NodeJS from single thread to worker threads

Sebastian Curland | @sebcurland

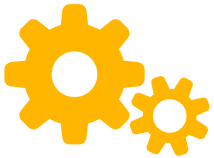
Hello!

I am Sebastian Curland

Senior Developer @ nielsen

@sebcurland

Node app



1 process
1 thread

Golden Rule

Don’t block the event loop

Running CPU Intensive tasks

`setImmediate()`

Partitioning long-running synchronous code

`setImmediate()`

- Complicated
- Not always possible

Offloading

Child Process Cluster

child_process / cluster

1 process
1 thread

1 process
1 thread

1 process
1 thread

Drawbacks

- No shared memory - all data is cloned
- Spawning processes it's not cheap

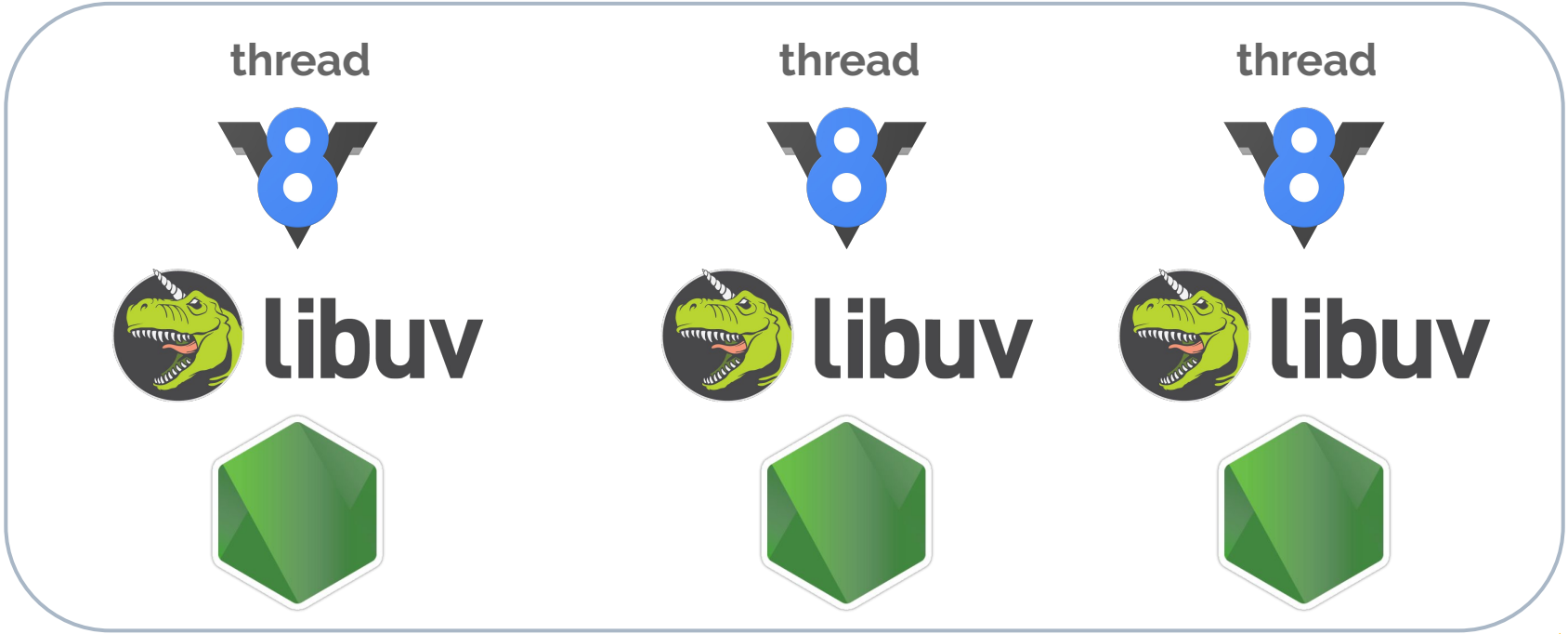
Worker threads

stable since Node v12.15

👏 👏 👏 👏 👏 👏

worker_threads

1 process



Parent

```
const { Worker } = require('worker_threads')
const worker = new Worker('./worker.js')

worker.on('message', message => console.log(message))
worker.postMessage('ping')
```

Worker

```
const { parentPort } = require('worker_threads')
parentPort.on('message', message =>
    parentPort.postMessage({ pong: message })
)
```

isMainThread and threadId

```
const {Worker, isMainThread} = require('worker_threads')

if (isMainThread) {
  const worker = new Worker(__filename)
  console.log(worker.threadId)
} else {
  console.log('Hello')
}
```

Worker Events

- **message** - worker thread invoked `parentPort.postMessage()`
- **exit** - worker has stopped
- **error** - worker thread throws an uncaught exception
- **online** - worker thread has started executing JS code

What makes Worker Threads special

1

Worker options

`new Worker(filename[, options])`

workerData

```
const {Worker, isMainThread, workerData} = require('worker_threads')

if (isMainThread) {
 const worker = new Worker(__filename, {workerData: 'Hello'})
} else {
 console.log(workerData)  // 'Hello'
}
```

env

```
const { Worker, isMainThread } = require('worker_threads')

if (isMainThread) {
 const worker = new Worker(__filename)
 worker.on('exit', () => {
   console.log(process.env.SET_IN_WORKER) // undefined
 })
} else { process.env.SET_IN_WORKER = 'foo' }
```

env: SHARE_ENV

```
const {Worker, isMainThread, SHARE_ENV} = require('worker_threads

if (isMainThread) {
 const worker = new Worker(__filename, {env: SHARE_ENV})
 worker.on('exit', () => {
   console.log(process.env.SET_IN_WORKER) // foo
 })
} else { process.env.SET_IN_WORKER = 'foo' }
```

eval

```
const { Worker, SHARE_ENV } = require('worker_threads')

new Worker('process.env.WORKER="foo"', {eval: true, env: SHARE_ENV })
 .on('exit', () => {
   console.log(process.env.WORKER)  // Prints 'foo'.
 })
```

resourceLimits

```
const { Worker } = require('worker_threads')
const worker = new Worker('.worker.js',
          {resourceLimits: {maxOldGenerationSizeMb: 5}})
```

max size of heap in MB

Error [ERR_WORKER_OUT_OF_MEMORY]:
Worker terminated due to reaching memory limit

2

MessageChannel

async two-way communication channel between threads

messageChannel

```
const { MessageChannel } = require('worker_threads')

const { port1, port2 } = new MessageChannel()

port1.on('message', (message) => console.log(message))
port2.postMessage({ foo: 'bar' })
```

(port1 and port2 are instances of **MessagePort**)

```
const worker = new Worker('./worker.js')
const { parentPort } = require('worker_threads')
```

3

MessagePort

Used to transfer data between workers

(extends `EventEmitter`)

MessagePort Events

- **message** - incoming message, containing the cloned input of `port.postMessage()`
- **close** - emitted once the channel has been disconnected or on **port.close()**

4

port.postMessage

(value[, transferList])

Ways to pass data between threads:

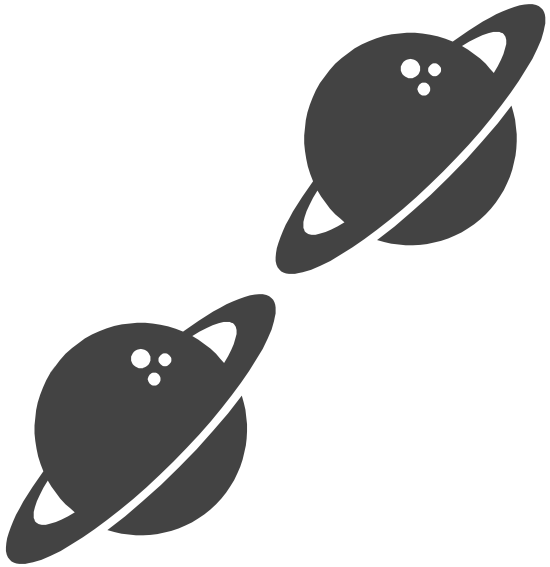
- Clone the data
- Transfer it
- Shared the same data

postMessage Value

- **Data is cloned** (using structured clone algorithm)
- JS types (Map, Set, BigInt, etc)
- ArrayBuffers
- Circular references

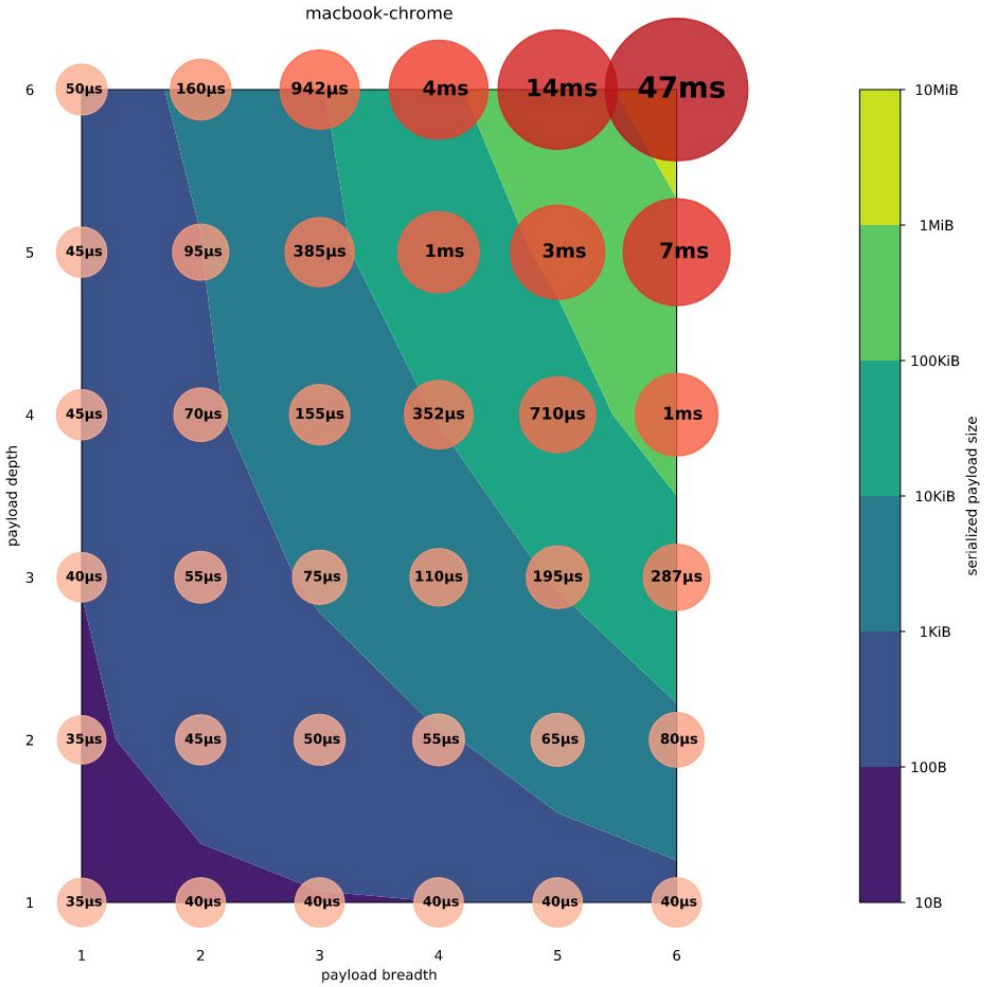
```
const circularData = {}
circularData.foo = circularData
```

The more complex the data structure, **the more computing power it takes to clone it.**

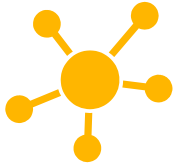


@DasSurma

https://surma.dev/things/is-postmessage-slow/

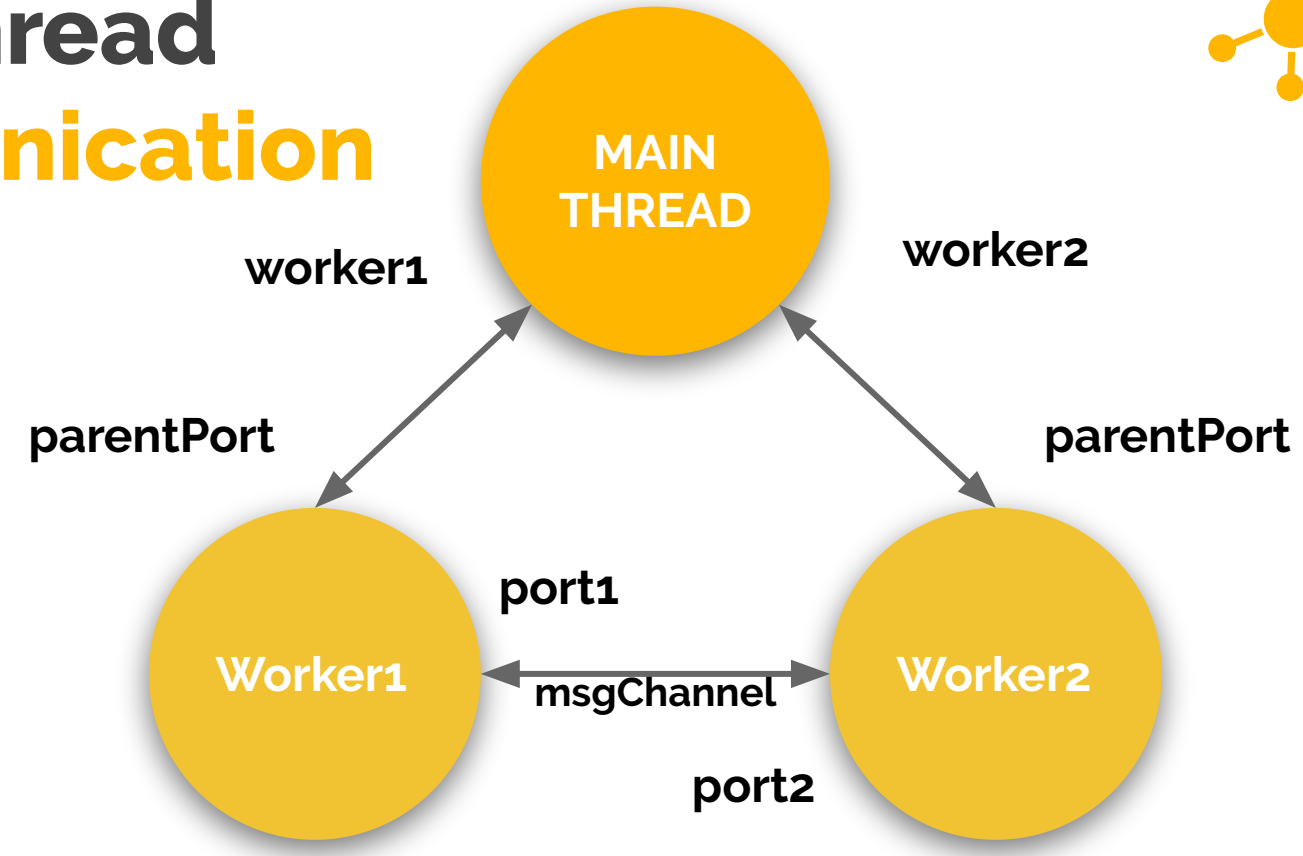


postMessage TransferList



- **Data is transferred**
- Unusable after transferring (on sending side) (guarantee no race conditions)
- **ArrayBuffer** and **MessagePort** objects
- Transferring MessagePort for **communication** between multiple **worker threads**

Multi thread communication



Main Thread

```
const { Worker, MessageChannel } = require('worker_threads')

const worker1 = new Worker('./worker.js')
const worker2 = new Worker('./worker.js')
const {port1, port2} = new MessageChannel()

worker1.on('message', message => console.log('Message from w1',
message))
worker2.on('message', message => console.log('Message from w2',
message))ostMessage({ port: port1, id: worker1.threadId },  [port1])
worker2.postMessage({ port: port2, id: worker2.threadId },  [port2])
```

Worker

```
const { parentPort, workerData } = require('worker_threads')

parentPort.postMessage('Hi parent')

parentPort.on('message', parentMsg => {
   parentMsg.port.postMessage(`Hi brother!, I am worker
${parentMsg.id}`)
   parentMsg.port.on('message', workerMsg => {
       console.log(
           `Worker ${parentMsg.id} got msg from other worker`, workerMsg
       )
   })

})
```

Output

```
Message from w1 Hi parent
Message from w2 Hi parent
Worker 1 got msg from other worker Hi brother!, I am worker 2
Worker 2 got msg from other worker Hi brother!, I am worker 1
```

5

Can share memory

SharedArrayBuffer

ArrayBuffer

- fixed-length raw binary data (like byte array)
- cannot directly manipulate the contents of an ArrayBuffer
- Use **typed arrays** to read/write buffers
  e.g **Int32Array, Float32Array, Uint8Array, etc**

SharedArrayBuffer

- Like **ArrayBuffer** but for **shared memory**

```
const worker = new Worker(__filename)

const sharedBuffer = new SharedArrayBuffer(1024)
const int32Array = new Int32Array(sharedBuffer)

worker.postMessage(int32Array)
```

6

Atomics

Available

safely accessing shared data

Atomics

- Shared memory can be **created and updated simultaneously** in workers or the main thread.
- Can take a while until the change is propagated to all contexts
- To synchronize, **atomics** operations are needed

Atomics

- **Atomics.wait()** / **notify()** - on array indexes
- **Atomics.store()** - stores a value on an array index
- **Atomics.load()** - returns a value on an array index
- **Atomics.add()** / **sub()** / **xor()** / **and()** / **or()**
  bitwise operations

```
if (isMainThread) {
  const worker = new Worker(__filename)
  const sharedBuffer = new SharedArrayBuffer(1024)
  const int32Array = new Int32Array(sharedBuffer) // all 0 by default
  worker.postMessage(int32Array)
  // sleeps if index 10 is 0 (until notify() or timeout)
  Atomics.wait(int32Array, 10, 0)
  console.log('Main Thread', int32Array[10]) // Main Thread 123
} else {
  parentPort.once('message', (int32Array) => {
      Atomics.store(int32Array, 10, 123)
      Atomics.notify(int32Array, 10)
  })
```

Recommendation

- Use **worker thread pools**
- Do not use them for parallelizing I/O
- Check Worker Threads official docs

Sudoku Solver

By Anna Henningsen (Node core maintainer)

https://github.com/addaleax/workers-sudoku

Input

```
curl -d
'[5,3,0,0,7,0,0,0,0,6,0,0,0,0,0,0,0,0,0,9,8,0,0,0,0,6,0,8,0,0,0,6,0,0,0,
3,4,0,0,8,0,3,0,0,1,7,0,0,0,2,0,0,0,6,0,6,0,0,0,0,2,8,0,0,0,0,4,1,9,0,0,
5,0,0,0,0,8,0,0,7,9]' http://localhost:3000/
```

Output

```
[5, 3, 2, 1, 7, 6, 9, 4, 8,
 6, 7, 4, 3, 9, 8, 5, 1, 2,
 1, 9, 8, 2, 4, 5, 3, 6, 7,
 8, 5, 9, 7, 6, 1, 4, 2, 3,
 4, 2, 6, 8, 5, 3, 7, 9, 1,
 7, 1, 3, 9, 2, 4, 8, 5, 6,
 9, 6, 1, 5, 3, 7, 2, 8, 4,
 2, 8, 7, 4, 1, 9, 6, 3, 5,
 3, 4, 5, 6, 8, 2, 1, 7, 9]
```

Worker

```
const { parentPort } = require('worker_threads')
const { solveSudoku } = require('./solve-sudoku.js')

parentPort.on('message', (sudokuData) => {
 const solution = solveSudoku(sudokuData)
 parentPort.postMessage(solution)
})
```

Server

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

const workerPool = [  // Start a pool of four workers
 new Worker('./worker.js'),
 new Worker('./worker.js'),
 new Worker('./worker.js'),
 new Worker('./worker.js'),
]

const waiting = []
```

Server

```
http.createServer((req, res) => {
. . . // get input data
   if (workerPool.length > 0) {
     handleRequest(res, sudokuData, workerPool.shift())
   } else {
     // Queue up requests when no worker is available.
     waiting.push((worker) => handleRequest(res, sudokuData, worker))
   }
 })
}).listen(3000)
```

Server

```
function handleRequest(res, sudokuData, worker) {
 worker.postMessage(sudokuData)
 worker.once('message', (solutionData) => {
   res.end(JSON.stringify([...solutionData]))

   if (waiting.length > 0){
     waiting.shift()(worker)
   }
   else {
     workerPool.push(worker) // Add the worker to pool
   }
 })
}
```

Thanks!

Any questions?

@sebcurland

@sebcurland