Worker threads

> *Stability: 2 - Stable*

The `worker_threads` module enables the use of threads that execute JavaScript in parallel. To access it:

```
const worker = require('worker_threads');
```

Workers (threads) are useful for performing CPU-intensive JavaScript operations. They do not help much with I/O-intensive work. The Node.js built-in asynchronous I/O operations are more efficient than Workers can be.

Unlike `child_process` or `cluster`, `worker_threads` can share memory. They do so by transferring `ArrayBuffer` instances or sharing `SharedArrayBuffer` instances.

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

if (isMainThread) {
  module.exports = function parseJSAsync(script) {
    return new Promise((resolve, reject) => {
      const worker = new Worker(__filename, {
        workerData: script
      });
      worker.on('message', resolve);
      worker.on('error', reject);
      worker.on('exit', (code) => {
        if (code !== 0)
          reject(new Error(`Worker stopped with exit code ${code}`));
      });
    });
  };
} else {
  const { parse } = require('some-js-parsing-library');
  const script = workerData;
  parentPort.postMessage(parse(script));
}
```

The above example spawns a Worker thread for each `parseJSAsync()` call. In practice, use a pool of Workers for these kinds of tasks. Otherwise, the overhead of creating Workers would likely exceed their benefit.

When implementing a worker pool, use the `AsyncResource` API to inform diagnostic tools (e.g. to provide asynchronous stack traces) about the correlation between tasks and their outcomes. See "Using `AsyncResource` for a `Worker` thread pool" in the `async_hooks` documentation for an example implementation.

Worker threads inherit non-process-specific options by default. Refer to `Worker constructor options` to know how to customize worker thread options, specifically `argv` and `execArgv` options.

`worker.getEnvironmentData(key)`

- `key` {any} Any arbitrary, cloneable JavaScript value that can be used as a {Map} key.
- Returns: {any}

Within a worker thread, `worker.getEnvironmentData()` returns a clone of data passed to the spawning thread's `worker.setEnvironmentData()`. Every new `Worker` receives its own copy of the environment data automatically.

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

if (isMainThread) {
  setEnvironmentData('Hello', 'World!');
  const worker = new Worker(__filename);
} else {
  console.log(getEnvironmentData('Hello'));  // Prints 'World!'.
}
```

`worker.isMainThread`

- {boolean}

Is `true` if this code is not running inside of a `Worker` thread.

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

if (isMainThread) {
  // This re-loads the current file inside a Worker instance.
  new Worker(__filename);
} else {
  console.log('Inside Worker!');
  console.log(isMainThread);  // Prints 'false'.
}
```

`worker.markAsUntransferable(object)`

Mark an object as not transferable. If `object` occurs in the transfer list of a `port.postMessage()` call, it is ignored.

In particular, this makes sense for objects that can be cloned, rather than transferred, and which are used by other objects on the sending side. For example, Node.js marks the `ArrayBuffer`s it uses for its `Buffer` pool with this.

This operation cannot be undone.

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

const pooledBuffer = new ArrayBuffer(8);
```

```
const typedArray1 = new Uint8Array(pooledBuffer);
const typedArray2 = new Float64Array(pooledBuffer);

markAsUntransferable(pooledBuffer);

const { port1 } = new MessageChannel();
port1.postMessage(typedArray1, [ typedArray1.buffer ]);

// The following line prints the contents of typedArray1 -- it still owns
// its memory and has been cloned, not transferred. Without
// `markAsUntransferable()`, this would print an empty Uint8Array.
// typedArray2 is intact as well.
console.log(typedArray1);
console.log(typedArray2);
```

There is no equivalent to this API in browsers.

`worker.moveMessagePortToContext(port, contextifiedSandbox)`

- `port` {MessagePort} The message port to transfer.
- `contextifiedSandbox` {Object} A contextified object as returned by the `vm.createContext()` method.
- Returns: {MessagePort}

Transfer a `MessagePort` to a different `vm` Context. The original `port` object is rendered unusable, and the returned `MessagePort` instance takes its place.

The returned `MessagePort` is an object in the target context and inherits from its global `Object` class. Objects passed to the `port.onmessage()` listener are also created in the target context and inherit from its global `Object` class.

However, the created `MessagePort` no longer inherits from `EventTarget`, and only `port.onmessage()` can be used to receive events using it.

`worker.parentPort`

- {null|MessagePort}

If this thread is a `Worker`, this is a `MessagePort` allowing communication with the parent thread. Messages sent using `parentPort.postMessage()` are available in the parent thread using `worker.on('message')`, and messages sent from the parent thread using `worker.postMessage()` are available in this thread using `parentPort.on('message')`.

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

if (isMainThread) {
  const worker = new Worker(__filename);
```

```
  worker.once('message', (message) => {
    console.log(message);  // Prints 'Hello, world!'.
  });
  worker.postMessage('Hello, world!');
} else {
  // When a message from the parent thread is received, send it back:
  parentPort.once('message', (message) => {
    parentPort.postMessage(message);
  });
}
```

`worker.receiveMessageOnPort(port)`

- `port` {MessagePort|BroadcastChannel}
- Returns: {Object|undefined}

Receive a single message from a given `MessagePort`. If no message is available, `undefined` is returned, otherwise an object with a single `message` property that contains the message payload, corresponding to the oldest message in the `MessagePort`'s queue.

```
const { MessageChannel, receiveMessageOnPort } = require('worker_threads');
const { port1, port2 } = new MessageChannel();
port1.postMessage({ hello: 'world' });

console.log(receiveMessageOnPort(port2));
// Prints: { message: { hello: 'world' } }
console.log(receiveMessageOnPort(port2));
// Prints: undefined
```

When this function is used, no `'message'` event is emitted and the `onmessage` listener is not invoked.

`worker.resourceLimits`

- {Object}
  - `maxYoungGenerationSizeMb` {number}
  - `maxOldGenerationSizeMb` {number}
  - `codeRangeSizeMb` {number}
  - `stackSizeMb` {number}

Provides the set of JS engine resource constraints inside this Worker thread. If the `resourceLimits` option was passed to the `Worker` constructor, this matches its values.

If this is used in the main thread, its value is an empty object.

`worker.SHARE_ENV`

- {symbol}

A special value that can be passed as the `env` option of the `Worker` constructor, to indicate that the current thread and the Worker thread should share read and write access to the same set of environment variables.

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

`worker.setEnvironmentData(key[, value])`

- `key` {any} Any arbitrary, cloneable JavaScript value that can be used as a {Map} key.
- `value` {any} Any arbitrary, cloneable JavaScript value that will be cloned and passed automatically to all new `Worker` instances. If `value` is passed as `undefined`, any previously set value for the `key` will be deleted.

The `worker.setEnvironmentData()` API sets the content of `worker.getEnvironmentData()` in the current thread and all new `Worker` instances spawned from the current context.

`worker.threadId`

- {integer}

An integer identifier for the current thread. On the corresponding worker object (if there is any), it is available as `worker.threadId`. This value is unique for each `Worker` instance inside a single process.

`worker.workerData`

An arbitrary JavaScript value that contains a clone of the data passed to this thread's `Worker` constructor.

The data is cloned as if using `postMessage()`, according to the HTML structured clone algorithm.

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

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

Class: `BroadcastChannel extends EventTarget`

Instances of `BroadcastChannel` allow asynchronous one-to-many communication with all other `BroadcastChannel` instances bound to the same channel name.

```
'use strict';

const {
  isMainThread,
```

```
  BroadcastChannel,
  Worker
} = require('worker_threads');

const bc = new BroadcastChannel('hello');

if (isMainThread) {
  let c = 0;
  bc.onmessage = (event) => {
    console.log(event.data);
    if (++c === 10) bc.close();
  };
  for (let n = 0; n < 10; n++)
    new Worker(__filename);
} else {
  bc.postMessage('hello from every worker');
  bc.close();
}
```

`new BroadcastChannel(name)`

- `name` {any} The name of the channel to connect to. Any JavaScript value that can be converted to a string using `` `${name}` `` is permitted.

`broadcastChannel.close()`

Closes the `BroadcastChannel` connection.

`broadcastChannel.onmessage`

- Type: {Function} Invoked with a single `MessageEvent` argument when a message is received.

`broadcastChannel.onmessageerror`

- Type: {Function} Invoked with a received message cannot be deserialized.

`broadcastChannel.postMessage(message)`

- `message` {any} Any cloneable JavaScript value.

`broadcastChannel.ref()`

Opposite of `unref()`. Calling `ref()` on a previously `unref()`ed BroadcastChannel does *not* let the program exit if it's the only active handle left (the default behavior). If the port is `ref()`ed, calling `ref()` again has no effect.

`broadcastChannel.unref()`

Calling `unref()` on a BroadcastChannel allows the thread to exit if this is the only active handle in the event system. If the BroadcastChannel is already `unref()`ed calling `unref()` again has no effect.

Class: `MessageChannel`

Instances of the `worker.MessageChannel` class represent an asynchronous, two-way communications channel. The `MessageChannel` has no methods of its own. `new MessageChannel()` yields an object with `port1` and `port2` properties, which refer to linked `MessagePort` instances.

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

const { port1, port2 } = new MessageChannel();
port1.on('message', (message) => console.log('received', message));
port2.postMessage({ foo: 'bar' });
// Prints: received { foo: 'bar' } from the `port1.on('message')` listener
```

Class: `MessagePort`

- Extends: {EventTarget}

Instances of the `worker.MessagePort` class represent one end of an asynchronous, two-way communications channel. It can be used to transfer structured data, memory regions and other `MessagePort` s between different `Worker` s.

This implementation matches browser `MessagePort` s.

Event: `'close'`

The `'close'` event is emitted once either side of the channel has been disconnected.

```
const { MessageChannel } = require('worker_threads');
const { port1, port2 } = new MessageChannel();

// Prints:
//   foobar
//   closed!
port2.on('message', (message) => console.log(message));
port2.on('close', () => console.log('closed!'));

port1.postMessage('foobar');
port1.close();
```

Event: `'message'`

- `value` {any} The transmitted value

The `'message'` event is emitted for any incoming message, containing the cloned input of `port.postMessage()` .

Listeners on this event receive a clone of the `value` parameter as passed to `postMessage()` and no further arguments.

Event: `'messageerror'`

- `error` {Error} An Error object

The `'messageerror'` event is emitted when deserializing a message failed.

Currently, this event is emitted when there is an error occurring while instantiating the posted JS object on the receiving end. Such situations are rare, but can happen, for instance, when certain Node.js API objects are received in a `vm.Context` (where Node.js APIs are currently unavailable).

`port.close()`

Disables further sending of messages on either side of the connection. This method can be called when no further communication will happen over this `MessagePort`.

The `'close'` event is emitted on both `MessagePort` instances that are part of the channel.

`port.postMessage(value[, transferList])`

- `value` {any}
- `transferList` {Object[]}

Sends a JavaScript value to the receiving side of this channel. `value` is transferred in a way which is compatible with the HTML structured clone algorithm.

In particular, the significant differences to `JSON` are:

- `value` may contain circular references.
- `value` may contain instances of builtin JS types such as `RegExp`s, `BigInt`s, `Map`s, `Set`s, etc.
- `value` may contain typed arrays, both using `ArrayBuffer`s and `SharedArrayBuffer`s.
- `value` may contain `WebAssembly.Module` instances.
- `value` may not contain native (C++-backed) objects other than:
  - {CryptoKey}s,
  - {FileHandle}s,
  - {Histogram}s,
  - {KeyObject}s,
  - {MessagePort}s,
  - {net.BlockList}s,
  - {net.SocketAddress}es,
  - {X509Certificate}s.

```
const { MessageChannel } = require('worker_threads');
const { port1, port2 } = new MessageChannel();

port1.on('message', (message) => console.log(message));

const circularData = {};
circularData.foo = circularData;
// Prints: { foo: [Circular] }
port2.postMessage(circularData);
```

`transferList` may be a list of `ArrayBuffer`, `MessagePort` and `FileHandle` objects. After transferring, they are not usable on the sending side of the channel anymore (even if they are not contained in `value`). Unlike with child processes, transferring handles such as network sockets is currently not supported.

If `value` contains `SharedArrayBuffer` instances, those are accessible from either thread. They cannot be listed in `transferList`.

`value` may still contain `ArrayBuffer` instances that are not in `transferList`; in that case, the underlying memory is copied rather than moved.

```
const { MessageChannel } = require('worker_threads');
const { port1, port2 } = new MessageChannel();

port1.on('message', (message) => console.log(message));

const uint8Array = new Uint8Array([ 1, 2, 3, 4 ]);
// This posts a copy of `uint8Array`:
port2.postMessage(uint8Array);
// This does not copy data, but renders `uint8Array` unusable:
port2.postMessage(uint8Array, [ uint8Array.buffer ]);

// The memory for the `sharedUint8Array` is accessible from both the
// original and the copy received by `.on('message')`:
const sharedUint8Array = new Uint8Array(new SharedArrayBuffer(4));
port2.postMessage(sharedUint8Array);

// This transfers a freshly created message port to the receiver.
// This can be used, for example, to create communication channels between
// multiple `Worker` threads that are children of the same parent thread.
const otherChannel = new MessageChannel();
port2.postMessage({ port: otherChannel.port1 }, [ otherChannel.port1 ]);
```

The message object is cloned immediately, and can be modified after posting without having side effects.

For more information on the serialization and deserialization mechanisms behind this API, see the serialization API of the `v8` module.

Considerations when transferring TypedArrays and Buffers

All `TypedArray` and `Buffer` instances are views over an underlying `ArrayBuffer`. That is, it is the `ArrayBuffer` that actually stores the raw data while the `TypedArray` and `Buffer` objects provide a way of viewing and manipulating the data. It is possible and common for multiple views to be created over the same `ArrayBuffer` instance. Great care must be taken when using a transfer list to transfer an `ArrayBuffer` as doing so causes all `TypedArray` and `Buffer` instances that share that same `ArrayBuffer` to become unusable.

```
const ab = new ArrayBuffer(10);

const u1 = new Uint8Array(ab);
const u2 = new Uint16Array(ab);

console.log(u2.length);  // prints 5

port.postMessage(u1, [u1.buffer]);

console.log(u2.length);  // prints 0
```

For `Buffer` instances, specifically, whether the underlying `ArrayBuffer` can be transferred or cloned depends entirely on how instances were created, which often cannot be reliably determined.

An `ArrayBuffer` can be marked with `markAsUntransferable()` to indicate that it should always be cloned and never transferred.

Depending on how a `Buffer` instance was created, it may or may not own its underlying `ArrayBuffer`. An `ArrayBuffer` must not be transferred unless it is known that the `Buffer` instance owns it. In particular, for `Buffer`s created from the internal `Buffer` pool (using, for instance `Buffer.from()` or `Buffer.allocUnsafe()`), transferring them is not possible and they are always cloned, which sends a copy of the entire `Buffer` pool. This behavior may come with unintended higher memory usage and possible security concerns.

See `Buffer.allocUnsafe()` for more details on `Buffer` pooling.

The `ArrayBuffer`s for `Buffer` instances created using `Buffer.alloc()` or `Buffer.allocUnsafeSlow()` can always be transferred but doing so renders all other existing views of those `ArrayBuffer`s unusable.

Considerations when cloning objects with prototypes, classes, and accessors

Because object cloning uses the HTML structured clone algorithm, non-enumerable properties, property accessors, and object prototypes are not preserved. In particular, `Buffer` objects will be read as plain `Uint8Array`s on the receiving side, and instances of JavaScript classes will be cloned as plain JavaScript objects.

```
const b = Symbol('b');

class Foo {
  #a = 1;
  constructor() {
    this[b] = 2;
    this.c = 3;
  }

  get d() { return 4; }
}

const { port1, port2 } = new MessageChannel();

port1.onmessage = ({ data }) => console.log(data);

port2.postMessage(new Foo());

// Prints: { c: 3 }
```

This limitation extends to many built-in objects, such as the global `URL` object:

```
const { port1, port2 } = new MessageChannel();

port1.onmessage = ({ data }) => console.log(data);
```

```
port2.postMessage(new URL('https://example.org'));

// Prints: { }
```

`port.ref()`

Opposite of `unref()`. Calling `ref()` on a previously `unref()`ed port does *not* let the program exit if it's the only active handle left (the default behavior). If the port is `ref()`ed, calling `ref()` again has no effect.

If listeners are attached or removed using `.on('message')`, the port is `ref()`ed and `unref()`ed automatically depending on whether listeners for the event exist.

`port.start()`

Starts receiving messages on this `MessagePort`. When using this port as an event emitter, this is called automatically once `'message'` listeners are attached.

This method exists for parity with the Web `MessagePort` API. In Node.js, it is only useful for ignoring messages when no event listener is present. Node.js also diverges in its handling of `.onmessage`. Setting it automatically calls `.start()`, but unsetting it lets messages queue up until a new handler is set or the port is discarded.

`port.unref()`

Calling `unref()` on a port allows the thread to exit if this is the only active handle in the event system. If the port is already `unref()`ed calling `unref()` again has no effect.

If listeners are attached or removed using `.on('message')`, the port is `ref()`ed and `unref()`ed automatically depending on whether listeners for the event exist.

Class: `Worker`

- Extends: {EventEmitter}

The `Worker` class represents an independent JavaScript execution thread. Most Node.js APIs are available inside of it.

Notable differences inside a Worker environment are:

- The `process.stdin`, `process.stdout` and `process.stderr` may be redirected by the parent thread.
- The `require('worker_threads').isMainThread` property is set to `false`.
- The `require('worker_threads').parentPort` message port is available.
- `process.exit()` does not stop the whole program, just the single thread, and `process.abort()` is not available.
- `process.chdir()` and `process` methods that set group or user ids are not available.
- `process.env` is a copy of the parent thread's environment variables, unless otherwise specified. Changes to one copy are not visible in other threads, and are not visible to native add-ons (unless `worker.SHARE_ENV` is passed as the `env` option to the `Worker` constructor).
- `process.title` cannot be modified.
- Signals are not delivered through `process.on('...')`.
- Execution may stop at any point as a result of `worker.terminate()` being invoked.
- IPC channels from parent processes are not accessible.

- The `trace_events` module is not supported.
- Native add-ons can only be loaded from multiple threads if they fulfill certain conditions.

Creating `Worker` instances inside of other `Worker`s is possible.

Like Web Workers and the `cluster` module, two-way communication can be achieved through inter-thread message passing. Internally, a `Worker` has a built-in pair of `MessagePort`s that are already associated with each other when the `Worker` is created. While the `MessagePort` object on the parent side is not directly exposed, its functionalities are exposed through `worker.postMessage()` and the `worker.on('message')` event on the `Worker` object for the parent thread.

To create custom messaging channels (which is encouraged over using the default global channel because it facilitates separation of concerns), users can create a `MessageChannel` object on either thread and pass one of the `MessagePort`s on that `MessageChannel` to the other thread through a pre-existing channel, such as the global one.

See `port.postMessage()` for more information on how messages are passed, and what kind of JavaScript values can be successfully transported through the thread barrier.

```
const assert = require('assert');
const {
  Worker, MessageChannel, MessagePort, isMainThread, parentPort
} = require('worker_threads');
if (isMainThread) {
  const worker = new Worker(__filename);
  const subChannel = new MessageChannel();
  worker.postMessage({ hereIsYourPort: subChannel.port1 }, [subChannel.port1]);
  subChannel.port2.on('message', (value) => {
    console.log('received:', value);
  });
} else {
  parentPort.once('message', (value) => {
    assert(value.hereIsYourPort instanceof MessagePort);
    value.hereIsYourPort.postMessage('the worker is sending this');
    value.hereIsYourPort.close();
  });
}
```

`new Worker(filename[, options])`

- `filename` {string|URL} The path to the Worker's main script or module. Must be either an absolute path or a relative path (i.e. relative to the current working directory) starting with `./` or `../`, or a WHATWG `URL` object using `file:` or `data:` protocol. When using a `data:` URL, the data is interpreted based on MIME type using the ECMAScript module loader. If `options.eval` is `true`, this is a string containing JavaScript code rather than a path.
- `options` {Object}
  - `argv` {any[]} List of arguments which would be stringified and appended to `process.argv` in the worker. This is mostly similar to the `workerData` but the values are available on the global `process.argv` as if they were passed as CLI options to the script.
  - `env` {Object} If set, specifies the initial value of `process.env` inside the Worker thread. As a special value, `worker.SHARE_ENV` may be used to specify that the parent thread and the child

thread should share their environment variables; in that case, changes to one thread's `process.env` object affect the other thread as well. **Default:** `process.env`.

- `eval` {boolean} If `true` and the first argument is a `string`, interpret the first argument to the constructor as a script that is executed once the worker is online.
- `execArgv` {string[]} List of node CLI options passed to the worker. V8 options (such as `--max-old-space-size`) and options that affect the process (such as `--title`) are not supported. If set, this is provided as `process.execArgv` inside the worker. By default, options are inherited from the parent thread.
- `stdin` {boolean} If this is set to `true`, then `worker.stdin` provides a writable stream whose contents appear as `process.stdin` inside the Worker. By default, no data is provided.
- `stdout` {boolean} If this is set to `true`, then `worker.stdout` is not automatically piped through to `process.stdout` in the parent.
- `stderr` {boolean} If this is set to `true`, then `worker.stderr` is not automatically piped through to `process.stderr` in the parent.
- `workerData` {any} Any JavaScript value that is cloned and made available as `require('worker_threads').workerData`. The cloning occurs as described in the HTML structured clone algorithm, and an error is thrown if the object cannot be cloned (e.g. because it contains `function`s).
- `trackUnmanagedFds` {boolean} If this is set to `true`, then the Worker tracks raw file descriptors managed through `fs.open()` and `fs.close()`, and closes them when the Worker exits, similar to other resources like network sockets or file descriptors managed through the `FileHandle` API. This option is automatically inherited by all nested `Worker`s. **Default:** `true`.
- `transferList` {Object[]} If one or more `MessagePort`-like objects are passed in `workerData`, a `transferList` is required for those items or `ERR_MISSING_MESSAGE_PORT_IN_TRANSFER_LIST` is thrown. See `port.postMessage()` for more information.
- `resourceLimits` {Object} An optional set of resource limits for the new JS engine instance. Reaching these limits leads to termination of the `Worker` instance. These limits only affect the JS engine, and no external data, including no `ArrayBuffer`s. Even if these limits are set, the process may still abort if it encounters a global out-of-memory situation.
  - `maxOldGenerationSizeMb` {number} The maximum size of the main heap in MB.
  - `maxYoungGenerationSizeMb` {number} The maximum size of a heap space for recently created objects.
  - `codeRangeSizeMb` {number} The size of a pre-allocated memory range used for generated code.
  - `stackSizeMb` {number} The default maximum stack size for the thread. Small values may lead to unusable Worker instances. **Default:** `4`.

Event: `'error'`

- `err` {Error}

The `'error'` event is emitted if the worker thread throws an uncaught exception. In that case, the worker is terminated.

Event: `'exit'`

- `exitCode` {integer}

The `'exit'` event is emitted once the worker has stopped. If the worker exited by calling `process.exit()`, the `exitCode` parameter is the passed exit code. If the worker was terminated, the `exitCode` parameter is `1`.

This is the final event emitted by any `Worker` instance.

Event: `'message'`

- `value` {any} The transmitted value

The `'message'` event is emitted when the worker thread has invoked `require('worker_threads').parentPort.postMessage()`. See the `port.on('message')` event for more details.

All messages sent from the worker thread are emitted before the `'exit'` event is emitted on the `Worker` object.

Event: `'messageerror'`

- `error` {Error} An Error object

The `'messageerror'` event is emitted when deserializing a message failed.

Event: `'online'`

The `'online'` event is emitted when the worker thread has started executing JavaScript code.

`worker.getHeapSnapshot()`

- Returns: {Promise} A promise for a Readable Stream containing a V8 heap snapshot

Returns a readable stream for a V8 snapshot of the current state of the Worker. See `v8.getHeapSnapshot()` for more details.

If the Worker thread is no longer running, which may occur before the `'exit'` event is emitted, the returned `Promise` is rejected immediately with an `ERR_WORKER_NOT_RUNNING` error.

`worker.performance`

An object that can be used to query performance information from a worker instance. Similar to `perf_hooks.performance`.

`performance.eventLoopUtilization([utilization1[, utilization2]])`

- `utilization1` {Object} The result of a previous call to `eventLoopUtilization()`.
- `utilization2` {Object} The result of a previous call to `eventLoopUtilization()` prior to `utilization1`.
- Returns {Object}
  - `idle` {number}
  - `active` {number}
  - `utilization` {number}

The same call as `perf_hooks` `eventLoopUtilization()`, except the values of the worker instance are returned.

One difference is that, unlike the main thread, bootstrapping within a worker is done within the event loop. So the event loop utilization is immediately available once the worker's script begins execution.

An `idle` time that does not increase does not indicate that the worker is stuck in bootstrap. The following examples shows how the worker's entire lifetime never accumulates any `idle` time, but is still be able to process messages.

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

if (isMainThread) {
  const worker = new Worker(__filename);
  setInterval(() => {
    worker.postMessage('hi');
    console.log(worker.performance.eventLoopUtilization());
  }, 100).unref();
  return;
}

parentPort.on('message', () => console.log('msg')).unref();
(function r(n) {
  if (--n < 0) return;
  const t = Date.now();
  while (Date.now() - t < 300);
  setImmediate(r, n);
})(10);
```

The event loop utilization of a worker is available only after the `'online'` event emitted, and if called before this, or after the `'exit'` event, then all properties have the value of `0`.

`worker.postMessage(value[, transferList])`

- `value` {any}
- `transferList` {Object[]}

Send a message to the worker that is received via `require('worker_threads').parentPort.on('message')`. See `port.postMessage()` for more details.

`worker.ref()`

Opposite of `unref()`, calling `ref()` on a previously `unref()`ed worker does *not* let the program exit if it's the only active handle left (the default behavior). If the worker is `ref()`ed, calling `ref()` again has no effect.

`worker.resourceLimits`

- {Object}
  - `maxYoungGenerationSizeMb` {number}
  - `maxOldGenerationSizeMb` {number}
  - `codeRangeSizeMb` {number}
  - `stackSizeMb` {number}

Provides the set of JS engine resource constraints for this Worker thread. If the `resourceLimits` option was passed to the `Worker` constructor, this matches its values.

If the worker has stopped, the return value is an empty object.

`worker.stderr`

- {stream.Readable}

This is a readable stream which contains data written to `process.stderr` inside the worker thread. If `stderr: true` was not passed to the `Worker` constructor, then data is piped to the parent thread's `process.stderr` stream.

`worker.stdin`

- {null|stream.Writable}

If `stdin: true` was passed to the `Worker` constructor, this is a writable stream. The data written to this stream will be made available in the worker thread as `process.stdin`.

`worker.stdout`

- {stream.Readable}

This is a readable stream which contains data written to `process.stdout` inside the worker thread. If `stdout: true` was not passed to the `Worker` constructor, then data is piped to the parent thread's `process.stdout` stream.

`worker.terminate()`

- Returns: {Promise}

Stop all JavaScript execution in the worker thread as soon as possible. Returns a Promise for the exit code that is fulfilled when the `'exit'` event is emitted.

`worker.threadId`

- {integer}

An integer identifier for the referenced thread. Inside the worker thread, it is available as `require('worker_threads').threadId`. This value is unique for each `Worker` instance inside a single process.

`worker.unref()`

Calling `unref()` on a worker allows the thread to exit if this is the only active handle in the event system. If the worker is already `unref()`ed calling `unref()` again has no effect.

Notes

Synchronous blocking of stdio

`Worker`s utilize message passing via {MessagePort} to implement interactions with `stdio`. This means that `stdio` output originating from a `Worker` can get blocked by synchronous code on the receiving end that is blocking the Node.js event loop.

```
import {
  Worker,
  isMainThread,
} from 'worker_threads';
```

```
if (isMainThread) {
  new Worker(new URL(import.meta.url));
  for (let n = 0; n < 1e10; n++) {
    // Looping to simulate work.
  }
} else {
  // This output will be blocked by the for loop in the main thread.
  console.log('foo');
}
```

```
'use strict';

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

if (isMainThread) {
  new Worker(__filename);
  for (let n = 0; n < 1e10; n++) {
    // Looping to simulate work.
  }
} else {
  // This output will be blocked by the for loop in the main thread.
  console.log('foo');
}
```

Launching worker threads from preload scripts

Take care when launching worker threads from preload scripts (scripts loaded and run using the `-r` command line flag). Unless the `execArgv` option is explicitly set, new Worker threads automatically inherit the command line flags from the running process and will preload the same preload scripts as the main thread. If the preload script unconditionally launches a worker thread, every thread spawned will spawn another until the application crashes.