declare module 'process' {

import \* as tty from 'node:tty';

import { Worker } from 'node:worker\_threads';

global {

var process: NodeJS.Process;

namespace NodeJS {

// this namespace merge is here because these are specifically used

// as the type for process.stdin, process.stdout, and process.stderr.

// they can't live in tty.d.ts because we need to disambiguate the imported name.

interface ReadStream extends tty.ReadStream {}

interface WriteStream extends tty.WriteStream {}

interface MemoryUsageFn {

/\*\*

\* The `process.memoryUsage()` method iterate over each page to gather informations about memory

\* usage which can be slow depending on the program memory allocations.

\*/

(): MemoryUsage;

/\*\*

\* method returns an integer representing the Resident Set Size (RSS) in bytes.

\*/

rss(): number;

}

interface MemoryUsage {

rss: number;

heapTotal: number;

heapUsed: number;

external: number;

arrayBuffers: number;

}

interface CpuUsage {

user: number;

system: number;

}

interface ProcessRelease {

name: string;

sourceUrl?: string | undefined;

headersUrl?: string | undefined;

libUrl?: string | undefined;

lts?: string | undefined;

}

interface ProcessVersions extends Dict<string> {

http\_parser: string;

node: string;

v8: string;

ares: string;

uv: string;

zlib: string;

modules: string;

openssl: string;

}

type Platform = 'aix' | 'android' | 'darwin' | 'freebsd' | 'haiku' | 'linux' | 'openbsd' | 'sunos' | 'win32' | 'cygwin' | 'netbsd';

type Signals =

| 'SIGABRT'

| 'SIGALRM'

| 'SIGBUS'

| 'SIGCHLD'

| 'SIGCONT'

| 'SIGFPE'

| 'SIGHUP'

| 'SIGILL'

| 'SIGINT'

| 'SIGIO'

| 'SIGIOT'

| 'SIGKILL'

| 'SIGPIPE'

| 'SIGPOLL'

| 'SIGPROF'

| 'SIGPWR'

| 'SIGQUIT'

| 'SIGSEGV'

| 'SIGSTKFLT'

| 'SIGSTOP'

| 'SIGSYS'

| 'SIGTERM'

| 'SIGTRAP'

| 'SIGTSTP'

| 'SIGTTIN'

| 'SIGTTOU'

| 'SIGUNUSED'

| 'SIGURG'

| 'SIGUSR1'

| 'SIGUSR2'

| 'SIGVTALRM'

| 'SIGWINCH'

| 'SIGXCPU'

| 'SIGXFSZ'

| 'SIGBREAK'

| 'SIGLOST'

| 'SIGINFO';

type UncaughtExceptionOrigin = 'uncaughtException' | 'unhandledRejection';

type MultipleResolveType = 'resolve' | 'reject';

type BeforeExitListener = (code: number) => void;

type DisconnectListener = () => void;

type ExitListener = (code: number) => void;

type RejectionHandledListener = (promise: Promise<unknown>) => void;

type UncaughtExceptionListener = (error: Error, origin: UncaughtExceptionOrigin) => void;

/\*\*

\* Most of the time the unhandledRejection will be an Error, but this should not be relied upon

\* as \*anything\* can be thrown/rejected, it is therefore unsafe to assume the the value is an Error.

\*/

type UnhandledRejectionListener = (reason: unknown, promise: Promise<unknown>) => void;

type WarningListener = (warning: Error) => void;

type MessageListener = (message: unknown, sendHandle: unknown) => void;

type SignalsListener = (signal: Signals) => void;

type MultipleResolveListener = (type: MultipleResolveType, promise: Promise<unknown>, value: unknown) => void;

type WorkerListener = (worker: Worker) => void;

interface Socket extends ReadWriteStream {

isTTY?: true | undefined;

}

// Alias for compatibility

interface ProcessEnv extends Dict<string> {

/\*\*

\* Can be used to change the default timezone at runtime

\*/

TZ?: string;

}

interface HRTime {

(time?: [number, number]): [number, number];

bigint(): bigint;

}

interface ProcessReport {

/\*\*

\* Directory where the report is written.

\* working directory of the Node.js process.

\* @default '' indicating that reports are written to the current

\*/

directory: string;

/\*\*

\* Filename where the report is written.

\* The default value is the empty string.

\* @default '' the output filename will be comprised of a timestamp,

\* PID, and sequence number.

\*/

filename: string;

/\*\*

\* Returns a JSON-formatted diagnostic report for the running process.

\* The report's JavaScript stack trace is taken from err, if present.

\*/

getReport(err?: Error): string;

/\*\*

\* If true, a diagnostic report is generated on fatal errors,

\* such as out of memory errors or failed C++ assertions.

\* @default false

\*/

reportOnFatalError: boolean;

/\*\*

\* If true, a diagnostic report is generated when the process

\* receives the signal specified by process.report.signal.

\* @defaul false

\*/

reportOnSignal: boolean;

/\*\*

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

\* @default false

\*/

reportOnUncaughtException: boolean;

/\*\*

\* The signal used to trigger the creation of a diagnostic report.

\* @default 'SIGUSR2'

\*/

signal: Signals;

/\*\*

\* 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.

\*

\* @param fileName 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.

\* @param error A custom error used for reporting the JavaScript stack.

\* @return Filename of the generated report.

\*/

writeReport(fileName?: string): string;

writeReport(error?: Error): string;

writeReport(fileName?: string, err?: Error): string;

}

interface ResourceUsage {

fsRead: number;

fsWrite: number;

involuntaryContextSwitches: number;

ipcReceived: number;

ipcSent: number;

majorPageFault: number;

maxRSS: number;

minorPageFault: number;

sharedMemorySize: number;

signalsCount: number;

swappedOut: number;

systemCPUTime: number;

unsharedDataSize: number;

unsharedStackSize: number;

userCPUTime: number;

voluntaryContextSwitches: number;

}

interface EmitWarningOptions {

/\*\*

\* When `warning` is a `string`, `type` is the name to use for the \_type\_ of warning being emitted.

\*

\* @default 'Warning'

\*/

type?: string | undefined;

/\*\*

\* A unique identifier for the warning instance being emitted.

\*/

code?: string | undefined;

/\*\*

\* When `warning` is a `string`, `ctor` is an optional function used to limit the generated stack trace.

\*

\* @default process.emitWarning

\*/

ctor?: Function | undefined;

/\*\*

\* Additional text to include with the error.

\*/

detail?: string | undefined;

}

interface ProcessConfig {

readonly target\_defaults: {

readonly cflags: any[];

readonly default\_configuration: string;

readonly defines: string[];

readonly include\_dirs: string[];

readonly libraries: string[];

};

readonly variables: {

readonly clang: number;

readonly host\_arch: string;

readonly node\_install\_npm: boolean;

readonly node\_install\_waf: boolean;

readonly node\_prefix: string;

readonly node\_shared\_openssl: boolean;

readonly node\_shared\_v8: boolean;

readonly node\_shared\_zlib: boolean;

readonly node\_use\_dtrace: boolean;

readonly node\_use\_etw: boolean;

readonly node\_use\_openssl: boolean;

readonly target\_arch: string;

readonly v8\_no\_strict\_aliasing: number;

readonly v8\_use\_snapshot: boolean;

readonly visibility: string;

};

}

interface Process extends EventEmitter {

/\*\*

\* 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`:

\*

\* ```js

\* import { stdin, stdout } from 'process';

\*

\* stdin.pipe(stdout);

\* ```

\*

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

\*/

stdout: WriteStream & {

fd: 1;

};

/\*\*

\* 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.

\*/

stderr: WriteStream & {

fd: 2;

};

/\*\*

\* 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.

\*/

stdin: ReadStream & {

fd: 0;

};

openStdin(): Socket;

/\*\*

\* 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 {@link 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`:

\*

\* ```js

\* import { argv } from 'process';

\*

\* // print process.argv

\* argv.forEach((val, index) => {

\* console.log(`${index}: ${val}`);

\* });

\* ```

\*

\* Launching the Node.js process as:

\*

\* ```console

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

\* ```

\*

\* Would generate the output:

\*

\* ```text

\* 0: /usr/local/bin/node

\* 1: /Users/mjr/work/node/process-args.js

\* 2: one

\* 3: two=three

\* 4: four

\* ```

\* @since v0.1.27

\*/

argv: string[];

/\*\*

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

\*

\* ```console

\* $ bash -c 'exec -a customArgv0 ./node'

\* > process.argv[0]

\* '/Volumes/code/external/node/out/Release/node'

\* > process.argv0

\* 'customArgv0'

\* ```

\* @since v6.4.0

\*/

argv0: 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 {@link 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.

\*

\* ```console

\* $ node --harmony script.js --version

\* ```

\*

\* Results in `process.execArgv`:

\*

\* ```js

\* ['--harmony']

\* ```

\*

\* And `process.argv`:

\*

\* ```js

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

\* ```

\*

\* Refer to `Worker constructor` for the detailed behavior of worker

\* threads with this property.

\* @since v0.7.7

\*/

execArgv: string[];

/\*\*

\* The `process.execPath` property returns the absolute pathname of the executable

\* that started the Node.js process. Symbolic links, if any, are resolved.

\*

\* ```js

\* '/usr/local/bin/node'

\* ```

\* @since v0.1.100

\*/

execPath: string;

/\*\*

\* 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.

\* @since v0.7.0

\*/

abort(): never;

/\*\*

\* 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).

\*

\* ```js

\* import { chdir, cwd } from '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.

\* @since v0.1.17

\*/

chdir(directory: string): void;

/\*\*

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

\* process.

\*

\* ```js

\* import { cwd } from 'process';

\*

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

\* ```

\* @since v0.1.8

\*/

cwd(): string;

/\*\*

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

\*

\* ```js

\* import process from 'process';

\*

\* process.debugPort = 5858;

\* ```

\* @since v0.7.2

\*/

debugPort: number;

/\*\*

\* 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.

\*

\* ```js

\* 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

\* ```

\*

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

\*

\* ```js

\* 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'

\* });

\* ```

\*

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

\* @since v8.0.0

\* @param warning The warning to emit.

\*/

emitWarning(warning: string | Error, ctor?: Function): void;

emitWarning(warning: string | Error, type?: string, ctor?: Function): void;

emitWarning(warning: string | Error, type?: string, code?: string, ctor?: Function): void;

emitWarning(warning: string | Error, options?: EmitWarningOptions): void;

/\*\*

\* The `process.env` property returns an object containing the user environment.

\* See [`environ(7)`](http://man7.org/linux/man-pages/man7/environ.7.html).

\*

\* An example of this object looks like:

\*

\* ```js

\* {

\* 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:

\*

\* ```console

\* $ node -e 'process.env.foo = "bar"' &#x26;&#x26; echo $foo

\* ```

\*

\* While the following will:

\*

\* ```js

\* import { env } from '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.

\*

\* ```js

\* import { env } from '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`.

\*

\* ```js

\* import { env } from 'process';

\*

\* env.TEST = 1;

\* delete env.TEST;

\* console.log(env.TEST);

\* // => undefined

\* ```

\*

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

\*

\* ```js

\* import { env } from '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.

\* @since v0.1.27

\*/

env: ProcessEnv;

/\*\*

\* 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:

\*

\* ```js

\* import { exit } from '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:

\*

\* ```js

\* import { exit } from '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:

\*

\* ```js

\* import process from '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.

\* @since v0.1.13

\* @param [code=0] The exit code.

\*/

exit(code?: number): never;

/\*\*

\* A number which will be the process exit code, when the process either

\* exits gracefully, or is exited via {@link exit} without specifying

\* a code.

\*

\* Specifying a code to {@link exit} will override any

\* previous setting of `process.exitCode`.

\* @since v0.11.8

\*/

exitCode?: number | undefined;

/\*\*

\* The `process.getgid()` method returns the numerical group identity of the

\* process. (See [`getgid(2)`](http://man7.org/linux/man-pages/man2/getgid.2.html).)

\*

\* ```js

\* import process from 'process';

\*

\* if (process.getgid) {

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

\* }

\* ```

\*

\* This function is only available on POSIX platforms (i.e. not Windows or

\* Android).

\* @since v0.1.31

\*/

getgid(): number;

/\*\*

\* The `process.setgid()` method sets the group identity of the process. (See [`setgid(2)`](http://man7.org/linux/man-pages/man2/setgid.2.html).) 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.

\*

\* ```js

\* import process from 'process';

\*

\* if (process.getgid &#x26;&#x26; 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.

\* @since v0.1.31

\* @param id The group name or ID

\*/

setgid(id: number | string): void;

/\*\*

\* The `process.getuid()` method returns the numeric user identity of the process.

\* (See [`getuid(2)`](http://man7.org/linux/man-pages/man2/getuid.2.html).)

\*

\* ```js

\* import process from 'process';

\*

\* if (process.getuid) {

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

\* }

\* ```

\*

\* This function is only available on POSIX platforms (i.e. not Windows or

\* Android).

\* @since v0.1.28

\*/

getuid(): number;

/\*\*

\* The `process.setuid(id)` method sets the user identity of the process. (See [`setuid(2)`](http://man7.org/linux/man-pages/man2/setuid.2.html).) 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.

\*

\* ```js

\* import process from 'process';

\*

\* if (process.getuid &#x26;&#x26; 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.

\* @since v0.1.28

\*/

setuid(id: number | string): void;

/\*\*

\* The `process.geteuid()` method returns the numerical effective user identity of

\* the process. (See [`geteuid(2)`](http://man7.org/linux/man-pages/man2/geteuid.2.html).)

\*

\* ```js

\* import process from 'process';

\*

\* if (process.geteuid) {

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

\* }

\* ```

\*

\* This function is only available on POSIX platforms (i.e. not Windows or

\* Android).

\* @since v2.0.0

\*/

geteuid(): number;

/\*\*

\* The `process.seteuid()` method sets the effective user identity of the process.

\* (See [`seteuid(2)`](http://man7.org/linux/man-pages/man2/seteuid.2.html).) 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.

\*

\* ```js

\* import process from 'process';

\*

\* if (process.geteuid &#x26;&#x26; 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.

\* @since v2.0.0

\* @param id A user name or ID

\*/

seteuid(id: number | string): void;

/\*\*

\* The `process.getegid()` method returns the numerical effective group identity

\* of the Node.js process. (See [`getegid(2)`](http://man7.org/linux/man-pages/man2/getegid.2.html).)

\*

\* ```js

\* import process from 'process';

\*

\* if (process.getegid) {

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

\* }

\* ```

\*

\* This function is only available on POSIX platforms (i.e. not Windows or

\* Android).

\* @since v2.0.0

\*/

getegid(): number;

/\*\*

\* The `process.setegid()` method sets the effective group identity of the process.

\* (See [`setegid(2)`](http://man7.org/linux/man-pages/man2/setegid.2.html).) 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.

\*

\* ```js

\* import process from 'process';

\*

\* if (process.getegid &#x26;&#x26; 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.

\* @since v2.0.0

\* @param id A group name or ID

\*/

setegid(id: number | string): void;

/\*\*

\* 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.

\*

\* ```js

\* import process from '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).

\* @since v0.9.4

\*/

getgroups(): number[];

/\*\*

\* 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.

\*

\* ```js

\* import process from 'process';

\*

\* if (process.getgroups &#x26;&#x26; 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.

\* @since v0.9.4

\*/

setgroups(groups: ReadonlyArray<string | number>): void;

/\*\*

\* 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.

\* @since v9.3.0

\*/

setUncaughtExceptionCaptureCallback(cb: ((err: Error) => void) | null): void;

/\*\*

\* Indicates whether a callback has been set using {@link setUncaughtExceptionCaptureCallback}.

\* @since v9.3.0

\*/

hasUncaughtExceptionCaptureCallback(): boolean;

/\*\*

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

\*

\* ```js

\* import { version } from 'process';

\*

\* console.log(`Version: ${version}`);

\* // Version: v14.8.0

\* ```

\*

\* To get the version string without the prepended \_v\_, use`process.versions.node`.

\* @since v0.1.3

\*/

readonly version: string;

/\*\*

\* 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.

\*

\* ```js

\* import { versions } from 'process';

\*

\* console.log(versions);

\* ```

\*

\* Will generate an object similar to:

\*

\* ```console

\* { 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' }

\* ```

\* @since v0.2.0

\*/

readonly versions: ProcessVersions;

/\*\*

\* 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:

\*

\* ```js

\* {

\* 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.

\* @since v0.7.7

\*/

readonly config: ProcessConfig;

/\*\*

\* 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)`](http://man7.org/linux/man-pages/man2/kill.2.html) 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.

\*

\* ```js

\* 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');

\* ```

\*

\* When `SIGUSR1` is received by a Node.js process, Node.js will start the

\* debugger. See `Signal Events`.

\* @since v0.0.6

\* @param pid A process ID

\* @param [signal='SIGTERM'] The signal to send, either as a string or number.

\*/

kill(pid: number, signal?: string | number): true;

/\*\*

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

\*

\* ```js

\* import { pid } from 'process';

\*

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

\* ```

\* @since v0.1.15

\*/

readonly pid: number;

/\*\*

\* The `process.ppid` property returns the PID of the parent of the

\* current process.

\*

\* ```js

\* import { ppid } from 'process';

\*

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

\* ```

\* @since v9.2.0, v8.10.0, v6.13.0

\*/

readonly ppid: number;

/\*\*

\* 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.

\* @since v0.1.104

\*/

title: 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'`, `'x32'`, and `'x64'`.

\*

\* ```js

\* import { arch } from 'process';

\*

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

\* ```

\* @since v0.5.0

\*/

readonly arch: string;

/\*\*

\* The `process.platform` property returns a string identifying the operating

\* system platform on which the Node.js process is running.

\*

\* Currently possible values are:

\*

\* \* `'aix'`

\* \* `'darwin'`

\* \* `'freebsd'`

\* \* `'linux'`

\* \* `'openbsd'`

\* \* `'sunos'`

\* \* `'win32'`

\*

\* ```js

\* import { platform } from '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](https://github.com/nodejs/node/blob/HEAD/BUILDING.md#androidandroid-based-devices-eg-firefox-os).

\* @since v0.1.16

\*/

readonly platform: Platform;

/\*\*

\* 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.

\* @since v0.1.17

\* @deprecated Since v14.0.0 - Use `main` instead.

\*/

mainModule?: Module | undefined;

memoryUsage: MemoryUsageFn;

/\*\*

\* 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.

\*

\* ```js

\* 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 }

\* ```

\* @since v6.1.0

\* @param previousValue A previous return value from calling `process.cpuUsage()`

\*/

cpuUsage(previousValue?: CpuUsage): CpuUsage;

/\*\*

\* `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](https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/#process-nexttick) guide for more background.

\*

\* ```js

\* import { nextTick } from '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:

\*

\* ```js

\* 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.

\* ```

\*

\* It is very important for APIs to be either 100% synchronous or 100%

\* asynchronous. Consider this example:

\*

\* ```js

\* // 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:

\*

\* ```js

\* 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:

\*

\* ```js

\* import { nextTick } from 'process';

\*

\* function definitelyAsync(arg, cb) {

\* if (arg) {

\* nextTick(cb);

\* return;

\* }

\*

\* fs.stat('file', cb);

\* }

\* ```

\* @since v0.1.26

\* @param args Additional arguments to pass when invoking the `callback`

\*/

nextTick(callback: Function, ...args: any[]): void;

/\*\*

\* 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:

\*

\* ```js

\* {

\* 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.

\* @since v3.0.0

\*/

readonly release: ProcessRelease;

features: {

inspector: boolean;

debug: boolean;

uv: boolean;

ipv6: boolean;

tls\_alpn: boolean;

tls\_sni: boolean;

tls\_ocsp: boolean;

tls: boolean;

};

/\*\*

\* `process.umask()` returns the Node.js process's file mode creation mask. Child

\* processes inherit the mask from the parent process.

\* @since v0.1.19

\* @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.

\*/

umask(): number;

/\*\*

\* Can only be set if not in worker thread.

\*/

umask(mask: string | number): 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.

\* @since v0.5.0

\*/

uptime(): number;

hrtime: HRTime;

/\*\*

\* 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.

\* @since v0.5.9

\* @param options used to parameterize the sending of certain types of handles.`options` supports the following properties:

\*/

send?(

message: any,

sendHandle?: any,

options?: {

swallowErrors?: boolean | undefined;

},

callback?: (error: Error | null) => void

): boolean;

/\*\*

\* 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`.

\* @since v0.7.2

\*/

disconnect(): void;

/\*\*

\* 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()`.

\* @since v0.7.2

\*/

connected: boolean;

/\*\*

\* 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:

\*

\* ```js

\* import { allowedNodeEnvironmentFlags } from '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 {@link config}), `process.allowedNodeEnvironmentFlags` will

\* contain what \_would have\_ been allowable.

\* @since v10.10.0

\*/

allowedNodeEnvironmentFlags: ReadonlySet<string>;

/\*\*

\* `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`.

\* @since v11.8.0

\*/

report?: ProcessReport | undefined;

/\*\*

\* ```js

\* 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

\* }

\*

\* ```

\* @since v12.6.0

\* @return the resource usage for the current process. All of these values come from the `uv\_getrusage` call which returns a [`uv\_rusage\_t` struct][uv\_rusage\_t].

\*/

resourceUsage(): ResourceUsage;

/\*\*

\* 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.

\* @since v0.8.0

\*/

traceDeprecation: boolean;

/\* EventEmitter \*/

addListener(event: 'beforeExit', listener: BeforeExitListener): this;

addListener(event: 'disconnect', listener: DisconnectListener): this;

addListener(event: 'exit', listener: ExitListener): this;

addListener(event: 'rejectionHandled', listener: RejectionHandledListener): this;

addListener(event: 'uncaughtException', listener: UncaughtExceptionListener): this;

addListener(event: 'uncaughtExceptionMonitor', listener: UncaughtExceptionListener): this;

addListener(event: 'unhandledRejection', listener: UnhandledRejectionListener): this;

addListener(event: 'warning', listener: WarningListener): this;

addListener(event: 'message', listener: MessageListener): this;

addListener(event: Signals, listener: SignalsListener): this;

addListener(event: 'multipleResolves', listener: MultipleResolveListener): this;

addListener(event: 'worker', listener: WorkerListener): this;

emit(event: 'beforeExit', code: number): boolean;

emit(event: 'disconnect'): boolean;

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

emit(event: 'rejectionHandled', promise: Promise<unknown>): boolean;

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

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

emit(event: 'unhandledRejection', reason: unknown, promise: Promise<unknown>): boolean;

emit(event: 'warning', warning: Error): boolean;

emit(event: 'message', message: unknown, sendHandle: unknown): this;

emit(event: Signals, signal: Signals): boolean;

emit(event: 'multipleResolves', type: MultipleResolveType, promise: Promise<unknown>, value: unknown): this;

emit(event: 'worker', listener: WorkerListener): this;

on(event: 'beforeExit', listener: BeforeExitListener): this;

on(event: 'disconnect', listener: DisconnectListener): this;

on(event: 'exit', listener: ExitListener): this;

on(event: 'rejectionHandled', listener: RejectionHandledListener): this;

on(event: 'uncaughtException', listener: UncaughtExceptionListener): this;

on(event: 'uncaughtExceptionMonitor', listener: UncaughtExceptionListener): this;

on(event: 'unhandledRejection', listener: UnhandledRejectionListener): this;

on(event: 'warning', listener: WarningListener): this;

on(event: 'message', listener: MessageListener): this;

on(event: Signals, listener: SignalsListener): this;

on(event: 'multipleResolves', listener: MultipleResolveListener): this;

on(event: 'worker', listener: WorkerListener): this;

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

once(event: 'beforeExit', listener: BeforeExitListener): this;

once(event: 'disconnect', listener: DisconnectListener): this;

once(event: 'exit', listener: ExitListener): this;

once(event: 'rejectionHandled', listener: RejectionHandledListener): this;

once(event: 'uncaughtException', listener: UncaughtExceptionListener): this;

once(event: 'uncaughtExceptionMonitor', listener: UncaughtExceptionListener): this;

once(event: 'unhandledRejection', listener: UnhandledRejectionListener): this;

once(event: 'warning', listener: WarningListener): this;

once(event: 'message', listener: MessageListener): this;

once(event: Signals, listener: SignalsListener): this;

once(event: 'multipleResolves', listener: MultipleResolveListener): this;

once(event: 'worker', listener: WorkerListener): this;

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

prependListener(event: 'beforeExit', listener: BeforeExitListener): this;

prependListener(event: 'disconnect', listener: DisconnectListener): this;

prependListener(event: 'exit', listener: ExitListener): this;

prependListener(event: 'rejectionHandled', listener: RejectionHandledListener): this;

prependListener(event: 'uncaughtException', listener: UncaughtExceptionListener): this;

prependListener(event: 'uncaughtExceptionMonitor', listener: UncaughtExceptionListener): this;

prependListener(event: 'unhandledRejection', listener: UnhandledRejectionListener): this;

prependListener(event: 'warning', listener: WarningListener): this;

prependListener(event: 'message', listener: MessageListener): this;

prependListener(event: Signals, listener: SignalsListener): this;

prependListener(event: 'multipleResolves', listener: MultipleResolveListener): this;

prependListener(event: 'worker', listener: WorkerListener): this;

prependOnceListener(event: 'beforeExit', listener: BeforeExitListener): this;

prependOnceListener(event: 'disconnect', listener: DisconnectListener): this;

prependOnceListener(event: 'exit', listener: ExitListener): this;

prependOnceListener(event: 'rejectionHandled', listener: RejectionHandledListener): this;

prependOnceListener(event: 'uncaughtException', listener: UncaughtExceptionListener): this;

prependOnceListener(event: 'uncaughtExceptionMonitor', listener: UncaughtExceptionListener): this;

prependOnceListener(event: 'unhandledRejection', listener: UnhandledRejectionListener): this;

prependOnceListener(event: 'warning', listener: WarningListener): this;

prependOnceListener(event: 'message', listener: MessageListener): this;

prependOnceListener(event: Signals, listener: SignalsListener): this;

prependOnceListener(event: 'multipleResolves', listener: MultipleResolveListener): this;

prependOnceListener(event: 'worker', listener: WorkerListener): this;

listeners(event: 'beforeExit'): BeforeExitListener[];

listeners(event: 'disconnect'): DisconnectListener[];

listeners(event: 'exit'): ExitListener[];

listeners(event: 'rejectionHandled'): RejectionHandledListener[];

listeners(event: 'uncaughtException'): UncaughtExceptionListener[];

listeners(event: 'uncaughtExceptionMonitor'): UncaughtExceptionListener[];

listeners(event: 'unhandledRejection'): UnhandledRejectionListener[];

listeners(event: 'warning'): WarningListener[];

listeners(event: 'message'): MessageListener[];

listeners(event: Signals): SignalsListener[];

listeners(event: 'multipleResolves'): MultipleResolveListener[];

listeners(event: 'worker'): WorkerListener[];

}

}

}

export = process;

}

declare module 'node:process' {

import process = require('process');

export = process;

}