/\*\*

\* The `child\_process` module provides the ability to spawn subprocesses in

\* a manner that is similar, but not identical, to [`popen(3)`](http://man7.org/linux/man-pages/man3/popen.3.html). This capability

\* is primarily provided by the {@link spawn} function:

\*

\* ```js

\* const { spawn } = require('child\_process');

\* const ls = spawn('ls', ['-lh', '/usr']);

\*

\* ls.stdout.on('data', (data) => {

\* console.log(`stdout: ${data}`);

\* });

\*

\* ls.stderr.on('data', (data) => {

\* console.error(`stderr: ${data}`);

\* });

\*

\* ls.on('close', (code) => {

\* console.log(`child process exited with code ${code}`);

\* });

\* ```

\*

\* By default, pipes for `stdin`, `stdout`, and `stderr` are established between

\* the parent Node.js process and the spawned subprocess. These pipes have

\* limited (and platform-specific) capacity. If the subprocess writes to

\* stdout in excess of that limit without the output being captured, the

\* subprocess blocks waiting for the pipe buffer to accept more data. This is

\* identical to the behavior of pipes in the shell. Use the `{ stdio: 'ignore' }`option if the output will not be consumed.

\*

\* The command lookup is performed using the `options.env.PATH` environment

\* variable if it is in the `options` object. Otherwise, `process.env.PATH` is

\* used.

\*

\* On Windows, environment variables are case-insensitive. Node.js

\* lexicographically sorts the `env` keys and uses the first one that

\* case-insensitively matches. Only first (in lexicographic order) entry will be

\* passed to the subprocess. This might lead to issues on Windows when passing

\* objects to the `env` option that have multiple variants of the same key, such as`PATH` and `Path`.

\*

\* The {@link spawn} method spawns the child process asynchronously,

\* without blocking the Node.js event loop. The {@link spawnSync} function provides equivalent functionality in a synchronous manner that blocks

\* the event loop until the spawned process either exits or is terminated.

\*

\* For convenience, the `child\_process` module provides a handful of synchronous

\* and asynchronous alternatives to {@link spawn} and {@link spawnSync}. Each of these alternatives are implemented on

\* top of {@link spawn} or {@link spawnSync}.

\*

\* \* {@link exec}: spawns a shell and runs a command within that

\* shell, passing the `stdout` and `stderr` to a callback function when

\* complete.

\* \* {@link execFile}: similar to {@link exec} except

\* that it spawns the command directly without first spawning a shell by

\* default.

\* \* {@link fork}: spawns a new Node.js process and invokes a

\* specified module with an IPC communication channel established that allows

\* sending messages between parent and child.

\* \* {@link execSync}: a synchronous version of {@link exec} that will block the Node.js event loop.

\* \* {@link execFileSync}: a synchronous version of {@link execFile} that will block the Node.js event loop.

\*

\* For certain use cases, such as automating shell scripts, the `synchronous counterparts` may be more convenient. In many cases, however,

\* the synchronous methods can have significant impact on performance due to

\* stalling the event loop while spawned processes complete.

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

\*/

declare module 'child\_process' {

import { ObjectEncodingOptions } from 'node:fs';

import { EventEmitter, Abortable } from 'node:events';

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

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

import { URL } from 'node:url';

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

type SendHandle = net.Socket | net.Server;

/\*\*

\* Instances of the `ChildProcess` represent spawned child processes.

\*

\* Instances of `ChildProcess` are not intended to be created directly. Rather,

\* use the {@link spawn}, {@link exec},{@link execFile}, or {@link fork} methods to create

\* instances of `ChildProcess`.

\* @since v2.2.0

\*/

class ChildProcess extends EventEmitter {

/\*\*

\* A `Writable Stream` that represents the child process's `stdin`.

\*

\* If a child process waits to read all of its input, the child will not continue

\* until this stream has been closed via `end()`.

\*

\* If the child was spawned with `stdio[0]` set to anything other than `'pipe'`,

\* then this will be `null`.

\*

\* `subprocess.stdin` is an alias for `subprocess.stdio[0]`. Both properties will

\* refer to the same value.

\*

\* The `subprocess.stdin` property can be `undefined` if the child process could

\* not be successfully spawned.

\* @since v0.1.90

\*/

stdin: Writable | null;

/\*\*

\* A `Readable Stream` that represents the child process's `stdout`.

\*

\* If the child was spawned with `stdio[1]` set to anything other than `'pipe'`,

\* then this will be `null`.

\*

\* `subprocess.stdout` is an alias for `subprocess.stdio[1]`. Both properties will

\* refer to the same value.

\*

\* ```js

\* const { spawn } = require('child\_process');

\*

\* const subprocess = spawn('ls');

\*

\* subprocess.stdout.on('data', (data) => {

\* console.log(`Received chunk ${data}`);

\* });

\* ```

\*

\* The `subprocess.stdout` property can be `null` if the child process could

\* not be successfully spawned.

\* @since v0.1.90

\*/

stdout: Readable | null;

/\*\*

\* A `Readable Stream` that represents the child process's `stderr`.

\*

\* If the child was spawned with `stdio[2]` set to anything other than `'pipe'`,

\* then this will be `null`.

\*

\* `subprocess.stderr` is an alias for `subprocess.stdio[2]`. Both properties will

\* refer to the same value.

\*

\* The `subprocess.stderr` property can be `null` if the child process could

\* not be successfully spawned.

\* @since v0.1.90

\*/

stderr: Readable | null;

/\*\*

\* The `subprocess.channel` property is a reference to the child's IPC channel. If

\* no IPC channel currently exists, this property is `undefined`.

\* @since v7.1.0

\*/

readonly channel?: Pipe | null | undefined;

/\*\*

\* A sparse array of pipes to the child process, corresponding with positions in

\* the `stdio` option passed to {@link spawn} that have been set

\* to the value `'pipe'`. `subprocess.stdio[0]`, `subprocess.stdio[1]`, and`subprocess.stdio[2]` are also available as `subprocess.stdin`,`subprocess.stdout`, and `subprocess.stderr`,

\* respectively.

\*

\* In the following example, only the child's fd `1` (stdout) is configured as a

\* pipe, so only the parent's `subprocess.stdio[1]` is a stream, all other values

\* in the array are `null`.

\*

\* ```js

\* const assert = require('assert');

\* const fs = require('fs');

\* const child\_process = require('child\_process');

\*

\* const subprocess = child\_process.spawn('ls', {

\* stdio: [

\* 0, // Use parent's stdin for child.

\* 'pipe', // Pipe child's stdout to parent.

\* fs.openSync('err.out', 'w'), // Direct child's stderr to a file.

\* ]

\* });

\*

\* assert.strictEqual(subprocess.stdio[0], null);

\* assert.strictEqual(subprocess.stdio[0], subprocess.stdin);

\*

\* assert(subprocess.stdout);

\* assert.strictEqual(subprocess.stdio[1], subprocess.stdout);

\*

\* assert.strictEqual(subprocess.stdio[2], null);

\* assert.strictEqual(subprocess.stdio[2], subprocess.stderr);

\* ```

\*

\* The `subprocess.stdio` property can be `undefined` if the child process could

\* not be successfully spawned.

\* @since v0.7.10

\*/

readonly stdio: [

Writable | null,

// stdin

Readable | null,

// stdout

Readable | null,

// stderr

Readable | Writable | null | undefined,

// extra

Readable | Writable | null | undefined // extra

];

/\*\*

\* The `subprocess.killed` property indicates whether the child process

\* successfully received a signal from `subprocess.kill()`. The `killed` property

\* does not indicate that the child process has been terminated.

\* @since v0.5.10

\*/

readonly killed: boolean;

/\*\*

\* Returns the process identifier (PID) of the child process. If the child process

\* fails to spawn due to errors, then the value is `undefined` and `error` is

\* emitted.

\*

\* ```js

\* const { spawn } = require('child\_process');

\* const grep = spawn('grep', ['ssh']);

\*

\* console.log(`Spawned child pid: ${grep.pid}`);

\* grep.stdin.end();

\* ```

\* @since v0.1.90

\*/

readonly pid?: number | undefined;

/\*\*

\* The `subprocess.connected` property indicates whether it is still possible to

\* send and receive messages from a child process. When `subprocess.connected` is`false`, it is no longer possible to send or receive messages.

\* @since v0.7.2

\*/

readonly connected: boolean;

/\*\*

\* The `subprocess.exitCode` property indicates the exit code of the child process.

\* If the child process is still running, the field will be `null`.

\*/

readonly exitCode: number | null;

/\*\*

\* The `subprocess.signalCode` property indicates the signal received by

\* the child process if any, else `null`.

\*/

readonly signalCode: NodeJS.Signals | null;

/\*\*

\* The `subprocess.spawnargs` property represents the full list of command-line

\* arguments the child process was launched with.

\*/

readonly spawnargs: string[];

/\*\*

\* The `subprocess.spawnfile` property indicates the executable file name of

\* the child process that is launched.

\*

\* For {@link fork}, its value will be equal to `process.execPath`.

\* For {@link spawn}, its value will be the name of

\* the executable file.

\* For {@link exec}, its value will be the name of the shell

\* in which the child process is launched.

\*/

readonly spawnfile: string;

/\*\*

\* The `subprocess.kill()` method sends a signal to the child process. If no

\* argument is given, the process will be sent the `'SIGTERM'` signal. See [`signal(7)`](http://man7.org/linux/man-pages/man7/signal.7.html) for a list of available signals. This function

\* returns `true` if [`kill(2)`](http://man7.org/linux/man-pages/man2/kill.2.html) succeeds, and `false` otherwise.

\*

\* ```js

\* const { spawn } = require('child\_process');

\* const grep = spawn('grep', ['ssh']);

\*

\* grep.on('close', (code, signal) => {

\* console.log(

\* `child process terminated due to receipt of signal ${signal}`);

\* });

\*

\* // Send SIGHUP to process.

\* grep.kill('SIGHUP');

\* ```

\*

\* The `ChildProcess` object may emit an `'error'` event if the signal

\* cannot be delivered. Sending a signal to a child process that has already exited

\* is not an error but may have unforeseen consequences. Specifically, if the

\* process identifier (PID) has been reassigned to another process, the signal will

\* be delivered to that process instead which can have unexpected results.

\*

\* While the function is called `kill`, the signal delivered to the child process

\* may not actually terminate the process.

\*

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

\*

\* On Windows, where POSIX signals do not exist, the `signal` argument will be

\* ignored, and the process will be killed forcefully and abruptly (similar to`'SIGKILL'`).

\* See `Signal Events` for more details.

\*

\* On Linux, child processes of child processes will not be terminated

\* when attempting to kill their parent. This is likely to happen when running a

\* new process in a shell or with the use of the `shell` option of `ChildProcess`:

\*

\* ```js

\* 'use strict';

\* const { spawn } = require('child\_process');

\*

\* const subprocess = spawn(

\* 'sh',

\* [

\* '-c',

\* `node -e "setInterval(() => {

\* console.log(process.pid, 'is alive')

\* }, 500);"`,

\* ], {

\* stdio: ['inherit', 'inherit', 'inherit']

\* }

\* );

\*

\* setTimeout(() => {

\* subprocess.kill(); // Does not terminate the Node.js process in the shell.

\* }, 2000);

\* ```

\* @since v0.1.90

\*/

kill(signal?: NodeJS.Signals | number): boolean;

/\*\*

\* When an IPC channel has been established between the parent and child (

\* i.e. when using {@link fork}), the `subprocess.send()` method can

\* be used to send messages to the child process. When the child process is a

\* Node.js instance, these messages can be received via the `'message'` event.

\*

\* The message goes through serialization and parsing. The resulting

\* message might not be the same as what is originally sent.

\*

\* For example, in the parent script:

\*

\* ```js

\* const cp = require('child\_process');

\* const n = cp.fork(`${\_\_dirname}/sub.js`);

\*

\* n.on('message', (m) => {

\* console.log('PARENT got message:', m);

\* });

\*

\* // Causes the child to print: CHILD got message: { hello: 'world' }

\* n.send({ hello: 'world' });

\* ```

\*

\* And then the child script, `'sub.js'` might look like this:

\*

\* ```js

\* process.on('message', (m) => {

\* console.log('CHILD got message:', m);

\* });

\*

\* // Causes the parent to print: PARENT got message: { foo: 'bar', baz: null }

\* process.send({ foo: 'bar', baz: NaN });

\* ```

\*

\* Child Node.js processes will have a `process.send()` method of their own

\* that allows the child to send messages back to the parent.

\*

\* There is a special case when sending a `{cmd: 'NODE\_foo'}` message. Messages

\* containing a `NODE\_` prefix in the `cmd` property are reserved for use within

\* Node.js core and will not be emitted in the child's `'message'` event. Rather, such messages are emitted using the`'internalMessage'` event and are consumed internally by Node.js.

\* Applications should avoid using such messages or listening for`'internalMessage'` events as it is subject to change without notice.

\*

\* The optional `sendHandle` argument that may be passed to `subprocess.send()` is

\* for passing a TCP server or socket object to the child process. The child will

\* receive the object as the second argument passed to the callback function

\* registered on the `'message'` event. Any data that is received

\* and buffered in the socket will not be sent to the child.

\*

\* The optional `callback` is a function that is invoked after the message is

\* sent but before the child may have received it. The function is called with a

\* single argument: `null` on success, or an `Error` object on failure.

\*

\* If no `callback` function is provided and the message cannot be sent, an`'error'` event will be emitted by the `ChildProcess` object. This can

\* happen, for instance, when the child process has already exited.

\*

\* `subprocess.send()` will return `false` if the channel has closed or when the

\* backlog of unsent messages exceeds a threshold that makes it unwise to send

\* more. Otherwise, the method returns `true`. The `callback` function can be

\* used to implement flow control.

\*

\* #### Example: sending a server object

\*

\* The `sendHandle` argument can be used, for instance, to pass the handle of

\* a TCP server object to the child process as illustrated in the example below:

\*

\* ```js

\* const subprocess = require('child\_process').fork('subprocess.js');

\*

\* // Open up the server object and send the handle.

\* const server = require('net').createServer();

\* server.on('connection', (socket) => {

\* socket.end('handled by parent');

\* });

\* server.listen(1337, () => {

\* subprocess.send('server', server);

\* });

\* ```

\*

\* The child would then receive the server object as:

\*

\* ```js

\* process.on('message', (m, server) => {

\* if (m === 'server') {

\* server.on('connection', (socket) => {

\* socket.end('handled by child');

\* });

\* }

\* });

\* ```

\*

\* Once the server is now shared between the parent and child, some connections

\* can be handled by the parent and some by the child.

\*

\* While the example above uses a server created using the `net` module, `dgram`module servers use exactly the same workflow with the exceptions of listening on

\* a `'message'` event instead of `'connection'` and using `server.bind()` instead

\* of `server.listen()`. This is, however, currently only supported on Unix

\* platforms.

\*

\* #### Example: sending a socket object

\*

\* Similarly, the `sendHandler` argument can be used to pass the handle of a

\* socket to the child process. The example below spawns two children that each

\* handle connections with "normal" or "special" priority:

\*

\* ```js

\* const { fork } = require('child\_process');

\* const normal = fork('subprocess.js', ['normal']);

\* const special = fork('subprocess.js', ['special']);

\*

\* // Open up the server and send sockets to child. Use pauseOnConnect to prevent

\* // the sockets from being read before they are sent to the child process.

\* const server = require('net').createServer({ pauseOnConnect: true });

\* server.on('connection', (socket) => {

\*

\* // If this is special priority...

\* if (socket.remoteAddress === '74.125.127.100') {

\* special.send('socket', socket);

\* return;

\* }

\* // This is normal priority.

\* normal.send('socket', socket);

\* });

\* server.listen(1337);

\* ```

\*

\* The `subprocess.js` would receive the socket handle as the second argument

\* passed to the event callback function:

\*

\* ```js

\* process.on('message', (m, socket) => {

\* if (m === 'socket') {

\* if (socket) {

\* // Check that the client socket exists.

\* // It is possible for the socket to be closed between the time it is

\* // sent and the time it is received in the child process.

\* socket.end(`Request handled with ${process.argv[2]} priority`);

\* }

\* }

\* });

\* ```

\*

\* Do not use `.maxConnections` on a socket that has been passed to a subprocess.

\* The parent cannot track when the socket is destroyed.

\*

\* Any `'message'` handlers in the subprocess should verify that `socket` exists,

\* as the connection may have been closed during the time it takes to send the

\* connection to the child.

\* @since v0.5.9

\* @param options The `options` argument, if present, is an object used to parameterize the sending of certain types of handles. `options` supports the following properties:

\*/

send(message: Serializable, callback?: (error: Error | null) => void): boolean;

send(message: Serializable, sendHandle?: SendHandle, callback?: (error: Error | null) => void): boolean;

send(message: Serializable, sendHandle?: SendHandle, options?: MessageOptions, callback?: (error: Error | null) => void): boolean;

/\*\*

\* Closes the IPC channel between parent and child, allowing the child to exit

\* gracefully once there are no other connections keeping it alive. After calling

\* this method the `subprocess.connected` and `process.connected` properties in

\* both the parent and child (respectively) will be set to `false`, and it will be

\* no longer possible to pass messages between the processes.

\*

\* The `'disconnect'` event will be emitted when there are no messages in the

\* process of being received. This will most often be triggered immediately after

\* calling `subprocess.disconnect()`.

\*

\* When the child process is a Node.js instance (e.g. spawned using {@link fork}), the `process.disconnect()` method can be invoked

\* within the child process to close the IPC channel as well.

\* @since v0.7.2

\*/

disconnect(): void;

/\*\*

\* By default, the parent will wait for the detached child to exit. To prevent the

\* parent from waiting for a given `subprocess` to exit, use the`subprocess.unref()` method. Doing so will cause the parent's event loop to not

\* include the child in its reference count, allowing the parent to exit

\* independently of the child, unless there is an established IPC channel between

\* the child and the parent.

\*

\* ```js

\* const { spawn } = require('child\_process');

\*

\* const subprocess = spawn(process.argv[0], ['child\_program.js'], {

\* detached: true,

\* stdio: 'ignore'

\* });

\*

\* subprocess.unref();

\* ```

\* @since v0.7.10

\*/

unref(): void;

/\*\*

\* Calling `subprocess.ref()` after making a call to `subprocess.unref()` will

\* restore the removed reference count for the child process, forcing the parent

\* to wait for the child to exit before exiting itself.

\*

\* ```js

\* const { spawn } = require('child\_process');

\*

\* const subprocess = spawn(process.argv[0], ['child\_program.js'], {

\* detached: true,

\* stdio: 'ignore'

\* });

\*

\* subprocess.unref();

\* subprocess.ref();

\* ```

\* @since v0.7.10

\*/

ref(): void;

/\*\*

\* events.EventEmitter

\* 1. close

\* 2. disconnect

\* 3. error

\* 4. exit

\* 5. message

\* 6. spawn

\*/

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

addListener(event: 'close', listener: (code: number | null, signal: NodeJS.Signals | null) => void): this;

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

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

addListener(event: 'exit', listener: (code: number | null, signal: NodeJS.Signals | null) => void): this;

addListener(event: 'message', listener: (message: Serializable, sendHandle: SendHandle) => void): this;

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

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

emit(event: 'close', code: number | null, signal: NodeJS.Signals | null): boolean;

emit(event: 'disconnect'): boolean;

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

emit(event: 'exit', code: number | null, signal: NodeJS.Signals | null): boolean;

emit(event: 'message', message: Serializable, sendHandle: SendHandle): boolean;

emit(event: 'spawn', listener: () => void): boolean;

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

on(event: 'close', listener: (code: number | null, signal: NodeJS.Signals | null) => void): this;

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

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

on(event: 'exit', listener: (code: number | null, signal: NodeJS.Signals | null) => void): this;

on(event: 'message', listener: (message: Serializable, sendHandle: SendHandle) => void): this;

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

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

once(event: 'close', listener: (code: number | null, signal: NodeJS.Signals | null) => void): this;

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

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

once(event: 'exit', listener: (code: number | null, signal: NodeJS.Signals | null) => void): this;

once(event: 'message', listener: (message: Serializable, sendHandle: SendHandle) => void): this;

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

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

prependListener(event: 'close', listener: (code: number | null, signal: NodeJS.Signals | null) => void): this;

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

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

prependListener(event: 'exit', listener: (code: number | null, signal: NodeJS.Signals | null) => void): this;

prependListener(event: 'message', listener: (message: Serializable, sendHandle: SendHandle) => void): this;

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

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

prependOnceListener(event: 'close', listener: (code: number | null, signal: NodeJS.Signals | null) => void): this;

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

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

prependOnceListener(event: 'exit', listener: (code: number | null, signal: NodeJS.Signals | null) => void): this;

prependOnceListener(event: 'message', listener: (message: Serializable, sendHandle: SendHandle) => void): this;

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

}

// return this object when stdio option is undefined or not specified

interface ChildProcessWithoutNullStreams extends ChildProcess {

stdin: Writable;

stdout: Readable;

stderr: Readable;

readonly stdio: [

Writable,

Readable,

Readable,

// stderr

Readable | Writable | null | undefined,

// extra, no modification

Readable | Writable | null | undefined // extra, no modification

];

}

// return this object when stdio option is a tuple of 3

interface ChildProcessByStdio<I extends null | Writable, O extends null | Readable, E extends null | Readable> extends ChildProcess {

stdin: I;

stdout: O;

stderr: E;

readonly stdio: [

I,

O,

E,

Readable | Writable | null | undefined,

// extra, no modification

Readable | Writable | null | undefined // extra, no modification

];

}

interface MessageOptions {

keepOpen?: boolean | undefined;

}

type IOType = 'overlapped' | 'pipe' | 'ignore' | 'inherit';

type StdioOptions = IOType | Array<IOType | 'ipc' | Stream | number | null | undefined>;

type SerializationType = 'json' | 'advanced';

interface MessagingOptions extends Abortable {

/\*\*

\* Specify the kind of serialization used for sending messages between processes.

\* @default 'json'

\*/

serialization?: SerializationType | undefined;

/\*\*

\* The signal value to be used when the spawned process will be killed by the abort signal.

\* @default 'SIGTERM'

\*/

killSignal?: NodeJS.Signals | number | undefined;

/\*\*

\* In milliseconds the maximum amount of time the process is allowed to run.

\*/

timeout?: number | undefined;

}

interface ProcessEnvOptions {

uid?: number | undefined;

gid?: number | undefined;

cwd?: string | URL | undefined;

env?: NodeJS.ProcessEnv | undefined;

}

interface CommonOptions extends ProcessEnvOptions {

/\*\*

\* @default true

\*/

windowsHide?: boolean | undefined;

/\*\*

\* @default 0

\*/

timeout?: number | undefined;

}

interface CommonSpawnOptions extends CommonOptions, MessagingOptions, Abortable {

argv0?: string | undefined;

stdio?: StdioOptions | undefined;

shell?: boolean | string | undefined;

windowsVerbatimArguments?: boolean | undefined;

}

interface SpawnOptions extends CommonSpawnOptions {

detached?: boolean | undefined;

}

interface SpawnOptionsWithoutStdio extends SpawnOptions {

stdio?: StdioPipeNamed | StdioPipe[] | undefined;

}

type StdioNull = 'inherit' | 'ignore' | Stream;

type StdioPipeNamed = 'pipe' | 'overlapped';

type StdioPipe = undefined | null | StdioPipeNamed;

interface SpawnOptionsWithStdioTuple<Stdin extends StdioNull | StdioPipe, Stdout extends StdioNull | StdioPipe, Stderr extends StdioNull | StdioPipe> extends SpawnOptions {

stdio: [Stdin, Stdout, Stderr];

}

/\*\*

\* The `child\_process.spawn()` method spawns a new process using the given`command`, with command-line arguments in `args`. If omitted, `args` defaults

\* to an empty array.

\*

\* \*\*If the `shell` option is enabled, do not pass unsanitized user input to this\*\*

\* \*\*function. Any input containing shell metacharacters may be used to trigger\*\*

\* \*\*arbitrary command execution.\*\*

\*

\* A third argument may be used to specify additional options, with these defaults:

\*

\* ```js

\* const defaults = {

\* cwd: undefined,

\* env: process.env

\* };

\* ```

\*

\* Use `cwd` to specify the working directory from which the process is spawned.

\* If not given, the default is to inherit the current working directory. If given,

\* but the path does not exist, the child process emits an `ENOENT` error

\* and exits immediately. `ENOENT` is also emitted when the command

\* does not exist.

\*

\* Use `env` to specify environment variables that will be visible to the new

\* process, the default is `process.env`.

\*

\* `undefined` values in `env` will be ignored.

\*

\* Example of running `ls -lh /usr`, capturing `stdout`, `stderr`, and the

\* exit code:

\*

\* ```js

\* const { spawn } = require('child\_process');

\* const ls = spawn('ls', ['-lh', '/usr']);

\*

\* ls.stdout.on('data', (data) => {

\* console.log(`stdout: ${data}`);

\* });

\*

\* ls.stderr.on('data', (data) => {

\* console.error(`stderr: ${data}`);

\* });

\*

\* ls.on('close', (code) => {

\* console.log(`child process exited with code ${code}`);

\* });

\* ```

\*

\* Example: A very elaborate way to run `ps ax | grep ssh`

\*

\* ```js

\* const { spawn } = require('child\_process');

\* const ps = spawn('ps', ['ax']);

\* const grep = spawn('grep', ['ssh']);

\*

\* ps.stdout.on('data', (data) => {

\* grep.stdin.write(data);

\* });

\*

\* ps.stderr.on('data', (data) => {

\* console.error(`ps stderr: ${data}`);

\* });

\*

\* ps.on('close', (code) => {

\* if (code !== 0) {

\* console.log(`ps process exited with code ${code}`);

\* }

\* grep.stdin.end();

\* });

\*

\* grep.stdout.on('data', (data) => {

\* console.log(data.toString());

\* });

\*

\* grep.stderr.on('data', (data) => {

\* console.error(`grep stderr: ${data}`);

\* });

\*

\* grep.on('close', (code) => {

\* if (code !== 0) {

\* console.log(`grep process exited with code ${code}`);

\* }

\* });

\* ```

\*

\* Example of checking for failed `spawn`:

\*

\* ```js

\* const { spawn } = require('child\_process');

\* const subprocess = spawn('bad\_command');

\*

\* subprocess.on('error', (err) => {

\* console.error('Failed to start subprocess.');

\* });

\* ```

\*

\* Certain platforms (macOS, Linux) will use the value of `argv[0]` for the process

\* title while others (Windows, SunOS) will use `command`.

\*

\* Node.js currently overwrites `argv[0]` with `process.execPath` on startup, so`process.argv[0]` in a Node.js child process will not match the `argv0`parameter passed to `spawn` from the parent,

\* retrieve it with the`process.argv0` property instead.

\*

\* If the `signal` option is enabled, calling `.abort()` on the corresponding`AbortController` is similar to calling `.kill()` on the child process except

\* the error passed to the callback will be an `AbortError`:

\*

\* ```js

\* const { spawn } = require('child\_process');

\* const controller = new AbortController();

\* const { signal } = controller;

\* const grep = spawn('grep', ['ssh'], { signal });

\* grep.on('error', (err) => {

\* // This will be called with err being an AbortError if the controller aborts

\* });

\* controller.abort(); // Stops the child process

\* ```

\* @since v0.1.90

\* @param command The command to run.

\* @param args List of string arguments.

\*/

function spawn(command: string, options?: SpawnOptionsWithoutStdio): ChildProcessWithoutNullStreams;

function spawn(command: string, options: SpawnOptionsWithStdioTuple<StdioPipe, StdioPipe, StdioPipe>): ChildProcessByStdio<Writable, Readable, Readable>;

function spawn(command: string, options: SpawnOptionsWithStdioTuple<StdioPipe, StdioPipe, StdioNull>): ChildProcessByStdio<Writable, Readable, null>;

function spawn(command: string, options: SpawnOptionsWithStdioTuple<StdioPipe, StdioNull, StdioPipe>): ChildProcessByStdio<Writable, null, Readable>;

function spawn(command: string, options: SpawnOptionsWithStdioTuple<StdioNull, StdioPipe, StdioPipe>): ChildProcessByStdio<null, Readable, Readable>;

function spawn(command: string, options: SpawnOptionsWithStdioTuple<StdioPipe, StdioNull, StdioNull>): ChildProcessByStdio<Writable, null, null>;

function spawn(command: string, options: SpawnOptionsWithStdioTuple<StdioNull, StdioPipe, StdioNull>): ChildProcessByStdio<null, Readable, null>;

function spawn(command: string, options: SpawnOptionsWithStdioTuple<StdioNull, StdioNull, StdioPipe>): ChildProcessByStdio<null, null, Readable>;

function spawn(command: string, options: SpawnOptionsWithStdioTuple<StdioNull, StdioNull, StdioNull>): ChildProcessByStdio<null, null, null>;

function spawn(command: string, options: SpawnOptions): ChildProcess;

// overloads of spawn with 'args'

function spawn(command: string, args?: ReadonlyArray<string>, options?: SpawnOptionsWithoutStdio): ChildProcessWithoutNullStreams;

function spawn(command: string, args: ReadonlyArray<string>, options: SpawnOptionsWithStdioTuple<StdioPipe, StdioPipe, StdioPipe>): ChildProcessByStdio<Writable, Readable, Readable>;

function spawn(command: string, args: ReadonlyArray<string>, options: SpawnOptionsWithStdioTuple<StdioPipe, StdioPipe, StdioNull>): ChildProcessByStdio<Writable, Readable, null>;

function spawn(command: string, args: ReadonlyArray<string>, options: SpawnOptionsWithStdioTuple<StdioPipe, StdioNull, StdioPipe>): ChildProcessByStdio<Writable, null, Readable>;

function spawn(command: string, args: ReadonlyArray<string>, options: SpawnOptionsWithStdioTuple<StdioNull, StdioPipe, StdioPipe>): ChildProcessByStdio<null, Readable, Readable>;

function spawn(command: string, args: ReadonlyArray<string>, options: SpawnOptionsWithStdioTuple<StdioPipe, StdioNull, StdioNull>): ChildProcessByStdio<Writable, null, null>;

function spawn(command: string, args: ReadonlyArray<string>, options: SpawnOptionsWithStdioTuple<StdioNull, StdioPipe, StdioNull>): ChildProcessByStdio<null, Readable, null>;

function spawn(command: string, args: ReadonlyArray<string>, options: SpawnOptionsWithStdioTuple<StdioNull, StdioNull, StdioPipe>): ChildProcessByStdio<null, null, Readable>;

function spawn(command: string, args: ReadonlyArray<string>, options: SpawnOptionsWithStdioTuple<StdioNull, StdioNull, StdioNull>): ChildProcessByStdio<null, null, null>;

function spawn(command: string, args: ReadonlyArray<string>, options: SpawnOptions): ChildProcess;

interface ExecOptions extends CommonOptions {

shell?: string | undefined;

signal?: AbortSignal | undefined;

maxBuffer?: number | undefined;

killSignal?: NodeJS.Signals | number | undefined;

}

interface ExecOptionsWithStringEncoding extends ExecOptions {

encoding: BufferEncoding;

}

interface ExecOptionsWithBufferEncoding extends ExecOptions {

encoding: BufferEncoding | null; // specify `null`.

}

interface ExecException extends Error {

cmd?: string | undefined;

killed?: boolean | undefined;

code?: number | undefined;

signal?: NodeJS.Signals | undefined;

}

/\*\*

\* Spawns a shell then executes the `command` within that shell, buffering any

\* generated output. The `command` string passed to the exec function is processed

\* directly by the shell and special characters (vary based on [shell](https://en.wikipedia.org/wiki/List\_of\_command-line\_interpreters))

\* need to be dealt with accordingly:

\*

\* ```js

\* const { exec } = require('child\_process');

\*

\* exec('"/path/to/test file/test.sh" arg1 arg2');

\* // Double quotes are used so that the space in the path is not interpreted as

\* // a delimiter of multiple arguments.

\*

\* exec('echo "The \\$HOME variable is $HOME"');

\* // The $HOME variable is escaped in the first instance, but not in the second.

\* ```

\*

\* \*\*Never pass unsanitized user input to this function. Any input containing shell\*\*

\* \*\*metacharacters may be used to trigger arbitrary command execution.\*\*

\*

\* If a `callback` function is provided, it is called with the arguments`(error, stdout, stderr)`. On success, `error` will be `null`. On error,`error` will be an instance of `Error`. The

\* `error.code` property will be

\* the exit code of the process. By convention, any exit code other than `0`indicates an error. `error.signal` will be the signal that terminated the

\* process.

\*

\* The `stdout` and `stderr` arguments passed to the callback will contain the

\* stdout and stderr output of the child process. By default, Node.js will decode

\* the output as UTF-8 and pass strings to the callback. The `encoding` option

\* can be used to specify the character encoding used to decode the stdout and

\* stderr output. If `encoding` is `'buffer'`, or an unrecognized character

\* encoding, `Buffer` objects will be passed to the callback instead.

\*

\* ```js

\* const { exec } = require('child\_process');

\* exec('cat \*.js missing\_file | wc -l', (error, stdout, stderr) => {

\* if (error) {

\* console.error(`exec error: ${error}`);

\* return;

\* }

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

\* console.error(`stderr: ${stderr}`);

\* });

\* ```

\*

\* If `timeout` is greater than `0`, the parent will send the signal

\* identified by the `killSignal` property (the default is `'SIGTERM'`) if the

\* child runs longer than `timeout` milliseconds.

\*

\* Unlike the [`exec(3)`](http://man7.org/linux/man-pages/man3/exec.3.html) POSIX system call, `child\_process.exec()` does not replace

\* the existing process and uses a shell to execute the command.

\*

\* If this method is invoked as its `util.promisify()` ed version, it returns

\* a `Promise` for an `Object` with `stdout` and `stderr` properties. The returned`ChildProcess` instance is attached to the `Promise` as a `child` property. In

\* case of an error (including any error resulting in an exit code other than 0), a

\* rejected promise is returned, with the same `error` object given in the

\* callback, but with two additional properties `stdout` and `stderr`.

\*

\* ```js

\* const util = require('util');

\* const exec = util.promisify(require('child\_process').exec);

\*

\* async function lsExample() {

\* const { stdout, stderr } = await exec('ls');

\* console.log('stdout:', stdout);

\* console.error('stderr:', stderr);

\* }

\* lsExample();

\* ```

\*

\* If the `signal` option is enabled, calling `.abort()` on the corresponding`AbortController` is similar to calling `.kill()` on the child process except

\* the error passed to the callback will be an `AbortError`:

\*

\* ```js

\* const { exec } = require('child\_process');

\* const controller = new AbortController();

\* const { signal } = controller;

\* const child = exec('grep ssh', { signal }, (error) => {

\* console.log(error); // an AbortError

\* });

\* controller.abort();

\* ```

\* @since v0.1.90

\* @param command The command to run, with space-separated arguments.

\* @param callback called with the output when process terminates.

\*/

function exec(command: string, callback?: (error: ExecException | null, stdout: string, stderr: string) => void): ChildProcess;

// `options` with `"buffer"` or `null` for `encoding` means stdout/stderr are definitely `Buffer`.

function exec(

command: string,

options: {

encoding: 'buffer' | null;

} & ExecOptions,

callback?: (error: ExecException | null, stdout: Buffer, stderr: Buffer) => void

): ChildProcess;

// `options` with well known `encoding` means stdout/stderr are definitely `string`.

function exec(

command: string,

options: {

encoding: BufferEncoding;

} & ExecOptions,

callback?: (error: ExecException | null, stdout: string, stderr: string) => void

): ChildProcess;

// `options` with an `encoding` whose type is `string` means stdout/stderr could either be `Buffer` or `string`.

// There is no guarantee the `encoding` is unknown as `string` is a superset of `BufferEncoding`.

function exec(

command: string,

options: {

encoding: BufferEncoding;

} & ExecOptions,

callback?: (error: ExecException | null, stdout: string | Buffer, stderr: string | Buffer) => void

): ChildProcess;

// `options` without an `encoding` means stdout/stderr are definitely `string`.

function exec(command: string, options: ExecOptions, callback?: (error: ExecException | null, stdout: string, stderr: string) => void): ChildProcess;

// fallback if nothing else matches. Worst case is always `string | Buffer`.

function exec(

command: string,

options: (ObjectEncodingOptions & ExecOptions) | undefined | null,

callback?: (error: ExecException | null, stdout: string | Buffer, stderr: string | Buffer) => void

): ChildProcess;

interface PromiseWithChild<T> extends Promise<T> {

child: ChildProcess;

}

namespace exec {

function \_\_promisify\_\_(command: string): PromiseWithChild<{

stdout: string;

stderr: string;

}>;

function \_\_promisify\_\_(

command: string,

options: {

encoding: 'buffer' | null;

} & ExecOptions

): PromiseWithChild<{

stdout: Buffer;

stderr: Buffer;

}>;

function \_\_promisify\_\_(

command: string,

options: {

encoding: BufferEncoding;

} & ExecOptions

): PromiseWithChild<{

stdout: string;

stderr: string;

}>;

function \_\_promisify\_\_(

command: string,

options: ExecOptions

): PromiseWithChild<{

stdout: string;

stderr: string;

}>;

function \_\_promisify\_\_(

command: string,

options?: (ObjectEncodingOptions & ExecOptions) | null

): PromiseWithChild<{

stdout: string | Buffer;

stderr: string | Buffer;

}>;

}

interface ExecFileOptions extends CommonOptions, Abortable {

maxBuffer?: number | undefined;

killSignal?: NodeJS.Signals | number | undefined;

windowsVerbatimArguments?: boolean | undefined;

shell?: boolean | string | undefined;

signal?: AbortSignal | undefined;

}

interface ExecFileOptionsWithStringEncoding extends ExecFileOptions {

encoding: BufferEncoding;

}

interface ExecFileOptionsWithBufferEncoding extends ExecFileOptions {

encoding: 'buffer' | null;

}

interface ExecFileOptionsWithOtherEncoding extends ExecFileOptions {

encoding: BufferEncoding;

}

type ExecFileException = ExecException & NodeJS.ErrnoException;

/\*\*

\* The `child\_process.execFile()` function is similar to {@link exec} except that it does not spawn a shell by default. Rather, the specified

\* executable `file` is spawned directly as a new process making it slightly more

\* efficient than {@link exec}.

\*

\* The same options as {@link exec} are supported. Since a shell is

\* not spawned, behaviors such as I/O redirection and file globbing are not

\* supported.

\*

\* ```js

\* const { execFile } = require('child\_process');

\* const child = execFile('node', ['--version'], (error, stdout, stderr) => {

\* if (error) {

\* throw error;

\* }

\* console.log(stdout);

\* });

\* ```

\*

\* The `stdout` and `stderr` arguments passed to the callback will contain the

\* stdout and stderr output of the child process. By default, Node.js will decode

\* the output as UTF-8 and pass strings to the callback. The `encoding` option

\* can be used to specify the character encoding used to decode the stdout and

\* stderr output. If `encoding` is `'buffer'`, or an unrecognized character

\* encoding, `Buffer` objects will be passed to the callback instead.

\*

\* If this method is invoked as its `util.promisify()` ed version, it returns

\* a `Promise` for an `Object` with `stdout` and `stderr` properties. The returned`ChildProcess` instance is attached to the `Promise` as a `child` property. In

\* case of an error (including any error resulting in an exit code other than 0), a

\* rejected promise is returned, with the same `error` object given in the

\* callback, but with two additional properties `stdout` and `stderr`.

\*

\* ```js

\* const util = require('util');

\* const execFile = util.promisify(require('child\_process').execFile);

\* async function getVersion() {

\* const { stdout } = await execFile('node', ['--version']);

\* console.log(stdout);

\* }

\* getVersion();

\* ```

\*

\* \*\*If the `shell` option is enabled, do not pass unsanitized user input to this\*\*

\* \*\*function. Any input containing shell metacharacters may be used to trigger\*\*

\* \*\*arbitrary command execution.\*\*

\*

\* If the `signal` option is enabled, calling `.abort()` on the corresponding`AbortController` is similar to calling `.kill()` on the child process except

\* the error passed to the callback will be an `AbortError`:

\*

\* ```js

\* const { execFile } = require('child\_process');

\* const controller = new AbortController();

\* const { signal } = controller;

\* const child = execFile('node', ['--version'], { signal }, (error) => {

\* console.log(error); // an AbortError

\* });

\* controller.abort();

\* ```

\* @since v0.1.91

\* @param file The name or path of the executable file to run.

\* @param args List of string arguments.

\* @param callback Called with the output when process terminates.

\*/

function execFile(file: string): ChildProcess;

function execFile(file: string, options: (ObjectEncodingOptions & ExecFileOptions) | undefined | null): ChildProcess;

function execFile(file: string, args?: ReadonlyArray<string> | null): ChildProcess;

function execFile(file: string, args: ReadonlyArray<string> | undefined | null, options: (ObjectEncodingOptions & ExecFileOptions) | undefined | null): ChildProcess;

// no `options` definitely means stdout/stderr are `string`.

function execFile(file: string, callback: (error: ExecFileException | null, stdout: string, stderr: string) => void): ChildProcess;

function execFile(file: string, args: ReadonlyArray<string> | undefined | null, callback: (error: ExecFileException | null, stdout: string, stderr: string) => void): ChildProcess;

// `options` with `"buffer"` or `null` for `encoding` means stdout/stderr are definitely `Buffer`.

function execFile(file: string, options: ExecFileOptionsWithBufferEncoding, callback: (error: ExecFileException | null, stdout: Buffer, stderr: Buffer) => void): ChildProcess;

function execFile(

file: string,

args: ReadonlyArray<string> | undefined | null,

options: ExecFileOptionsWithBufferEncoding,

callback: (error: ExecFileException | null, stdout: Buffer, stderr: Buffer) => void

): ChildProcess;

// `options` with well known `encoding` means stdout/stderr are definitely `string`.

function execFile(file: string, options: ExecFileOptionsWithStringEncoding, callback: (error: ExecFileException | null, stdout: string, stderr: string) => void): ChildProcess;

function execFile(

file: string,

args: ReadonlyArray<string> | undefined | null,

options: ExecFileOptionsWithStringEncoding,

callback: (error: ExecFileException | null, stdout: string, stderr: string) => void

): ChildProcess;

// `options` with an `encoding` whose type is `string` means stdout/stderr could either be `Buffer` or `string`.

// There is no guarantee the `encoding` is unknown as `string` is a superset of `BufferEncoding`.

function execFile(file: string, options: ExecFileOptionsWithOtherEncoding, callback: (error: ExecFileException | null, stdout: string | Buffer, stderr: string | Buffer) => void): ChildProcess;

function execFile(

file: string,

args: ReadonlyArray<string> | undefined | null,

options: ExecFileOptionsWithOtherEncoding,

callback: (error: ExecFileException | null, stdout: string | Buffer, stderr: string | Buffer) => void

): ChildProcess;

// `options` without an `encoding` means stdout/stderr are definitely `string`.

function execFile(file: string, options: ExecFileOptions, callback: (error: ExecFileException | null, stdout: string, stderr: string) => void): ChildProcess;

function execFile(

file: string,

args: ReadonlyArray<string> | undefined | null,

options: ExecFileOptions,

callback: (error: ExecFileException | null, stdout: string, stderr: string) => void

): ChildProcess;

// fallback if nothing else matches. Worst case is always `string | Buffer`.

function execFile(

file: string,

options: (ObjectEncodingOptions & ExecFileOptions) | undefined | null,

callback: ((error: ExecFileException | null, stdout: string | Buffer, stderr: string | Buffer) => void) | undefined | null

): ChildProcess;

function execFile(

file: string,

args: ReadonlyArray<string> | undefined | null,

options: (ObjectEncodingOptions & ExecFileOptions) | undefined | null,

callback: ((error: ExecFileException | null, stdout: string | Buffer, stderr: string | Buffer) => void) | undefined | null

): ChildProcess;

namespace execFile {

function \_\_promisify\_\_(file: string): PromiseWithChild<{

stdout: string;

stderr: string;

}>;

function \_\_promisify\_\_(

file: string,

args: ReadonlyArray<string> | undefined | null

): PromiseWithChild<{

stdout: string;

stderr: string;

}>;

function \_\_promisify\_\_(

file: string,

options: ExecFileOptionsWithBufferEncoding

): PromiseWithChild<{

stdout: Buffer;

stderr: Buffer;

}>;

function \_\_promisify\_\_(

file: string,

args: ReadonlyArray<string> | undefined | null,

options: ExecFileOptionsWithBufferEncoding

): PromiseWithChild<{

stdout: Buffer;

stderr: Buffer;

}>;

function \_\_promisify\_\_(

file: string,

options: ExecFileOptionsWithStringEncoding

): PromiseWithChild<{

stdout: string;

stderr: string;

}>;

function \_\_promisify\_\_(

file: string,

args: ReadonlyArray<string> | undefined | null,

options: ExecFileOptionsWithStringEncoding

): PromiseWithChild<{

stdout: string;

stderr: string;

}>;

function \_\_promisify\_\_(

file: string,

options: ExecFileOptionsWithOtherEncoding

): PromiseWithChild<{

stdout: string | Buffer;

stderr: string | Buffer;

}>;

function \_\_promisify\_\_(

file: string,

args: ReadonlyArray<string> | undefined | null,

options: ExecFileOptionsWithOtherEncoding

): PromiseWithChild<{

stdout: string | Buffer;

stderr: string | Buffer;

}>;

function \_\_promisify\_\_(

file: string,

options: ExecFileOptions

): PromiseWithChild<{

stdout: string;

stderr: string;

}>;

function \_\_promisify\_\_(

file: string,

args: ReadonlyArray<string> | undefined | null,

options: ExecFileOptions

): PromiseWithChild<{

stdout: string;

stderr: string;

}>;

function \_\_promisify\_\_(

file: string,

options: (ObjectEncodingOptions & ExecFileOptions) | undefined | null

): PromiseWithChild<{

stdout: string | Buffer;

stderr: string | Buffer;

}>;

function \_\_promisify\_\_(

file: string,

args: ReadonlyArray<string> | undefined | null,

options: (ObjectEncodingOptions & ExecFileOptions) | undefined | null

): PromiseWithChild<{

stdout: string | Buffer;

stderr: string | Buffer;

}>;

}

interface ForkOptions extends ProcessEnvOptions, MessagingOptions, Abortable {

execPath?: string | undefined;

execArgv?: string[] | undefined;

silent?: boolean | undefined;

stdio?: StdioOptions | undefined;

detached?: boolean | undefined;

windowsVerbatimArguments?: boolean | undefined;

}

/\*\*

\* The `child\_process.fork()` method is a special case of {@link spawn} used specifically to spawn new Node.js processes.

\* Like {@link spawn}, a `ChildProcess` object is returned. The

\* returned `ChildProcess` will have an additional communication channel

\* built-in that allows messages to be passed back and forth between the parent and

\* child. See `subprocess.send()` for details.

\*

\* Keep in mind that spawned Node.js child processes are

\* independent of the parent with exception of the IPC communication channel

\* that is established between the two. Each process has its own memory, with

\* their own V8 instances. Because of the additional resource allocations

\* required, spawning a large number of child Node.js processes is not

\* recommended.

\*

\* By default, `child\_process.fork()` will spawn new Node.js instances using the `process.execPath` of the parent process. The `execPath` property in the`options` object allows for an alternative

\* execution path to be used.

\*

\* Node.js processes launched with a custom `execPath` will communicate with the

\* parent process using the file descriptor (fd) identified using the

\* environment variable `NODE\_CHANNEL\_FD` on the child process.

\*

\* Unlike the [`fork(2)`](http://man7.org/linux/man-pages/man2/fork.2.html) POSIX system call, `child\_process.fork()` does not clone the

\* current process.

\*

\* The `shell` option available in {@link spawn} is not supported by`child\_process.fork()` and will be ignored if set.

\*

\* If the `signal` option is enabled, calling `.abort()` on the corresponding`AbortController` is similar to calling `.kill()` on the child process except

\* the error passed to the callback will be an `AbortError`:

\*

\* ```js

\* if (process.argv[2] === 'child') {

\* setTimeout(() => {

\* console.log(`Hello from ${process.argv[2]}!`);

\* }, 1\_000);

\* } else {

\* const { fork } = require('child\_process');

\* const controller = new AbortController();

\* const { signal } = controller;

\* const child = fork(\_\_filename, ['child'], { signal });

\* child.on('error', (err) => {

\* // This will be called with err being an AbortError if the controller aborts

\* });

\* controller.abort(); // Stops the child process

\* }

\* ```

\* @since v0.5.0

\* @param modulePath The module to run in the child.

\* @param args List of string arguments.

\*/

function fork(modulePath: string, options?: ForkOptions): ChildProcess;

function fork(modulePath: string, args?: ReadonlyArray<string>, options?: ForkOptions): ChildProcess;

interface SpawnSyncOptions extends CommonSpawnOptions {

input?: string | NodeJS.ArrayBufferView | undefined;

maxBuffer?: number | undefined;

encoding?: BufferEncoding | 'buffer' | null | undefined;

}

interface SpawnSyncOptionsWithStringEncoding extends SpawnSyncOptions {

encoding: BufferEncoding;

}

interface SpawnSyncOptionsWithBufferEncoding extends SpawnSyncOptions {

encoding?: 'buffer' | null | undefined;

}

interface SpawnSyncReturns<T> {

pid: number;

output: Array<T | null>;

stdout: T;

stderr: T;

status: number | null;

signal: NodeJS.Signals | null;

error?: Error | undefined;

}

/\*\*

\* The `child\_process.spawnSync()` method is generally identical to {@link spawn} with the exception that the function will not return

\* until the child process has fully closed. When a timeout has been encountered

\* and `killSignal` is sent, the method won't return until the process has

\* completely exited. If the process intercepts and handles the `SIGTERM` signal

\* and doesn't exit, the parent process will wait until the child process has

\* exited.

\*

\* \*\*If the `shell` option is enabled, do not pass unsanitized user input to this\*\*

\* \*\*function. Any input containing shell metacharacters may be used to trigger\*\*

\* \*\*arbitrary command execution.\*\*

\* @since v0.11.12

\* @param command The command to run.

\* @param args List of string arguments.

\*/

function spawnSync(command: string): SpawnSyncReturns<Buffer>;

function spawnSync(command: string, options: SpawnSyncOptionsWithStringEncoding): SpawnSyncReturns<string>;

function spawnSync(command: string, options: SpawnSyncOptionsWithBufferEncoding): SpawnSyncReturns<Buffer>;

function spawnSync(command: string, options?: SpawnSyncOptions): SpawnSyncReturns<string | Buffer>;

function spawnSync(command: string, args: ReadonlyArray<string>): SpawnSyncReturns<Buffer>;

function spawnSync(command: string, args: ReadonlyArray<string>, options: SpawnSyncOptionsWithStringEncoding): SpawnSyncReturns<string>;

function spawnSync(command: string, args: ReadonlyArray<string>, options: SpawnSyncOptionsWithBufferEncoding): SpawnSyncReturns<Buffer>;

function spawnSync(command: string, args?: ReadonlyArray<string>, options?: SpawnSyncOptions): SpawnSyncReturns<string | Buffer>;

interface CommonExecOptions extends CommonOptions {

input?: string | NodeJS.ArrayBufferView | undefined;

stdio?: StdioOptions | undefined;

killSignal?: NodeJS.Signals | number | undefined;

maxBuffer?: number | undefined;

encoding?: BufferEncoding | 'buffer' | null | undefined;

}

interface ExecSyncOptions extends CommonExecOptions {

shell?: string | undefined;

}

interface ExecSyncOptionsWithStringEncoding extends ExecSyncOptions {

encoding: BufferEncoding;

}

interface ExecSyncOptionsWithBufferEncoding extends ExecSyncOptions {

encoding?: 'buffer' | null | undefined;

}

/\*\*

\* The `child\_process.execSync()` method is generally identical to {@link exec} with the exception that the method will not return

\* until the child process has fully closed. When a timeout has been encountered

\* and `killSignal` is sent, the method won't return until the process has

\* completely exited. If the child process intercepts and handles the `SIGTERM`signal and doesn't exit, the parent process will wait until the child process

\* has exited.

\*

\* If the process times out or has a non-zero exit code, this method will throw.

\* The `Error` object will contain the entire result from {@link spawnSync}.

\*

\* \*\*Never pass unsanitized user input to this function. Any input containing shell\*\*

\* \*\*metacharacters may be used to trigger arbitrary command execution.\*\*

\* @since v0.11.12

\* @param command The command to run.

\* @return The stdout from the command.

\*/

function execSync(command: string): Buffer;

function execSync(command: string, options: ExecSyncOptionsWithStringEncoding): string;

function execSync(command: string, options: ExecSyncOptionsWithBufferEncoding): Buffer;

function execSync(command: string, options?: ExecSyncOptions): string | Buffer;

interface ExecFileSyncOptions extends CommonExecOptions {

shell?: boolean | string | undefined;

}

interface ExecFileSyncOptionsWithStringEncoding extends ExecFileSyncOptions {

encoding: BufferEncoding;

}

interface ExecFileSyncOptionsWithBufferEncoding extends ExecFileSyncOptions {

encoding?: 'buffer' | null; // specify `null`.

}

/\*\*

\* The `child\_process.execFileSync()` method is generally identical to {@link execFile} with the exception that the method will not

\* return until the child process has fully closed. When a timeout has been

\* encountered and `killSignal` is sent, the method won't return until the process

\* has completely exited.

\*

\* If the child process intercepts and handles the `SIGTERM` signal and

\* does not exit, the parent process will still wait until the child process has

\* exited.

\*

\* If the process times out or has a non-zero exit code, this method will throw an `Error` that will include the full result of the underlying {@link spawnSync}.

\*

\* \*\*If the `shell` option is enabled, do not pass unsanitized user input to this\*\*

\* \*\*function. Any input containing shell metacharacters may be used to trigger\*\*

\* \*\*arbitrary command execution.\*\*

\* @since v0.11.12

\* @param file The name or path of the executable file to run.

\* @param args List of string arguments.

\* @return The stdout from the command.

\*/

function execFileSync(file: string): Buffer;

function execFileSync(file: string, options: ExecFileSyncOptionsWithStringEncoding): string;

function execFileSync(file: string, options: ExecFileSyncOptionsWithBufferEncoding): Buffer;

function execFileSync(file: string, options?: ExecFileSyncOptions): string | Buffer;

function execFileSync(file: string, args: ReadonlyArray<string>): Buffer;

function execFileSync(file: string, args: ReadonlyArray<string>, options: ExecFileSyncOptionsWithStringEncoding): string;

function execFileSync(file: string, args: ReadonlyArray<string>, options: ExecFileSyncOptionsWithBufferEncoding): Buffer;

function execFileSync(file: string, args?: ReadonlyArray<string>, options?: ExecFileSyncOptions): string | Buffer;

}

declare module 'node:child\_process' {

export \* from 'child\_process';

}