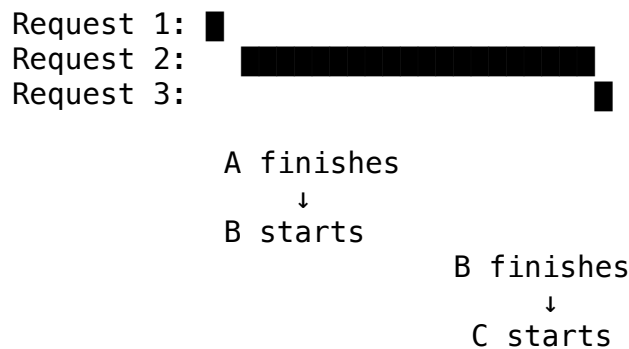


Application Layer (IV)

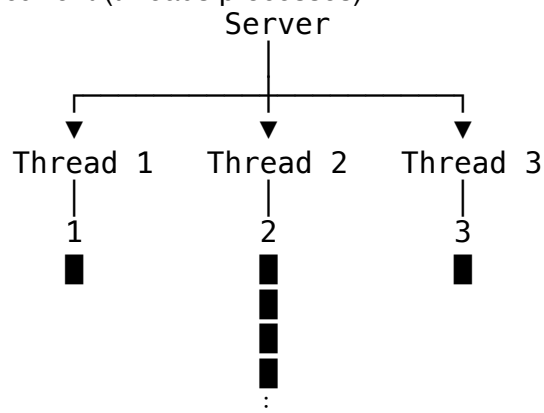
Server Architecture

- Generally speaking, the server accepts and process the incoming requests while listening at a port.
- Typically, there are two ways to process the accepted requests, sequentially and concurrently.
- The web servers in real life must accept concurrent requests. So, processing the requests currently is a more feasible way.
 - The server can allocate a process or thread to handle a request, the main process and thread of the server deals with accepting the incoming requests.
- Exercise:
 - Your server receives three requests at the same time:
 1. read a small local file (2 ms)
 2. query a remote database (2 seconds)
 3. return "Hello" (1 ms)
 - If the server processes them sequentially, when can 3 finish? If concurrently, when?
 - Sequential

Time 

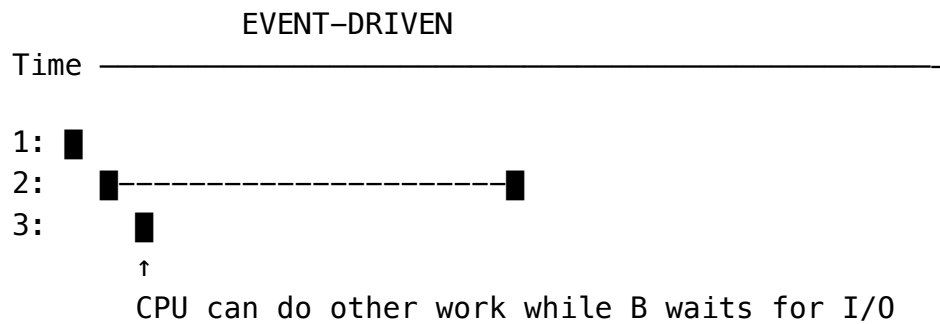


- Concurrent (threads/processes)



- If B spends most of its time waiting for network I/O, should the CPU really wait for B before handling C?

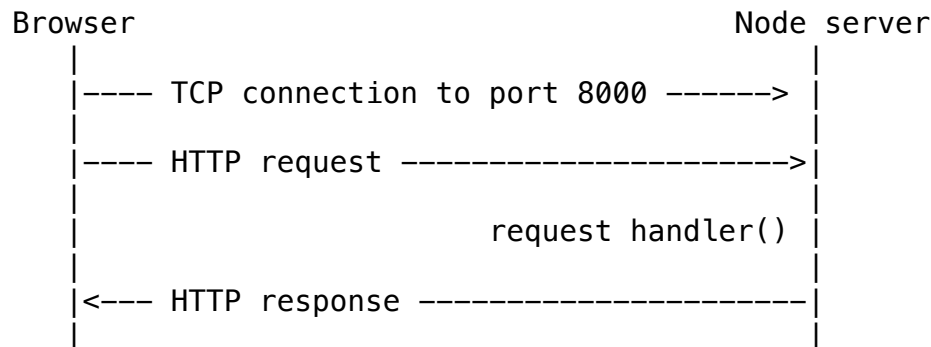
- Event-driven is another approach to handle concurrent requests.
 - The server does a **quick bit of processing** and put the request in an event loop, **scheduling a callback when** (typically) **I/O is done**.
 - Node.js (<https://nodejs.org/en/>) provides a framework to implement an event-driven server.
 - **An event loop works well for I/O-heavy workloads. CPU-intensive work must be managed carefully** because long computations can block the event loop unless moved to workers or otherwise parallelized.



One event loop can manage many
I/O operations concurrently

Node.js Example 1

- We'd like a server listening at port 8000 for incoming request, and then send a response "Hello World" back.



```

1  const http = require('http');
2
3  const PORT = 8000;
4
5  const server = http.createServer((req, res) => {
6    res.writeHead(200, {
7      'Content-Type': 'text/plain; charset=utf-8'
8    });
9
10   res.end('Hello World!\n');
11 });
12
13 server.listen(PORT, () => {
14   console.log(`Server running at http://localhost:${PORT}/`);
15 });

```

- `http.createServer((req, res) => {...})`: When an HTTP request arrives, Node gives our program a request object and response object.
 - The request object has a `url` attribute that contains the pathname and the query.
 - The response object includes the HTTP status code, the response header, and the response content (e.g., the requested file).
- `writeHead(status code, header)` send a response header to the request. **Must be called before `resp.end()`.**
- `res.end(data)` signals to the server that all of the response headers and body have been sent. **Must be called on each response.**
- Exercise:
 - **What are the four pieces of information needed to build the connection?** [`:::1 8000`, and `:::1` another ephemeral port]
- Let's understand how both sides exchange information over HTTP using Wireshark.
 - open Wireshark and listen at Loopback, filter `tcp.port==8000`
 - run `server1.js` locally
 - open a browser and type <http://localhost:8000/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	:::1	:::1	TCP	88	58994 → 8000 [SYN] Seq=2628888301 Win=65535 Len=0 MSS=16324 WS=64 TSval=3...
2	0.000249	:::1	:::1	TCP	88	8000 → 58994 [SYN, ACK] Seq=584812095 Ack=2628888302 Win=65535 Len=0 MSS=...
3	0.000292	:::1	:::1	TCP	76	58994 → 8000 [ACK] Seq=2628888302 Ack=584812096 Win=407744 Len=0 TSval=30...
4	0.000310	:::1	:::1	TCP	76	[TCP Window Update] 8000 → 58994 [ACK] Seq=584812096 Ack=2628888302 Win=4...
5	0.032829	:::1	:::1	HTTP	513	GET / HTTP/1.1
6	0.032921	:::1	:::1	TCP	76	8000 → 58994 [ACK] Seq=584812096 Ack=2628888739 Win=407360 Len=0 TSval=34...
7	0.033725	:::1	:::1	HTTP	255	HTTP/1.1 200 OK (text/plain)
8	0.033753	:::1	:::1	TCP	76	58994 → 8000 [ACK] Seq=2628888739 Ack=584812275 Win=407616 Len=0 TSval=30...
9	5.035291	:::1	:::1	TCP	76	8000 → 58994 [FIN, ACK] Seq=584812275 Ack=2628888739 Win=407360 Len=0 TSv...
10	5.035395	:::1	:::1	TCP	76	58994 → 8000 [ACK] Seq=2628888739 Ack=584812276 Win=407616 Len=0 TSval=30...
11	5.035544	:::1	:::1	TCP	76	58994 → 8000 [FIN, ACK] Seq=2628888739 Ack=584812276 Win=407616 Len=0 TSv...
12	5.035622	:::1	:::1	TCP	76	8000 → 58994 [ACK] Seq=584812276 Ack=2628888740 Win=407360 Len=0 TSval=34...

- Observe the capture, and answer the following questions.
 - What is the client's port? [58994]
 - Which HTTP version is used here? [HTTP/1.1]
 - Which packet contains the HTTP request? [GET request is the packet with GET / HTTP/1.1]
 - Which packet contains the HTTP response? [GET response is the packet with HTTP/1.1 200 OK]
- Show the HTTP response from the capture:
 - The code `resp.writeHead(200, {'Content-Type': 'text/plain'}); resp.end('Hello World\n');` would conceptually do:


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

Hello World
```

Node.js Example 2

- Suppose I type “http://localhost:8000/form.html” in the browser, what information does the browser have to send the server? [server hostname: localhost; port: 8000; path: /form.html]
- Therefore, the server has to react to the request to send the expected resources. The node server needs to inspect the path and see if the requested resource exists. If so, send the resource back to the client. Otherwise, send an error message to the client.
- Show server2.js

```
1  const http = require('http');
2  const fs = require('fs');
3
4  const PORT = 8000;
5
6  const server = http.createServer((req, res) => {
7    // req.url contains only the path/query, so provide a base URL
8    const requestURL = new URL(req.url, `http://${req.headers.host}`);
9    const pathname = requestURL.pathname;
10
11    console.log(`Request for ${pathname} received.`);
12
13    // Remove the leading "/" to get the local filename
14    const filename = pathname.substring(1);
15
16    fs.readFile(filename, (err, data) => {
17      if (err) {
18        console.error(err);
19
20        res.writeHead(404, {
21          'Content-Type': 'text/html; charset=utf-8'
22        });
23
24        res.end('<h1>404 - File Not Found</h1>');
25        return;
26      }
27
28      res.writeHead(200, {
29        'Content-Type': 'text/html; charset=utf-8'
30      });
31
32      res.end(data);
33    });
34  });
35
36  server.listen(PORT, () => {
37    console.log(`Server running at http://localhost:${PORT}/`);
38  });
39
```

- The module fs provides a variety of file system methods.
- The easiest way to read the entire file is using readFile method:

- `fs.readFile(file, [encoding], [callback]);`
 - `file` is a string indicating the path of the file to read
 - `encoding` is an optional parameter that specifies the type of encoding to read the file, such as `'ascii'`, `'utf8'`, and `'base64'`. The default is `null` if no encoding is provided.
 - `callback` is a function to call when the file has been read and the contents are ready. It has two parameters, `error` and `data`. If there is no error, `error` will be `null` and `data` will contain the file content. Otherwise, `error` will contain the error message.
- run `server2.js` and request `form1.html` and `form2.html` from web browser

```

mbp-190250:code yingli$ node node2.js
Server running at http://localhost:8000/
Request for /form.html received.

```

- The `server2.js` server can send the responses to the web browser according to the requests.
- However, the response page cannot reflect the submitted data correctly. Submitted “data” and return “test”.

- Because the `action` field of the `<form>` in `form1.html` and `form2.html` is a statical html page, which cannot reflect the data submitted by the client.
- `action` defines the response after submission

```

< > form2.html > f <form action="/response.html" method="POST">
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Form</title>
5  </head>
6  <body>
7    <form action="/response.html" method="POST">
8      <input type="text" name="q">
9      <input type="submit">
10 </body>
11 </html>

```

```

< > response.html > No Selection
1  <!DOCTYPE html>
2  <html>
3  <p>User entered: test</p>
4  </html>

```

- To make an interactive webpage, there should be a way to transmit data between the client and server back and forth, even better, without impaction to the current page display (no need to reload the page for every submission).

Node.js Example 3

- Client side AJAX
 - send and retrieve data from a server asynchronously without interfering with the display and behavior of the existing page (open form3.html using browser to illustrate).
 - It's common to use GET in AJAX unless sending a sensitive data or calling script processing data on the server (open form3.html using a text editor)

```

<script>
$( "form" ).submit( function( f ) {
    f.preventDefault();
    $.ajax({
        url: "/request",
        data: "q="+$( "#q" ).val(),
        type: "GET",
        success: function( data ) {
            $( "#result" ).html( data.result );
        },
        error: function( e ) {
            alert( "Error: "+e.status+": "+e.statusText );
        }
    });
});
</script>

```

GET /request?q=test HTTP/1.1

{"result": "server response"}

- Server side
 - Since we want to display the user entered string in the Results table, we need to fetch the string from the query part of the url, and then send it back with the HTTP response.

- Once the input is submitted, the pathname becomes “/request.” So, we can use it as a trigger to parse the query. If the pathname is not “/request,” send the required file back to the client. (open server3.js)
- Run server3.js and open form3.html on web browser.

```
if (pathname === '/request') {  
  const queryValue = requestURL.searchParams.get('q') ?? '';  
  
  console.log(`query ${queryValue} received.`);  
  |  
  resp.writeHead(200, {  
    'Content-Type': 'application/json; charset=utf-8'  
  });  
  
  resp.end(JSON.stringify({  
    result: queryValue  
  }));  
}
```

- JSON.stringify({result: queryValue}) will encapsulate the queryValue in a JSON object [{"result": "inputVal"}] and send back to the client. The object will be then processed by ajax function in the form3 to parse the JSON object and put result's value, inputVal, to the “result” field in the html file.

Node.js Example 3 Extension

- In this specific example, the server doesn't need to do anymore computation but send the query string back to the client. However, sometime, the server side needs to calculate the results and then send the results back. E.g., the server may need to search into a database or run some program to calculate the results. Then, we can let the server spawn a child process to do the calculation and then send the result back when the calculation is done.
- Using the same example, we can let the server executes an echo command that prints the stdin string to the stdout. The server echos the query, and once it's done, send the stdout string to the client side.

```

if (pathname === '/request') {
  const queryValue = requestURL.searchParams.get('q') ?? '';

  console.log(`query ${queryValue} received.`);

  // Way 2: generate a child process to produce data and send back
  const cp = spawn('echo', [
    JSON.stringify({ result: queryValue })
  ]);

  cp.stdout.on('data', (data) => {
    console.log(`child process stdout:\n${data}`);

    resp.writeHead(200, {
      'Content-Type': 'application/json; charset=utf-8'
    });

    resp.end(data);
  });
}

```

Exercise:

- After typing <https://www.example.com/search?q=networks> into a browser and press Enter. What will happen from beginning to end using as many concepts from the course we've discussed so far as possible. Here is a set of concepts. You don't necessarily have to use every concept.

DNS
 IP address
 port
 TCP
 QUIC
 HTTP
 TLS
 GET
 server
 socket
 routing
 ARP
 Ethernet/Wi-Fi
 HTTP response/request
 cookie
 cache
 CDN

