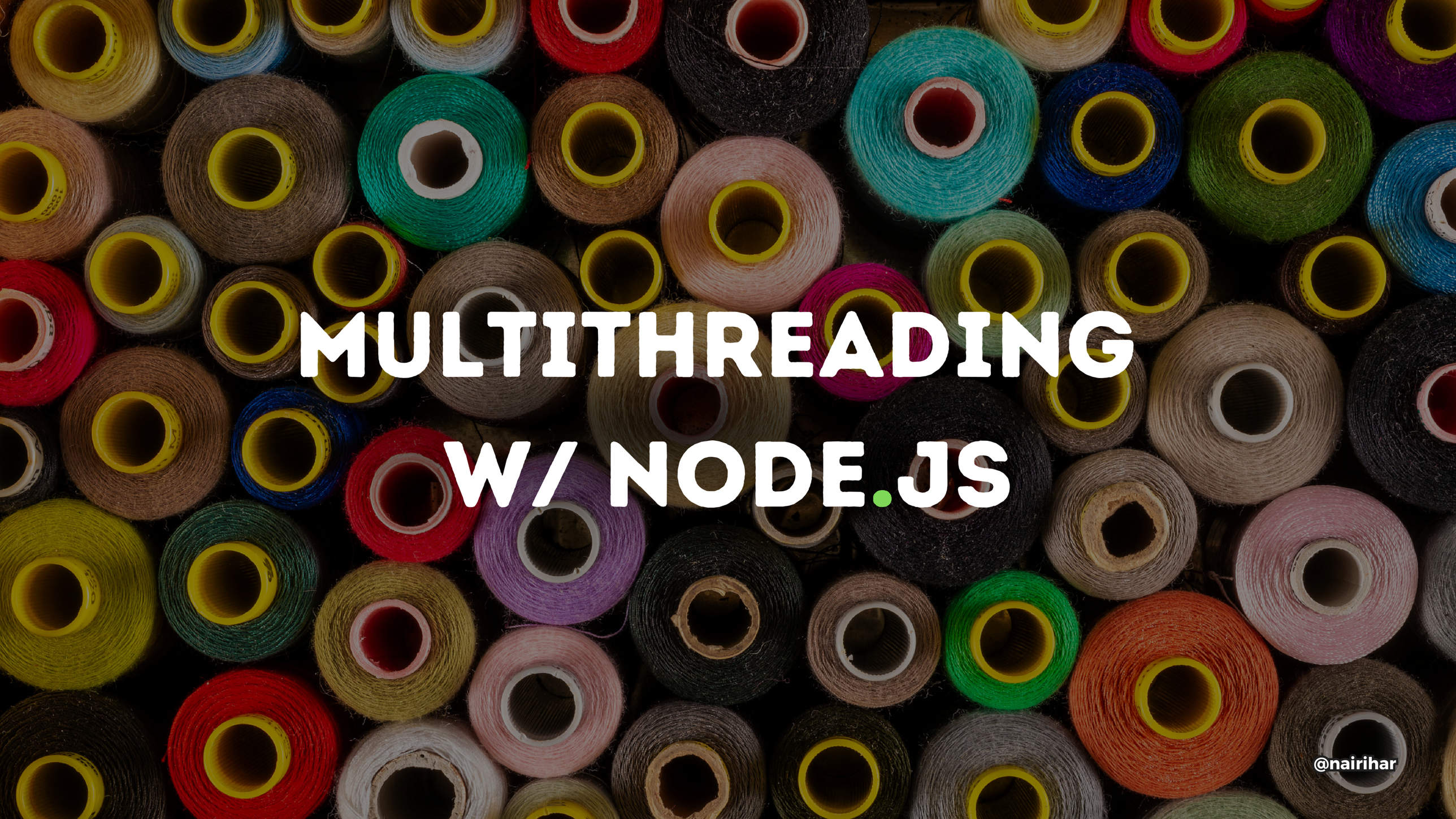

@nairihar


MULTITHREADING
W/ NODE.JS
@nairihar

Before a brief introduction
about myself ...

HEY I'M NAIRI *_*

- 🥷 Backend Engineer at Screenful
- 💛 Creator of the JavaScript Armenia Community
- ✍️ Writing about JavaScript and related technologies
- 🎙️ Hosting podcasts for the Armenian tech community
- 🎶 Speaker, Conference/Meetup Organizer

@nairihar

Do you already know
about worker_threads?

AGENDA

- Node.js & Libuv (not so single-threaded)
- worker_threads (basics + demo)
- SharedArrayBuffer (shared data)
- Atomics (race conditions)
- Abstractions for Node.js Multithreading

JavaScript

Simpler than you think...

We use it everywhere...

IOT

DESKTOP

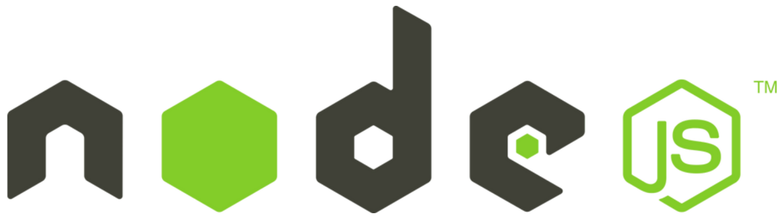
WEB

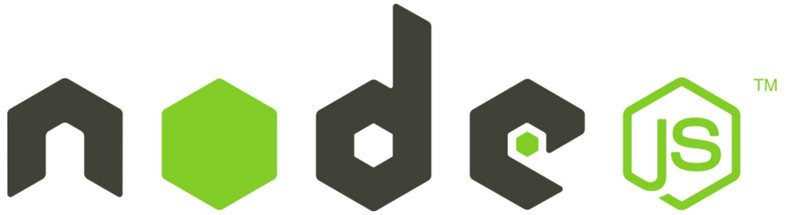
BACKEND

GAME

MOBILE

...


node
JS
TM


node
JS
TM


libuv

I/O in Operation systems

I/O in Operation systems

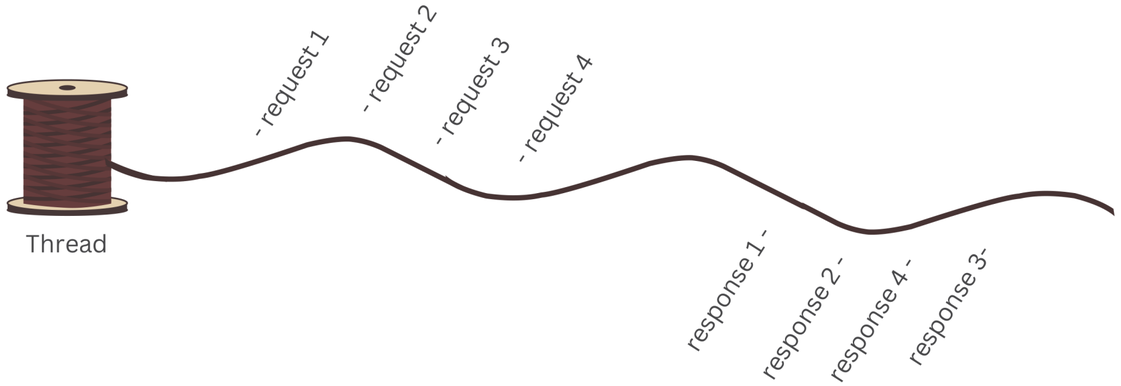
Non blocking I/O

- *Network*

I/O in Operation systems

Non blocking I/O

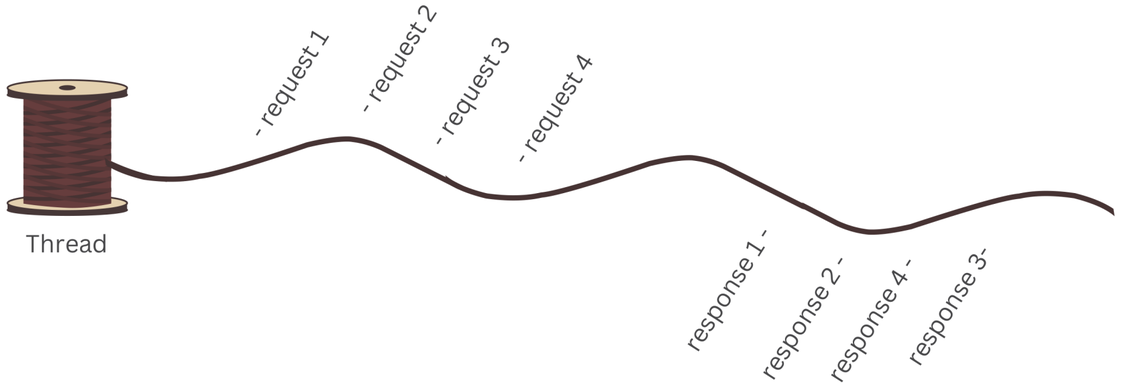
- *Network*



I/O in Operation systems

Non blocking I/O

- *Network*



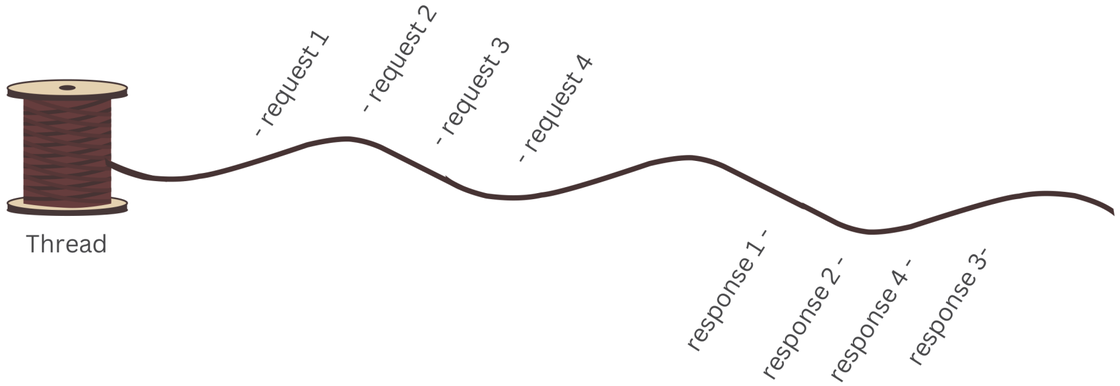
Blocking I/O

- *File system*
- *DNS.lookup*

I/O in Operation systems

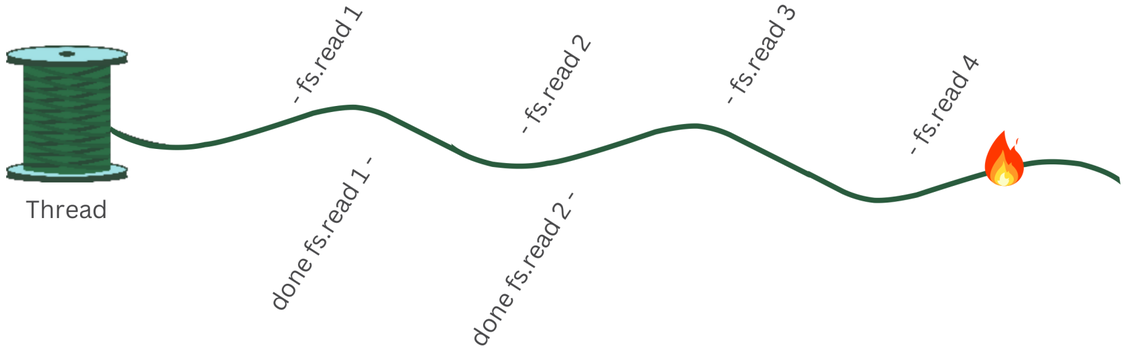
Non blocking I/O

- *Network*



Blocking I/O

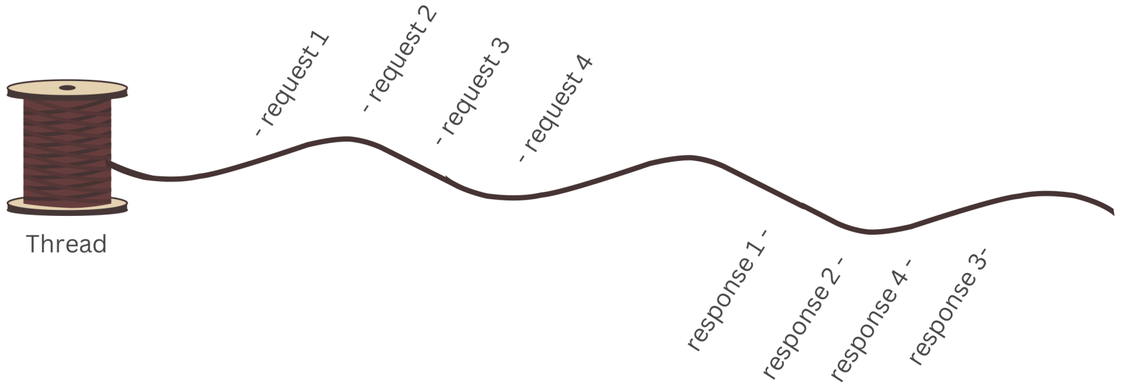
- *File system*
- *DNS.lookup*



I/O in Operation systems

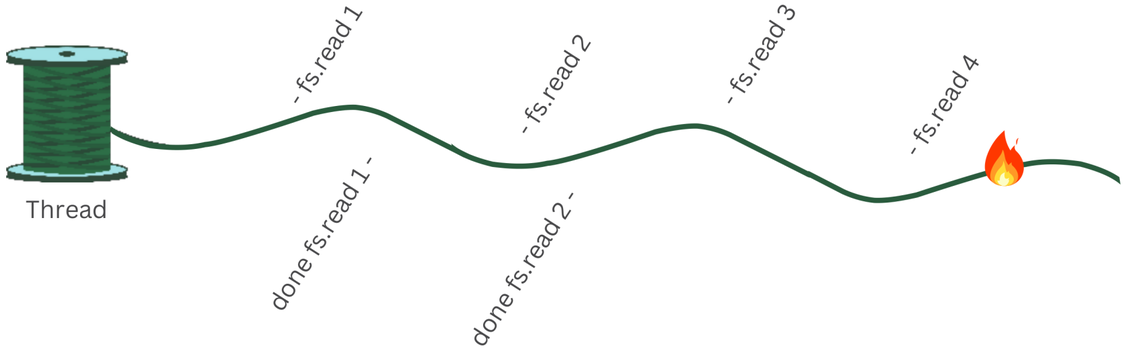
Non blocking I/O

- *Network*



Blocking I/O

- *File system*
- *DNS.lookup*



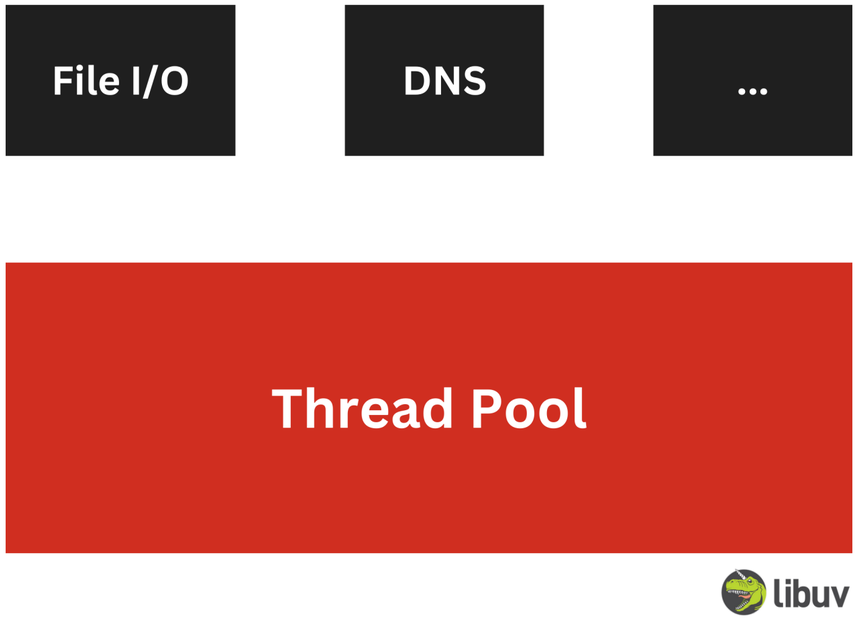
*__io_uring__ is a Linux kernel system call interface for storage device **asynchronous I/O** operations.

Node. js is a single-threaded runtime

```
const fs = require('fs').promises;

(async () => {
    const [ user, balance ] = await Promise.all([
        fs.readFile('user.json'),
        fs.readFile('balance.json'),
    ]);

    // ...
})()
```

How?

Blocking I/O & CPU intensive



Thread Pool

Thread Pool

UV_THREADPOOL_SIZE=10 (Up to 1024 threads)

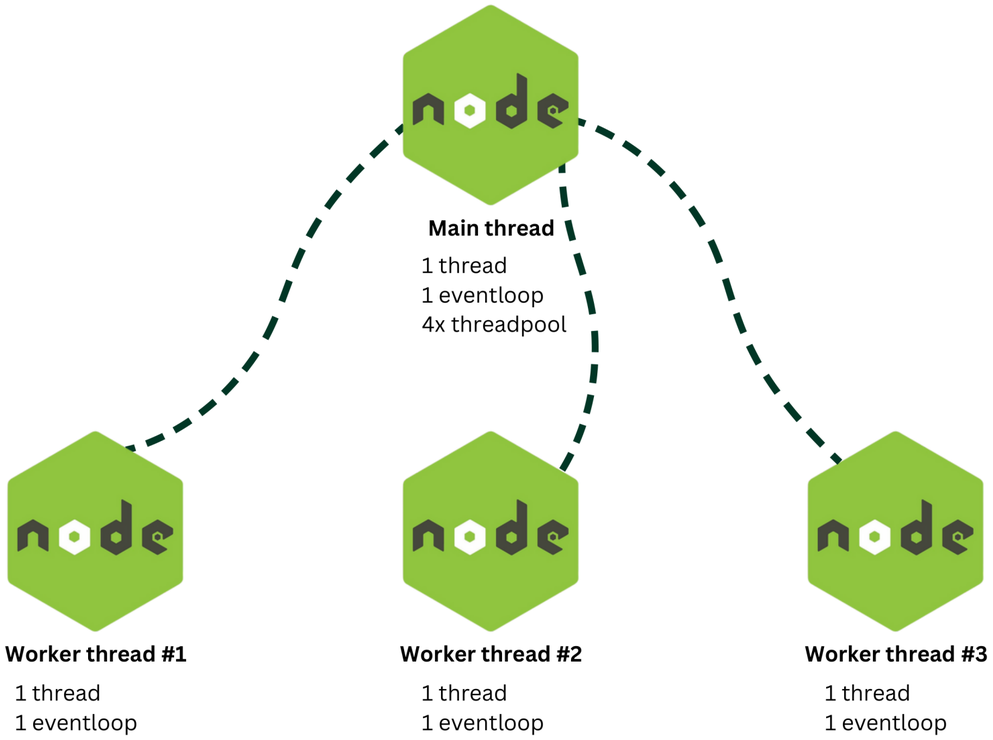
Before node v12.5.0: Up to 128 threads (before libuv v*1.30.0*)

We can’t use **ThreadPool threads** directly…

```
const { Worker } = require('worker_threads');
```

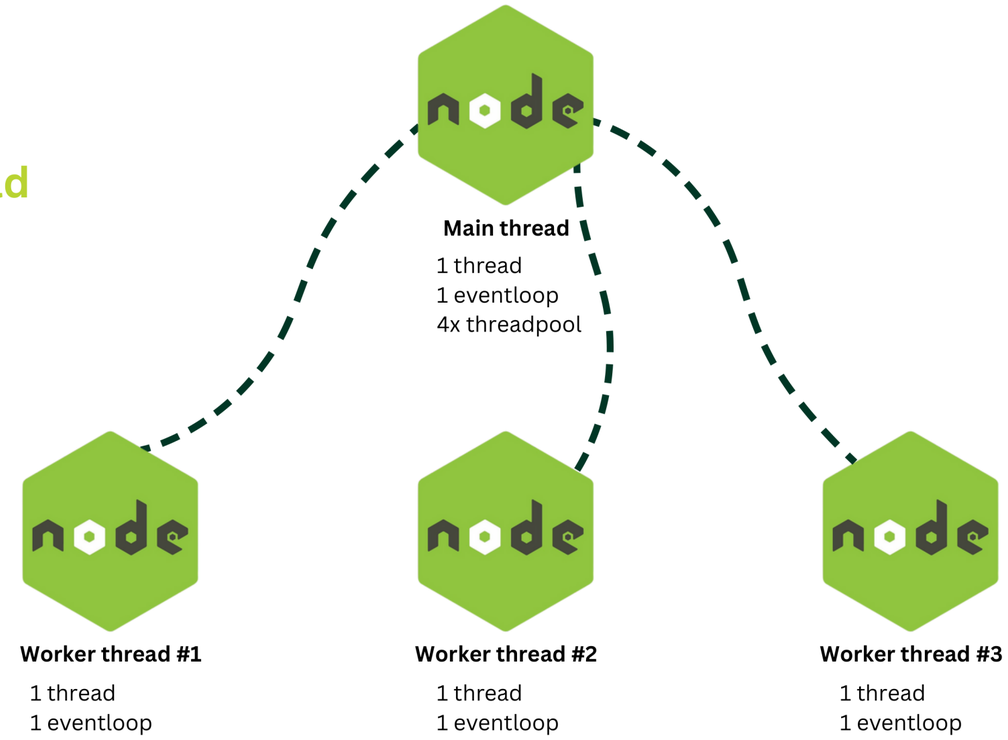
Unlike **ThreadPool** threads...

'worker_threads'



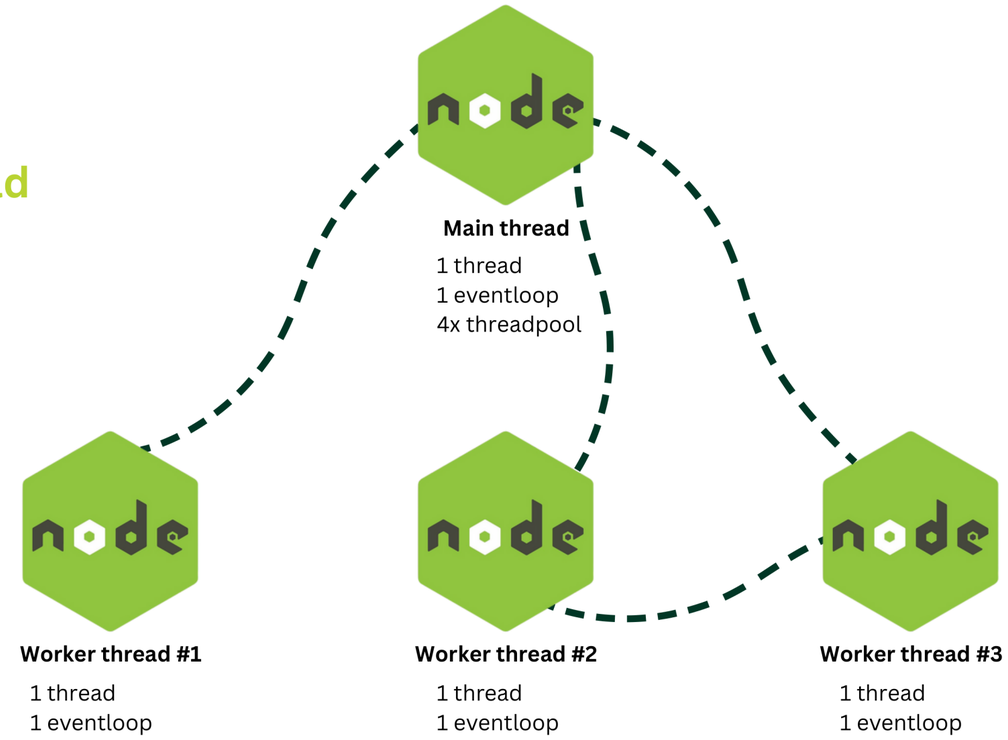
'worker_threads'

Main thread <-> Thread
communication



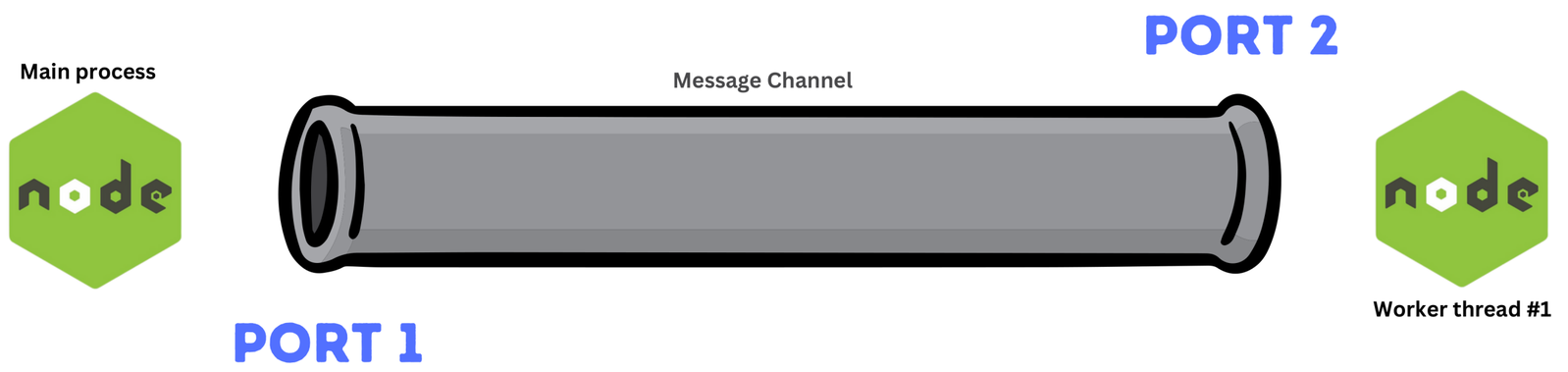
'worker_threads'

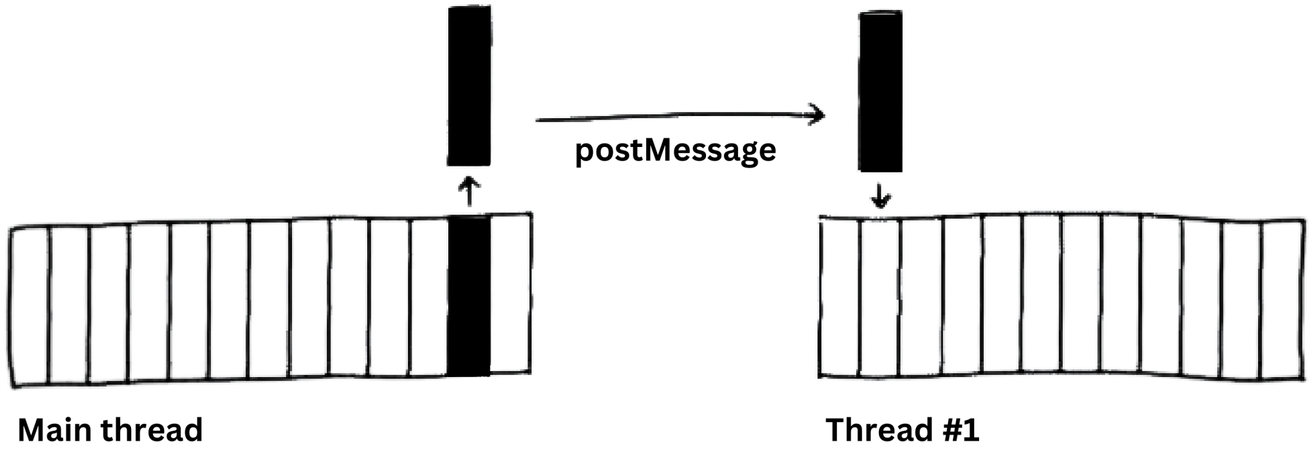
Main thread <-> Thread
communication



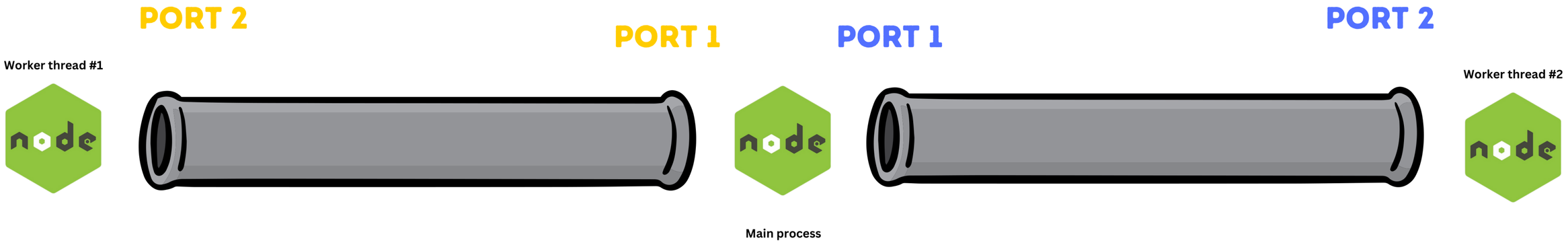
Thread <-> Thread
communication

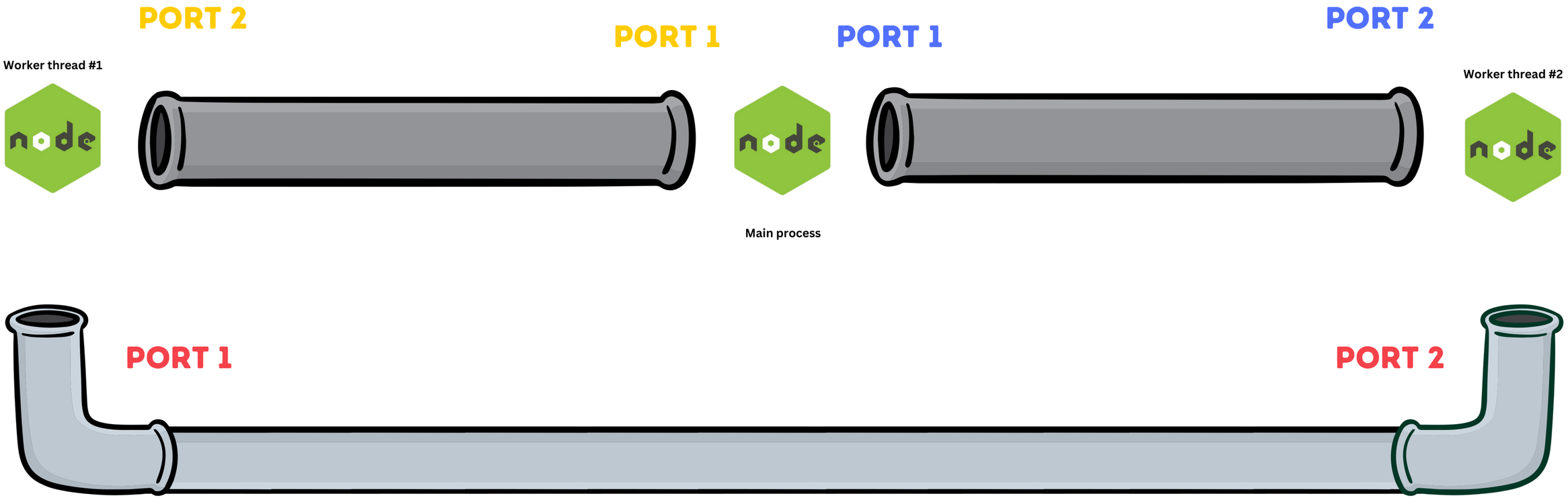
DEMO


PORT 2
Main process
Message Channel
node
node
Worker thread #1
PORT 1



Message Channels


PORT 2
PORT 1
PORT 1
PORT 2
Worker thread #1
Worker thread #2
Main process



*A **new Message Channel** created by the main process*

Abstractions

THREADS.JS

Basics

```
// master.js
import { spawn, Thread, Worker } from "threads"

const auth = await spawn(new Worker("./workers/auth"))
const hashed = await auth.hashPassword("Super secret password", "1234")

console.log("Hashed password:", hashed)

await Thread.terminate(auth)
```

```
// workers/auth.js
import sha256 from "js-sha256"
import { expose } from "threads/worker"

expose({
  hashPassword(password, salt) {
    return sha256(password + salt)
  }
})
```

dependencies none | downloads 8k | license MIT

My experimental open-source project. #1

funthreads

A simple library that provides an abstraction for the Node.js `worker_threads` module. It enables you to run your function in a separate thread. You receive a Promise that resolves with the result of your function.

Example

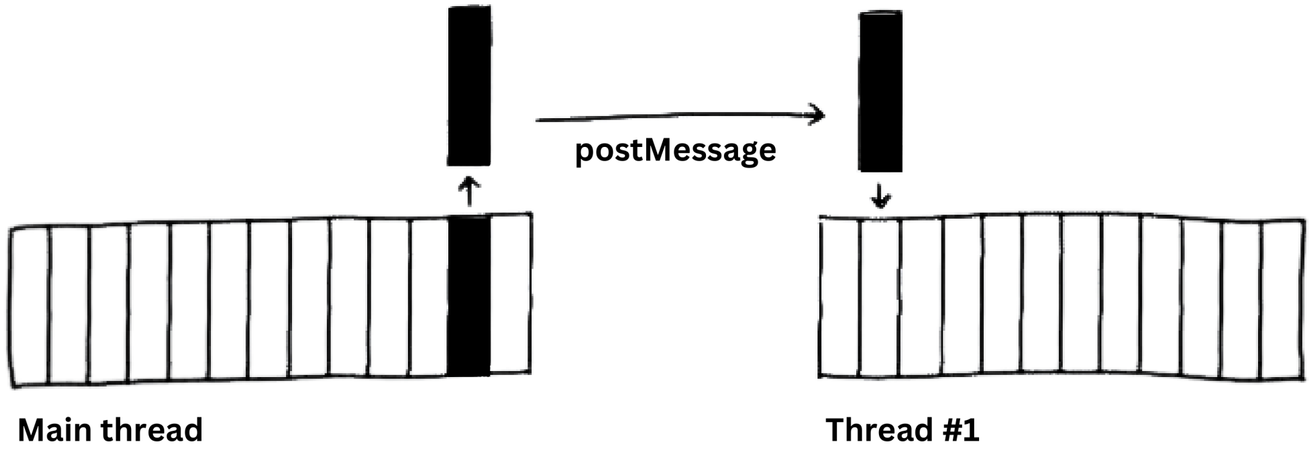
```
const { executeInThread } = require('funthreads');

// heavy operation (this will not block the main thread)
const num = await executeInThread((limit) => {
  let result = 0, i = 1;

  while (i <= limit) {
    result += i.toString().split('').reverse().join('').length;
    i++;
  }

  return result;
}, 12345678);
```

This example highlights the optimization of a resource-intensive calculation. By executing the function in a separate thread, we prevent the main thread from being blocked.

Surprisingly simple, isn't it?

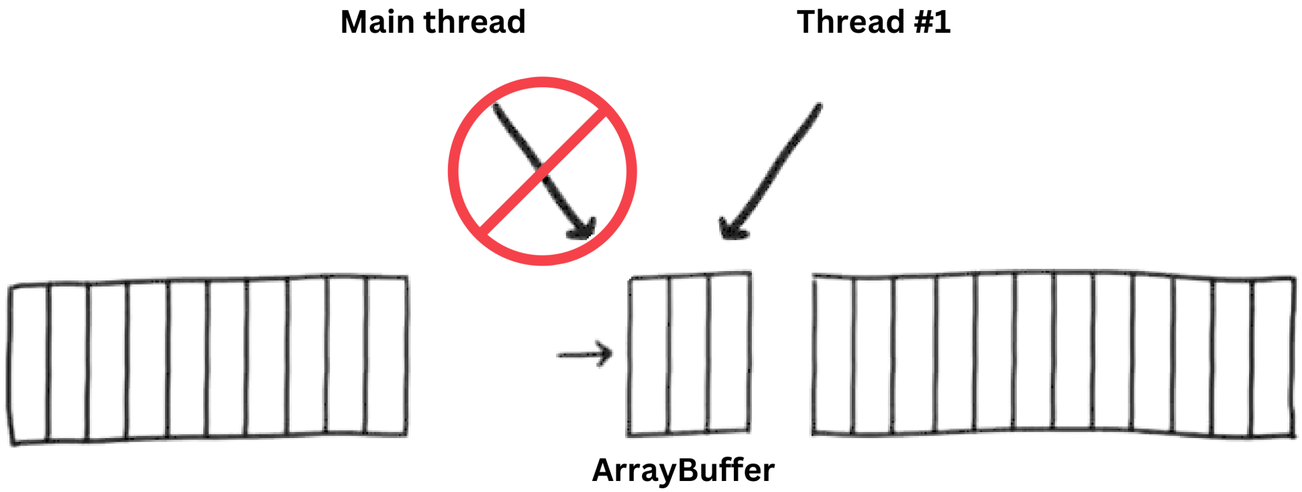
@nairihar

How to share a state/data?



ArrayBuffer

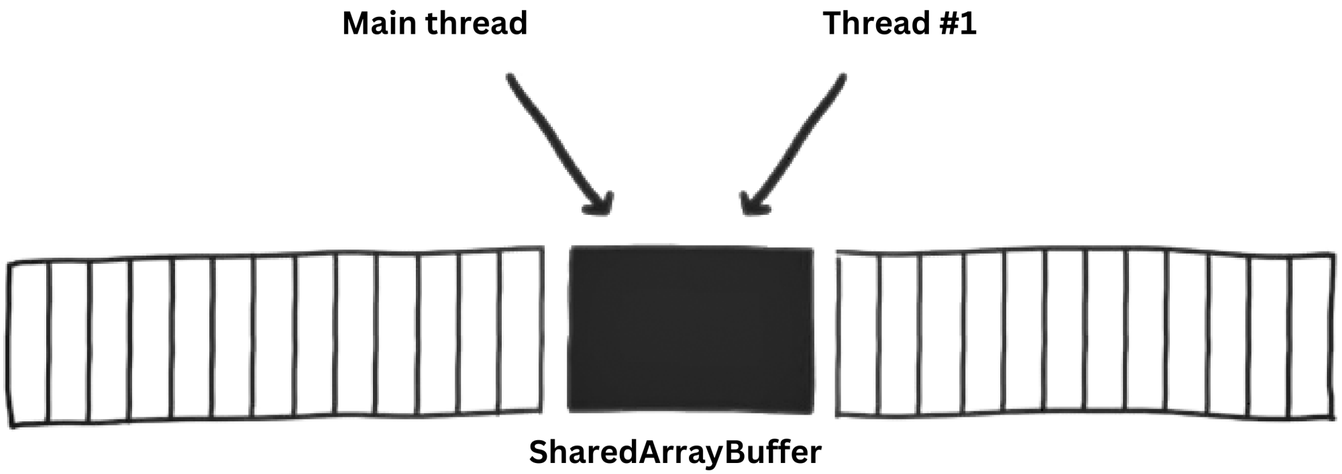
This object is used to represent a generic raw binary data buffer.



SharedArrayBuffer

This object allows thread communication by sharing a common memory space.

It is not a Transferable Object, unlike an ArrayBuffer which is transferable.



01001011101000011 0...

Typed Arrays(Buffer views)

```
Int8Array

Uint8Array

Uint8ClampedArray

Int16Array

Uint16Array

Int32Array

Uint32Array

Float32Array

Float64Array

BigInt64Array

BigUint64Array
```

Int8Array

The **Int8Array** typed array represents an array of **8-bit** signed integers.

It can store values ranging from **-128 to 127**, as it represents 8-bit signed integers.

Uint8Array

An array of **8-bit unsigned integers**, meaning it can store values from **0 to 255**.

Int16Array

Uint16Array

Int32Array

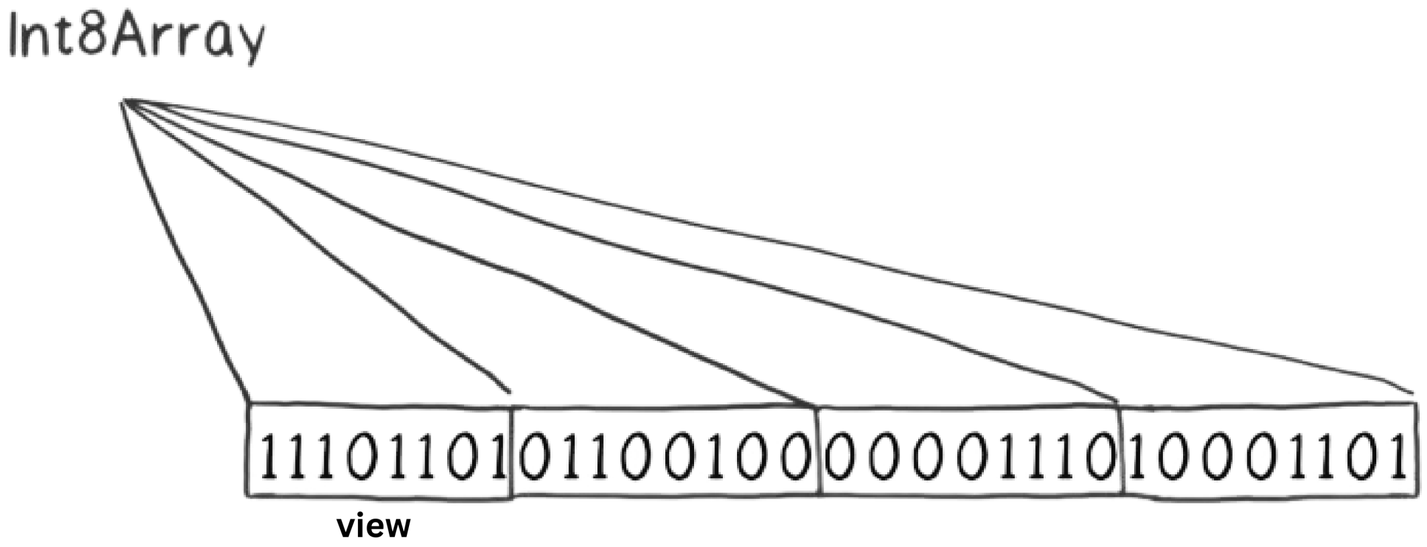
Uint32Array

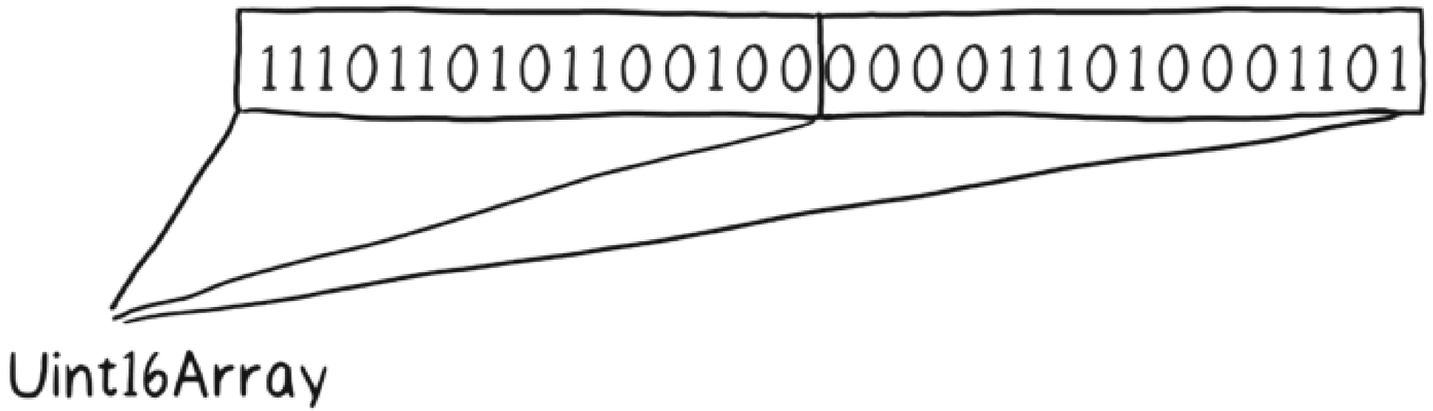
Float32Array

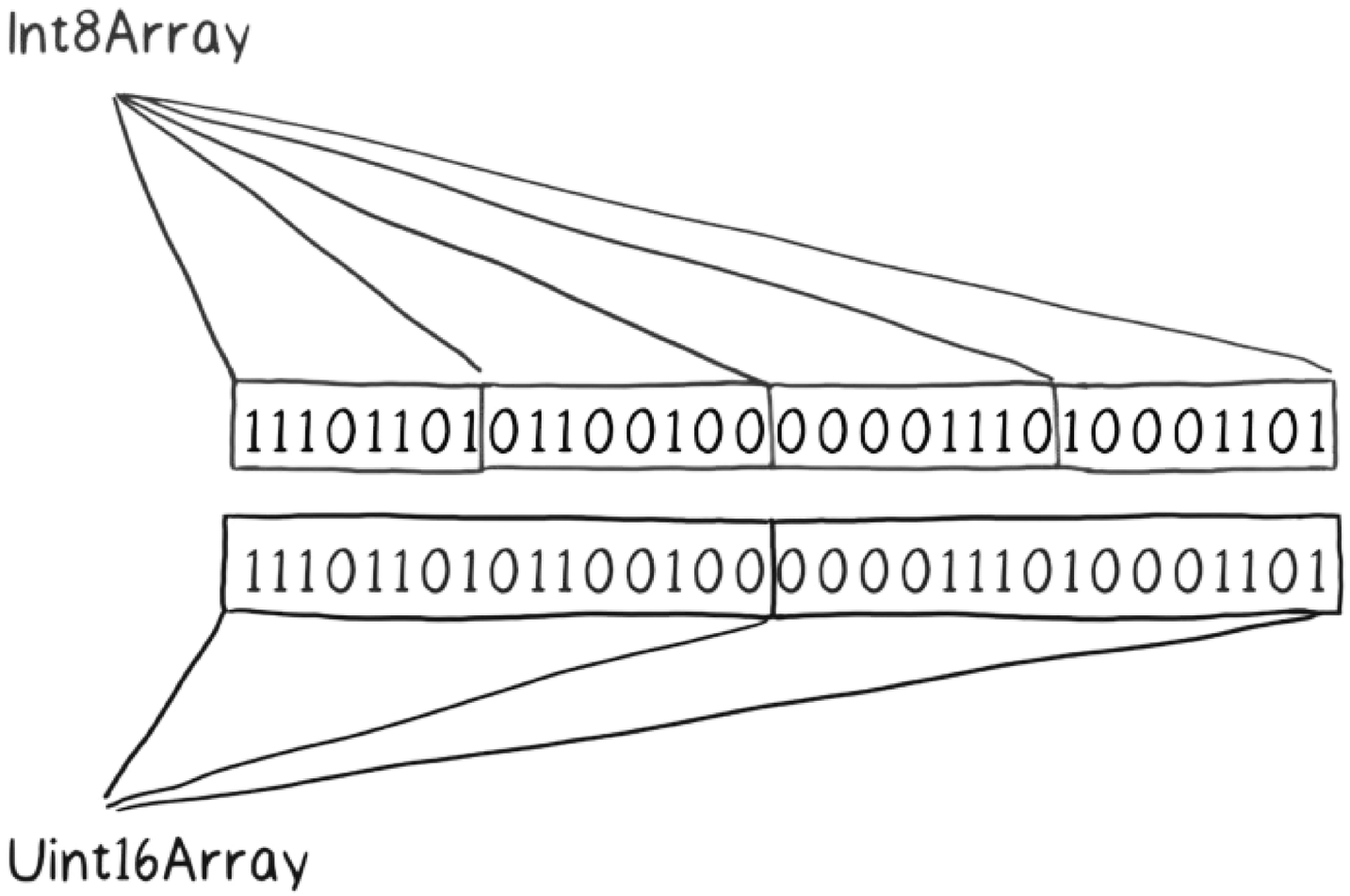
Float64Array

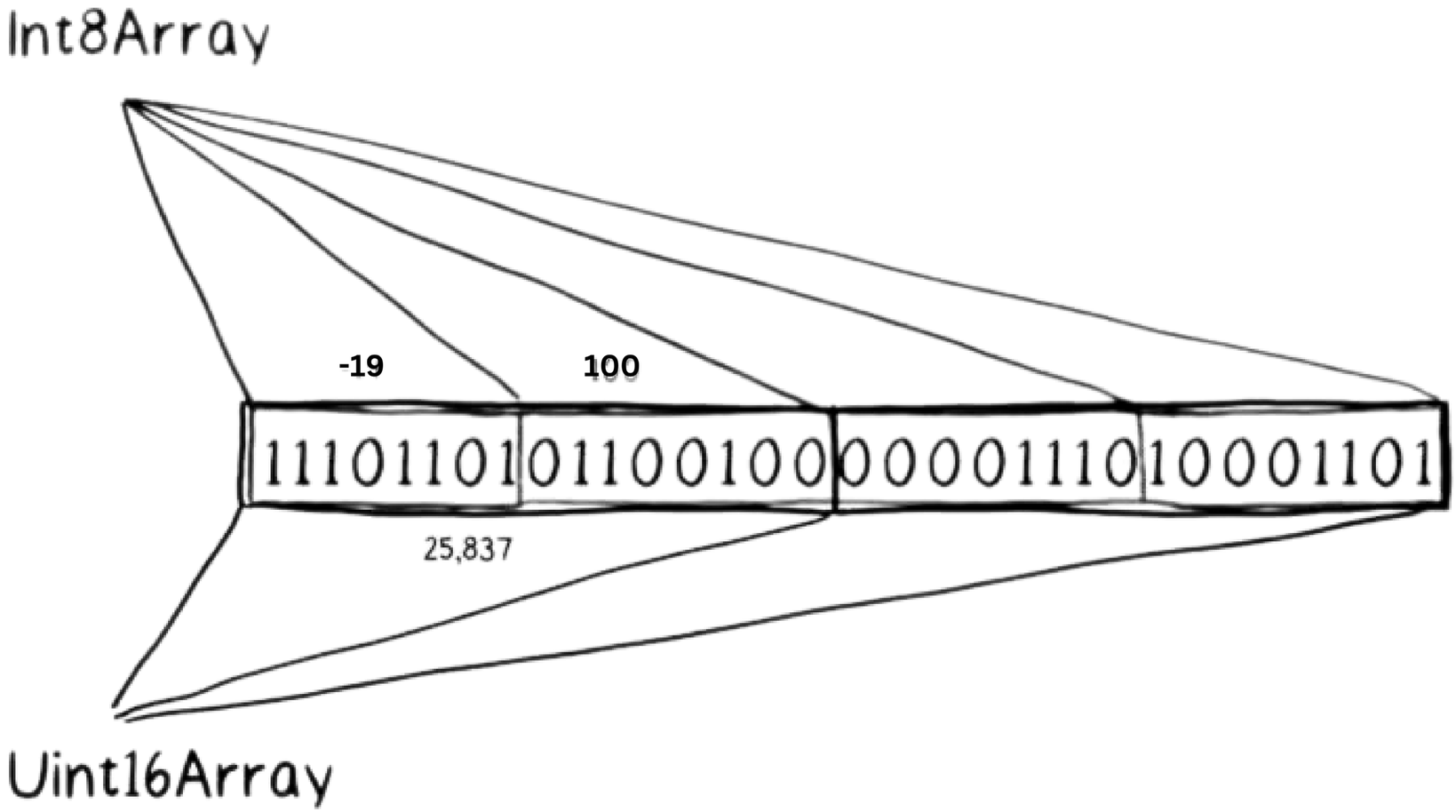
BigInt64Array

BigUint64Array






Int8Array
11101101
01100100
00001110
10001101
1110110101100100
0000111010001101
Uint16Array



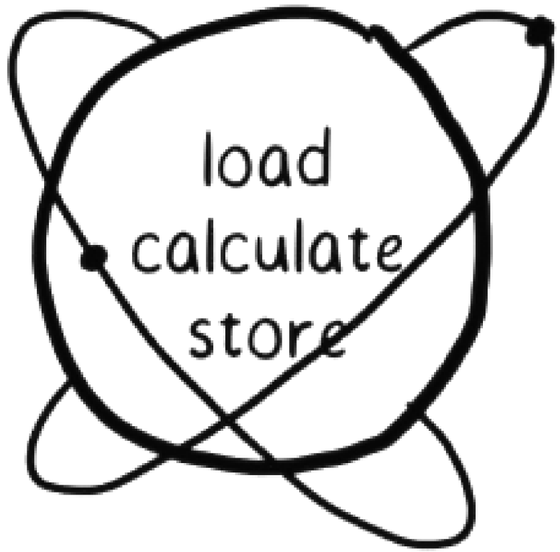
How to keep data using numbers?

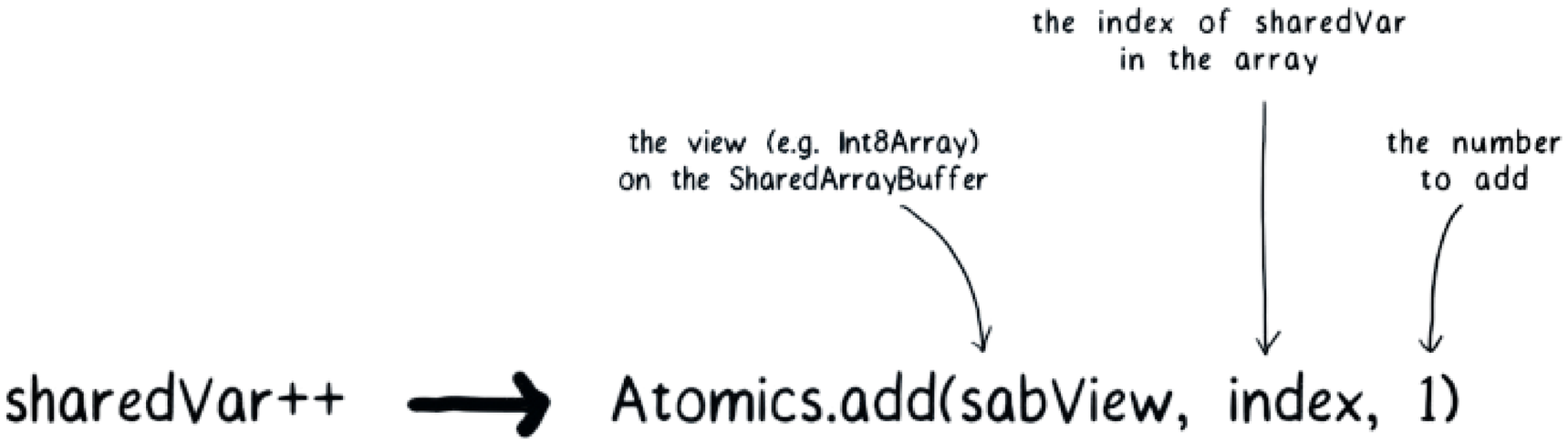
Race conditions

Atomics

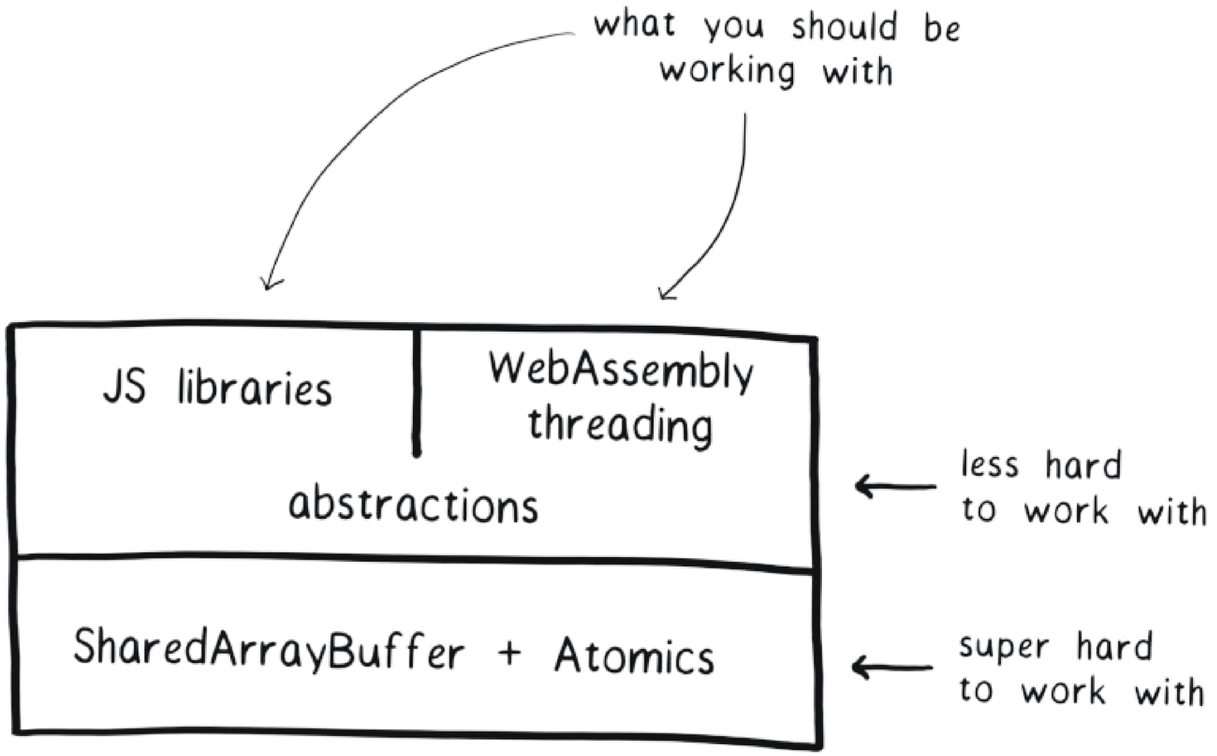
This object contains static methods for carrying out atomic operations.

They are used with SharedArrayBuffer and ArrayBuffer objects.



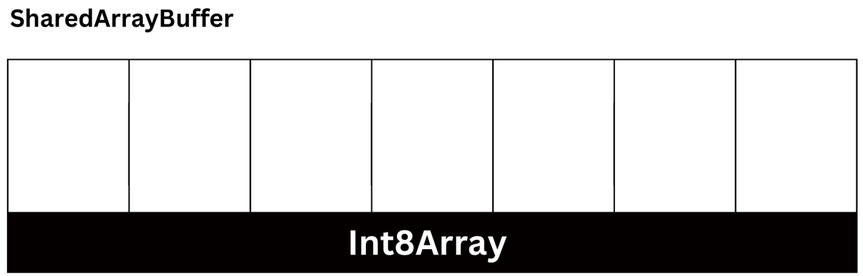

the index of sharedVar
in the array
the view (e.g. Int8Array)
on the SharedArrayBuffer
the number
to add
sharedVar++
Atomics.add(sabView, index, 1)

Atomics and **SharedBuffers** can be pretty challenging to handle.



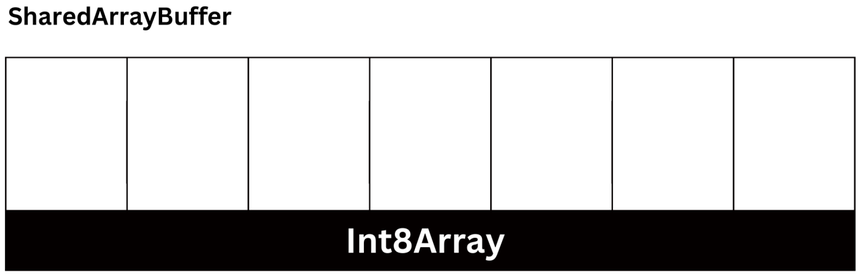
How to keep data using numbers?

```
const b = new SharedArrayBuffer( 7 );
const v = new Int8Array( b );
```

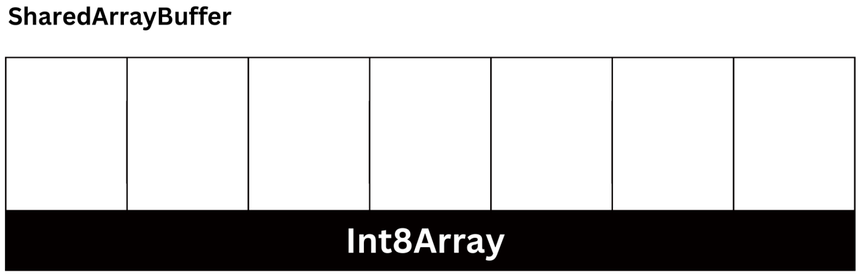


```
const b = new SharedArrayBuffer( 7 );
const v = new Int8Array( b );
```

h e l l o



```
const b = new SharedArrayBuffer( 7 );
const v = new Int8Array( b );
```



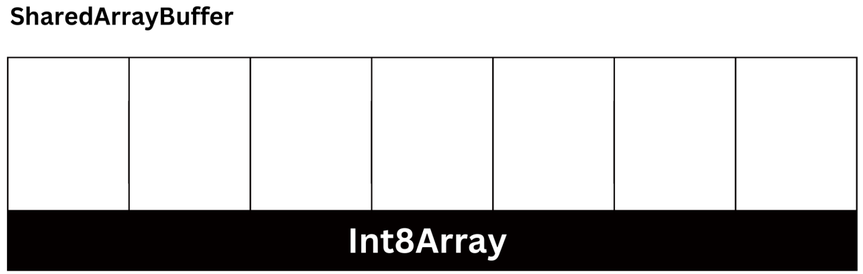
h e l l o

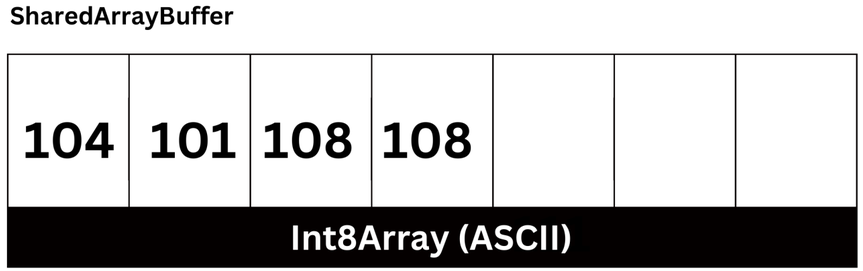
104 101 108 108 111

```
const b = new SharedArrayBuffer( 7 );
const v = new Int8Array( b );
```

```
h   e   l   l   o

104 101 108 108 111
```

```
v[0] = 104;
v[1] = 101;
...
```





worker_map

My experimental open-source project. #2

First, let's create a simple hash map structure in main process, then create a worker thread and share the hash.

```
// main.js
const { Worker } = require('worker_threads');
const { WorkerMap } = require('worker_map');

const map = new WorkerMap();
map.set('balance', 100); // sync operation

new Worker('./worker.js', {
  workerData: {
    mapBuffer: map.toSharedBuffer(),
  },
});

setTimeout(() => {
  console.log(map.get('balance')); // 200
}, 50);
```

Now, let's access the shared hash map structure in the worker thread.

```
// worker.js
const { WorkerMap } = require('worker_map');
const { workerData } = require('worker_threads');

const map = new WorkerMap(workerData.mapBuffer);
console.log(map.get('balance')); // 100

// The change will be reflected in the main process as well
map.set('balance', 200);
```

worker_map

Instance methods

Worker_map is much like JavaScript's regular Map.

`map.set(key, value)`

```
map.set('name', 'John'); // true
```

`map.get(key):`

```
const name = map.get('name'); // 'John'
```

`map.delete(key):`

```
map.delete('name'); // true
map.delete('something'); // false because it doesn't exist
```

`map.clear():`

```
map.clear();
map.size(); // 0
```

`map.has(key)`

```
map.has('name'); // true
map.has('country'); // false
```

`map.size()`

```
map.has('size'); // 1
```

`map.keys()`

```
map.keys(); // [ 'name' ]
```

`map.entries()`

```
for (const [ key, value ] of map.entries()) {
  console.log(`${key}: ${value}`); // name: 'John'
}
```

`map.forEach()`

```
map.forEach(function(key, value, map) {
  console.log(`${key}: ${value}`); // name: 'John'
});
```

`map.toSharedBuffer()`

```
const buffer = map.toSharedBuffer();
const sameMap = new WorkerMap(buffer);
```

`map.toObject()`

```
const mapObject = map.toObject(); // { ... }
mapObject.name; // 'John'
```

@nairihar

worker_map

Instance methods

Worker_map is much like JavaScript's regular Map.

`map.set(key, value)`

```
map.set('name', 'John'); // true
```

`map.get(key):`

```
const name = map.get('name'); // 'John'
```

`map.delete(key):`

```
map.delete('name'); // true
map.delete('something'); // false because it doesn't exist
```

`map.clear():`

```
map.clear();
map.size(); // 0
```

`map.has(key)`

```
map.has('name'); // true
map.has('country'); // false
```

`map.size()`

```
map.has('size'); // 1
```

`map.keys()`

```
map.keys(); // [ 'name' ]
```

`map.entries()`

```
for (const [ key, value ] of map.entries()) {
  console.log(`${key}: ${value}`); // name: 'John'
}
```

`map.forEach()`

```
map.forEach(function(key, value, map) {
  console.log(`${key}: ${value}`); // name: 'John'
});
```

`map.toSharedBuffer()`

```
const buffer = map.toSharedBuffer();
const sameMap = new WorkerMap(buffer);
```

`map.toObject()`

```
const mapObject = map.toObject(); // { ... }
mapObject.name; // 'John'
```

@nairihar

worker_map

My experimental open-source project. #2

First, let's create a simple hash map structure in main process, then create a worker thread and share the hash.

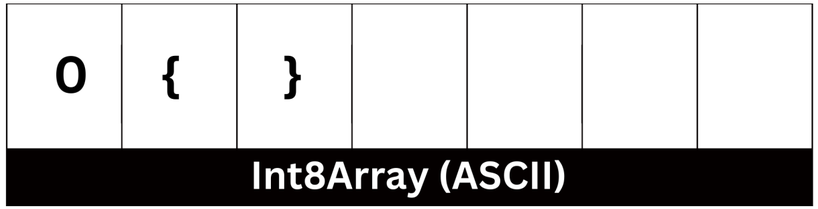
```
// main.js
const { Worker } = require('worker_threads');
const { WorkerMap } = require('worker_map');

const map = new WorkerMap();
map.set('balance', 100); // sync operation

new Worker('./worker.js', {
  workerData: {
    mapBuffer: map.toSharedBuffer(),
  },
});

setTimeout(() => {
  console.log(map.get('balance')); // 200
}, 50);
```

Now, let's access the shared hash map structure in the worker thread.

```
// worker.js
const { WorkerMap } = require('worker_map');
const { workerData } = require('worker_threads');

const map = new WorkerMap(workerData.mapBuffer);
console.log(map.get('balance')); // 100

// The change will be reflected in the main process as well
map.set('balance', 200);
```

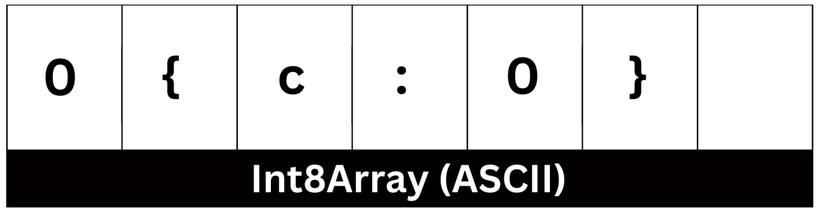
```
const map = new WorkerMap();
```

SharedArrayBuffer



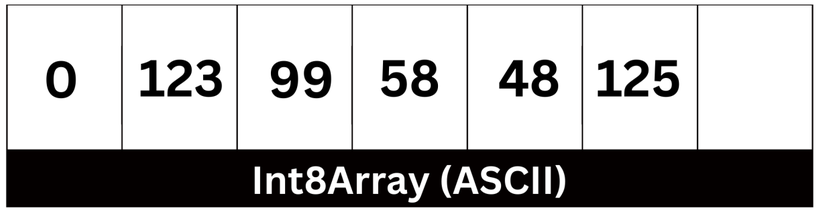

```
map.set(`c`, 0);
```

SharedArrayBuffer

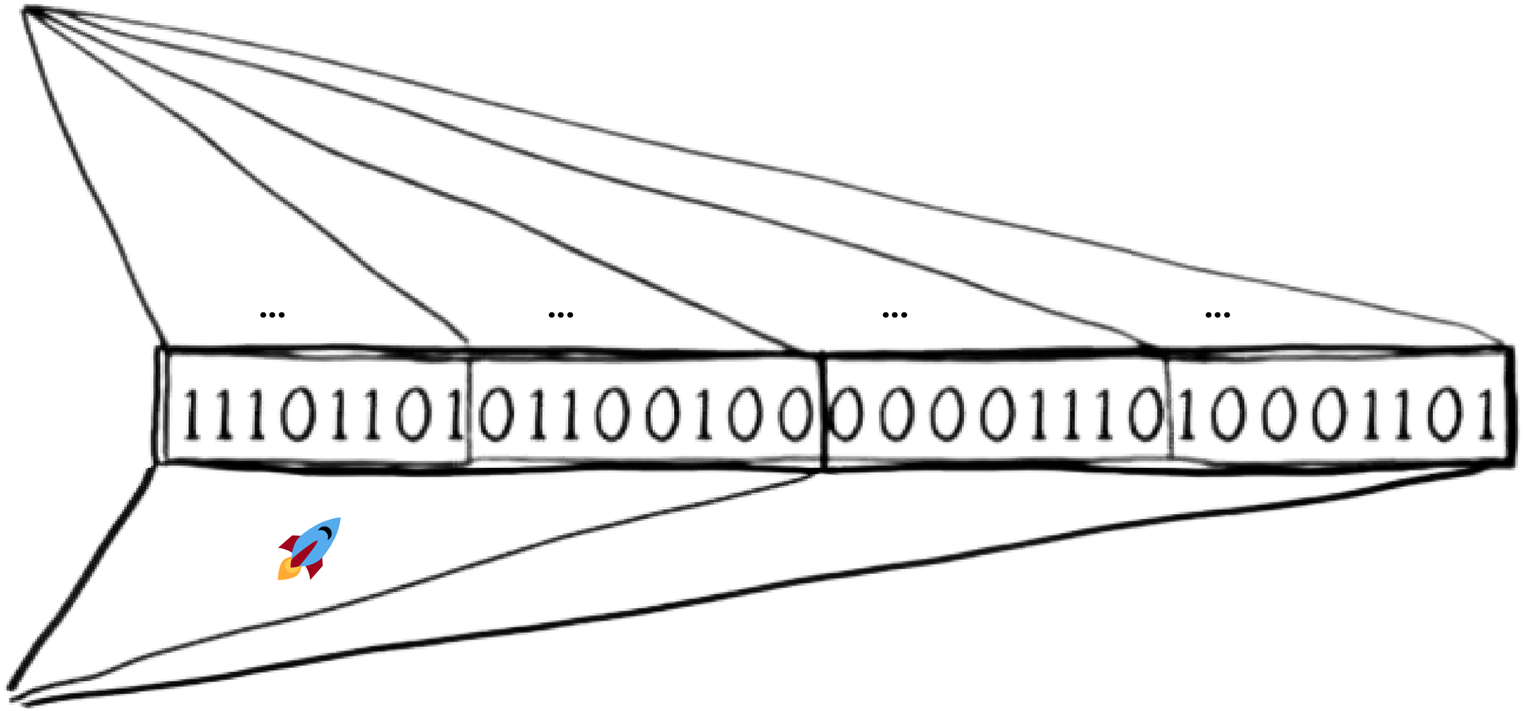



```
map.get(`c`);
```

0 - unlocked
1 - locked



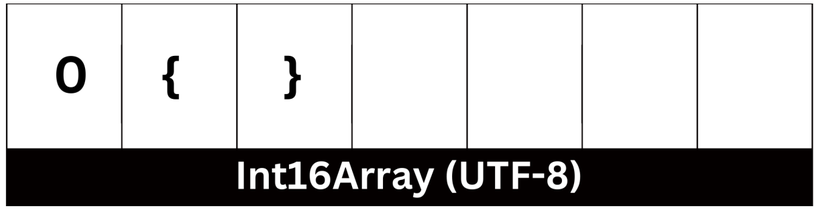
Int8Array



Int16Array

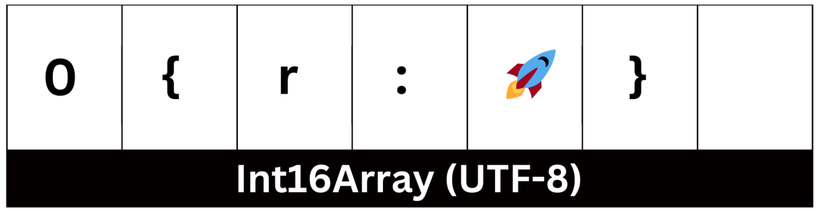
```
const map = new WorkerMap();
```

SharedArrayBuffer (64 bytes)



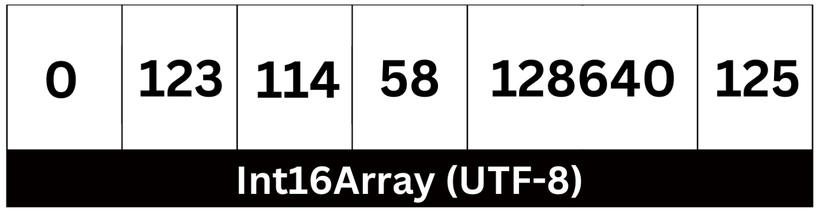

```
map.set(`r`, `🚀`);
```

SharedArrayBuffer (64 bytes)




```
map.get(`r`);
```

0 - unlocked
1 - locked



@nairihar

SharedArrayBuffer

▼ Constructor

SharedArrayBuffer() constructor

▼ Properties

SharedArrayBuffer[@@species]

SharedArrayBuffer.prototype.byteLeng

SharedArrayBuffer.prototype.growable

SharedArrayBuffer.prototype.maxByteL

▼ Methods

SharedArrayBuffer.prototype.grow()

SharedArrayBuffer.prototype.slice()

Growing SharedArrayBuffers

`SharedArrayBuffer` objects can be made growable by including the `maxByteLength` option when calling the `SharedArrayBuffer()` constructor. You can query whether a `SharedArrayBuffer` is growable and what its maximum size is by accessing its `growable` and `maxByteLength` properties, respectively. You can assign a new size to a growable `SharedArrayBuffer` with a `grow()` call. New bytes are initialized to 0.

These features make growing `SharedArrayBuffer`s more efficient — otherwise, you have to make a copy of the buffer with a new size. It also gives JavaScript parity with WebAssembly in this regard (Wasm linear memory can be resized with `WebAssembly.Memory.prototype.grow()`).

For security reasons, `SharedArrayBuffer`s cannot be reduced in size, only grown.

Available in Node >= v20

Is the `worker_thread` lightweight?

Any questions


THANK
YOU!
github.com/nairihar/ijs-conf-2023-munich
@nairihar


@nairihar