O'REILLY®

Asynchronous Programming with Node.js

Chetan Karande

December/2019

What you'll learn?

- How asynchronous programming enables high performance and scalable applications
- Asynchronous programming features in JavaScript Language and Node.js
- Applications of Asynchronous programming for I/O and memory heavy operations
- Common pitfalls associated with asynchronous coding

How you can apply it?

- Write asynchronous programs in a robust, maintainable and comprehensible way

As an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications.

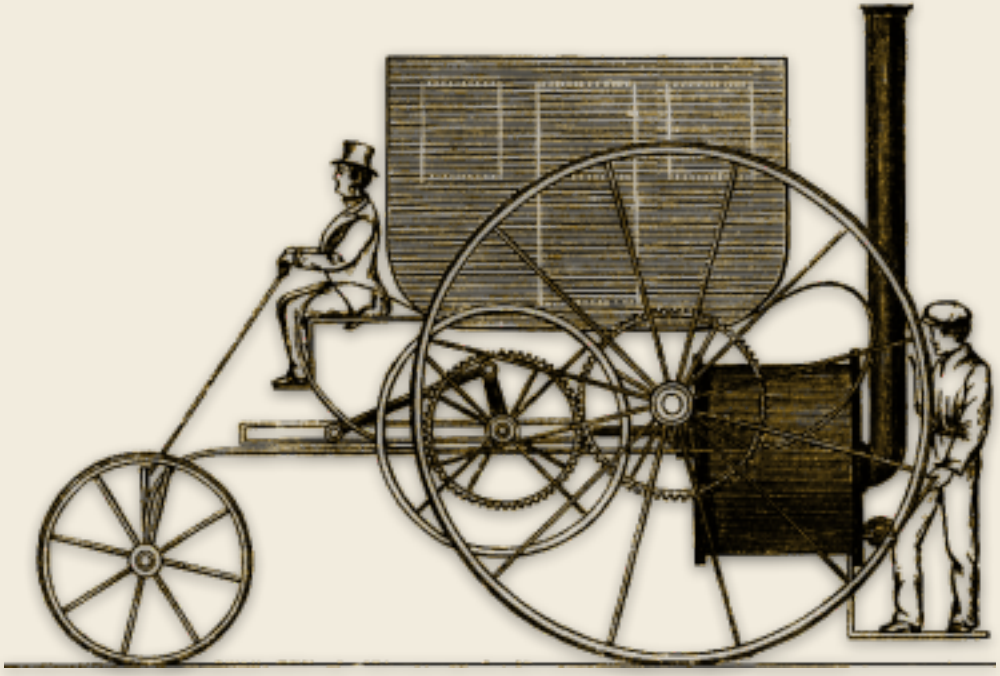
https://nodejs.org/en/about/

Asynchronous Programming, Concurrency & Throughput

Richard Trevithick

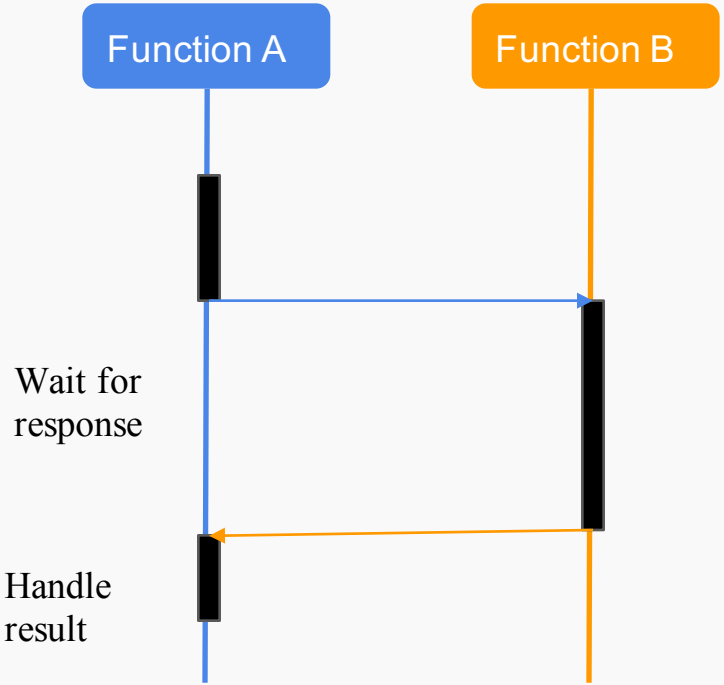

Heat Loss


Heat Loss



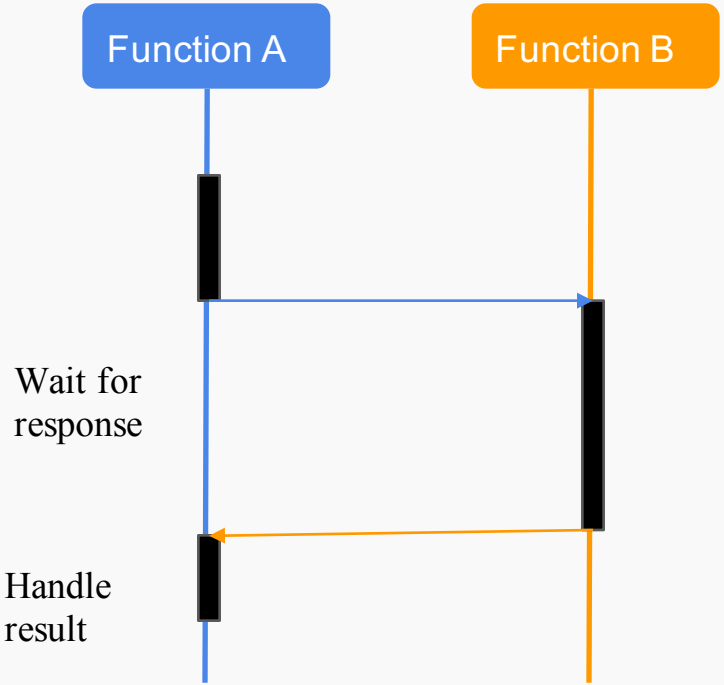
First self-propelled Vehicle
Trevithick Steam Carriage (Year

Synchronous vs Asynchronous execution

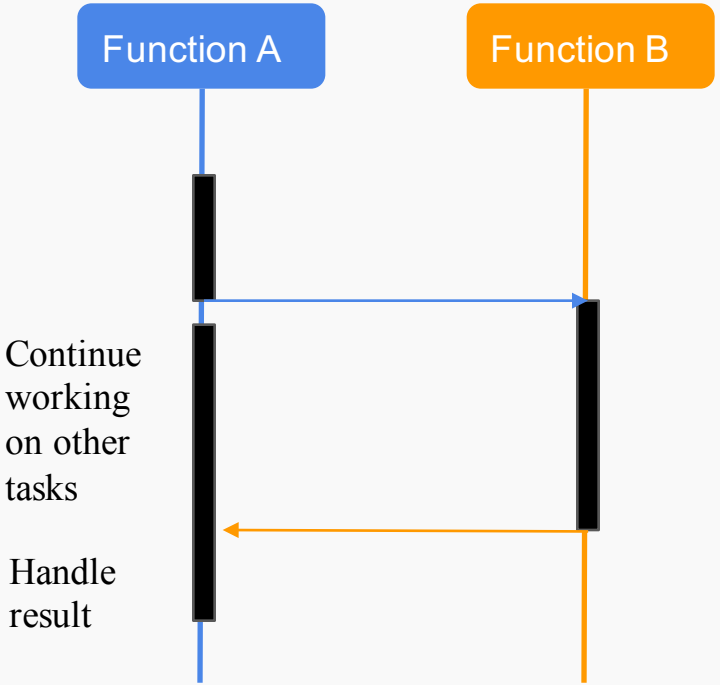


Synchronous

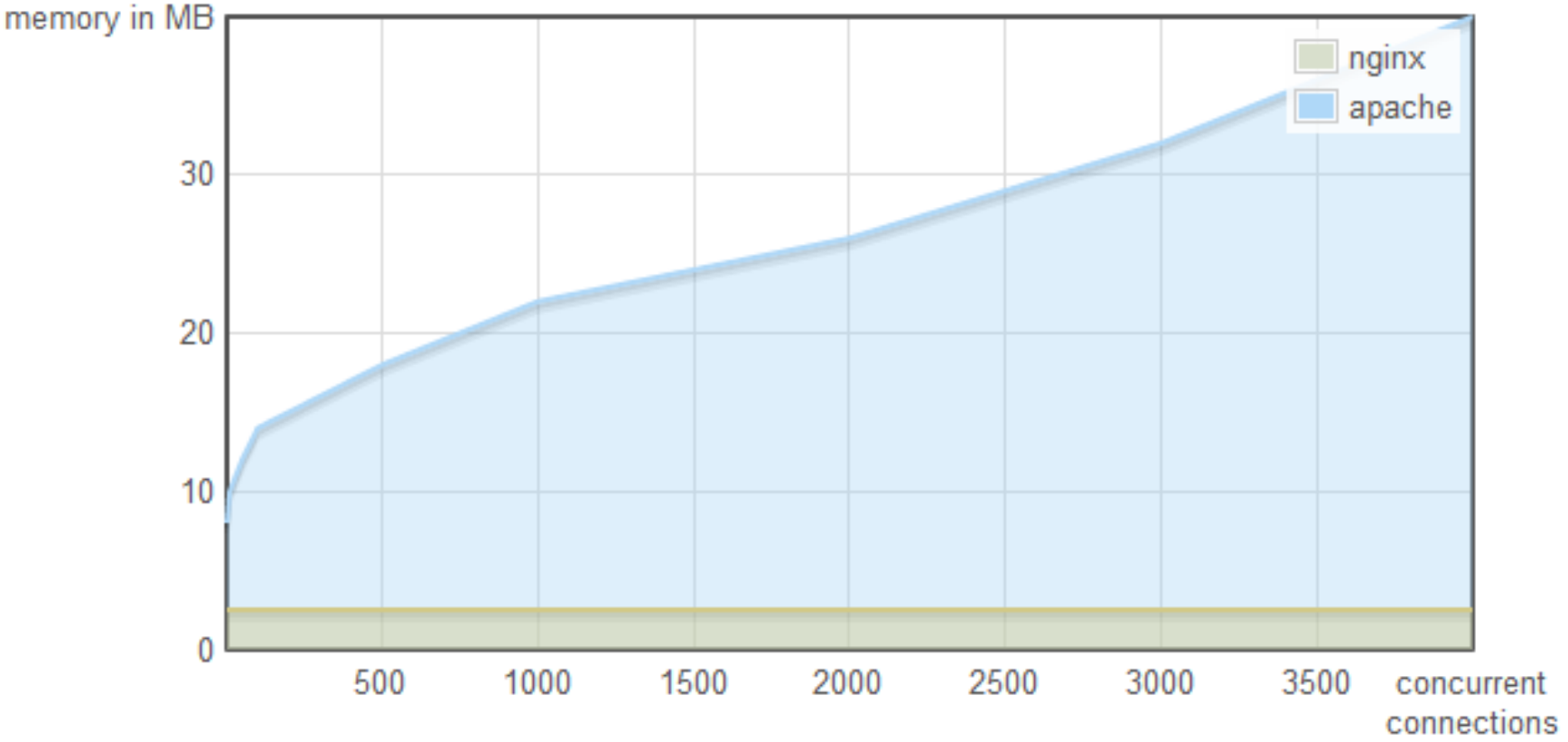
Synchronous vs Asynchronous execution



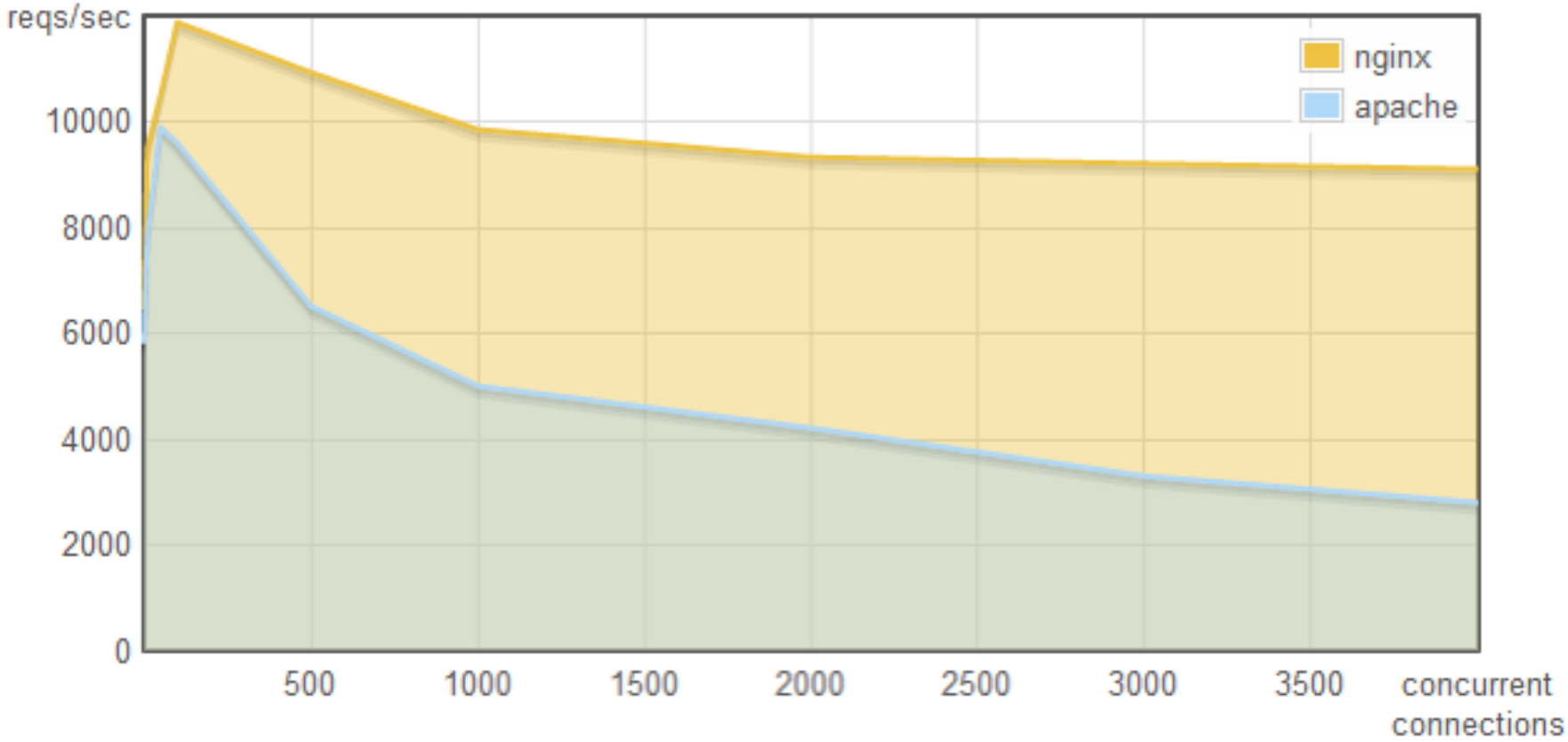
Synchronous



Asynchronous

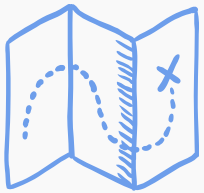


https://blog.webfaction.com/2008/12/a-little-holiday-present-10000-reqssec-with-nginx-2/



https://blog.webfaction.com/2008/12/a-little-holiday-present-10000-reqssec-with-nginx-2/

Roadmap



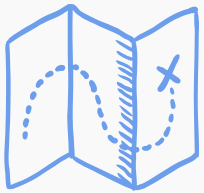
Async Programming Essentials

- Async Programming Features in JavaScript
- Node.js Event Loop Phases

Applied Async and Parallel Programming

- Handling I/O Operations
- Handling Memory Intensive Operations
- Handling CPU Heavy Operations

Roadmap



Async Programming Essentials

- Async Programming Features in JavaScript
- Node.js Event Loop Phases

Applied Async and Parallel Programming

- Handling I/O Operations
- Handling Memory Intensive Operations
- Handling CPU Heavy Operations

Asynchronous Programming Features in JavaScript Language

Built-in Objects

- Promise
- Generator

Expressions and Operators

- await
- yield and yield*

Statements & Declarations:

- async function
- async iterables
- for await...of
- function*

Callback Hell

```
function processOrder(order) {
    chargePaymentAsync(order, function (err, updatedOrder) {
        updateInventoryAsync(updatedOrder, function (err, updatedOrder) {
            shipAsync(order, function (err, updatedOrder) {
                sendEmailAsync(updatedOrder, function (err, updatedOrder) {
                    console.log("Done!");
                });
                console.log("Exiting shipAsync!");
            });
        });
    });
    // ...run some logic only if payment was successful
}
processOrder(order);
```

Pyramid of Doom

```
1  function processOrder(order) {
2      chargePaymentAsync(order, function (err, updatedOrder) {
3          updateInventoryAsync(updatedOrder, function (err, updatedOrder) {
4              shipAsync(order, function (err, updatedOrder) {
5                  sendEmailAsync(updatedOrder, function (err, updatedOrder) {
6                      console.log("Done!");
7                  });
8                  console.log("Exiting shipAsync!");
9              });
10         });
11     });
12
13 }
14 processOrder(order);
```

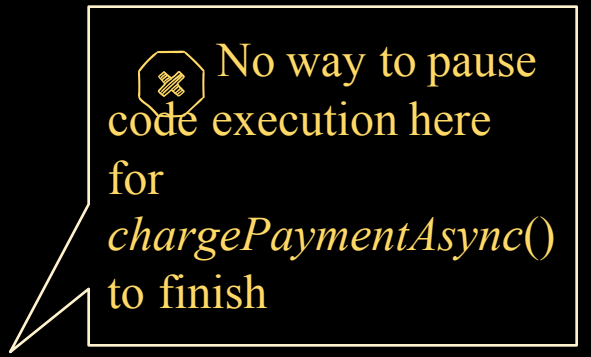
Lack of Guarantee (Inversion of Control)

```
1  function processOrder(order) {
2      chargePaymentAsync(order, function (err, order) {
3          updateInventoryAsync(order, function (err, order) {
4              shipAsync(order, function (err, order) {
5                  sendEmailAsync(order, function (err, order) {
6                      console.log("Done!");
7                  });
8                  console.log("Exiting shipAsync!");
9              });
10         });
11     });
12
13 }
14 processOrder(order);
```

No guarantees that callback gets invoked:
- once & only once
- not too early
- with right context
- with right arguments

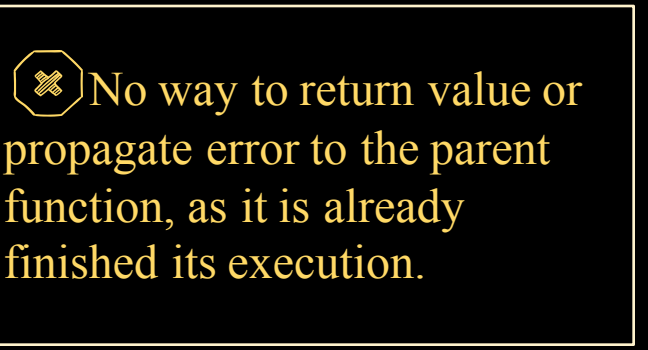
Lack of Execution Flow Control

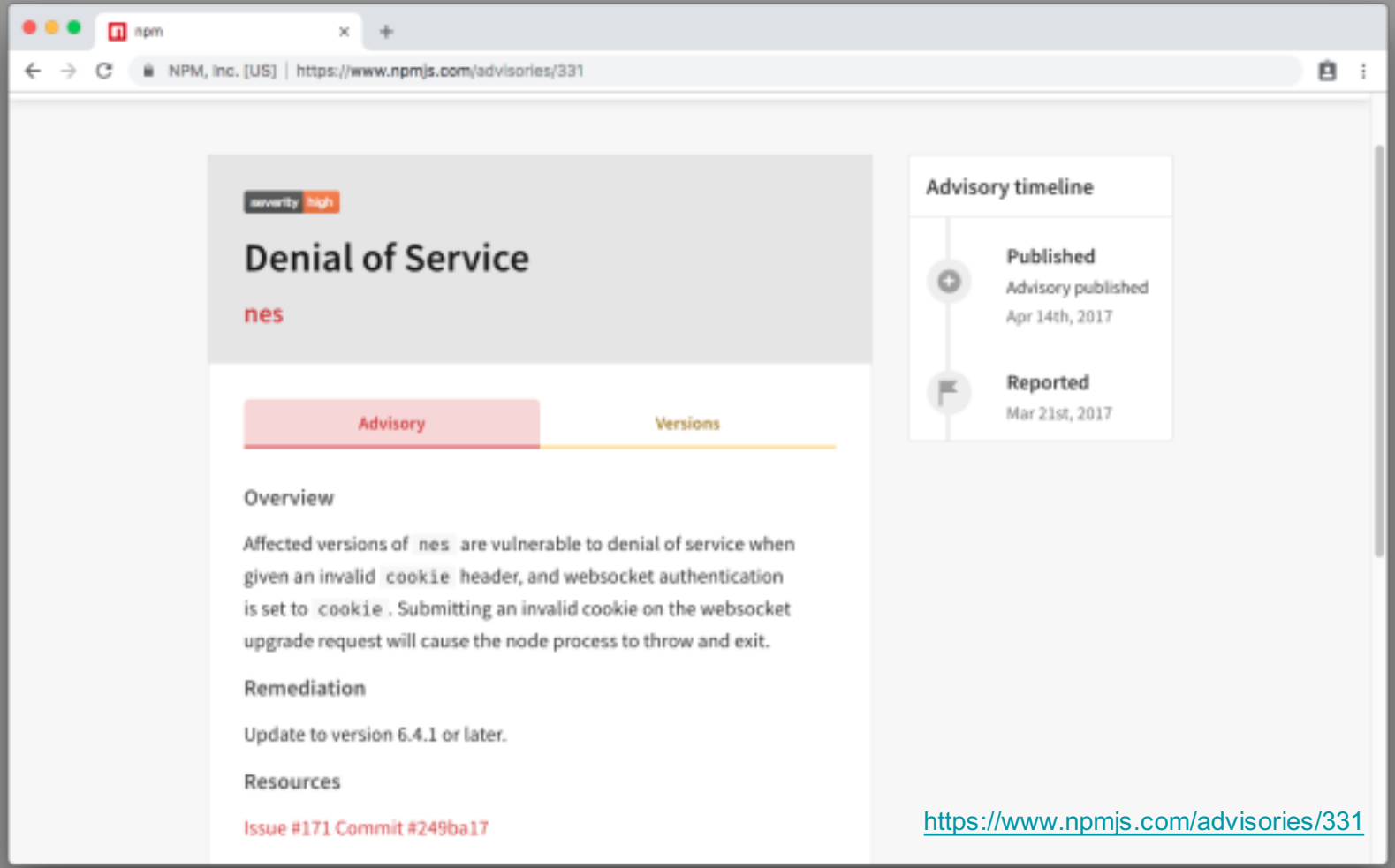
```
1  function processOrder(order) {
2      chargePaymentAsync(order, function (err, updatedOrder) {
3          updateInventoryAsync(order, function (err, updatedOrder) {
4              shipAsync(order, function (err, updatedOrder) {
5                  sendEmailAsync(order, function (err, updatedOrder) {
6                      console.log("Done!");
7                  });
8                  console.log("Exiting shipAsync!");
9              });
10         });
11     });
12     // ...run some logic only if payment was successful
13 }
14 processOrder(order);
```



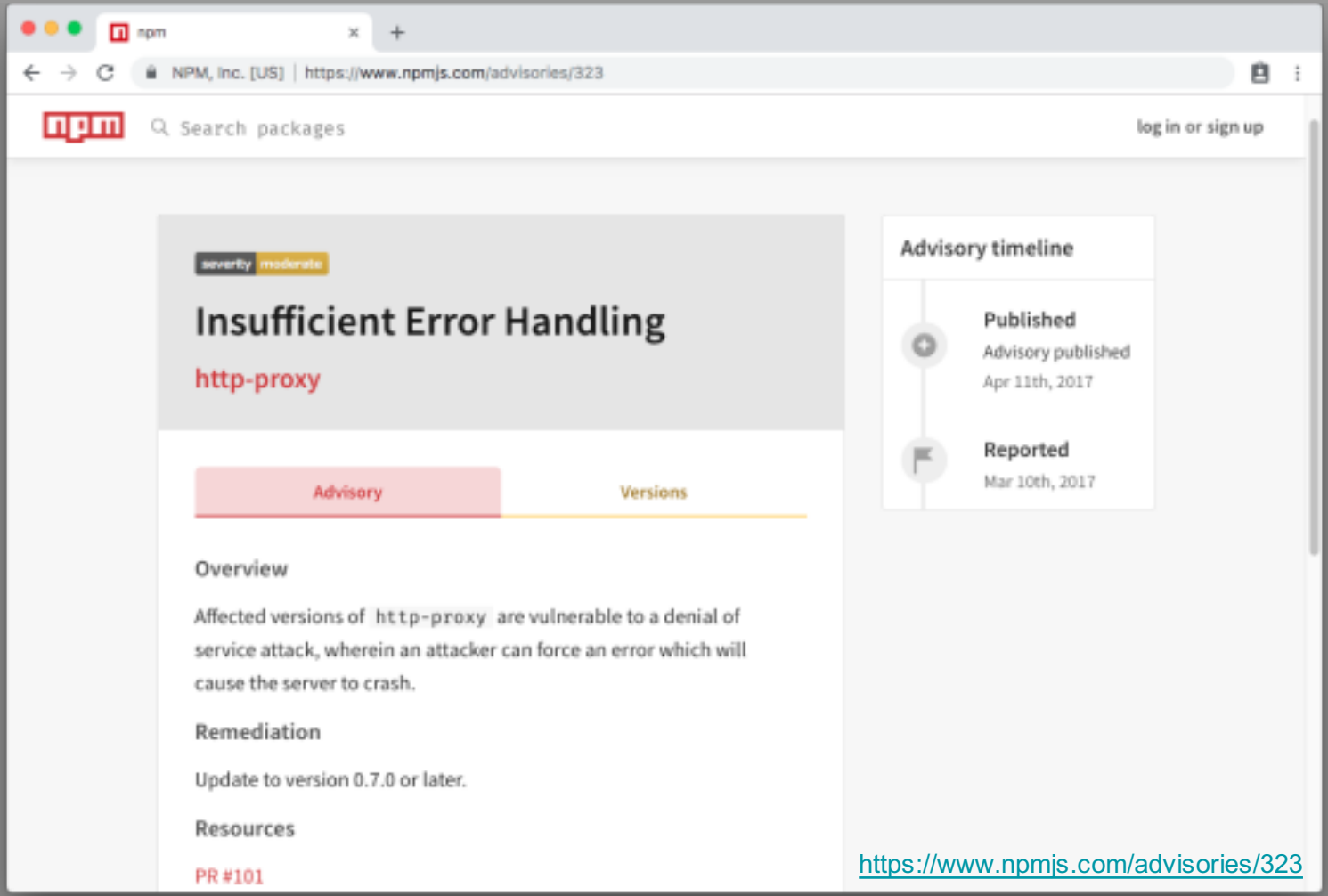
Lack of Data and Error Propagation Control

```
1  function processOrder(order) {
2      chargePaymentAsync(order, function (err, updatedOrder) {
3          updateInventoryAsync(updatedOrder, function (err, updatedOrder) {
4              shipAsync(updatedOrder, function (err, updatedOrder) {
5                  sendEmailAsync(order, function (err, updatedOrder) {
6                      if(err) throw error;
7                      return updatedOrder;
8                  });
9                  console.log("Exiting shipAsync!");
10             });
11         });
12     });
13     // any code here runs immediately after chargePaymentAsync() is scheduled
14 }
```




npm
NPM, Inc. [US] | https://www.npmjs.com/advisories/331
severity high
Denial of Service
nes
Advisory
Versions
Overview
Affected versions of nes are vulnerable to denial of service when given an invalid cookie header, and websocket authentication is set to cookie. Submitting an invalid cookie on the websocket upgrade request will cause the node process to throw and exit.
Remediation
Update to version 6.4.1 or later.
Resources
Issue #171 Commit #249ba17
Advisory timeline
Published
Advisory published
Apr 14th, 2017
Reported
Mar 21st, 2017
https://www.npmjs.com/advisories/331

https://www.npmjs.com/advisories/331 (PR)



https://www.npmjs.com/advisories/323

14 lib/node-http-proxy.js

```
@@ -125,11 +125,15 @@ exports.stack = function stack (middlewares, proxy) {
     handle = function (req, res) {
       var next = function (err) {
         if (err) {
-          throw err;
-          //
-          // TODO: figure out where to send errors.
-          // return error(req, res, err);
-          //
+          console.error(err.stack);
+          if (res._headerSent) {
+            res.destroy();
+          } else {
+            res.statusCode = 500;
+            res.setHeader('Content-Type', 'text/plain');
+            res.end('Internal Server Error');
+          }
+          return;
         }
         child(req, res);
       }
```

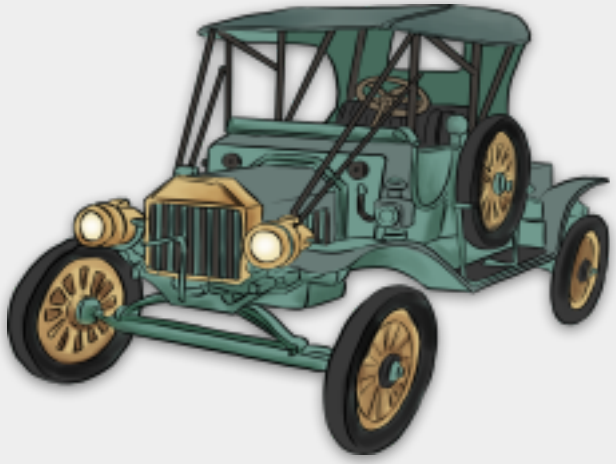
https://github.com/nodejitsu/node-http-proxy/pull/101/files

Impact on Code Comprehensibility

```
1  function processOrder(order) {
2      chargePaymentAsync(order, function (err, order) {
3          updateInventoryAsync(order, function (err, order) {
4              shipAsync(order, function (err, order) {
5                  sendEmailAsync(order, function (err, order) {
6                      console.log("Done!");
7                  });
8                  console.log("Exiting shipAsync!");
9              });
10         });
11     });
12
13 }
14 processOrder(order);
```

Human brain works linearly and good at comprehending sequential synchronous code than the nested code.

Promise

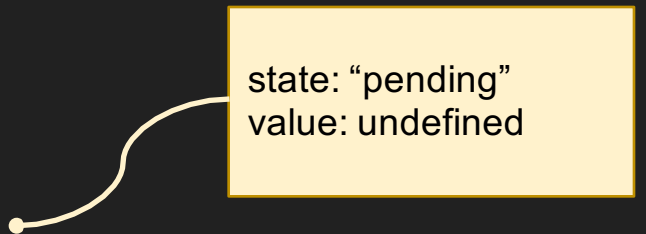


Ford Model T (year 1900)

Promise = Future Value

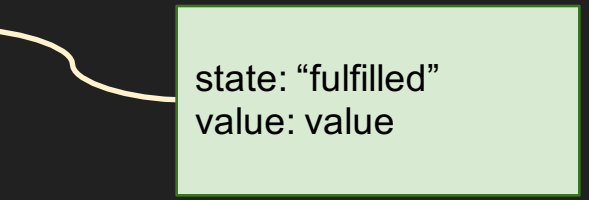
Promise API: How to create a promise instance?

```
1  const executor = function (resolve, reject) {
2
3  ...
4
5  }
6  const promise =  new Promise(executor);
```



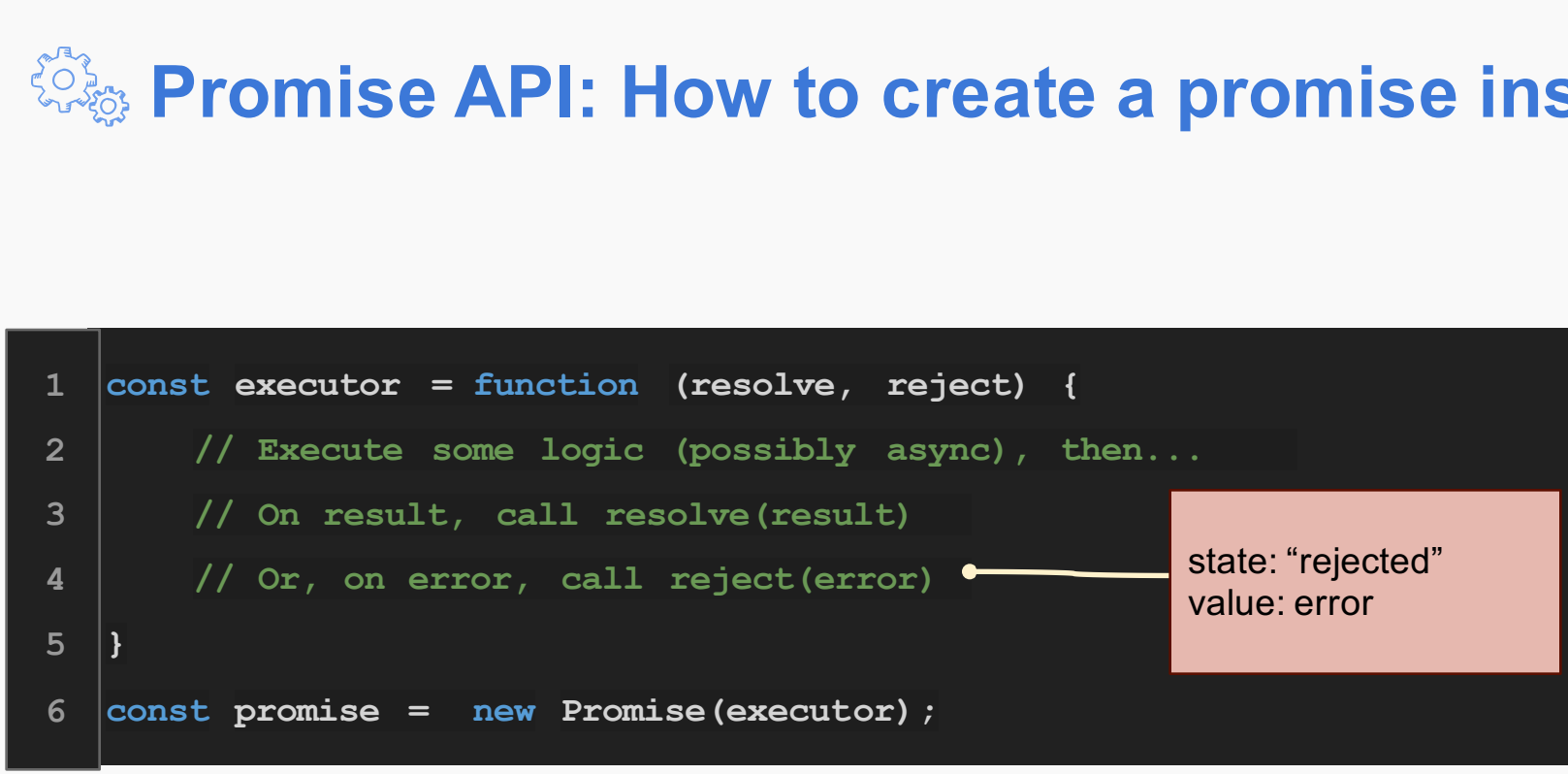
Promise API: How to create a promise instance?

```
1  const executor = function (resolve, reject) {
2      // Execute some logic (possibly async), then...
3      // On result, call resolve(value)
4
5  }
6  const promise =  new Promise(executor);
```

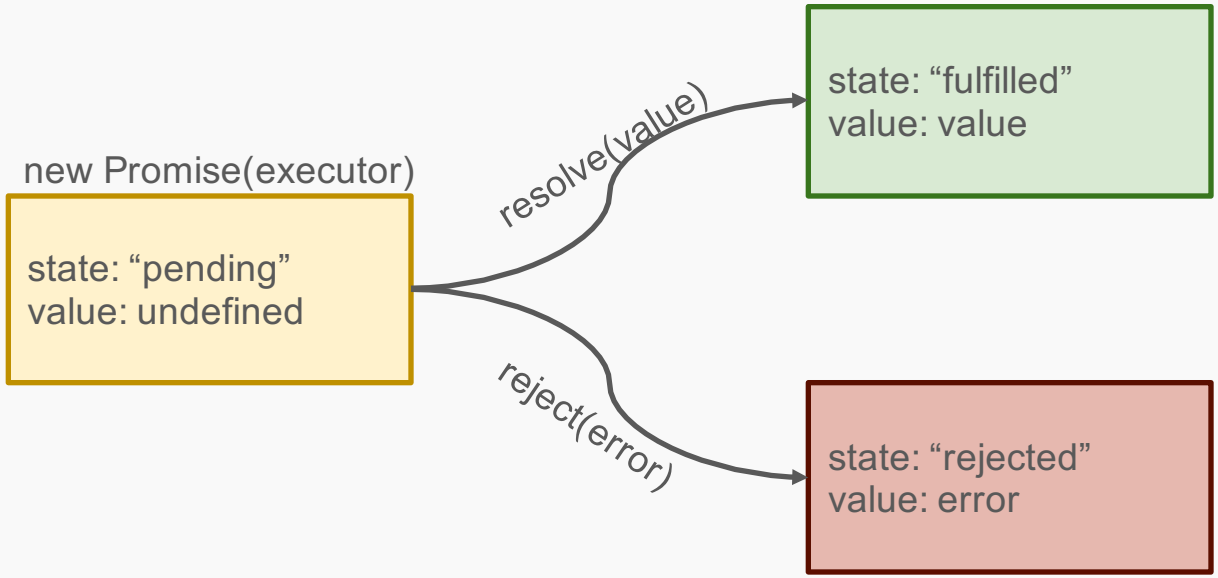


Promise API: How to create a promise instance?

```
1  const executor = function (resolve, reject) {
2      // Execute some logic (possibly async), then...
3      // On result, call resolve(result)
4      // Or, on error, call reject(error)
5  }
6  const promise =  new Promise(executor);
```



Promise API: States of a promise instance



Promise = Future Value + Completion Events

Promise API: How to add handlers on Completion Events?

promise instance internal slots:

[[PromiseFulfillReactions]]: [...]
[[PromiseRejectReactions]]: [...]
[[PromiseState]]: "..."
[[PromiseResult]]: ...

Adding a fulfillment handler:

```
promise.then(onFulfilled);
```

Adding a rejection handler

```
promise.then(onFulfilled, onRejected);
promise.then(undefined, onRejected);
promise.catch(onRejected);
```

Escaping Callback Hell: Fixing Inversion of Control

```
1  function someAsyncAction(onSuccess, onFailure) {
2      setTimeout(function() {
3          const result = Boolean(Math.round(Math.random()));
4          if (result === true) onSuccess('Success!');
5          else onFailure(Error('Failed!!'));
6      }, 100);
7  }
8  function main() {
9      function onSuccess (value) { console.log (value)};
10     function onFailure (err) { console.log (err.message)};
11     someAsyncAction(onSuccess, onFailure);
12 }
13 main();
14
```

Inversion of Control when using Callbacks

main() function has no control over invoking callbacks

Using Promise

```
function someAsyncAction() {
    const promise = new Promise((resolve, reject) => {
        setTimeout(() => {
            const result = Boolean(Math.round(Math.random()));
            if (result == true) resolve('Success!');
            else reject(Error('Failed!!'));
        }, 100);
    });
    return promise;
}
(function main() {
    function onSuccess (value) { console.log (value)};
    function onFailure (err) { console.log (err.message)};
    const promise = someAsyncAction();
    promise.then(onSuccess, onFailure);
})();
```

```
function someAsyncAction() {
    const promise = new Promise((resolve, reject) => {
        setTimeout(() => {
            const result = Boolean(Math.round(Math.random()));
            if (result == true) resolve('Success!');
            else reject(Error('Failed!!'));
        }, 100);
    });
    return promise;
}
(function main() {
    function onSuccess (value) { console.log (value)};
    function onFailure (err) { console.log (err.message)};
    const promise = someAsyncAction();
    promise.then(onSuccess, onFailure);
})();
```

```
function someAsyncAction() {
    const promise = new Promise((resolve, reject) => {
        setTimeout(() => {
            const result = Boolean(Math.round(Math.random()));
            if (result == true) resolve('Success!');
            else reject(Error('Failed!!'));
        }, 100);
    });
    return promise;
}
(function main() {
    function onSuccess (value) { console.log (value)};
    function onFailure (err) { console.log (err.message)};
    const promise = someAsyncAction();
    promise.then(onSuccess, onFailure);
})();
```

```
1  function someAsyncAction() {
2      const promise = new Promise((resolve, reject) => {
3          setTimeout(() => {
4              const result =  Boolean(Math.round(Math.random()));
5              if (result == true) resolve('Success!');
6              else reject(Error('Failed!!'));
7          }, 100);
8      });
9      return promise;
10 }
11 (function main() {
12     function onSuccess (value) { console.log (value)};
13     function onFailure (err) { console.log (err.message)};
14     const promise = someAsyncAction();
15     promise.then(onSuccess, onFailure);
16 })();
```

No callback passed to async function!

```
function someAsyncAction() {
    const promise = new Promise((resolve, reject) => {
        setTimeout(() => {
            const result = Boolean(Math.round(Math.random()));
            if (result == true) resolve('Success!');
            else reject(Error('Failed!!'));
        }, 100);
    });
    return promise;
}
(function main() {
    function onSuccess (value) { console.log (value)};
    function onFailure (err) { console.log (err.message)};
    const promise = someAsyncAction();
    promise.then(onSuccess, onFailure);
})();
```

```
function someAsyncAction() {
    const promise = new Promise((resolve, reject) => {
        setTimeout(() => {
            const result = Boolean(Math.round(Math.random()));
            if (result == true) resolve('Success!');
            else reject(Error('Failed!!'));
        }, 100);
    });
    return promise;
}
(function main() {
    function onSuccess (value) { console.log (value)};
    function onFailure (err) { console.log (err.message)};
    const promise = someAsyncAction();
    promise.then(onSuccess, onFailure);
})();
```

```
function someAsyncAction() {
    const promise = new Promise((resolve, reject) => {
        setTimeout(() => {
            const result = Boolean(Math.round(Math.random()));
            if (result == true) resolve('Success!');
            else reject(Error('Failed!!'));
        }, 100);
    });
    return promise;
}
(function main() {
    function onSuccess (value) { console.log (value)};
    function onFailure (err) { console.log (err.message)};
    const promise = someAsyncAction();
    promise.then(onSuccess, onFailure);
})();
```

```
1  function someAsyncAction() {
2      const promise = new Promise((resolve, reject) => {
3          setTimeout(() => {
4              const result =  Boolean(Math.round(Math.random()));
5              if (result == true) resolve('Success!');
6              else reject(Error('Failed!!'));
7          }, 100);
8      });
9      return promise;
10 }
11 (function main() {
12     function onSuccess (value) { console.log (value)};
13     function onFailure (err) { console.log (err.message)};
14     const promise = someAsyncAction();
15     promise.then(onSuccess, onFailure);
16 })();
```

state: “pending”
value: undefined

```
1  function someAsyncAction() {
2      const promise = new Promise((resolve, reject) => {
3          setTimeout(() => {
4              const result =  Boolean(Math.round(Math.random()));
5              if (result == true) resolve('Success!');
6              else reject(Error('Failed!!'));
7          }, 100);
8      });
9      return promise;
10 }
11 (function main() {
12     function onSuccess (value) { console.log (value)};
13     function onFailure (err) { console.log (err.message)};
14     const promise = someAsyncAction();
15     promise.then(onSuccess, onFailure);
16 })();
```

```
1  function someAsyncAction() {
2      const promise = new Promise((resolve, reject) => {
3          setTimeout(() => {
4              const result =  Boolean(Math.round(Math.random()));
5              if (result == true) resolve('Success!');
6              else reject(Error('Failed!!'));
7          }, 100);
8      });
9      return promise;
10 }
11 (function main() {
12     function onSuccess (value) { console.log (value)};
13     function onFailure (err) { console.log (err.message)};
14     const promise = someAsyncAction();
15     promise.then(onSuccess, onFailure);
16 })();
```

.then() adds callbacks to promise instance's internal list of fulfillment and rejection reactions

```
1  function someAsyncAction() {
2      const promise = new Promise((resolve, reject) => {
3          setTimeout( () => {
4              const result = Boolean(Math.round(Math.random()));
5              if (result == true) resolve('Success!');
6              else reject(Error('Failed!!'));
7          }, 100);
8      });
9      return promise;
10 }
11 (function main() {
12     function onSuccess (value) { console.log (value)};
13     function onFailure (err) { console.log (err.message)};
14     const promise = someAsyncAction();
15     promise.then(onSuccess, onFailure);
16 })();
```

After 100 ms the setTimeout() callback gets added to the event loop timer queue and eventually executes

```
1  function someAsyncAction() {
2      const promise = new Promise((resolve, reject) => {
3          setTimeout(() => {
4              const result = Boolean(Math.round(Math.random())); // true
5              if (result == true)   resolve('Success!');
6              else reject(Error('Failed!!'));
7          }, 100);
8      });
9      return promise;
10 }
11 (function main() {
12     function onSuccess (value) { console.log (value)};
13     function onFailure (err) { console.log (err.message)};
14     const promise = someAsyncAction();
15     promise.then(onSuccess, onFailure);
16 })();
```

- Case 1 -
If the result was true

```
1  function someAsyncAction() {
2      const promise = new Promise((resolve, reject) => {
3          setTimeout(() => {
4              const result = Boolean(Math.round(Math.random())); // true
5              if (result == true) resolve('Success!');
6              else reject(Error('Failed!!'));
7          }, 100);
8      });
9      return promise;
10 }
11 (function main() {
12     function onSuccess (value) { console.log (value)};
13     function onFailure (err) { console.log (err.message)};
14     const promise = someAsyncAction(); // No callback passed
15     promise.then(onSuccess, onFailure);
16 })();
```

state: “fulfilled”
value: “Success!”

```
1  function someAsyncAction() {
2      const promise = new Promise((resolve, reject) => {
3          setTimeout(() => {
4              const result =  Boolean(Math.round(Math.random()));
5              if (result == true) resolve('Success!');
6              else reject(Error('Failed!!'));
7          }, 100);
8      });
9      return promise;
10 }
11 (function main() {
12     function onSuccess (value) { console.log (value)};
13     function onFailure (err) { console.log (err.message)};
14     const promise = someAsyncAction();
15     promise.then(onSuccess, onFailure);
16 })();
```

onSuccess callback gets added to promise **Microtask Queue** and eventually executes

```
1  function someAsyncAction() {
2      const promise = new Promise((resolve, reject) => {
3          setTimeout(() => {
4              const result = Boolean(Math.round(Math.random())); // false
5              if (result == true)   resolve('Success!');
6              else reject(Error('Failed!!'));
7          }, 100);
8      });
9      return promise;
10 }
11 (function main() {
12     function onSuccess (value) { console.log (value)};
13     function onFailure (err) { console.log (err.message)};
14     const promise = someAsyncAction();
15     promise.then(onSuccess, onFailure);
16 })();
```

- OR Case 2 - If the result was false

```
1  function someAsyncAction() {
2      const promise = new Promise((resolve, reject) => {
3          setTimeout(() => {
4              const result = Boolean(Math.round(Math.random())); // false
5              if (result == true)   resolve('Success!');
6              else reject(Error('Failed!!'));
7          }, 100);
8      });
9      return promise;
10 }
11 (function main() {
12     function onSuccess (value) { console.log (value)};
13     function onFailure (err) { console.log (err.message)};
14     const promise = someAsyncAction();
15     promise.then(onSuccess, onFailure);
16 })();
```

state: “rejected”
value: error

```
1  function someAsyncAction() {
2      const promise = new Promise((resolve, reject) => {
3          setTimeout(() => {
4              const result =  Boolean(Math.round(Math.random()));
5              if (result == true) resolve('Success!');
6              else reject(Error('Failed!!'));
7          }, 100);
8      });
9      return promise;
10 }
11 (function main() {
12     function onSuccess (value) {console.log (value)};
13     function onFailure (err) { console.log (err.message)};
14     const promise = someAsyncAction();
15     promise.then(onSuccess, onFailure);
16 })();
```

onFailure callback gets added to promise **Microtask Queue** and eventually executes

```
1  function someAsyncAction() {
2      const promise = new Promise((resolve, reject) => {
3          setTimeout(() => {
4              const result =  Boolean(Math.round(Math.random()));
5              if (result == true) resolve('Success!');
6              else reject(Error('Failed!!'));
7          }, 100);
8      });
9      return promise;
10 }
11 (function main() {
12     function onSuccess (value) {console.log (value)};
13     function onFailure (err) {console.log (err.message)};
14     const promise = someAsyncAction();
15     promise.then(onSuccess, onFailure);
16 })();
```

```
1  function someAsyncAction() {
2      const promise = new Promise((resolve, reject) => {
3          setTimeout(() => {
4              const result =  Boolean(Math.round(Math.random()));
5              if (result == true) resolve('Success!');
6              else reject(Error('Failed!!'));
7          }, 100);
8      });
9      return promise;
10 }
11 (function main() {
12     function onSuccess (value) {console.log (value)};
13     function onFailure (err) {   console.log (err.message)};
14     const promise = someAsyncAction();
15     promise.then(onSuccess, onFailure);
16 })();
```

Still callbacks?

Promise: Trust Principles

- A promise can only succeed or fail once. Once settled it guarantees to stay in the same state.
- It is immutable once settled. Calling resolve() or reject() multiple times has no effect on the state of the promise object.
- It guarantees to call success or failed callback only once, even when success/failure callback are added to a promise after it is settled.

Pitfall:
Broken Promise Chain

Activity: Can you spot an issue with this code?

```
1  const fs = require('fs').promises;
2  // NOTE: All fs promises API functions return a promise
3  function truncateFile() {
4      // make a copy of the file to truncate
5      fs.copyFile('large.txt', 'truncated.txt')
6          .then(function () {
7              // truncate file to first 100 bytes
8              fs.truncate('truncated.txt', 100);
9          })
10         .catch(function (err) {
11             console.error(err.message);
12         });
13 }
14 truncateFile();
```

Broken Promise Chain

```
1  const fs = require('fs').promises;
2  // NOTE: All fs promises API functions return a promise
3  function truncateFile() {
4      // make a copy of the file to truncate
5      fs.copyFile('large.txt', 'truncated.txt')
6          .then(function () {
7              // truncate file to first 100 bytes
8              fs.truncate('truncated.txt', 100);
9          })
10         .catch(function (err) {
11             console.error(err.message);
12         });
13 }
14 truncateFile();
```

Two things done by .then()

1. Adds handler function to internal callback list [[PromiseFulfillReactions]] [...]

which on completion of fs.copyFile() ends up in microtask queue and gets invoked

Broken Promise Chain

```
1  const fs = require('fs').promises;
2  // NOTE: All fs promises API functions return a promise
3  function truncateFile() {
4      // make a copy of the file to truncate
5      fs.copyFile('large.txt', 'truncated.txt')
6          .then(function () {
7              // truncate file to first 100 bytes
8              fs.truncate('truncated.txt', 100);
9          })
10         .catch(function (err) {
11             console.error(err.message);
12         });
13 }
14 truncateFile();
```

Two things done by .then()

2. Immediately returns a promise in a pending state, that can be used for chaining. This promise eventually gets settled based on the returned value of its handler function.

Fix: Broken Promise Chain

```
1  const fs = require('fs').promises;
2  // NOTE: All fs promises API functions return a promise
3  function truncateFile() {
4      // make a copy of the file to truncate
5      fs.copyFile('large.txt', 'truncated.txt')
6          .then(function () {
7              // truncate file to first 100 bytes
8              return fs.truncate('truncated.txt', 100);
9          })
10         .catch(function (err) {
11             console.error(err.message);
12         });
13 }
14 truncateFile();
```

Chained! Now the promise returned by .then() gets settled with the value of the promise returned by fs.truncate()

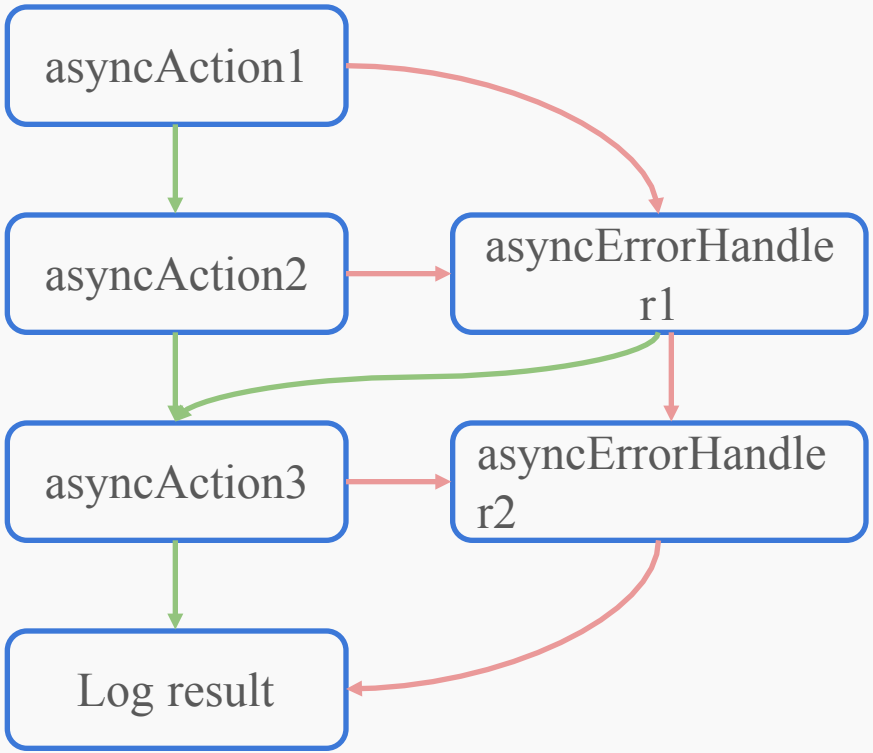
Escaping Callback Hell: Control of Execution and Data Flow

```
1  const fs = require('fs').promises;
2  // NOTE: All fs promises API functions return a promise
3  function truncateFile() {
4      // make a copy of the file to truncate
5      fs.copyFile('large.txt', 'truncated.txt')
6          .then(function () {
7              // truncate file to first 100 bytes
8              return fs.truncate('truncated.txt', 100);
9          })
10         .catch(function (err) {
11             console.error(err.message);
12         });
13 }
14 truncateFile();
```

When done right, **promise chaining** mechanism allows us to gain control over Execution flow and data flow in async programming!

Escaping Callback Hell: Control of Execution and Data Flow

```
1  //Assume all functions return either
2  a fulfilled or a rejected promise
3  asyncAction1()
4    .then(asyncAction2)
5    .catch(asyncErrorHandler1)
6    .then(asyncAction3)
7    .catch(asyncErrorHandler2)
8    .then((result) => {
9        console.log(result);
10   });
```



Pitfall:
Unhandled Promise Rejections

Activity: Can you spot an issue with this code?

```
1  const fsPromises = require('fs').promises;
2  function doTruncate() {
3      let filehandle = null;
4      return fsPromises.open('large.txt', 'r+')
5          .then(result => {
6              filehandle = result;
7              return filehandle.truncate(100); // Keep first 100 bytes
8          })
9          .finally(() => {
10             if (filehandle) { return filehandle.close();} // Close file descriptor
11         })
12 }
13 doTruncate();
```

Activity: Can you spot an issue with this code?

```
1  const fsPromises = require('fs').promises;
2  function doTruncate() {
3      let filehandle = null;
4      return fsPromises.open('large.txt', 'r+')
5          .then(result => {
6              filehandle = result;
7              return filehandle.truncate(100); // Keep first 100 bytes
8          })
9          .finally(() => {
10             if (filehandle) { return filehandle.close();} // Close file descriptor
11         })
12 }
13 doTruncate().catch(console.error);
```

Issues with UnhandledPromiseRejectionWarning

(node:10764) UnhandledPromiseRejectionWarning: Unhandled promise rejection. **This error originated either by throwing inside of an async function without a catch block, or by rejecting a promise which was not handled with .catch(). (rejection id: 1)**

(node:10764) [DEP0018] DeprecationWarning: Unhandled promise rejections are deprecated. In the future, promise rejections that are not handled will terminate the Node.js process with a non-zero exit code.

Issues with UnhandledPromiseRejectionWarning

- ➔ Difficult to debug errors
- ➔ Application in unstable state
- ➔ Memory and file descriptor leaks

DEP0018: Unhandled promise rejections

▼ History

Version	Changes
v7.0.0	Runtime deprecation.

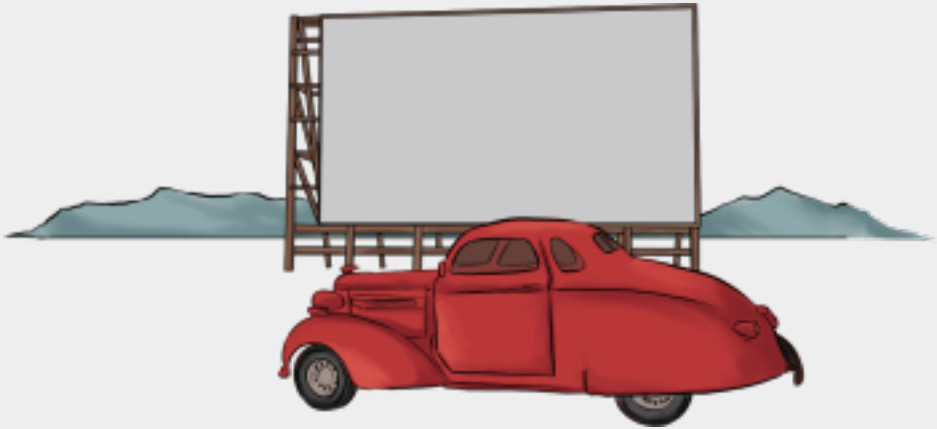
Type: Runtime

Unhandled promise rejections are deprecated. In the future, promise rejections that are not handled will terminate the Node.js process with a non-zero exit code.

Handling Unhandled Promise Rejections

- Handle rejected promises with `.catch()` or equivalent methods.
- Raise the unhandled rejection as an uncaught exception using:
  - `--unhandled-rejections=strict` node option (added in v12)
  - `make-promises-safe` npm package
  - `--throw-deprecation` node option (use with caution. It will convert all runtime deprecation warnings to error).

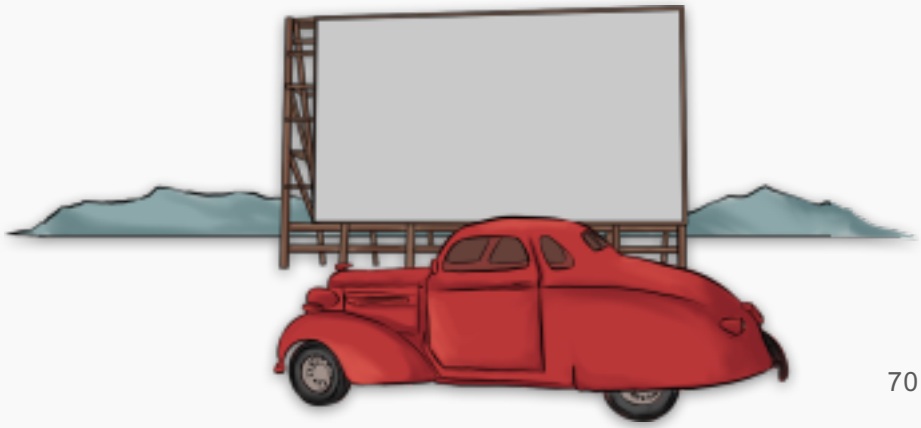
async / await



Buick Special (Year 1940)

async / await

- Allows us to write async program in a synchronous style
- Simplifies error handling by using try/catch block like synchronous programs



async/await key Principles

- Inside a function marked as async, always returns a promise, which:
  - resolves with the value returned by the function,
  - or rejects with an uncaught exception thrown from within the function

async function

```
1  async function someFunction() {
2
3  }
4
5  const result = someFunction();
6
7
8
9
10
11
12
13
```

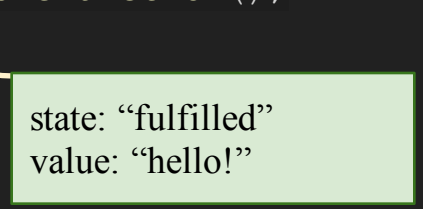
state: “fulfilled”
value: undefined

async function:

Always returns a promise

async function

```
1  async function someFunction() {
2      return 'hello!';
3  }
4
5  const result = someFunction();
6
7
8
9
10
11
12
13
```



async function:

Always returns a promise, which -

resolves with the value returned by the function,

async function

```
1  async function someFunction() {
2      throw Error('Boo!!');
3  }
4
5  const result = someFunction();
6
7
8
9
10
11
12
13
```

state: “rejected”
value: error

async function:

Always returns a promise, which -

rejects with an uncaught exception thrown from within the function

async function

```
async function someFunction() {
    throw Error('Boo!!');
}

const result = someFunction();
result.catch(err => console.log(err.message)); //-> Boo!!
```

async/await key Principles

- Inside a function marked as async, you can place the await keyword in front of an expression.
- If the await keyword is used in front of an expression that returns a promise, the execution of async function pauses until the promise returned by the expression settles.

await keyword

```
async function someFunction() {
    const result = await new Promise((resolve) => {
      setTimeout(() => { resolve('Hello!'); }, 1000);
    });
    console.log(result); //-> Hello!

}
someFunction();
```

With await keyword :

execution of async function pauses until the promise returned by the expression on its right side settles.

await keyword

```
1  async function someFunction() {
2      try {
3          await new Promise((resolve, reject) => {
4              setTimeout(() => { reject('Boo!!'); }, 1000);
5          });
6          console.log(result);
7      } catch (err) {
8          console.log(err); //-> Boo!!
9      }
10 }
11 someFunction();
12
13
```

With await keyword :

execution of async function pauses until the promise returned by the expression on its right side settles.

await keyword

```
1  async function bar() {
2      throw Error('Boom!');
3  }
4  async function foo() {
5      const result = await bar();
6      return result;
7  }
8  async function  main() {
9      var result = await foo();
10     return result;
11 }
12 main();
13
14
```

Async stack traces with async/await:

```
(node:32596) UnhandledPromiseRejectionWarning: Error:
Boom!
    at bar (/example.js:2:11)
    at foo (/example.js:5:26)
    at main (/example.js:9:24)
    at Object.<anonymous> (/example.js:14:1)
    ...
```

Pitfalls:
Common Pitfalls with async/await

Activity: Can you spot an issue with this code?

```
1  async function foo() {
2      throw Error('foo!');
3  }
4  async function bar() {
5      throw Error('bar!!');
6  }
7  (async function  main() {
8      const a = foo();
9      const b = bar();
10     try {
11         await a;
12         await b;
13     } catch(err) { console.log(err.message);}
14 })();
```

FIX

```
async function foo() {
    throw Error('foo!');
}
async function bar() {
    throw Error('bar!!');
}
(async function main() {
    const a = foo();
    const b = bar();
    try {
        await Promise.all([a, b]);
    } catch(err) { console.log(err.message);}
})();
```

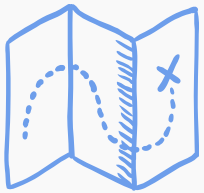
Activity: Can you spot an issue with this code?

```
async function foo() {
    // Wait for a second and then throw an error
    await new Promise(resolve => setTimeout(resolve, 1000));
    throw Error('foo!');
}

(async function main() {
    try {
        return foo();
    } catch(err) {
        console.log(err.message);
    }
})();
```

FIX

```
async function foo() {
    // Wait for a second and then throw an error
    await new Promise(resolve => setTimeout(resolve, 1000));
    throw Error('foo!');
}

(async function main() {
    try {
        return await foo();
    } catch(err) {
        console.log(err.message);
    }
})();
```

Roadmap



Async Programming Essentials

- Async Programming Features in JavaScript
- Node.js Event Loop Phases

Applied Async and Parallel Programming

- Handling I/O Operations
- Handling Memory Intensive Operations
- Handling CPU Heavy Operations

Node.js Async Programming APIs

- Scheduling Timers API
  - setImmediate(callback[, ...args])
  - setInterval(callback, delay[, ...args])
  - setTimeout(callback, delay[, ...args])
  - Corresponding clear timer APIs
- Process
  - process.nextTick(callback[, ...args])
- Node.js Asynchronous I/O APIs

Pitfall:
Memory and I/O Starvation

Denial of Service (DoS)

Affecting **podium** package, versions **<1.2.5 >=1.2.2**

Overview

`podium` is Node compatible event emitter with extra features. Affected versions of the package are vulnerable to Denial of Service (DoS). When calling the `setImmediate()` multiple times, the callbacks are queued for execution and is processed every event loop iteration. This makes it possible for attacker to manipulate calls to the server and cause it consume a high amount of CPU, before crashing.

https://snyk.io/vuln/npm:podium:20160913

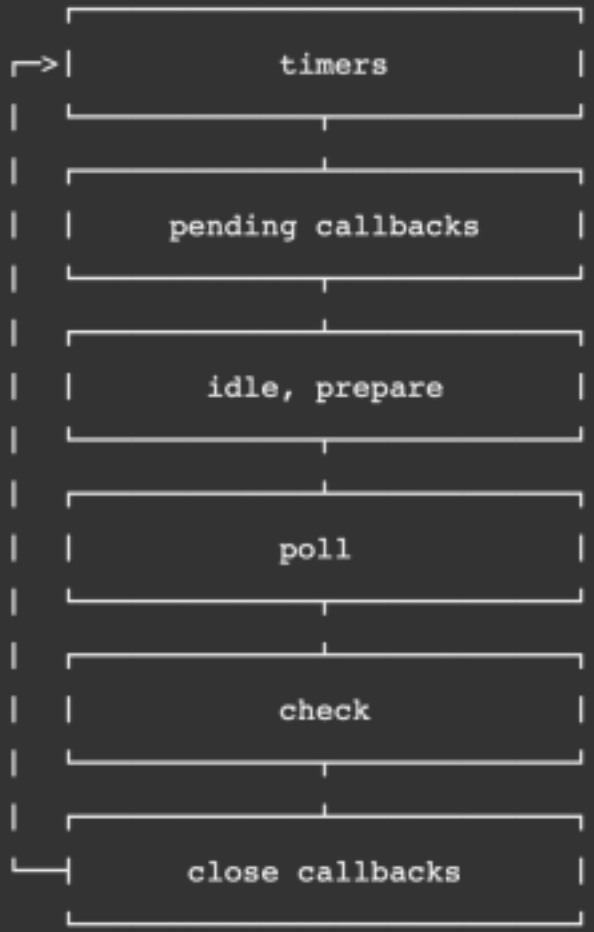
Fix to eliminate closures to reduce memory

13 lib/index.js

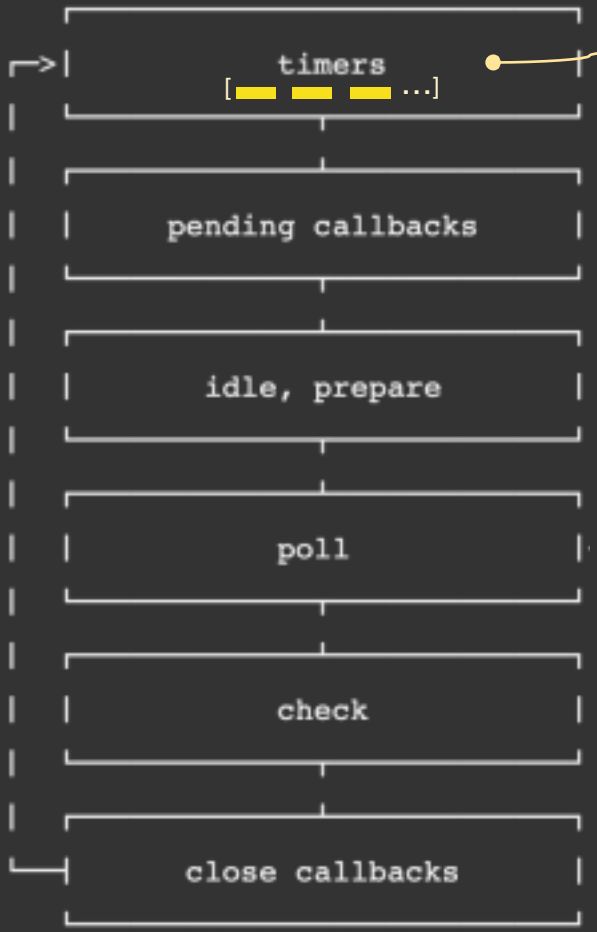
```
                                                  +const itemCallback = function (item) {
                                                  +
                                                  +    item.callback();
                                                  +};
                                                  +
                                                  +const emitEmitter = function (emitter) {
                                                  +
                                                  +    internals.emit(emitter);
                                                  +};

 internals.emit = function (emitter, notification) {

@@ -162,11 +171,11 @@ internals.emit = function (emitter, notification) {
     const finalize = () => {

         if (item.callback) {
-            setImmediate(() => item.callback());
+            process.nextTick(itemCallback, item);
         }

         emitter._eventsProcessing = false;
-        setImmediate(() => internals.emit(emitter));
+        process.nextTick(emitEmitter, emitter);
     };
```

Fix to eliminate closures to reduce memory

Result of load test:
Reduced memory usage from 1 GB to 100-150 MB

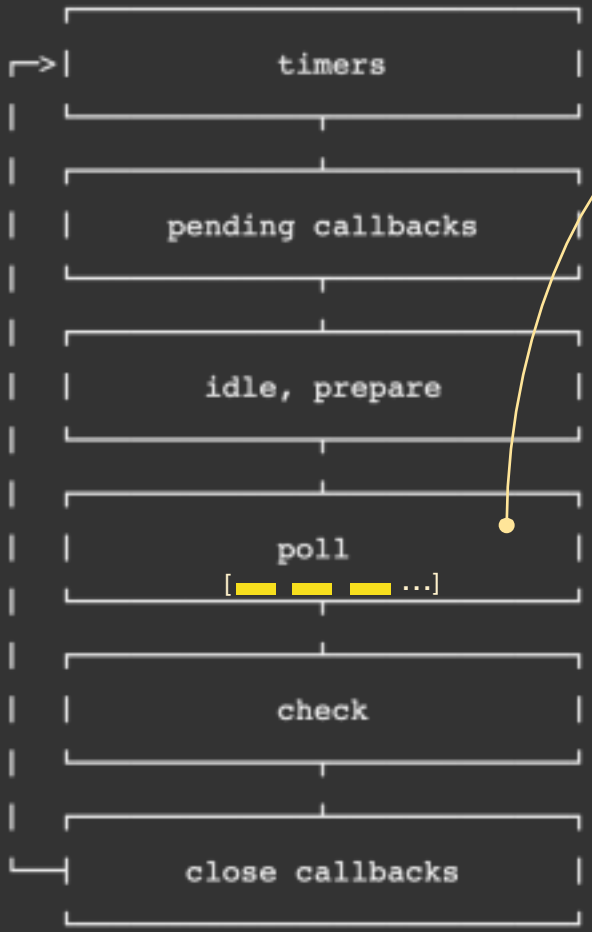


https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/



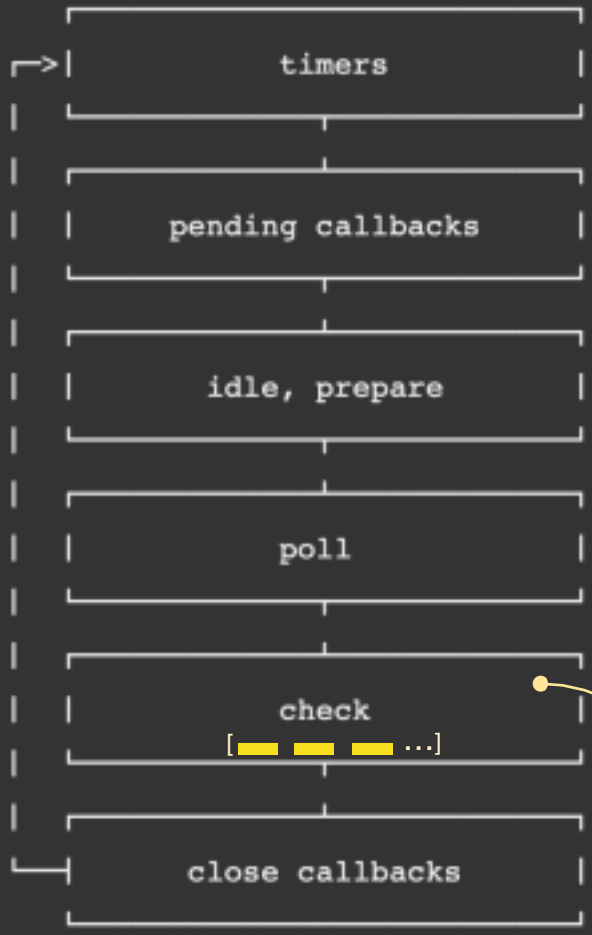
Timer Phase processes callbacks for:
setTimeout() and setInterval()
after the specified amount of time has passed

```
setTimeout(function () {...}, 0);
setInterval(function () {...}, 200);
```



Poll Phase processes callbacks for async I/O operations

```
fs.readFile(fullPath, function (err, data) {...});
```



Check phase processed callback for setImmediate()

```
setImmediate(function () { ... })
```

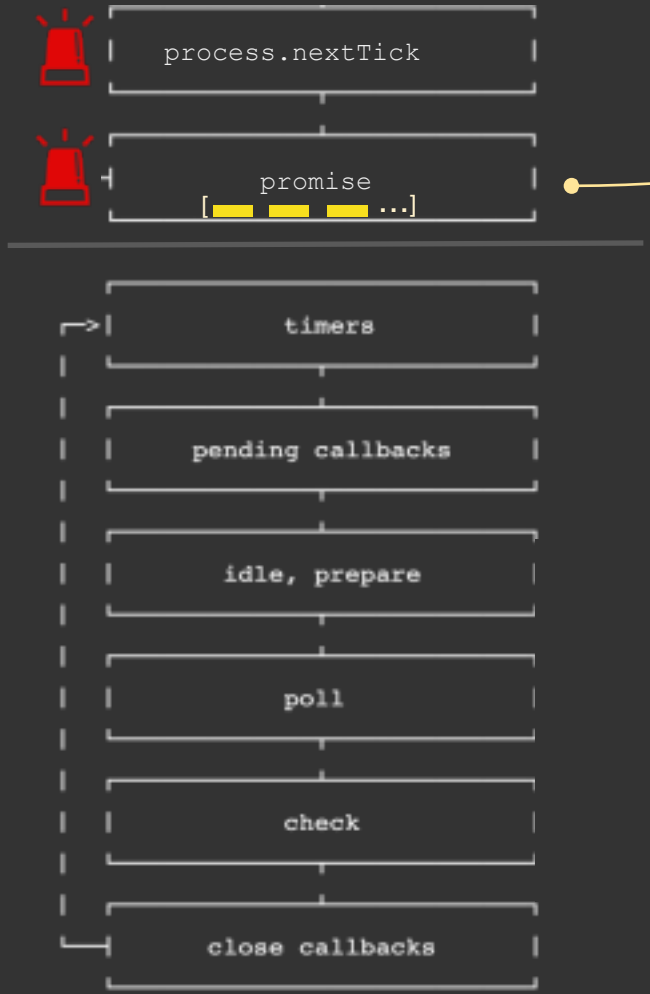
Process.nextTick Microtask Queue:
Processed immediately after the event loop's current operation is completed, regardless of the current phase of the event loop.

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

Confusing names:

process.nextTick() fires immediately on the same phase

setImmediate() fires on the following iteration or 'tick' of the event loop



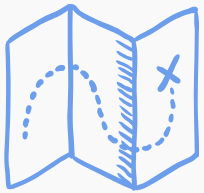
Promises Microtask Queue:

Once a promise is settles, or has already settled, a microtask gets queues for executing its callbacks.

```
promise.then(function () { ... })
```

Activity:
Identify the order of execution of async functions

Roadmap

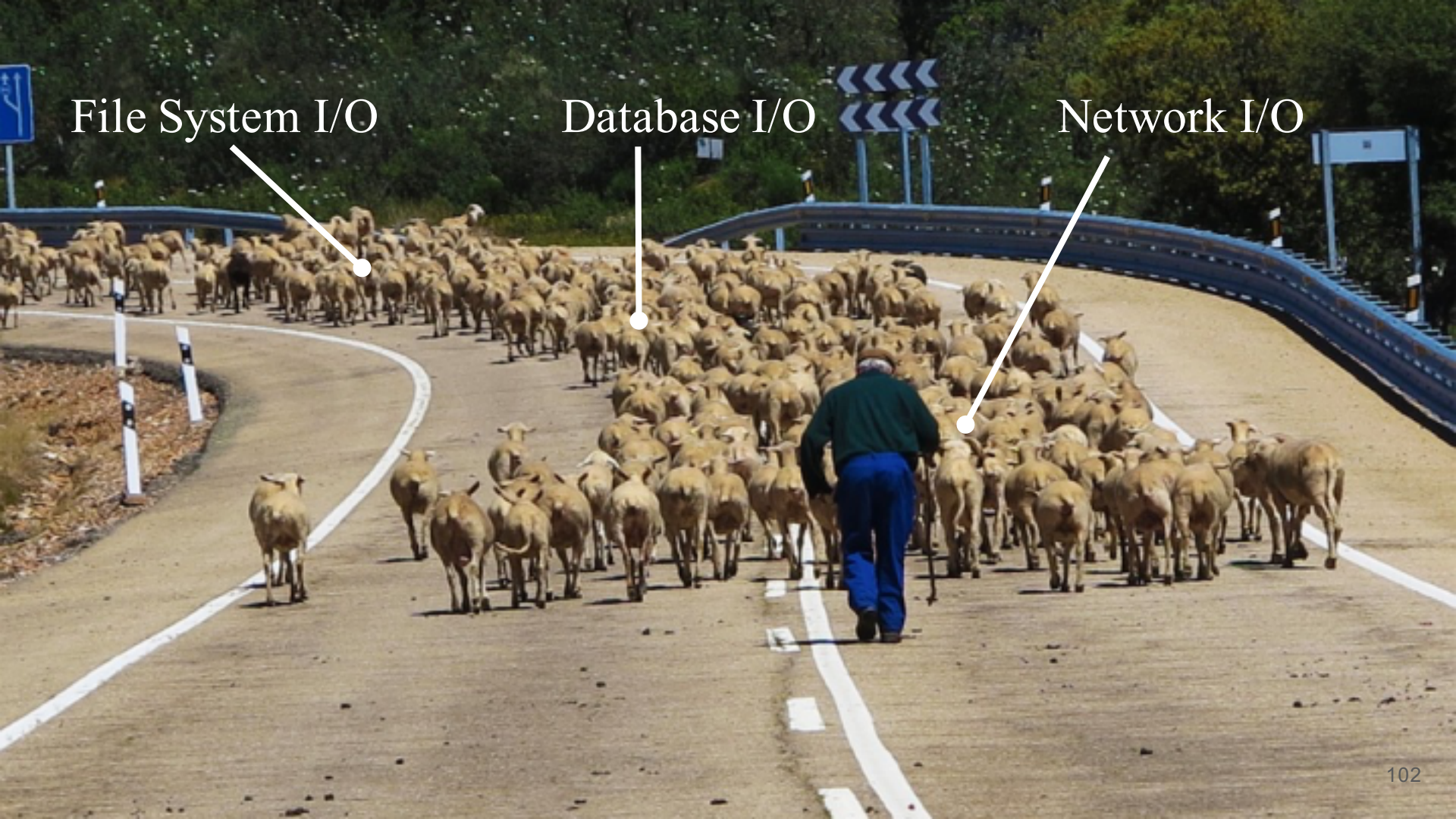


Async Programming Essentials

- Async Programming Features in JavaScript
- Node.js Event Loop Phases

Applied Async and Parallel Programming

- Handling I/O Operations
- Handling Memory Intensive Operations
- Handling CPU Heavy Operations


File System I/O
Database I/O
Network I/O

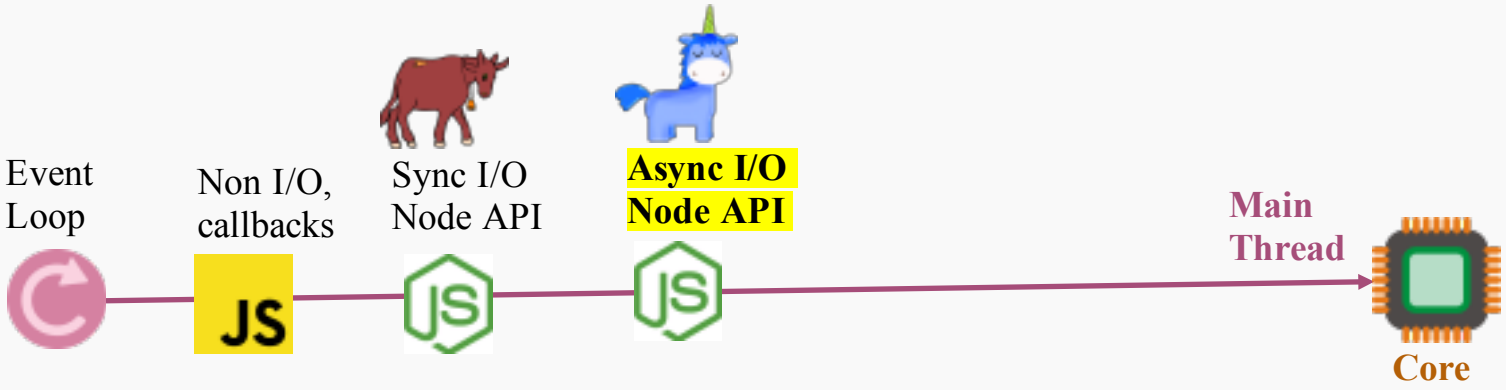
- Callbacks
- Non I/O or CPU intensive Node APIs
- Application logic

Examples:

- fs.readFileSync()
- zlib.gzipSync()

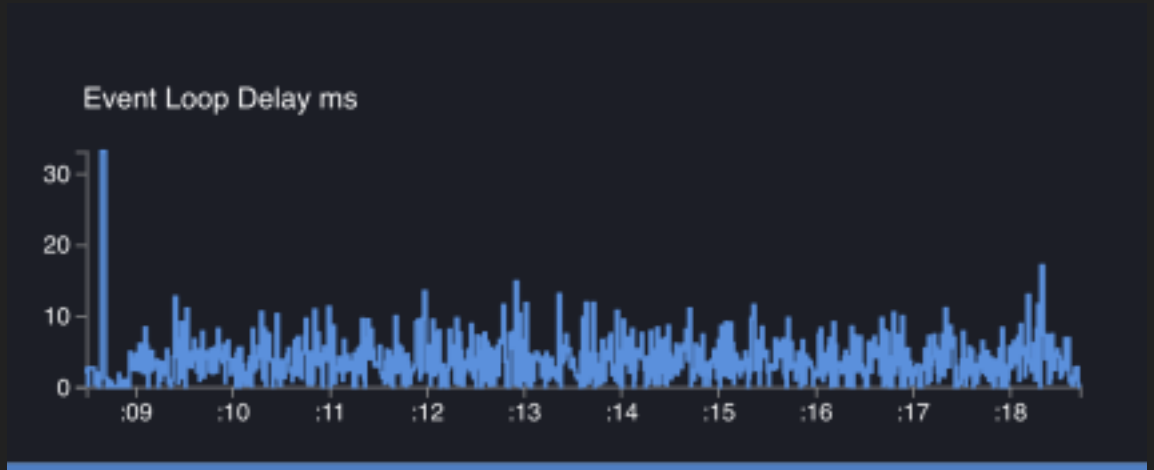
Synchronous APIs

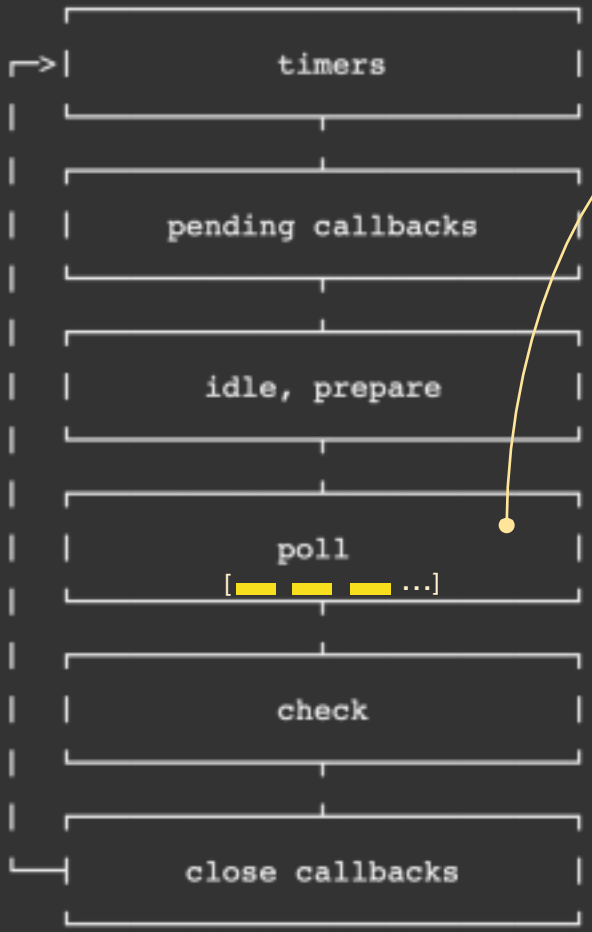
```
1  const serveFileSync = function (req, res) {
2      try {
3          const data = fs.readFileSync(fullPath);
4          res.writeHead(200, { 'Content-Type': 'text/plain' });
5          res.write(data);
6          return res.end();
7
8      } catch (err) {
9          terminate(res);
10     }
11 };
12 }
13
14
```


Event
Loop
Non I/O,
callbacks
JS
Sync I/O
Node API
Async I/O
Node API
Main
Thread
Core

Async APIs

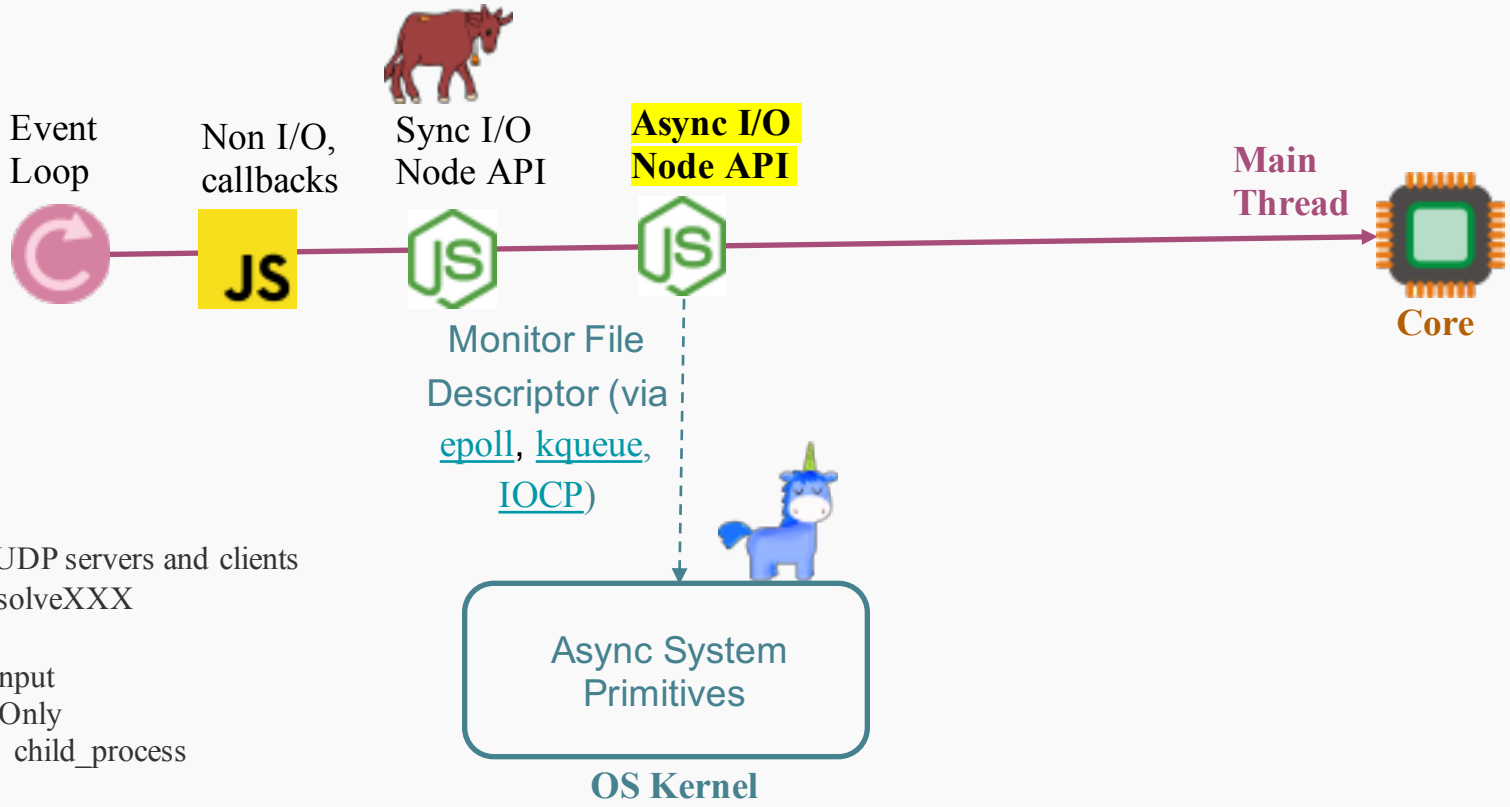
```
1  const serveFileAsync = function (req, res) {
2      fs.readFile(fullPath, function (err, data) {
3          if (err) {
4              terminate(res);
5          }
6          res.writeHead(200, { 'Content-Type': 'text/plain' });
7          res.write(data);
8          return res.end();
9      });
10 };
11
12
13
14
```





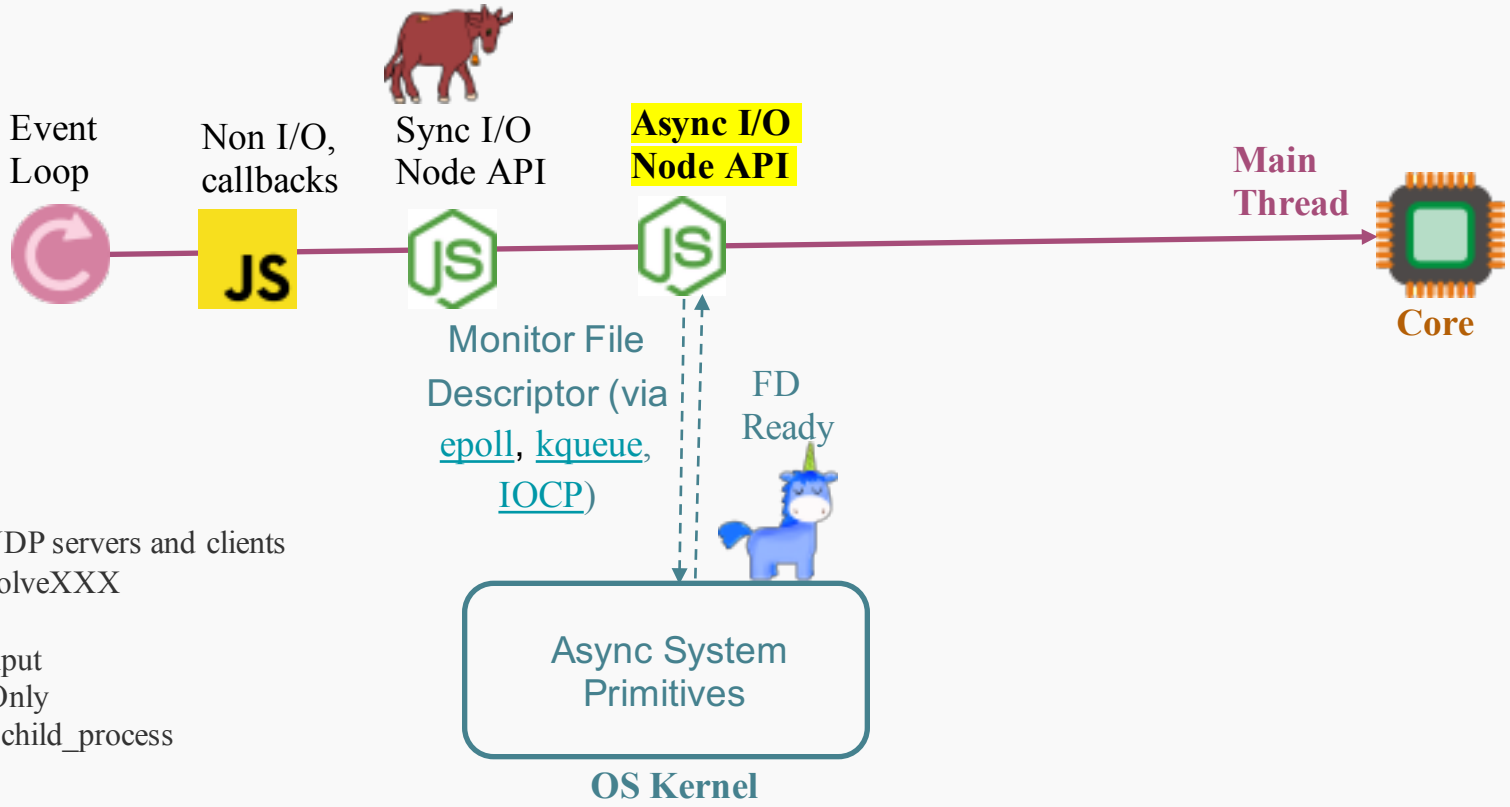
Poll Phase processes callbacks for async I/O operations

```
fs.readFile(fullPath, function (err, data) {...});
```



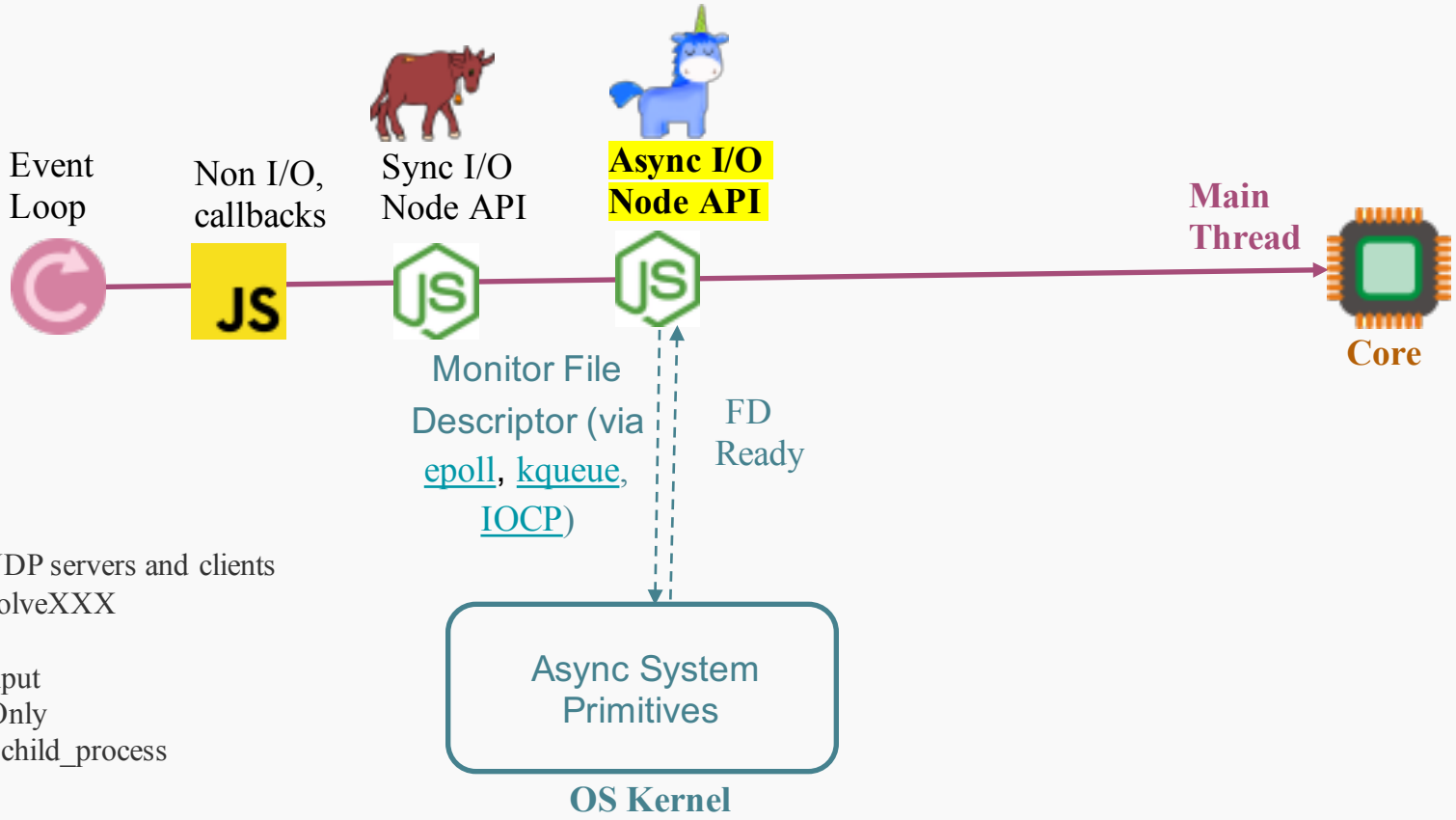
Examples:

- TCP/ UDP servers and clients
- dns.resolveXXX
- Pipes
- TTY Input
- *NIX Only
  - child_process



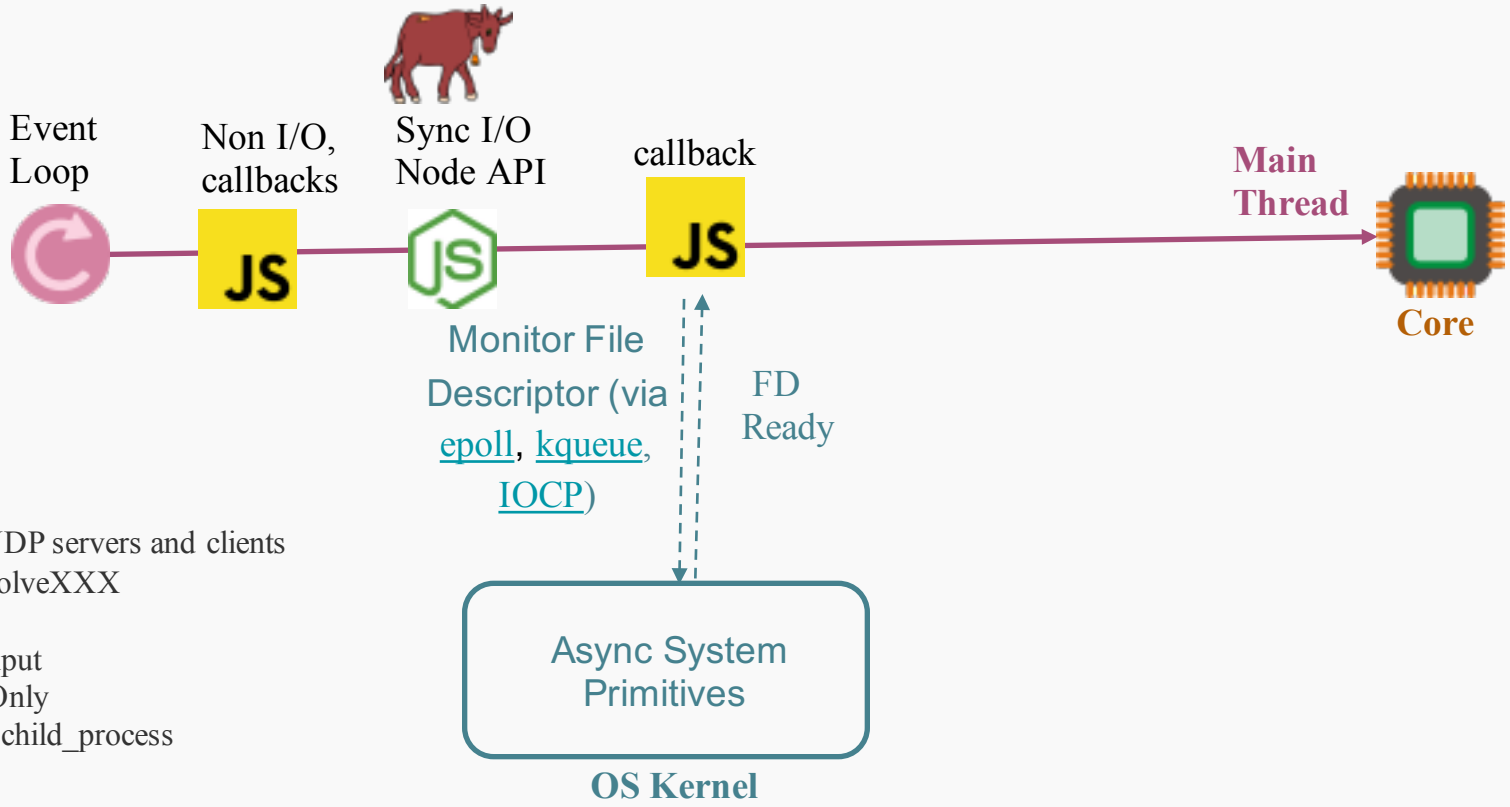
Examples:

- TCP/ UDP servers and clients
- dns.resolveXXX
- Pipes
- TTY Input
- *NIX Only
  - child_process



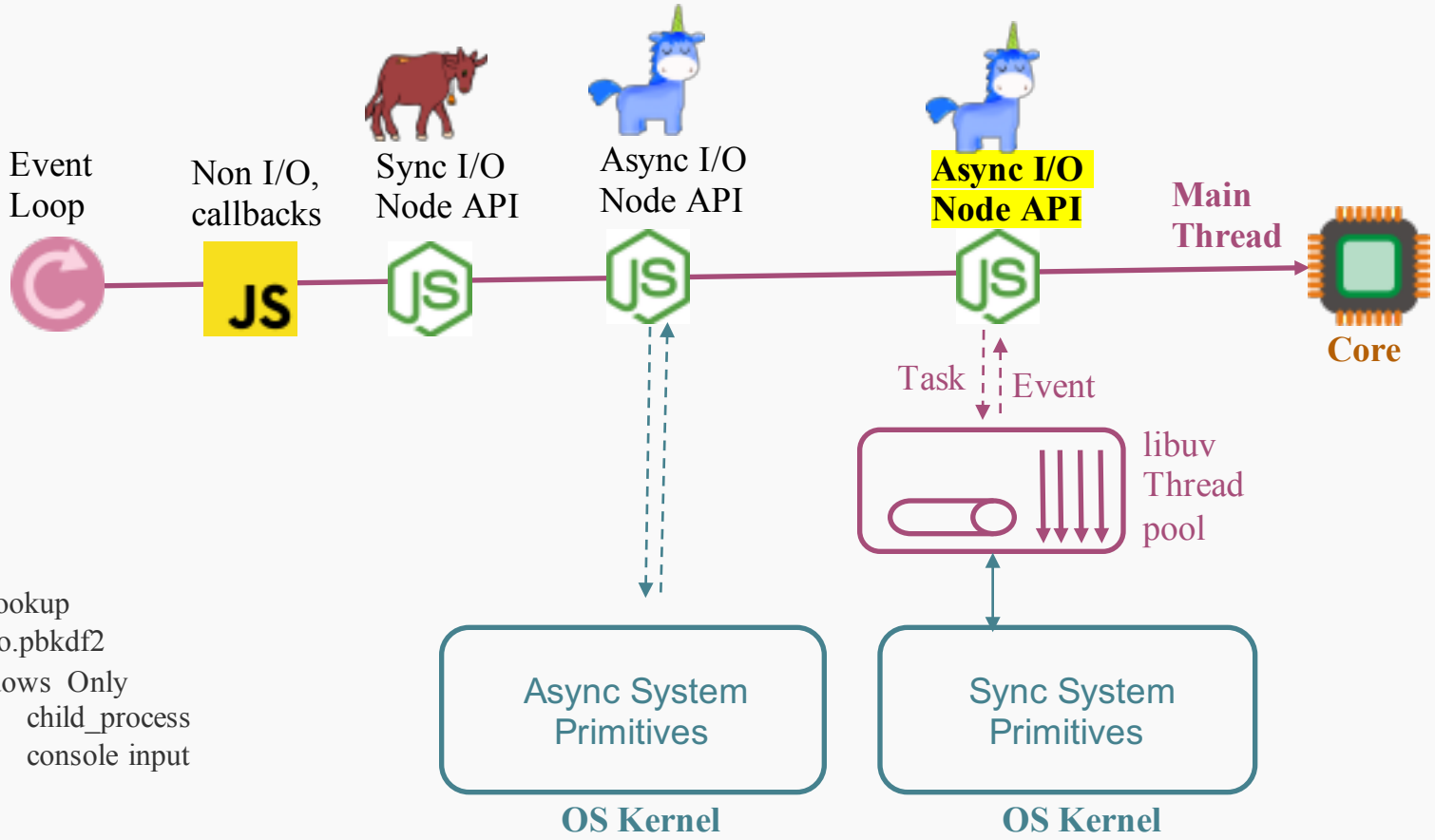
Examples:

- TCP/ UDP servers and clients
- dns.resolveXXX
- Pipes
- TTY Input
- *NIX Only
  - child_process



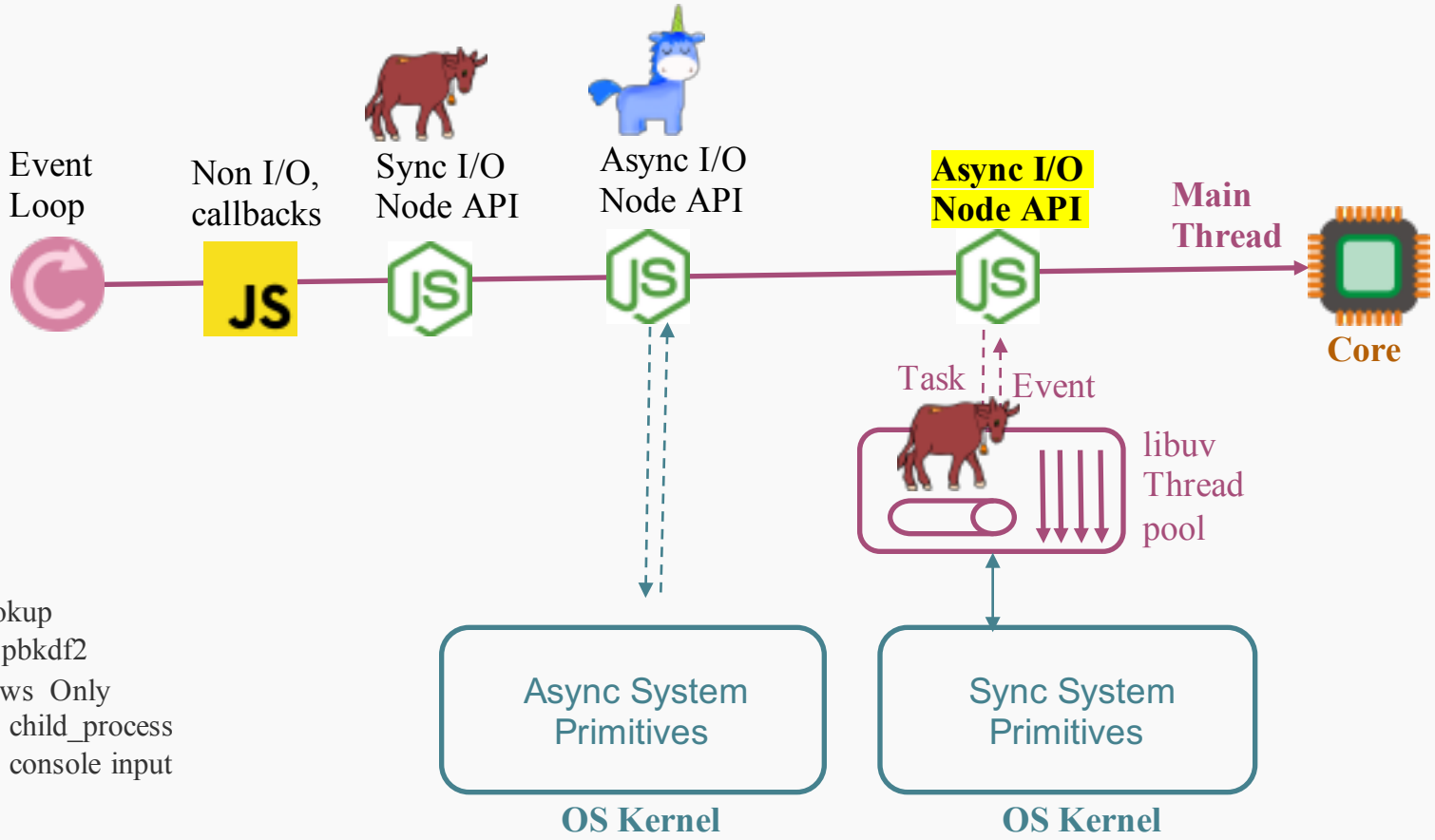
Examples:

- TCP/ UDP servers and clients
- dns.resolveXXX
- Pipes
- TTY Input
- *NIX Only
  - child_process



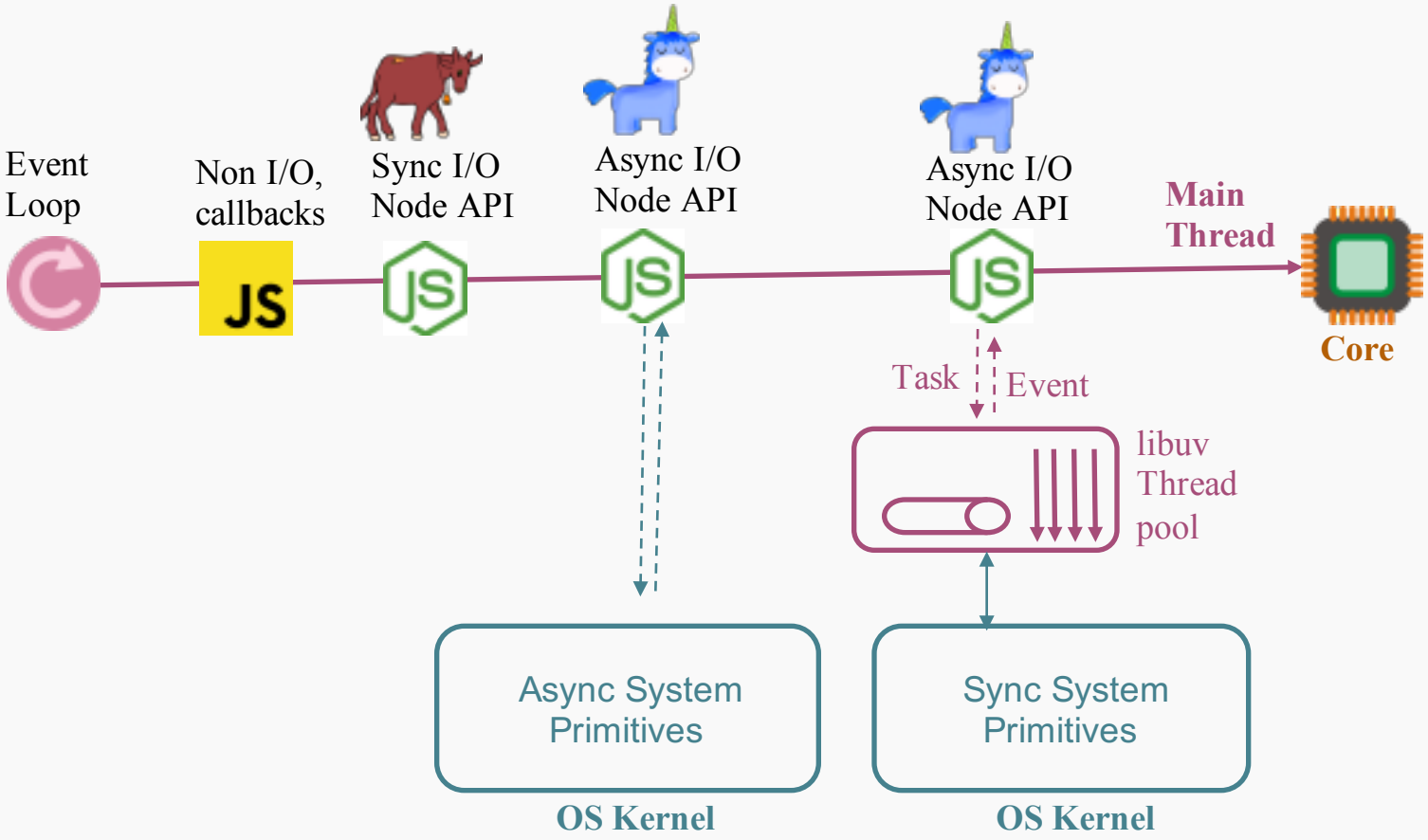
Examples:

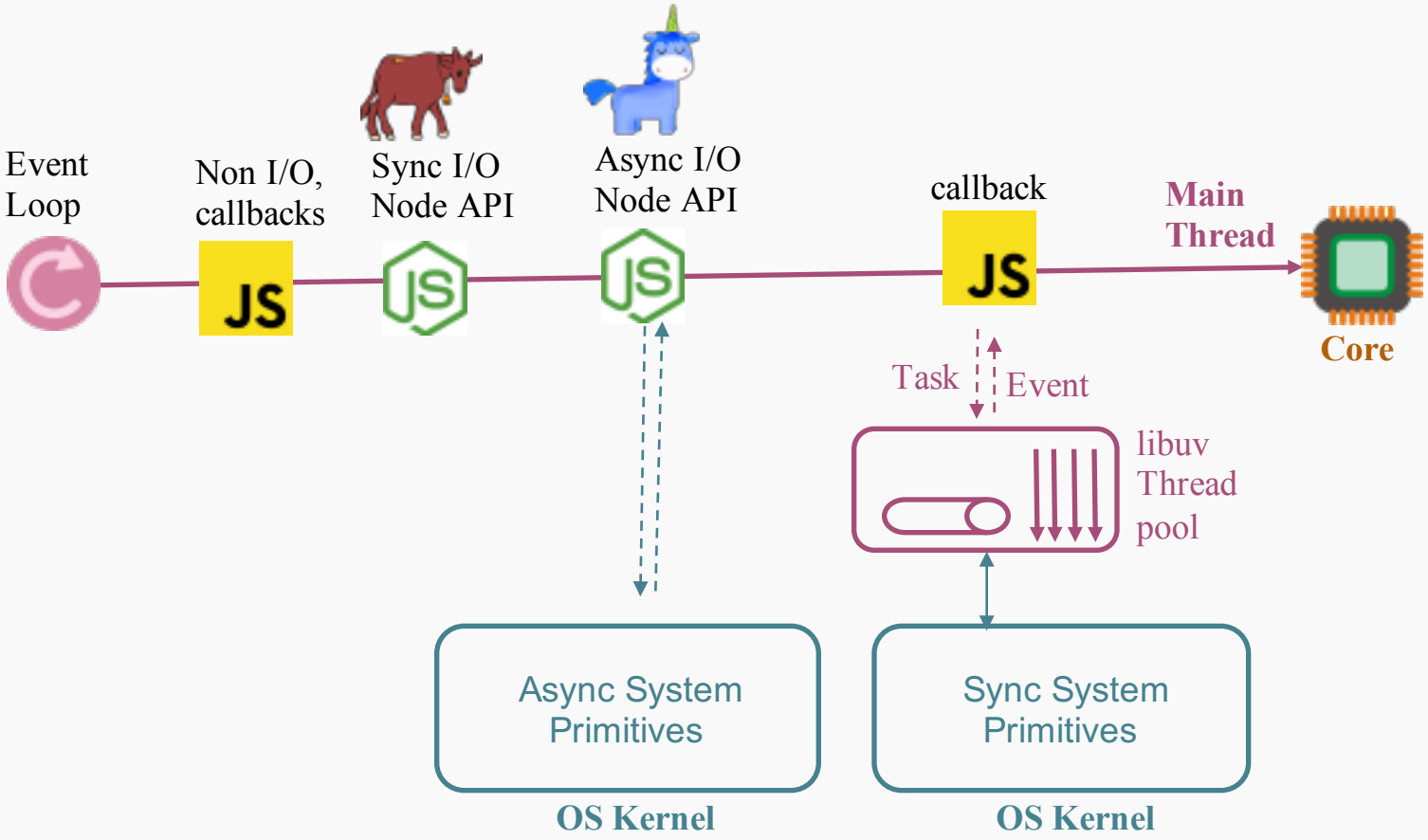
- fs.*
- dns.lookup
- crypto.pbkdf2
- Windows Only
  - child_process
  - console input

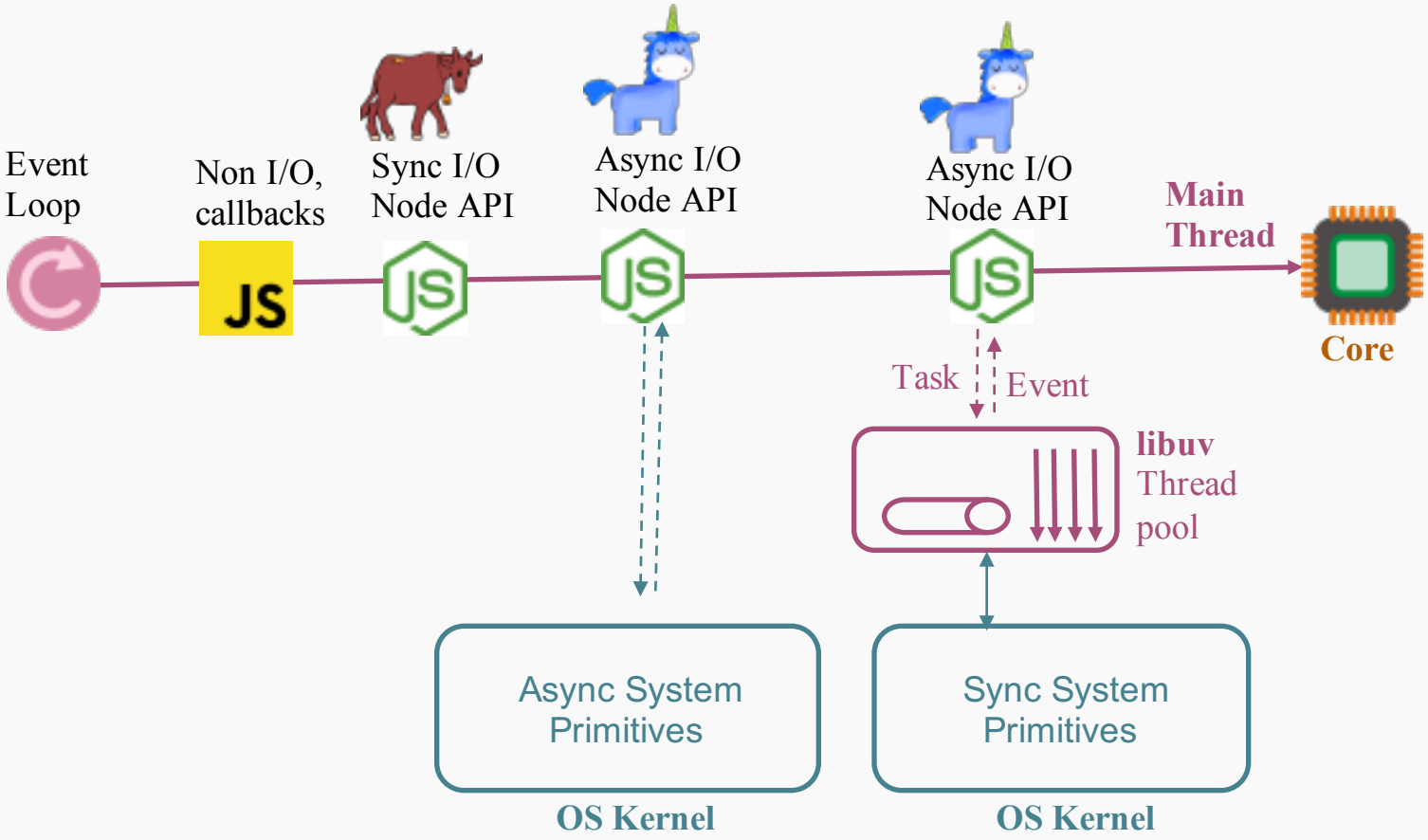


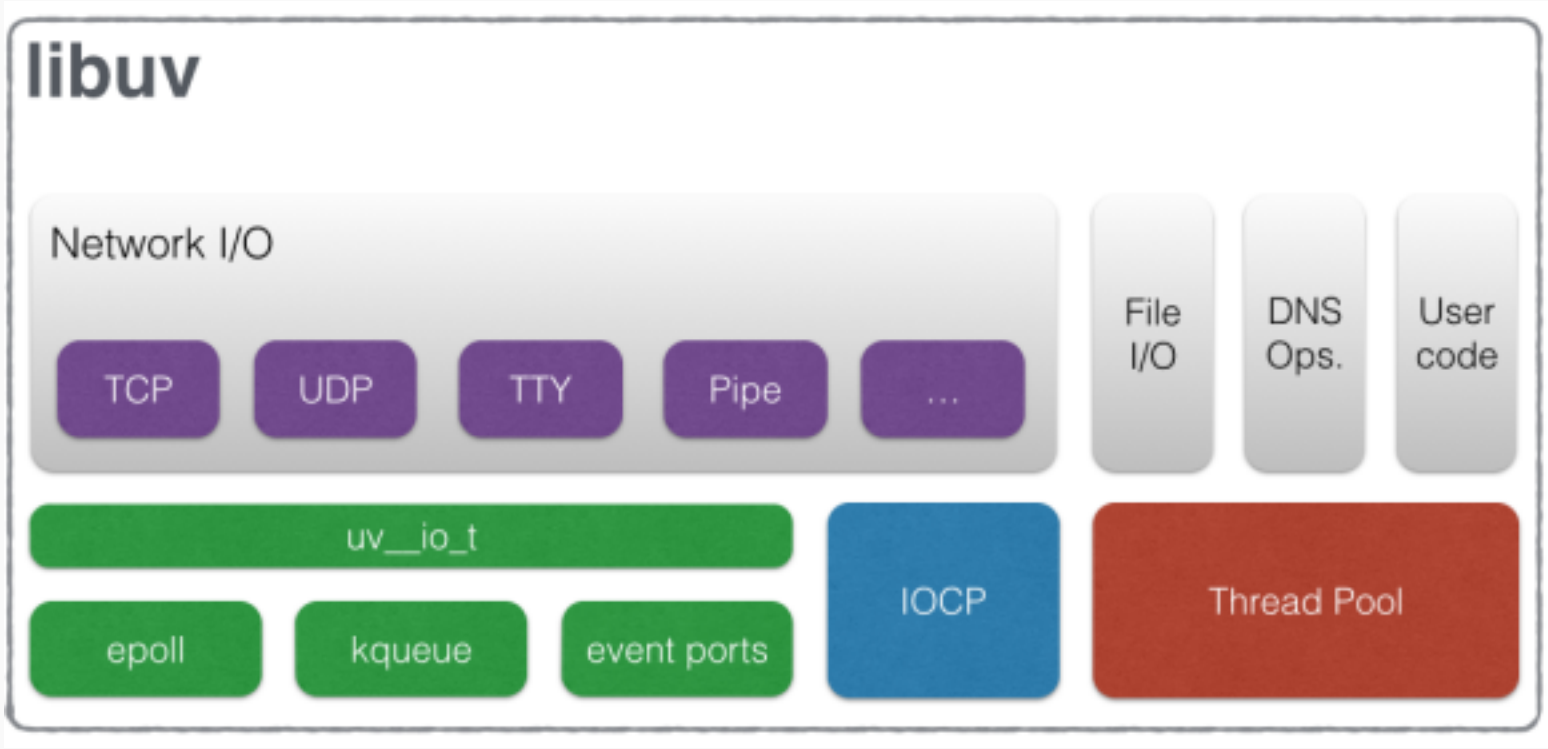
Examples:

- fs.*
- dns.lookup
- crypto.pbkdf2
- Windows Only
  - child_process
  - console input


Event
Loop
Non I/O,
callbacks
JS
Sync I/O
Node API
Async I/O
Node API
Async I/O
Node API
Main
Thread
Core
Task
Event
libuv
Thread
pool
Async System
Primitives
Sync System
Primitives
OS Kernel
OS Kernel

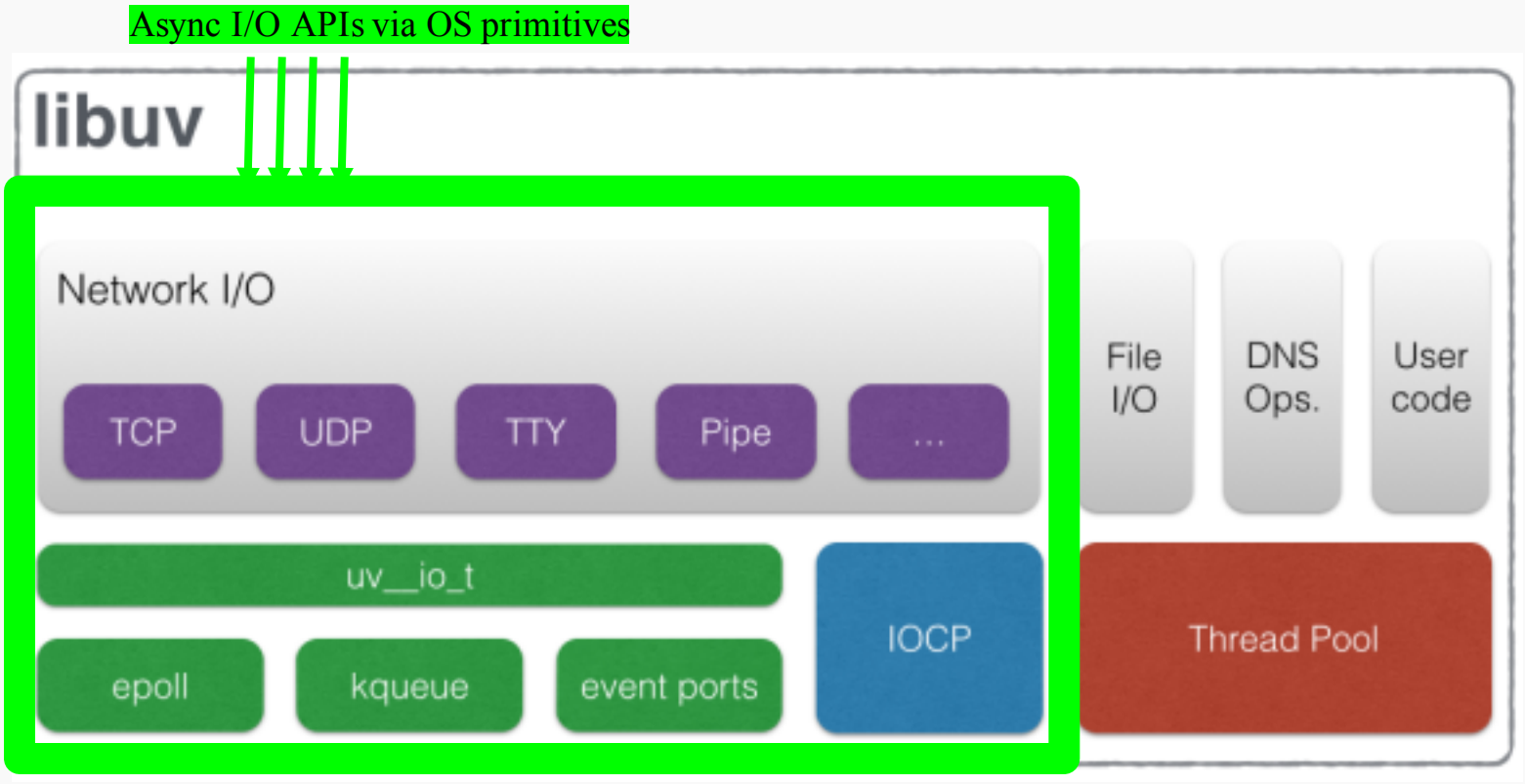

Event
Loop
Non I/O,
callbacks
Sync I/O
Node API
Async I/O
Node API
callback
Main
Thread
Core
Task
Event
libuv
Thread
pool
Async System
Primitives
Sync System
Primitives
OS Kernel
OS Kernel


Event Loop
Non I/O, callbacks
Sync I/O Node API
Async I/O Node API
Async I/O Node API
Main Thread
Core
Task
Event
libuv Thread pool
Async System Primitives
Sync System Primitives
OS Kernel
OS Kernel



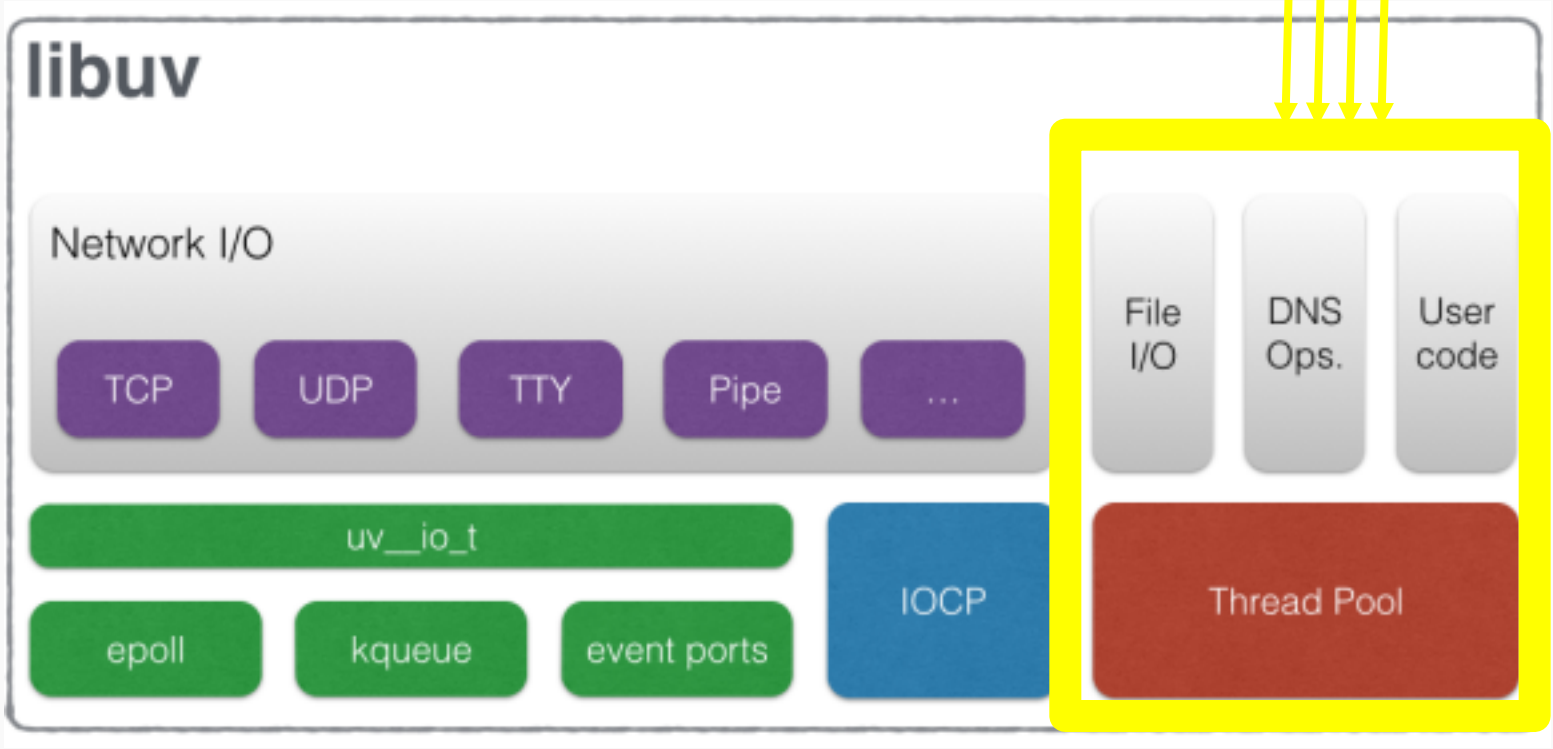
http://docs.libuv.org/en/latest/design.html

Libuv Internal Design



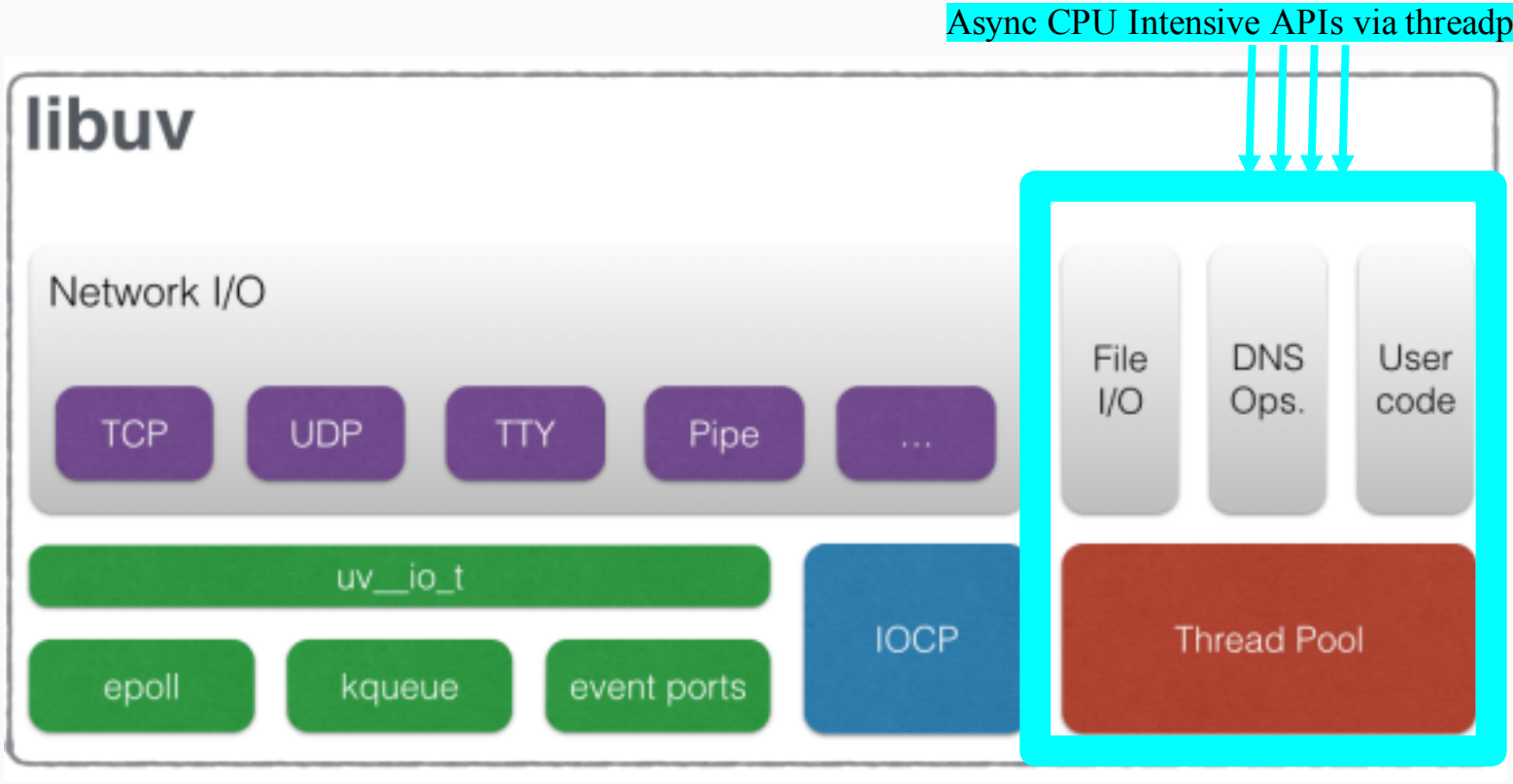
http://docs.libuv.org/en/latest/design.html

Libuv Internal Design



http://docs.libuv.org/en/latest/design.html

Libuv Internal Design



http://docs.libuv.org/en/latest/design.html

Libuv Internal Design

Pitfall:

Blocking event loop and worker threads

Bound user inputs

To prevent Malicious inputs:

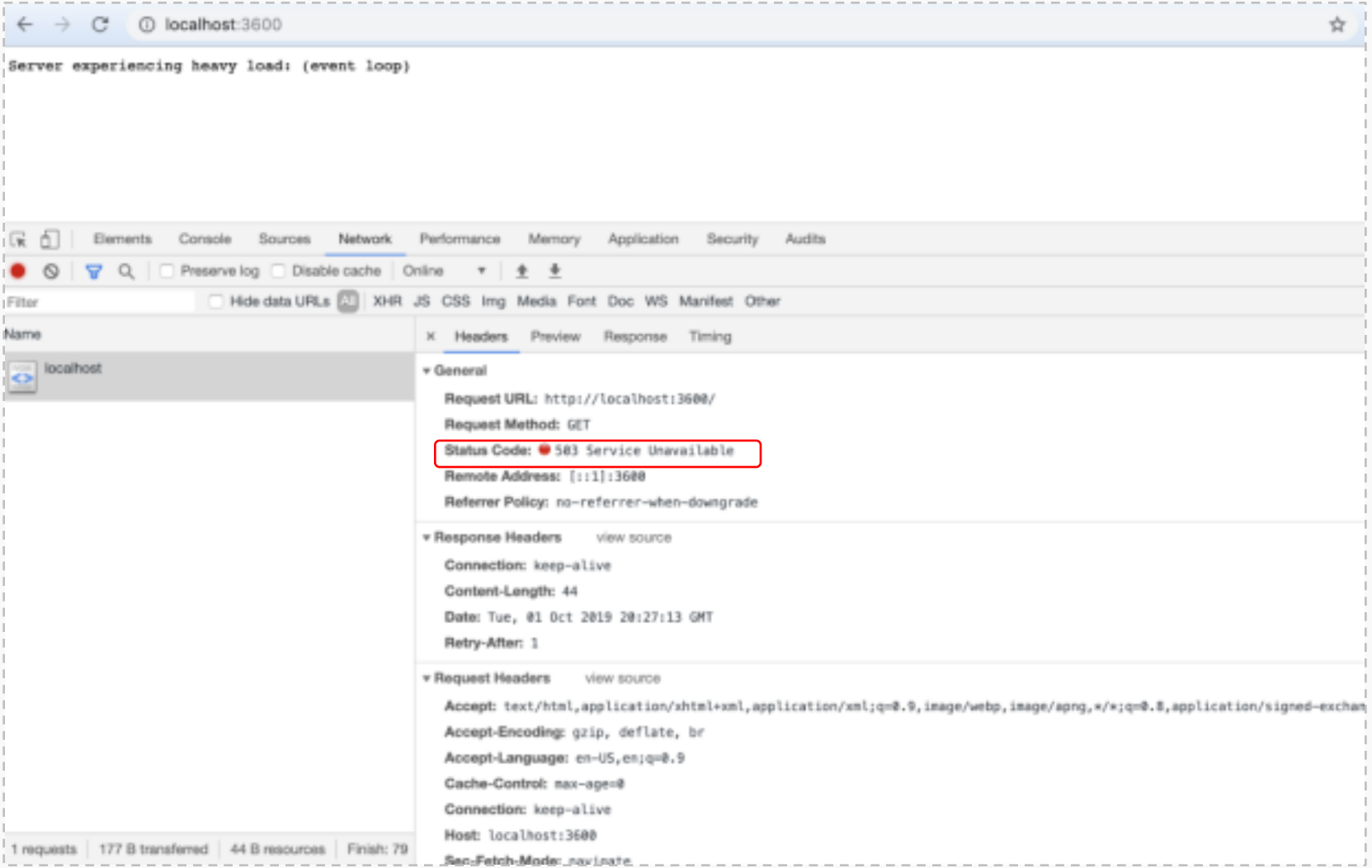
- When execution time of an I/O callback function is dependent on a user input, limit the size and reject inputs that are too long.

Configure Overload Protection

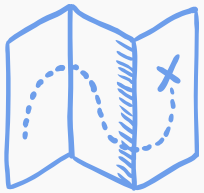
- Involves monitoring event loop response time. If it exceeds a threshold, a HTTP 503 Service Unavailable is sent.
- Allows load balancer to route traffic to a different service instance

Tools:

- Express, Koa, Restify, http: overload-protection
- Hapi: Server load sampleInterval option
- Fastify: under-pressure


localhost:3600
Server experiencing heavy load: (event loop)
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Audits
Preserve log
Disable cache
Online
Filter
Hide data URLs
All
XHR
JS
CSS
Img
Media
Font
Doc
WS
Manifest
Other
Name
localhost
Headers
Preview
Response
Timing
General
Request URL: http://localhost:3600/
Request Method: GET
Status Code: 503 Service Unavailable
Remote Address: [::1]:3600
Referrer Policy: no-referrer-when-downgrade
Response Headers
view source
Connection: keep-alive
Content-Length: 44
Date: Tue, 01 Oct 2019 20:27:13 GMT
Retry-After: 1
Request Headers
view source
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchan
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Cache-Control: max-age=0
Connection: keep-alive
Host: localhost:3600
Sec-Fetch-Mode: navigate
1 requests
177 B transferred
44 B resources
Finish: 79

Roadmap



Async Programming Essentials

- Async Programming Features in JavaScript
- Node.js Event Loop Phases

Applied Async and Parallel Programming

- Handling I/O Operations
- Handling Memory Intensive Operations
- Handling CPU Heavy Operations

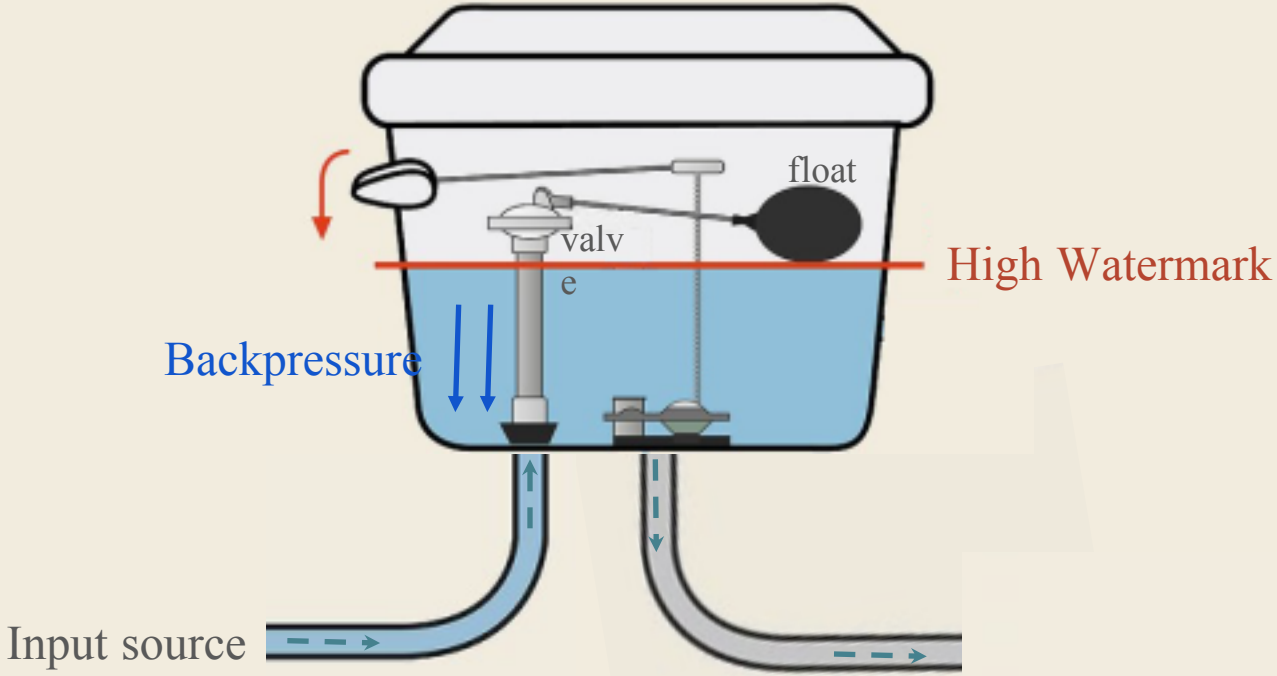
How to do async handling of multiple values?

How to deal with large data in async operations?

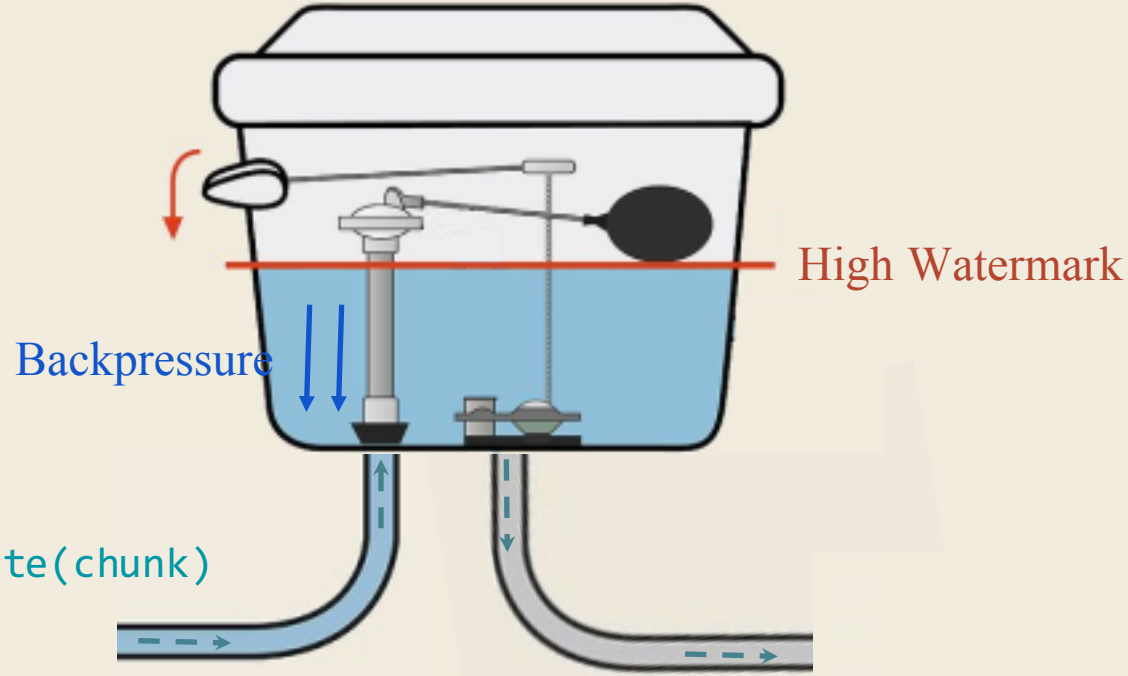
Streams

Node.js Streams = Event Emitter on Steroids

- Allows asynchronous handling of large data in chunks
- Allows consumer communicate with producer when overwhelmed or ready for more data
- Allows transforming data as it comes through


float
valv
e
High Watermark
Backpressure
Input source

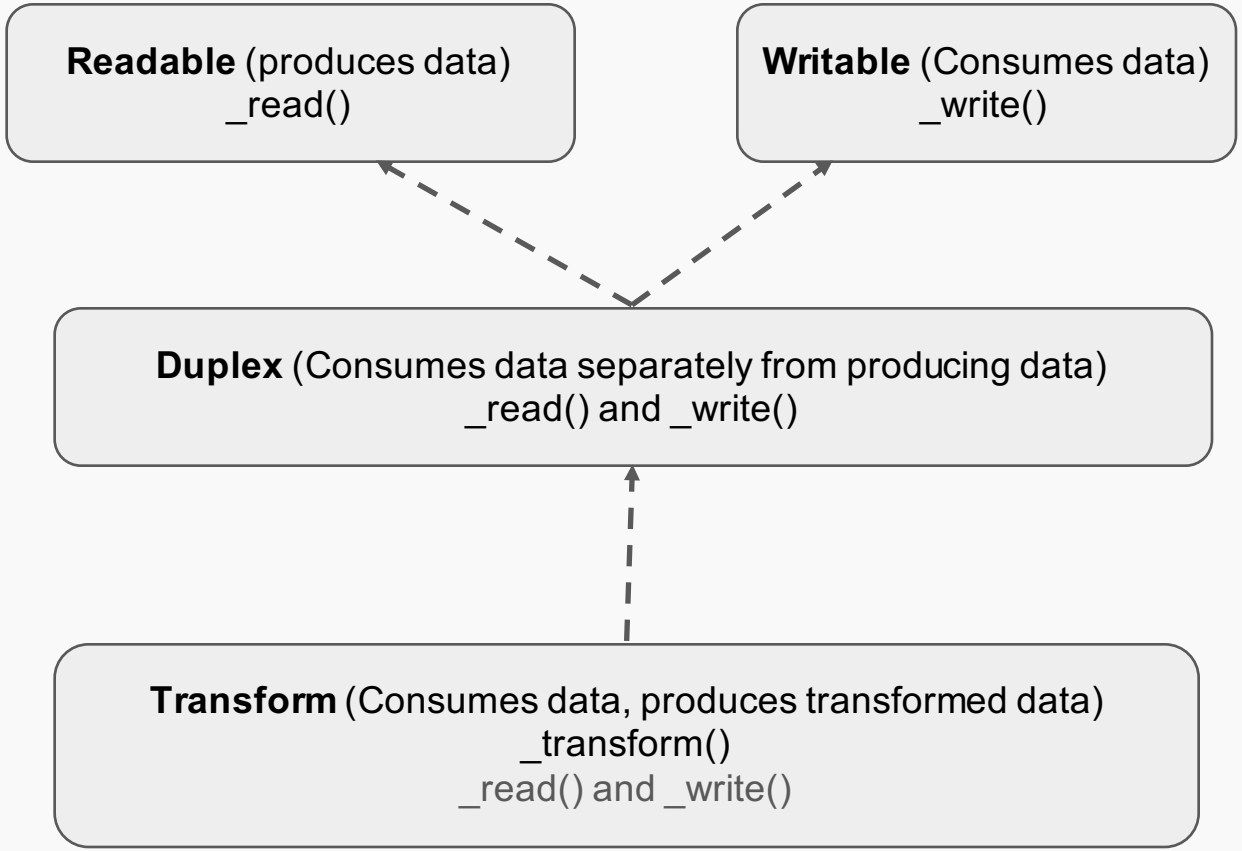
Writable Stream



Stream Performance Benefits

- **Memory Efficient:** Operates on data chunk-by-chunk without buffering everything in memory. Allows having hundreds of concurrent streams without memory exhaustion.
- **CPU Efficient:** Less memory, hence shorter GC cycles as less computing to determine what to clean up.

Types of Streams



Node.js stream based APIs

Readable Streams:

- HTTP responses, on the client
- HTTP requests, on the server
- fs read streams
- child process stdout and stderr
- process.stdin

Writable Streams:

- HTTP requests, on the client
- HTTP responses, on the server
- fs write streams
- child process stdin
- process.stdout, process.stderr

Duplex Streams:

- TCP sockets

Transform Streams:

- zlib streams
- crypto streams

Readable Stream API

Readable stream API: Events and Methods

Events

- 'close', 'end', 'error'
- 'data'
- 'pause', 'resume'
- 'readable'

Methods

- destroy()
- isPaused(), pause(), resume()
- pipe(), unpipe()
- read(), setEncoding()
- unshift(), wrap()
- readable[Symbol.asyncIterator]()
- stream.Readable.from(iterable)

Readable stream API: Events and Methods

Events

- 'close', 'end', 'error'
- 'data'
- 'pause', 'resume'
- 'readable'

Methods

- destroy()
- isPaused(), pause(), resume()
- pipe(), unpipe()
- read(), setEncoding()
- unshift(), wrap()
- **readable[Symbol.asyncIterator]()**
- **stream.Readable.from(iterable)**

Consuming data from a Readable Stream

Readable stream API: Events and Methods

Events

- 'close', 'end', 'error'
- 'data'
- 'pause', 'resume'
- 'readable'

Methods

- destroy()
- isPaused(), pause(), resume()
- pipe(), unpipe()
- read(), setEncoding()
- unshift(), wrap()
- **readable[Symbol.asyncIterator]()**
- stream.Readable.from(iterable)

Iterable and Iterator Protocols

```
// Traversable data structure
interface Iterable {
    [Symbol.iterator]() : Iterator;
}
// Traverses items inside an Iterable
interface Iterator {
    next() : IteratorResult;
}
// An object returned on each iteration of iterator
interface IteratorResult {
    value: any;
    done: boolean;
}

```

Iterable and Iterator Protocols

```
1  const numbers = ['1', '2', '3'];
2
3  // Option 1: Accessing contents using iterator
4  const iterator = numbers[Symbol.iterator]();
5  iterator.next(); // {value: 1, done: false}
6  iterator.next(); // {value: 2, done: false}
7  iterator.next(); // {value: 3, done: false}
8  iterator.next(); // {value: undefined, done: true}
9
10 // Option 2: Accessing contents using for-of loop
11 for (const number of numbers) {
12     number; // 1 2 3
13 }
14
```

Node.js Buffer and JavaScript String, Array, TypedArray, Map and Set are all built-in iterables, because each of their prototype objects have Symbol.iterator method.

Generators

```
1  function* createNumbers() {
2      yield 1;
3      yield 2;
4      yield 3;
5  }
6
7  const iter = createNumbers();
8  iter.next(); // Result: Object {value: 1, done: false}
9  iter.next(); // Result: Object {value: 2, done: false}
10 iter.next(); // Result: Object {value: 3, done: false}
11 iter.next(); // Result: Object {value: undefined, done: true}
12
13
```

Generator returns an iterator instance (which is also an iterable for convenience) that allows fetching data lazily.

Async Iterable and Async Iterator Protocols

```
interface Iterable {
    [Symbol.iterator]() : Iterator;
}

interface Iterator {
    next() : IteratorResult;
}

interface IteratorResult {
    value: any;
    done: boolean;
}
```

```
interface AsyncIterable {
 [Symbol.asyncIterator]() : AsyncIterator;
}

interface AsyncIterator {
    next() : Promise<IteratorResult>;
}

interface IteratorResult {
    value: any;
    done: boolean;
}
```

Async Generators

```
1  async function* createNumbers() {
2      yield 1;
3      yield 2;
4      yield 3;
5  }
6
7  const iter = createNumbers();
8  iter.next(); // Promise <pending> resolves with {value: 1, done: false}
9  iter.next(); // Promise <pending> resolves with {value: 2, done: false}
10 iter.next(); // Promise <pending> resolves with {value: 3, done: false}
11 iter.next(); // Promise <pending> resolves with {value: undefined, done: true}
12
13
```

Async Generator returns an **Async Iterator** instance

Async Iterator and for-await..of

```
async function*  createNumbers()  {
   yield '1';
   yield '2';
   yield '3';
}
const asyncIterable  = createNumbers();

// In each iteration, Async Iterator waits to resolve promise before continuing
for await (const number of asyncIterable)  {
    number // 1, 2, 3
}
```

Consuming Readable Steam using Async Iterator

```
1  const fs = require('fs');
2
3  async function run() {
4    const stream = fs.createReadStream('file.txt', {encoding: 'utf-8'});
5    for await (let chunk of stream) {
6      console.log(chunk);
7    }
8  }
9  run().catch(console.error);
10
11
12
13
14
```

As stream instance implements async Iterable protocol, JavaScript **for await..of** loop can consume stream in chunk sizes of highWatermark

Consuming Readable Steam using Async Iterator

```
1  const fs = require('fs');
2
3  async function run() {
4    const stream = fs.createReadStream('file.txt', {encoding: 'utf-8'});
5    for await (let chunk of stream) {
6      throw Error('Boo!!');
7      console.log(chunk);
8    }
9  }
10 run().catch(console.error);
11
12
13
14
```

No memory leaks in case of error!
Automatically closes the stream and cleans up underlying resources on throw or break..

Creating Readable Stream from an iterable

Readable stream API: Events and Methods

Events

- 'close', 'end', 'error'
- 'data'
- 'pause', 'resume'
- 'readable'

Methods

- destroy()
- isPaused(), pause(), resume()
- pipe(), unpipe()
- read(), setEncoding()
- unshift(), wrap()
- readable[Symbol.asyncIterator]()
- **stream.Readable.from(iterable)**

Creating a Readable Stream from an iterable

```
1  const util = require('util');
2  const stream = require('stream');
3  const pipeline = util.promisify(stream.pipeline);
4  const Readable = stream.Readable;
5
6  const numbers = ['1', '2', '3'];
7  function run() {
8     const stream = Readable.from(numbers);
9     pipeline(stream, process.stdout);
10 }
11 run();
12
13
14
```

Creating a Readable stream from JavaScript built-in iterables

Creating a Readable Stream from an iterable

```
1  const util = require('util');
2  const stream = require('stream');
3  const pipeline = util.promisify(stream.pipeline);
4  const Readable = stream.Readable;
5
6  const numbers = ['1', '2', '3'];
7  function run() {
8     const stream = Readable.from(numbers);
9     pipeline(stream, process.stdout); //1 2 3
10 }
11 run();
12
13
14
```

Backpressure kicks in if destination stream is slower than the source

Cleans up stream and underlying resources if errors

Creating a Readable Stream using a Generator

```
1  const pipeline = util.promisify(stream.pipeline);
2
3  function* createNumbers() {
4     yield '1';
5     yield '2';
6     yield '3';
7  }
8
9  function run() {
10    const stream = Readable.from(createNumbers());
11    pipeline(stream, process.stdout); //1 2 3
12 }
13 run();
14
```

Creating a Readable stream using an **iterable** returned from a **generator()**

Creating a Readable Stream using a Generator

```
1  const pipeline = util.promisify(stream.pipeline);
2
3  function* createNumbers() {
4     yield '1';
5     yield '2';
6     yield '3';
7  }
8
9  function run() {
10    const stream = Readable.from(createNumbers());
11    pipeline(stream, process.stdout); //1 2 3
12 }
13 run();
14
```

pipeline() method iterates to consume values based on the backpressure from downstream. Each iteration results in executing the generator function till next 'yield'!

Creating a Readable Stream using a Generator

```
1  const pipeline = util.promisify(stream.pipeline);
2
3  function* createNumbers() {
4     yield '1';
5     yield '2';
6     yield '3';
7  }
8
9  function run() {
10    const stream = Readable.from(createNumbers());
11    pipeline(stream, process.stdout); //1 2 3
12 }
13 run();
14
```

Creating Readable Stream with Async Generator

```
1  const pipeline = util.promisify(stream.pipeline);
2
3  async function* createNumbers() {
4      yield '1';
5      yield '2';
6      yield '3';
7  }
8
9  function run() {
10     const stream = Readable.from(createNumbers());
11     pipeline(stream, process.stdout); //1 2 3
12 }
13 run();
14
```

Readable stream .from() can also take an **async iterable** as returned by an **async generator**.

Creating Readable Stream with Async Generator

```
1  const pipeline = util.promisify(stream.pipeline);
2  async function* numbers () {
3     yield '1';
4     await new Promise(resolve => setTimeout(resolve, 1000));
5     yield '2';
6     await new Promise(resolve => setTimeout(resolve, 1000));
7     yield '3';
8  }
9  function run() {
10    const stream = Readable.from(createNumbers());
11    pipeline(stream, process.stdout); //1 2 3
12 }
13 run();
14
```

Best of everything!

await: synchronous style async workflow +

Generator: create data on demand +

Stream: Performance efficiency

Pitfall:

Memory leakage on error with .pipe()

pipe()

```
1  const gzip = require('zlib')
2  const fs = require('fs');
3
4  const input = fs.createReadStream('largefile.txt');
5  const output = fs.createWriteStream('largefile.txt.gz');
6
7  input
8      .pipe (gzip.createGzip())
9      .pipe(output);
10
11
12
13
14
```

Automatically manages backpressure so that the destination stream is not overwhelmed by a faster source stream.

pipe()

```
1  const gzip = require('zlib')
2  const fs = require('fs');
3
4  const input = fs.createReadStream('largefile.txt');
5  const output = fs.createWriteStream('largefile.txt.gz');
6
7  input
8      .pipe (gzip.createGzip())
9      .pipe(output);
10
11
12
13
14
```

Does not propagate errors and clean up automatically - could lead to memory leaks

pipeline()

```
1  const { pipeline } = require('stream');
2  const fs = require('fs');
3  const zlib = require('zlib');
4
5  pipeline(
6    fs.createReadStream('largefile.txt'),
7    zlib.createGzip(),
8    fs.createWriteStream('largefile.txt.gz'),
9    (err) => {
10     if (err) { console.error('Pipeline failed', err);}
11     else { console.log('Pipeline succeeded'); }
12   }
13 );
14
```

Automatically closes all piped streams on error

pipeline() - using promise

```
1  const util = require('util');
2  const pipeline = util.promisify(stream.pipeline);
3
4  async function run() {
5    try {
6      await pipeline(
7        fs.createReadStream('largefile.txt'),
8        zlib.createGzip(),
9        fs.createWriteStream('largefile.txt.gz'),
10     );
11     console.log('Pipeline succeeded');
12   } catch (err) {console.error('Pipeline failed', err);}
13 }
14 run();
```

Activity - Make the code performant using streams..

Activity - Make the code performant using stream..

```
1  const fs = require('fs');
2
3  async function print(path) {
4    let contents = await fs.promises.readFile(path, 'utf8');
5    contents = contents.toUpperCase();
6    console.log(contents);
7  }
8  print('./largefile.txt').catch(console.error);
9
10
11
12
13
14
```

Activity - Make the code performant using stream..

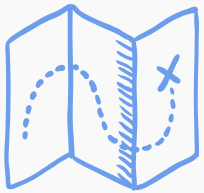
```
1  const fs = require('fs');
2
3  async function print(path) {
4    let contents = await fs.promises.readFile(path, 'utf8');
5    contents = contents.toUpperCase();
6    console.log(contents);
7  }
8  print('./largefile.txt').catch(console.error);
9
10
11
12
13
14
```

Hint: Use Readable FIle stream and use async iterator to extract contents from it

Solution - Make it more performant using streams..

```
1  const fs = require('fs');
2
3  async function print(path) {
4    const stream = fs.createReadStream(path, {encoding: 'utf-8'});
5    for await (let chunk of stream) {
6      chunk = chunk.toUpperCase();
7      console.log(chunk);
8    }
9  }
10 print('./largefile.txt').catch(console.error);
11
12
13
14
```

Roadmap



Async Programming Essentials

- Async Programming Features in JavaScript
- Node.js Event Loop Phases

Applied Async and Parallel Programming

- Handling I/O Operations
- Handling Memory Intensive Operations
- Handling CPU Heavy Operations

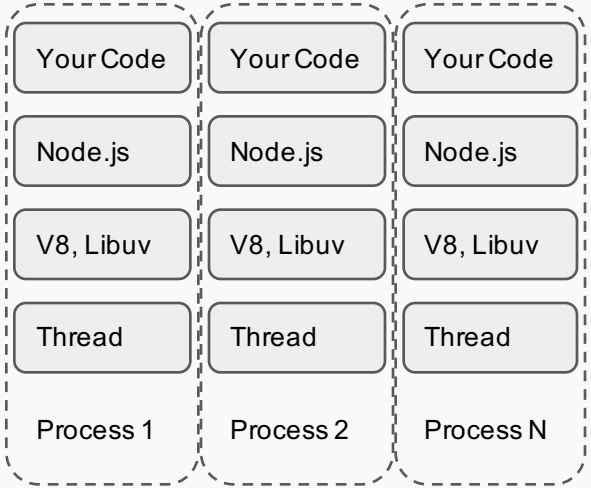
Concurrency is not enough...

when the work associated with each client at any given time is not "small".

Concurrency is not enough...

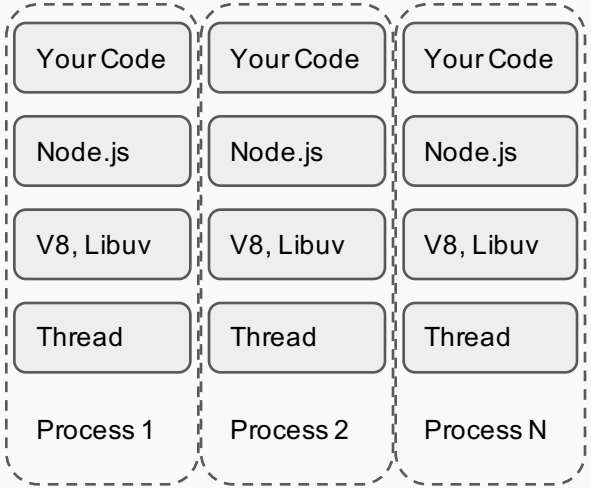
when fully utilizing modern CPU cores

Multiple Processes



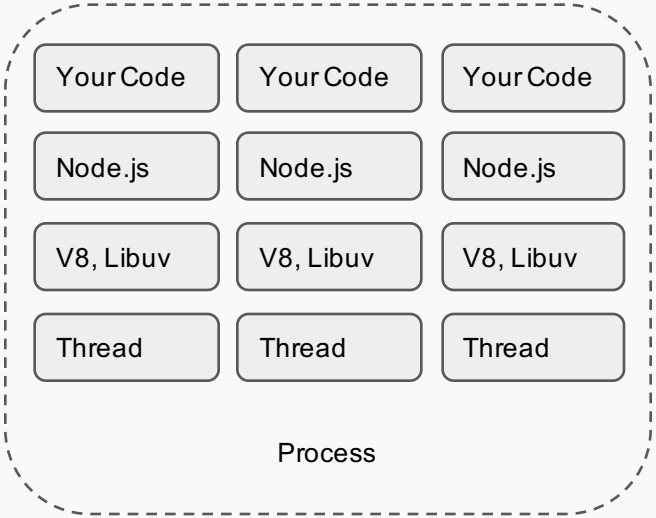
- Via child_process.fork(), Cluster, tiny-worker package
- Isolated memory space per process
- No shared memory, costly inter-process Communication

Multiple Processes



- Via child_process.fork(), Cluster, tiny-worker package
- Isolated memory space per process
- No shared memory, costly inter-process Communication

Worker Threads



- Via worker_thread API
- Lightweight option
- Threads can share memory by -
  - transferring ArrayBuffer instances
  - sharing SharedArrayBuffer instances

Simple Worker Example - parent.js

```
1  const { Worker } = require('worker_threads');
2  const worker = new Worker('./worker.js', { workerData: 'world' });
3
4  worker.on('message', (messageFromWorker) => {
5    console.log(messageFromWorker);
6  });
7  worker.on('error', (err) => {
8    console.log(err.message);
9  });
10 worker.on('exit', (code) => {
11   if (code !== 0)
12   console.log(`Exit code ${code}`);
13 });
```

Worker data sent as part of constructor options

Simple Worker Example - worker.js

```
1  const { workerData, parentPort } = require('worker_threads')
2  const result = {'hello': workerData };
3  parentPort.postMessage(result);
```

You can do any CPU intensive work here without blocking the "main thread"

Simple Worker Example - parent.js

```
1  const { Worker } = require('worker_threads');
2  const worker = new Worker('./worker.js', { workerData: 'world' });
3
4  worker.on('message', (messageFromWorker) => {
5    console.log(messageFromWorker); //-> { hello: 'world' }
6  });
7  worker.on('error', (err) => {
8    console.log(err.message);
9  });
10 worker.on('exit', (code) => {
11   if (code !== 0)
12   console.log(`Exit code ${code}`);
13 });
```

Simple Worker Example - worker.js

```
1  const { workerData, parentPort } = require('worker_threads')
2  throw new Error('Bang!');
3  parentPort.postMessage({ hello: workerData })
```

Simple Worker Example - parent.js

```
1  const { Worker } = require('worker_threads');
2  const worker = new Worker('./worker.js', { workerData: 'world' });
3
4  worker.on('message', (messageFromWorker) => {
5    console.log(messageFromWorker);
6  });
7  worker.on('error', (err) => {
8    console.log(err.message); //-> Bang!
9  });
10 worker.on('exit', (code) => {
11   if (code !== 0)
12   console.log(`Exit code ${code}`); //-> Exit code 1
13
   });
```

Simple Worker Example - using Promise

```
1  const { Worker } = require('worker_threads');
2
3  const promise = new Promise((resolve, reject) => {
4      let messageFromWorker;
5      const worker = new Worker('./worker.js', { workerData: 'world!'});
6      worker.on('message', message => {
7          messageFromWorker = message;
8      })
9      worker.on('exit', () => resolve(messageFromWorker));
10     worker.on('error', reject);
11 });
12
13 promise.then(result => console.log(result)).catch(err => console.error(err));
```

Communication over Default Message Channel

```
1  const { Worker, isMainThread, parentPort } = require('worker_threads');
2
3  if (isMainThread) {
4    const worker = new Worker(__filename);
5    worker.postMessage('world!');
6    worker.on('message', (message) => {
7      console.log(message);   // Prints 'Hello, world!'.
8    });
9  } else {
10   // When a message from the parent thread is received, send it back:
11   parentPort.on('message', (message) => {
12     parentPort.postMessage(`Hello, ${message}`);
13   });
   }
```

Communication over Default Message Channel

```
1  const { Worker, isMainThread, parentPort } = require('worker_threads');
2
3  if (isMainThread) {
4    const worker = new Worker(__filename);
5    worker.postMessage('world!');
6    worker.on('message', (message) => {
7      console.log(message);   // Prints 'Hello, world!'.
8    });
9  } else {
10   // When a message from the parent thread is received, send it back:
11   parentPort.on('message', (message) => {
12     parentPort.postMessage(`Hello, ${message}`);
13   });
   }
```

Communication over Custom Message Channel

```
1  const { Worker, MessageChannel} = require('worker_threads');
2  const worker = new Worker('./worker.js');
3
4  // Create custom message channel
5  const subChannel = new MessageChannel();
6
7  // Send one MessagePort to worker through a pre-existing channel
8  worker.postMessage(subChannel.port1, [subChannel.port1]);
9
10 // Start sending and listening from worker on second port
11 subChannel.port2.postMessage('world!')
12 subChannel.port2.on('message', (value) => {
13     console.log(value); //-> prints { hello: 'world!' }
14 });
```

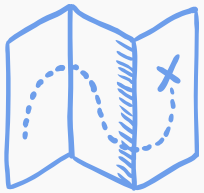
Communication over Custom Message Channel

```
1  // worker.js
2  const { parentPort } = require('worker_threads');
3  // Receive custom message port over the default channel
4  parentPort.once('message', (customPort) => {
5      // Then use custom port to communicate with parent
6      customPort.on('message', (message) => {
7          customPort.postMessage({'hello': message});
8      });
9  });
10
11
12
13
```

Using Worker Threads Optimally

- Don’t use worker threads as a mechanism to run sync I/O operations in parallel.
- Use worker pool instead of spawning a worker thread per request (e.g workerpool package)

Roadmap



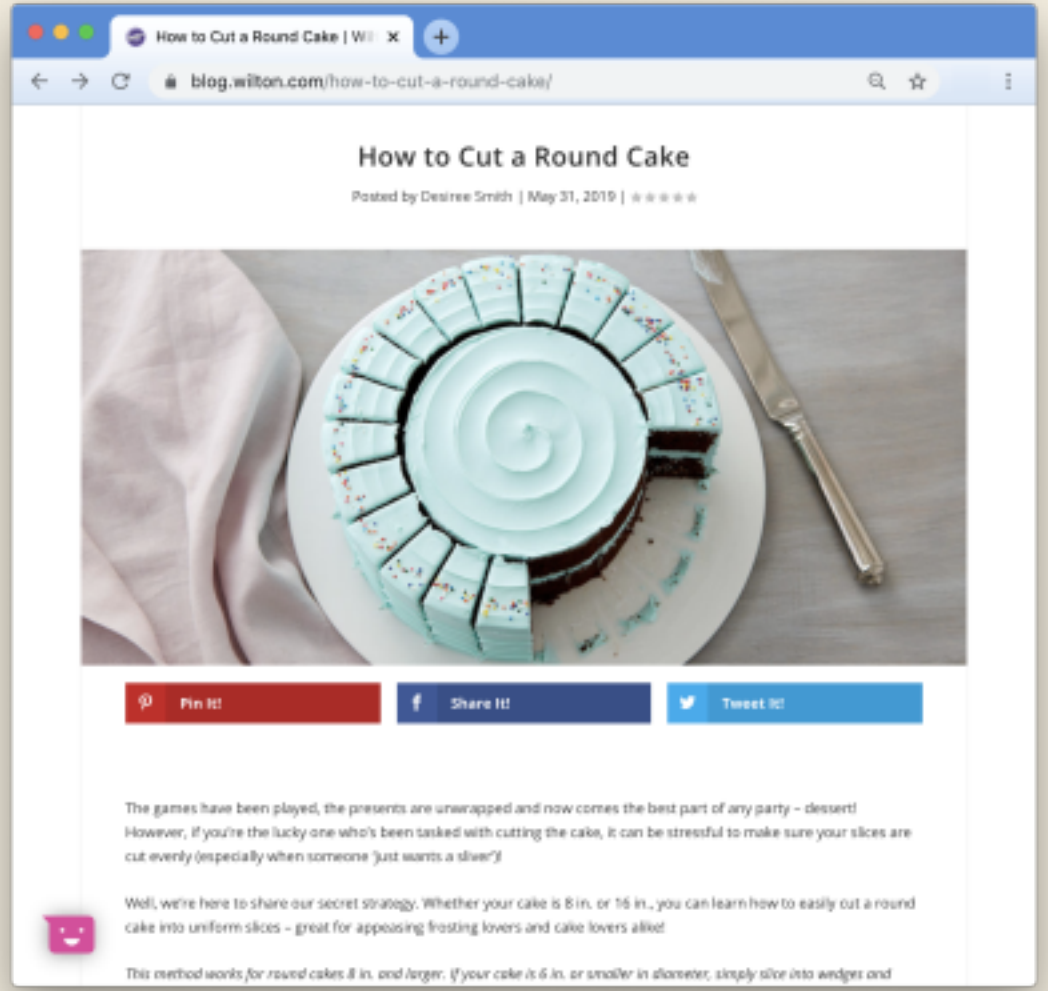
Async Programming Essentials

- Async Programming Features in JavaScript
- Node.js Event Loop Phases

Applied Async and Parallel Programming

- Handling I/O Operations
- Handling Memory Intensive Operations
- Handling CPU Heavy Operations

Wrap up



https://blog.wilton.com/how-to-cut-a-round-cake/

Tools and Techniques for Slicing Your Application Logic

Node.js Runtime:

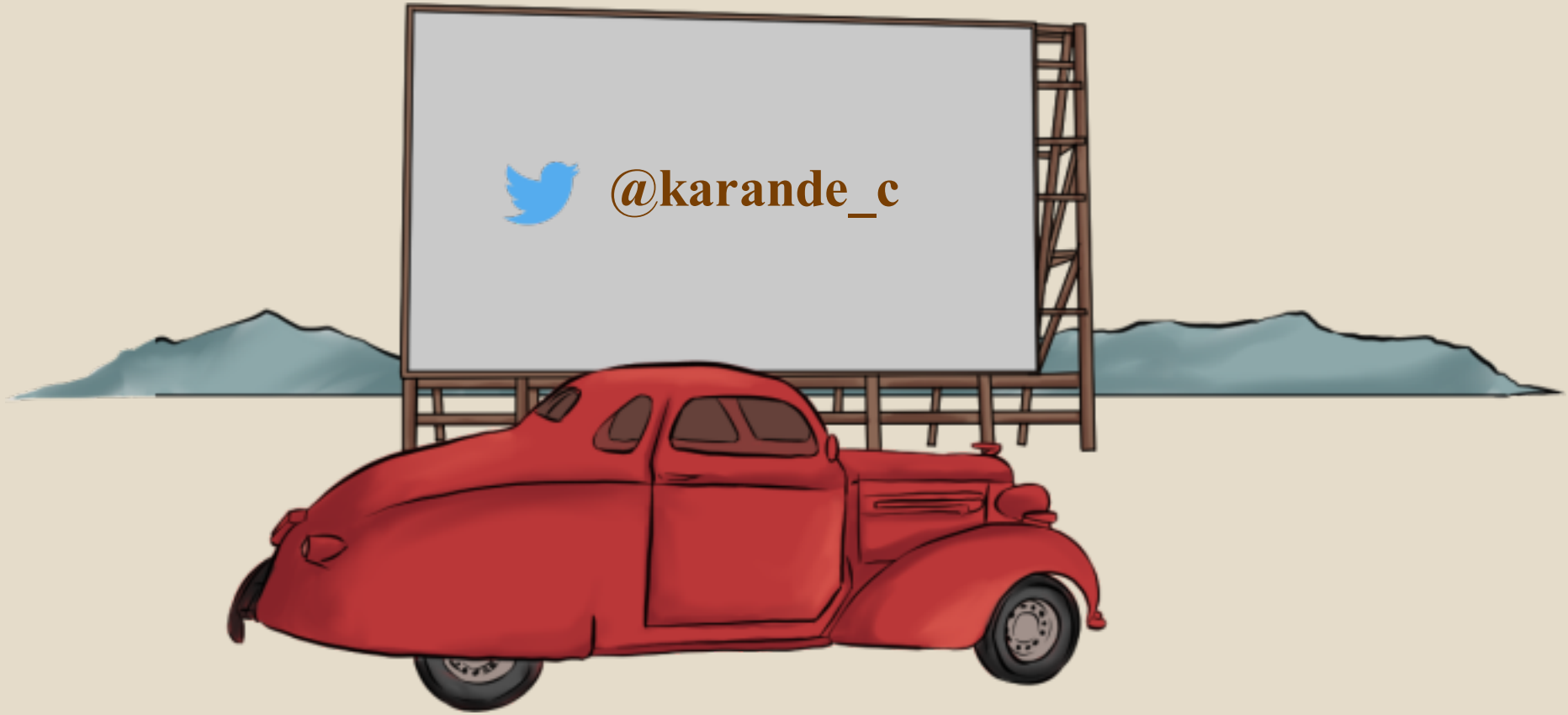
- Event Loop, OS Level Async Primitives, Libuv thread Pool

JavaScript Language Features:

- Promises, Generator, Async Functions, Await keyword, Async Generator, Async Iterator

Node.js APIs:

- Stream API, Worker Threads


@karande_c