



@nairihar

The background of the slide is a dense, overlapping pile of numerous spools of thread in various colors, including yellow, blue, red, green, brown, and black. The spools are arranged in a somewhat organized manner, creating a textured, patterned base for the title text.

# A DEEP DIVE INTO NODE.JS MULTITHREADING

Before a brief introduction  
about myself . . .

# ***HEY I'M NAIRI \*\_\****

- 💻 Backend Engineer
- ❤️ Creator of the **JavaScript Armenia** Community
- ✍️ Writing about JavaScript and related technologies
- 🎙️ Hosting podcasts for the Armenian tech community
- 👉 Speaker, **JavaScript Summit** Conference Organizer
- ✉️ Newsletter writer on telegram: **@javascript**

**nairi.dev**



**@nairihar**

# AGENDA

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

We use it everywhere...

WEB

MOBILE

IOT

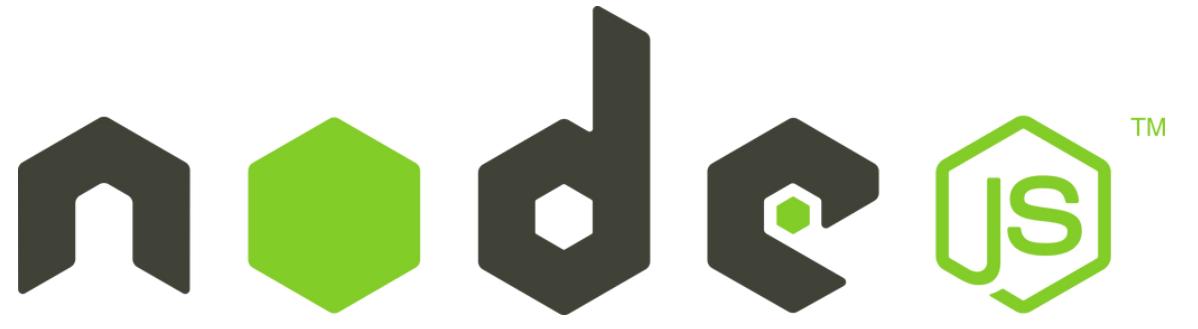
BACKEND

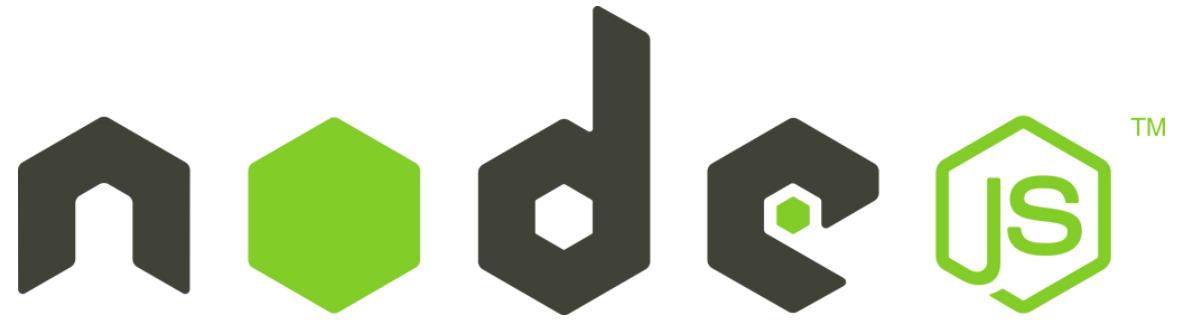


DESKTOP

GAME

...





# I/O in Operation systems

# I/O in Operation systems

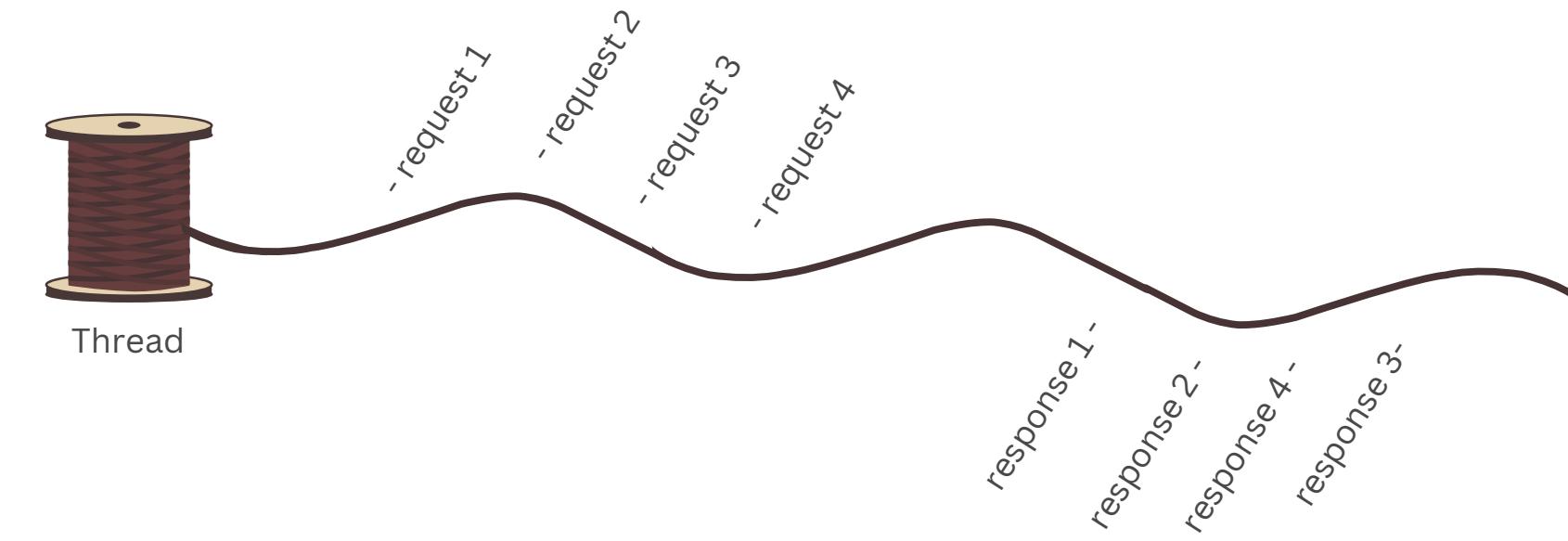
## Non blocking I/O

- *Network*

# I/O in Operation systems

## Non blocking I/O

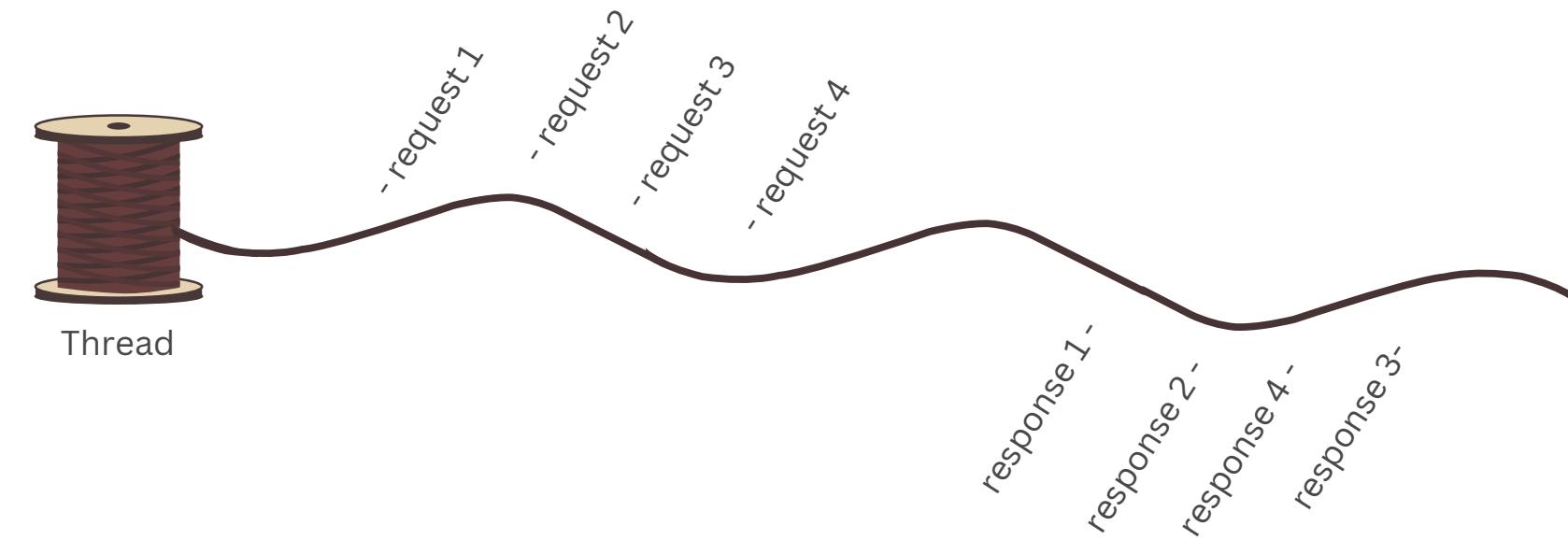
- *Network*



# I/O in Operation systems

## Non blocking I/O

- *Network*



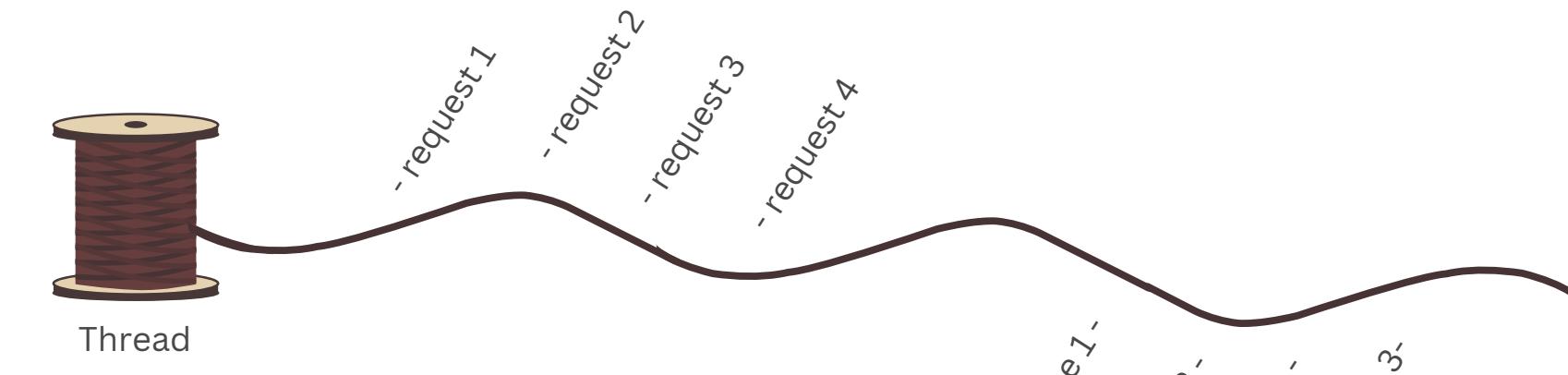
## Blocking I/O

- *File system*
- *DNS.lookup*

# I/O in Operation systems

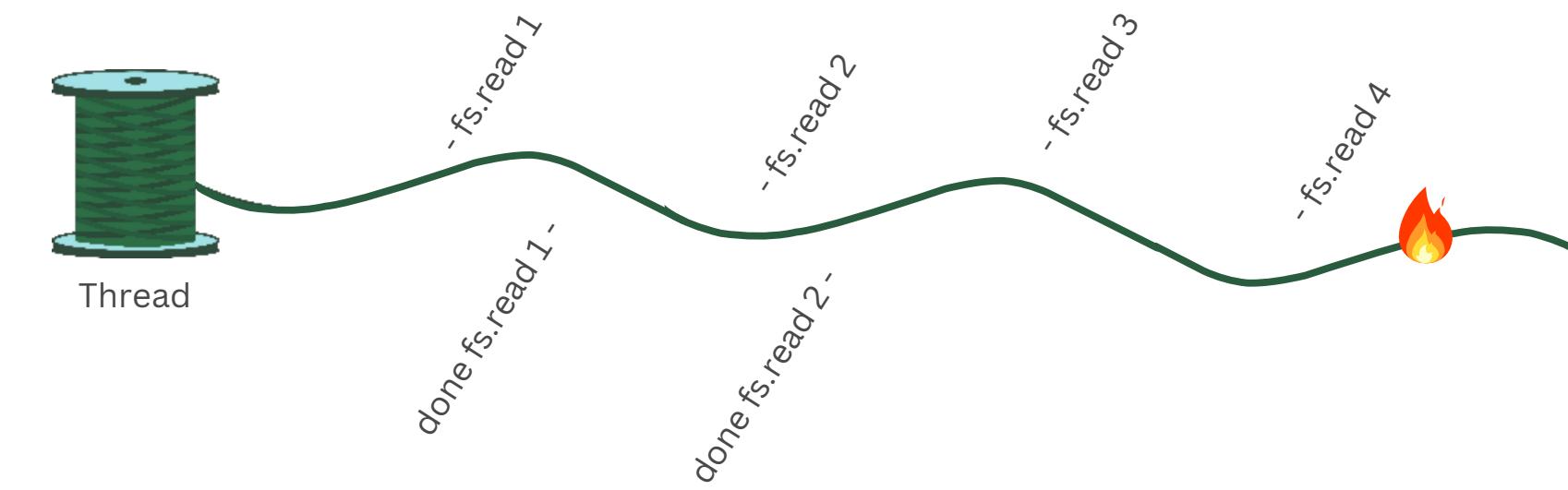
## Non blocking I/O

- *Network*



## Blocking I/O

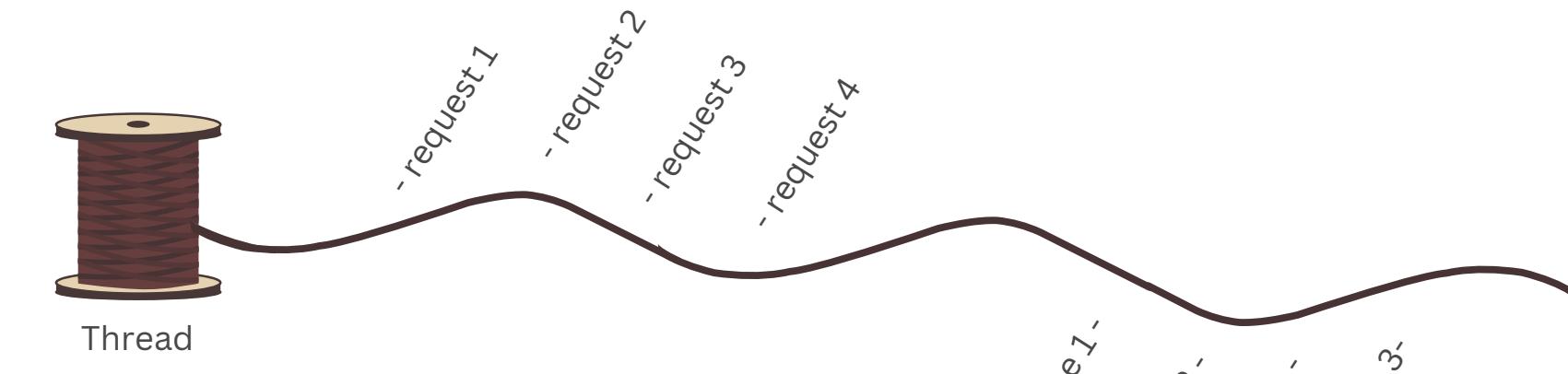
- *File system*
- *DNS.lookup*



# I/O in Operation systems

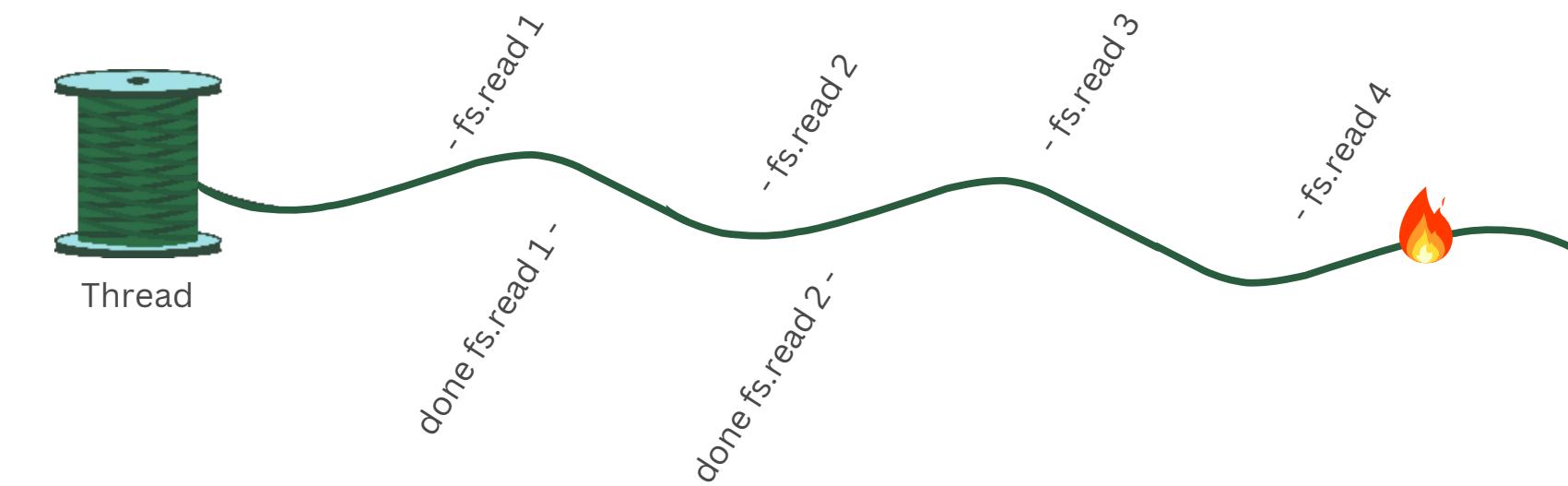
## Non blocking I/O

- *Network*



## Blocking I/O

- *File system*
- *DNS.lookup*



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

@nairihar

# 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

File I/O

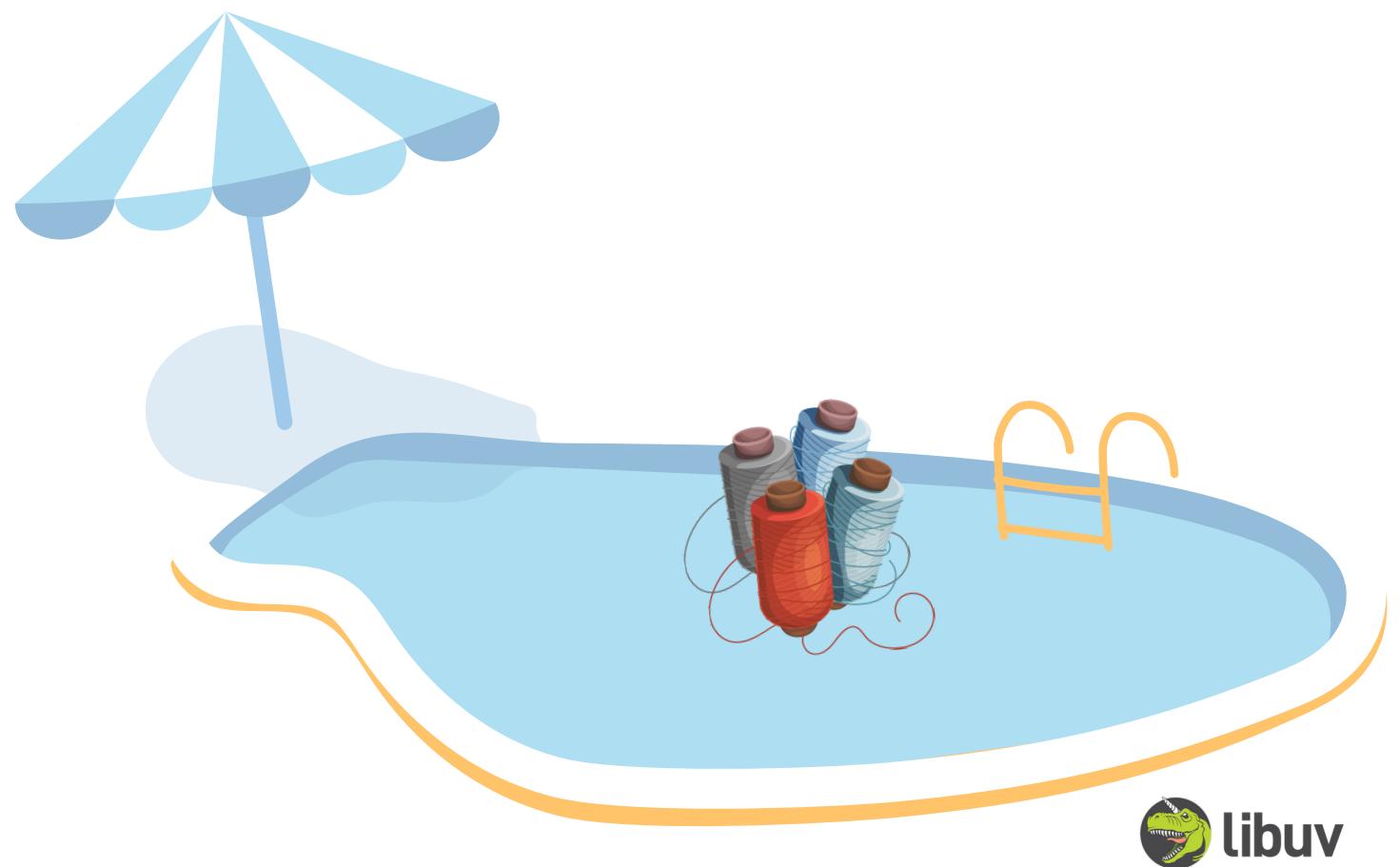
DNS

...

Thread Pool



# Thread Pool



# Thread Pool



**UV\_THREADPOOL\_SIZE=10** (Up to 1024 threads)

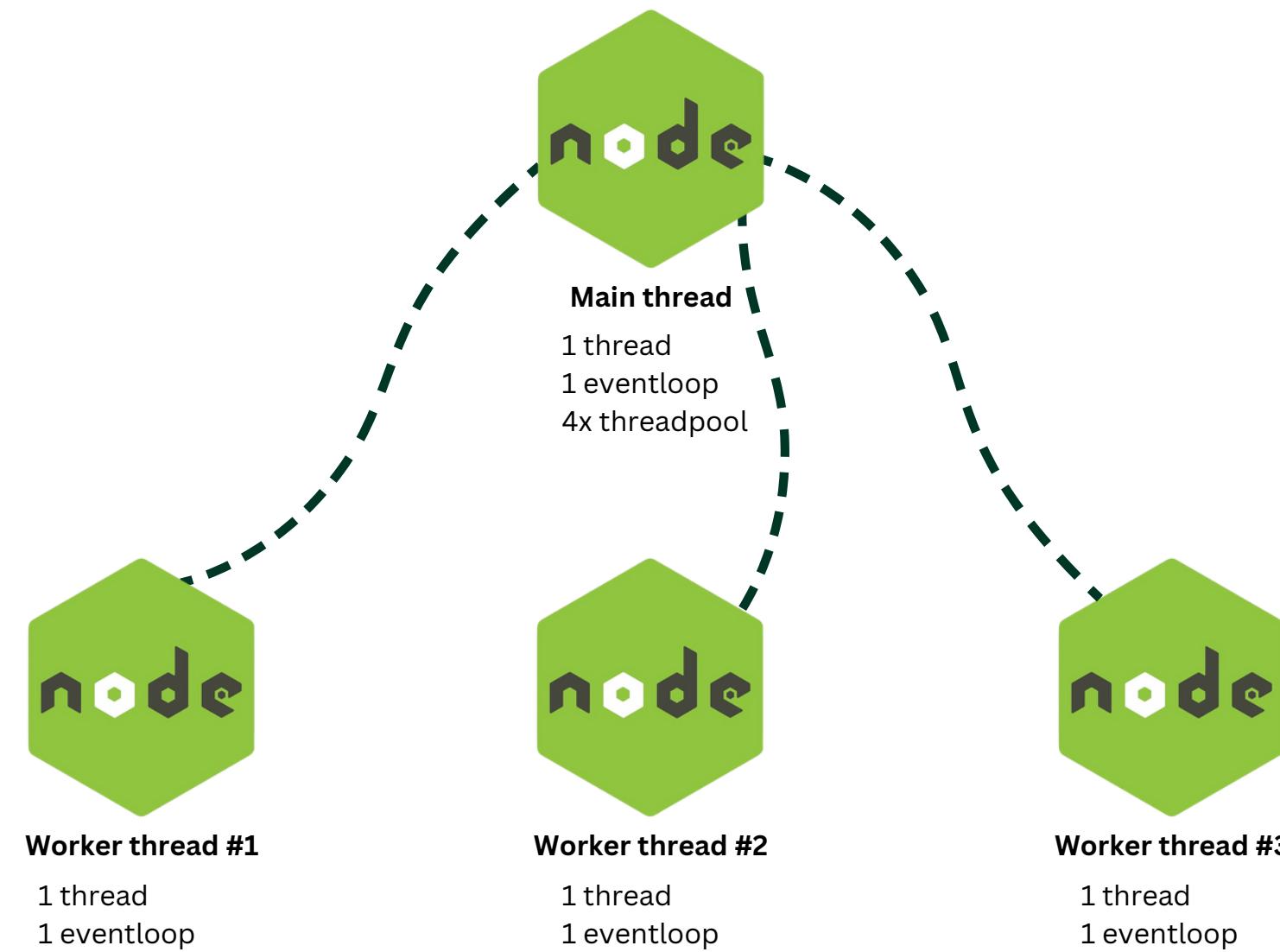
Before node v12.5.0: Up to 128 threads (before libuv v1.30.0)

We can't use **ThreadPool**  
**threads** directly...

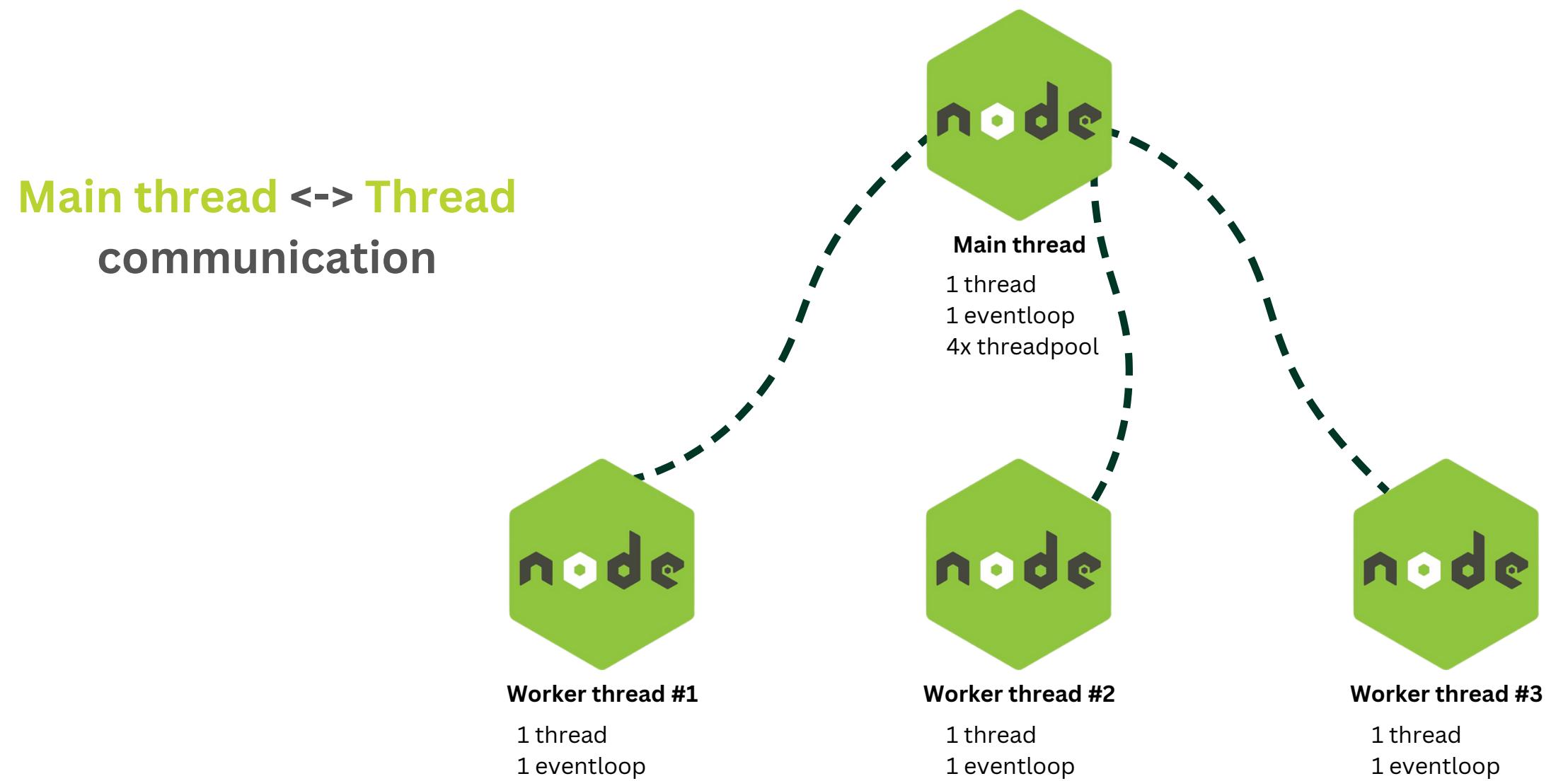
```
const { Worker } = require('worker_threads');
```

**Unlike ThreadPool threads...**

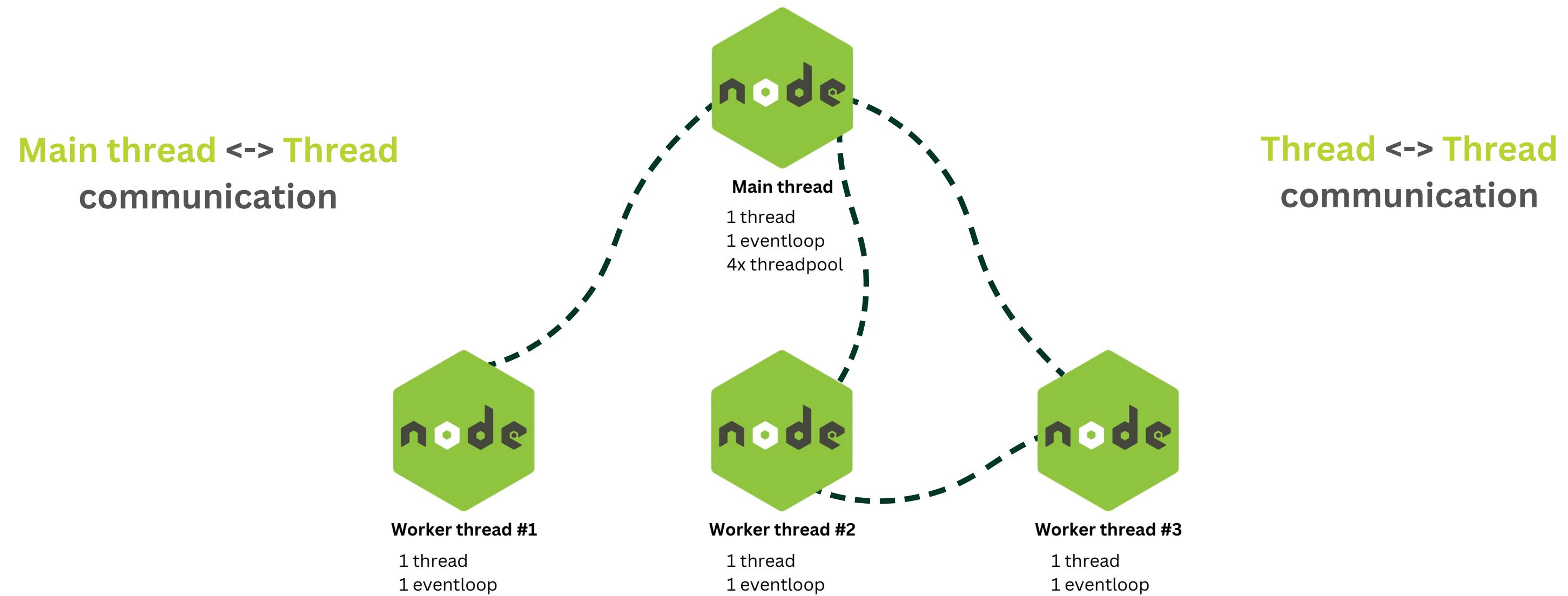
# 'worker\_threads'



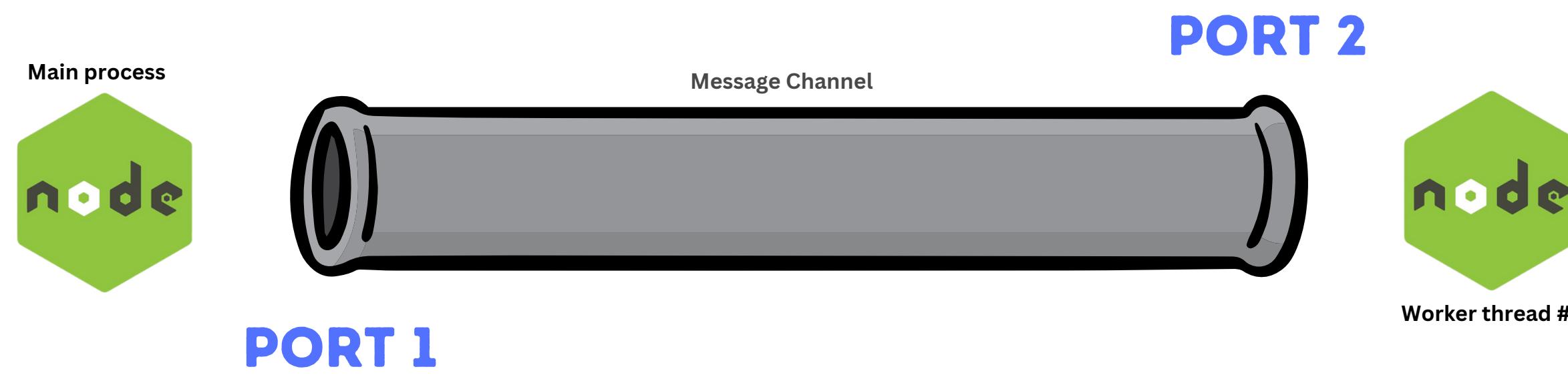
# 'worker\_threads'

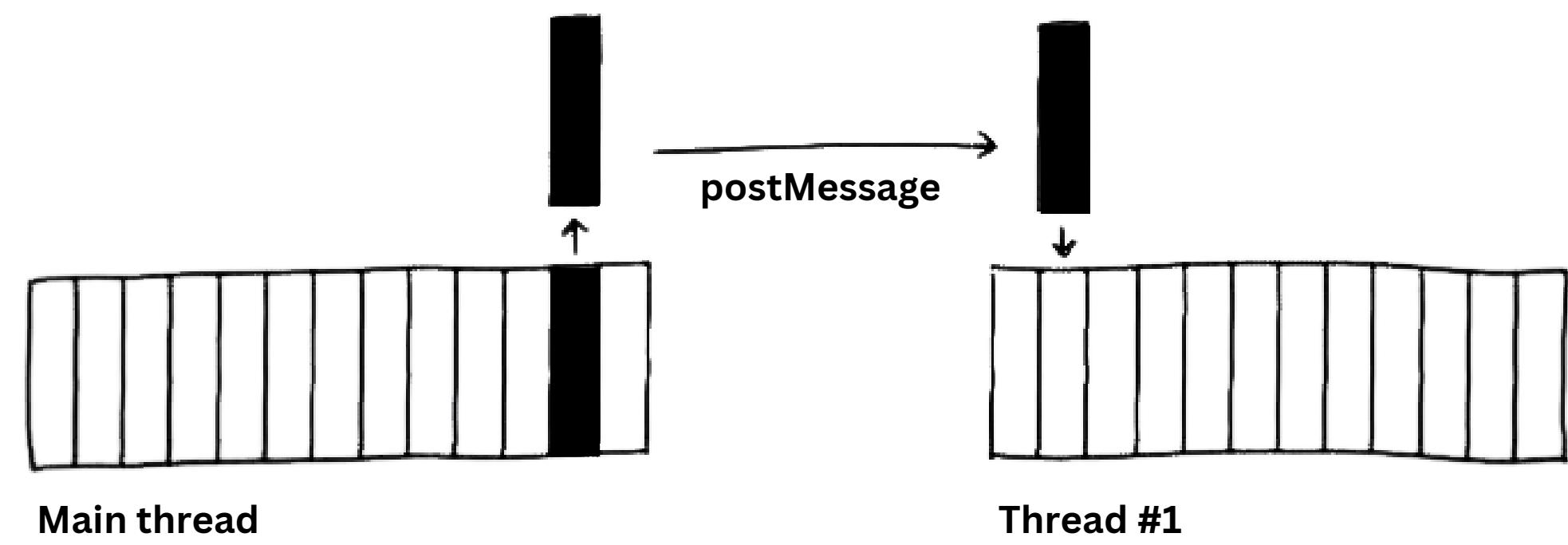


# 'worker\_threads'

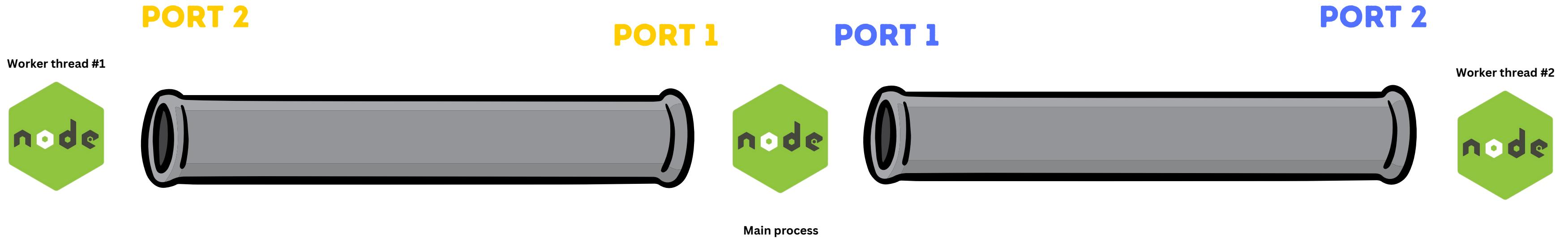


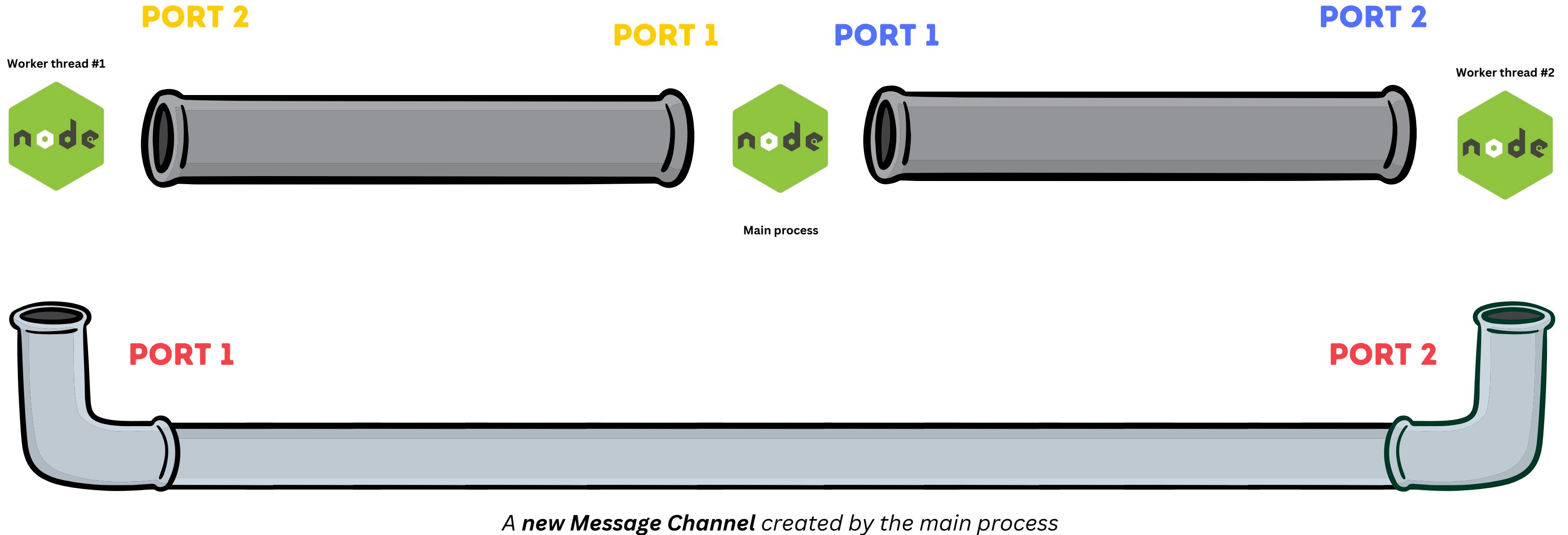
# DEMO





# Message Channels





# Abstractions



## 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)
  }
})
```



# 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.

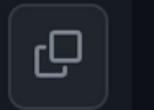
## Example

```
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++;
  }

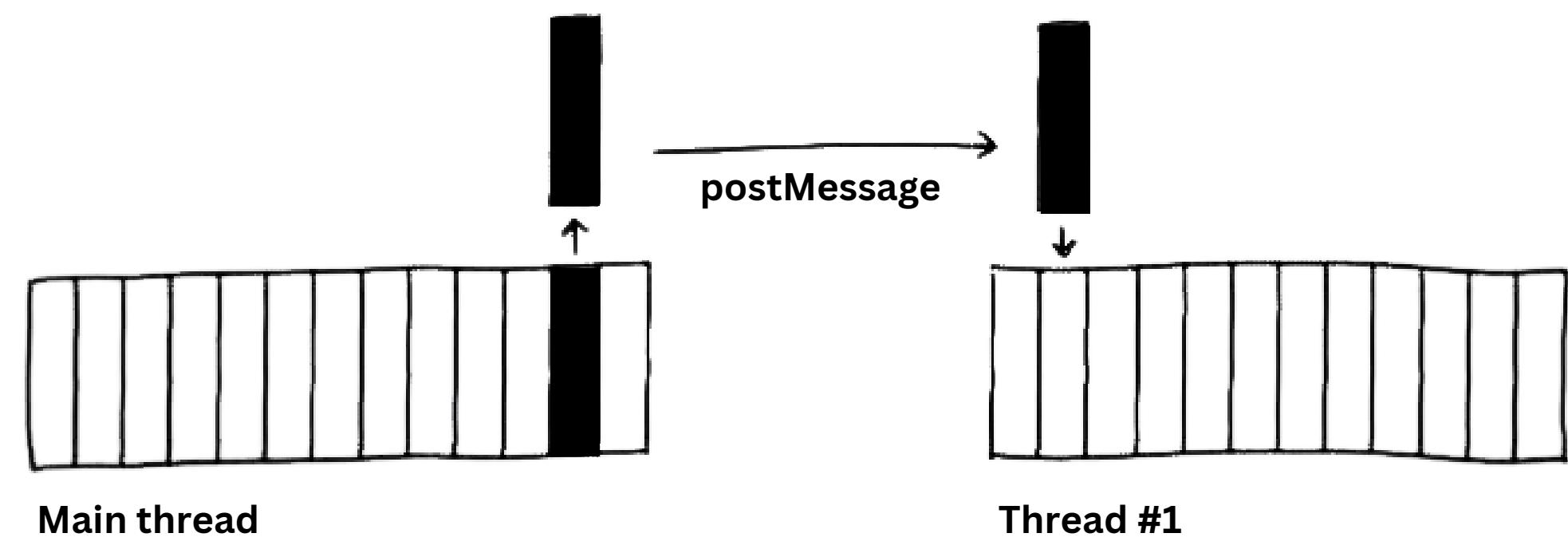
  return result;
}, 12345678);
```



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?

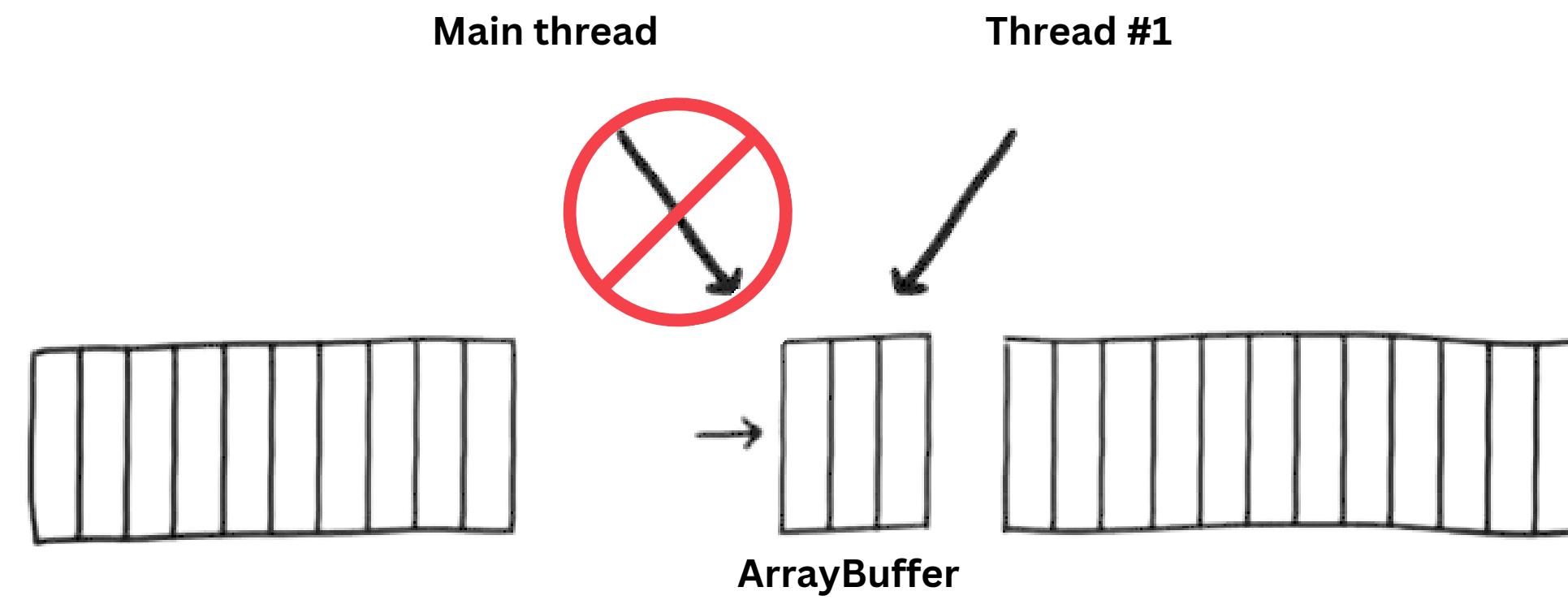
# How to share a state/data?



# ArrayBuffer

---

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

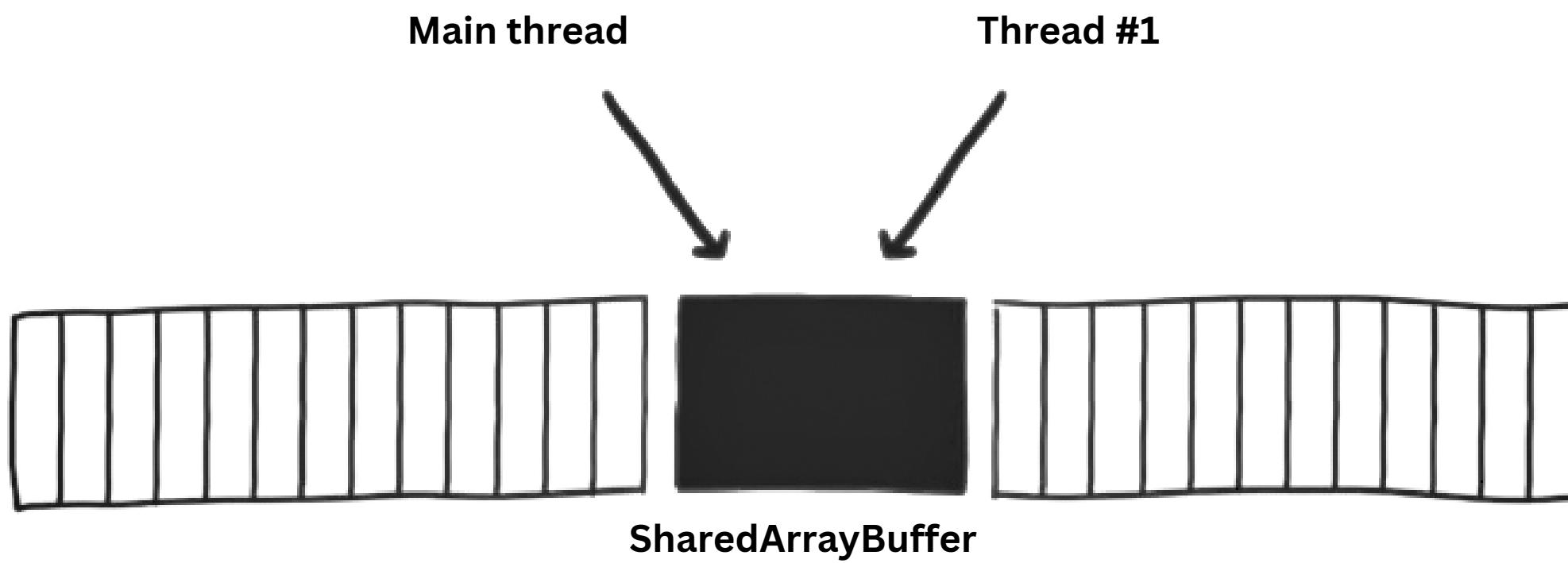


# SharedArrayBuffer

---

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

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



010010111010000110...

# 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**

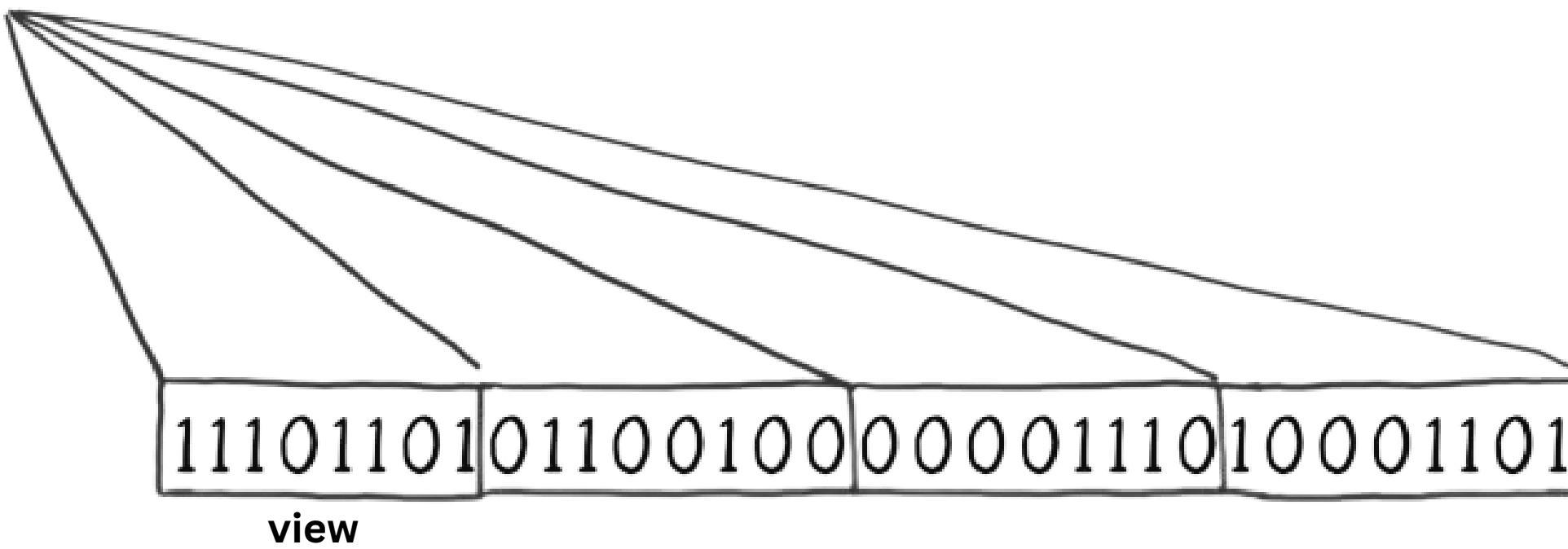
**Float32Array**

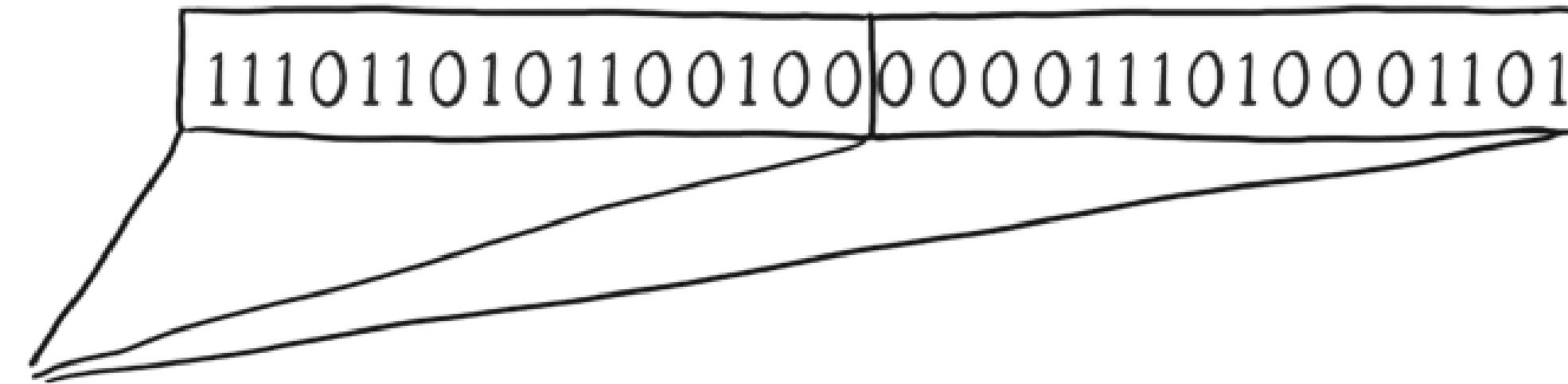
**Float64Array**

**BigInt64Array**

**BigUint64Array**

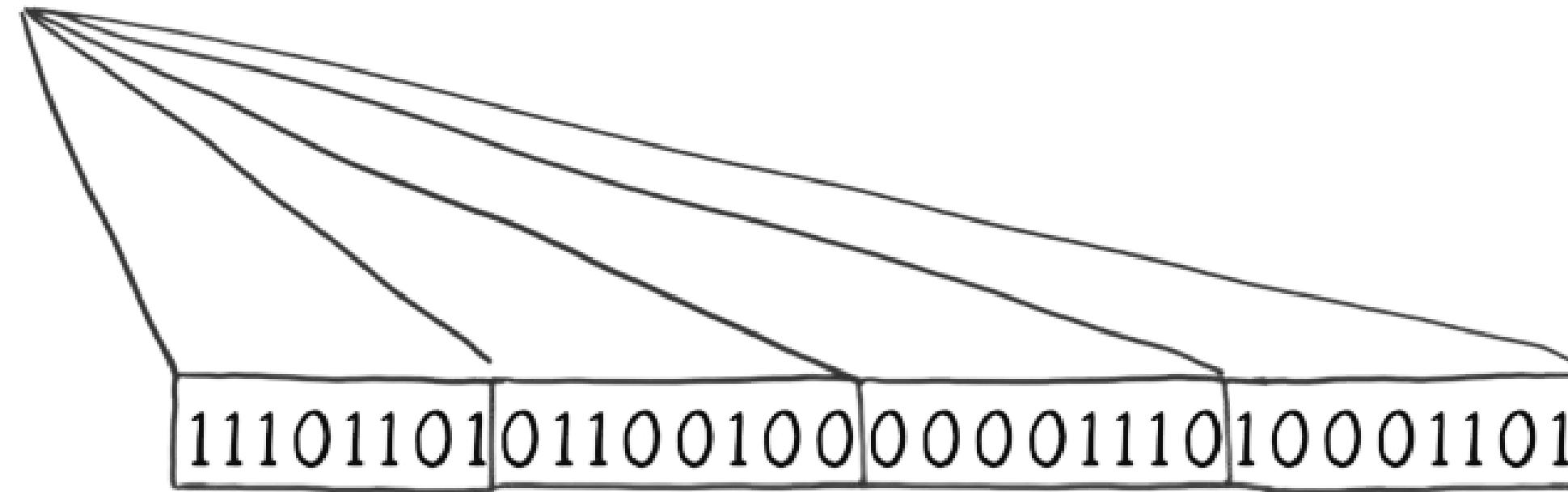
Int8Array



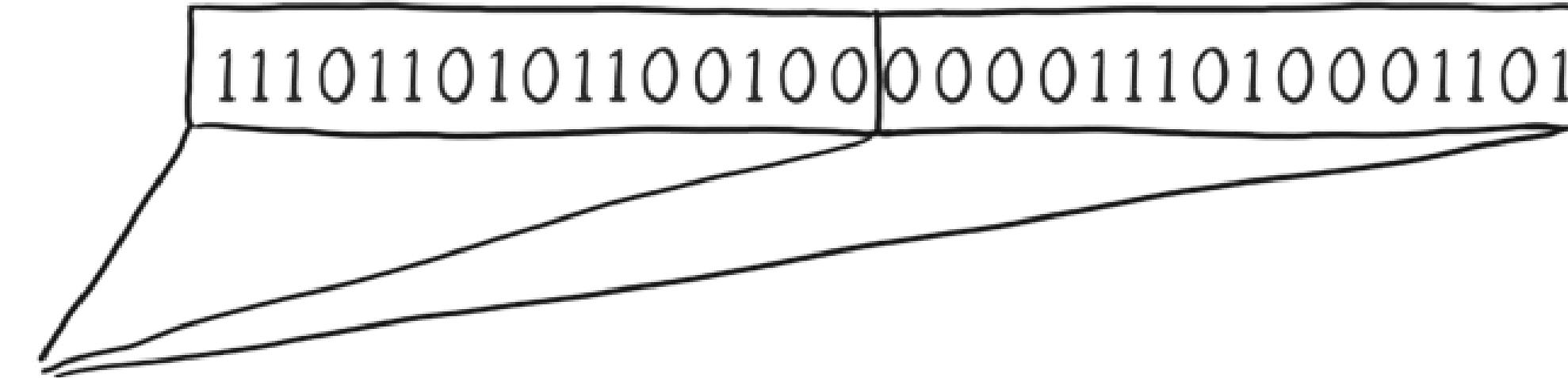


Uint16Array

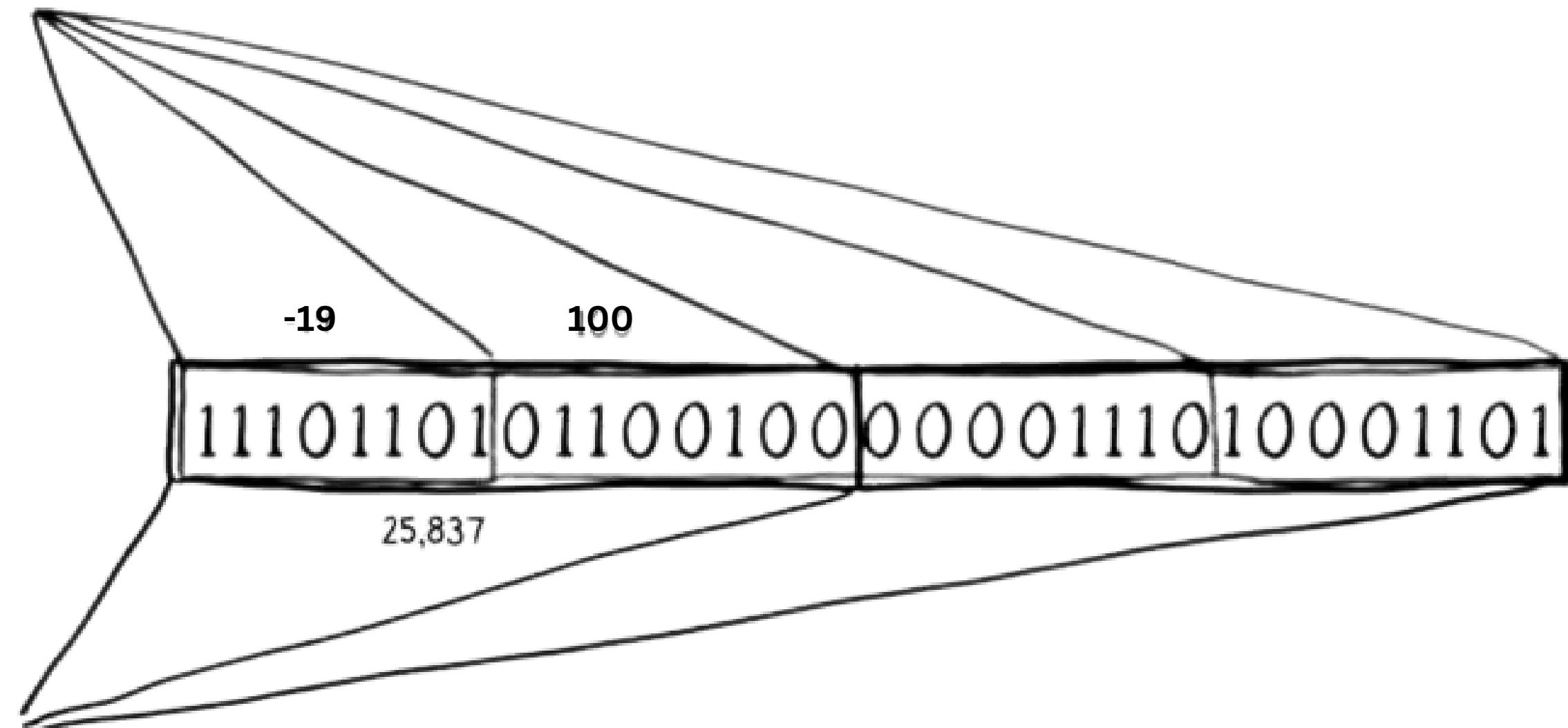
Int8Array



Uint16Array



Int8Array



Uint16Array

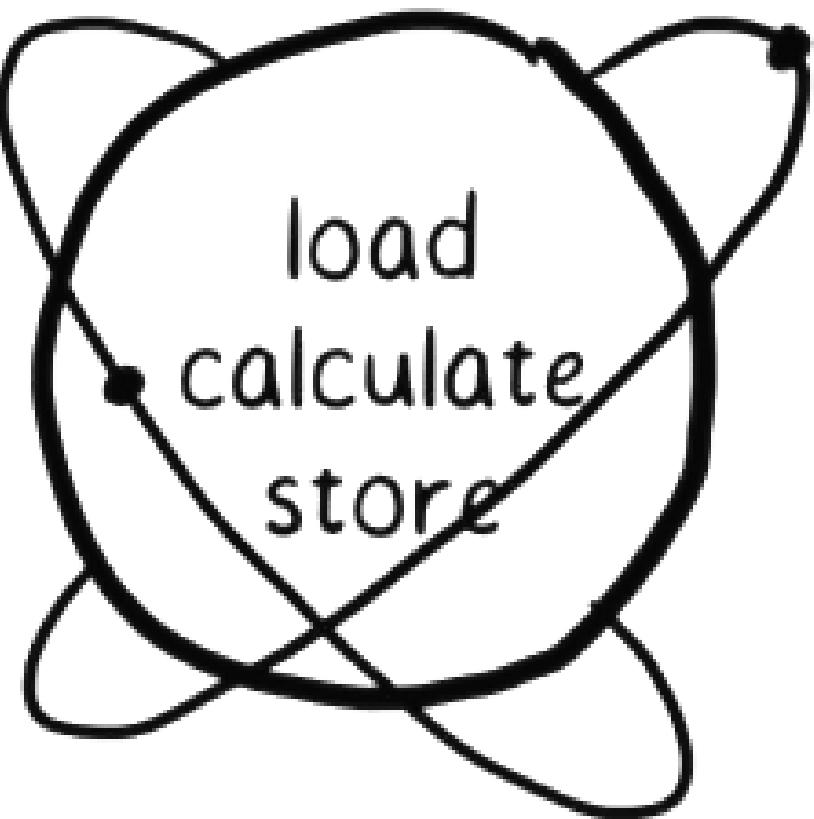
# Race conditions

# Atomicals

---

This object contains static methods for carrying out atomic operations.

They are used with SharedArrayBuffer and ArrayBuffer objects.



`sharedVar++` → `Atomics.add(sabView, index, 1)`

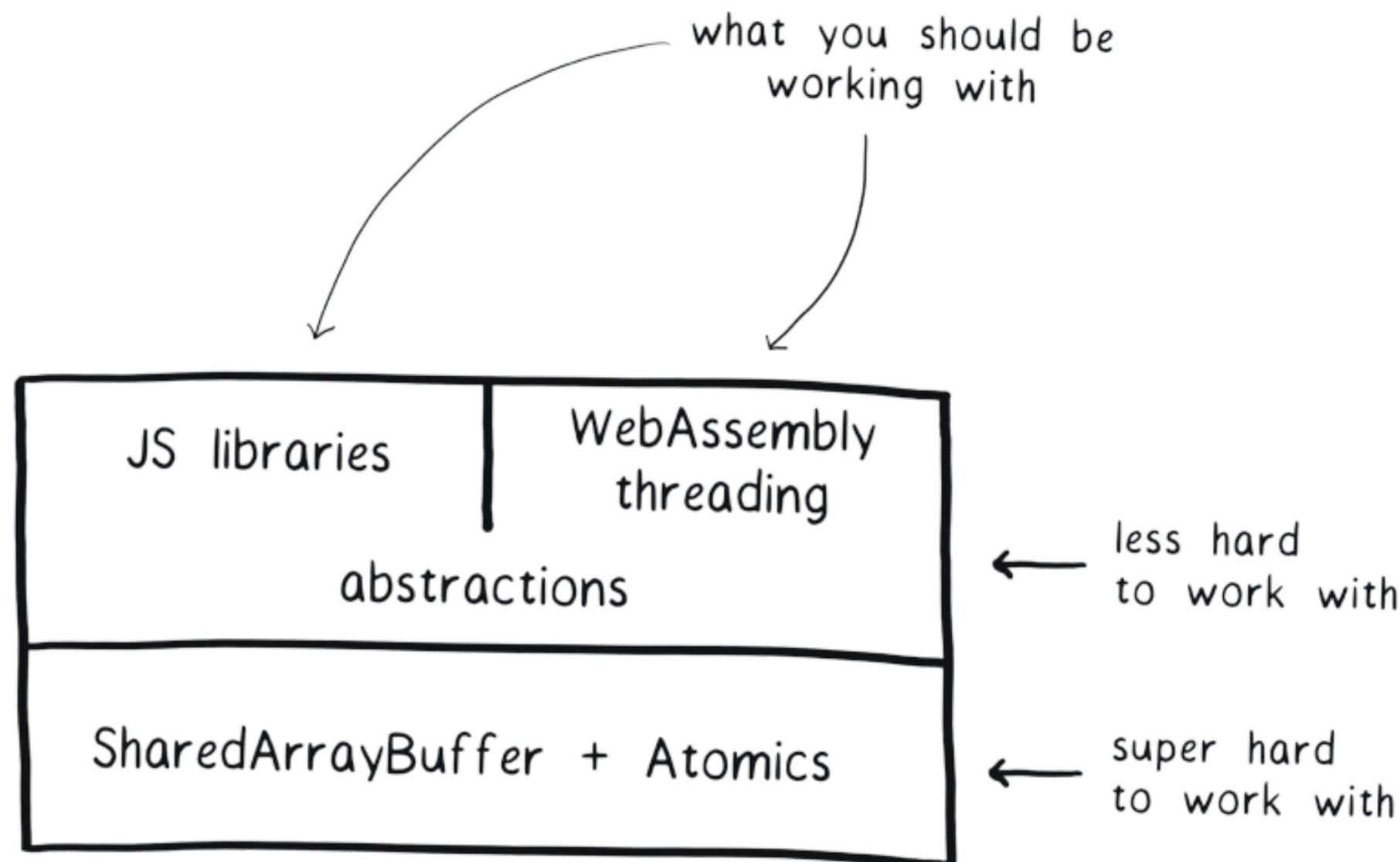
the view (e.g. Int8Array)  
on the SharedArrayBuffer

the index of sharedVar  
in the array

the number  
to add

```
graph LR; A[sharedVar++] --> B[Atomics.add(sabView, index, 1)]; C[the view (e.g. Int8Array) on the SharedArrayBuffer] --> D[sabView]; E[the index of sharedVar in the array] --> F[index]; G[the number to add] --> H[1];
```

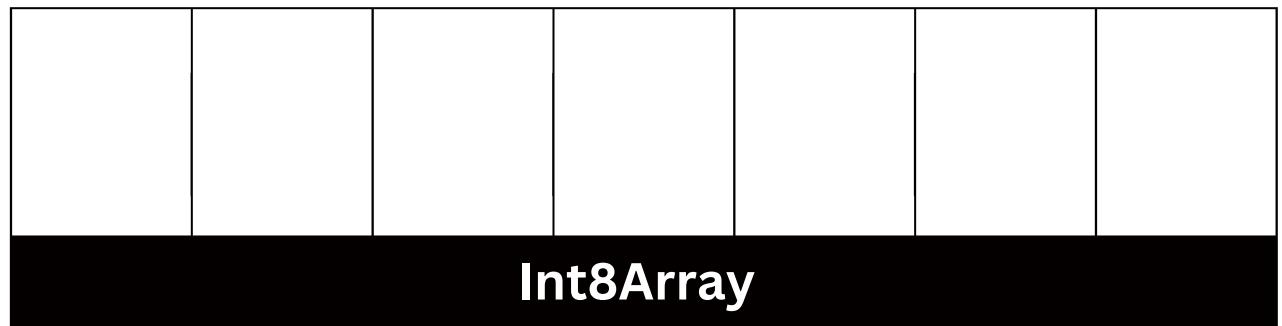
# Atomics and SharedBuffers can be pretty challenging to handle.



# How to keep data using numbers?

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

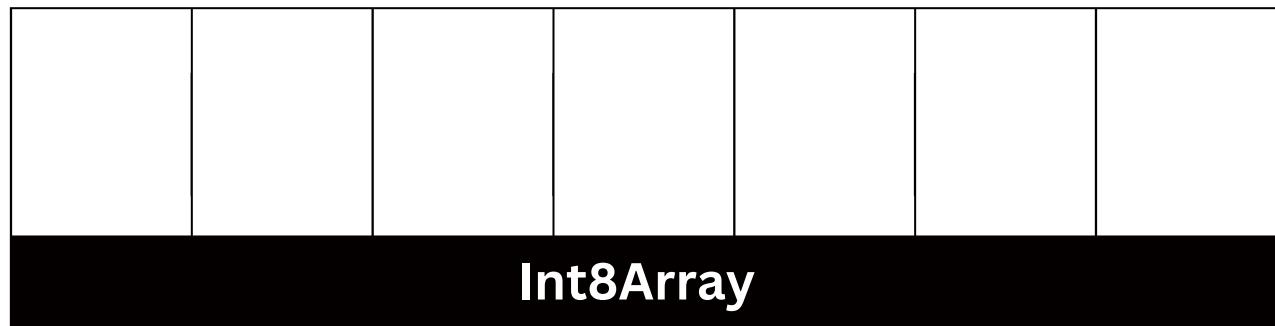
SharedArrayBuffer



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

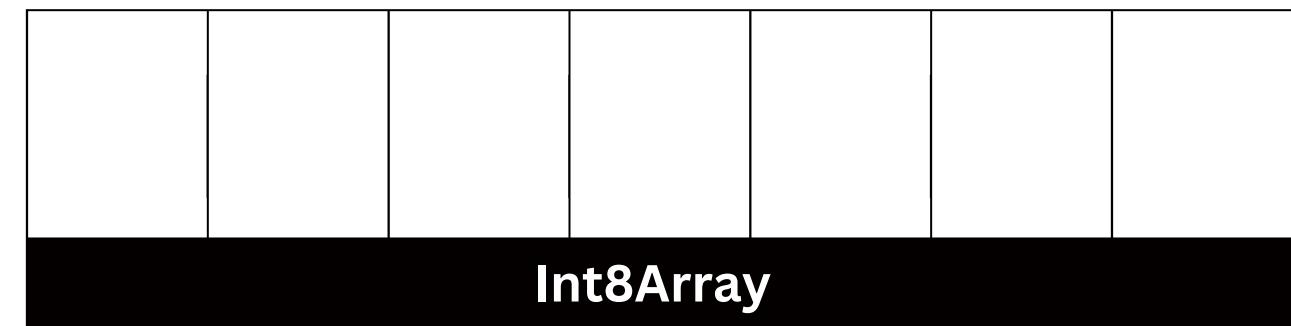
h e l l o

SharedArrayBuffer



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

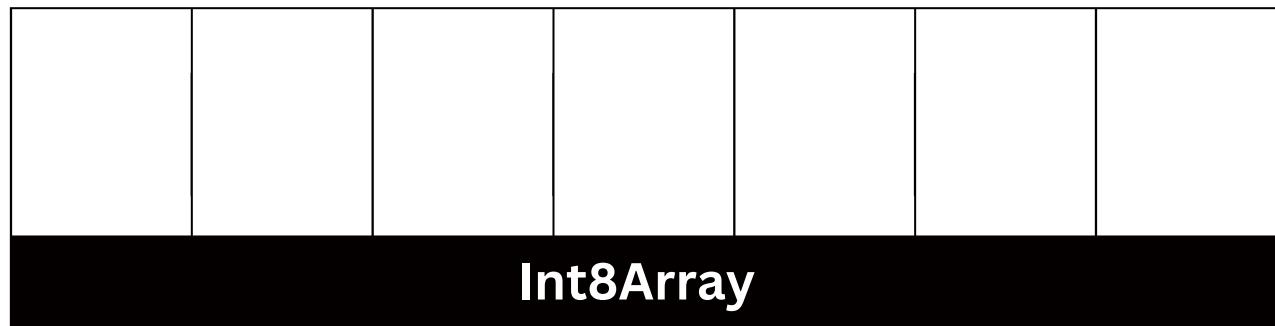
SharedArrayBuffer



h e l l o  
104 101 108 108 111

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

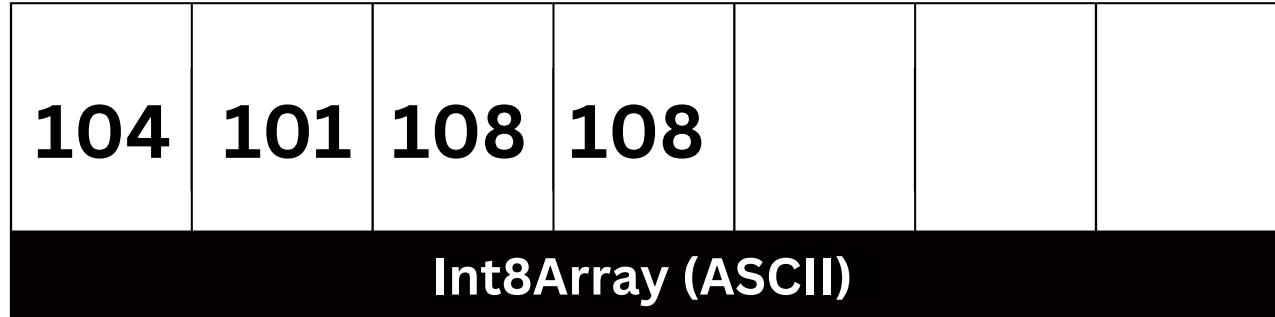
SharedArrayBuffer



h e l l o  
104 101 108 108 111

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

SharedArrayBuffer



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: ${key}`); // name: 'John'  
}
```

### map.forEach() ↵

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

### map.toSharedBuffer() ↵

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

### map.toObject() ↵

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

# 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'
```

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.

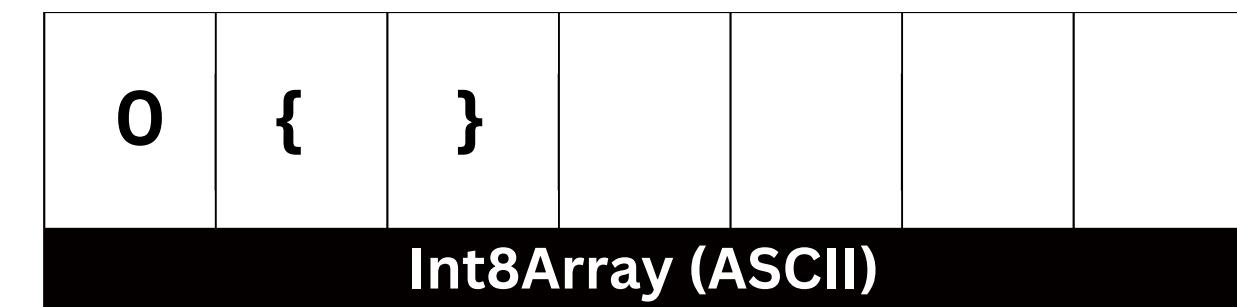
```
// worker.js
const { WorkerMap } = require('worker_map');
const { workerData } = require('worker_threads');

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);
```

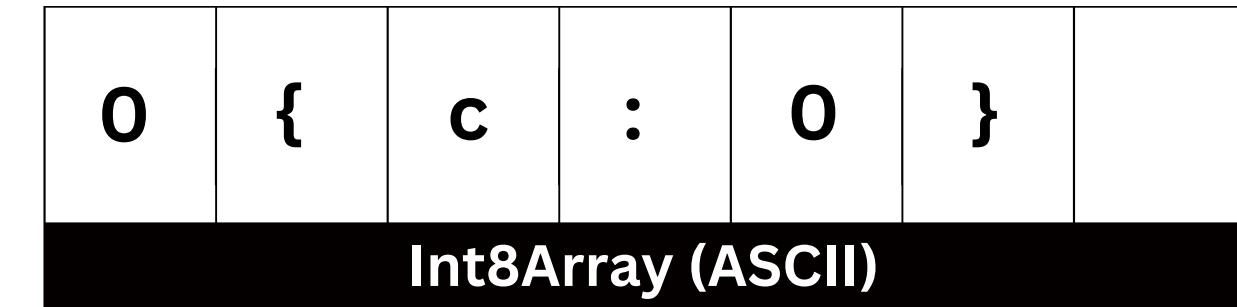
```
const map = new WorkerMap();
```

SharedArrayBuffer



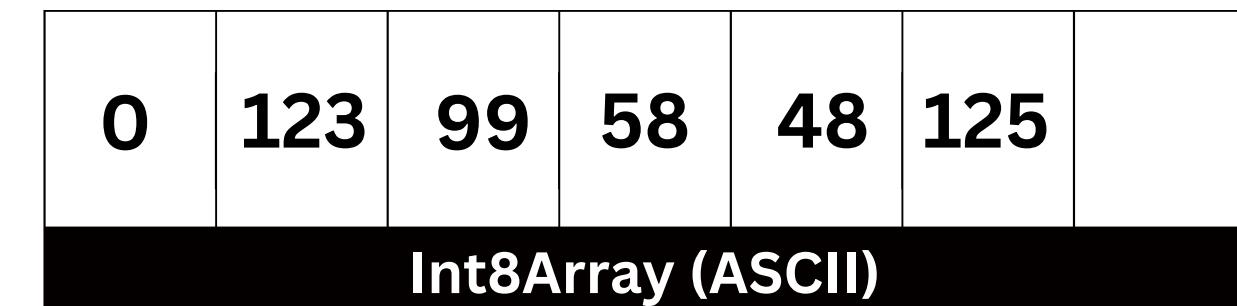
```
map.set(`c`, 0);
```

SharedArrayBuffer

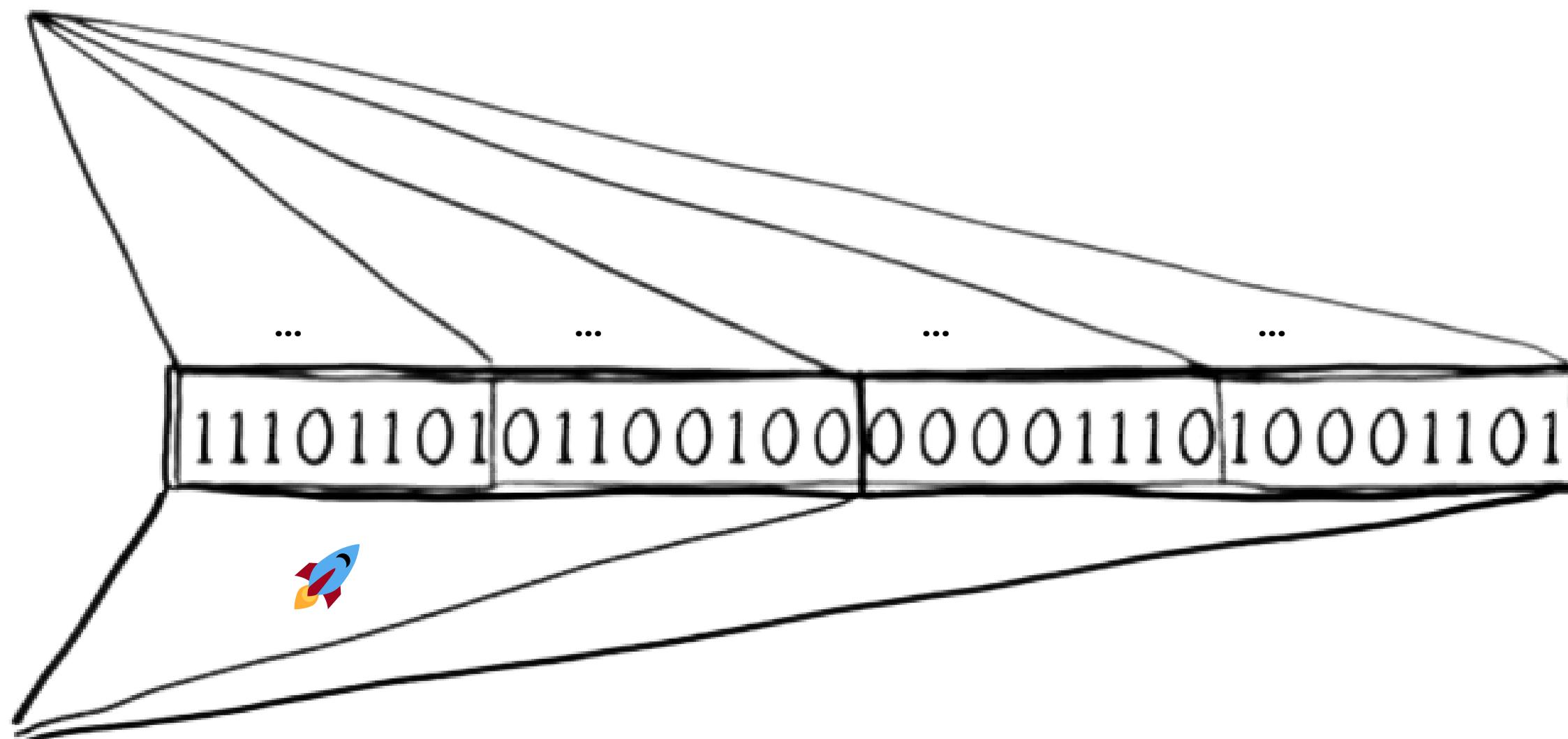


```
map.get(`c`);
```

0 - unlocked  
1 - locked



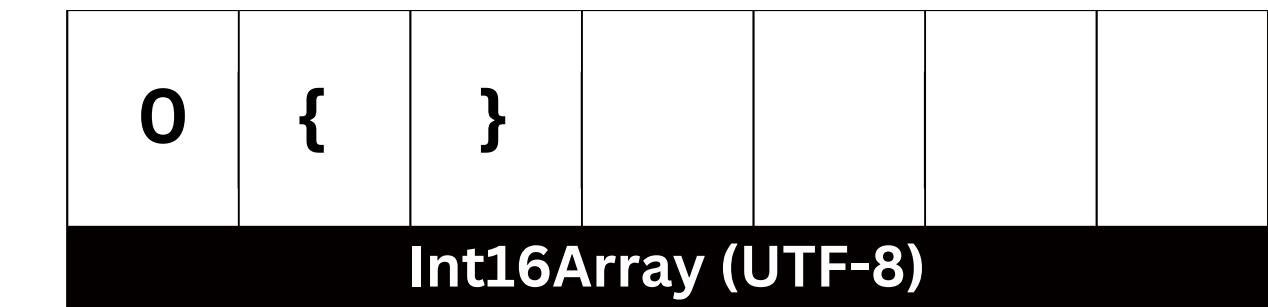
# **Int8Array**



# **Int16Array**

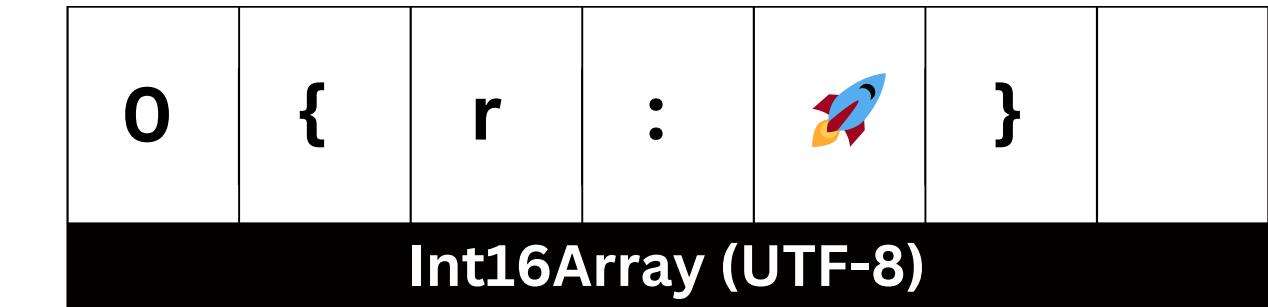
```
const map = new WorkerMap();
```

SharedArrayBuffer (64 bytes)



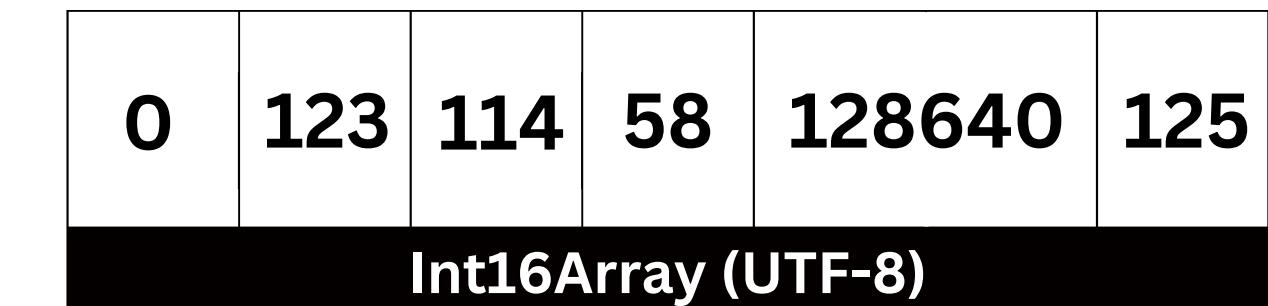
```
map.set(`r`, `🚀`);
```

SharedArrayBuffer (64 bytes)



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

0 - unlocked  
1 - locked



## 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!

