/\*\*

\* The `worker\_threads` module enables the use of threads that execute JavaScript

\* in parallel. To access it:

\*

\* ```js

\* 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.

\*

\* ```js

\* 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 `parse()` call. In actual

\* 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.

\* @see [source](https://github.com/nodejs/node/blob/v17.0.0/lib/worker\_threads.js)

\*/

declare module 'worker\_threads' {

import { Blob } from 'node:buffer';

import { Context } from 'node:vm';

import { EventEmitter } from 'node:events';

import { EventLoopUtilityFunction } from 'node:perf\_hooks';

import { FileHandle } from 'node:fs/promises';

import { Readable, Writable } from 'node:stream';

import { URL } from 'node:url';

import { X509Certificate } from 'node:crypto';

const isMainThread: boolean;

const parentPort: null | MessagePort;

const resourceLimits: ResourceLimits;

const SHARE\_ENV: unique symbol;

const threadId: number;

const workerData: any;

/\*\*

\* 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.

\*

\* ```js

\* 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

\* ```

\* @since v10.5.0

\*/

class MessageChannel {

readonly port1: MessagePort;

readonly port2: MessagePort;

}

interface WorkerPerformance {

eventLoopUtilization: EventLoopUtilityFunction;

}

type TransferListItem = ArrayBuffer | MessagePort | FileHandle | X509Certificate | Blob;

/\*\*

\* 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`](https://developer.mozilla.org/en-US/docs/Web/API/MessagePort) s.

\* @since v10.5.0

\*/

class MessagePort extends EventEmitter {

/\*\*

\* 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.

\* @since v10.5.0

\*/

close(): void;

/\*\*

\* 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](https://developer.mozilla.org/en-US/docs/Web/API/Web\_Workers\_API/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`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/WebAssembly/Module) instances.

\* \* `value` may not contain native (C++-backed) objects other than:

\*

\* ```js

\* 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`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/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`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/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.

\*

\* ```js

\* 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`.

\* @since v10.5.0

\*/

postMessage(value: any, transferList?: ReadonlyArray<TransferListItem>): void;

/\*\*

\* 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.

\* @since v10.5.0

\*/

ref(): void;

/\*\*

\* 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.

\* @since v10.5.0

\*/

unref(): void;

/\*\*

\* 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.

\* @since v10.5.0

\*/

start(): void;

addListener(event: 'close', listener: () => void): this;

addListener(event: 'message', listener: (value: any) => void): this;

addListener(event: 'messageerror', listener: (error: Error) => void): this;

addListener(event: string | symbol, listener: (...args: any[]) => void): this;

emit(event: 'close'): boolean;

emit(event: 'message', value: any): boolean;

emit(event: 'messageerror', error: Error): boolean;

emit(event: string | symbol, ...args: any[]): boolean;

on(event: 'close', listener: () => void): this;

on(event: 'message', listener: (value: any) => void): this;

on(event: 'messageerror', listener: (error: Error) => void): this;

on(event: string | symbol, listener: (...args: any[]) => void): this;

once(event: 'close', listener: () => void): this;

once(event: 'message', listener: (value: any) => void): this;

once(event: 'messageerror', listener: (error: Error) => void): this;

once(event: string | symbol, listener: (...args: any[]) => void): this;

prependListener(event: 'close', listener: () => void): this;

prependListener(event: 'message', listener: (value: any) => void): this;

prependListener(event: 'messageerror', listener: (error: Error) => void): this;

prependListener(event: string | symbol, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'close', listener: () => void): this;

prependOnceListener(event: 'message', listener: (value: any) => void): this;

prependOnceListener(event: 'messageerror', listener: (error: Error) => void): this;

prependOnceListener(event: string | symbol, listener: (...args: any[]) => void): this;

removeListener(event: 'close', listener: () => void): this;

removeListener(event: 'message', listener: (value: any) => void): this;

removeListener(event: 'messageerror', listener: (error: Error) => void): this;

removeListener(event: string | symbol, listener: (...args: any[]) => void): this;

off(event: 'close', listener: () => void): this;

off(event: 'message', listener: (value: any) => void): this;

off(event: 'messageerror', listener: (error: Error) => void): this;

off(event: string | symbol, listener: (...args: any[]) => void): this;

}

interface WorkerOptions {

/\*\*

\* 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 will be available on the global `process.argv` as if they

\* were passed as CLI options to the script.

\*/

argv?: any[] | undefined;

env?: NodeJS.Dict<string> | typeof SHARE\_ENV | undefined;

eval?: boolean | undefined;

workerData?: any;

stdin?: boolean | undefined;

stdout?: boolean | undefined;

stderr?: boolean | undefined;

execArgv?: string[] | undefined;

resourceLimits?: ResourceLimits | undefined;

/\*\*

\* Additional data to send in the first worker message.

\*/

transferList?: TransferListItem[] | undefined;

/\*\*

\* @default true

\*/

trackUnmanagedFds?: boolean | undefined;

}

interface ResourceLimits {

/\*\*

\* The maximum size of a heap space for recently created objects.

\*/

maxYoungGenerationSizeMb?: number | undefined;

/\*\*

\* The maximum size of the main heap in MB.

\*/

maxOldGenerationSizeMb?: number | undefined;

/\*\*

\* The size of a pre-allocated memory range used for generated code.

\*/

codeRangeSizeMb?: number | undefined;

/\*\*

\* The default maximum stack size for the thread. Small values may lead to unusable Worker instances.

\* @default 4

\*/

stackSizeMb?: number | undefined;

}

/\*\*

\* 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](https://developer.mozilla.org/en-US/docs/Web/API/Web\_Workers\_API) 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.

\*

\* ```js

\* 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();

\* });

\* }

\* ```

\* @since v10.5.0

\*/

class Worker extends EventEmitter {

/\*\*

\* 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`.

\* @since v10.5.0

\*/

readonly stdin: Writable | null;

/\*\*

\* 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.

\* @since v10.5.0

\*/

readonly stdout: 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.

\* @since v10.5.0

\*/

readonly stderr: Readable;

/\*\*

\* 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.

\* @since v10.5.0

\*/

readonly threadId: 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.

\* @since v13.2.0, v12.16.0

\*/

readonly resourceLimits?: ResourceLimits | undefined;

/\*\*

\* An object that can be used to query performance information from a worker

\* instance. Similar to `perf\_hooks.performance`.

\* @since v15.1.0, v14.17.0, v12.22.0

\*/

readonly performance: WorkerPerformance;

/\*\*

\* @param filename 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: protocol. If options.eval is true, this is a string containing JavaScript code rather than a path.

\*/

constructor(filename: string | URL, options?: WorkerOptions);

/\*\*

\* Send a message to the worker that is received via `require('worker\_threads').parentPort.on('message')`.

\* See `port.postMessage()` for more details.

\* @since v10.5.0

\*/

postMessage(value: any, transferList?: ReadonlyArray<TransferListItem>): void;

/\*\*

\* 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.

\* @since v10.5.0

\*/

ref(): void;

/\*\*

\* 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.

\* @since v10.5.0

\*/

unref(): void;

/\*\*

\* 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.

\* @since v10.5.0

\*/

terminate(): Promise<number>;

/\*\*

\* 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.

\* @since v13.9.0, v12.17.0

\* @return A promise for a Readable Stream containing a V8 heap snapshot

\*/

getHeapSnapshot(): Promise<Readable>;

addListener(event: 'error', listener: (err: Error) => void): this;

addListener(event: 'exit', listener: (exitCode: number) => void): this;

addListener(event: 'message', listener: (value: any) => void): this;

addListener(event: 'messageerror', listener: (error: Error) => void): this;

addListener(event: 'online', listener: () => void): this;

addListener(event: string | symbol, listener: (...args: any[]) => void): this;

emit(event: 'error', err: Error): boolean;

emit(event: 'exit', exitCode: number): boolean;

emit(event: 'message', value: any): boolean;

emit(event: 'messageerror', error: Error): boolean;

emit(event: 'online'): boolean;

emit(event: string | symbol, ...args: any[]): boolean;

on(event: 'error', listener: (err: Error) => void): this;

on(event: 'exit', listener: (exitCode: number) => void): this;

on(event: 'message', listener: (value: any) => void): this;

on(event: 'messageerror', listener: (error: Error) => void): this;

on(event: 'online', listener: () => void): this;

on(event: string | symbol, listener: (...args: any[]) => void): this;

once(event: 'error', listener: (err: Error) => void): this;

once(event: 'exit', listener: (exitCode: number) => void): this;

once(event: 'message', listener: (value: any) => void): this;

once(event: 'messageerror', listener: (error: Error) => void): this;

once(event: 'online', listener: () => void): this;

once(event: string | symbol, listener: (...args: any[]) => void): this;

prependListener(event: 'error', listener: (err: Error) => void): this;

prependListener(event: 'exit', listener: (exitCode: number) => void): this;

prependListener(event: 'message', listener: (value: any) => void): this;

prependListener(event: 'messageerror', listener: (error: Error) => void): this;

prependListener(event: 'online', listener: () => void): this;

prependListener(event: string | symbol, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'error', listener: (err: Error) => void): this;

prependOnceListener(event: 'exit', listener: (exitCode: number) => void): this;

prependOnceListener(event: 'message', listener: (value: any) => void): this;

prependOnceListener(event: 'messageerror', listener: (error: Error) => void): this;

prependOnceListener(event: 'online', listener: () => void): this;

prependOnceListener(event: string | symbol, listener: (...args: any[]) => void): this;

removeListener(event: 'error', listener: (err: Error) => void): this;

removeListener(event: 'exit', listener: (exitCode: number) => void): this;

removeListener(event: 'message', listener: (value: any) => void): this;

removeListener(event: 'messageerror', listener: (error: Error) => void): this;

removeListener(event: 'online', listener: () => void): this;

removeListener(event: string | symbol, listener: (...args: any[]) => void): this;

off(event: 'error', listener: (err: Error) => void): this;

off(event: 'exit', listener: (exitCode: number) => void): this;

off(event: 'message', listener: (value: any) => void): this;

off(event: 'messageerror', listener: (error: Error) => void): this;

off(event: 'online', listener: () => void): this;

off(event: string | symbol, listener: (...args: any[]) => void): this;

}

interface BroadcastChannel extends NodeJS.RefCounted {}

/\*\*

\* Instances of `BroadcastChannel` allow asynchronous one-to-many communication

\* with all other `BroadcastChannel` instances bound to the same channel name.

\*

\* ```js

\* '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();

\* }

\* ```

\* @since v15.4.0

\* @experimental

\*/

class BroadcastChannel {

readonly name: string;

/\*\*

\* Invoked with a single \`MessageEvent\` argument when a message is received.

\* @since v15.4.0

\*/

onmessage: (message: unknown) => void;

/\*\*

\* Invoked with a received message cannot be deserialized.

\* @since v15.4.0

\*/

onmessageerror: (message: unknown) => void;

constructor(name: string);

/\*\*

\* Closes the `BroadcastChannel` connection.

\* @since v15.4.0

\*/

close(): void;

/\*\*

\* @since v15.4.0

\* @param message Any cloneable JavaScript value.

\*/

postMessage(message: unknown): void;

}

/\*\*

\* 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.

\*

\* ```js

\* 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.

\* @since v14.5.0, v12.19.0

\*/

function markAsUntransferable(object: object): void;

/\*\*

\* 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()`](https://developer.mozilla.org/en-US/docs/Web/API/MessagePort/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`](https://developer.mozilla.org/en-US/docs/Web/API/EventTarget), and only

\* [`port.onmessage()`](https://developer.mozilla.org/en-US/docs/Web/API/MessagePort/onmessage) can be used to receive

\* events using it.

\* @since v11.13.0

\* @param port The message port to transfer.

\* @param contextifiedSandbox A `contextified` object as returned by the `vm.createContext()` method.

\*/

function moveMessagePortToContext(port: MessagePort, contextifiedSandbox: Context): MessagePort;

/\*\*

\* 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.

\*

\* ```js

\* 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.

\* @since v12.3.0

\*/

function receiveMessageOnPort(port: MessagePort):

| {

message: any;

}

| undefined;

type Serializable = string | object | number | boolean | bigint;

/\*\*

\* 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.

\*

\* ```js

\* 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!'.

\* }

\* ```

\* @since v15.12.0, v14.18.0

\* @experimental

\* @param key Any arbitrary, cloneable JavaScript value that can be used as a {Map} key.

\*/

function getEnvironmentData(key: Serializable): Serializable;

/\*\*

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

\* @since v15.12.0, v14.18.0

\* @experimental

\* @param key Any arbitrary, cloneable JavaScript value that can be used as a {Map} key.

\* @param value 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.

\*/

function setEnvironmentData(key: Serializable, value: Serializable): void;

}

declare module 'node:worker\_threads' {

export \* from 'worker\_threads';

}