Child process

> *Stability: 2 - Stable*

The `child_process` module provides the ability to spawn subprocesses in a manner that is similar, but not identical, to popen(3). This capability is primarily provided by the `child_process.spawn()` function:

```
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 `env` is in the `options` object. Otherwise, `process.env.PATH` is used. If `options.env` is set without `PATH`, lookup on Unix is performed on a default search path search of `/usr/bin:/bin` (see your operating system's manual for execvpe/execvp), on Windows the current processes environment variable `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 `child_process.spawn()` method spawns the child process asynchronously, without blocking the Node.js event loop. The `child_process.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 `child_process.spawn()` and `child_process.spawnSync()`. Each of these alternatives are implemented on top of `child_process.spawn()` or `child_process.spawnSync()`.

- `child_process.exec()`: spawns a shell and runs a command within that shell, passing the `stdout` and `stderr` to a callback function when complete.
- `child_process.execFile()`: similar to `child_process.exec()` except that it spawns the command directly without first spawning a shell by default.

- `child_process.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.
- `child_process.execSync()` : a synchronous version of `child_process.exec()` that will block the Node.js event loop.
- `child_process.execFileSync()` : a synchronous version of `child_process.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.

Asynchronous process creation

The `child_process.spawn()` , `child_process.fork()` , `child_process.exec()` , and `child_process.execFile()` methods all follow the idiomatic asynchronous programming pattern typical of other Node.js APIs.

Each of the methods returns a `ChildProcess` instance. These objects implement the Node.js `EventEmitter` API, allowing the parent process to register listener functions that are called when certain events occur during the life cycle of the child process.

The `child_process.exec()` and `child_process.execFile()` methods additionally allow for an optional `callback` function to be specified that is invoked when the child process terminates.

Spawning `.bat` and `.cmd` files on Windows

The importance of the distinction between `child_process.exec()` and `child_process.execFile()` can vary based on platform. On Unix-type operating systems (Unix, Linux, macOS) `child_process.execFile()` can be more efficient because it does not spawn a shell by default. On Windows, however, `.bat` and `.cmd` files are not executable on their own without a terminal, and therefore cannot be launched using `child_process.execFile()` . When running on Windows, `.bat` and `.cmd` files can be invoked using `child_process.spawn()` with the `shell` option set, with `child_process.exec()` , or by spawning `cmd.exe` and passing the `.bat` or `.cmd` file as an argument (which is what the `shell` option and `child_process.exec()` do). In any case, if the script filename contains spaces it needs to be quoted.

```
// On Windows Only...
const { spawn } = require('child_process');
const bat = spawn('cmd.exe', ['/c', 'my.bat']);

bat.stdout.on('data', (data) => {
  console.log(data.toString());
});

bat.stderr.on('data', (data) => {
  console.error(data.toString());
});

bat.on('exit', (code) => {
  console.log(`Child exited with code ${code}`);
});
```

```
// OR...
const { exec, spawn } = require('child_process');
exec('my.bat', (err, stdout, stderr) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(stdout);
});

// Script with spaces in the filename:
const bat = spawn('"my script.cmd"', ['a', 'b'], { shell: true });
// or:
exec('"my script.cmd" a b', (err, stdout, stderr) => {
  // ...
});
```

`child_process.exec(command[, options][, callback])`

- `command` {string} The command to run, with space-separated arguments.
- `options` {Object}
  - `cwd` {string|URL} Current working directory of the child process. **Default:** `process.cwd()`.
  - `env` {Object} Environment key-value pairs. **Default:** `process.env`.
  - `encoding` {string} **Default:** `'utf8'`
  - `shell` {string} Shell to execute the command with. See Shell requirements and Default Windows shell. **Default:** `'/bin/sh'` on Unix, `process.env.ComSpec` on Windows.
  - `signal` {AbortSignal} allows aborting the child process using an AbortSignal.
  - `timeout` {number} **Default:** `0`
  - `maxBuffer` {number} Largest amount of data in bytes allowed on stdout or stderr. If exceeded, the child process is terminated and any output is truncated. See caveat at `maxBuffer` and Unicode. **Default:** `1024 * 1024`.
  - `killSignal` {string|integer} **Default:** `'SIGTERM'`
  - `uid` {number} Sets the user identity of the process (see setuid(2)).
  - `gid` {number} Sets the group identity of the process (see setgid(2)).
  - `windowsHide` {boolean} Hide the subprocess console window that would normally be created on Windows systems. **Default:** `false`.
- `callback` {Function} called with the output when process terminates.
  - `error` {Error}
  - `stdout` {string|Buffer}
  - `stderr` {string|Buffer}
- Returns: {ChildProcess}

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) need to be dealt with accordingly:

```
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.

```
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) 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`.

```
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`:

```
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();
```

`child_process.execFile(file[, args][, options][, callback])`

- `file` {string} The name or path of the executable file to run.
- `args` {string[]} List of string arguments.
- `options` {Object}
  - `cwd` {string|URL} Current working directory of the child process.
  - `env` {Object} Environment key-value pairs. **Default:** `process.env`.
  - `encoding` {string} **Default:** `'utf8'`
  - `timeout` {number} **Default:** `0`
  - `maxBuffer` {number} Largest amount of data in bytes allowed on stdout or stderr. If exceeded, the child process is terminated and any output is truncated. See caveat at `maxBuffer` and Unicode. **Default:** `1024 * 1024`.
  - `killSignal` {string|integer} **Default:** `'SIGTERM'`
  - `uid` {number} Sets the user identity of the process (see setuid(2)).
  - `gid` {number} Sets the group identity of the process (see setgid(2)).
  - `windowsHide` {boolean} Hide the subprocess console window that would normally be created on Windows systems. **Default:** `false`.
  - `windowsVerbatimArguments` {boolean} No quoting or escaping of arguments is done on Windows. Ignored on Unix. **Default:** `false`.
  - `shell` {boolean|string} If `true`, runs `command` inside of a shell. Uses `'/bin/sh'` on Unix, and `process.env.ComSpec` on Windows. A different shell can be specified as a string. See Shell requirements and Default Windows shell. **Default:** `false` (no shell).
  - `signal` {AbortSignal} allows aborting the child process using an AbortSignal.
- `callback` {Function} Called with the output when process terminates.
  - `error` {Error}
  - `stdout` {string|Buffer}
  - `stderr` {string|Buffer}
- Returns: {ChildProcess}

The `child_process.execFile()` function is similar to `child_process.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 `child_process.exec()`.

The same options as `child_process.exec()` are supported. Since a shell is not spawned, behaviors such as I/O redirection and file globbing are not supported.

```
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`.

```
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`:

```
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();
```

`child_process.fork(modulePath[, args][, options])`

- `modulePath` {string|URL} The module to run in the child.
- `args` {string[]} List of string arguments.
- `options` {Object}

- `cwd` {string|URL} Current working directory of the child process.
- `detached` {boolean} Prepare child to run independently of its parent process. Specific behavior depends on the platform, see `options.detached`).
- `env` {Object} Environment key-value pairs. **Default:** `process.env`.
- `execPath` {string} Executable used to create the child process.
- `execArgv` {string[]} List of string arguments passed to the executable. **Default:** `process.execArgv`.
- `gid` {number} Sets the group identity of the process (see setgid(2)).
- `serialization` {string} Specify the kind of serialization used for sending messages between processes. Possible values are `'json'` and `'advanced'`. See Advanced serialization for more details. **Default:** `'json'`.
- `signal` {AbortSignal} Allows closing the child process using an AbortSignal.
- `killSignal` {string|integer} The signal value to be used when the spawned process will be killed by timeout or abort signal. **Default:** `'SIGTERM'`.
- `silent` {boolean} If `true`, stdin, stdout, and stderr of the child will be piped to the parent, otherwise they will be inherited from the parent, see the `'pipe'` and `'inherit'` options for `child_process.spawn()`'s `stdio` for more details. **Default:** `false`.
- `stdio` {Array|string} See `child_process.spawn()`'s `stdio`. When this option is provided, it overrides `silent`. If the array variant is used, it must contain exactly one item with value `'ipc'` or an error will be thrown. For instance `[0, 1, 2, 'ipc']`.
- `uid` {number} Sets the user identity of the process (see setuid(2)).
- `windowsVerbatimArguments` {boolean} No quoting or escaping of arguments is done on Windows. Ignored on Unix. **Default:** `false`.
- `timeout` {number} In milliseconds the maximum amount of time the process is allowed to run. **Default:** `undefined`.

- Returns: {ChildProcess}

The `child_process.fork()` method is a special case of `child_process.spawn()` used specifically to spawn new Node.js processes. Like `child_process.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) POSIX system call, `child_process.fork()` does not clone the current process.

The `shell` option available in `child_process.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`:

```
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
}
```

`child_process.spawn(command[, args][, options])`

- `command` {string} The command to run.
- `args` {string[]} List of string arguments.
- `options` {Object}
  - `cwd` {string|URL} Current working directory of the child process.
  - `env` {Object} Environment key-value pairs. **Default:** `process.env`.
  - `argv0` {string} Explicitly set the value of `argv[0]` sent to the child process. This will be set to `command` if not specified.
  - `stdio` {Array|string} Child's stdio configuration (see `options.stdio`).
  - `detached` {boolean} Prepare child to run independently of its parent process. Specific behavior depends on the platform, see `options.detached`).
  - `uid` {number} Sets the user identity of the process (see setuid(2)).
  - `gid` {number} Sets the group identity of the process (see setgid(2)).
  - `serialization` {string} Specify the kind of serialization used for sending messages between processes. Possible values are `'json'` and `'advanced'`. See Advanced serialization for more details. **Default:** `'json'`.
  - `shell` {boolean|string} If `true`, runs `command` inside of a shell. Uses `'/bin/sh'` on Unix, and `process.env.ComSpec` on Windows. A different shell can be specified as a string. See Shell requirements and Default Windows shell. **Default:** `false` (no shell).
  - `windowsVerbatimArguments` {boolean} No quoting or escaping of arguments is done on Windows. Ignored on Unix. This is set to `true` automatically when `shell` is specified and is CMD. **Default:** `false`.
  - `windowsHide` {boolean} Hide the subprocess console window that would normally be created on Windows systems. **Default:** `false`.
  - `signal` {AbortSignal} allows aborting the child process using an AbortSignal.
  - `timeout` {number} In milliseconds the maximum amount of time the process is allowed to run. **Default:** `undefined`.
  - `killSignal` {string|integer} The signal value to be used when the spawned process will be killed by timeout or abort signal. **Default:** `'SIGTERM'`.

- Returns: {ChildProcess}

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:

```
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:

```
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`

```
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`:

```
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`:

```
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
```

`options.detached`

On Windows, setting `options.detached` to `true` makes it possible for the child process to continue running after the parent exits. The child will have its own console window. Once enabled for a child process, it cannot be disabled.

On non-Windows platforms, if `options.detached` is set to `true`, the child process will be made the leader of a new process group and session. Child processes may continue running after the parent exits regardless of whether they are detached or not. See setsid(2) for more information.

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.

When using the `detached` option to start a long-running process, the process will not stay running in the background after the parent exits unless it is provided with a `stdio` configuration that is not connected to the parent. If the parent's `stdio` is inherited, the child will remain attached to the controlling terminal.

Example of a long-running process, by detaching and also ignoring its parent `stdio` file descriptors, in order to ignore the parent's termination:

```
const { spawn } = require('child_process');

const subprocess = spawn(process.argv[0], ['child_program.js'], {
  detached: true,
  stdio: 'ignore'
});

subprocess.unref();
```

Alternatively one can redirect the child process' output into files:

```
const fs = require('fs');
const { spawn } = require('child_process');
const out = fs.openSync('./out.log', 'a');
const err = fs.openSync('./out.log', 'a');

const subprocess = spawn('prg', [], {
  detached: true,
  stdio: [ 'ignore', out, err ]
});

subprocess.unref();
```

`options.stdio`

The `options.stdio` option is used to configure the pipes that are established between the parent and child process. By default, the child's stdin, stdout, and stderr are redirected to corresponding `subprocess.stdin`, `subprocess.stdout`, and `subprocess.stderr` streams on the `ChildProcess` object. This is equivalent to setting the `options.stdio` equal to `['pipe', 'pipe', 'pipe']`.

For convenience, `options.stdio` may be one of the following strings:

- `'pipe'`: equivalent to `['pipe', 'pipe', 'pipe']` (the default)
- `'overlapped'`: equivalent to `['overlapped', 'overlapped', 'overlapped']`
- `'ignore'`: equivalent to `['ignore', 'ignore', 'ignore']`
- `'inherit'`: equivalent to `['inherit', 'inherit', 'inherit']` or `[0, 1, 2]`

Otherwise, the value of `options.stdio` is an array where each index corresponds to an fd in the child. The fds 0, 1, and 2 correspond to stdin, stdout, and stderr, respectively. Additional fds can be specified to create additional pipes between the parent and child. The value is one of the following:

1. `'pipe'`: Create a pipe between the child process and the parent process. The parent end of the pipe is exposed to the parent as a property on the `child_process` object as `subprocess.stdio[fd]`. Pipes created for fds 0, 1, and 2 are also available as `subprocess.stdin`, `subprocess.stdout` and `subprocess.stderr`, respectively. Currently, these are not actual Unix pipes and therefore the child process can not use them by their descriptor files, e.g. `/dev/fd/2` or `/dev/stdout`.

2. `'overlapped'`: Same as `'pipe'` except that the `FILE_FLAG_OVERLAPPED` flag is set on the handle. This is necessary for overlapped I/O on the child process's stdio handles. See the docs for more details. This is exactly the same as `'pipe'` on non-Windows systems.

3. `'ipc'`: Create an IPC channel for passing messages/file descriptors between parent and child. A `ChildProcess` may have at most one IPC stdio file descriptor. Setting this option enables the `subprocess.send()` method. If the child is a Node.js process, the presence of an IPC channel will enable `process.send()` and `process.disconnect()` methods, as well as `'disconnect'` and `'message'` events within the child.

   Accessing the IPC channel fd in any way other than `process.send()` or using the IPC channel with a child process that is not a Node.js instance is not supported.

4. `'ignore'`: Instructs Node.js to ignore the fd in the child. While Node.js will always open fds 0, 1, and 2 for the processes it spawns, setting the fd to `'ignore'` will cause Node.js to open `/dev/null` and attach it to the child's fd.

5. `'inherit'`: Pass through the corresponding stdio stream to/from the parent process. In the first three positions, this is equivalent to `process.stdin`, `process.stdout`, and `process.stderr`, respectively. In any other position, equivalent to `'ignore'`.

6. {Stream} object: Share a readable or writable stream that refers to a tty, file, socket, or a pipe with the child process. The stream's underlying file descriptor is duplicated in the child process to the fd that corresponds to the index in the `stdio` array. The stream must have an underlying descriptor (file streams do not until the `'open'` event has occurred).

7. Positive integer: The integer value is interpreted as a file descriptor that is currently open in the parent process. It is shared with the child process, similar to how {Stream} objects can be shared. Passing sockets is not supported on Windows.

8. `null`, `undefined`: Use default value. For stdio fds 0, 1, and 2 (in other words, stdin, stdout, and stderr) a pipe is created. For fd 3 and up, the default is `'ignore'`.

```
const { spawn } = require('child_process');

// Child will use parent's stdios.
spawn('prg', [], { stdio: 'inherit' });

// Spawn child sharing only stderr.
spawn('prg', [], { stdio: ['pipe', 'pipe', process.stderr] });

// Open an extra fd=4, to interact with programs presenting a
// startd-style interface.
spawn('prg', [], { stdio: ['pipe', null, null, null, 'pipe'] });
```

It is worth noting that when an IPC channel is established between the parent and child processes, and the child is a Node.js process, the child is launched with the IPC channel unreferenced (using `unref()`) until the child registers an event handler for the `'disconnect'` event or the `'message'` event. This allows the child to exit normally without the process being held open by the open IPC channel.

On Unix-like operating systems, the `child_process.spawn()` method performs memory operations synchronously before decoupling the event loop from the child. Applications with a large memory footprint may find frequent `child_process.spawn()` calls to be a bottleneck. For more information, see V8 issue 7381.

See also: `child_process.exec()` and `child_process.fork()`.

Synchronous process creation

The `child_process.spawnSync()`, `child_process.execSync()`, and `child_process.execFileSync()` methods are synchronous and will block the Node.js event loop, pausing execution of any additional code until the spawned process exits.

Blocking calls like these are mostly useful for simplifying general-purpose scripting tasks and for simplifying the loading/processing of application configuration at startup.

`child_process.execFileSync(file[, args][, options])`

- `file` {string} The name or path of the executable file to run.
- `args` {string[]} List of string arguments.
- `options` {Object}
  - `cwd` {string|URL} Current working directory of the child process.
  - `input` {string|Buffer|TypedArray|DataView} The value which will be passed as stdin to the spawned process. Supplying this value will override `stdio[0]`.
  - `stdio` {string|Array} Child's stdio configuration. `stderr` by default will be output to the parent process' stderr unless `stdio` is specified. **Default:** `'pipe'`.
  - `env` {Object} Environment key-value pairs. **Default:** `process.env`.
  - `uid` {number} Sets the user identity of the process (see setuid(2)).
  - `gid` {number} Sets the group identity of the process (see setgid(2)).

    - `timeout` {number} In milliseconds the maximum amount of time the process is allowed to run. **Default:** `undefined`.
    - `killSignal` {string|integer} The signal value to be used when the spawned process will be killed. **Default:** `'SIGTERM'`.
    - `maxBuffer` {number} Largest amount of data in bytes allowed on stdout or stderr. If exceeded, the child process is terminated. See caveat at `maxBuffer` and Unicode. **Default:** `1024 * 1024`.
    - `encoding` {string} The encoding used for all stdio inputs and outputs. **Default:** `'buffer'`.
    - `windowsHide` {boolean} Hide the subprocess console window that would normally be created on Windows systems. **Default:** `false`.
    - `shell` {boolean|string} If `true`, runs `command` inside of a shell. Uses `'/bin/sh'` on Unix, and `process.env.ComSpec` on Windows. A different shell can be specified as a string. See Shell requirements and Default Windows shell. **Default:** `false` (no shell).
- Returns: {Buffer|string} The stdout from the command.

The `child_process.execFileSync()` method is generally identical to `child_process.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 `child_process.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.

`child_process.execSync(command[, options])`

- `command` {string} The command to run.
- `options` {Object}
    - `cwd` {string|URL} Current working directory of the child process.
    - `input` {string|Buffer|TypedArray|DataView} The value which will be passed as stdin to the spawned process. Supplying this value will override `stdio[0]`.
    - `stdio` {string|Array} Child's stdio configuration. `stderr` by default will be output to the parent process' stderr unless `stdio` is specified. **Default:** `'pipe'`.
    - `env` {Object} Environment key-value pairs. **Default:** `process.env`.
    - `shell` {string} Shell to execute the command with. See Shell requirements and Default Windows shell. **Default:** `'/bin/sh'` on Unix, `process.env.ComSpec` on Windows.
    - `uid` {number} Sets the user identity of the process. (See setuid(2)).
    - `gid` {number} Sets the group identity of the process. (See setgid(2)).
    - `timeout` {number} In milliseconds the maximum amount of time the process is allowed to run. **Default:** `undefined`.
    - `killSignal` {string|integer} The signal value to be used when the spawned process will be killed. **Default:** `'SIGTERM'`.
    - `maxBuffer` {number} Largest amount of data in bytes allowed on stdout or stderr. If exceeded, the child process is terminated and any output is truncated. See caveat at `maxBuffer` and Unicode. **Default:** `1024 * 1024`.

    - `encoding` {string} The encoding used for all stdio inputs and outputs. **Default:** `'buffer'`.
    - `windowsHide` {boolean} Hide the subprocess console window that would normally be created on Windows systems. **Default:** `false`.
- Returns: {Buffer|string} The stdout from the command.

The `child_process.execSync()` method is generally identical to `child_process.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 `child_process.spawnSync()`.

Never pass unsanitized user input to this function. Any input containing shell metacharacters may be used to trigger arbitrary command execution.

`child_process.spawnSync(command[, args][, options])`

- `command` {string} The command to run.
- `args` {string[]} List of string arguments.
- `options` {Object}
    - `cwd` {string|URL} Current working directory of the child process.
    - `input` {string|Buffer|TypedArray|DataView} The value which will be passed as stdin to the spawned process. Supplying this value will override `stdio[0]`.
    - `argv0` {string} Explicitly set the value of `argv[0]` sent to the child process. This will be set to `command` if not specified.
    - `stdio` {string|Array} Child's stdio configuration.
    - `env` {Object} Environment key-value pairs. **Default:** `process.env`.
    - `uid` {number} Sets the user identity of the process (see setuid(2)).
    - `gid` {number} Sets the group identity of the process (see setgid(2)).
    - `timeout` {number} In milliseconds the maximum amount of time the process is allowed to run. **Default:** `undefined`.
    - `killSignal` {string|integer} The signal value to be used when the spawned process will be killed. **Default:** `'SIGTERM'`.
    - `maxBuffer` {number} Largest amount of data in bytes allowed on stdout or stderr. If exceeded, the child process is terminated and any output is truncated. See caveat at `maxBuffer` and Unicode. **Default:** `1024 * 1024`.
    - `encoding` {string} The encoding used for all stdio inputs and outputs. **Default:** `'buffer'`.
    - `shell` {boolean|string} If `true`, runs `command` inside of a shell. Uses `'/bin/sh'` on Unix, and `process.env.ComSpec` on Windows. A different shell can be specified as a string. See Shell requirements and Default Windows shell. **Default:** `false` (no shell).
    - `windowsVerbatimArguments` {boolean} No quoting or escaping of arguments is done on Windows. Ignored on Unix. This is set to `true` automatically when `shell` is specified and is CMD. **Default:** `false`.
    - `windowsHide` {boolean} Hide the subprocess console window that would normally be created on Windows systems. **Default:** `false`.
- Returns: {Object}

- `pid` {number} Pid of the child process.
- `output` {Array} Array of results from stdio output.
- `stdout` {Buffer|string} The contents of `output[1]`.
- `stderr` {Buffer|string} The contents of `output[2]`.
- `status` {number|null} The exit code of the subprocess, or `null` if the subprocess terminated due to a signal.
- `signal` {string|null} The signal used to kill the subprocess, or `null` if the subprocess did not terminate due to a signal.
- `error` {Error} The error object if the child process failed or timed out.

The `child_process.spawnSync()` method is generally identical to `child_process.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.

Class: `ChildProcess`

- Extends: {EventEmitter}

Instances of the `ChildProcess` represent spawned child processes.

Instances of `ChildProcess` are not intended to be created directly. Rather, use the `child_process.spawn()`, `child_process.exec()`, `child_process.execFile()`, or `child_process.fork()` methods to create instances of `ChildProcess`.

Event: `'close'`

- `code` {number} The exit code if the child exited on its own.
- `signal` {string} The signal by which the child process was terminated.

The `'close'` event is emitted after a process has ended *and* the stdio streams of a child process have been closed. This is distinct from the `'exit'` event, since multiple processes might share the same stdio streams. The `'close'` event will always emit after `'exit'` was already emitted, or `'error'` if the child failed to spawn.

```
const { spawn } = require('child_process');
const ls = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', (data) => {
  console.log(`stdout: ${data}`);
});

ls.on('close', (code) => {
  console.log(`child process close all stdio with code ${code}`);
});

ls.on('exit', (code) => {
```

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

Event: `'disconnect'`

The `'disconnect'` event is emitted after calling the `subprocess.disconnect()` method in parent process or `process.disconnect()` in child process. After disconnecting it is no longer possible to send or receive messages, and the `subprocess.connected` property is `false`.

Event: `'error'`

- `err` {Error} The error.

The `'error'` event is emitted whenever:

1. The process could not be spawned, or
2. The process could not be killed, or
3. Sending a message to the child process failed.

The `'exit'` event may or may not fire after an error has occurred. When listening to both the `'exit'` and `'error'` events, guard against accidentally invoking handler functions multiple times.

See also `subprocess.kill()` and `subprocess.send()`.

Event: `'exit'`

- `code` {number} The exit code if the child exited on its own.
- `signal` {string} The signal by which the child process was terminated.

The `'exit'` event is emitted after the child process ends. If the process exited, `code` is the final exit code of the process, otherwise `null`. If the process terminated due to receipt of a signal, `signal` is the string name of the signal, otherwise `null`. One of the two will always be non-`null`.

When the `'exit'` event is triggered, child process stdio streams might still be open.

Node.js establishes signal handlers for `SIGINT` and `SIGTERM` and Node.js processes will not terminate immediately due to receipt of those signals. Rather, Node.js will perform a sequence of cleanup actions and then will re-raise the handled signal.

See waitpid(2).

Event: `'message'`

- `message` {Object} A parsed JSON object or primitive value.
- `sendHandle` {Handle} A `net.Socket` or `net.Server` object, or undefined.

The `'message'` event is triggered when a child process uses `process.send()` to send messages.

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

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

Event: `'spawn'`

The `'spawn'` event is emitted once the child process has spawned successfully. If the child process does not spawn successfully, the `'spawn'` event is not emitted and the `'error'` event is emitted instead.

If emitted, the `'spawn'` event comes before all other events and before any data is received via `stdout` or `stderr`.

The `'spawn'` event will fire regardless of whether an error occurs **within** the spawned process. For example, if `bash some-command` spawns successfully, the `'spawn'` event will fire, though `bash` may fail to spawn `some-command`. This caveat also applies when using `{ shell: true }`.

`subprocess.channel`

- {Object} A pipe representing the IPC channel to the child process.

The `subprocess.channel` property is a reference to the child's IPC channel. If no IPC channel currently exists, this property is `undefined`.

`subprocess.channel.ref()`

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

`subprocess.channel.unref()`

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

`subprocess.connected`

- {boolean} Set to `false` after `subprocess.disconnect()` is called.

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.

`subprocess.disconnect()`

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 `child_process.fork()`), the `process.disconnect()` method can be invoked within the child process to close the IPC channel as well.

`subprocess.exitCode`

- {integer}

The `subprocess.exitCode` property indicates the exit code of the child process. If the child process is still running, the field will be `null`.

`subprocess.kill([signal])`

- `signal` {number|string}
- Returns: {boolean}

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) for a list of available signals. This function returns `true` if kill(2) succeeds, and `false` otherwise.

```
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) 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`:

```
'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);
```

`subprocess.killed`

- {boolean} Set to `true` after `subprocess.kill()` is used to successfully send a signal to the child process.

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.

`subprocess.pid`

- {integer|undefined}

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.

```
const { spawn } = require('child_process');
const grep = spawn('grep', ['ssh']);

console.log(`Spawned child pid: ${grep.pid}`);
grep.stdin.end();
```

`subprocess.ref()`

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.

```
const { spawn } = require('child_process');

const subprocess = spawn(process.argv[0], ['child_program.js'], {
  detached: true,
  stdio: 'ignore'
});

subprocess.unref();
subprocess.ref();
```

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

- `message` {Object}
- `sendHandle` {Handle}
- `options` {Object} The `options` argument, if present, is an object used to parameterize the sending of certain types of handles. `options` supports the following properties:
  - `keepOpen` {boolean} A value that can be used when passing instances of `net.Socket`. When `true`, the socket is kept open in the sending process. **Default:** `false`.
- `callback` {Function}
- Returns: {boolean}

When an IPC channel has been established between the parent and child (i.e. when using `child_process.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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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.

`subprocess.signalCode`

- {string|null}

The `subprocess.signalCode` property indicates the signal received by the child process if any, else `null`.

`subprocess.spawnargs`

- {Array}

The `subprocess.spawnargs` property represents the full list of command-line arguments the child process was launched with.

`subprocess.spawnfile`

- {string}

The `subprocess.spawnfile` property indicates the executable file name of the child process that is launched.

For `child_process.fork()`, its value will be equal to `process.execPath`. For `child_process.spawn()`, its value will be the name of the executable file. For `child_process.exec()`, its value will be the name of the shell in which the child process is launched.

`subprocess.stderr`

- {stream.Readable}

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.

`subprocess.stdin`

- {stream.Writable}

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.

`subprocess.stdio`

- {Array}

A sparse array of pipes to the child process, corresponding with positions in the `stdio` option passed to `child_process.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`.

```
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.

`subprocess.stdout`

- {stream.Readable}

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.

```
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.

`subprocess.unref()`

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.

```
const { spawn } = require('child_process');

const subprocess = spawn(process.argv[0], ['child_program.js'], {
  detached: true,
  stdio: 'ignore'
});

subprocess.unref();
```

`maxBuffer` and Unicode

The `maxBuffer` option specifies the largest number of bytes allowed on `stdout` or `stderr`. If this value is exceeded, then the child process is terminated. This impacts output that includes multibyte character encodings such as UTF-8 or UTF-16. For instance, `console.log('中文测试')` will send 13 UTF-8 encoded bytes to `stdout` although there are only 4 characters.

Shell requirements

The shell should understand the `-c` switch. If the shell is `'cmd.exe'`, it should understand the `/d /s /c` switches and command-line parsing should be compatible.

Default Windows shell

Although Microsoft specifies `%COMSPEC%` must contain the path to `'cmd.exe'` in the root environment, child processes are not always subject to the same requirement. Thus, in `child_process` functions where a shell can be spawned, `'cmd.exe'` is used as a fallback if `process.env.ComSpec` is unavailable.

Advanced serialization

Child processes support a serialization mechanism for IPC that is based on the serialization API of the `v8` module, based on the HTML structured clone algorithm. This is generally more powerful and supports more built-in JavaScript object types, such as `BigInt`, `Map` and `Set`, `ArrayBuffer` and `TypedArray`, `Buffer`, `Error`, `RegExp` etc.

However, this format is not a full superset of JSON, and e.g. properties set on objects of such built-in types will not be passed on through the serialization step. Additionally, performance may not be equivalent to that of JSON, depending on the structure of the passed data. Therefore, this feature requires opting in by setting the `serialization` option to `'advanced'` when calling `child_process.spawn()` or `child_process.fork()`.