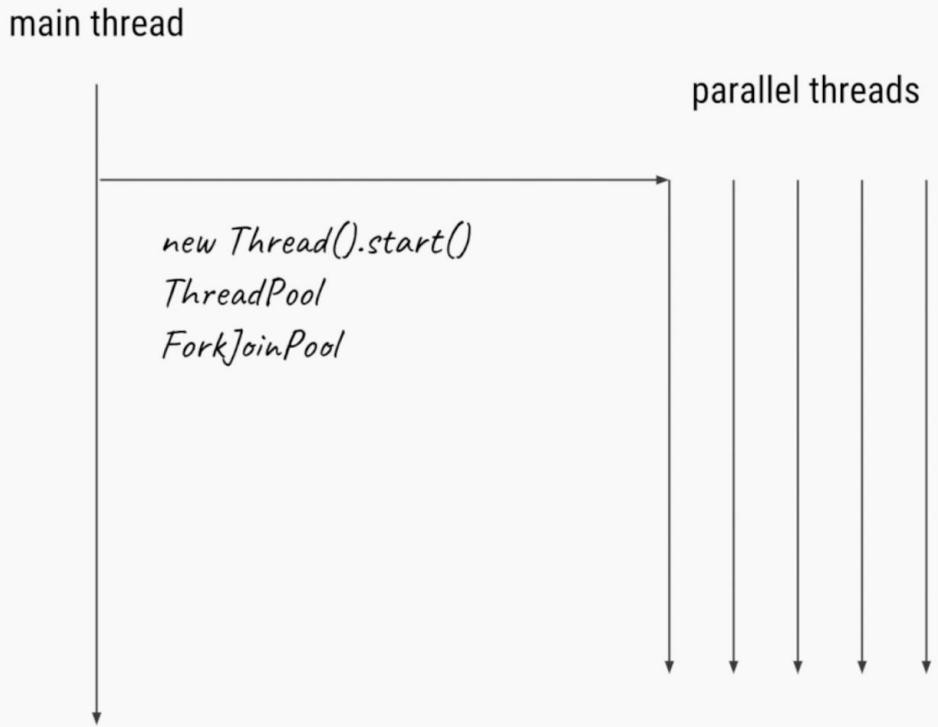
Java - Async Programming

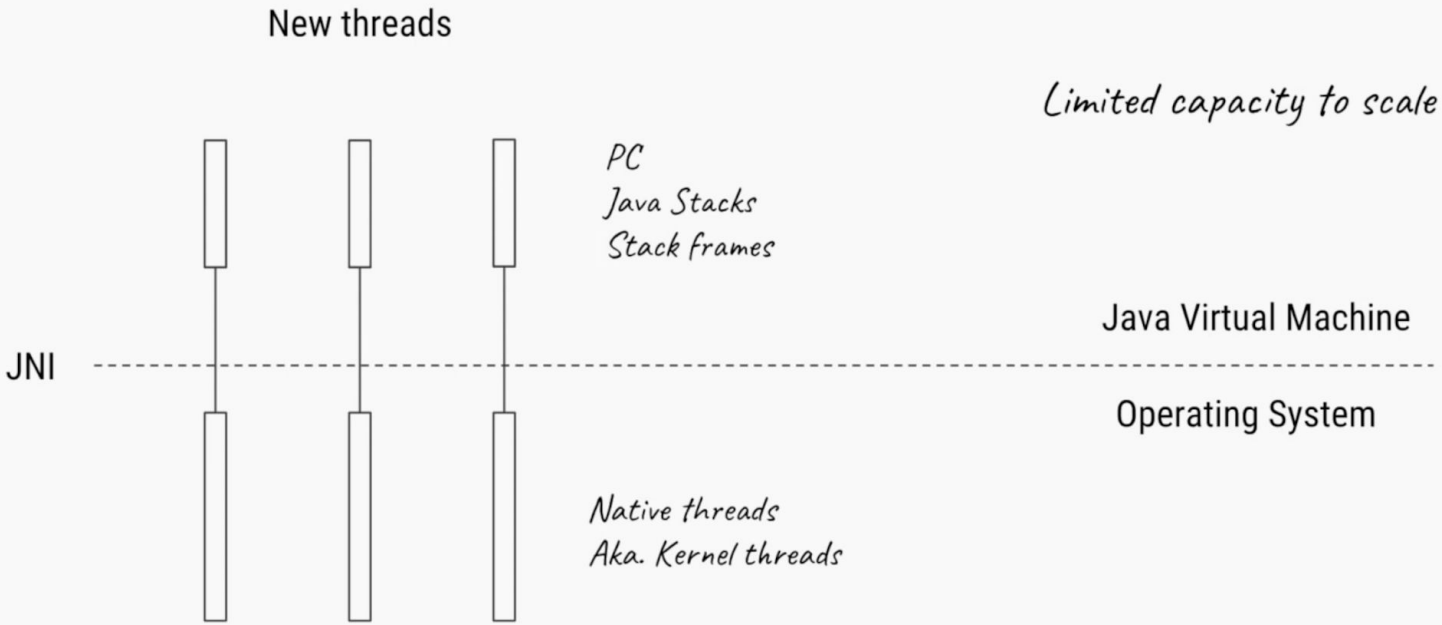
Need for speed



Core 1	Core 2
Core 3	Core 4

CPU

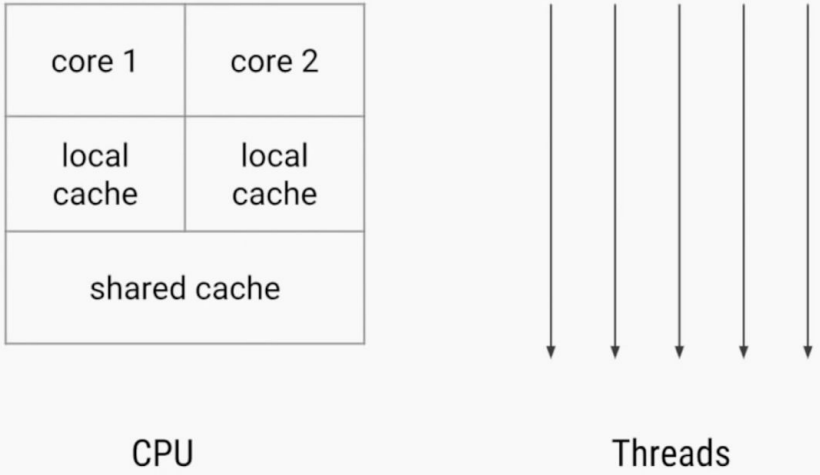
Java thread = OS thread



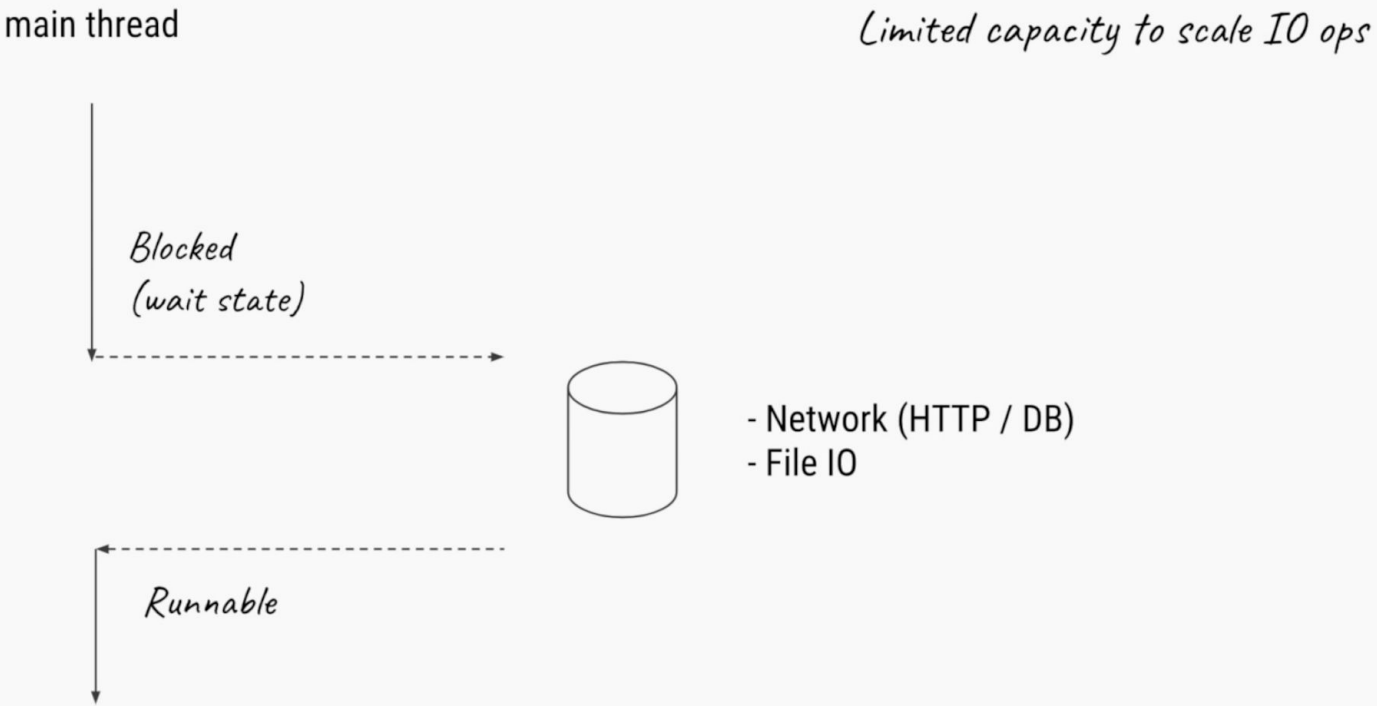
Other issues too...

Scheduling Overhead,

Data Locality



IO operations require more threads



Synchronous API

```
for (Integer id : employeeIds) {

    // Step 1: Fetch Employee details from DB
    Future<Employee> future = service.submit(new EmployeeFetcher(id));
    Employee emp = future.get(); // blocking

    // Step 2: Fetch Employee tax rate from REST service
    Future<TaxRate> rateFuture = service.submit(new TaxRateFetcher(emp));
    TaxRate taxRate = rateFuture.get(); // blocking

    // Step 3: Calculate current year tax
    BigDecimal tax = calculateTax(emp, taxRate);

    // Step 4: Send email to employee using REST service
    service.submit(new SendEmail(emp, tax));
}
```

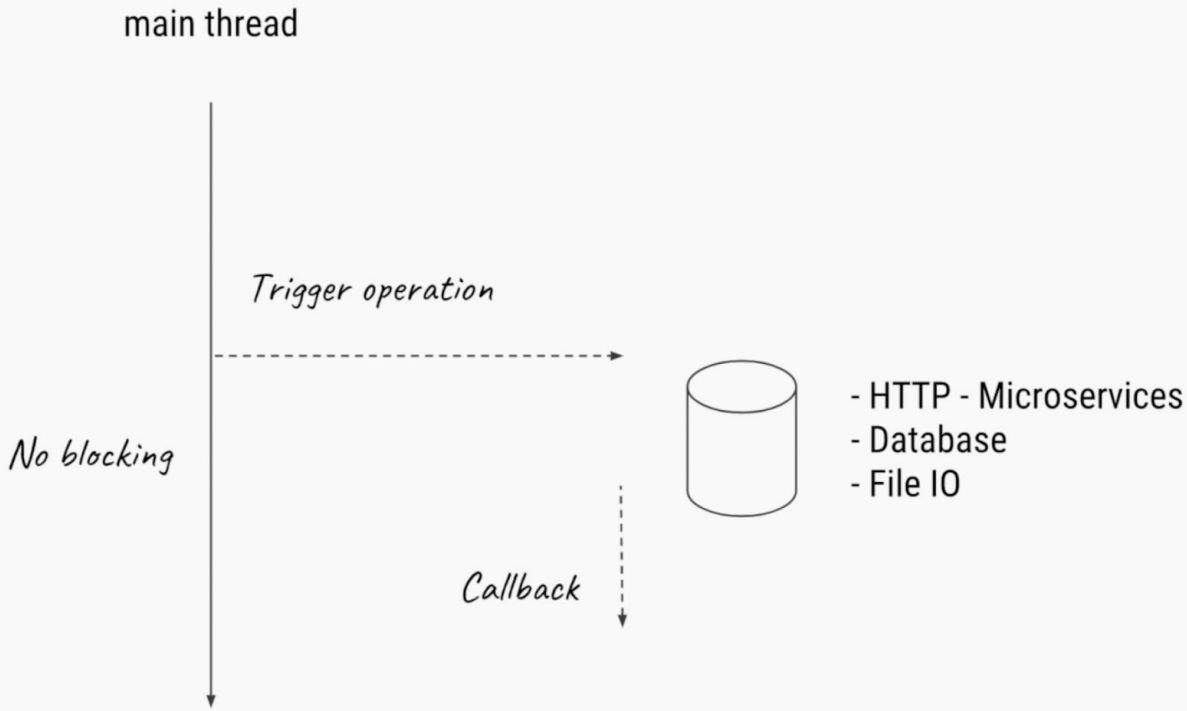
Problem

Expensive Threads
&
Blocking IO Ops
=
Limited Scalability

Solution

Non-blocking IO
&
Asynchronous API

Non-blocking IO



Conceptual Depiction

Java NIO

- Buffers, Channels, Selectors
- Low-level API for asynchronous / non-blocking IO
- Applicable for Files, Sockets,
- Listener based (callbacks)

```
ByteBuffer buffer = ByteBuffer.allocate(1024);

Path path = Paths.get("/home/file2");
AsynchronousFileChannel fileChannel = AsynchronousFileChannel.open(path, StandardOpenOption.READ);

fileChannel.read(buffer,  position: 0, buffer, new CompletionHandler<Integer, ByteBuffer>() {

    @Override
    public void completed(Integer result, ByteBuffer data) {
        // process data
    }
```

<= Callback

Servlets 3.0 - Allows more concurrent requests

```
@WebServlet(urlPatterns={"/user"}, asyncSupported=true)
public class UserAsyncServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response) {

        final AsyncContext context = request.startAsync();
        context.start(new Runnable() {
            public void run() {
                // make the network call
                ServletResponse response = context.getResponse();
                // print to the response
                context.complete();
            }
        });
    }
}
```

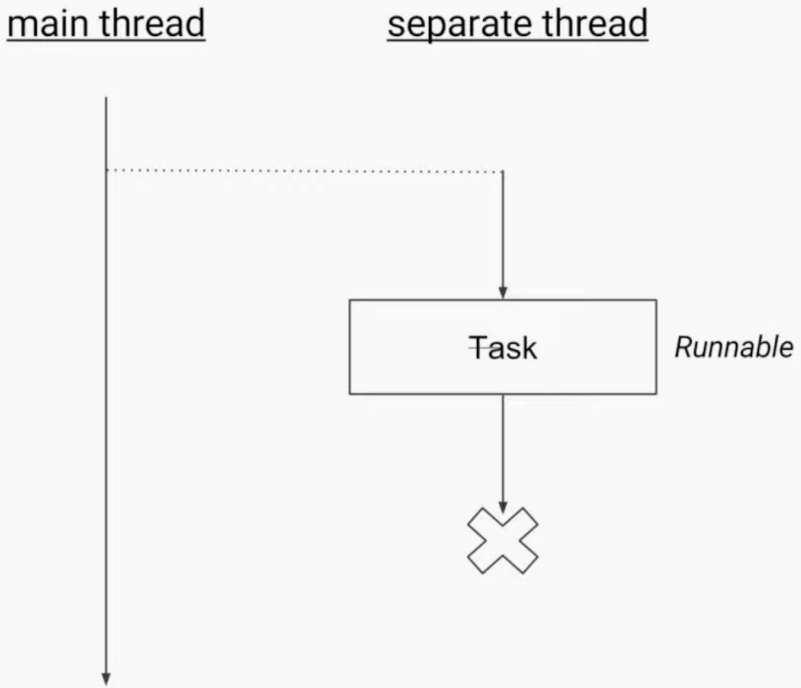
Run operations in separate thread

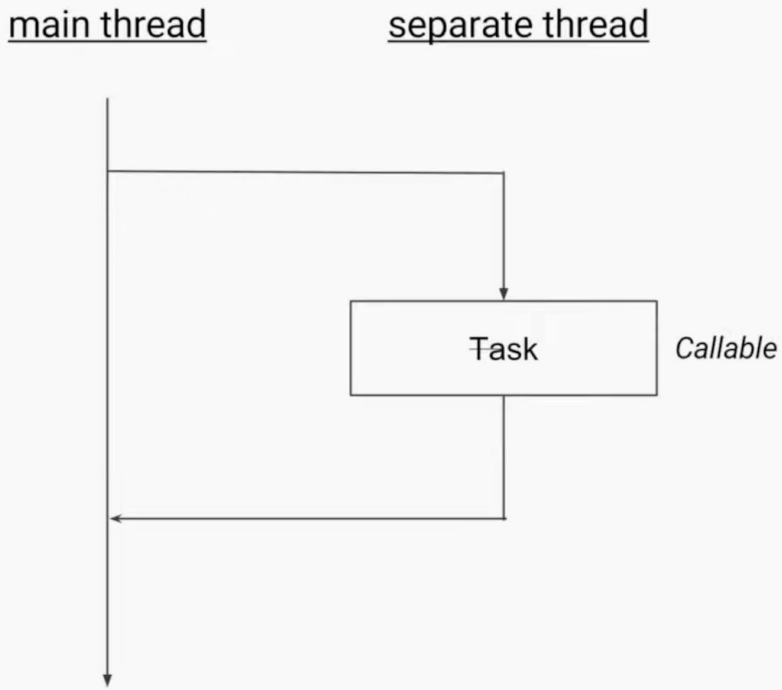
Immediately return to serve other requests

Completable Future

What is it used for?

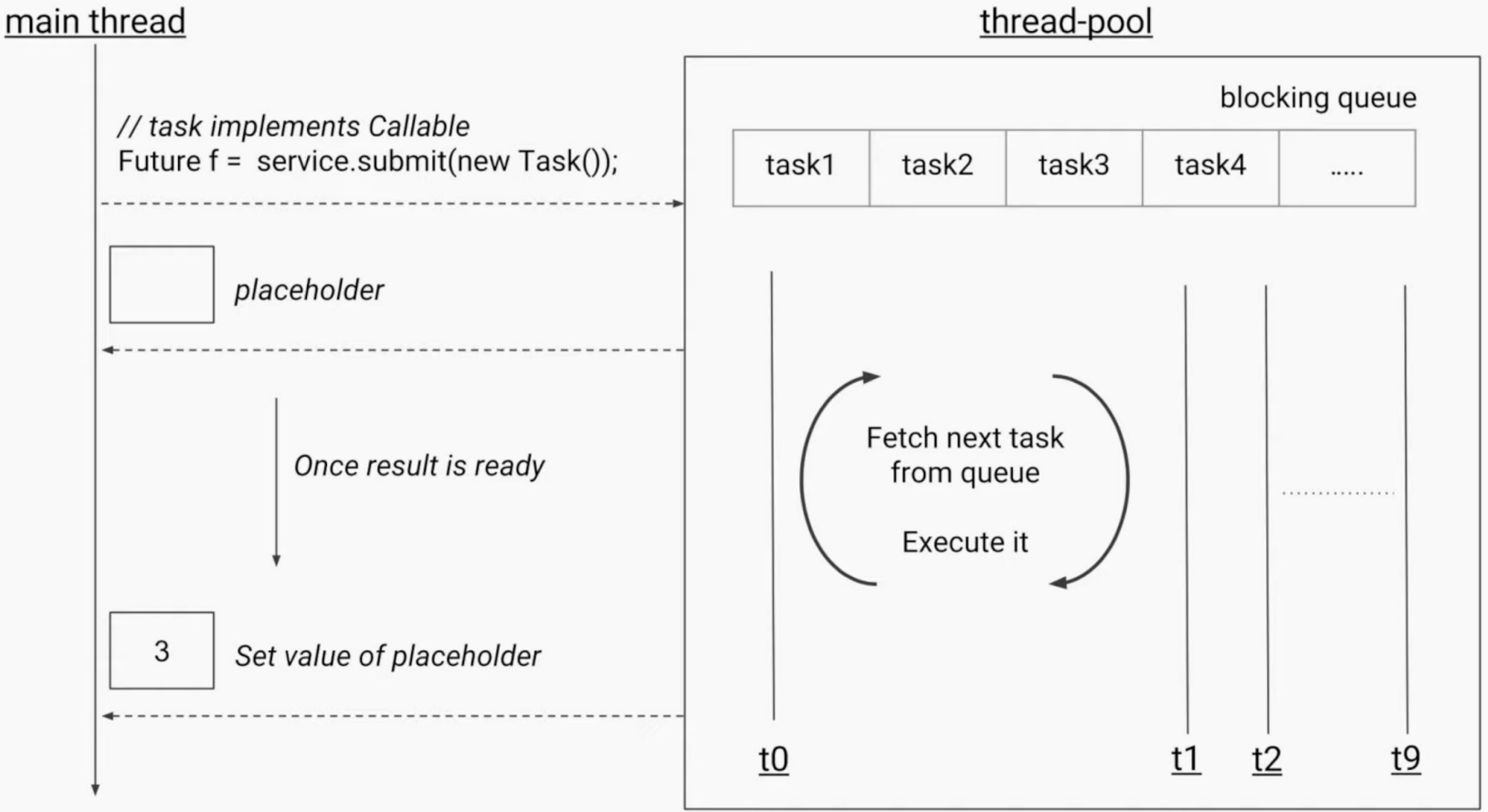
Perform possible asynchronous (non-blocking) computation and trigger dependant computations which could also be asynchronous.

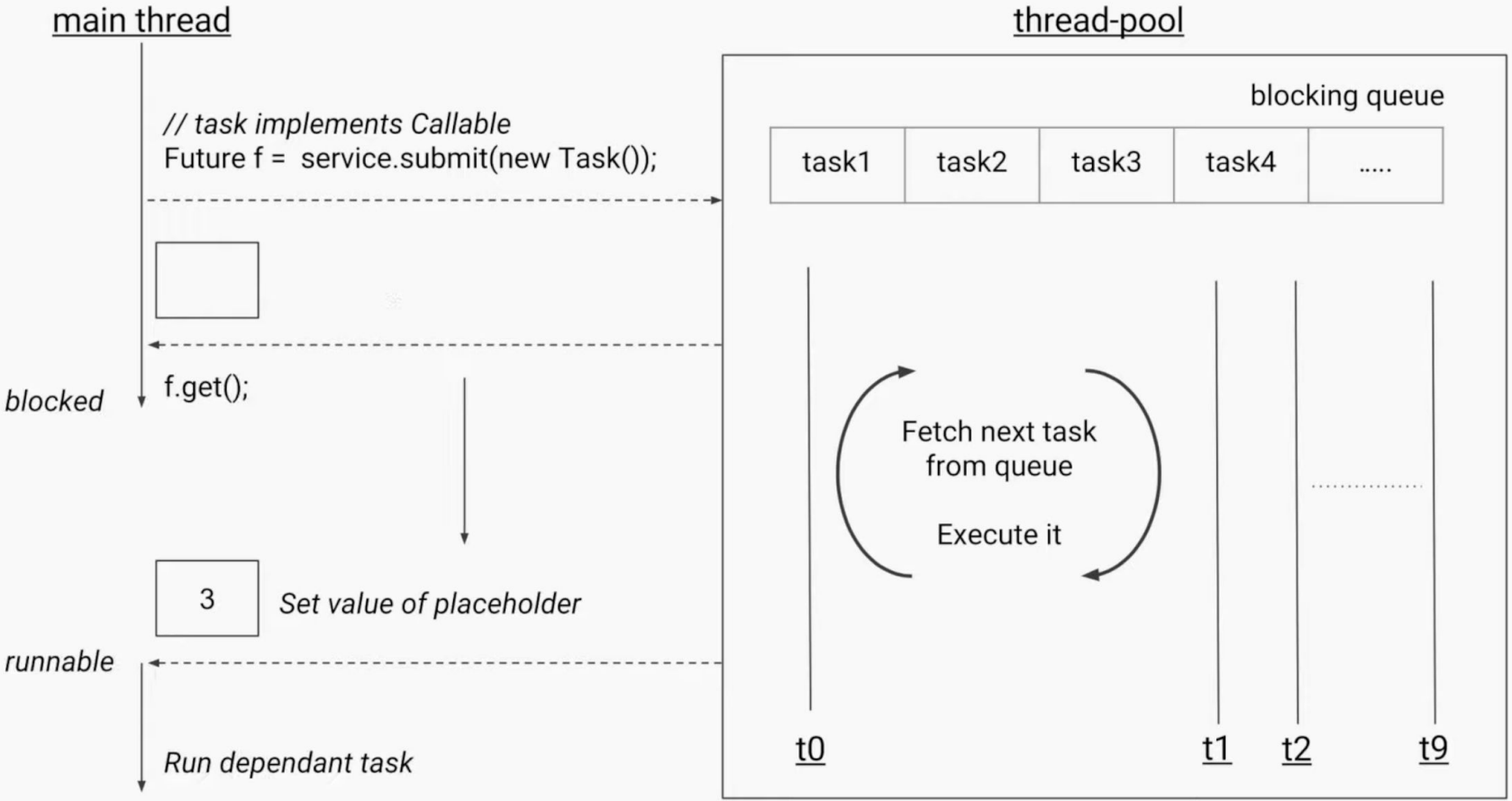

main thread
separate thread
Task
Runnable


main thread
separate thread
Task
Callable

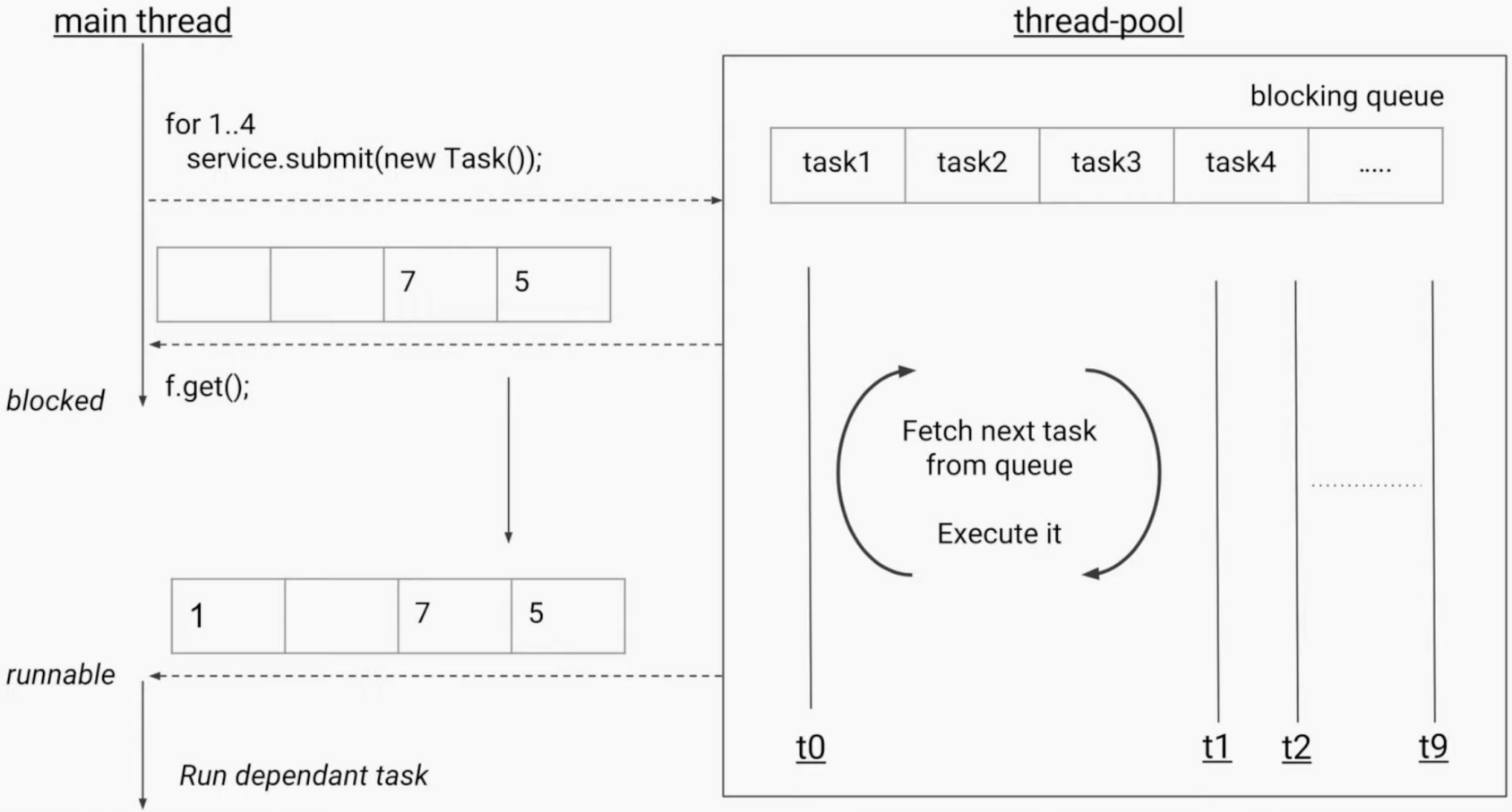
Working with Callable / Futures

```
public static void main(String[] args) {

    ExecutorService service = Executors.newFixedThreadPool( nThreads: 10);

    // submit task and accept the placeholder object for return value
    Future<Integer> future = service.submit(new Task());

    try {
        // get the task return value
        Integer result = future.get(); // blocking
        System.out.println("Result from the task is " + result);

    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}

static class Task implements Callable<Integer> {
    @Override
    public Integer call() {
        return new Random().nextInt();
    }
}
```

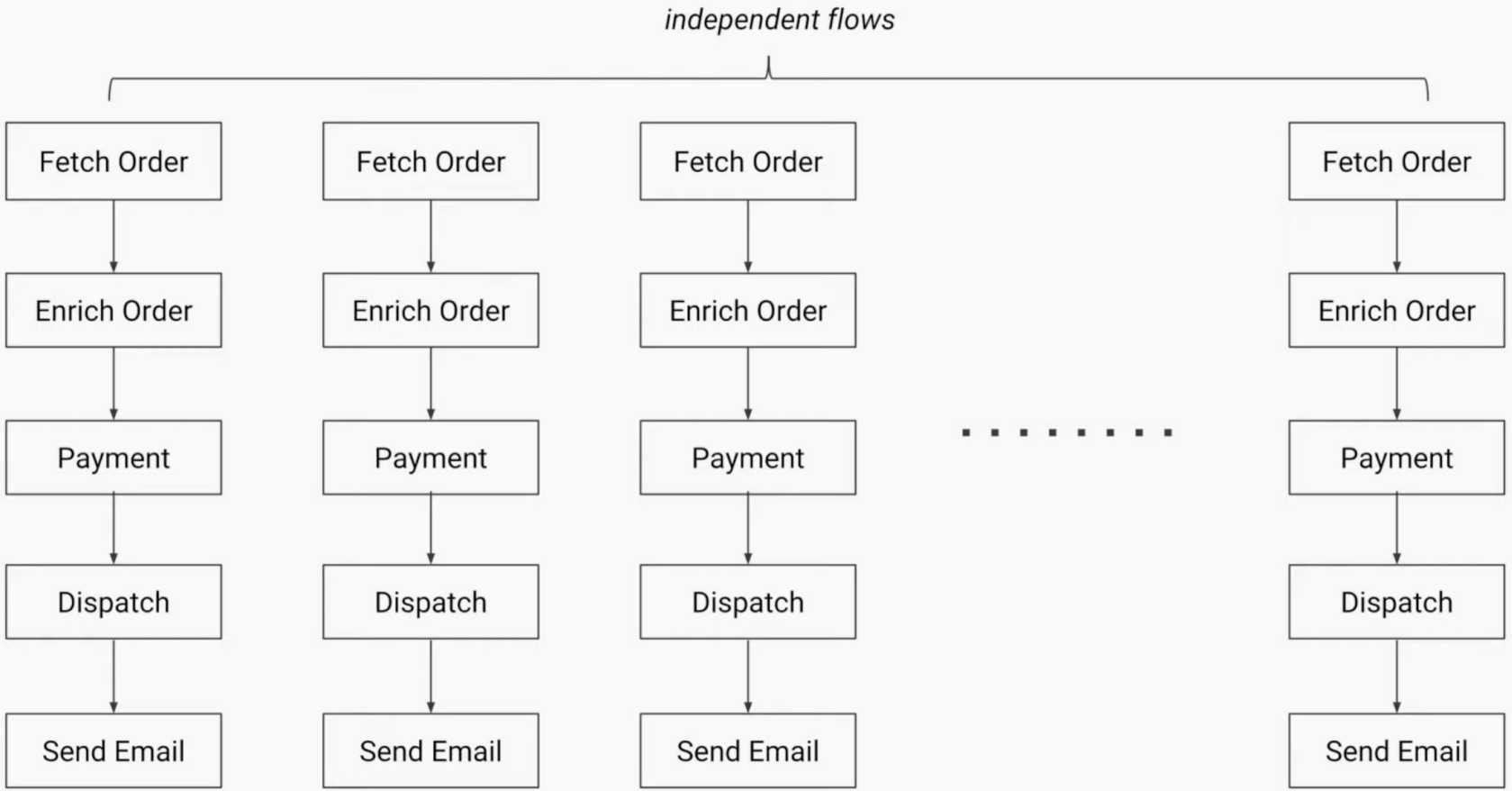

main thread
thread-pool
// task implements Callable
Future f = service.submit(new Task());
blocking queue
task1
task2
task3
task4
.....
placeholder
Once result is ready
Fetch next task
from queue
Execute it
3
Set value of placeholder
t0
t1
t2
t9

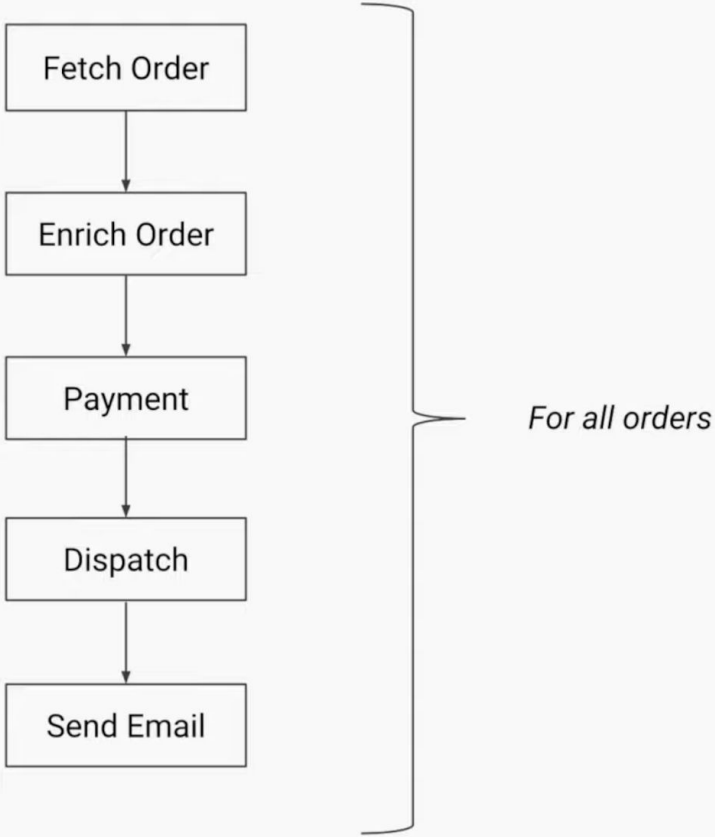


Blocking get operation



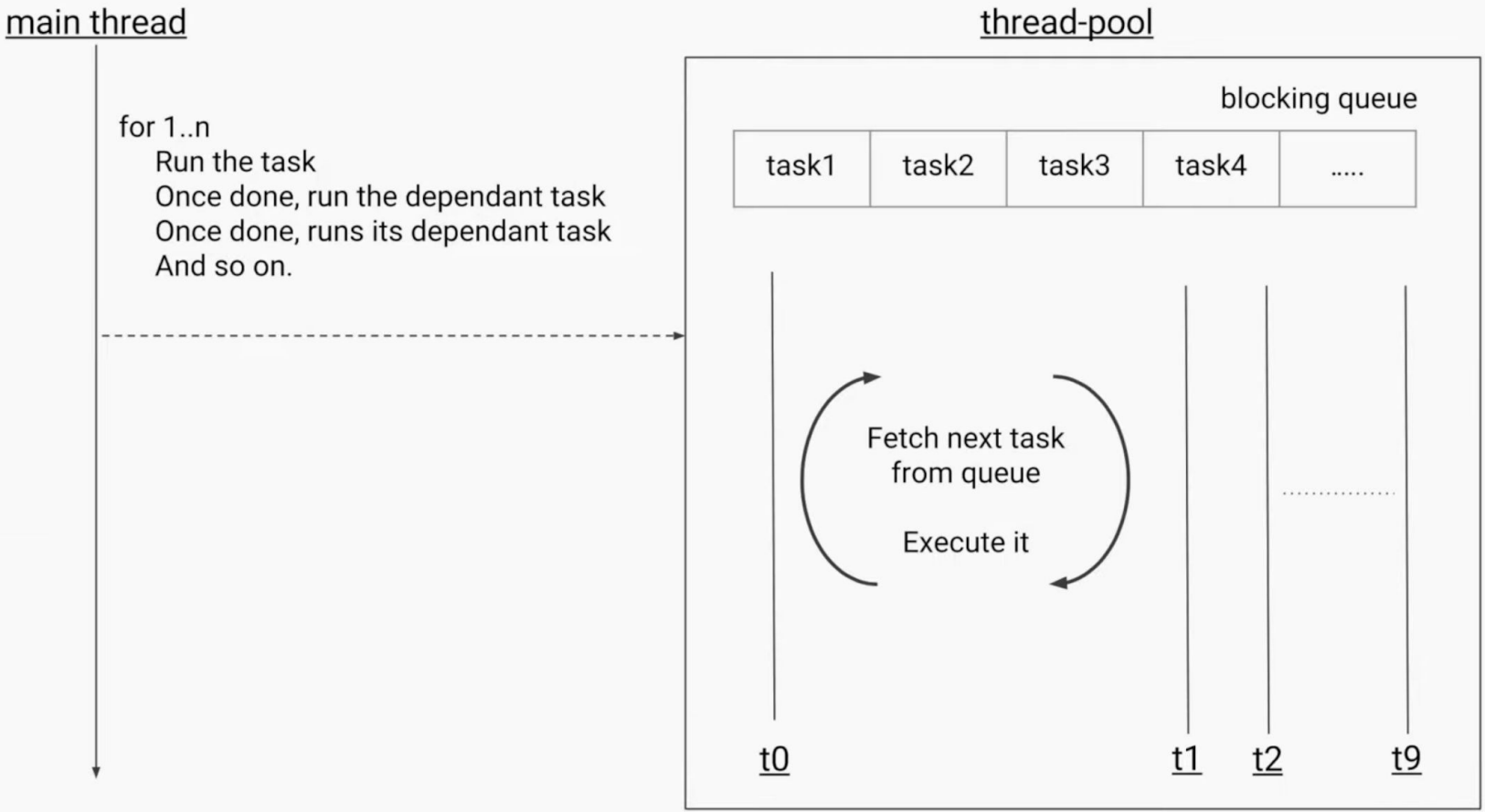
There is a problem!


independent flows
Fetch Order
Enrich Order
Payment
Dispatch
Send Email
Fetch Order
Enrich Order
Payment
Dispatch
Send Email
Fetch Order
Enrich Order
Payment
Dispatch
Send Email
Fetch Order
Enrich Order
Payment
Dispatch
Send Email



Dependant tasks

```java
ExecutorService service = Executors.newFixedThreadPool( nThreads: 10);

try {

    Future<Order> future = service.submit(getOrderTask());
    Order order = future.get();  // blocking

    Future<Order> future1 = service.submit(enrichTask(order));
    order = future1.get(); // blocking

    Future<Order> future2 = service.submit(performPaymentTask(order));
    order = future2.get(); // blocking

    Future<Order> future3 = service.submit(dispatchTask(order));
    order = future3.get(); // blocking

    Future<Order> future4 = service.submit(sendEmailTask(order));
    order = future4.get(); // blocking

} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
```

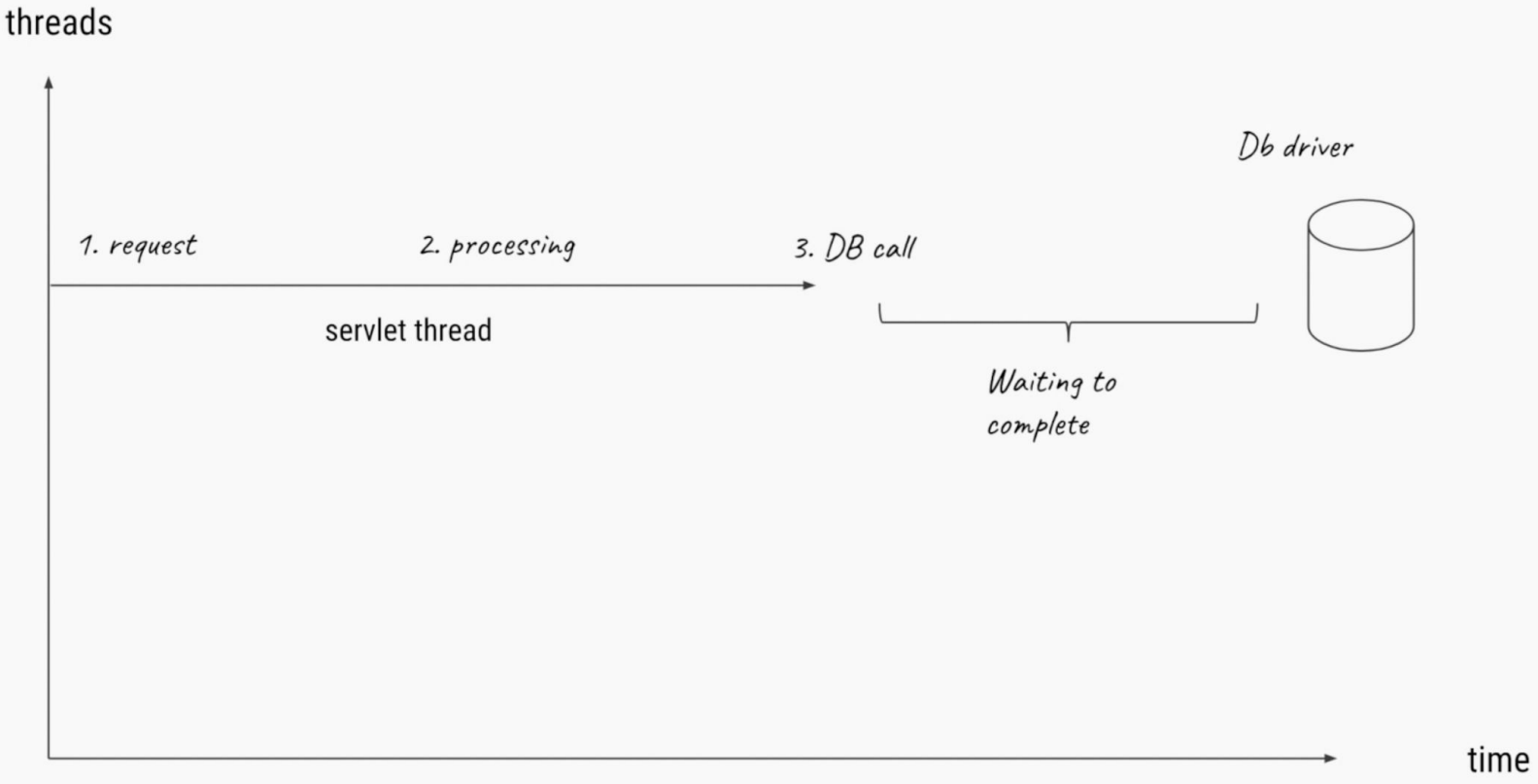


What we actually want

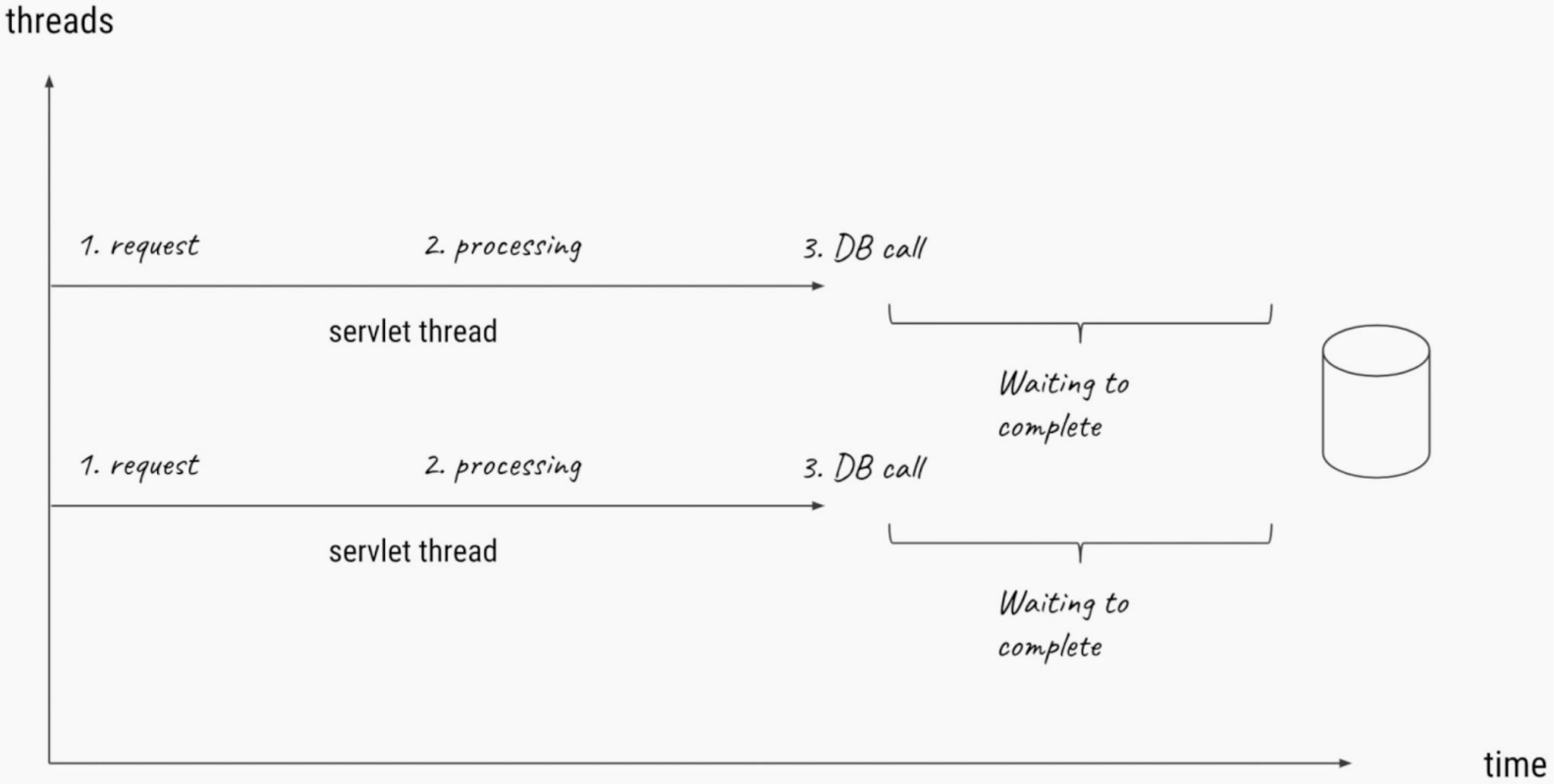
What we actually want

Thread Perspective

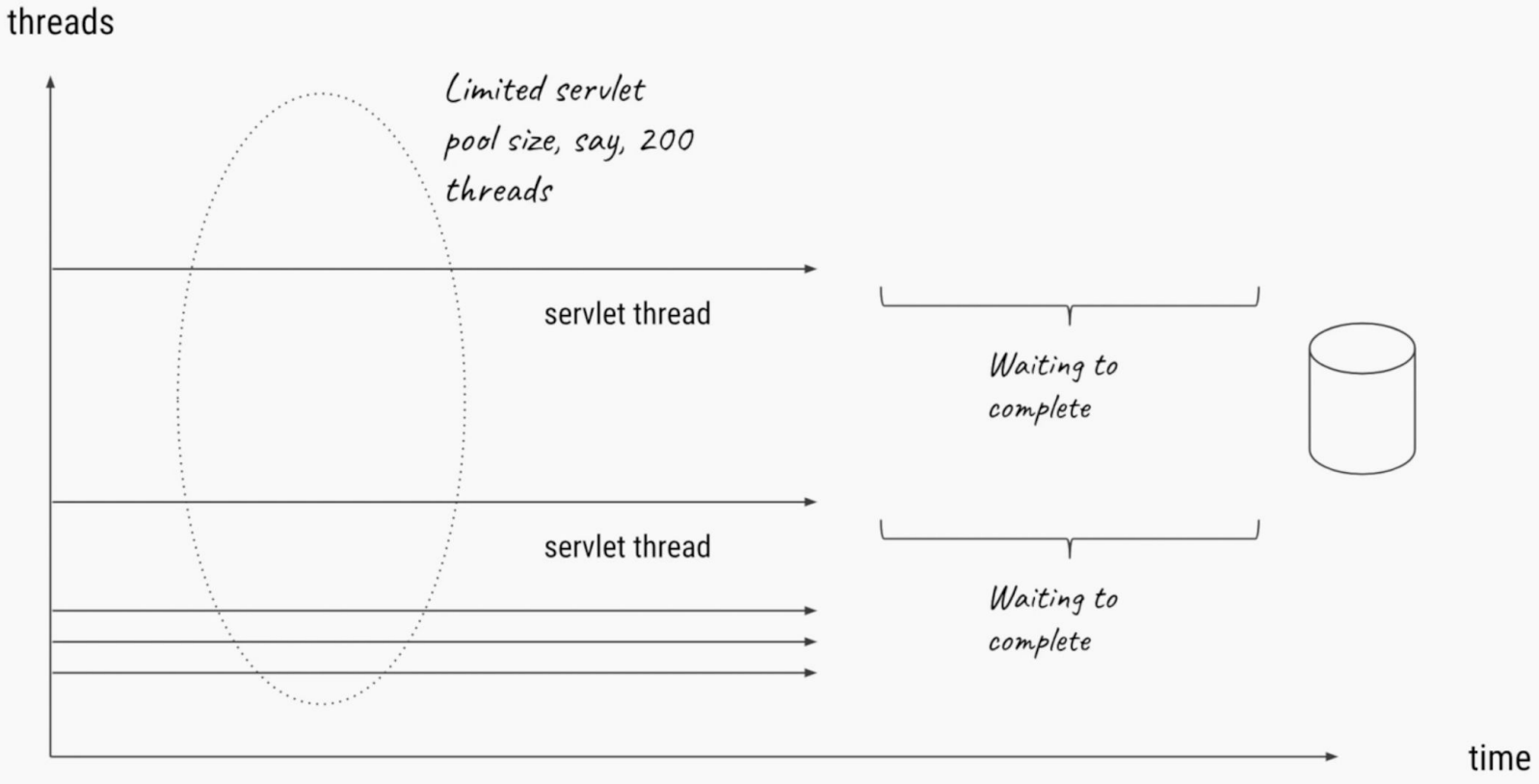
Web requests with IO



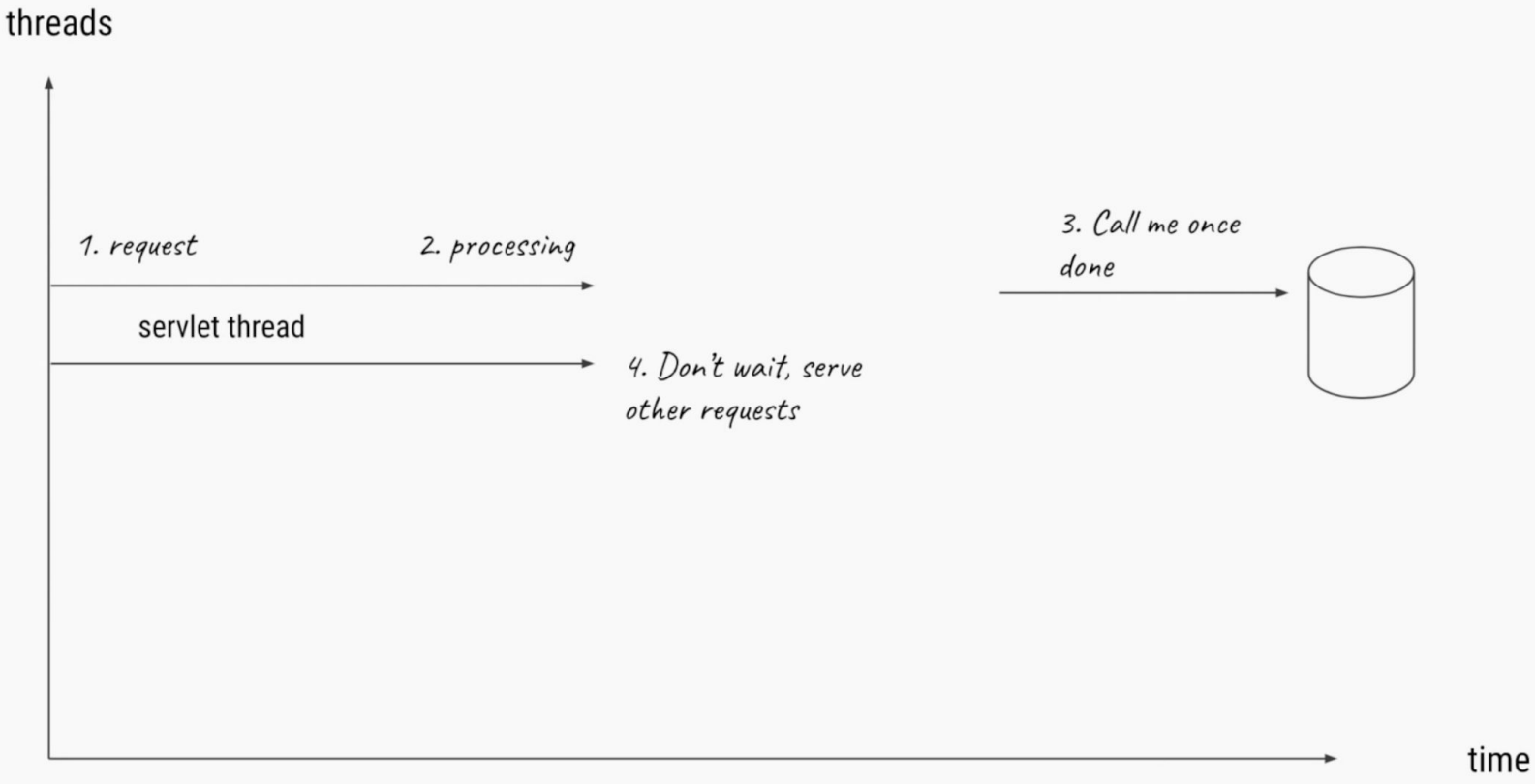
Web requests with IO



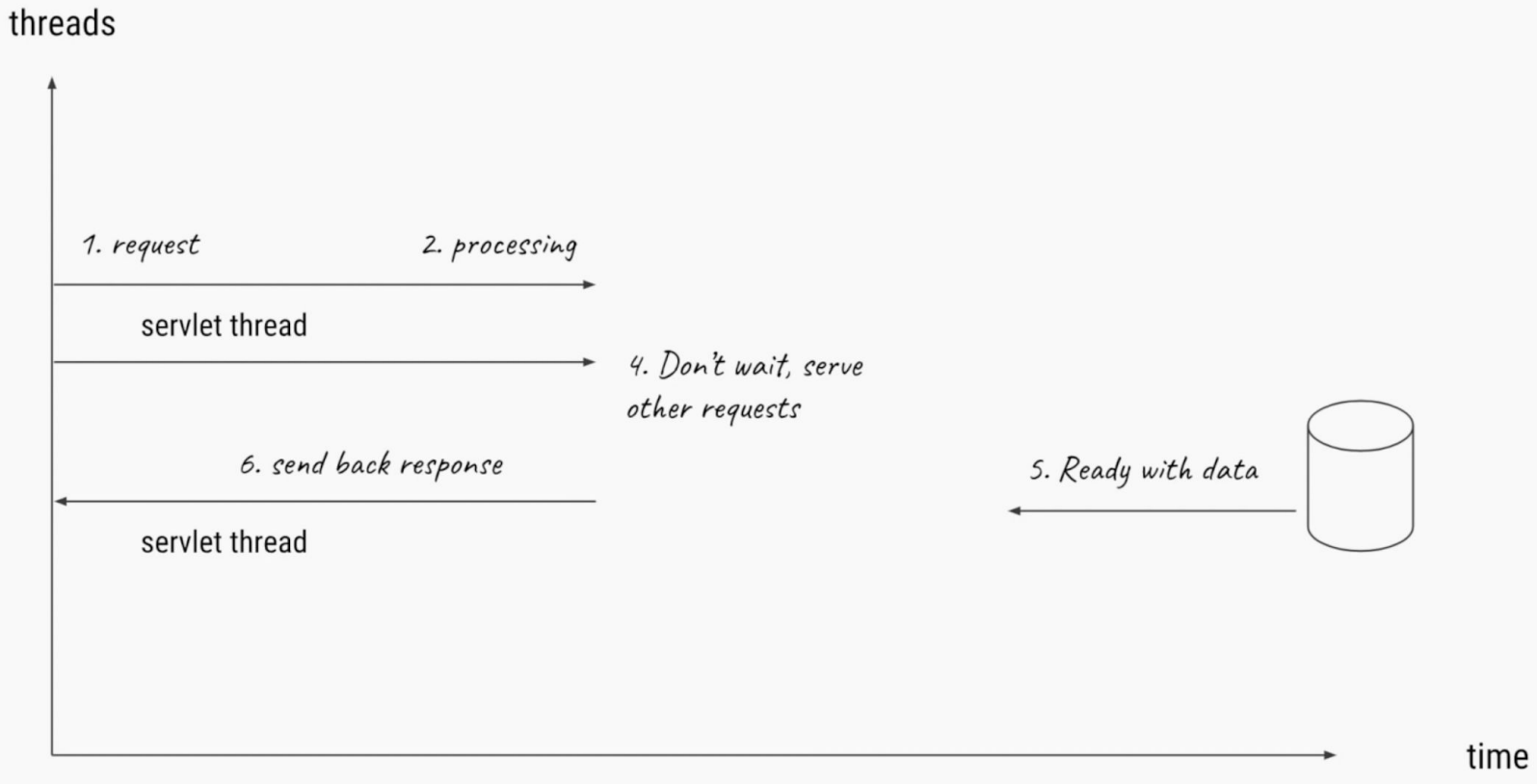
Web requests with IO



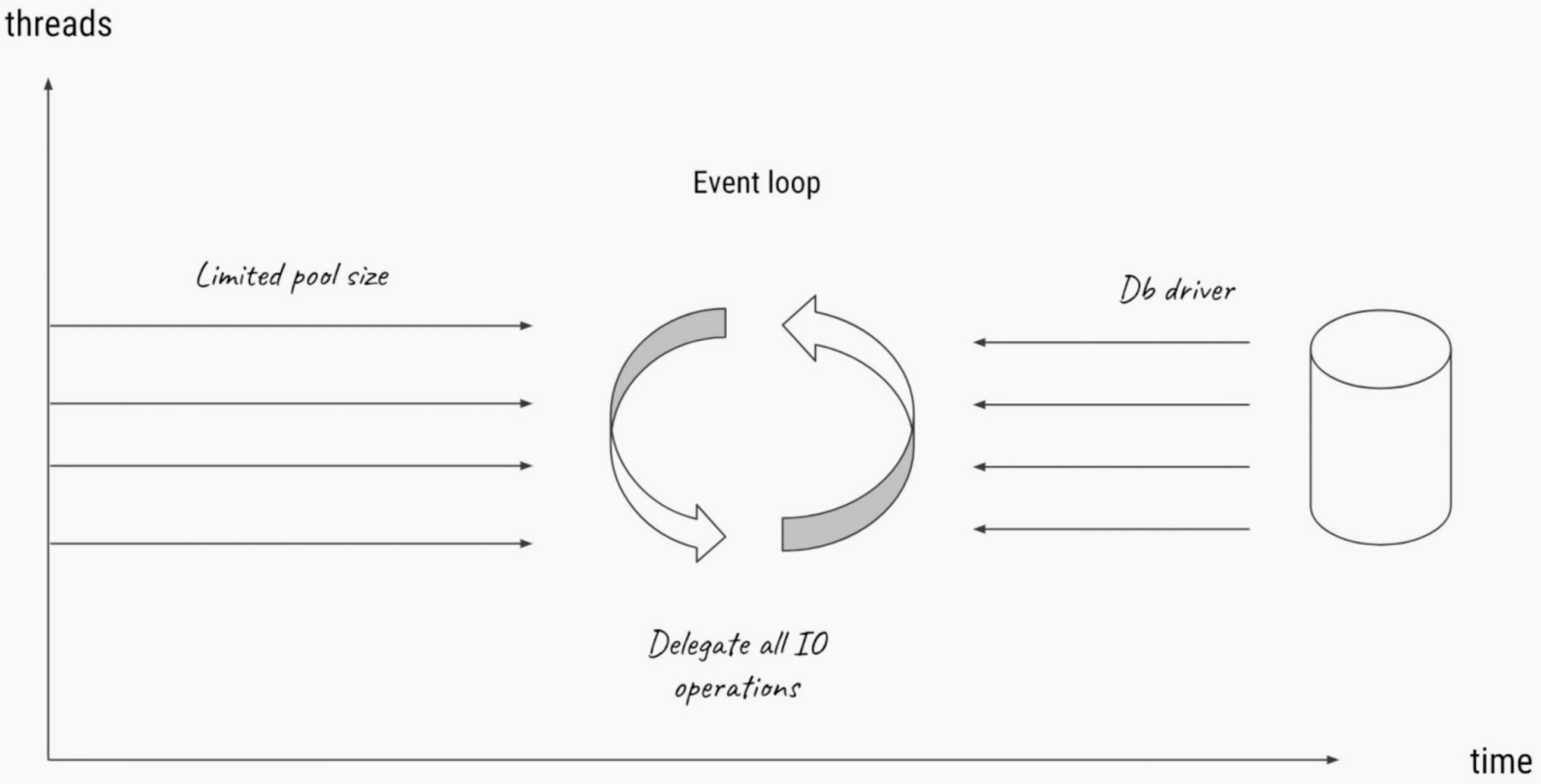
Web requests with IO



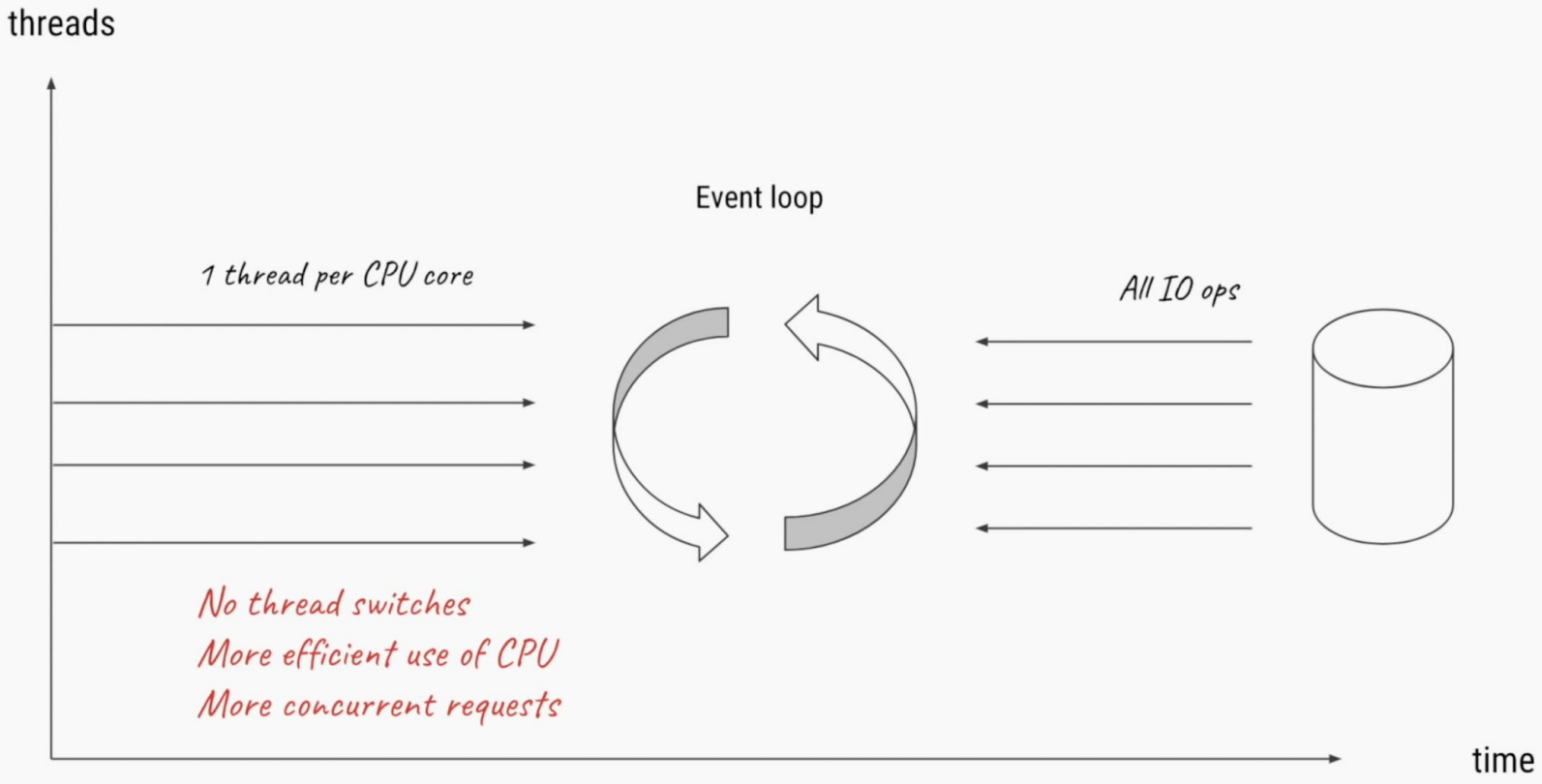
Web requests with IO



High throughput

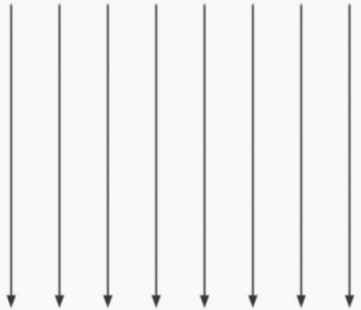


High throughput



Traditional vs Event Loop

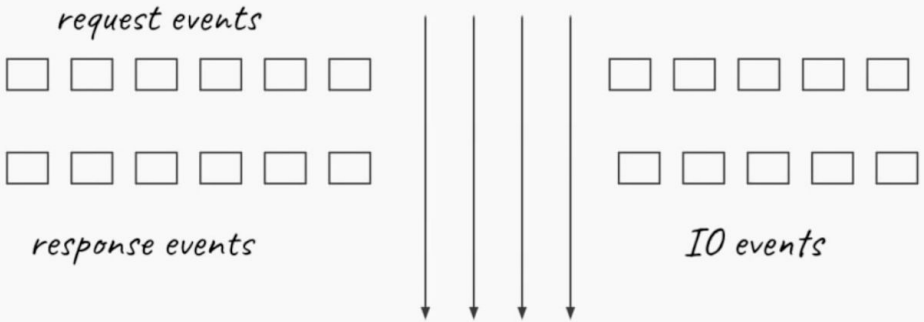
1 thread per request flow



core1	core2
core3	core4

Lot of thread switches
Scheduling of threads

1 thread per CPU core

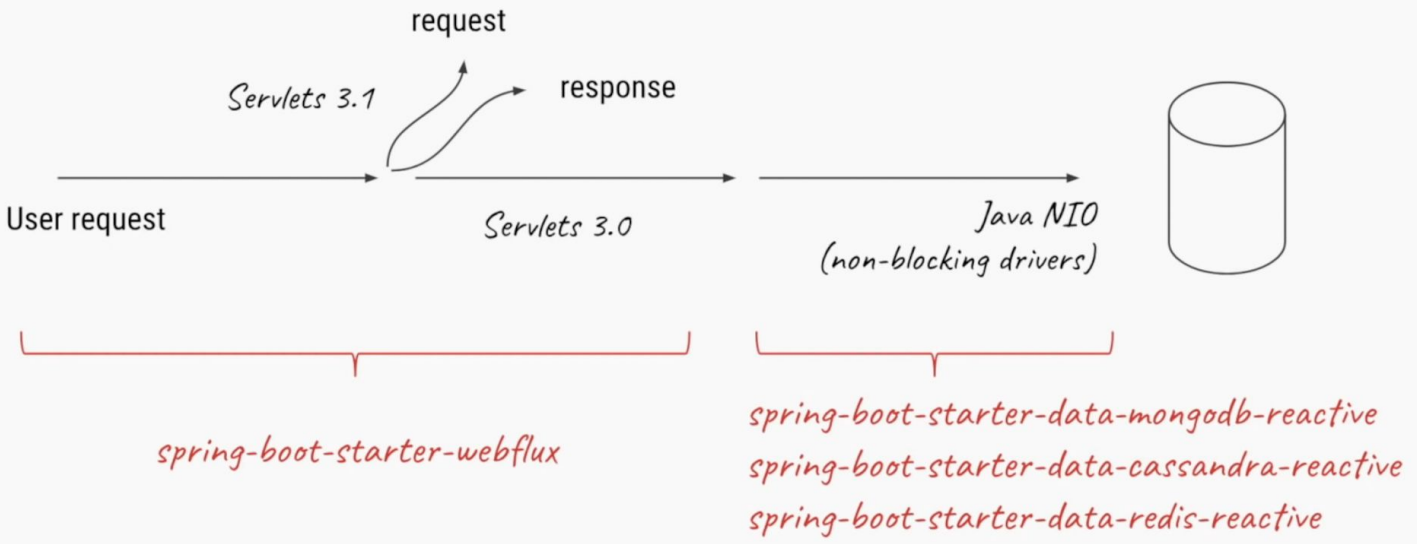


core1	core2
core3	core4

No thread switching
Increased complexity

End to end reactive

- Servlets 3.0 and 3.1 (Async and NIO)
- Java NIO (File, Network IO)
- MongoDB, Cassandra, Redis (more to come)



Code Perspective

Simple web request flow

```java
@RestController
public class MyRestController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/get/user/{id}")
    public User getUser(@PathVariable String id) {
        return userRepository.findUser(id);
    }
```

Blocking call

Future as placeholders?

```
@RestController
public class MyRestController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/get/user/{id}")
    public Future<User> getUser(@PathVariable String id) {
        return userRepository.findUser(id);
    }
```

CompletableFuture as placeholders?

```
@RestController
public class MyRestController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/get/user/{id}")
    public CompletableFuture<User> getUser(@PathVariable String id) {
        return userRepository.findUser(id);
    }
```

Spring Webflux - using Reactor

```
@RestController
public class MyRestController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/get/user/{id}")
    public Mono<User> getUser(@PathVariable String id) {
        return userRepository.findUser(id);
    }
```

Mono = 0..1 elements

Flux = 0..N elements

Continuous data responses

```
@RestController
public class PricingController {

    @Autowired
    private PricesRepository pricesRepository;

    @GetMapping("/get/prices/{id}")
    public Mono<Price> getPrice(@PathVariable String id) {
        return pricesRepository.findById(id);
    }

    @GetMapping("/get/prices/all")
    public Flux<Price> getAllPrice() {
        return pricesRepository.findAll();
    }

    @GetMapping(value = "/get/prices/live", produces = "text/event-stream")
    public Flux<Price> getLivePrice() {
        return pricesRepository.findAll();
    }
```

Reactive DB driver

Single value

List of values

Live list of values

Continuous data responses