Asynchronous JavaScript

Pau Fernández Rafa Genés Jose L. Muñoz

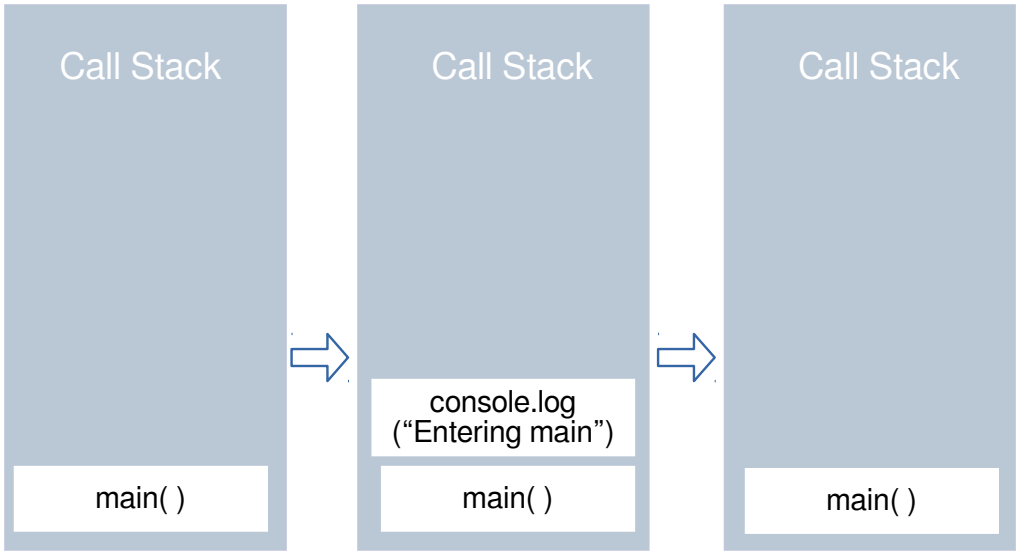
Universitat Politècnica de Catalunya (UPC)

Asynchronous Javascript

Outline

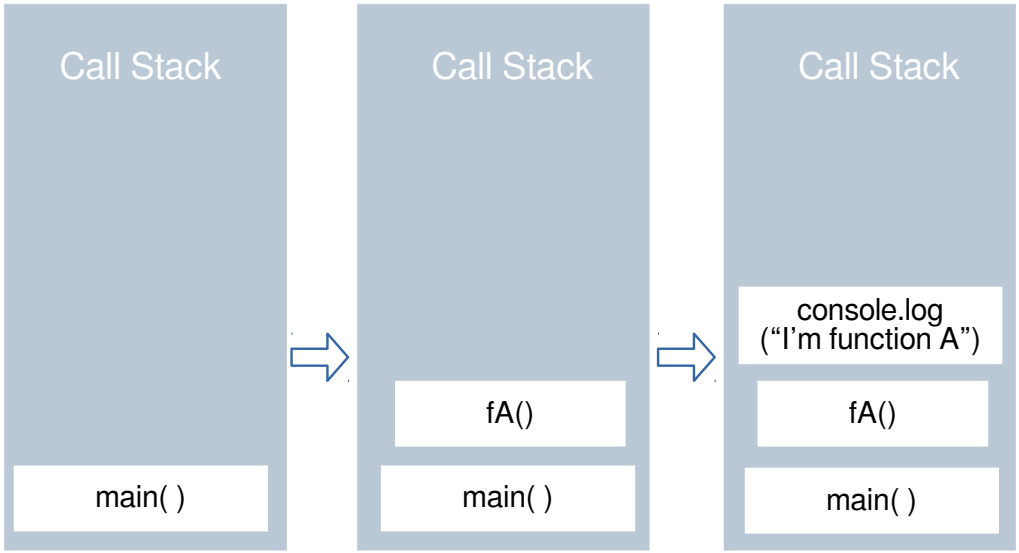
Synchronous Javascript and the Call Stack i

```
console.log("Entering main");

function fA(){
  console.log("I'm function A");
}

fA();

console.log("Exiting main");
```



Synchronous Javascript and the Call Stack ii

```
console.log("Entering main");

function fA(){
  console.log("I'm function A");
}

fA();

console.log("Exiting main");
```

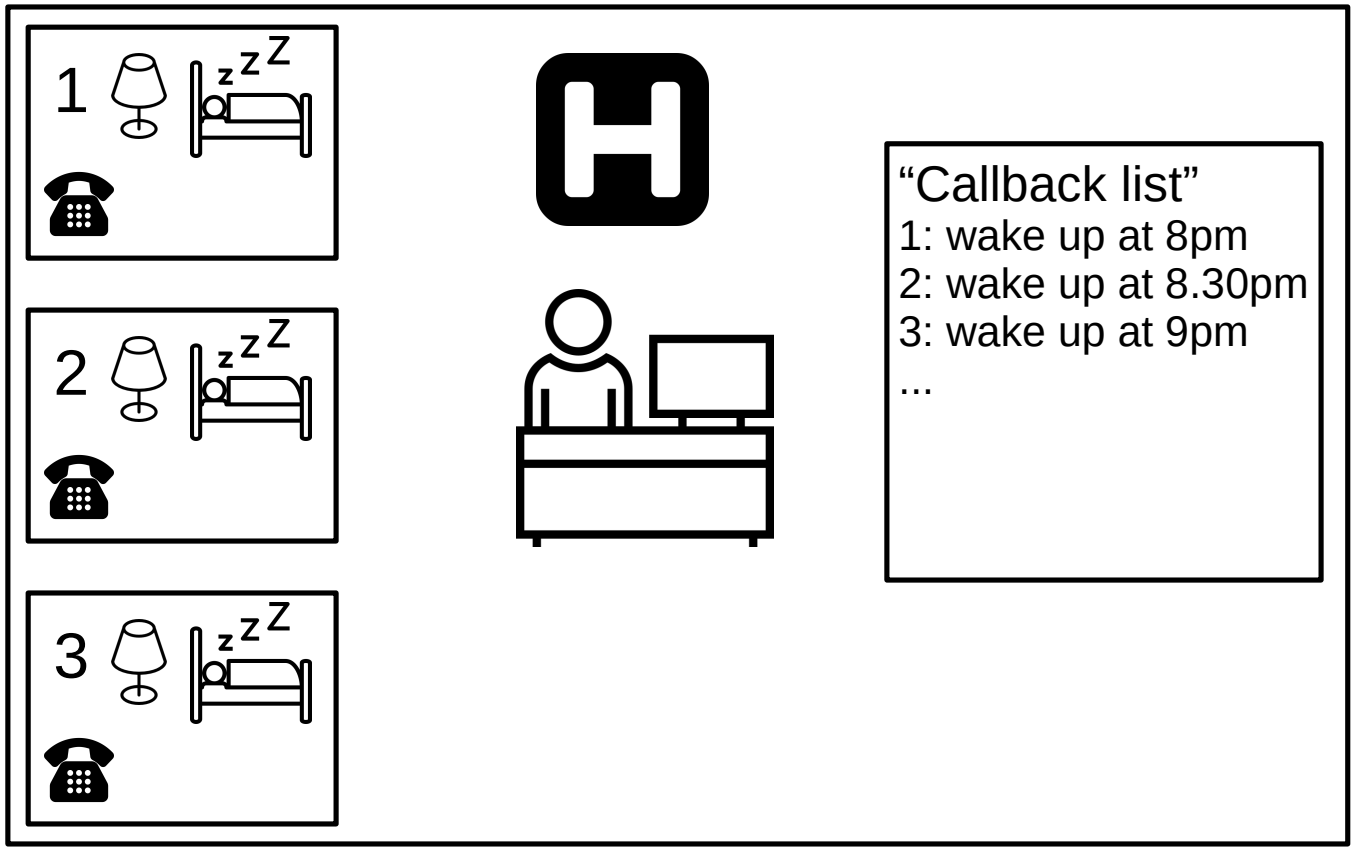


Returning values in synchronous Javascript

In synchronous code, we can read and use the return value of functions:

```
function sum(a, b) {
  return a + b;
}

const value = sum(2, 3);

console.log(value); // needs to wait until sum is executed
```

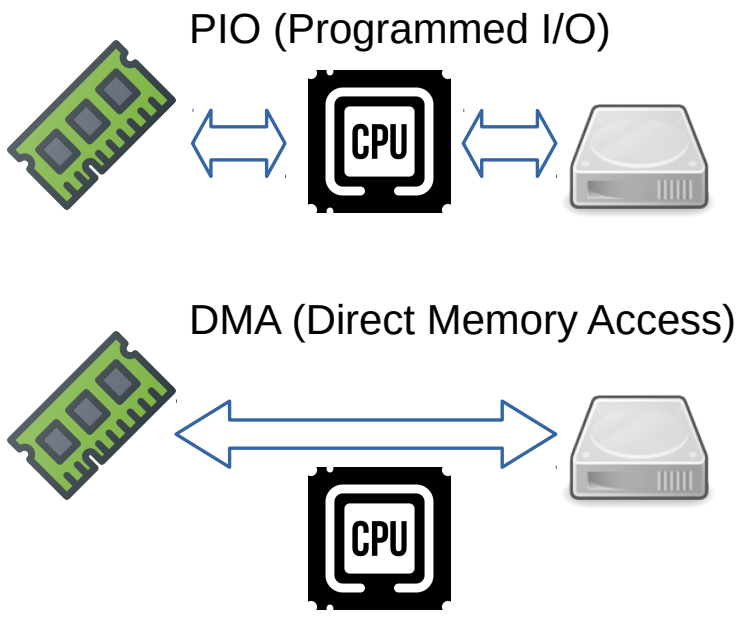
Asynchronous Javascript: Motivation Toy Example



Asynchronous Javascript: Real Motivation

Javascript is built to run over asynchronous, non-blocking, event-based & "single-threaded[1]" platforms.

- Although DMA technologies may periodically steal cycles from the CPU, data can move much faster when the CPU does not go through every byte.
- With sync code, our program makes the CPU **wait doing nothing**.



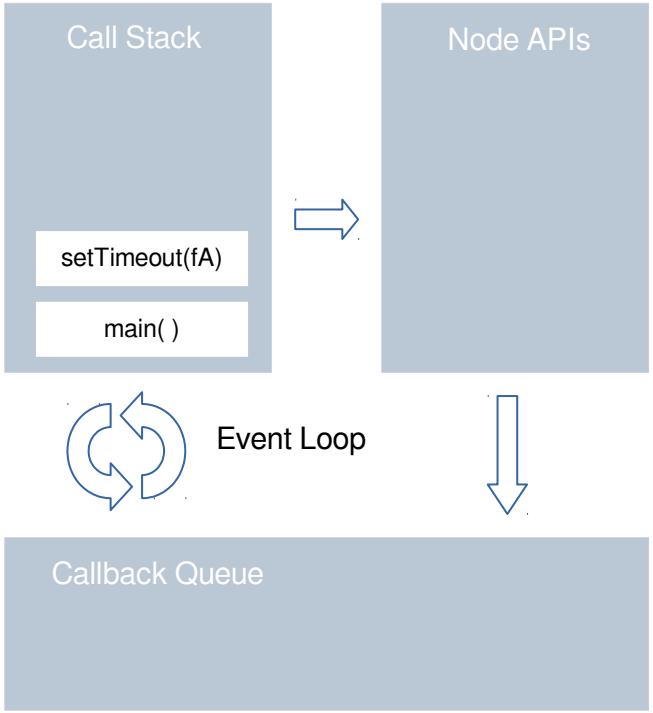
[1] This is not exactly correct as we will see later.

```
console.log("Entering main");

function fA() {
  console.log("I'm function A");
}

function fB() {
  console.log("I'm function B");
}

setTimeout(fA, 2000);
setTimeout(fB, 0);

console.log("Exiting main");
```

Which do you think that is the output of the previous JS code?

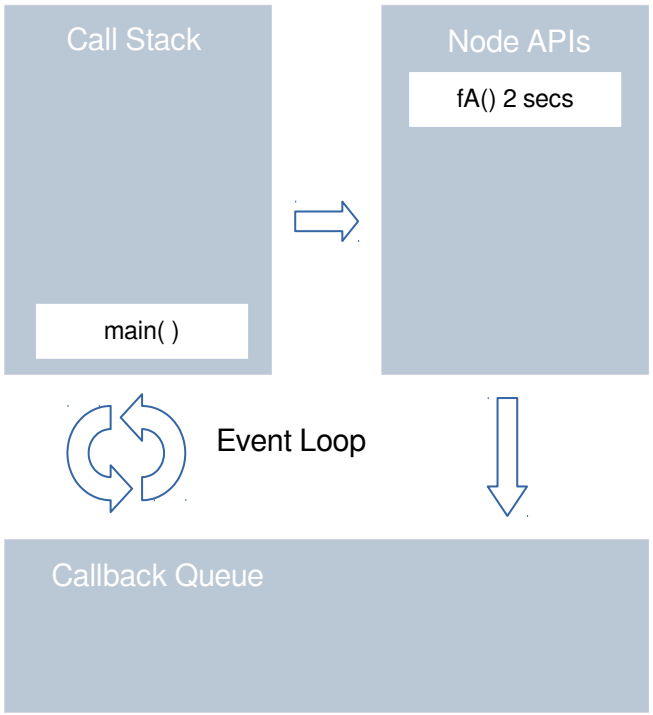
Asynchronous JS ii

```
console.log("Entering main");

function fA() {
  console.log("I'm function A");
}

function fB() {
  console.log("I'm function B");
}

setTimeout(fA, 2000);
setTimeout(fB, 0);

console.log("Exiting main");
```



Asynchronous JS iii

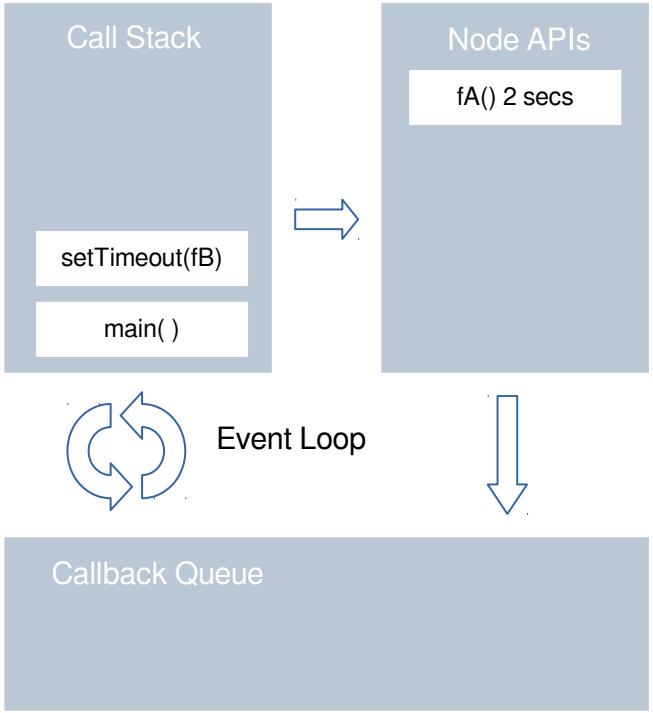
```
console.log("Entering main");

function fA() {
  console.log("I'm function A");
}

function fB() {
  console.log("I'm function B");
}

setTimeout(fA, 2000);
setTimeout(fB, 0);

console.log("Exiting main");
```



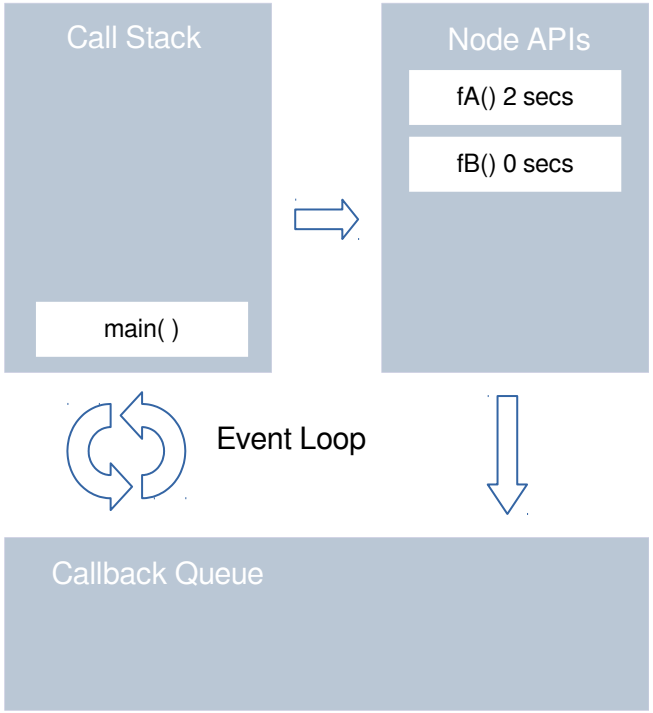
fA goes to wait 2 seconds in the **Node APIs**

Asynchronous JS iv

```
console.log("Entering main");

function fA() {
  console.log("I'm function A");
}

function fB() {
  console.log("I'm function B");
}

setTimeout(fA, 2000);
setTimeout(fB, 0);

console.log("Exiting main");
```

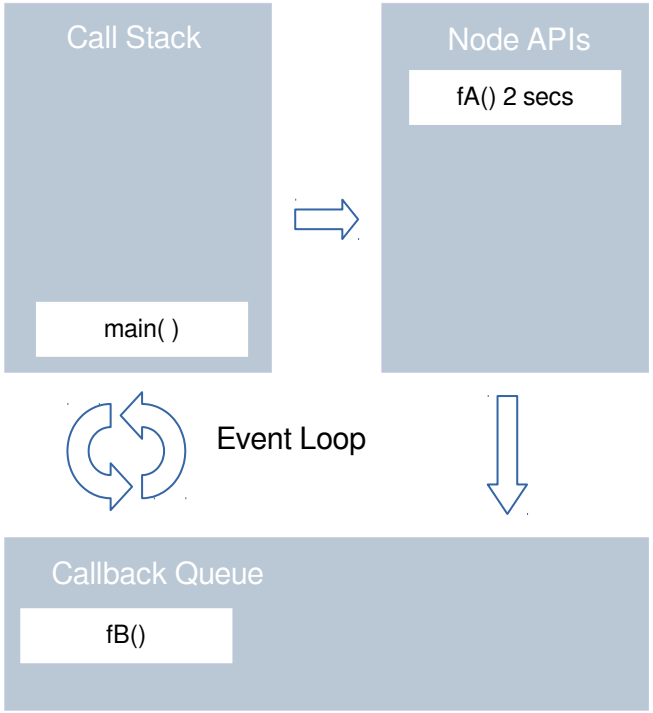


```
console.log("Entering main");

function fA() {
  console.log("I'm function A");
}

function fB() {
  console.log("I'm function B");
}

setTimeout(fA, 2000);
setTimeout(fB, 0);

console.log("Exiting main");
```

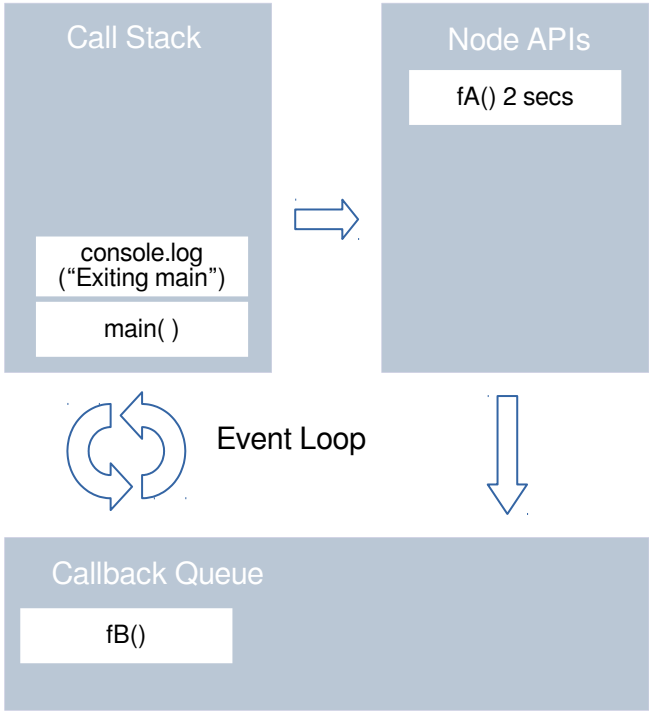


Asynchronous JS vi

```
console.log("Entering main");

function fA() {
  console.log("I'm function A");
}

function fB() {
  console.log("I'm function B");
}

setTimeout(fA, 2000);
setTimeout(fB, 0);

console.log("Exiting main");
```

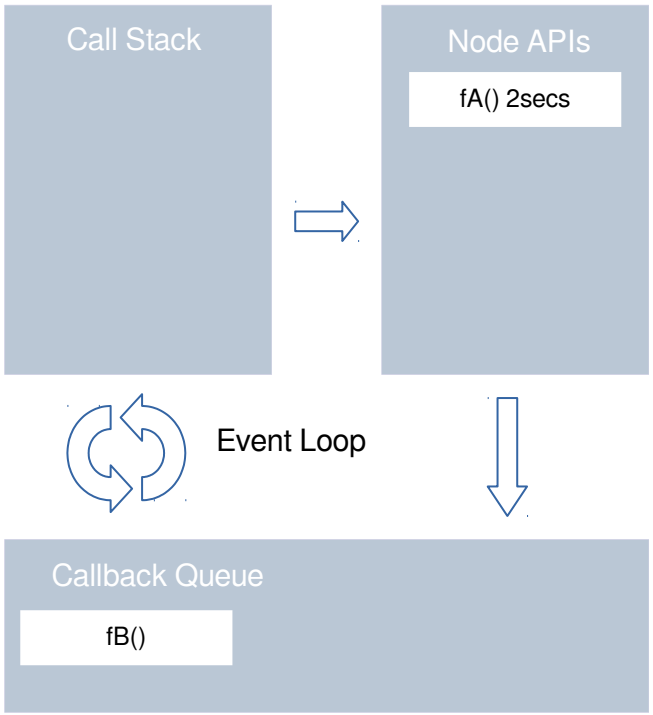


```
console.log("Entering main");

function fA() {
  console.log("I'm function A");
}

function fB() {
  console.log("I'm function B");
}

setTimeout(fA, 2000);
setTimeout(fB, 0);

console.log("Exiting main");
```



Asynchronous JS viii

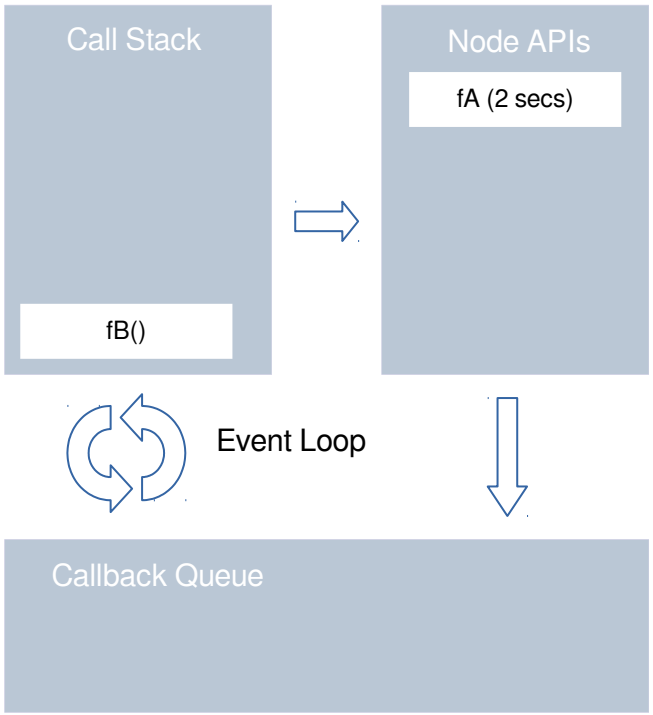
```
console.log("Entering main");

function fA() {
  console.log("I'm function A");
}

function fB() {
  console.log("I'm function B");
}

setTimeout(fA, 2000);
setTimeout(fB, 0);

console.log("Exiting main");
```



The `main()` ends

Asynchronous JS ix

```
console.log("Entering main");

function fA() {
  console.log("I'm function A");
}

function fB() {
  console.log("I'm function B");
}

setTimeout(fA, 2000);
setTimeout(fB, 0);

console.log("Exiting main");
```



Now we work driven by **events**, the program will end when no more events.

Call Stack, node APIs, Callback Queue, Event Loop

In JS platforms, which are asynchronous, we have 4 main parts:

1. A **call stack**.
    - Contains stack frames (of waiting functions).
    - If we call a function we push a frame.
    - If we return we pop a frame.
    - Synchronous code only uses the stack.
2. **node APIs**
    - The **node API** stores functions to be called for certain events.
    - These functions are referred to as **callback**s, or event handlers.
    - The call stack can proceed even if there are callbacks registered in the node API.
3. A **callback queue**
    - A queue storing the callback functions.
4. An **event loop**
    - A process that extracts callbacks to be executed.

Accessing Objects from Callbacks

In the following code the callbacks access and modify an object:

```
function fA() {
  myCounter.counter++;
  console.log(`I'm function A, counter is ${myCounter.counter}`);
}
function fB() {
  myCounter.counter++;
  console.log(`I'm function B, counter is ${myCounter.counter}`);
}

let myCounter = { counter: 1 };

console.log("Entering main");
setTimeout(fA, 2000);
setTimeout(fB, 0);
console.log("Exiting main");
```

```
Entering main
Exiting main
I'm function B, counter is 2
I'm function A, counter is 3
```

Async Functions with Arguments

```
let getUser = id => {
  let user = {
    id: id,
    name: 'joe'
  };

  setTimeout(() => {
    console.log(user);
  }, 3000);
};
getUser(31);
```

How to receive a return from an async function?

```
const user = getUser(31); // Does not work!!!
```

Pass a Callback

To use results from async functions we use a callback function:

```
let getUser = (id, callback) => {
  let user = {
    id: id,
    name: 'joe'
  };

  setTimeout(() => {
    callback(user);
  }, 3000);
};


getUser(31, userObject => console.log(userObject) );
```

Example: Synchronous vs. Asynchronous i

A synchronous function blocks while it is waiting for a resource (but it is not computing anything):

```
// before
const search = "pragma";
const filename = "codefile.cpp";

// blocking (wait for the disk to find the data, super-slow)
const lines = readAllLinesOfFileSync(filename);

// after
lines.filter(line => line.includes(search)).forEach((line, i) => {
   console.log(i + ": " + line);
})

// do more stuff...
```

Since the processor is waiting for the disk, it sits idle doing nothing, because the flow of control as is implemented has to wait until the function returns.

Example: Synchronous vs. Asynchronous ii

To avoid this, we use an asynchronous function passing a callback to it:

```
// before
const search = "pragma";
const filename = "codefile.cpp";

readAllLinesOfFileASync(filename, (lines) => {
   // after
   lines.filter(line => line.includes(search)).forEach(line => {
      console.log(index + ": " + line);
   })
})

// do more stuff...
```

Now, we program what to do after lines are available, the program **does not block**, and we can continue with more processing after the call to `readAllLinesOfFileASync`.

Exercises!

1. Implement the following function in a async version in which you can pass a callback:

   ```
   const addInts = (a, b) => a + b;
   ```

   and then call it once with 2 and 3.
2. Implement the function **`readAllLinesOfFileAsync`** used in the "Callbacks" slide. (It is forbidden to use any synchronous method.)
3. Implement a **`grep`** function that receives a **`filename`**, a **`search`** string and a *callback* which will be called with every line of the file that contains the **`search`**. (Maybe use **`readAllLinesOfFileAsync`**?)

Outline

Request a REST HTTP API i

- We will use etherscan API, **https://etherscan.io**
- Register, login and obtain an API token.
- Request from command line (put the following in a single line):

```
$ curl -i -X GET "https://api.etherscan.io/api?
module=account&action=balance&address=0xddbd2b932c763ba5b1b7ae3b362eac3e8d40121a&
tag=latest&apikey=AIX5NGPXXVWI66MCYCZ4MYZHBJBQ6KPSN4"
```

- Where:
  -i: include protocol headers in the output
  -X: HTTP method

The response:

```
HTTP/1.1 200 OK
Cache-Control: private
Content-Type: application/json; charset=utf-8
Server: Microsoft-IIS/10.0
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: Content-Type
Access-Control-Allow-Methods: GET, POST, OPTIONS
X-Frame-Options: SAMEORIGIN
Date: Mon, 09 Apr 2018 09:41:07 GMT
Content-Length: 64

{"status":"1","message":"OK","result":"40807168564070000000000"}
```

40807 Ethers

request (Ask APIs) i

```
const request = require('request');

const address = "0xddbd2b932c763ba5b1b7ae3b362eac3e8d40121a";
const API_KEY = "AIX5NGPXXVWI66MCYCZ4MYZHBJBQ6KPSN4";
const BASE ="https://api.etherscan.io/api?module=account&action=balance&tag=latest"
const URL = `${BASE}&address=${address}&apikey=${API_KEY}`;

request({ url: URL, json: true }, (err, res) => {
    console.error(`Error: ${JSON.stringify(err)}`);
    console.log(`Body: ${JSON.stringify(res)}`);
});
```

With ”json: true” the body is converted to an object (no need to parse)

request (Ask APIs) ii

In fact, we can use a third parameter in our callback[2]:

```
const request = require('request');

const address = "0xddbd2b932c763ba5b1b7ae3b362eac3e8d40121a";
const API_KEY = "AIX5NGPXXVWI66MCYCZ4MYZHBJBQ6KPSN4";
const BASE ="https://api.etherscan.io/api?module=account&action=balance&tag=latest"
const URL = `${BASE}&address=${address}&apikey=${API_KEY}`;

request({ url: URL, json: true }, (err, res, body) => {
    console.error(`Error: ${JSON.stringify(err)}`);
    console.log(parseInt(body.result,10)/10**18);
});
```

[2] You will have to check documentation to know the arguments of each async lib.

Using Previous Results

In sync programming we can use previous results as follows:

```
const r1 = f1(x1, x2);
const r2 = f2(y1, y2, r1);
f3(z1, z2, r2);

// Or
f3(z1, z2, f2(y1, y2, f1(x1, x2)));
// More clearly written (when having many functions):
f3(z1, z2,
   f2(y1, y2,
      f1(x1, x2)
   )
);
```

Calling an Async Function

- Calling an asynchronous function:

```
f(x1, x2, ..., (err, res) => { /* logic */ });
```

- Which variables are available to `callback` (i.e. which is its scope)?

```
const a = 1;

function f(x1, x2, callback) {  // f is an async function
    setTimeout(() => callback(null, {x1, x2}),0);
}

function g(y1, y2) {  // g uses the async function
    const c1 = (err, res) => console.log(`Sum: ${res.x1+res.x2}`);
    const c2 = (err, res) => console.log(`Multiplication: ${res.x1*res.x2}`);
    f(y1, y2, c1);
    f(y1, y2, c2);
}

g(2,3);
```

- `a`, `y1`, `y2`, `err` and `res` are available to the callback (if it does not shadow them), however `x1`, `x2` are NOT available to the callback.
- To use `x1` and `x2`, we have to pass these values to the callback.

Chaining Callbacks

- To provide order in callbacks, we chain them:

```
func1(args1..., (err1, res1) => {
  /* we can do logic with err1 and res1 */
  func2(args2..., (err2, res2) => {
    /* we can do logic with err1, res1, err2 and res2 */
    func3(args3..., (err3, res3) => {
      /* we can do logic with err1, res1, err2, res2, err3 and
    })
  })
})
```

- Notice that chaining callbacks is not so nice, this is called by many the **"callback hell"** or the **"callback pyramid"**.
- Promises are the ES6 syntax to avoid the callback hell.

Chaining Callbacks Example

For example, we need to chain callback to get the balance of the account in Euros.

We can check the ETH-EUR rate with the following API:

```
$ curl "https://min-api.cryptocompare.com/data/price?fsym=ETH&tsyms=EUR"
{"EUR":123.44}
```

The callback-based code to get the balance in EUR is the following:

```
const request = require('request');
const address = "0xddbd2b932c763ba5b1b7ae3b362eac3e8d40121a";
const API_KEY = "AIX5NGPXXVWI66MCYCZ4MYZHBJBQ6KPSN4";
const BASE ="https://api.etherscan.io/api?module=account&action=balance&tag=latest"
const URL = `${BASE}&address=${address}&apikey=${API_KEY}`;
const URL2 = "https://min-api.cryptocompare.com/data/price?fsym=ETH&tsyms=EUR"

request({ url: URL, json: true}, (err1, res1) => {
  request({ url: URL2, json: true}, (err2, res2) => {
    console.log(parseInt(res1.body.result,10)/10**18*res2.body.EUR);
  })
});
```

Outline

- With callbacks, the **API between the asynchronous function consumer and the producer is a callback function**:
  - The function has to check its arguments to distinguish between error and success.
  - The code is mixing situations which makes it dirty:

```
(err, res) => {
  if (err != null){ /* handle sucess */ }
  else { /* handle error */ }
}
```

- With Promises the consumer sets:
  - A function for **error**.
  - A function for **success**.
- **How is this done?** We are not going to pass a callback as a parameter any more... we will use an **object** as API between producer and consumer.

Consumer Code & Promise Object

- The ”producer” (asynchronous library function) returns an **object** that allows the consumer to set the error and success functions.
- Example of consumer code:

```
// Consumer of the Promise
const p = myAsyncFuncPromise();
p.then( result => console.log(result), error => console.error(error));
```

- `myAsyncFuncPromise()` is an asynchronous function that returns a ”Promise” which is an object with a `.then` method.
- We can write the previous code directly as follows:

```
myAsyncFuncPromise()
  .then( result => console.log(result), error => console.error(error));
```

- Most of the time you will write **consumer code**.

Producer Code (Asynchronous Function)

- An asynchronous function is a function that returns a Promise:

```
let positiveAddAsync = function(a,b) {
  return Promise.resolve(7);
  // return Promise.reject(7);
};

positiveAddAsync(2,5)
  .then(result => console.log("ok", result), error => console.error("ko", error));
```

- In the previous code, the promise is either resolved or rejected.

  But, **how we implement logic to decide about to resolve or reject?**

- We have to **create a Promise object** using the Promise class (which is provided by Javascript):

```
let positiveAddAsync = function(a,b) {
  return new Promise(aFunction);
};
```

- The function let's us define the logic!

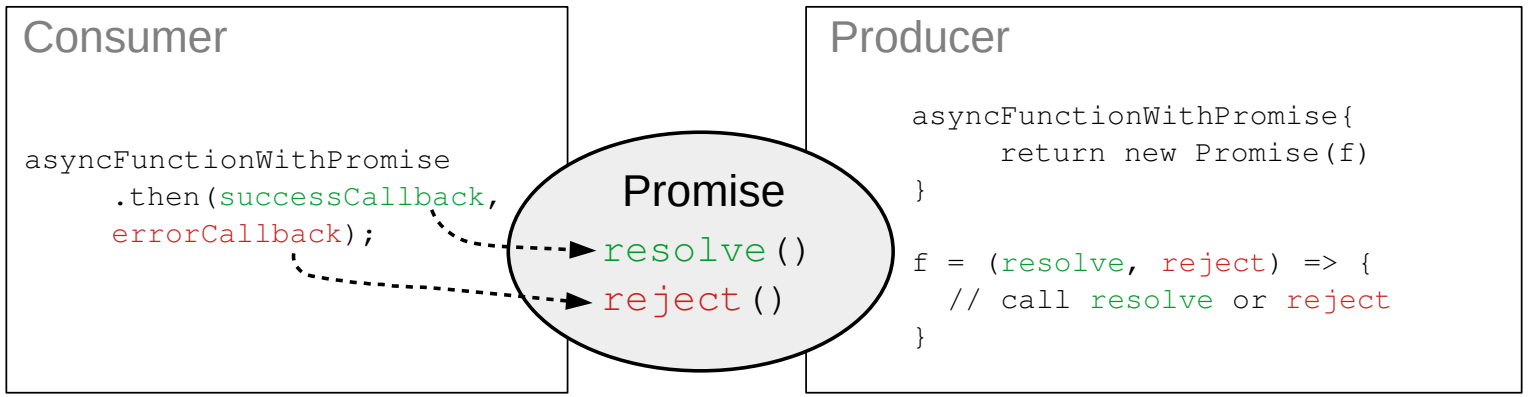
Producer Code Example

- Now, we put some logic for **positiveAddAsync**:

```
let positiveAddAsync = function(a,b) {
  return new Promise ( (resolve, reject ) => {
      setTimeout( () => {
          if ((a > 0) && (b > 0)){
              resolve(a+b);
          } else {
              reject('both numbers must be positive');
          }
      }, 2000);
  });
};

positiveAddAsync(2, 3)
.then(result => console.log("ok", result), error => console.error("ko", error));

positiveAddAsync(2, -3)
.then(result => console.log("ok", result), error => console.error("ko", error));
```

Summary of the Promise Object

- The Promise is an object that serves as a link between:
  - **Producing** code (executor).
  - **Consuming** functions which want to receive the result/error.



- The Promises mechanism is "syntactic sugar[3]" over callbacks using the Promise object.

[3] Means another way of writing things without introducing deep changes.

Promises vs. Callbacks

1. A first difference:
   - We do not need to check parameters of callbacks to distinguish between errors and normal results.
   - This is done with different functions, which makes code cleaner.

2. A second big difference:
   - We can call only once reject or resolve to settle the Promise.
   - With callbacks nothing prevents you to call the callback several times.

But where Promises really shine is when we chain them!

Chaining Promises and `.catch`

- The `.then` not only let us set the handlers for error and success.
- It always **makes the handler to return another Promise object**:
  1. A handler can explicitly return a Promise.
  2. But if the a handler returns just a value, "`.then`" implicitly converts this return to a settled Promise with this value as result.

- This also allows us to use the "`.catch`" of Promise objects to set the handler for any error:

```
myAsyncFuncPromise()
    .then(result => console.log(result))
    .catch(error => console.error(error));
```

- In fact, this is the typical way of setting error handlers!

A Promise Chain (Multiple Processing for an Event)

- Example of a chain of (implicit) Promises:

```
let myPromise = new Promise( (resolve, reject) => {
  setTimeout( () => resolve(2), 2000);
});

myPromise
 .then(result => result * 2) // handler returns a settled Promise with value 4
 .then(result => result * 2) // handler returns a settled Promise with value 8
 .then(result => result * 2) // handler returns a settled Promise with value 16
 .then(result => console.log(result)); // logs 16
```

- Note that:
  1. we can apply different functions to the event in a certain order.
  2. We cleverly **avoided the callback pyramid!**

Including Explicit Promises in the Chain

We can return explicit Promises in chains:

```
let myPromise = new Promise((resolve,reject) => {
    setTimeout(() => {resolve(2);},2000);
});

myPromise.then(result => result*2)
         .then(result => { console.log(result); // logs 4 after 2 secs approx
                           return (result*2)} )
         .then(result => {
             return new Promise( (resolve, reject) => {
                setTimeout( () => {resolve(result*2);},2000);
             });
         })
         .then(result => console.log(result)); // logs 16 after 4 secs approx
```

Variables of other Handlers are not Available

Note that we cannot use arguments of previous functions later in the chain:

```
let myPromise = new Promise( (resolve, reject) => {
  setTimeout( () => resolve(2), 2000);
});

myPromise
 .then(result1 => result1 * 2) // handler returns a settled Promise with value 4
 .then(result2 => result2 * 2) // handler returns a settled Promise with value 8
 .then(result3 => console.log(result2)); // Error
```

We only have the result of the previous handler (this is the way to pass information down the chain).

Errors in Promises Chains

- Asynchronous actions may sometimes fail:
  - In case of an error, the corresponding Promise becomes rejected.
  - Promise chaining is great at that aspect.
  - When a Promise rejects, the control jumps to the closest rejection handler down the chain (.catch).
- In the next example, we will reject in the chain:

```
let myPromise = new Promise((resolve, reject) => { setTimeout(() => resolve(2), 2000); });

myPromise.then(result => result*2)
         .then(result => { console.log(result);
                           return (result*2)})
         .then((result) => {
             return new Promise((resolve, reject) => {
                setTimeout( () => reject(`reject with ${result*2}`), 2000);
             });
         })
         .then(result => result*2)
         .then(result => console.log(result))
         .catch(error => console.error(error));

// Output:
4
reject with 16
```

"Invisible" `try ... catch`

- The code of the executor and Promise handlers have an "invisible" `try ... catch` around them.
- If an error happens, **it gets caught and treated as a rejection**:

```
myPromise.then(result => result*2)
         .then(result => {console.log(result);
                          return (result*2)})
         .then(result => {throw new Error(`Reject with ${result*2}`)})
         .then(result => result*2)
         .then(result => console.log(result))
         .catch(error => console.error(error));
```

- After the error is handled, we can continue chaining:

```
myPromise.then(result => result*2)
         .then(result => { console.log(result);
                           return (result*2)})
         .then(result => { throw new Error(`Reject with ${result*2}`) })
         .then(result => result*2)
         .then(result => console.log(result))
         .catch(error => { console.error(error);
                           return 2;})
         .then(result => console.log(result))
```

Promise Static Methods: resolve & reject

There are 5 static methods in the Promise class.

1. **Promise.resolve**

   ```
   let promise = Promise.resolve(value);
   ```

   Returns a resolved Promise with the given value, same as:

   ```
   let promise = new Promise(resolve => resolve(value));
   ```

2. **Promise.reject**

   ```
   let promise = Promise.reject(error);
   ```

   Creates a rejected Promise with the error, same as:

   ```
   let promise = new Promise((resolve, reject) => reject(error));
   ```

The previous are rarely used (the more important are the following).

Promise Static Methods: Promise.all i

3. Promise.all

- **Informally**: multiple events and a single processing!
- **Technically**: the method to run many Promises in parallel and wait till **all of them are ready**:

```
let promise = Promise.all(iterable);
```

- It takes an iterable object with Promises, technically it can be any iterable, but usually it's an array, and returns a new Promise.
- The new Promise resolves with when all of them are settled and has an array of their results.

Promise Static Methods: Promise.all ii

- For instance, the Promise.all below settles after 3 seconds, and then its result is an array [1, 2, 3]:

```
Promise.all([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 3000)), // 1
  new Promise((resolve, reject) => setTimeout(() => resolve(2), 2000)), // 2
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 1000))  // 3
  ])
   .then( result => console.log(result) ); // 1,2,3 when Promises are ready
                                           // each Promise contributes to
                                           // an array member
```

- Note that the relative order is the same.
- Even though the first Promise takes the longest time to resolve, it is still first in the array of results.
- `Promise.all` rejects as a whole if any promise rejects.

4. Promise.race

Waits for the first result (or error), and goes on with it.

```
let promise = Promise.race(iterable);
```

For instance, here the result will be 1:

```
Promise.race([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() =>
                                   reject(new Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
  ])
  .then(console.log); // 1
```

So, the first result/error becomes the result of the whole Promise.race.

After the first settled Promise “wins the race”, all further results/errors are ignored.

Example: HTTP Request with Promises

- We saw the package ”request”, which uses callbacks.
- We can wrap its functionality intro Promises.
- The native package that uses Promises is ”fetch”.
- However, fetch does not support timeouts.
- A complete and widely used package that supports Promises and timeouts is **”axios”**.

```
const axios = require("axios");

const address = "0xddbd2b932c763ba5b1b7ae3b362eac3e8d40121a";
const API_KEY = "AIX5NGPXXVWI66MCYCZ4MYZHBJBQ6KPSN4";
const BASE ="https://api.etherscan.io/api?module=account&action=balance&tag=latest";
const URL = `${BASE}&address=${address}&apikey=${API_KEY}`;

axios.get(URL)
  .then( res => console.log(parseInt(res.data.result,10)/10**18))
  .catch( err => console.error(`${err}`));
```

Example: Multiple HTTP Requests with Promises

- Now, we want to get the balance in Euros (double query).
- With a single Promise chain, we would need to access a previous value in the chain which is not directly possible[4].
- An elegant solution is to create different Promises (not chained) which are waited with `Promise.all`:

```
const axios = require("axios");
const address = "0xddbd2b932c763ba5b1b7ae3b362eac3e8d40121a";
const API_KEY = "AIX5NGPXXVWI66MCYCZ4MYZHBJBQ6KPSN4";
const BASE ="https://api.etherscan.io/api?module=account&action=balance&tag=latest";
const URL = `${BASE}&address=${address}&apikey=${API_KEY}`;
const URL2 = "https://min-api.cryptocompare.com/data/price?fsym=ETH&tsyms=EUR";

Promise.all([axios.get(URL), axios.get(URL2)])
  .then( ([balance, rate ]) =>
    console.log(parseInt(balance.data.result,10)/10**18*rate.data.EUR));
```

[4] The async/await syntax facilitates a lot this feature of using previous values of promises.

New Promise Static Method: Promise.allSettled

5. `Promise.allSettled` (is a recent addition to the language)

   - `Promise.all` rejects if any promise rejects, it is ”all or nothing”.
   - This is not good when you want to proceed with available data:

```
Promise.all([
  fetch('/template.html'),
  fetch('/style.css'),
  fetch('/data.json')
]).then(render); // render method needs results of all fetches
```

   - `Promise.allSettled` just waits for all promises to settle, regardless of the result.
   - The array from `Promise.allSettled` will be in the form:

```
[ { status:"fulfilled", value:result }, { status:"rejected", reason:error }, ... ]
```

Example Promise.allSettled

Example: fetch info about multiple users (even if one request fails, we're still interested in the others).

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://no-such-url'
];

Promise.allSettled(urls.map(url => fetch(url)))
  .then( results => {
    results.forEach( (result, num) => {
      if (result.status == "fulfilled") {
        console.log(`${urls[num]}: ${result.value.status}`);
      }
      if (result.status == "rejected") {
        console.log(`${urls[num]}: ${result.reason}`);
      }
    });
  });
```

Outline

async

- This is a special ES6 syntax to work with Promises in a more comfort fashion (i.e. ”syntactic sugar” over Promises).
- Async functions:
  - The word ”async” before a function means that the function always returns a Promise.
  - If the code has return <non-promise> in it, then JavaScript automatically wraps it into a resolved Promise with that value.
- Example:

```
async function f() {
  return 1;
}

f().then(console.log); // 1
```

- We could explicitly return a Promise, that would be the same:

```
async function f() {
  return Promise.resolve(1);
}

f().then(console.log); // 1
```

await

- `await` can be used (only) inside async functions.
- It makes JavaScript wait until that Promise settles and returns its result.
- Here's example with a Promise that resolves in 1 second:

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });

  let result = await promise; // wait till the Promise resolves

  console.log(result);
}

f();
```

- `await` is a more elegant syntax of getting the Promise result than `promise.then`, easier to read and write.

Remarks about **await**

Important notes:

1. **await can't be used in regular functions** (only in **async** functions).
2. **await** does **not work in the top-level** code.
   For this, you can create a wrapping async function that awaits:

   ```
   // syntax error in top-level code
   let response = await axios.get('/article/promise-chaining/user.json');
   let user = await response.json();
   ```

3. If you have several **await**s that set several variables, you will have **these variables available in the whole async function**.

Errors i

- In real situations the Promise may take some time before it rejects.
- So **await** will wait, and then an error will be thrown.
- We can **catch** that error using **try/catch**, the same way as a regular throw:

```
async function f() {

  try {
    let response = await axios.get('http://no-such-url');
  } catch(err) {
    console.error(err);
  }
}

f();
```

- In case of a rejection, it throws the error, just as if there was a throw statement at that line, like this:

```
async function f() {
  await Promise.reject(new Error("Whoops!"));
}
```

Errors ii

- Is the same as:

```
async function f() {
  throw new Error("Whoops!");
}
```

- We can also wrap multiple lines:

```
async function f() {

  try {
    let response = await axios.get('/no-user-here');
    let user = await response.json();
  } catch(err) {
    // catches errors both in fetch and response.json
    console.error(err);
  }
}

f();
```

Errors iii

- If we don't have a **try/catch** inside an async function and the Promise awaited becomes rejected, we can append a **.catch** in the outer function to handle this:

```
async function f() {
  let response = await axios.get('http://no-such-url');
}

// f() becomes a rejected Promise
f().catch(console.error);
```

- If we forget to add **.catch** here, then we get an unhandled Promise error (and we can see it in the console).

Awaiting Multiple Promises i

- When we need to wait for multiple Promises, we can simply use multiple awaits:

```
const getSeveralValues = async () => {
 try {
   const res1 = await axios.get(URL1);
   const res2 = await axios.get(URL2);

   // Do logic with res1 and res2
   console.log(res1.data, res2.data);
 } catch (error){
   console.error(`${error}`);
   }
 }

getSeveralValues();
```

- Notice that we are waiting first for the settling of `res1` and then we proceed to obtain `res2`.

Awaiting Multiple Promises ii

- We can also wrap them in **Promise.all** and, then **await**:

```
async function myFunc(){
  // wait for the array of results
  let results = await Promise.all([
    axios.get(URL1),
    axios.get(URL2),
    ...
  ]);
  console.log(results);
}
```

- In the previous case, all the GETs are sent simultaneously.
- In case of an error, it propagates as usual:
  - From the failed Promise to **Promise.all**.
  - And then, becomes an exception that we can catch using **try/catch** around the call.

Request HTTP API with **async/await**

We can get the balance of an account in EUR with the following code:

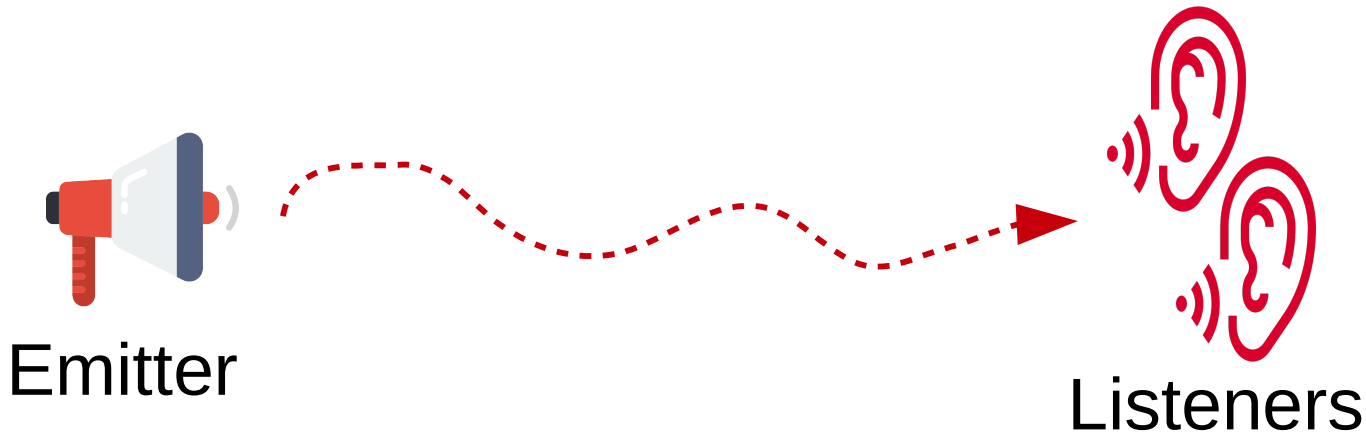
```
const axios = require("axios");
const address = "0xddbd2b932c763ba5b1b7ae3b362eac3e8d40121a";
const API_KEY = "AIX5NGPXXVWI66MCYCZ4MYZHBJBQ6KPSN4";
const BASE ="https://api.etherscan.io/api?module=account&action=balance&tag=latest";
const URL = `${BASE}&address=${address}&apikey=${API_KEY}`;
const URL2 = "https://min-api.cryptocompare.com/data/price?fsym=ETH&tsyms=EUR";

const getEthBalance = async () => {
 try {
   const balance = await axios.get(URL);
   const rate = await axios.get(URL2);
   console.log(parseInt(balance.data.result,10)/10**18*rate.data.EUR);
 } catch (error){
   console.error(`${error}`);
   }
 }

getEthBalance();
```

Request HTTP API with **async/await** + **Promise.all**

Better launch the two queries simultaneously:

```
const axios = require("axios");
const address = "0xddbd2b932c763ba5b1b7ae3b362eac3e8d40121a";
const API_KEY = "AIX5NGPXXVWI66MCYCZ4MYZHBJBQ6KPSN4";
const BASE ="https://api.etherscan.io/api?module=account&action=balance&tag=latest";
const URL = `${BASE}&address=${address}&apikey=${API_KEY}`;
const URL2 = "https://min-api.cryptocompare.com/data/price?fsym=ETH&tsyms=EUR";

const getEthBalance = async () => {
 try {
   [balance, rate] = await Promise.all([axios.get(URL),axios.get(URL2)]);
   console.log(parseInt(balance.data.result,10)/10**18*rate.data.EUR);
 } catch (error){
   console.error(`${error}`);
   }
 }

getEthBalance();
```

Outline



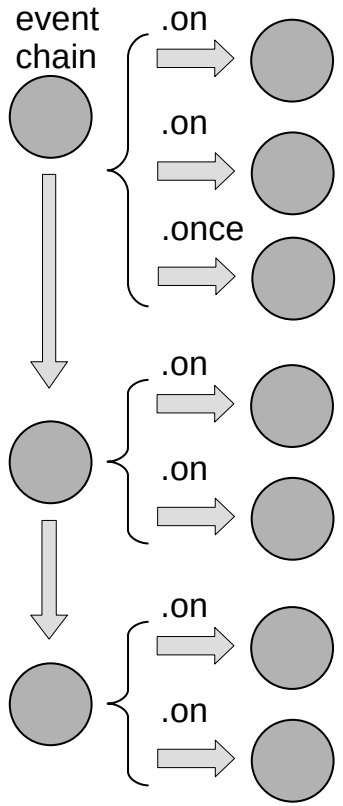
Event-driven Architectures

Much of the Node.js core API is built around an asynchronous event-driven architecture in which certain kinds of objects (called "`emitters`") emit named events that cause function objects ("`listeners`") to be called.

See `https://nodejs.org/docs/latest/api/events.html`

Reactive Programming

- Differently from Promises, events are a continuous flow.
- Callbacks are executed for each event and event can occur multiple times.
- This is called **”reactive programming”**.



`EventEmitter` class

- All objects that emit events are instances of the **EventEmitter** class:
  - These objects expose an `eventEmitter.on()` function that allows one or more functions to be attached to named events emitted by the object.
  - Typically, event names are camel-cased strings but any valid JavaScript property key can be used.
- When the **EventEmitter** object emits an event:
  - All of the functions attached to that specific event (**listeners**) are called **synchronously**.
  - Any values returned by the called listeners are ignored and will be discarded.

Example of **EventEmitter**

In the following example we have two Listeners:

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();

myEmitter.on('minedBlock', () => {
  console.log('Listerner1: a block was mined!');
});

myEmitter.on('minedBlock', () => {
  console.log('Listener2: a block was mined!');
});


myEmitter.emit('minedBlock');
```

Asynchronous Listeners

- The EventEmitter **calls all listeners synchronously** in the order in which they were registered.
- This is important to ensure the proper sequencing of events and to avoid race conditions or logic errors.
- Listener functions can also switch to an **asynchronous mode** as follows:

```
const myEmitter = new MyEmitter();

myEmitter.on('myevent', (a, b) => {
  setTimeout(() => console.log('this happens asynchronously'), 0);
});

myEmitter.emit('myevent', 'a', 'b'); // We can pass arguments to listeners
```

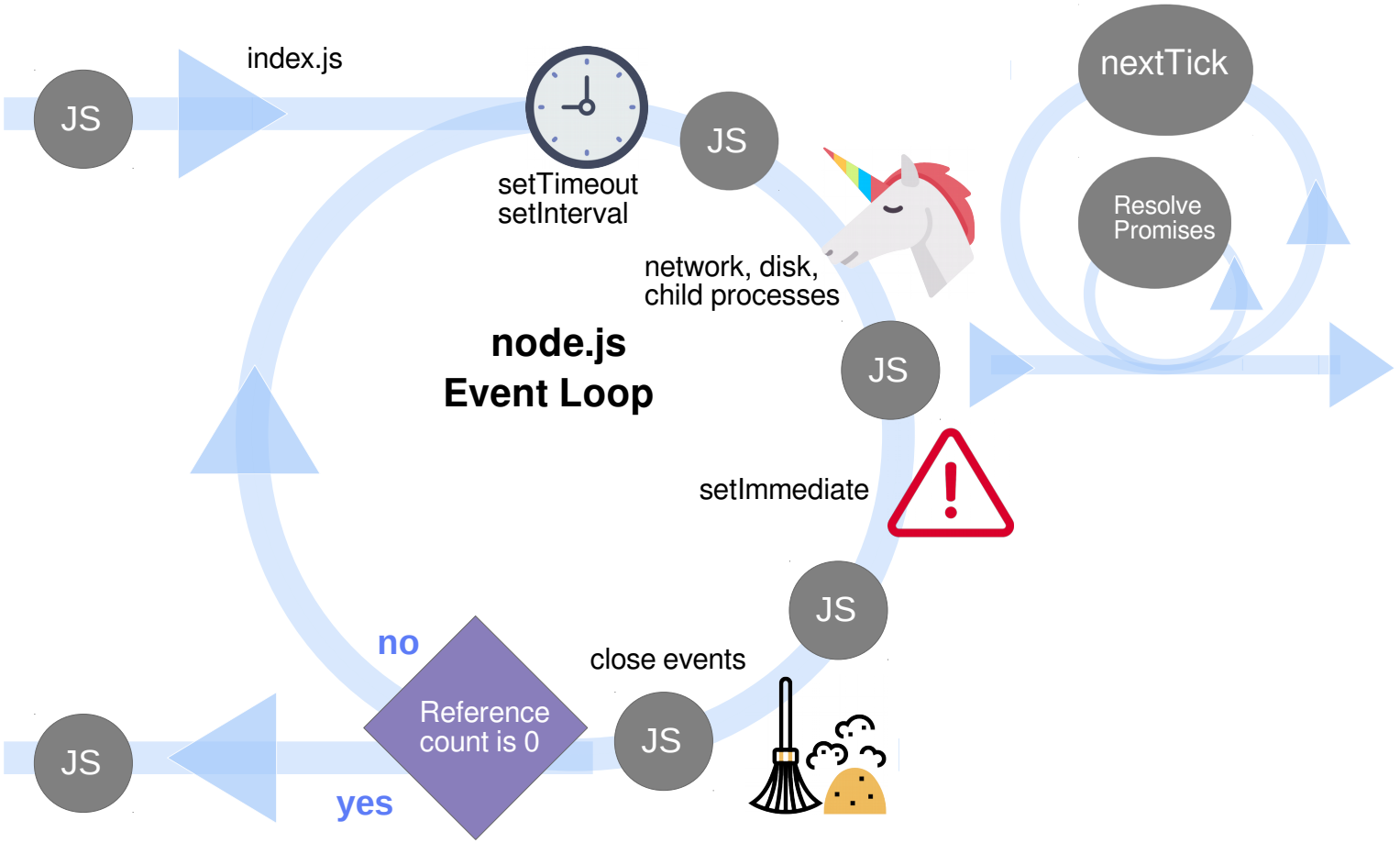
Outline

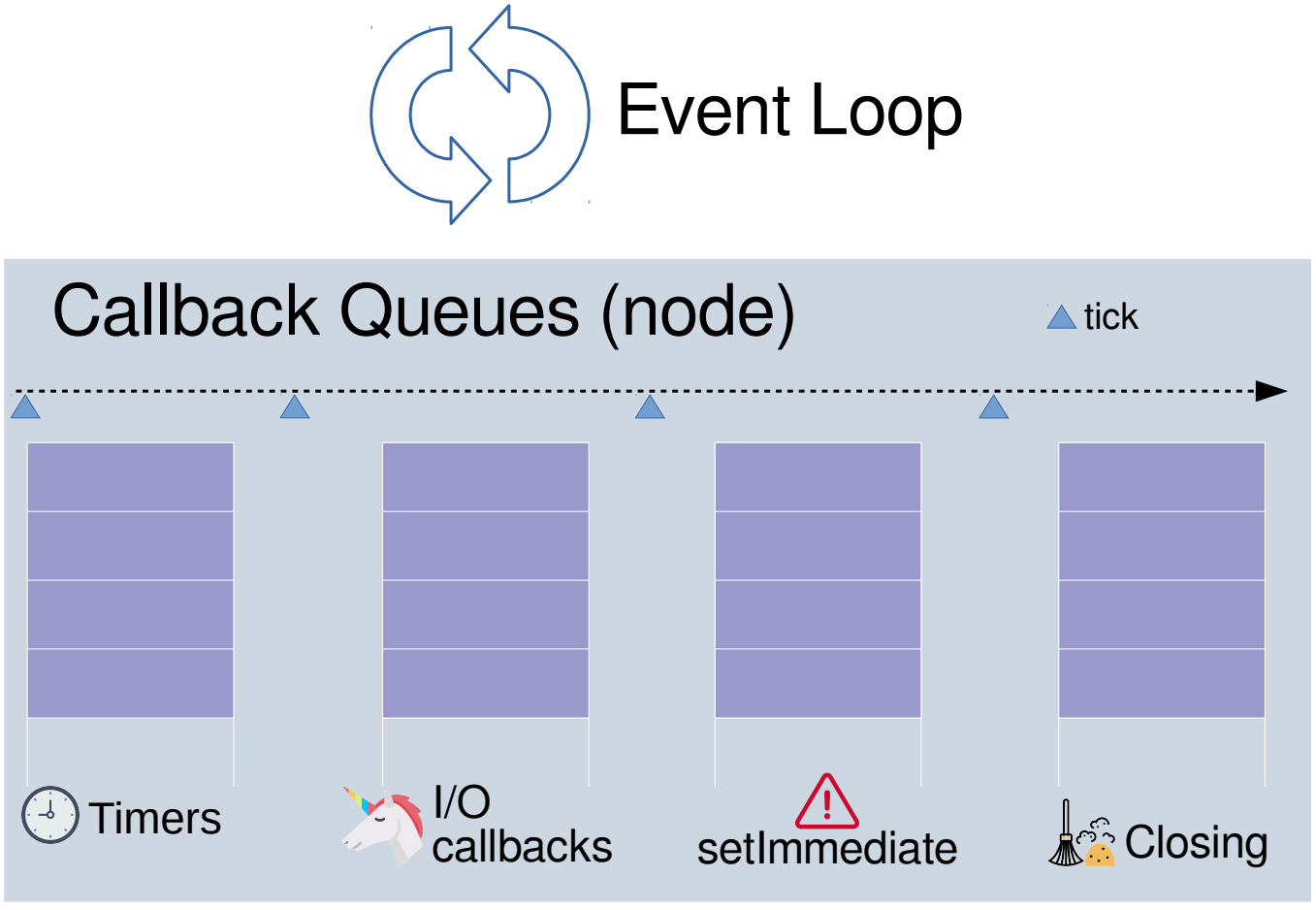
Event Loop in More Detail i

Callbacks management is abstracted away from the developer:

- This is all handled by a library called libuv which among other things provides the event loop.
- There are 4 basic steps that the event loop repeats in each iteration:
  1. Figure out if there are any timeouts
  2. Unicorn: disk, network, child processes... (most of the magic happens here)
  3. setImmediate
  4. Close events
- Between each of the previous steps, node calls your JS code.

Event Loop in More Detail ii




Event Loop
Callback Queues (node)
tick
Timers
I/O
callbacks
setImmediate
Closing

`setTimeout`, `setImmediate` & `nextTick` ii

- `setTimeout`:
  - Whenever a function is called with `setTimeout` it is not executed immediately but it is queued.
  - `setTimeout` uses the ”timers queue”.
  - The function is executed after all the previous functions in the timers queue have been executed.
  - No guarantees can be made about how long it could take to execute a particular function.
  - `setTimeout(f,0)` essentially means execute after all current functions in the timers queue get executed.
- `setImmediate`:
  - Uses the ”check queue”.
  - The ”check queue” (and also the ”timers queue”) is served only after I/O callbacks queue is empty.
  - `setImmediate` and `setTimeout(f,0)` are very similar.
  - But in general, `setTimeout(f,0)` is slower than `setImmediate` because the timer has to be checked before executing the corresponding callback.
  - Remark that `setImmediate` is a node-only function (not implemented by browsers).

- `nextTick`:
  - In each round, a callback on each queue (timers, I/O callbacks, **`setImmediate`** or closing) is dequeued.
  - By the contrary, **`nextTick`** is fired in each tick:
    - Immediately after the current code is done executing
    - And **BEFORE** going back to the event loop.
  - **ALL** nextTick callbacks are executed before returning to the event loop.
  - ALL settled callbacks of promises are executed within each tick.
  - **`nextTick`** callbacks are also known as "microtasks" while the other callbacks are called "macrotasks".

Example of **setImmediate** i

```
setImmediate(function A() {

   setImmediate(function B() {
        console.log(1);
        setImmediate(function D() { console.log(2); });
        setImmediate(function E() { console.log(3); });
  });

   setImmediate(function C() {
        console.log(4);
        setImmediate(function F() { console.log(5); });
        setImmediate(function G() { console.log(6); });
  });
});

setTimeout(function timeout() {
  console.log('TIMEOUT FIRED');
}, 0)

// 'TIMEOUT FIRED' 1 4 2 3 5 6
// Or
// 1 'TIMEOUT FIRED' 4 2 3 5 6
```

Example of **setImmediate** ii

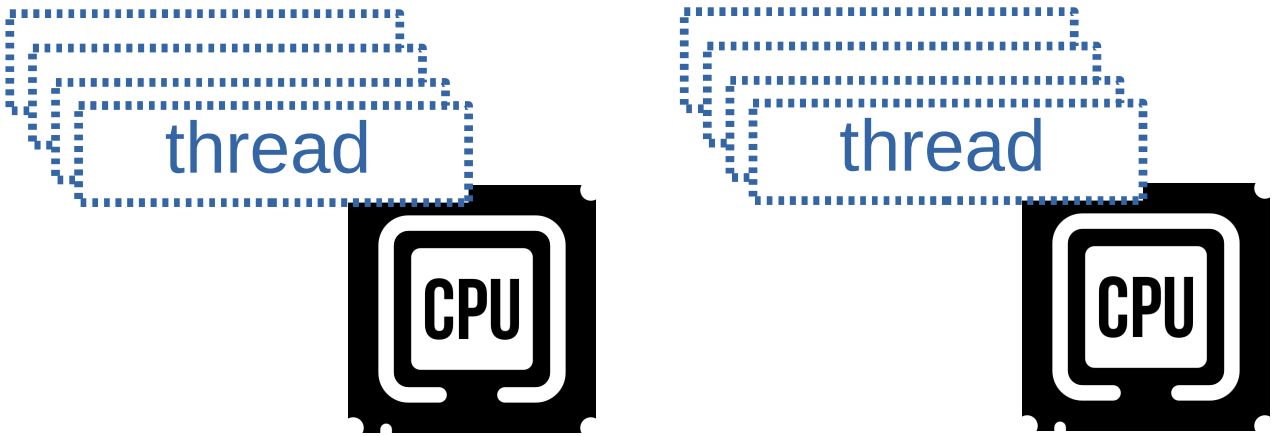
- **setImmediate** callbacks are fired off the event loop, **once per iteration** in the order that they were queued:
  - So, on the first iteration of the event loop, callback A is fired.
  - Then on the second iteration of the event loop, callback B is fired, then on the third iteration of the event loop callback C is fired, etc.
- This prevents the event loop from being **blocked**.
- Also allows other I/O or timer callbacks to be called in the mean time.
- As is the case of the 0 ms timeout, which is fired on the 1st or 2nd loop iteration.

Example of **process.nextTick**

```
process.nextTick(function A() {

  process.nextTick(function B() {
    log(1);
    process.nextTick(function D() { log(2); });
    process.nextTick(function E() { log(3); });
  });

  process.nextTick(function C() {
    log(4);
    process.nextTick(function F() { log(5); });
    process.nextTick(function G() { log(6); });
  });
});

setTimeout(function timeout() {
  console.log('TIMEOUT FIRED');
}, 0)

// 1 4 2 3 5 6 'TIMEOUT FIRED'
```

Outline

Threads



- A thread of execution can be thought of as a ”mini CPU” executing the program.
- However, with threads, there is a **single program**.
- Since threads execute the same program, they read and write the same memory simultaneously.
- This makes multithreading very challenging: read/write can result in errors not seen in a single-threaded programs.

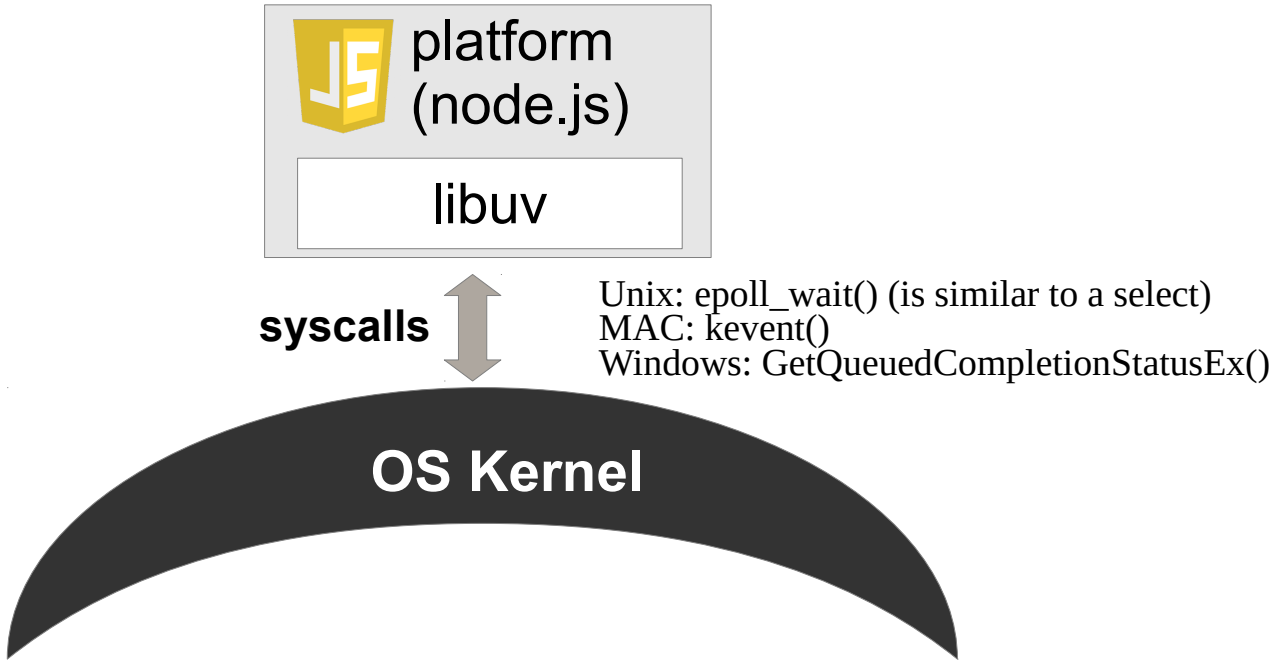
Threads Scaling Problem

A thread per connection using **accept()**:

```
1  int server = socket();
2  bind(server, 80);
3  listen(server); /* accept connections */
```

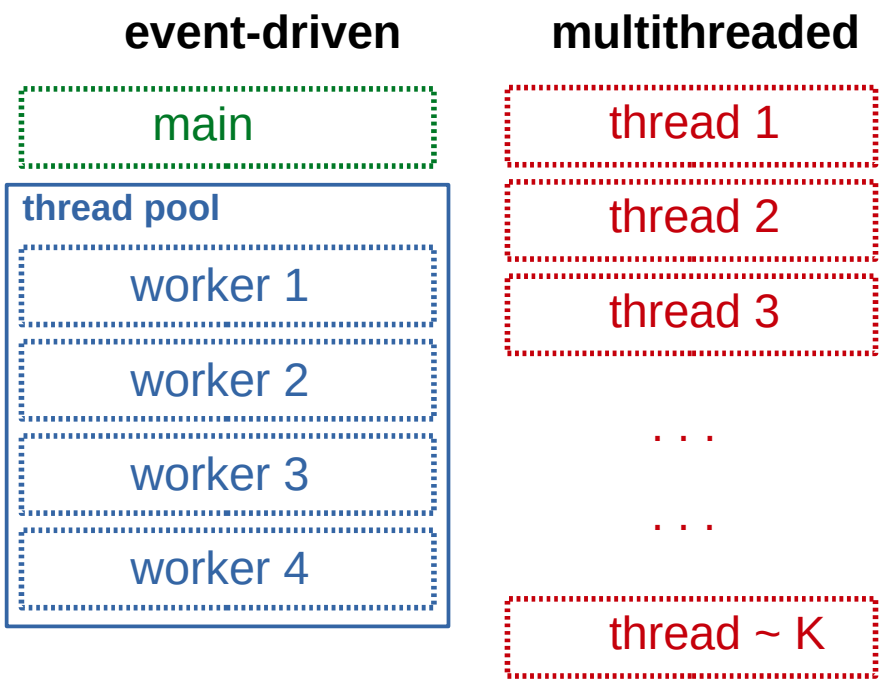
- The accept system call blocks until new packets arrive (the thread receives no more CPU until this happens)
- Threads are relatively heavy weight.
- Nowadays we can have a few thousand threads (K threads).
- Not scale for servers with millions of connections.

Another Approach: Events from OS



- Today's Operating Systems (OS) already provide **asynchronous** interfaces for many I/O tasks.
- Node.js tries to use the OS API to subscribe to event notifications when possible.
- We need to use just **one thread**!

Event-driven vs. Multithreaded Processes

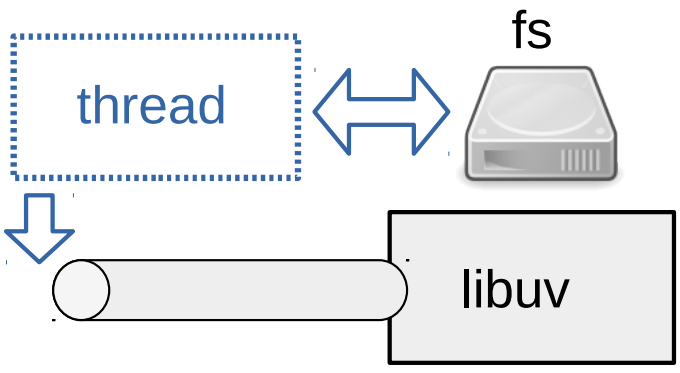


- Main drawbacks of big multithreaded applications:
  1. There is a lot of ”context switching” that wastes CPU.
  2. Difficult and prone to errors for programmers.

Linux Epoll

- We create an **epoll descriptor** to tell the kernel in which events we are interested.
- Then, the **kernel** is going to use it to tell us when the event happens.
- Pollable events: tcp/udp sockets clients and servers, unix domain sockets clients and servers, pipes, tty input, dns.resolveXXXX calls,...
- The problem is that:
  - Not everything in the **node API** epoll-able.
  - E.g. the file system operations are not pollable.

Trick for non Pollable Events: Self-pipes



- Trick to use epoll with non pollable events: **self-pipes**
- A pipe, where one end is written by a thread (or signal handler) and the other end is polled in the loop.

Thread Pool

- **Libuv** by default creates a thread pool (called workers) to offload asynchronous work to.
- Whenever possible, libuv will use those asynchronous interfaces, avoiding usage of the thread pool.
- By default the node's thread pool has **4 threads** but you can configure the size.
- The following events are managed via the thread pool:
  - fs events
  - dns (only dns.lookup())
  - crypto (only crypto.randomBytes() and crypto.pbkdf2())
  - http.get/request() if called with a name because of dns.lookup
  - C++ addons