- DA343A -
Objektorienterad programutveckling, trådar och datakommunikation

Föreläsning 12
Servers

Ben Blamey

Föreläsning 12 (Datakom F9)

- HTTP / TCP Recap
- Server Performance, Concurrency & Design
- Iterative server
- Iterative server with UDP
- Iterative server with TCP
- Multi-threaded server
- Multi-threaded server with UDP
- Multi-threaded server with TCP
- Thread Pools
- Async IO (not for exam)

1. HTTP / TCP Recap

- **HTTP:**
  - is an **application protocol** used for, e.g.
  - (a) requesting resources (pages, images, etc.) when browsing the web. (typically GET verb)
  - (b) submitting forms/data to web servers from browsers. (typically POST verb)
  - Also, used for REST "APIs" – often to send/receive resources in the form of JSON or XML documents. Such APIs can be queried from scripts inside webpages, and/or used outside the browser – e.g. mobile applications.
  - I even use the Canvas API to get a list of student submissions! https://canvas.instructure.com/doc/api/
  - HTTP is built on top of **TCP (**which is a **transport protocol)**
  - **Why not UDP?**
  - **TCP** is a good fit for HTTP – because of TCP's guarantees about data ordering and corruption-checking.
  - HTTP sends the request / response headers as lines of text.

(Show API example in postman)

2. HTTP / TCP Demo

# Open netcat in listening mode, port 5000
$ nc -l 5000

Open browser and make a web request to that port:
http://localhost:5000/

http:// - application protocol HTTP. (unencrypted).
  localhost – host to send request to
    :5000 – custom port (default for HTTP is 80)
      / - the page (resource) to request (root)

3. HTTP / TCP Demo

Response (text):

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 11

Hej DA343A!
```

Note: ISO-8859-1 (Latin-1) encoding is assumed by default.

4. HTTP / TCP Demo

Response (HTML):

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 64

<html><body><marquee>Hej <b>DA343A</b>!</marquee></body></html>
```

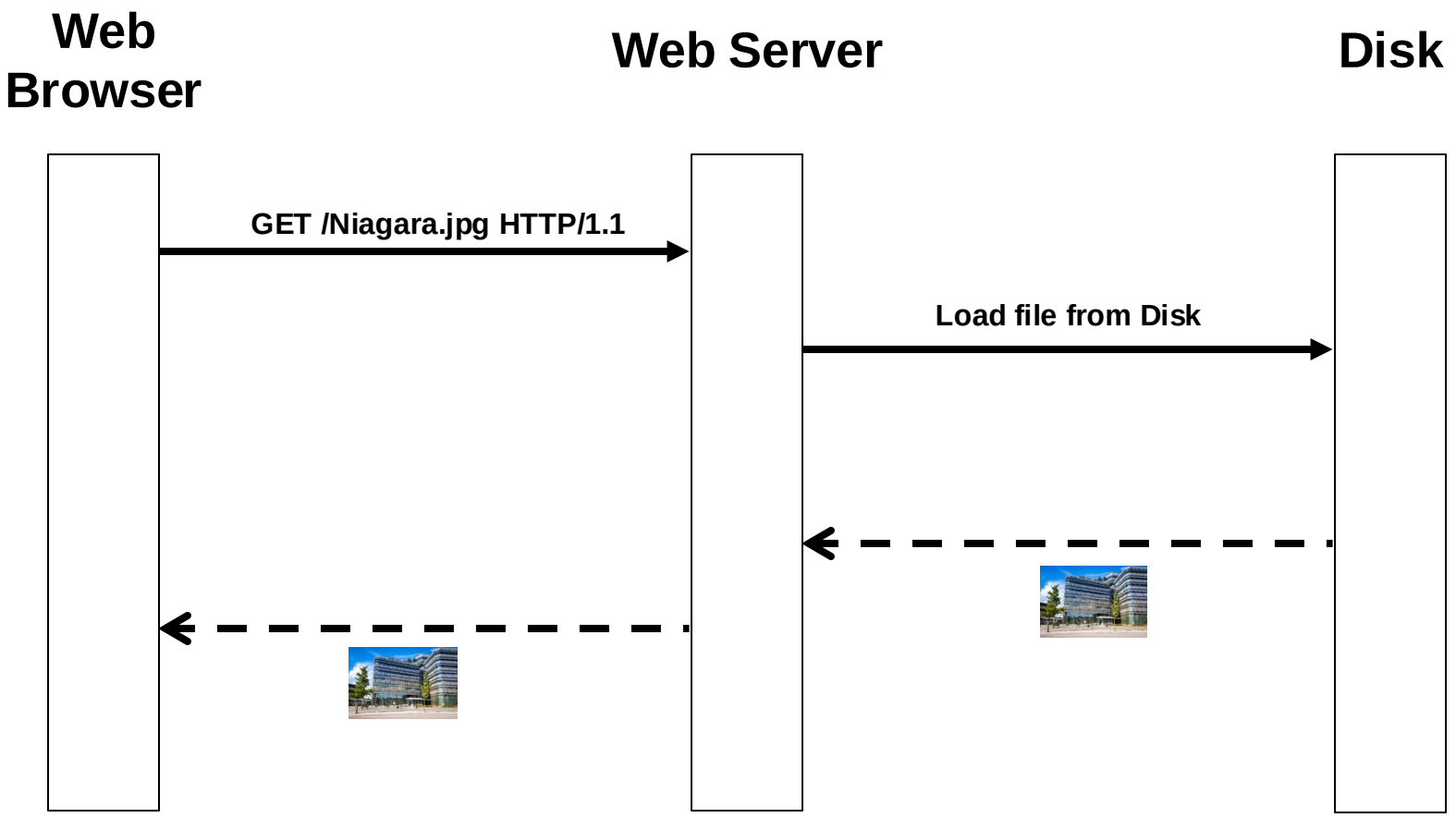
Note: ISO-8859-1 (Latin-1) encoding is assumed by default.

5. Other Examples of Transport Protocols?

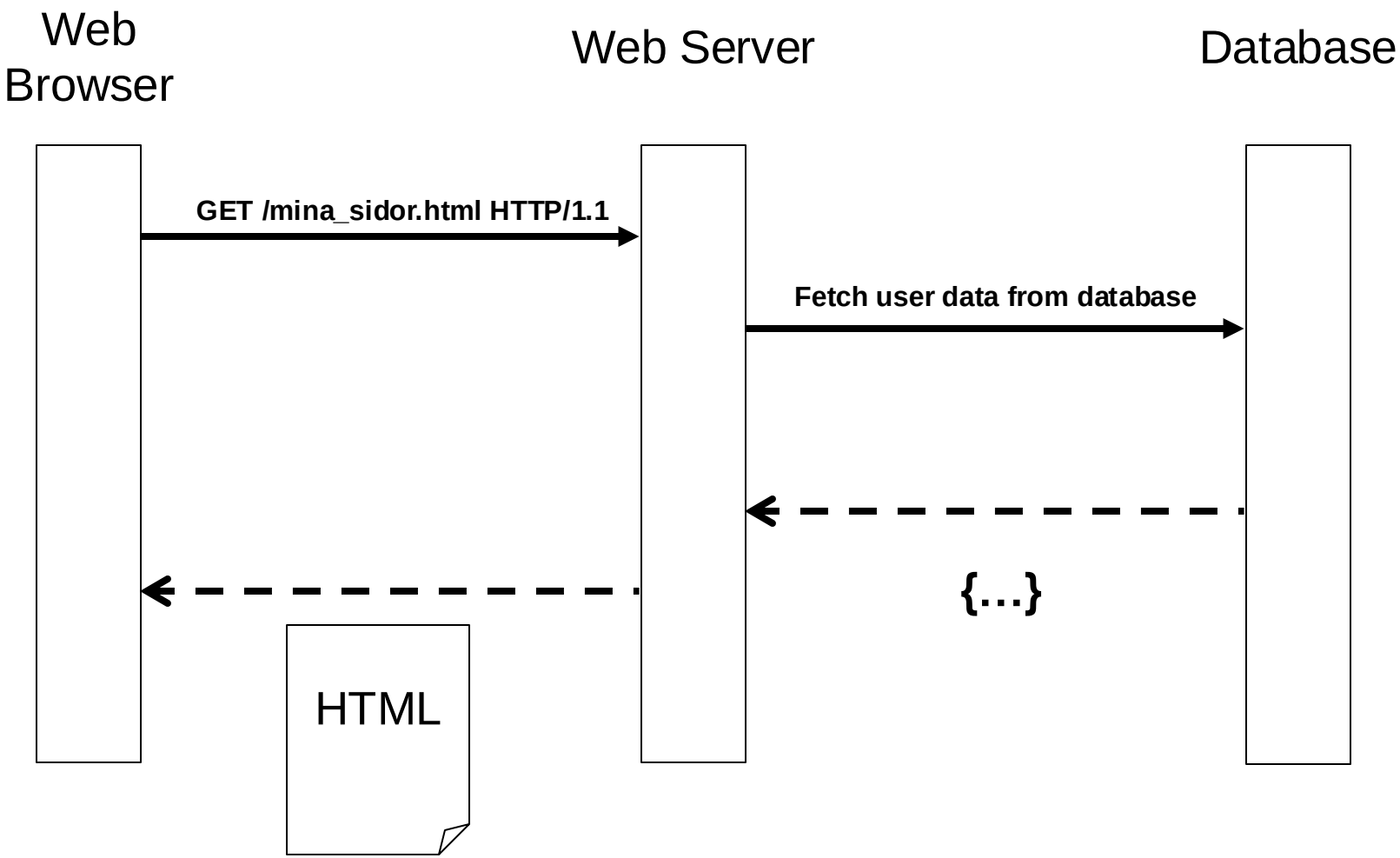
- QUIC – Alternative transport protocol, intended for use with HTTP version 3.
- Uses UDP instead of TCP.
- Multiple “channels” of UDP messages are multiplexed.
- Because UDP does not prevent data loss, this has to implemented within the application protocol.
- Reduces latency, particular for establishment of secure connections, with initial key exchange.
- https://en.wikipedia.org/wiki/QUIC

- (Not for Exam)

6. Example: (Static) Web Server - HTTP



7. Example: (Dynamic) Web Server - HTTP

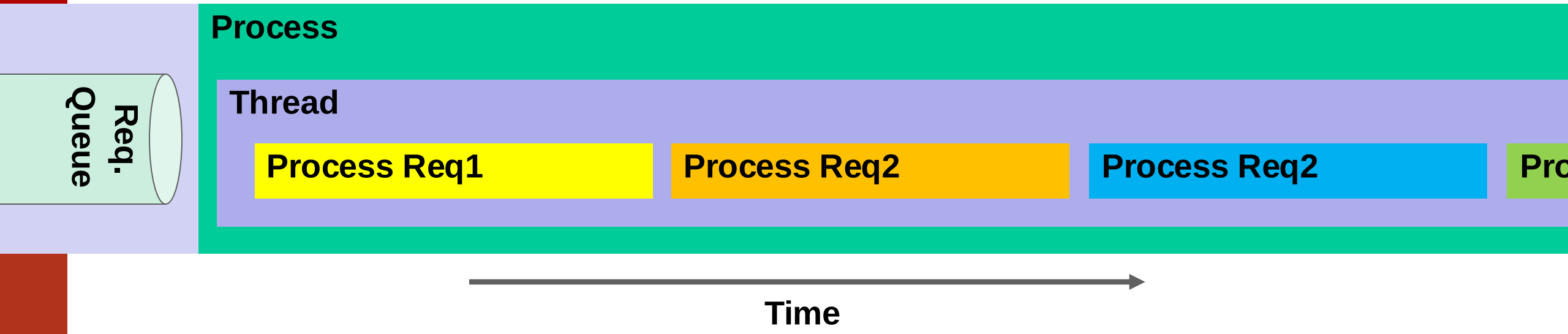


Server Performance, Concurrency & Design

9. Iterative vs Concurrent Servers

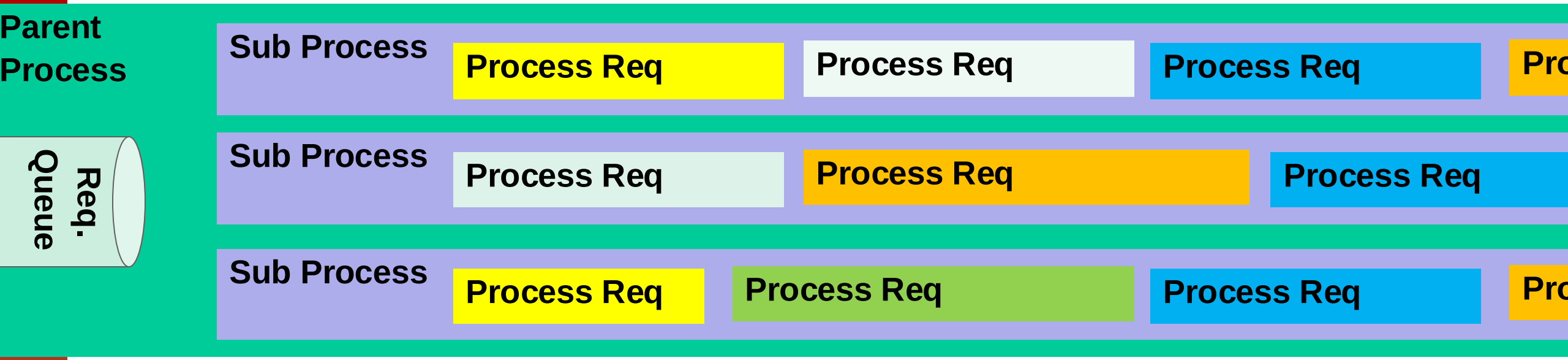
Iterative Web Server	Single thread. Handle requests in turn.
Concurrent Server, One **Process** per Request	For each new request, start a new (sub)process to handle the request. Process handles the request, then exits.
Concurrent Server, (New) **Thread** per Request	For each new request, start a new thread to handle the request. Thread handles the request, then exits.
Concurrent Server, **Thread** per Request with **Thread Pool**	Server has a queue of requests, and a pool of worker threads. Each thread takes a request off the queue, processes it, and fetches another when it is finished.
Concurrent Server, single thread (Asychonous IO)	Requests are handled concurrently, but on a single thread. (Not on syllabus / exam.)

10. Iterative Web Server, Single Thread



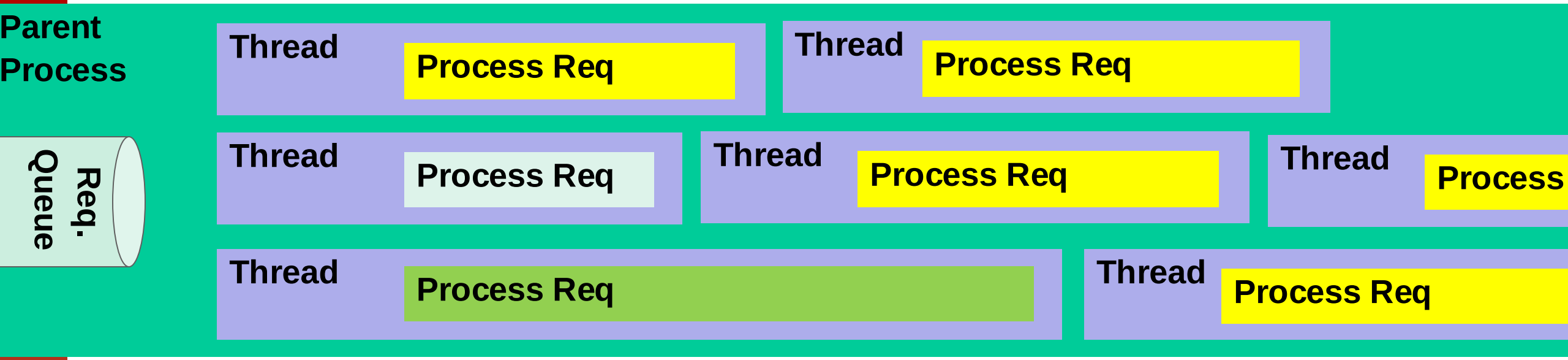
- No concurrency.
- Queue is managed by the OS.
- Queue is truncated if it exceeds a set limit.
- Disadvantage: pending requests wait in the queue, increasing latency for clients unless traffic is low.
- Advantage: simplicity of implementation.
- Limited use in practice (teaching examples!)

11. Concurrent Server, One **Process** per Request



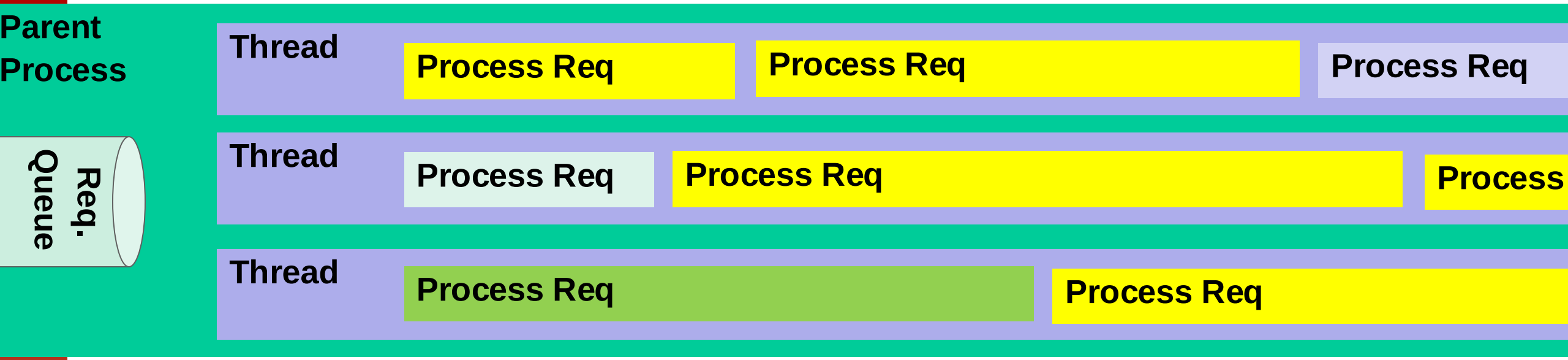
- Concurrency – multiple requests processed at the same time! Better performance.
- Pool of processes to handle requests from the queue. Fixed number of processes in pool.
- Parent process manages queue and processes.
- Processes (e.g.) re-used set number of times and then restarted.
- Advantage: allows use of libraries which are not thread-safe. (e.g. older PHP libraries)
- Disadvantage: overhead resource (memory) cost to using threads.
- Example: 'Prefork' module in Apache HTTPD https://httpd.apache.org/docs/2.4/mod/prefork.html

12. Concurrent Server, (New) **Thread** per Request



- Or, new thread **per connection** – and re-use TCP connection for multiple requests.
- Concurrency – multiple requests processed at the same time!
- Each request/connection gets a **new** thread. Thread processes a single request.
- Limit on maximum number of threads, otherwise requests are queued.
- Advantage: threads are 'lighter' than processes.
- Disadvantage: threads still have overhead. Increased complexity, implementing thread safety. Performance cost of restarting threads.

13. Concurrent Server, **Thread** Pool



- Thread **per connection –** but a ‘pool’ of threads is used (and recycled).
- Concurrency – multiple requests processed at the same time!
- Thread processes a single request at a time.
- Limit on maximum number of threads, otherwise requests are queued.
- Advantage vs. new threads: avoid cost of starting a thread each time.
- Threads periodically restarted to avoid memory leaks.
- Common Implementation (e.g. JavaEE, ASP.NET)

Code Examples

14. Iterative server, UDP

LotteryServerA

LotteryClientAC

An iterative server that uses UDP is simple to write.

1. Create DatagramSocket.
2. Listen for request
3. Manage request
4. Send Response
5. Back to step 2

```
public Server(port) throws SocketException {
    socket = new DatagramSocket(port);
    thread.start();
}
public void run() {
    DatagramPacket packet;
    String response, ticket;
    byte[] buffer = new byte[256];
    byte[] outData;

    while(true) {
        try {
            packet = new DatagramPacket(buffer,buffer.length);
            socket.receive(packet);

            ticket = new String(packet.getData(),0,packet.getLength());
            response = checkTicket(ticket);

            outData = response.getBytes();
            packet = new DatagramPacket(outData,outData.length,
                                        packet.getAddress(),packet.getPort());
            socket.send(packet);
        } catch(Exception e) {
            System.err.println(e);
        }
    }
}
```

I LotteryServerA är *checkTicket* långsam (0-3 sek).

15. Iterative server, TCP

LotteryServerB

LotteryClientBDF

An iterative server that uses TCP is simple to write.

Create ServerSocket.

Listen for connecting client

Create streams

Read request

Manage request

Send Response

Close connection (exit the ”try-with-resources”)

Back to step 2

```
public Server(int port) throws IOException {
    serverSocket = new ServerSocket(port);
    thread.start();
}

public void run() {
    String ticket,response;
    System.out.println("Server running");
    while(true) {
        try (Socket socket = serverSocket.accept();
             DataInputStream dis =
                  new DataInputStream( socket.getInputStream());
             DataOutputStream dos =
                  new DataOutputStream( socket.getOutputStream())) {
            ticket = dis.readUTF();

            response = checkTicket(ticket);

            dos.writeUTF(response);
            dos.flush();

        } catch(IOException e) {
            System.err.println(e);
        }
    }
}
```

I LotteryServerB är *checkTicket* långsam (0-3 sek).

Break

16. Multi-Threaded Server (TCP)

LotteryServerD.java

LotteryClientBDF.java

A design for a multi-threaded server is to let each client be processed by a **new** thread, in this case instance of ClientHandler.

1. Create ServerSocket
2. Listen for connecting client
3. Create the thread that handles the request and start the thread.
4. Back to step 2

```
public LotteryServerD(int port) throws IOException {
    serverSocket = new ServerSocket(port);
    server.start();
}

public void run() {
    System.out.println("Server running");
    while(true) {
        try  {

            Socket socket = serverSocket.accept();

            new ClientHandler(socket).start();

        } catch(IOException e) {
            System.err.println(e);
        }
    }
}
```

17. Multi-Threaded Server (TCP) – ctd.

LotteryServerD.java

LotteryClientBDF.java

A design for a multithreaded server is to allow each client to be processed by a new thread, in this case the instance of ClientHandler.

1. Creates the thread and initializes it (previous slide)
2. Create streams
3. Read request
4. Manage request
5. Send Response
6. Terminate

Here, 1 Request/Response Per Connection.

```
private class ClientHandler extends Thread {
    private Socket socket;

    public ClientHandler(Socket socket) {
        this.socket = socket;
    }

    public void run() {
        String ticket,response;
        try (DataOutputStream dos =
                  new DataOutputStream( socket.getOutputStream());
             DataInputStream dis =
                  new DataInputStream( socket.getInputStream()) ) {
            ticket = dis.readUTF();
            response = getResponse(ticket);
            dos.writeUTF(response);
            dos.flush();
        } catch(IOException e) {}
        try {
            socket.close();
        } catch(Exception e) {}
    }
}
```

18. MultiThreaded server (TCP)

LotteryServerE.java

LotteryClientE.java

Many requests on same connection

Often we keep the connection open, and re-use it for more request/response cycles. (As in Asgt.2)

In the while loop, the thread reads and writes repeatedly.

- When the client closes its socket, IOException is thrown. The exception is caught after the while loop at which point the client thread ends.

(in HTTP/1 we use a Keep Alive Header header for this purpose. By default HTTP/2 and HTTP/3)

```
private class ClientHandler extends Thread {
    private Socket socket;

    public ClientHandler(Socket socket) {
        this.socket = socket;
    }

    public void run() {
        String ticket,response;
        System.out.println("Klient uppkopplad");
        try (DataOutputStream dos =
                  new DataOutputStream( socket.getOutputStream());
             DataInputStream dis =
                  new DataInputStream( socket.getInputStream())) {

            while(true) {
                ticket = dis.readUTF();
                response = getResponse(ticket);
                dos.writeUTF(response);
                dos.flush();
            }
        } catch(IOException e) {}
        try {
            socket.close();
        } catch(Exception e) {}
        System.out.println("Klient nerkopplad");
    }
}
```

19. Multi-Threaded Server (UDP) – Server class

LotteryServerC.java

LotteryClientAC.java

A design for a multithreaded server is to have each client processed by a thread, (in this case the instance of ClientHandler).

1. Create DatagramSocket and then some.
2. Listen for request
3. Create Thread That Handles Request
4. Back to step 2

```
public Server(int port) throws SocketException {
    socket = new DatagramSocket(port);
    thread.start();
}

public void run() {
    DatagramPacket packet;
    byte[] buffer = new byte[256];
    String ticket;
    System.out.println("Server running");
    while(true) {
        try {
            packet = new DatagramPacket(buffer,buffer.length);
            socket.receive(packet);

            ticket = new String(packet.getData(),0,packet.getLength());

            new ClientHandler(packet.getAddress(), packet.getPort(), ticket);

        } catch(Exception e) {
            System.err.println(e);
        }
    }
}
```

20. Multi-Threaded Server (UDP) – ClientHandler

LotteryServerC.java

LotteryClientAC.java

A design for a multithreaded server is to allow each client to be processed by one thread, in this case by the instance of ClientHandler.

1. Creates the thread and initializes it
2. Read request
3. Manage request
4. Create the response Packet
5. Send Response

(No connection to close)

This allows for blocking calls to fulfill each request (for example, disk, network, database access). There is no ”connnection” in UDP, so no ”connection re-use”.

```
private class ClientHandler extends Thread {
    private InetAddress address;
    private int port;
    private String ticket;

    public ClientHandler(InetAddress address, int port, String ticket) {
        this.address = address;
        this.port = port;
        this.ticket = ticket;
        start(); // tråden startar sig själv
    }

    public void run() {
        try {
            String response = getResponse(ticket);
            byte[] outData = response.getBytes();
            DatagramPacket packet =
                new DatagramPacket(outData, outData.length, address, port);

            socket.send(packet);

        } catch (IOException e) {}
    }
}
```

LotteryServerF.java

21. MultiThreaded server (TCP) with Thread Pool

- See Slide 13.
- There is an overhead to starting/stopping threads.
- The alternative is have a “pool” of threads.
- Tasks (implementing the Runnable interface) are sent to the thread pool, where they are run on an available thread.

- Details about using thread pools are outside the content of this course (not for exam).
- This is studied more in DA218.

22. Why is Multi-Threading advantageous?

- Blocking IO.
  - Threads can be blocked fullfilling the request:
    - Disk or Network Access
    - Waiting for Databases.
  - Theads can be blocked by connections (in TCP):
    - Waiting for a client to send all data.
    - Waiting for a client to send next request.
    - Waiting for TCP connection itself (latency if data is re-sent by the connection, or simply slow connection)
  - We don’t want to block all clients while we are waiting.
  - Multiple threads allows request handling for many clients to occur concurrently.
  - We can process more requests/clients per second; clients don’t have to wait so long.
  - Downsides:
    - Complexity! Need to implement thread safety.
    - DDoS attack – many numbers of client requests, attempting to fullfill all these concurrently can start many threads – too many and the server will crash.

.

Exam DA343A-20240612. Q21

```
public void run(int port) throws IOException {
    ServerSocket serverSocket = new ServerSocket(port);
    String id, response;
    System.out.println("Server running");

    while (!Thread.interrupted()) {
        try (Socket socket = serverSocket.accept();
             DataInputStream dis =
                     new DataInputStream(socket.getInputStream());
             DataOutputStream dos =
                     new DataOutputStream(socket.getOutputStream())) {

            id = dis.readUTF();
            response = queryDatabase(id);
            dos.writeUTF(response);
            dos.flush();
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

Titta på kodexemplet, det visar en TCP-server. Servern startas genom att anropa metoden run(). Observera att metoden queryDatabase() inte visas.

Vad är namnet på denna serverimplementering?

Skriv ett ord, på **engelska**.

Skriv in ditt svar här

Hur många trådar använder den? Hur hänger detta ihop med antalet förfrågningar?

Skriv 1-2 hela meningar.

Skriv in ditt svar här

Förklara tydligt, med hänvisning till de inblandade TCP-anslutningarna, händelseförloppet relaterade till hantering av förfrågningar från två olika klienter. Vilken data läses/skrivs till strömmarna (streams)? När öppnas och stängs TCP-anslutningarna?

Skriv en list av meningar.

Skriv in ditt svar här

Exam DA343A – 20240821 – Q30

A) Förklara skillnaden mellan en iterativ server och en flertrådad server med en trådpool?

Skriv in ditt svar här

B) Om en iterativ server kan bearbeta högst 10 begäranden per sekund, hur många kan vi förvänta oss att en flertrådad server (med en trådpool på storleken 10) ska bearbeta per sekund?

Skriv in ditt svar här

C) Motivera ditt svar till (B)

Skriv in ditt svar här

24. Async IO

Outside the course content!

- Goal is to serve **many users/requests**, with **low latency** – whilst having the **fewest/smallest servers**.
- Threads spend a lot of time waiting!
  - Waiting for new connections
  - Waiting for request to come from client
  - Waiting on data from a disk (e.g. images, static web server)
  - Waiting on a database, or another service (e.g. dynamic web server)
  - Waiting for the response to be transferrred to the client.

….none of this waiting is CPU intensive!

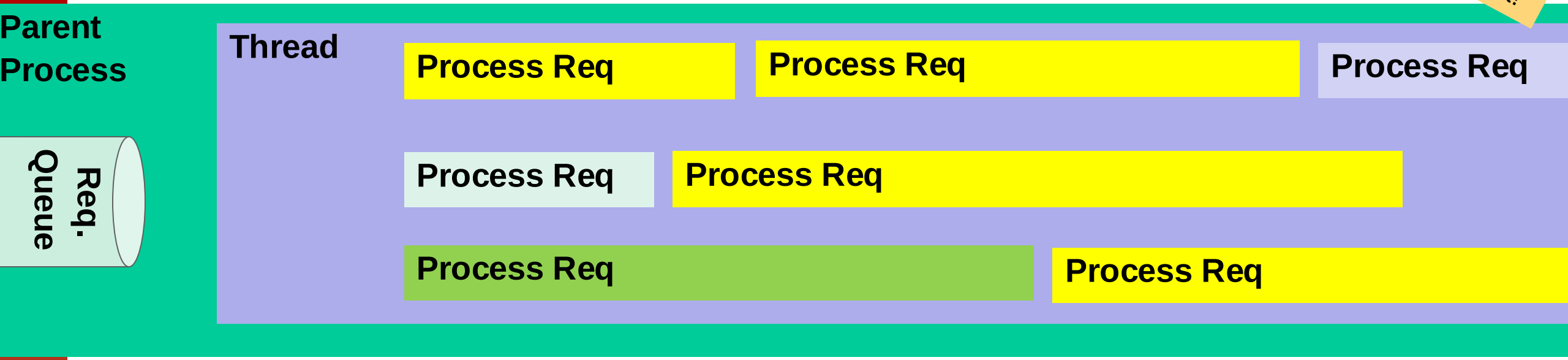
…but the IO (Input/Output) is a limiting factor.

…each thread can be idle much of the time.

…is there a better way?

25. Concurrent Server, Single Thread, Async

Outside the course content!



- **Single** Thread, **Single** process. Use of ***Asynchronous IO***, requests processed **concurrently**.
- Event-driven programming paradigm with (e.g.) ***promises***. (not procedural!)
- Ideal if we need to query many different servers (microservices, DBs) to handle a request.
- Relatively new!
- Examples: NodeJS (2009) – "javascript everywhere paradigm", also Nginx (2004).
- Advantage: no overhead from multiple threads/processes, better performance overall.

Outside the course content!

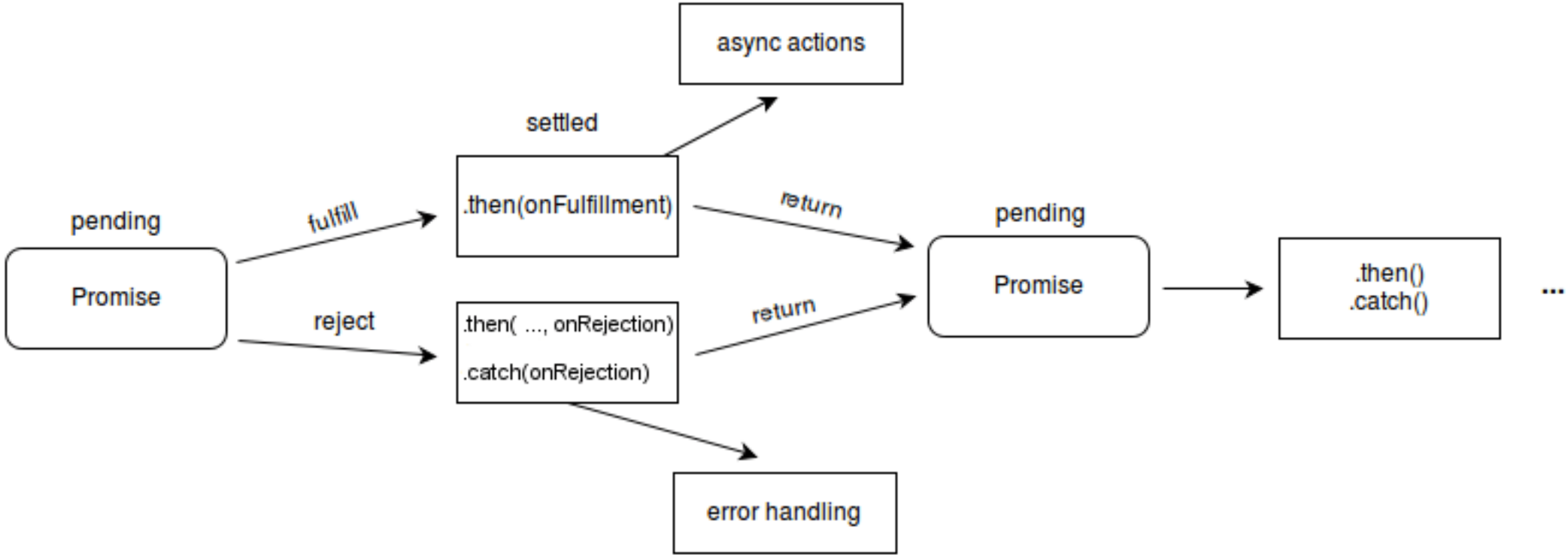
```javascript
// Example NodeJS code  - Javascript.

// Chaining promises to perform a sequence of asynchronous operations
fetchUserData(123)
    .then(user => {
        console.log('User data:', user);
        return fetchPosts(user.id); // Return a promise for the next asynchronous operation
    })
    .then(posts => {
        console.log('User posts:', posts);
        const postId = posts[0].id; // Assuming we're interested in the first post
        return fetchComments(postId); // Return a promise for the next asynchronous operation
    })
    .then(comments => {
        console.log('Comments for the first post:', comments);
    })
    .catch(error => {
        console.error('An error occurred:', error);
    });
```

26. Promise

Outside the course content!

A promise is an object representing a future operation.



https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Questions?