Process

The `process` object provides information about, and control over, the current Node.js process.

```
import process from 'process';
```

```
const process = require('process');
```

Process events

The `process` object is an instance of `EventEmitter`.

Event: `'beforeExit'`

The `'beforeExit'` event is emitted when Node.js empties its event loop and has no additional work to schedule. Normally, the Node.js process will exit when there is no work scheduled, but a listener registered on the `'beforeExit'` event can make asynchronous calls, and thereby cause the Node.js process to continue.

The listener callback function is invoked with the value of `process.exitCode` passed as the only argument.

The `'beforeExit'` event is *not* emitted for conditions causing explicit termination, such as calling `process.exit()` or uncaught exceptions.

The `'beforeExit'` should *not* be used as an alternative to the `'exit'` event unless the intention is to schedule additional work.

```
import process from 'process';

process.on('beforeExit', (code) => {
  console.log('Process beforeExit event with code: ', code);
});

process.on('exit', (code) => {
  console.log('Process exit event with code: ', code);
});

console.log('This message is displayed first.');

// Prints:
// This message is displayed first.
// Process beforeExit event with code: 0
// Process exit event with code: 0
```

```
const process = require('process');

process.on('beforeExit', (code) => {
  console.log('Process beforeExit event with code: ', code);
```

```
});

process.on('exit', (code) => {
  console.log('Process exit event with code: ', code);
});

console.log('This message is displayed first.');

// Prints:
// This message is displayed first.
// Process beforeExit event with code: 0
// Process exit event with code: 0
```

Event: `'disconnect'`

If the Node.js process is spawned with an IPC channel (see the Child Process and Cluster documentation), the `'disconnect'` event will be emitted when the IPC channel is closed.

Event: `'exit'`

- `code` {integer}

The `'exit'` event is emitted when the Node.js process is about to exit as a result of either:

- The `process.exit()` method being called explicitly;
- The Node.js event loop no longer having any additional work to perform.

There is no way to prevent the exiting of the event loop at this point, and once all `'exit'` listeners have finished running the Node.js process will terminate.

The listener callback function is invoked with the exit code specified either by the `process.exitCode` property, or the `exitCode` argument passed to the `process.exit()` method.

```
import process from 'process';

process.on('exit', (code) => {
  console.log(`About to exit with code: ${code}`);
});
```

```
const process = require('process');

process.on('exit', (code) => {
  console.log(`About to exit with code: ${code}`);
});
```

Listener functions **must** only perform **synchronous** operations. The Node.js process will exit immediately after calling the `'exit'` event listeners causing any

additional work still queued in the event loop to be abandoned. In the following example, for instance, the timeout will never occur:

```
import process from 'process';

process.on('exit', (code) => {
  setTimeout(() => {
    console.log('This will not run');
  }, 0);
});
```

```
const process = require('process');

process.on('exit', (code) => {
  setTimeout(() => {
    console.log('This will not run');
  }, 0);
});
```

Event: `'message'`

- `message` { Object | boolean | number | string | null } a parsed JSON object or a serializable primitive value.
- `sendHandle` {net.Server|net.Socket} a `net.Server` or `net.Socket` object, or undefined.

If the Node.js process is spawned with an IPC channel (see the Child Process and Cluster documentation), the `'message'` event is emitted whenever a message sent by a parent process using `childprocess.send()` is received by the child process.

The message goes through serialization and parsing. The resulting message might not be the same as what is originally sent.

If the `serialization` option was set to `advanced` used when spawning the process, the `message` argument can contain data that JSON is not able to represent. See Advanced serialization for `child_process` for more details.

Event: `'multipleResolves'`

Stability: 0 - Deprecated

- `type` {string} The resolution type. One of `'resolve'` or `'reject'`.
- `promise` {Promise} The promise that resolved or rejected more than once.
- `value` {any} The value with which the promise was either resolved or rejected after the original resolve.

The `'multipleResolves'` event is emitted whenever a `Promise` has been either:

- Resolved more than once.

- Rejected more than once.
- Rejected after resolve.
- Resolved after reject.

This is useful for tracking potential errors in an application while using the `Promise` constructor, as multiple resolutions are silently swallowed. However, the occurrence of this event does not necessarily indicate an error. For example, `Promise.race()` can trigger a `'multipleResolves'` event.

Because of the unreliability of the event in cases like the `Promise.race()` example above it has been deprecated.

```
import process from 'process';

process.on('multipleResolves', (type, promise, reason) => {
  console.error(type, promise, reason);
  setImmediate(() => process.exit(1));
});

async function main() {
  try {
    return await new Promise((resolve, reject) => {
      resolve('First call');
      resolve('Swallowed resolve');
      reject(new Error('Swallowed reject'));
    });
  } catch {
    throw new Error('Failed');
  }
}

main().then(console.log);
// resolve: Promise { 'First call' } 'Swallowed resolve'
// reject: Promise { 'First call' } Error: Swallowed reject
//     at Promise (*)
//     at new Promise (<anonymous>)
//     at main (*)
// First call
```

```
const process = require('process');

process.on('multipleResolves', (type, promise, reason) => {
  console.error(type, promise, reason);
  setImmediate(() => process.exit(1));
});

async function main() {
  try {
```

```
    return await new Promise((resolve, reject) => {
      resolve('First call');
      resolve('Swallowed resolve');
      reject(new Error('Swallowed reject'));
    });
  } catch {
    throw new Error('Failed');
  }
}

main().then(console.log);
// resolve: Promise { 'First call' } 'Swallowed resolve'
// reject: Promise { 'First call' } Error: Swallowed reject
//     at Promise (*)
//     at new Promise (<anonymous>)
//     at main (*)
// First call
```

Event: `'rejectionHandled'`

- `promise` {Promise} The late handled promise.

The `'rejectionHandled'` event is emitted whenever a `Promise` has been rejected and an error handler was attached to it (using `promise.catch()`, for example) later than one turn of the Node.js event loop.

The `Promise` object would have previously been emitted in an `'unhandledRejection'` event, but during the course of processing gained a rejection handler.

There is no notion of a top level for a `Promise` chain at which rejections can always be handled. Being inherently asynchronous in nature, a `Promise` rejection can be handled at a future point in time, possibly much later than the event loop turn it takes for the `'unhandledRejection'` event to be emitted.

Another way of stating this is that, unlike in synchronous code where there is an ever-growing list of unhandled exceptions, with Promises there can be a growing-and-shrinking list of unhandled rejections.

In synchronous code, the `'uncaughtException'` event is emitted when the list of unhandled exceptions grows.

In asynchronous code, the `'unhandledRejection'` event is emitted when the list of unhandled rejections grows, and the `'rejectionHandled'` event is emitted when the list of unhandled rejections shrinks.

```
import process from 'process';

const unhandledRejections = new Map();
process.on('unhandledRejection', (reason, promise) => {
```

```
  unhandledRejections.set(promise, reason);
});
process.on('rejectionHandled', (promise) => {
  unhandledRejections.delete(promise);
});
```

```
const process = require('process');

const unhandledRejections = new Map();
process.on('unhandledRejection', (reason, promise) => {
  unhandledRejections.set(promise, reason);
});
process.on('rejectionHandled', (promise) => {
  unhandledRejections.delete(promise);
});
```

In this example, the `unhandledRejections` `Map` will grow and shrink over time, reflecting rejections that start unhandled and then become handled. It is possible to record such errors in an error log, either periodically (which is likely best for long-running application) or upon process exit (which is likely most convenient for scripts).

Event: `'uncaughtException'`

- `err` {Error} The uncaught exception.
- `origin` {string} Indicates if the exception originates from an unhandled rejection or from a synchronous error. Can either be `'uncaughtException'` or `'unhandledRejection'`. The latter is used when in an exception happens in a `Promise` based async context (or if a `Promise` is rejected) and `--unhandled-rejections` flag set to `strict` or `throw` (which is the default) and the rejection is not handled, or when a rejection happens during the command line entry point's ES module static loading phase.

The `'uncaughtException'` event is emitted when an uncaught JavaScript exception bubbles all the way back to the event loop. By default, Node.js handles such exceptions by printing the stack trace to `stderr` and exiting with code 1, overriding any previously set `process.exitCode`. Adding a handler for the `'uncaughtException'` event overrides this default behavior. Alternatively, change the `process.exitCode` in the `'uncaughtException'` handler which will result in the process exiting with the provided exit code. Otherwise, in the presence of such handler the process will exit with 0.

```
import process from 'process';

process.on('uncaughtException', (err, origin) => {
  fs.writeSync(
    process.stderr.fd,
    `Caught exception: ${err}\n` +
```

```
    `Exception origin: ${origin}`
  );
});

setTimeout(() => {
  console.log('This will still run.');
}, 500);

// Intentionally cause an exception, but don't catch it.
nonexistentFunc();
console.log('This will not run.');
```

```
const process = require('process');

process.on('uncaughtException', (err, origin) => {
  fs.writeSync(
    process.stderr.fd,
    `Caught exception: ${err}\n` +
    `Exception origin: ${origin}`
  );
});

setTimeout(() => {
  console.log('This will still run.');
}, 500);

// Intentionally cause an exception, but don't catch it.
nonexistentFunc();
console.log('This will not run.');
```

It is possible to monitor `'uncaughtException'` events without overriding the default behavior to exit the process by installing a `'uncaughtExceptionMonitor'` listener.

Warning: Using `'uncaughtException'` correctly `'uncaughtException'` is a crude mechanism for exception handling intended to be used only as a last resort. The event *should not* be used as an equivalent to `On Error Resume Next`. Unhandled exceptions inherently mean that an application is in an undefined state. Attempting to resume application code without properly recovering from the exception can cause additional unforeseen and unpredictable issues.

Exceptions thrown from within the event handler will not be caught. Instead the process will exit with a non-zero exit code and the stack trace will be printed. This is to avoid infinite recursion.

Attempting to resume normally after an uncaught exception can be similar to pulling out the power cord when upgrading a computer. Nine out of ten times,

nothing happens. But the tenth time, the system becomes corrupted.

The correct use of `'uncaughtException'` is to perform synchronous cleanup of allocated resources (e.g. file descriptors, handles, etc) before shutting down the process. **It is not safe to resume normal operation after `'uncaughtException'`.**

To restart a crashed application in a more reliable way, whether `'uncaughtException'` is emitted or not, an external monitor should be employed in a separate process to detect application failures and recover or restart as needed.

Event: `'uncaughtExceptionMonitor'`

- `err` {Error} The uncaught exception.
- `origin` {string} Indicates if the exception originates from an unhandled rejection or from synchronous errors. Can either be `'uncaughtException'` or `'unhandledRejection'`. The latter is used when in an exception happens in a `Promise` based async context (or if a `Promise` is rejected) and `--unhandled-rejections` flag set to `strict` or `throw` (which is the default) and the rejection is not handled, or when a rejection happens during the command line entry point's ES module static loading phase.

The `'uncaughtExceptionMonitor'` event is emitted before an `'uncaughtException'` event is emitted or a hook installed via `process.setUncaughtExceptionCaptureCallback()` is called.

Installing an `'uncaughtExceptionMonitor'` listener does not change the behavior once an `'uncaughtException'` event is emitted. The process will still crash if no `'uncaughtException'` listener is installed.

```
import process from 'process';

process.on('uncaughtExceptionMonitor', (err, origin) => {
  MyMonitoringTool.logSync(err, origin);
});

// Intentionally cause an exception, but don't catch it.
nonexistentFunc();
// Still crashes Node.js
```

```
const process = require('process');

process.on('uncaughtExceptionMonitor', (err, origin) => {
  MyMonitoringTool.logSync(err, origin);
});

// Intentionally cause an exception, but don't catch it.
nonexistentFunc();
// Still crashes Node.js
```

Event: `'unhandledRejection'`

- `reason` {Error|any} The object with which the promise was rejected (typically an `Error` object).
- `promise` {Promise} The rejected promise.

The `'unhandledRejection'` event is emitted whenever a `Promise` is rejected and no error handler is attached to the promise within a turn of the event loop. When programming with Promises, exceptions are encapsulated as "rejected promises". Rejections can be caught and handled using `promise.catch()` and are propagated through a `Promise` chain. The `'unhandledRejection'` event is useful for detecting and keeping track of promises that were rejected whose rejections have not yet been handled.

```
import process from 'process';

process.on('unhandledRejection', (reason, promise) => {
  console.log('Unhandled Rejection at:', promise, 'reason:', reason);
  // Application specific logging, throwing an error, or other logic here
});

somePromise.then((res) => {
  return reportToUser(JSON.pasre(res)); // Note the typo (`pasre`)
}); // No `.catch()` or `.then()`
```

```
const process = require('process');

process.on('unhandledRejection', (reason, promise) => {
  console.log('Unhandled Rejection at:', promise, 'reason:', reason);
  // Application specific logging, throwing an error, or other logic here
});

somePromise.then((res) => {
  return reportToUser(JSON.pasre(res)); // Note the typo (`pasre`)
}); // No `.catch()` or `.then()`
```

The following will also trigger the `'unhandledRejection'` event to be emitted:

```
import process from 'process';

function SomeResource() {
  // Initially set the loaded status to a rejected promise
  this.loaded = Promise.reject(new Error('Resource not yet loaded!'));
}

const resource = new SomeResource();
// no .catch or .then on resource.loaded for at least a turn
```

```
const process = require('process');
```

```
function SomeResource() {
  // Initially set the loaded status to a rejected promise
  this.loaded = Promise.reject(new Error('Resource not yet loaded!'));
}

const resource = new SomeResource();
// no .catch or .then on resource.loaded for at least a turn
```

In this example case, it is possible to track the rejection as a developer error as would typically be the case for other `'unhandledRejection'` events. To address such failures, a non-operational `.catch(() => { })` handler may be attached to `resource.loaded`, which would prevent the `'unhandledRejection'` event from being emitted.

Event: `'warning'`

- `warning` {Error} Key properties of the warning are:
  - `name` {string} The name of the warning. **Default:** `'Warning'`.
  - `message` {string} A system-provided description of the warning.
  - `stack` {string} A stack trace to the location in the code where the warning was issued.

The `'warning'` event is emitted whenever Node.js emits a process warning.

A process warning is similar to an error in that it describes exceptional conditions that are being brought to the user's attention. However, warnings are not part of the normal Node.js and JavaScript error handling flow. Node.js can emit warnings whenever it detects bad coding practices that could lead to sub-optimal application performance, bugs, or security vulnerabilities.

```
import process from 'process';

process.on('warning', (warning) => {
  console.warn(warning.name);    // Print the warning name
  console.warn(warning.message); // Print the warning message
  console.warn(warning.stack);   // Print the stack trace
});
```

```
const process = require('process');

process.on('warning', (warning) => {
  console.warn(warning.name);    // Print the warning name
  console.warn(warning.message); // Print the warning message
  console.warn(warning.stack);   // Print the stack trace
});
```

By default, Node.js will print process warnings to `stderr`. The `--no-warnings` command-line option can be used to suppress the default console output but the

'warning' event will still be emitted by the process object.

The following example illustrates the warning that is printed to stderr when too many listeners have been added to an event:

```
$ node
> events.defaultMaxListeners = 1;
> process.on('foo', () => {});
> process.on('foo', () => {});
> (node:38638) MaxListenersExceededWarning: Possible EventEmitter memory leak
detected. 2 foo listeners added. Use emitter.setMaxListeners() to increase limit
```

In contrast, the following example turns off the default warning output and adds a custom handler to the 'warning' event:

```
$ node --no-warnings
> const p = process.on('warning', (warning) => console.warn('Do not do that!'));
> events.defaultMaxListeners = 1;
> process.on('foo', () => {});
> process.on('foo', () => {});
> Do not do that!
```

The --trace-warnings command-line option can be used to have the default console output for warnings include the full stack trace of the warning.

Launching Node.js using the --throw-deprecation command-line flag will cause custom deprecation warnings to be thrown as exceptions.

Using the --trace-deprecation command-line flag will cause the custom deprecation to be printed to stderr along with the stack trace.

Using the --no-deprecation command-line flag will suppress all reporting of the custom deprecation.

The *-deprecation command-line flags only affect warnings that use the name 'DeprecationWarning'.

Event: 'worker'

- worker {Worker} The {Worker} that was created.

The 'worker' event is emitted after a new {Worker} thread has been created.

Emitting custom warnings See the process.emitWarning() method for issuing custom or application-specific warnings.

Node.js warning names There are no strict guidelines for warning types (as identified by the name property) emitted by Node.js. New types of warnings can be added at any time. A few of the warning types that are most common include:

- `'DeprecationWarning'` - Indicates use of a deprecated Node.js API or feature. Such warnings must include a `'code'` property identifying the deprecation code.
- `'ExperimentalWarning'` - Indicates use of an experimental Node.js API or feature. Such features must be used with caution as they may change at any time and are not subject to the same strict semantic-versioning and long-term support policies as supported features.
- `'MaxListenersExceededWarning'` - Indicates that too many listeners for a given event have been registered on either an `EventEmitter` or `EventTarget`. This is often an indication of a memory leak.
- `'TimeoutOverflowWarning'` - Indicates that a numeric value that cannot fit within a 32-bit signed integer has been provided to either the `setTimeout()` or `setInterval()` functions.
- `'UnsupportedWarning'` - Indicates use of an unsupported option or feature that will be ignored rather than treated as an error. One example is use of the HTTP response status message when using the HTTP/2 compatibility API.

Signal events

Signal events will be emitted when the Node.js process receives a signal. Please refer to signal(7) for a listing of standard POSIX signal names such as `'SIGINT'`, `'SIGHUP'`, etc.

Signals are not available on `Worker` threads.

The signal handler will receive the signal's name (`'SIGINT'`, `'SIGTERM'`, etc.) as the first argument.

The name of each event will be the uppercase common name for the signal (e.g. `'SIGINT'` for `SIGINT` signals).

```
import process from 'process';

// Begin reading from stdin so the process does not exit.
process.stdin.resume();

process.on('SIGINT', () => {
  console.log('Received SIGINT. Press Control-D to exit.');
});

// Using a single function to handle multiple signals
function handle(signal) {
  console.log(`Received ${signal}`);
}

process.on('SIGINT', handle);
process.on('SIGTERM', handle);
```

```
const process = require('process');

// Begin reading from stdin so the process does not exit.
process.stdin.resume();

process.on('SIGINT', () => {
  console.log('Received SIGINT. Press Control-D to exit.');
});

// Using a single function to handle multiple signals
function handle(signal) {
  console.log(`Received ${signal}`);
}

process.on('SIGINT', handle);
process.on('SIGTERM', handle);
```

- `'SIGUSR1'` is reserved by Node.js to start the debugger. It's possible to install a listener but doing so might interfere with the debugger.
- `'SIGTERM'` and `'SIGINT'` have default handlers on non-Windows platforms that reset the terminal mode before exiting with code `128 + signal number`. If one of these signals has a listener installed, its default behavior will be removed (Node.js will no longer exit).
- `'SIGPIPE'` is ignored by default. It can have a listener installed.
- `'SIGHUP'` is generated on Windows when the console window is closed, and on other platforms under various similar conditions. See signal(7). It can have a listener installed, however Node.js will be unconditionally terminated by Windows about 10 seconds later. On non-Windows platforms, the default behavior of `SIGHUP` is to terminate Node.js, but once a listener has been installed its default behavior will be removed.
- `'SIGTERM'` is not supported on Windows, it can be listened on.
- `'SIGINT'` from the terminal is supported on all platforms, and can usually be generated with Ctrl+C (though this may be configurable). It is not generated when terminal raw mode is enabled and Ctrl+C is used.
- `'SIGBREAK'` is delivered on Windows when Ctrl+Break is pressed. On non-Windows platforms, it can be listened on, but there is no way to send or generate it.
- `'SIGWINCH'` is delivered when the console has been resized. On Windows, this will only happen on write to the console when the cursor is being moved, or when a readable tty is used in raw mode.
- `'SIGKILL'` cannot have a listener installed, it will unconditionally terminate Node.js on all platforms.
- `'SIGSTOP'` cannot have a listener installed.
- `'SIGBUS'`, `'SIGFPE'`, `'SIGSEGV'` and `'SIGILL'`, when not raised artificially using kill(2), inherently leave the process in a state from which it is not safe to call JS listeners. Doing so might cause the process to stop responding.

- `0` can be sent to test for the existence of a process, it has no effect if the process exists, but will throw an error if the process does not exist.

Windows does not support signals so has no equivalent to termination by signal, but Node.js offers some emulation with `process.kill()`, and `subprocess.kill()`:

- Sending `SIGINT`, `SIGTERM`, and `SIGKILL` will cause the unconditional termination of the target process, and afterwards, subprocess will report that the process was terminated by signal.
- Sending signal `0` can be used as a platform independent way to test for the existence of a process.

`process.abort()`

The `process.abort()` method causes the Node.js process to exit immediately and generate a core file.

This feature is not available in `Worker` threads.

`process.allowedNodeEnvironmentFlags`

- {Set}

The `process.allowedNodeEnvironmentFlags` property is a special, read-only `Set` of flags allowable within the `NODE_OPTIONS` environment variable.

`process.allowedNodeEnvironmentFlags` extends `Set`, but overrides `Set.prototype.has` to recognize several different possible flag representations. `process.allowedNodeEnvironmentFlags.has()` will return `true` in the following cases:

- Flags may omit leading single (`-`) or double (`--`) dashes; e.g., `inspect-brk` for `--inspect-brk`, or `r` for `-r`.
- Flags passed through to V8 (as listed in `--v8-options`) may replace one or more *non-leading* dashes for an underscore, or vice-versa; e.g., `--perf_basic_prof`, `--perf-basic-prof`, `--perf_basic-prof`, etc.
- Flags may contain one or more equals (`=`) characters; all characters after and including the first equals will be ignored; e.g., `--stack-trace-limit=100`.
- Flags *must* be allowable within `NODE_OPTIONS`.

When iterating over `process.allowedNodeEnvironmentFlags`, flags will appear only *once*; each will begin with one or more dashes. Flags passed through to V8 will contain underscores instead of non-leading dashes:

```
import { allowedNodeEnvironmentFlags } from 'process';

allowedNodeEnvironmentFlags.forEach((flag) => {
  // -r
  // --inspect-brk
```

```
  // --abort_on_uncaught_exception
  // ...
});

const { allowedNodeEnvironmentFlags } = require('process');

allowedNodeEnvironmentFlags.forEach((flag) => {
  // -r
  // --inspect-brk
  // --abort_on_uncaught_exception
  // ...
});
```

The methods `add()`, `clear()`, and `delete()` of `process.allowedNodeEnvironmentFlags` do nothing, and will fail silently.

If Node.js was compiled *without* `NODE_OPTIONS` support (shown in `process.config`), `process.allowedNodeEnvironmentFlags` will contain what *would have* been allowable.

process.arch

- {string}

The operating system CPU architecture for which the Node.js binary was compiled. Possible values are: `'arm'`, `'arm64'`, `'ia32'`, `'mips'`,`'mipsel'`, `'ppc'`, `'ppc64'`, `'s390'`, `'s390x'`, and `'x64'`.

```
import { arch } from 'process';

console.log(`This processor architecture is ${arch}`);

const { arch } = require('process');

console.log(`This processor architecture is ${arch}`);
```

process.argv

- {string[]}

The `process.argv` property returns an array containing the command-line arguments passed when the Node.js process was launched. The first element will be `process.execPath`. See `process.argv0` if access to the original value of `argv[0]` is needed. The second element will be the path to the JavaScript file being executed. The remaining elements will be any additional command-line arguments.

For example, assuming the following script for `process-args.js`:

```
import { argv } from 'process';

// print process.argv
argv.forEach((val, index) => {
  console.log(`${index}: ${val}`);
});
```

```
const { argv } = require('process');

// print process.argv
argv.forEach((val, index) => {
  console.log(`${index}: ${val}`);
});
```

Launching the Node.js process as:

```
$ node process-args.js one two=three four
```

Would generate the output:

```
0: /usr/local/bin/node
1: /Users/mjr/work/node/process-args.js
2: one
3: two=three
4: four
```

process.argv0

- {string}

The `process.argv0` property stores a read-only copy of the original value of `argv[0]` passed when Node.js starts.

```
$ bash -c 'exec -a customArgv0 ./node'
> process.argv[0]
'/Volumes/code/external/node/out/Release/node'
> process.argv0
'customArgv0'
```

process.channel

- {Object}

If the Node.js process was spawned with an IPC channel (see the Child Process documentation), the `process.channel` property is a reference to the IPC channel. If no IPC channel exists, this property is `undefined`.

`process.channel.ref()`

This method makes the IPC channel keep the event loop of the process running if `.unref()` has been called before.

Typically, this is managed through the number of `'disconnect'` and `'message'` listeners on the `process` object. However, this method can be used to explicitly request a specific behavior.

`process.channel.unref()`

This method makes the IPC channel not keep the event loop of the process running, and lets it finish even while the channel is open.

Typically, this is managed through the number of `'disconnect'` and `'message'` listeners on the `process` object. However, this method can be used to explicitly request a specific behavior.

`process.chdir(directory)`

- `directory` {string}

The `process.chdir()` method changes the current working directory of the Node.js process or throws an exception if doing so fails (for instance, if the specified `directory` does not exist).

```
import { chdir, cwd } from 'process';

console.log(`Starting directory: ${cwd()}`);
try {
  chdir('/tmp');
  console.log(`New directory: ${cwd()}`);
} catch (err) {
  console.error(`chdir: ${err}`);
}
```

```
const { chdir, cwd } = require('process');

console.log(`Starting directory: ${cwd()}`);
try {
  chdir('/tmp');
  console.log(`New directory: ${cwd()}`);
} catch (err) {
  console.error(`chdir: ${err}`);
}
```

This feature is not available in `Worker` threads.

process.config

- {Object}

The `process.config` property returns an `Object` containing the JavaScript representation of the configure options used to compile the current Node.js executable. This is the same as the `config.gypi` file that was produced when running the `./configure` script.

An example of the possible output looks like:

```
{
  target_defaults:
   { cflags: [],
     default_configuration: 'Release',
     defines: [],
     include_dirs: [],
     libraries: [] },
  variables:
   {
     host_arch: 'x64',
     napi_build_version: 5,
     node_install_npm: 'true',
     node_prefix: '',
     node_shared_cares: 'false',
     node_shared_http_parser: 'false',
     node_shared_libuv: 'false',
     node_shared_zlib: 'false',
     node_use_dtrace: 'false',
     node_use_openssl: 'true',
     node_shared_openssl: 'false',
     strict_aliasing: 'true',
     target_arch: 'x64',
     v8_use_snapshot: 1
   }
}
```

The `process.config` property is **not** read-only and there are existing modules in the ecosystem that are known to extend, modify, or entirely replace the value of `process.config`.

Modifying the `process.config` property, or any child-property of the `process.config` object has been deprecated. The `process.config` will be made read-only in a future release.

process.connected

- {boolean}

If the Node.js process is spawned with an IPC channel (see the Child Process and Cluster documentation), the `process.connected` property will return `true` so long as the IPC channel is connected and will return `false` after `process.disconnect()` is called.

Once `process.connected` is `false`, it is no longer possible to send messages over the IPC channel using `process.send()`.

`process.cpuUsage([previousValue])`

- `previousValue` {Object} A previous return value from calling `process.cpuUsage()`
- Returns: {Object}
  - `user` {integer}
  - `system` {integer}

The `process.cpuUsage()` method returns the user and system CPU time usage of the current process, in an object with properties `user` and `system`, whose values are microsecond values (millionth of a second). These values measure time spent in user and system code respectively, and may end up being greater than actual elapsed time if multiple CPU cores are performing work for this process.

The result of a previous call to `process.cpuUsage()` can be passed as the argument to the function, to get a diff reading.

```
import { cpuUsage } from 'process';

const startUsage = cpuUsage();
// { user: 38579, system: 6986 }

// spin the CPU for 500 milliseconds
const now = Date.now();
while (Date.now() - now < 500);

console.log(cpuUsage(startUsage));
// { user: 514883, system: 11226 }
```

```
const { cpuUsage } = require('process');

const startUsage = cpuUsage();
// { user: 38579, system: 6986 }

// spin the CPU for 500 milliseconds
const now = Date.now();
while (Date.now() - now < 500);

console.log(cpuUsage(startUsage));
// { user: 514883, system: 11226 }
```

process.cwd()

- Returns: {string}

The `process.cwd()` method returns the current working directory of the Node.js process.

```
import { cwd } from 'process';

console.log(`Current directory: ${cwd()}`);
```

```
const { cwd } = require('process');

console.log(`Current directory: ${cwd()}`);
```

process.debugPort

- {number}

The port used by the Node.js debugger when enabled.

```
import process from 'process';

process.debugPort = 5858;
```

```
const process = require('process');

process.debugPort = 5858;
```

process.disconnect()

If the Node.js process is spawned with an IPC channel (see the Child Process and Cluster documentation), the `process.disconnect()` method will close the IPC channel to the parent process, allowing the child process to exit gracefully once there are no other connections keeping it alive.

The effect of calling `process.disconnect()` is the same as calling `ChildProcess.disconnect()` from the parent process.

If the Node.js process was not spawned with an IPC channel, `process.disconnect()` will be `undefined`.

process.dlopen(module, filename[, flags])

- `module` {Object}
- `filename` {string}
- `flags` {os.constants.dlopen} **Default:** `os.constants.dlopen.RTLD_LAZY`

The `process.dlopen()` method allows dynamically loading shared objects. It is primarily used by `require()` to load C++ Addons, and should not be used directly, except in special cases. In other words, `require()` should be preferred

over `process.dlopen()` unless there are specific reasons such as custom dlopen flags or loading from ES modules.

The `flags` argument is an integer that allows to specify dlopen behavior. See the `os.constants.dlopen` documentation for details.

An important requirement when calling `process.dlopen()` is that the `module` instance must be passed. Functions exported by the C++ Addon are then accessible via `module.exports`.

The example below shows how to load a C++ Addon, named `local.node`, that exports a `foo` function. All the symbols are loaded before the call returns, by passing the `RTLD_NOW` constant. In this example the constant is assumed to be available.

```
import { dlopen } from 'process';
import { constants } from 'os';
import { fileURLToPath } from 'url';

const module = { exports: {} };
dlopen(module, fileURLToPath(new URL('local.node', import.meta.url)),
       constants.dlopen.RTLD_NOW);
module.exports.foo();
```

```
const { dlopen } = require('process');
const { constants } = require('os');
const { join } = require('path');

const module = { exports: {} };
dlopen(module, join(__dirname, 'local.node'), constants.dlopen.RTLD_NOW);
module.exports.foo();
```

`process.emitWarning(warning[, options])`

- `warning` {string|Error} The warning to emit.
- `options` {Object}
  - `type` {string} When `warning` is a `String`, `type` is the name to use for the *type* of warning being emitted. **Default:** `'Warning'`.
  - `code` {string} A unique identifier for the warning instance being emitted.
  - `ctor` {Function} When `warning` is a `String`, `ctor` is an optional function used to limit the generated stack trace. **Default:** `process.emitWarning`.
  - `detail` {string} Additional text to include with the error.

The `process.emitWarning()` method can be used to emit custom or application specific process warnings. These can be listened for by adding a handler to the `'warning'` event.

```
import { emitWarning } from 'process';

// Emit a warning with a code and additional detail.
emitWarning('Something happened!', {
  code: 'MY_WARNING',
  detail: 'This is some additional information'
});
// Emits:
// (node:56338) [MY_WARNING] Warning: Something happened!
// This is some additional information
```

```
const { emitWarning } = require('process');

// Emit a warning with a code and additional detail.
emitWarning('Something happened!', {
  code: 'MY_WARNING',
  detail: 'This is some additional information'
});
// Emits:
// (node:56338) [MY_WARNING] Warning: Something happened!
// This is some additional information
```

In this example, an `Error` object is generated internally by `process.emitWarning()` and passed through to the `'warning'` handler.

```
import process from 'process';

process.on('warning', (warning) => {
  console.warn(warning.name);    // 'Warning'
  console.warn(warning.message); // 'Something happened!'
  console.warn(warning.code);    // 'MY_WARNING'
  console.warn(warning.stack);   // Stack trace
  console.warn(warning.detail);  // 'This is some additional information'
});
```

```
const process = require('process');

process.on('warning', (warning) => {
  console.warn(warning.name);    // 'Warning'
  console.warn(warning.message); // 'Something happened!'
  console.warn(warning.code);    // 'MY_WARNING'
  console.warn(warning.stack);   // Stack trace
  console.warn(warning.detail);  // 'This is some additional information'
});
```

If `warning` is passed as an `Error` object, the `options` argument is ignored.

`process.emitWarning(warning[, type[, code]][, ctor])`

- `warning` {string|Error} The warning to emit.
- `type` {string} When `warning` is a `String`, `type` is the name to use for the *type* of warning being emitted. **Default:** `'Warning'`.
- `code` {string} A unique identifier for the warning instance being emitted.
- `ctor` {Function} When `warning` is a `String`, `ctor` is an optional function used to limit the generated stack trace. **Default:** `process.emitWarning`.

The `process.emitWarning()` method can be used to emit custom or application specific process warnings. These can be listened for by adding a handler to the `'warning'` event.

```
import { emitWarning } from 'process';

// Emit a warning using a string.
emitWarning('Something happened!');
// Emits: (node: 56338) Warning: Something happened!
```

```
const { emitWarning } = require('process');

// Emit a warning using a string.
emitWarning('Something happened!');
// Emits: (node: 56338) Warning: Something happened!
```

```
import { emitWarning } from 'process';

// Emit a warning using a string and a type.
emitWarning('Something Happened!', 'CustomWarning');
// Emits: (node:56338) CustomWarning: Something Happened!
```

```
const { emitWarning } = require('process');

// Emit a warning using a string and a type.
emitWarning('Something Happened!', 'CustomWarning');
// Emits: (node:56338) CustomWarning: Something Happened!
```

```
import { emitWarning } from 'process';

emitWarning('Something happened!', 'CustomWarning', 'WARN001');
// Emits: (node:56338) [WARN001] CustomWarning: Something happened!
```

```
const { emitWarning } = require('process');

process.emitWarning('Something happened!', 'CustomWarning', 'WARN001');
// Emits: (node:56338) [WARN001] CustomWarning: Something happened!
```

In each of the previous examples, an `Error` object is generated internally by `process.emitWarning()` and passed through to the `'warning'` handler.

```
import process from 'process';

process.on('warning', (warning) => {
  console.warn(warning.name);
  console.warn(warning.message);
  console.warn(warning.code);
  console.warn(warning.stack);
});
```

```
const process = require('process');

process.on('warning', (warning) => {
  console.warn(warning.name);
  console.warn(warning.message);
  console.warn(warning.code);
  console.warn(warning.stack);
});
```

If `warning` is passed as an `Error` object, it will be passed through to the `'warning'` event handler unmodified (and the optional `type`, `code` and `ctor` arguments will be ignored):

```
import { emitWarning } from 'process';

// Emit a warning using an Error object.
const myWarning = new Error('Something happened!');
// Use the Error name property to specify the type name
myWarning.name = 'CustomWarning';
myWarning.code = 'WARN001';

emitWarning(myWarning);
// Emits: (node:56338) [WARN001] CustomWarning: Something happened!
```

```
const { emitWarning } = require('process');

// Emit a warning using an Error object.
const myWarning = new Error('Something happened!');
// Use the Error name property to specify the type name
myWarning.name = 'CustomWarning';
myWarning.code = 'WARN001';

emitWarning(myWarning);
// Emits: (node:56338) [WARN001] CustomWarning: Something happened!
```

A `TypeError` is thrown if `warning` is anything other than a string or `Error` object.

While process warnings use `Error` objects, the process warning mechanism is **not** a replacement for normal error handling mechanisms.

The following additional handling is implemented if the warning `type` is `'DeprecationWarning'`:

- If the `--throw-deprecation` command-line flag is used, the deprecation warning is thrown as an exception rather than being emitted as an event.
- If the `--no-deprecation` command-line flag is used, the deprecation warning is suppressed.
- If the `--trace-deprecation` command-line flag is used, the deprecation warning is printed to `stderr` along with the full stack trace.

Avoiding duplicate warnings

As a best practice, warnings should be emitted only once per process. To do so, place the `emitWarning()` behind a boolean.

```
import { emitWarning } from 'process';

function emitMyWarning() {
  if (!emitMyWarning.warned) {
    emitMyWarning.warned = true;
    emitWarning('Only warn once!');
  }
}
emitMyWarning();
// Emits: (node: 56339) Warning: Only warn once!
emitMyWarning();
// Emits nothing
```

```
const { emitWarning } = require('process');

function emitMyWarning() {
  if (!emitMyWarning.warned) {
    emitMyWarning.warned = true;
    emitWarning('Only warn once!');
  }
}
emitMyWarning();
// Emits: (node: 56339) Warning: Only warn once!
emitMyWarning();
// Emits nothing
```

`process.env`

- {Object}

The `process.env` property returns an object containing the user environment. See environ(7).

An example of this object looks like:

```
{
  TERM: 'xterm-256color',
  SHELL: '/usr/local/bin/bash',
  USER: 'maciej',
  PATH: '~/.bin/:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin',
  PWD: '/Users/maciej',
  EDITOR: 'vim',
  SHLVL: '1',
  HOME: '/Users/maciej',
  LOGNAME: 'maciej',
  _: '/usr/local/bin/node'
}
```

It is possible to modify this object, but such modifications will not be reflected outside the Node.js process, or (unless explicitly requested) to other `Worker` threads. In other words, the following example would not work:

```
$ node -e 'process.env.foo = "bar"' && echo $foo
```

While the following will:

```
import { env } from 'process';

env.foo = 'bar';
console.log(env.foo);
```

```
const { env } = require('process');

env.foo = 'bar';
console.log(env.foo);
```

Assigning a property on `process.env` will implicitly convert the value to a string. **This behavior is deprecated.** Future versions of Node.js may throw an error when the value is not a string, number, or boolean.

```
import { env } from 'process';

env.test = null;
console.log(env.test);
// => 'null'
env.test = undefined;
console.log(env.test);
// => 'undefined'
```

```
const { env } = require('process');

env.test = null;
console.log(env.test);
// => 'null'
env.test = undefined;
```

```
console.log(env.test);
// => 'undefined'
```

Use `delete` to delete a property from `process.env`.

```
import { env } from 'process';

env.TEST = 1;
delete env.TEST;
console.log(env.TEST);
// => undefined
```

```
const { env } = require('process');

env.TEST = 1;
delete env.TEST;
console.log(env.TEST);
// => undefined
```

On Windows operating systems, environment variables are case-insensitive.

```
import { env } from 'process';

env.TEST = 1;
console.log(env.test);
// => 1
```

```
const { env } = require('process');

env.TEST = 1;
console.log(env.test);
// => 1
```

Unless explicitly specified when creating a `Worker` instance, each `Worker` thread has its own copy of `process.env`, based on its parent thread's `process.env`, or whatever was specified as the `env` option to the `Worker` constructor. Changes to `process.env` will not be visible across `Worker` threads, and only the main thread can make changes that are visible to the operating system or to native add-ons.

`process.execArgv`

- {string[]}

The `process.execArgv` property returns the set of Node.js-specific command-line options passed when the Node.js process was launched. These options do not appear in the array returned by the `process.argv` property, and do not include the Node.js executable, the name of the script, or any options following the script name. These options are useful in order to spawn child processes with the same execution environment as the parent.

```
$ node --harmony script.js --version
```

Results in `process.execArgv`:

```
['--harmony']
```

And `process.argv`:

```
['/usr/local/bin/node', 'script.js', '--version']
```

Refer to `Worker` constructor for the detailed behavior of worker threads with this property.

process.execPath

- {string}

The `process.execPath` property returns the absolute pathname of the executable that started the Node.js process. Symbolic links, if any, are resolved.

```
'/usr/local/bin/node'
```

process.exit([code])

- `code` {integer} The exit code. **Default:** `0`.

The `process.exit()` method instructs Node.js to terminate the process synchronously with an exit status of `code`. If `code` is omitted, exit uses either the 'success' code `0` or the value of `process.exitCode` if it has been set. Node.js will not terminate until all the `'exit'` event listeners are called.

To exit with a 'failure' code:

```
import { exit } from 'process';

exit(1);
```

```
const { exit } = require('process');

exit(1);
```

The shell that executed Node.js should see the exit code as `1`.

Calling `process.exit()` will force the process to exit as quickly as possible even if there are still asynchronous operations pending that have not yet completed fully, including I/O operations to `process.stdout` and `process.stderr`.

In most situations, it is not actually necessary to call `process.exit()` explicitly. The Node.js process will exit on its own *if there is no additional work pending* in the event loop. The `process.exitCode` property can be set to tell the process which exit code to use when the process exits gracefully.

For instance, the following example illustrates a *misuse* of the `process.exit()` method that could lead to data printed to stdout being truncated and lost:

```
import { exit } from 'process';

// This is an example of what *not* to do:
if (someConditionNotMet()) {
  printUsageToStdout();
  exit(1);
}
```

```
const { exit } = require('process');

// This is an example of what *not* to do:
if (someConditionNotMet()) {
  printUsageToStdout();
  exit(1);
}
```

The reason this is problematic is because writes to `process.stdout` in Node.js are sometimes *asynchronous* and may occur over multiple ticks of the Node.js event loop. Calling `process.exit()`, however, forces the process to exit *before* those additional writes to `stdout` can be performed.

Rather than calling `process.exit()` directly, the code *should* set the `process.exitCode` and allow the process to exit naturally by avoiding scheduling any additional work for the event loop:

```
import process from 'process';

// How to properly set the exit code while letting
// the process exit gracefully.
if (someConditionNotMet()) {
  printUsageToStdout();
  process.exitCode = 1;
}
```

```
const process = require('process');

// How to properly set the exit code while letting
// the process exit gracefully.
if (someConditionNotMet()) {
  printUsageToStdout();
  process.exitCode = 1;
}
```

If it is necessary to terminate the Node.js process due to an error condition, throwing an *uncaught* error and allowing the process to terminate accordingly is safer than calling `process.exit()`.

In `Worker` threads, this function stops the current thread rather than the current process.

`process.exitCode`

- {integer}

A number which will be the process exit code, when the process either exits gracefully, or is exited via `process.exit()` without specifying a code.

Specifying a code to `process.exit(code)` will override any previous setting of `process.exitCode`.

`process.getActiveResourcesInfo()`

Stability: 1 - Experimental

- Returns: {string[]}

The `process.getActiveResourcesInfo()` method returns an array of strings containing the types of the active resources that are currently keeping the event loop alive.

```
import { getActiveResourcesInfo } from 'process';
import { setTimeout } from 'timers';

console.log('Before:', getActiveResourcesInfo());
setTimeout(() => {}, 1000);
console.log('After:', getActiveResourcesInfo());
// Prints:
//   Before: [ 'CloseReq', 'TTYWrap', 'TTYWrap', 'TTYWrap' ]
//   After: [ 'CloseReq', 'TTYWrap', 'TTYWrap', 'TTYWrap', 'Timeout' ]
```

```
const { getActiveResourcesInfo } = require('process');
const { setTimeout } = require('timers');

console.log('Before:', getActiveResourcesInfo());
setTimeout(() => {}, 1000);
console.log('After:', getActiveResourcesInfo());
// Prints:
//   Before: [ 'TTYWrap', 'TTYWrap', 'TTYWrap' ]
//   After: [ 'TTYWrap', 'TTYWrap', 'TTYWrap', 'Timeout' ]
```

`process.getegid()`

The `process.getegid()` method returns the numerical effective group identity of the Node.js process. (See getegid(2).)

```
import process from 'process';
```

```
if (process.getegid) {
  console.log(`Current gid: ${process.getegid()}`);
}
```

```
const process = require('process');

if (process.getegid) {
  console.log(`Current gid: ${process.getegid()}`);
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.geteuid()

- Returns: {Object}

The `process.geteuid()` method returns the numerical effective user identity of the process. (See geteuid(2).)

```
import process from 'process';

if (process.geteuid) {
  console.log(`Current uid: ${process.geteuid()}`);
}
```

```
const process = require('process');

if (process.geteuid) {
  console.log(`Current uid: ${process.geteuid()}`);
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.getgid()

- Returns: {Object}

The `process.getgid()` method returns the numerical group identity of the process. (See getgid(2).)

```
import process from 'process';

if (process.getgid) {
  console.log(`Current gid: ${process.getgid()}`);
}
```

```
const process = require('process');

if (process.getgid) {
```

```
  console.log(`Current gid: ${process.getgid()}`);
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.getgroups()

- Returns: {integer[]}

The `process.getgroups()` method returns an array with the supplementary group IDs. POSIX leaves it unspecified if the effective group ID is included but Node.js ensures it always is.

```
import process from 'process';

if (process.getgroups) {
  console.log(process.getgroups()); // [ 16, 21, 297 ]
}
```

```
const process = require('process');

if (process.getgroups) {
  console.log(process.getgroups()); // [ 16, 21, 297 ]
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.getuid()

- Returns: {integer}

The `process.getuid()` method returns the numeric user identity of the process. (See getuid(2).)

```
import process from 'process';

if (process.getuid) {
  console.log(`Current uid: ${process.getuid()}`);
}
```

```
const process = require('process');

if (process.getuid) {
  console.log(`Current uid: ${process.getuid()}`);
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.hasUncaughtExceptionCaptureCallback()

- Returns: {boolean}

Indicates whether a callback has been set using `process.setUncaughtExceptionCaptureCallback()`.

`process.hrtime([time])`

Stability: 3 - Legacy. Use `process.hrtime.bigint()` instead.

- `time` {integer[]} The result of a previous call to `process.hrtime()`
- Returns: {integer[]}

This is the legacy version of `process.hrtime.bigint()` before `bigint` was introduced in JavaScript.

The `process.hrtime()` method returns the current high-resolution real time in a `[seconds, nanoseconds]` tuple `Array`, where `nanoseconds` is the remaining part of the real time that can't be represented in second precision.

`time` is an optional parameter that must be the result of a previous `process.hrtime()` call to diff with the current time. If the parameter passed in is not a tuple `Array`, a `TypeError` will be thrown. Passing in a user-defined array instead of the result of a previous call to `process.hrtime()` will lead to undefined behavior.

These times are relative to an arbitrary time in the past, and not related to the time of day and therefore not subject to clock drift. The primary use is for measuring performance between intervals:

```
import { hrtime } from 'process';

const NS_PER_SEC = 1e9;
const time = hrtime();
// [ 1800216, 25 ]

setTimeout(() => {
  const diff = hrtime(time);
  // [ 1, 552 ]

  console.log(`Benchmark took ${diff[0] * NS_PER_SEC + diff[1]} nanoseconds`);
  // Benchmark took 1000000552 nanoseconds
}, 1000);
```

```
const { hrtime } = require('process');

const NS_PER_SEC = 1e9;
const time = hrtime();
// [ 1800216, 25 ]

setTimeout(() => {
  const diff = hrtime(time);
  // [ 1, 552 ]
```

```
  console.log(`Benchmark took ${diff[0] * NS_PER_SEC + diff[1]} nanoseconds`);
  // Benchmark took 1000000552 nanoseconds
}, 1000);
```

process.hrtime.bigint()

- Returns: {bigint}

The `bigint` version of the `process.hrtime()` method returning the current high-resolution real time in nanoseconds as a `bigint`.

Unlike `process.hrtime()`, it does not support an additional `time` argument since the difference can just be computed directly by subtraction of the two `bigint`s.

```
import { hrtime } from 'process';

const start = hrtime.bigint();
// 191051479007711n

setTimeout(() => {
  const end = hrtime.bigint();
  // 191052633396993n

  console.log(`Benchmark took ${end - start} nanoseconds`);
  // Benchmark took 1154389282 nanoseconds
}, 1000);
```

```
const { hrtime } = require('process');

const start = hrtime.bigint();
// 191051479007711n

setTimeout(() => {
  const end = hrtime.bigint();
  // 191052633396993n

  console.log(`Benchmark took ${end - start} nanoseconds`);
  // Benchmark took 1154389282 nanoseconds
}, 1000);
```

process.initgroups(user, extraGroup)

- `user` {string|number} The user name or numeric identifier.
- `extraGroup` {string|number} A group name or numeric identifier.

The `process.initgroups()` method reads the `/etc/group` file and initializes

the group access list, using all groups of which the user is a member. This is a privileged operation that requires that the Node.js process either have `root` access or the `CAP_SETGID` capability.

Use care when dropping privileges:

```
import { getgroups, initgroups, setgid } from 'process';

console.log(getgroups());         // [ 0 ]
initgroups('nodeuser', 1000);     // switch user
console.log(getgroups());         // [ 27, 30, 46, 1000, 0 ]
setgid(1000);                     // drop root gid
console.log(getgroups());         // [ 27, 30, 46, 1000 ]
```

```
const { getgroups, initgroups, setgid } = require('process');

console.log(getgroups());         // [ 0 ]
initgroups('nodeuser', 1000);     // switch user
console.log(getgroups());         // [ 27, 30, 46, 1000, 0 ]
setgid(1000);                     // drop root gid
console.log(getgroups());         // [ 27, 30, 46, 1000 ]
```

This function is only available on POSIX platforms (i.e. not Windows or Android). This feature is not available in `Worker` threads.

process.kill(pid[, signal])

- `pid` {number} A process ID
- `signal` {string|number} The signal to send, either as a string or number. **Default:** `'SIGTERM'`.

The `process.kill()` method sends the `signal` to the process identified by `pid`.

Signal names are strings such as `'SIGINT'` or `'SIGHUP'`. See Signal Events and kill(2) for more information.

This method will throw an error if the target `pid` does not exist. As a special case, a signal of `0` can be used to test for the existence of a process. Windows platforms will throw an error if the `pid` is used to kill a process group.

Even though the name of this function is `process.kill()`, it is really just a signal sender, like the `kill` system call. The signal sent may do something other than kill the target process.

```
import process, { kill } from 'process';

process.on('SIGHUP', () => {
  console.log('Got SIGHUP signal.');
});
```

```
setTimeout(() => {
  console.log('Exiting.');
  process.exit(0);
}, 100);

kill(process.pid, 'SIGHUP');
```

```
const process = require('process');

process.on('SIGHUP', () => {
  console.log('Got SIGHUP signal.');
});

setTimeout(() => {
  console.log('Exiting.');
  process.exit(0);
}, 100);

process.kill(process.pid, 'SIGHUP');
```

When `SIGUSR1` is received by a Node.js process, Node.js will start the debugger. See Signal Events.

process.mainModule

Stability: 0 - Deprecated: Use `require.main` instead.

- {Object}

The `process.mainModule` property provides an alternative way of retrieving `require.main`. The difference is that if the main module changes at runtime, `require.main` may still refer to the original main module in modules that were required before the change occurred. Generally, it's safe to assume that the two refer to the same module.

As with `require.main`, `process.mainModule` will be `undefined` if there is no entry script.

process.memoryUsage()

- Returns: {Object}
  - `rss` {integer}
  - `heapTotal` {integer}
  - `heapUsed` {integer}
  - `external` {integer}
  - `arrayBuffers` {integer}

Returns an object describing the memory usage of the Node.js process measured in bytes.

```
import { memoryUsage } from 'process';

console.log(memoryUsage());
// Prints:
// {
//  rss: 4935680,
//  heapTotal: 1826816,
//  heapUsed: 650472,
//  external: 49879,
//  arrayBuffers: 9386
// }
```

```
const { memoryUsage } = require('process');

console.log(memoryUsage());
// Prints:
// {
//  rss: 4935680,
//  heapTotal: 1826816,
//  heapUsed: 650472,
//  external: 49879,
//  arrayBuffers: 9386
// }
```

- `heapTotal` and `heapUsed` refer to V8's memory usage.
- `external` refers to the memory usage of C++ objects bound to JavaScript objects managed by V8.
- `rss`, Resident Set Size, is the amount of space occupied in the main memory device (that is a subset of the total allocated memory) for the process, including all C++ and JavaScript objects and code.
- `arrayBuffers` refers to memory allocated for `ArrayBuffer`s and `SharedArrayBuffer`s, including all Node.js `Buffer`s. This is also included in the `external` value. When Node.js is used as an embedded library, this value may be `0` because allocations for `ArrayBuffer`s may not be tracked in that case.

When using `Worker` threads, `rss` will be a value that is valid for the entire process, while the other fields will only refer to the current thread.

The `process.memoryUsage()` method iterates over each page to gather information about memory usage which might be slow depending on the program memory allocations.

`process.memoryUsage.rss()`

- Returns: {integer}

The `process.memoryUsage.rss()` method returns an integer representing the

Resident Set Size (RSS) in bytes.

The Resident Set Size, is the amount of space occupied in the main memory device (that is a subset of the total allocated memory) for the process, including all C++ and JavaScript objects and code.

This is the same value as the `rss` property provided by `process.memoryUsage()` but `process.memoryUsage.rss()` is faster.

```
import { memoryUsage } from 'process';

console.log(memoryUsage.rss());
// 35655680
```

```
const { rss } = require('process');

console.log(memoryUsage.rss());
// 35655680
```

process.nextTick(callback[, ...args])

- `callback` {Function}
- `...args` {any} Additional arguments to pass when invoking the `callback`

`process.nextTick()` adds `callback` to the "next tick queue". This queue is fully drained after the current operation on the JavaScript stack runs to completion and before the event loop is allowed to continue. It's possible to create an infinite loop if one were to recursively call `process.nextTick()`. See the Event Loop guide for more background.

```
import { nextTick } from 'process';

console.log('start');
nextTick(() => {
  console.log('nextTick callback');
});
console.log('scheduled');
// Output:
// start
// scheduled
// nextTick callback
```

```
const { nextTick } = require('process');

console.log('start');
nextTick(() => {
  console.log('nextTick callback');
});
console.log('scheduled');
```

```
// Output:
// start
// scheduled
// nextTick callback
```

This is important when developing APIs in order to give users the opportunity to assign event handlers *after* an object has been constructed but before any I/O has occurred:

```
import { nextTick } from 'process';

function MyThing(options) {
  this.setupOptions(options);

  nextTick(() => {
    this.startDoingStuff();
  });
}

const thing = new MyThing();
thing.getReadyForStuff();

// thing.startDoingStuff() gets called now, not before.
```

```
const { nextTick } = require('process');

function MyThing(options) {
  this.setupOptions(options);

  nextTick(() => {
    this.startDoingStuff();
  });
}

const thing = new MyThing();
thing.getReadyForStuff();

// thing.startDoingStuff() gets called now, not before.
```

It is very important for APIs to be either 100% synchronous or 100% asynchronous. Consider this example:

```
// WARNING!  DO NOT USE!  BAD UNSAFE HAZARD!
function maybeSync(arg, cb) {
  if (arg) {
    cb();
    return;
  }
```

```
  fs.stat('file', cb);
}
```

This API is hazardous because in the following case:

```
const maybeTrue = Math.random() > 0.5;

maybeSync(maybeTrue, () => {
  foo();
});

bar();
```

It is not clear whether `foo()` or `bar()` will be called first.

The following approach is much better:

```
import { nextTick } from 'process';

function definitelyAsync(arg, cb) {
  if (arg) {
    nextTick(cb);
    return;
  }

  fs.stat('file', cb);
}
```

```
const { nextTick } = require('process');

function definitelyAsync(arg, cb) {
  if (arg) {
    nextTick(cb);
    return;
  }

  fs.stat('file', cb);
}
```

When to use `queueMicrotask()` vs. `process.nextTick()`

The `queueMicrotask()` API is an alternative to `process.nextTick()` that also defers execution of a function using the same microtask queue used to execute the then, catch, and finally handlers of resolved promises. Within Node.js, every time the "next tick queue" is drained, the microtask queue is drained immediately after.

```
import { nextTick } from 'process';

Promise.resolve().then(() => console.log(2));
queueMicrotask(() => console.log(3));
nextTick(() => console.log(1));
// Output:
// 1
// 2
// 3
```

```
const { nextTick } = require('process');

Promise.resolve().then(() => console.log(2));
queueMicrotask(() => console.log(3));
nextTick(() => console.log(1));
// Output:
// 1
// 2
// 3
```

For *most* userland use cases, the `queueMicrotask()` API provides a portable and reliable mechanism for deferring execution that works across multiple JavaScript platform environments and should be favored over `process.nextTick()`. In simple scenarios, `queueMicrotask()` can be a drop-in replacement for `process.nextTick()`.

```
console.log('start');
queueMicrotask(() => {
  console.log('microtask callback');
});
console.log('scheduled');
// Output:
// start
// scheduled
// microtask callback
```

One note-worthy difference between the two APIs is that `process.nextTick()` allows specifying additional values that will be passed as arguments to the deferred function when it is called. Achieving the same result with `queueMicrotask()` requires using either a closure or a bound function:

```
function deferred(a, b) {
  console.log('microtask', a + b);
}

console.log('start');
queueMicrotask(deferred.bind(undefined, 1, 2));
console.log('scheduled');
```

```
// Output:
// start
// scheduled
// microtask 3
```

There are minor differences in the way errors raised from within the next tick queue and microtask queue are handled. Errors thrown within a queued microtask callback should be handled within the queued callback when possible. If they are not, the `process.on('uncaughtException')` event handler can be used to capture and handle the errors.

When in doubt, unless the specific capabilities of `process.nextTick()` are needed, use `queueMicrotask()`.

`process.noDeprecation`

- {boolean}

The `process.noDeprecation` property indicates whether the `--no-deprecation` flag is set on the current Node.js process. See the documentation for the `'warning'` event and the `emitWarning()` method for more information about this flag's behavior.

`process.pid`

- {integer}

The `process.pid` property returns the PID of the process.

```
import { pid } from 'process';

console.log(`This process is pid ${pid}`);
```

```
const { pid } = require('process');

console.log(`This process is pid ${pid}`);
```

`process.platform`

- {string}

The `process.platform` property returns a string identifying the operating system platform for which the Node.js binary was compiled.

Currently possible values are:

- `'aix'`
- `'darwin'`
- `'freebsd'`
- `'linux'`

- `'openbsd'`
- `'sunos'`
- `'win32'`

```
import { platform } from 'process';

console.log(`This platform is ${platform}`);
```

```
const { platform } = require('process');

console.log(`This platform is ${platform}`);
```

The value `'android'` may also be returned if the Node.js is built on the Android operating system. However, Android support in Node.js is experimental.

process.ppid

- {integer}

The `process.ppid` property returns the PID of the parent of the current process.

```
import { ppid } from 'process';

console.log(`The parent process is pid ${ppid}`);
```

```
const { ppid } = require('process');

console.log(`The parent process is pid ${ppid}`);
```

process.release

- {Object}

The `process.release` property returns an `Object` containing metadata related to the current release, including URLs for the source tarball and headers-only tarball.

`process.release` contains the following properties:

- `name` {string} A value that will always be `'node'`.
- `sourceUrl` {string} an absolute URL pointing to a *`.tar.gz`* file containing the source code of the current release.
- `headersUrl`{string} an absolute URL pointing to a *`.tar.gz`* file containing only the source header files for the current release. This file is significantly smaller than the full source file and can be used for compiling Node.js native add-ons.
- `libUrl` {string} an absolute URL pointing to a *`node.lib`* file matching the architecture and version of the current release. This file is used for compiling Node.js native add-ons. *This property is only present on Windows builds of Node.js and will be missing on all other platforms.*

- `lts` {string} a string label identifying the LTS label for this release. This property only exists for LTS releases and is `undefined` for all other release types, including *Current* releases. Valid values include the LTS Release code names (including those that are no longer supported).
  - `'Dubnium'` for the 10.x LTS line beginning with 10.13.0.
  - `'Erbium'` for the 12.x LTS line beginning with 12.13.0. For other LTS Release code names, see Node.js Changelog Archive

```
{
  name: 'node',
  lts: 'Erbium',
  sourceUrl: 'https://nodejs.org/download/release/v12.18.1/node-v12.18.1.tar.gz',
  headersUrl: 'https://nodejs.org/download/release/v12.18.1/node-v12.18.1-headers.tar.gz',
  libUrl: 'https://nodejs.org/download/release/v12.18.1/win-x64/node.lib'
}
```

In custom builds from non-release versions of the source tree, only the `name` property may be present. The additional properties should not be relied upon to exist.

process.report

- {Object}

`process.report` is an object whose methods are used to generate diagnostic reports for the current process. Additional documentation is available in the report documentation.

process.report.compact

- {boolean}

Write reports in a compact format, single-line JSON, more easily consumable by log processing systems than the default multi-line format designed for human consumption.

```
import { report } from 'process';

console.log(`Reports are compact? ${report.compact}`);
```

```
const { report } = require('process');

console.log(`Reports are compact? ${report.compact}`);
```

process.report.directory

- {string}

Directory where the report is written. The default value is the empty string, indicating that reports are written to the current working directory of the Node.js process.

```
import { report } from 'process';

console.log(`Report directory is ${report.directory}`);
```

```
const { report } = require('process');

console.log(`Report directory is ${report.directory}`);
```

`process.report.filename`

- {string}

Filename where the report is written. If set to the empty string, the output filename will be comprised of a timestamp, PID, and sequence number. The default value is the empty string.

```
import { report } from 'process';

console.log(`Report filename is ${report.filename}`);
```

```
const { report } = require('process');

console.log(`Report filename is ${report.filename}`);
```

`process.report.getReport([err])`

- `err` {Error} A custom error used for reporting the JavaScript stack.
- Returns: {Object}

Returns a JavaScript Object representation of a diagnostic report for the running process. The report's JavaScript stack trace is taken from `err`, if present.

```
import { report } from 'process';

const data = report.getReport();
console.log(data.header.nodejsVersion);

// Similar to process.report.writeReport()
import fs from 'fs';
fs.writeFileSync('my-report.log', util.inspect(data), 'utf8');
```

```
const { report } = require('process');

const data = report.getReport();
console.log(data.header.nodejsVersion);
```

```
// Similar to process.report.writeReport()
const fs = require('fs');
fs.writeFileSync('my-report.log', util.inspect(data), 'utf8');
```

Additional documentation is available in the report documentation.

`process.report.reportOnFatalError`

- {boolean}

If `true`, a diagnostic report is generated on fatal errors, such as out of memory errors or failed C++ assertions.

```
import { report } from 'process';

console.log(`Report on fatal error: ${report.reportOnFatalError}`);
```

```
const { report } = require('process');

console.log(`Report on fatal error: ${report.reportOnFatalError}`);
```

`process.report.reportOnSignal`

- {boolean}

If `true`, a diagnostic report is generated when the process receives the signal specified by `process.report.signal`.

```
import { report } from 'process';

console.log(`Report on signal: ${report.reportOnSignal}`);
```

```
const { report } = require('process');

console.log(`Report on signal: ${report.reportOnSignal}`);
```

`process.report.reportOnUncaughtException`

- {boolean}

If `true`, a diagnostic report is generated on uncaught exception.

```
import { report } from 'process';

console.log(`Report on exception: ${report.reportOnUncaughtException}`);
```

```
const { report } = require('process');

console.log(`Report on exception: ${report.reportOnUncaughtException}`);
```

`process.report.signal`

- {string}

The signal used to trigger the creation of a diagnostic report. Defaults to `'SIGUSR2'`.

```
import { report } from 'process';

console.log(`Report signal: ${report.signal}`);
```

```
const { report } = require('process');

console.log(`Report signal: ${report.signal}`);
```

`process.report.writeReport([filename][, err])`

- `filename` {string} Name of the file where the report is written. This should be a relative path, that will be appended to the directory specified in `process.report.directory`, or the current working directory of the Node.js process, if unspecified.
- `err` {Error} A custom error used for reporting the JavaScript stack.
- Returns: {string} Returns the filename of the generated report.

Writes a diagnostic report to a file. If `filename` is not provided, the default filename includes the date, time, PID, and a sequence number. The report's JavaScript stack trace is taken from `err`, if present.

```
import { report } from 'process';

report.writeReport();
```

```
const { report } = require('process');

report.writeReport();
```

Additional documentation is available in the report documentation.

`process.resourceUsage()`

- Returns: {Object} the resource usage for the current process. All of these values come from the `uv_getrusage` call which returns a `uv_rusage_t` struct.
  - `userCPUTime` {integer} maps to `ru_utime` computed in microseconds. It is the same value as `process.cpuUsage().user`.
  - `systemCPUTime` {integer} maps to `ru_stime` computed in microseconds. It is the same value as `process.cpuUsage().system`.
  - `maxRSS` {integer} maps to `ru_maxrss` which is the maximum resident set size used in kilobytes.

- `sharedMemorySize` {integer} maps to `ru_ixrss` but is not supported by any platform.
- `unsharedDataSize` {integer} maps to `ru_idrss` but is not supported by any platform.
- `unsharedStackSize` {integer} maps to `ru_isrss` but is not supported by any platform.
- `minorPageFault` {integer} maps to `ru_minflt` which is the number of minor page faults for the process, see this article for more details.
- `majorPageFault` {integer} maps to `ru_majflt` which is the number of major page faults for the process, see this article for more details. This field is not supported on Windows.
- `swappedOut` {integer} maps to `ru_nswap` but is not supported by any platform.
- `fsRead` {integer} maps to `ru_inblock` which is the number of times the file system had to perform input.
- `fsWrite` {integer} maps to `ru_oublock` which is the number of times the file system had to perform output.
- `ipcSent` {integer} maps to `ru_msgsnd` but is not supported by any platform.
- `ipcReceived` {integer} maps to `ru_msgrcv` but is not supported by any platform.
- `signalsCount` {integer} maps to `ru_nsignals` but is not supported by any platform.
- `voluntaryContextSwitches` {integer} maps to `ru_nvcsw` which is the number of times a CPU context switch resulted due to a process voluntarily giving up the processor before its time slice was completed (usually to await availability of a resource). This field is not supported on Windows.
- `involuntaryContextSwitches` {integer} maps to `ru_nivcsw` which is the number of times a CPU context switch resulted due to a higher priority process becoming runnable or because the current process exceeded its time slice. This field is not supported on Windows.

```
import { resourceUsage } from 'process';

console.log(resourceUsage());
/*
  Will output:
  {
    userCPUTime: 82872,
    systemCPUTime: 4143,
    maxRSS: 33164,
    sharedMemorySize: 0,
    unsharedDataSize: 0,
    unsharedStackSize: 0,
    minorPageFault: 2469,
```

```
    majorPageFault: 0,
    swappedOut: 0,
    fsRead: 0,
    fsWrite: 8,
    ipcSent: 0,
    ipcReceived: 0,
    signalsCount: 0,
    voluntaryContextSwitches: 79,
    involuntaryContextSwitches: 1
  }
*/

const { resourceUsage } = require('process');

console.log(resourceUsage());
/*
  Will output:
  {
    userCPUTime: 82872,
    systemCPUTime: 4143,
    maxRSS: 33164,
    sharedMemorySize: 0,
    unsharedDataSize: 0,
    unsharedStackSize: 0,
    minorPageFault: 2469,
    majorPageFault: 0,
    swappedOut: 0,
    fsRead: 0,
    fsWrite: 8,
    ipcSent: 0,
    ipcReceived: 0,
    signalsCount: 0,
    voluntaryContextSwitches: 79,
    involuntaryContextSwitches: 1
  }
*/
```

`process.send(message[, sendHandle[, options]][, callback])`

- `message` {Object}
- `sendHandle` {net.Server|net.Socket}
- `options` {Object} used to parameterize the sending of certain types of handles.`options` supports the following properties:
  - `keepOpen` {boolean} A value that can be used when passing instances of `net.Socket`. When `true`, the socket is kept open in the sending process. **Default:** `false`.

- `callback` {Function}
- Returns: {boolean}

If Node.js is spawned with an IPC channel, the `process.send()` method can be used to send messages to the parent process. Messages will be received as a `'message'` event on the parent's `ChildProcess` object.

If Node.js was not spawned with an IPC channel, `process.send` will be `undefined`.

The message goes through serialization and parsing. The resulting message might not be the same as what is originally sent.

`process.setegid(id)`

- `id` {string|number} A group name or ID

The `process.setegid()` method sets the effective group identity of the process. (See setegid(2).) The `id` can be passed as either a numeric ID or a group name string. If a group name is specified, this method blocks while resolving the associated a numeric ID.

```
import process from 'process';

if (process.getegid && process.setegid) {
  console.log(`Current gid: ${process.getegid()}`);
  try {
    process.setegid(501);
    console.log(`New gid: ${process.getegid()}`);
  } catch (err) {
    console.log(`Failed to set gid: ${err}`);
  }
}
```

```
const process = require('process');

if (process.getegid && process.setegid) {
  console.log(`Current gid: ${process.getegid()}`);
  try {
    process.setegid(501);
    console.log(`New gid: ${process.getegid()}`);
  } catch (err) {
    console.log(`Failed to set gid: ${err}`);
  }
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android). This feature is not available in `Worker` threads.

process.seteuid(id)

- `id` {string|number} A user name or ID

The `process.seteuid()` method sets the effective user identity of the process. (See seteuid(2).) The `id` can be passed as either a numeric ID or a username string. If a username is specified, the method blocks while resolving the associated numeric ID.

```
import process from 'process';

if (process.geteuid && process.seteuid) {
  console.log(`Current uid: ${process.geteuid()}`);
  try {
    process.seteuid(501);
    console.log(`New uid: ${process.geteuid()}`);
  } catch (err) {
    console.log(`Failed to set uid: ${err}`);
  }
}
```

```
const process = require('process');

if (process.geteuid && process.seteuid) {
  console.log(`Current uid: ${process.geteuid()}`);
  try {
    process.seteuid(501);
    console.log(`New uid: ${process.geteuid()}`);
  } catch (err) {
    console.log(`Failed to set uid: ${err}`);
  }
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android). This feature is not available in `Worker` threads.

process.setgid(id)

- `id` {string|number} The group name or ID

The `process.setgid()` method sets the group identity of the process. (See setgid(2).) The `id` can be passed as either a numeric ID or a group name string. If a group name is specified, this method blocks while resolving the associated numeric ID.

```
import process from 'process';

if (process.getgid && process.setgid) {
  console.log(`Current gid: ${process.getgid()}`);
```

```
  try {
    process.setgid(501);
    console.log(`New gid: ${process.getgid()}`);
  } catch (err) {
    console.log(`Failed to set gid: ${err}`);
  }
}

const process = require('process');

if (process.getgid && process.setgid) {
  console.log(`Current gid: ${process.getgid()}`);
  try {
    process.setgid(501);
    console.log(`New gid: ${process.getgid()}`);
  } catch (err) {
    console.log(`Failed to set gid: ${err}`);
  }
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android). This feature is not available in `Worker` threads.

`process.setgroups(groups)`

- `groups` {integer[]}

The `process.setgroups()` method sets the supplementary group IDs for the Node.js process. This is a privileged operation that requires the Node.js process to have `root` or the `CAP_SETGID` capability.

The `groups` array can contain numeric group IDs, group names, or both.

```
import process from 'process';

if (process.getgroups && process.setgroups) {
  try {
    process.setgroups([501]);
    console.log(process.getgroups()); // new groups
  } catch (err) {
    console.log(`Failed to set groups: ${err}`);
  }
}

const process = require('process');

if (process.getgroups && process.setgroups) {
  try {
    process.setgroups([501]);
```

```
    console.log(process.getgroups()); // new groups
  } catch (err) {
    console.log(`Failed to set groups: ${err}`);
  }
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android). This feature is not available in `Worker` threads.

process.setuid(id)

- `id` {integer | string}

The `process.setuid(id)` method sets the user identity of the process. (See setuid(2).) The `id` can be passed as either a numeric ID or a username string. If a username is specified, the method blocks while resolving the associated numeric ID.

```
import process from 'process';

if (process.getuid && process.setuid) {
  console.log(`Current uid: ${process.getuid()}`);
  try {
    process.setuid(501);
    console.log(`New uid: ${process.getuid()}`);
  } catch (err) {
    console.log(`Failed to set uid: ${err}`);
  }
}
```

```
const process = require('process');

if (process.getuid && process.setuid) {
  console.log(`Current uid: ${process.getuid()}`);
  try {
    process.setuid(501);
    console.log(`New uid: ${process.getuid()}`);
  } catch (err) {
    console.log(`Failed to set uid: ${err}`);
  }
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android). This feature is not available in `Worker` threads.

process.setSourceMapsEnabled(val)

Stability: 1 - Experimental

- `val` {boolean}

This function enables or disables the Source Map v3 support for stack traces.

It provides same features as launching Node.js process with commandline options `--enable-source-maps`.

Only source maps in JavaScript files that are loaded after source maps has been enabled will be parsed and loaded.

`process.setUncaughtExceptionCaptureCallback(fn)`

- `fn` {Function|null}

The `process.setUncaughtExceptionCaptureCallback()` function sets a function that will be invoked when an uncaught exception occurs, which will receive the exception value itself as its first argument.

If such a function is set, the `'uncaughtException'` event will not be emitted. If `--abort-on-uncaught-exception` was passed from the command line or set through `v8.setFlagsFromString()`, the process will not abort. Actions configured to take place on exceptions such as report generations will be affected too

To unset the capture function, `process.setUncaughtExceptionCaptureCallback(null)` may be used. Calling this method with a non-`null` argument while another capture function is set will throw an error.

Using this function is mutually exclusive with using the deprecated `domain` built-in module.

`process.stderr`

- {Stream}

The `process.stderr` property returns a stream connected to `stderr` (fd `2`). It is a `net.Socket` (which is a Duplex stream) unless fd `2` refers to a file, in which case it is a Writable stream.

`process.stderr` differs from other Node.js streams in important ways. See note on process I/O for more information.

`process.stderr.fd`

- {number}

This property refers to the value of underlying file descriptor of `process.stderr`. The value is fixed at `2`. In `Worker` threads, this field does not exist.

process.stdin

- {Stream}

The `process.stdin` property returns a stream connected to `stdin` (fd `0`). It is a `net.Socket` (which is a Duplex stream) unless fd `0` refers to a file, in which case it is a Readable stream.

For details of how to read from `stdin` see `readable.read()`.

As a Duplex stream, `process.stdin` can also be used in "old" mode that is compatible with scripts written for Node.js prior to v0.10. For more information see Stream compatibility.

In "old" streams mode the `stdin` stream is paused by default, so one must call `process.stdin.resume()` to read from it. Note also that calling `process.stdin.resume()` itself would switch stream to "old" mode.

process.stdin.fd

- {number}

This property refers to the value of underlying file descriptor of `process.stdin`. The value is fixed at `0`. In `Worker` threads, this field does not exist.

process.stdout

- {Stream}

The `process.stdout` property returns a stream connected to `stdout` (fd `1`). It is a `net.Socket` (which is a Duplex stream) unless fd `1` refers to a file, in which case it is a Writable stream.

For example, to copy `process.stdin` to `process.stdout`:

```
import { stdin, stdout } from 'process';

stdin.pipe(stdout);
```

```
const { stdin, stdout } = require('process');

stdin.pipe(stdout);
```

`process.stdout` differs from other Node.js streams in important ways. See note on process I/O for more information.

process.stdout.fd

- {number}

This property refers to the value of underlying file descriptor of `process.stdout`. The value is fixed at `1`. In `Worker` threads, this field does not exist.

A note on process I/O

`process.stdout` and `process.stderr` differ from other Node.js streams in important ways:

1. They are used internally by `console.log()` and `console.error()`, respectively.
2. Writes may be synchronous depending on what the stream is connected to and whether the system is Windows or POSIX:
   - Files: *synchronous* on Windows and POSIX
   - TTYs (Terminals): *asynchronous* on Windows, *synchronous* on POSIX
   - Pipes (and sockets): *synchronous* on Windows, *asynchronous* on POSIX

These behaviors are partly for historical reasons, as changing them would create backward incompatibility, but they are also expected by some users.

Synchronous writes avoid problems such as output written with `console.log()` or `console.error()` being unexpectedly interleaved, or not written at all if `process.exit()` is called before an asynchronous write completes. See `process.exit()` for more information.

Warning: Synchronous writes block the event loop until the write has completed. This can be near instantaneous in the case of output to a file, but under high system load, pipes that are not being read at the receiving end, or with slow terminals or file systems, it's possible for the event loop to be blocked often enough and long enough to have severe negative performance impacts. This may not be a problem when writing to an interactive terminal session, but consider this particularly careful when doing production logging to the process output streams.

To check if a stream is connected to a TTY context, check the `isTTY` property.

For instance:

```
$ node -p "Boolean(process.stdin.isTTY)"
true
$ echo "foo" | node -p "Boolean(process.stdin.isTTY)"
false
$ node -p "Boolean(process.stdout.isTTY)"
true
$ node -p "Boolean(process.stdout.isTTY)" | cat
false
```

See the TTY documentation for more information.

`process.throwDeprecation`

- {boolean}

The initial value of `process.throwDeprecation` indicates whether the `--throw-deprecation` flag is set on the current Node.js process. `process.throwDeprecation` is mutable, so whether or not deprecation warnings result in errors may be altered at runtime. See the documentation for the `'warning'` event and the `emitWarning()` method for more information.

```
$ node --throw-deprecation -p "process.throwDeprecation"
true
$ node -p "process.throwDeprecation"
undefined
$ node
> process.emitWarning('test', 'DeprecationWarning');
undefined
> (node:26598) DeprecationWarning: test
> process.throwDeprecation = true;
true
> process.emitWarning('test', 'DeprecationWarning');
Thrown:
[DeprecationWarning: test] { name: 'DeprecationWarning' }
```

`process.title`

- {string}

The `process.title` property returns the current process title (i.e. returns the current value of `ps`). Assigning a new value to `process.title` modifies the current value of `ps`.

When a new value is assigned, different platforms will impose different maximum length restrictions on the title. Usually such restrictions are quite limited. For instance, on Linux and macOS, `process.title` is limited to the size of the binary name plus the length of the command-line arguments because setting the `process.title` overwrites the `argv` memory of the process. Node.js v0.8 allowed for longer process title strings by also overwriting the `environ` memory but that was potentially insecure and confusing in some (rather obscure) cases.

Assigning a value to `process.title` might not result in an accurate label within process manager applications such as macOS Activity Monitor or Windows Services Manager.

`process.traceDeprecation`

- {boolean}

The `process.traceDeprecation` property indicates whether the `--trace-deprecation` flag is set on the current Node.js process. See the documentation for the `'warning'` event and the `emitWarning()` method for more information about this flag's behavior.

process.umask()

> Stability: 0 - Deprecated. Calling `process.umask()` with no argument causes the process-wide umask to be written twice. This introduces a race condition between threads, and is a potential security vulnerability. There is no safe, cross-platform alternative API.

`process.umask()` returns the Node.js process's file mode creation mask. Child processes inherit the mask from the parent process.

process.umask(mask)

- `mask` {string|integer}

`process.umask(mask)` sets the Node.js process's file mode creation mask. Child processes inherit the mask from the parent process. Returns the previous mask.

```
import { umask } from 'process';

const newmask = 0o022;
const oldmask = umask(newmask);
console.log(
  `Changed umask from ${oldmask.toString(8)} to ${newmask.toString(8)}`
);
```

```
const { umask } = require('process');

const newmask = 0o022;
const oldmask = umask(newmask);
console.log(
  `Changed umask from ${oldmask.toString(8)} to ${newmask.toString(8)}`
);
```

In `Worker` threads, `process.umask(mask)` will throw an exception.

process.uptime()

- Returns: {number}

The `process.uptime()` method returns the number of seconds the current Node.js process has been running.

The return value includes fractions of a second. Use `Math.floor()` to get whole seconds.

process.version

- {string}

The `process.version` property contains the Node.js version string.

```
import { version } from 'process';

console.log(`Version: ${version}`);
// Version: v14.8.0
```

```
const { version } = require('process');

console.log(`Version: ${version}`);
// Version: v14.8.0
```

To get the version string without the prepended *v*, use `process.versions.node`.

`process.versions`

- {Object}

The `process.versions` property returns an object listing the version strings of Node.js and its dependencies. `process.versions.modules` indicates the current ABI version, which is increased whenever a C++ API changes. Node.js will refuse to load modules that were compiled against a different module ABI version.

```
import { versions } from 'process';

console.log(versions);
```

```
const { versions } = require('process');

console.log(versions);
```

Will generate an object similar to:

```
{ node: '11.13.0',
  v8: '7.0.276.38-node.18',
  uv: '1.27.0',
  zlib: '1.2.11',
  brotli: '1.0.7',
  ares: '1.15.0',
  modules: '67',
  nghttp2: '1.34.0',
  napi: '4',
  llhttp: '1.1.1',
  openssl: '1.1.1b',
  cldr: '34.0',
  icu: '63.1',
  tz: '2018e',
  unicode: '11.0' }
```

Exit codes

Node.js will normally exit with a `0` status code when no more async operations are pending. The following status codes are used in other cases:

- `1` **Uncaught Fatal Exception**: There was an uncaught exception, and it was not handled by a domain or an `'uncaughtException'` event handler.
- `2`: Unused (reserved by Bash for builtin misuse)
- `3` **Internal JavaScript Parse Error**: The JavaScript source code internal in the Node.js bootstrapping process caused a parse error. This is extremely rare, and generally can only happen during development of Node.js itself.
- `4` **Internal JavaScript Evaluation Failure**: The JavaScript source code internal in the Node.js bootstrapping process failed to return a function value when evaluated. This is extremely rare, and generally can only happen during development of Node.js itself.
- `5` **Fatal Error**: There was a fatal unrecoverable error in V8. Typically a message will be printed to stderr with the prefix `FATAL ERROR`.
- `6` **Non-function Internal Exception Handler**: There was an uncaught exception, but the internal fatal exception handler function was somehow set to a non-function, and could not be called.
- `7` **Internal Exception Handler Run-Time Failure**: There was an uncaught exception, and the internal fatal exception handler function itself threw an error while attempting to handle it. This can happen, for example, if an `'uncaughtException'` or `domain.on('error')` handler throws an error.
- `8`: Unused. In previous versions of Node.js, exit code 8 sometimes indicated an uncaught exception.
- `9` **Invalid Argument**: Either an unknown option was specified, or an option requiring a value was provided without a value.
- `10` **Internal JavaScript Run-Time Failure**: The JavaScript source code internal in the Node.js bootstrapping process threw an error when the bootstrapping function was called. This is extremely rare, and generally can only happen during development of Node.js itself.
- `12` **Invalid Debug Argument**: The `--inspect` and/or `--inspect-brk` options were set, but the port number chosen was invalid or unavailable.
- `13` **Unfinished Top-Level Await**: `await` was used outside of a function in the top-level code, but the passed `Promise` never resolved.
- `>128` **Signal Exits**: If Node.js receives a fatal signal such as `SIGKILL` or `SIGHUP`, then its exit code will be `128` plus the value of the signal code. This is a standard POSIX practice, since exit codes are defined to be 7-bit integers, and signal exits set the high-order bit, and then contain the value of the signal code. For example, signal `SIGABRT` has value `6`, so the expected exit code will be `128 + 6`, or `134`.