Node.js v10.16.0 Documentation

File System

Stability: 2 - Stable

The `fs` module provides an API for interacting with the file system in a manner closely modeled around standard POSIX functions.

To use this module:

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

All file system operations have synchronous and asynchronous forms.

The asynchronous form always takes a completion callback as its last argument. The arguments passed to the completion callback depend on the method, but the first argument is always reserved for an exception. If the operation was completed successfully, then the first argument will be `null` or `undefined`.

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

fs.unlink('/tmp/hello', (err) => {
  if (err) throw err;
  console.log('successfully deleted /tmp/hello');
});
```

Exceptions that occur using synchronous operations are thrown immediately and may be handled using `try` / `catch`, or may be allowed to bubble up.

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

try {
  fs.unlinkSync('/tmp/hello');
  console.log('successfully deleted /tmp/hello');
} catch (err) {
  // handle the error
}
```

There is no guaranteed ordering when using asynchronous methods. So the following is prone to error because the `fs.stat()` operation may complete before the `fs.rename()` operation:

```
fs.rename('/tmp/hello', '/tmp/world', (err) => {
  if (err) throw err;
  console.log('renamed complete');
});
fs.stat('/tmp/world', (err, stats) => {
  if (err) throw err;
  console.log(`stats: ${JSON.stringify(stats)}`);
});
```

To correctly order the operations, move the `fs.stat()` call into the callback of the `fs.rename()` operation:

```
fs.rename('/tmp/hello', '/tmp/world', (err) => {
  if (err) throw err;
  fs.stat('/tmp/world', (err, stats) => {
    if (err) throw err;
    console.log(`stats: ${JSON.stringify(stats)}`);
  });
});
```

In busy processes, the programmer is *strongly encouraged* to use the asynchronous versions of these calls. The synchronous versions will block the entire process until they complete — halting all connections.

While it is not recommended, most fs functions allow the callback argument to be omitted, in which case a default callback is used that rethrows errors. To get a trace to the original call site, set the `NODE_DEBUG` environment variable:

Omitting the callback function on asynchronous fs functions is deprecated and may result in an error being thrown in the future.

```
$ cat script.js
function bad() {
  require('fs').readFile('/');
}
bad();

$ env NODE_DEBUG=fs node script.js
fs.js:88
        throw backtrace;
        ^
Error: EISDIR: illegal operation on a directory, read
    <stack trace.>
```

File paths

Most `fs` operations accept filepaths that may be specified in the form of a string, a `Buffer`, or a `URL` object using the `file:` protocol.

String form paths are interpreted as UTF-8 character sequences identifying the absolute or relative filename. Relative paths will be resolved relative to the current working directory as specified by `process.cwd()`.

Example using an absolute path on POSIX:

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

fs.open('/open/some/file.txt', 'r', (err, fd) => {
  if (err) throw err;
  fs.close(fd, (err) => {
    if (err) throw err;
  });
});
```

Example using a relative path on POSIX (relative to `process.cwd()`):

```
fs.open('file.txt', 'r', (err, fd) => {
  if (err) throw err;
  fs.close(fd, (err) => {
    if (err) throw err;
```

```
  });
});
```

Paths specified using a `Buffer` are useful primarily on certain POSIX operating systems that treat file paths as opaque byte sequences. On such systems, it is possible for a single file path to contain sub-sequences that use multiple character encodings. As with string paths, `Buffer` paths may be relative or absolute:

Example using an absolute path on POSIX:

```
fs.open(Buffer.from('/open/some/file.txt'), 'r', (err, fd) => {
  if (err) throw err;
  fs.close(fd, (err) => {
    if (err) throw err;
  });
});
```

On Windows, Node.js follows the concept of per-drive working directory. This behavior can be observed when using a drive path without a backslash. For example `fs.readdirSync('c:\\')` can potentially return a different result than `fs.readdirSync('c:')`. For more information, see this MSDN page.

URL object support

For most `fs` module functions, the `path` or `filename` argument may be passed as a WHATWG `URL` object. Only `URL` objects using the `file:` protocol are supported.

```
const fs = require('fs');
const fileUrl = new URL('file:///tmp/hello');

fs.readFileSync(fileUrl);
```

`file:` URLs are always absolute paths.

Using WHATWG `URL` objects might introduce platform-specific behaviors.

On Windows, `file:` URLs with a hostname convert to UNC paths, while `file:` URLs with drive letters convert to local absolute paths. `file:` URLs without a hostname nor a drive letter will result in a throw:

```
// On Windows :

// - WHATWG file URLs with hostname convert to UNC path
// file://hostname/p/a/t/h/file => \\hostname\p\a\t\h\file
fs.readFileSync(new URL('file://hostname/p/a/t/h/file'));

// - WHATWG file URLs with drive letters convert to absolute path
// file:///C:/tmp/hello => C:\tmp\hello
fs.readFileSync(new URL('file:///C:/tmp/hello'));

// - WHATWG file URLs without hostname must have a drive letters
fs.readFileSync(new URL('file:///notdriveletter/p/a/t/h/file'));
fs.readFileSync(new URL('file:///c/p/a/t/h/file'));
// TypeError [ERR_INVALID_FILE_URL_PATH]: File URL path must be absolute
```

`file:` URLs with drive letters must use `:` as a separator just after the drive letter. Using another separator will result in a throw.

On all other platforms, `file:` URLs with a hostname are unsupported and will result in a throw:

```
// On other platforms:

// - WHATWG file URLs with hostname are unsupported
// file://hostname/p/a/t/h/file => throw!
fs.readFileSync(new URL('file://hostname/p/a/t/h/file'));
// TypeError [ERR_INVALID_FILE_URL_PATH]: must be absolute

// - WHATWG file URLs convert to absolute path
// file:///tmp/hello => /tmp/hello
fs.readFileSync(new URL('file:///tmp/hello'));
```

A `file:` URL having encoded slash characters will result in a throw on all platforms:

```
// On Windows
fs.readFileSync(new URL('file:///C:/p/a/t/h/%2F'));
fs.readFileSync(new URL('file:///C:/p/a/t/h/%2f'));
/* TypeError [ERR_INVALID_FILE_URL_PATH]: File URL path must not include encoded
\ or / characters */

// On POSIX
fs.readFileSync(new URL('file:///p/a/t/h/%2F'));
fs.readFileSync(new URL('file:///p/a/t/h/%2f'));
/* TypeError [ERR_INVALID_FILE_URL_PATH]: File URL path must not include encoded
/ characters */
```

On Windows, `file:` URLs having encoded backslash will result in a throw:

```
// On Windows
fs.readFileSync(new URL('file:///C:/path/%5C'));
fs.readFileSync(new URL('file:///C:/path/%5c'));

/* TypeError [ERR_INVALID_FILE_URL_PATH]: File URL path must not include encoded
\ or / characters */
```

File Descriptors

On POSIX systems, for every process, the kernel maintains a table of currently open files and resources. Each open file is assigned a simple numeric identifier called a *file descriptor*. At the system-level, all file system operations use these file descriptors to identify and track each specific file. Windows systems use a different but conceptually similar mechanism for tracking resources. To simplify things for users, Node.js abstracts away the specific differences between operating systems and assigns all open files a numeric file descriptor.

The `fs.open()` method is used to allocate a new file descriptor. Once allocated, the file descriptor may be used to read data from, write data to, or request information about the file.

```
fs.open('/open/some/file.txt', 'r', (err, fd) => {
  if (err) throw err;
  fs.fstat(fd, (err, stat) => {
    if (err) throw err;
    // use stat

    // always close the file descriptor!
    fs.close(fd, (err) => {
      if (err) throw err;
    });
```

```
    });
  });
```

Most operating systems limit the number of file descriptors that may be open at any given time so it is critical to close the descriptor when operations are completed. Failure to do so will result in a memory leak that will eventually cause an application to crash.

Threadpool Usage

All file system APIs except `fs.FSWatcher()` and those that are explicitly synchronous use libuv's threadpool, which can have surprising and negative performance implications for some applications. See the `UV_THREADPOOL_SIZE` documentation for more information.

Class: fs.Dirent

When `fs.readdir()` or `fs.readdirSync()` is called with the `withFileTypes` option set to `true`, the resulting array is filled with `fs.Dirent` objects, rather than strings or `Buffers`.

dirent.isBlockDevice()

- Returns: `<boolean>`

Returns `true` if the `fs.Dirent` object describes a block device.

dirent.isCharacterDevice()

- Returns: `<boolean>`

Returns `true` if the `fs.Dirent` object describes a character device.

dirent.isDirectory()

- Returns: `<boolean>`

Returns `true` if the `fs.Dirent` object describes a file system directory.

dirent.isFIFO()

- Returns: `<boolean>`

Returns `true` if the `fs.Dirent` object describes a first-in-first-out (FIFO) pipe.

dirent.isFile()

- Returns: `<boolean>`

Returns `true` if the `fs.Dirent` object describes a regular file.

dirent.isSocket()

- Returns: `<boolean>`

Returns `true` if the `fs.Dirent` object describes a socket.

dirent.isSymbolicLink()

- Returns: `<boolean>`

Returns `true` if the `fs.Dirent` object describes a symbolic link.

dirent.name

- `<string>` | `<Buffer>`

The file name that this `fs.Dirent` object refers to. The type of this value is determined by the `options.encoding` passed to `fs.readdir()` or `fs.readdirSync()`.

Class: fs.FSWatcher

A successful call to `fs.watch()` method will return a new `fs.FSWatcher` object.

All `fs.FSWatcher` objects are `EventEmitter`'s that will emit a `'change'` event whenever a specific watched file is modified.

Event: 'change'

- `eventType` `<string>` The type of change event that has occurred
- `filename` `<string>` | `<Buffer>` The filename that changed (if relevant/available)

Emitted when something changes in a watched directory or file. See more details in `fs.watch()`.

The `filename` argument may not be provided depending on operating system support. If `filename` is provided, it will be provided as a `Buffer` if `fs.watch()` is called with its `encoding` option set to `'buffer'`, otherwise `filename` will be a UTF-8 string.

```
// Example when handled through fs.watch() listener
fs.watch('./tmp', { encoding: 'buffer' }, (eventType, filename) => {
  if (filename) {
    console.log(filename);
    // Prints: <Buffer ...>
  }
});
```

Event: 'close'

Emitted when the watcher stops watching for changes. The closed `fs.FSWatcher` object is no longer usable in the event handler.

Event: 'error'

- `error` `<Error>`

Emitted when an error occurs while watching the file. The errored `fs.FSWatcher` object is no longer usable in the event handler.

watcher.close()

Stop watching for changes on the given `fs.FSWatcher`. Once stopped, the `fs.FSWatcher` object is no longer usable.

Class: fs.ReadStream

A successful call to `fs.createReadStream()` will return a new `fs.ReadStream` object.

All `fs.ReadStream` objects are Readable Streams.

Event: 'close'

Emitted when the `fs.ReadStream`'s underlying file descriptor has been closed.

Event: 'open'

- `fd` `<integer>` Integer file descriptor used by the `ReadStream`.

Emitted when the `fs.ReadStream`'s file descriptor has been opened.

Event: 'ready'

Emitted when the `fs.ReadStream` is ready to be used.

Fires immediately after `'open'`.

readStream.bytesRead

- `<number>`

The number of bytes that have been read so far.

readStream.path

- `<string>` | `<Buffer>`

The path to the file the stream is reading from as specified in the first argument to `fs.createReadStream()`. If `path` is passed as a string, then `readStream.path` will be a string. If `path` is passed as a `Buffer`, then `readStream.path` will be a `Buffer`.

readStream.pending

- `<boolean>`

This property is `true` if the underlying file has not been opened yet, i.e. before the `'ready'` event is emitted.

Class: fs.Stats

A `fs.Stats` object provides information about a file.

Objects returned from `fs.stat()`, `fs.lstat()` and `fs.fstat()` and their synchronous counterparts are of this type. If `bigint` in the `options` passed to those methods is true, the numeric values will be `bigint` instead of `number`.

```
Stats {
  dev: 2114,
  ino: 48064969,
  mode: 33188,
  nlink: 1,
  uid: 85,
  gid: 100,
  rdev: 0,
  size: 527,
  blksize: 4096,
  blocks: 8,
  atimeMs: 1318289051000.1,
  mtimeMs: 1318289051000.1,
  ctimeMs: 1318289051000.1,
  birthtimeMs: 1318289051000.1,
  atime: Mon, 10 Oct 2011 23:24:11 GMT,
  mtime: Mon, 10 Oct 2011 23:24:11 GMT,
  ctime: Mon, 10 Oct 2011 23:24:11 GMT,
  birthtime: Mon, 10 Oct 2011 23:24:11 GMT }
```

`bigint` version:

```
Stats {
  dev: 2114n,
  ino: 48064969n,
  mode: 33188n,
  nlink: 1n,
  uid: 85n,
  gid: 100n,
```

```
  rdev: 0n,
  size: 527n,
  blksize: 4096n,
  blocks: 8n,
  atimeMs: 1318289051000n,
  mtimeMs: 1318289051000n,
  ctimeMs: 1318289051000n,
  birthtimeMs: 1318289051000n,
  atime: Mon, 10 Oct 2011 23:24:11 GMT,
  mtime: Mon, 10 Oct 2011 23:24:11 GMT,
  ctime: Mon, 10 Oct 2011 23:24:11 GMT,
  birthtime: Mon, 10 Oct 2011 23:24:11 GMT }
```

stats.isBlockDevice()

- Returns: `<boolean>`

Returns `true` if the `fs.Stats` object describes a block device.

stats.isCharacterDevice()

- Returns: `<boolean>`

Returns `true` if the `fs.Stats` object describes a character device.

stats.isDirectory()

- Returns: `<boolean>`

Returns `true` if the `fs.Stats` object describes a file system directory.

stats.isFIFO()

- Returns: `<boolean>`

Returns `true` if the `fs.Stats` object describes a first-in-first-out (FIFO) pipe.

stats.isFile()

- Returns: `<boolean>`

Returns `true` if the `fs.Stats` object describes a regular file.

stats.isSocket()

- Returns: `<boolean>`

Returns `true` if the `fs.Stats` object describes a socket.

stats.isSymbolicLink()

- Returns: `<boolean>`

Returns `true` if the `fs.Stats` object describes a symbolic link.

This method is only valid when using `fs.lstat()`.

stats.dev

- `<number>` | `<bigint>`

The numeric identifier of the device containing the file.

stats.ino

- `<number>` | `<bigint>`

The file system specific "Inode" number for the file.

stats.mode

- `<number>` | `<bigint>`

A bit-field describing the file type and mode.

stats.nlink

- `<number>` | `<bigint>`

The number of hard-links that exist for the file.

stats.uid

- `<number>` | `<bigint>`

The numeric user identifier of the user that owns the file (POSIX).

stats.gid

- `<number>` | `<bigint>`

The numeric group identifier of the group that owns the file (POSIX).

stats.rdev

- `<number>` | `<bigint>`

A numeric device identifier if the file is considered "special".

stats.size

- `<number>` | `<bigint>`

The size of the file in bytes.

stats.blksize

- `<number>` | `<bigint>`

The file system block size for i/o operations.

stats.blocks

- `<number>` | `<bigint>`

The number of blocks allocated for this file.

stats.atimeMs

- `<number>` | `<bigint>`

The timestamp indicating the last time this file was accessed expressed in milliseconds since the POSIX Epoch.

stats.mtimeMs

- `<number>` | `<bigint>`

The timestamp indicating the last time this file was modified expressed in milliseconds since the POSIX Epoch.

stats.ctimeMs

- `<number>` | `<bigint>`

The timestamp indicating the last time the file status was changed expressed in milliseconds since the POSIX Epoch.

stats.birthtimeMs

- `<number>` | `<bigint>`

The timestamp indicating the creation time of this file expressed in milliseconds since the POSIX Epoch.

stats.atime

- `<Date>`

The timestamp indicating the last time this file was accessed.

stats.mtime

- `<Date>`

The timestamp indicating the last time this file was modified.

stats.ctime

- `<Date>`

The timestamp indicating the last time the file status was changed.

stats.birthtime

- `<Date>`

The timestamp indicating the creation time of this file.

Stat Time Values

The `atimeMs`, `mtimeMs`, `ctimeMs`, `birthtimeMs` properties are numbers that hold the corresponding times in milliseconds. Their precision is platform specific. `atime`, `mtime`, `ctime`, and `birthtime` are `Date` object alternate representations of the various times. The `Date` and number values are not connected. Assigning a new number value, or mutating the `Date` value, will not be reflected in the corresponding alternate representation.

The times in the stat object have the following semantics:

- `atime` "Access Time" - Time when file data last accessed. Changed by the `mknod(2)`, `utimes(2)`, and `read(2)` system calls.
- `mtime` "Modified Time" - Time when file data last modified. Changed by the `mknod(2)`, `utimes(2)`, and `write(2)` system calls.
- `ctime` "Change Time" - Time when file status was last changed (inode data modification). Changed by the `chmod(2)`, `chown(2)`, `link(2)`, `mknod(2)`, `rename(2)`, `unlink(2)`, `utimes(2)`, `read(2)`, and `write(2)` system calls.
- `birthtime` "Birth Time" - Time of file creation. Set once when the file is created. On filesystems where birthtime is not available, this field may instead hold either the `ctime` or `1970-01-01T00:00Z` (ie, unix epoch timestamp `0`). This value may be greater than `atime` or `mtime` in this case. On Darwin and other FreeBSD variants, also set if the `atime` is explicitly set to an earlier value than the current `birthtime` using the `utimes(2)` system call.

Prior to Node.js 0.12, the `ctime` held the `birthtime` on Windows systems. As of 0.12, `ctime` is not "creation time", and on Unix systems, it never was.

Class: fs.WriteStream

`WriteStream` is a Writable Stream.

Event: 'close'

Emitted when the `WriteStream` 's underlying file descriptor has been closed.

Event: 'open'

- `fd` `<integer>` Integer file descriptor used by the `WriteStream` .

Emitted when the `WriteStream` 's file is opened.

Event: 'ready'

Emitted when the `fs.WriteStream` is ready to be used.

Fires immediately after `'open'` .

writeStream.bytesWritten

The number of bytes written so far. Does not include data that is still queued for writing.

writeStream.path

The path to the file the stream is writing to as specified in the first argument to `fs.createWriteStream()` . If `path` is passed as a string, then `writeStream.path` will be a string. If `path` is passed as a `Buffer` , then `writeStream.path` will be a `Buffer` .

writeStream.pending

- `<boolean>`

This property is `true` if the underlying file has not been opened yet, i.e. before the `'ready'` event is emitted.

fs.access(path[, mode], callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `mode` `<integer>` **Default:** `fs.constants.F_OK`
- `callback` `<Function>`
    - `err` `<Error>`

Tests a user's permissions for the file or directory specified by `path` . The `mode` argument is an optional integer that specifies the accessibility checks to be performed. Check File Access Constants for possible values of `mode` . It is possible to create a mask consisting of the bitwise OR of two or more values (e.g. `fs.constants.W_OK | fs.constants.R_OK`).

The final argument, `callback` , is a callback function that is invoked with a possible error argument. If any of the accessibility checks fail, the error argument will be an `Error` object. The following examples check if `package.json` exists, and if it is readable or writable.

```
const file = 'package.json';

// Check if the file exists in the current directory.
fs.access(file, fs.constants.F_OK, (err) => {
  console.log(`${file} ${err ? 'does not exist' : 'exists'}`);
});

// Check if the file is readable.
fs.access(file, fs.constants.R_OK, (err) => {
  console.log(`${file} ${err ? 'is not readable' : 'is readable'}`);
});

// Check if the file is writable.
fs.access(file, fs.constants.W_OK, (err) => {
  console.log(`${file} ${err ? 'is not writable' : 'is writable'}`);
});
```

```
});

// Check if the file exists in the current directory, and if it is writable.
fs.access(file, fs.constants.F_OK | fs.constants.W_OK, (err) => {
  if (err) {
    console.error(
      `${file} ${err.code === 'ENOENT' ? 'does not exist' : 'is read-only'}`);
  } else {
    console.log(`${file} exists, and it is writable`);
  }
});
```

Using `fs.access()` to check for the accessibility of a file before calling `fs.open()`, `fs.readFile()` or `fs.writeFile()` is not recommended. Doing so introduces a race condition, since other processes may change the file's state between the two calls. Instead, user code should open/read/write the file directly and handle the error raised if the file is not accessible.

write (NOT RECOMMENDED)

```
fs.access('myfile', (err) => {
  if (!err) {
    console.error('myfile already exists');
    return;
  }

  fs.open('myfile', 'wx', (err, fd) => {
    if (err) throw err;
    writeMyData(fd);
  });
});
```

write (RECOMMENDED)

```
fs.open('myfile', 'wx', (err, fd) => {
  if (err) {
    if (err.code === 'EEXIST') {
      console.error('myfile already exists');
      return;
    }

    throw err;
  }

  writeMyData(fd);
});
```

read (NOT RECOMMENDED)

```
fs.access('myfile', (err) => {
  if (err) {
    if (err.code === 'ENOENT') {
      console.error('myfile does not exist');
      return;
    }

    throw err;
  }
```

```
  fs.open('myfile', 'r', (err, fd) => {
    if (err) throw err;
    readMyData(fd);
  });
});
```

read (RECOMMENDED)

```
fs.open('myfile', 'r', (err, fd) => {
  if (err) {
    if (err.code === 'ENOENT') {
      console.error('myfile does not exist');
      return;
    }

    throw err;
  }

  readMyData(fd);
});
```

The "not recommended" examples above check for accessibility and then use the file; the "recommended" examples are better because they use the file directly and handle the error, if any.

In general, check for the accessibility of a file only if the file will not be used directly, for example when its accessibility is a signal from another process.

On Windows, access-control policies (ACLs) on a directory may limit access to a file or directory. The `fs.access()` function, however, does not check the ACL and therefore may report that a path is accessible even if the ACL restricts the user from reading or writing to it.

fs.accessSync(path[, mode])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `mode` `<integer>` **Default:** `fs.constants.F_OK`

Synchronously tests a user's permissions for the file or directory specified by `path`. The `mode` argument is an optional integer that specifies the accessibility checks to be performed. Check File Access Constants for possible values of `mode`. It is possible to create a mask consisting of the bitwise OR of two or more values (e.g. `fs.constants.W_OK | fs.constants.R_OK`).

If any of the accessibility checks fail, an `Error` will be thrown. Otherwise, the method will return `undefined`.

```
try {
  fs.accessSync('etc/passwd', fs.constants.R_OK | fs.constants.W_OK);
  console.log('can read/write');
} catch (err) {
  console.error('no access!');
}
```

fs.appendFile(path, data[, options], callback)

- `path` `<string>` | `<Buffer>` | `<URL>` | `<number>` filename or file descriptor
- `data` `<string>` | `<Buffer>`
- `options` `<Object>` | `<string>`
  - `encoding` `<string>` | `<null>` **Default:** `'utf8'`

    - `mode` `<integer>` **Default:** `0o666`
    - `flag` `<string>` See support of file system `flags`. **Default:** `'a'`.
- `callback` `<Function>`
    - `err` `<Error>`

Asynchronously append data to a file, creating the file if it does not yet exist. `data` can be a string or a `Buffer`.

```
fs.appendFile('message.txt', 'data to append', (err) => {
  if (err) throw err;
  console.log('The "data to append" was appended to file!');
});
```

If `options` is a string, then it specifies the encoding:

```
fs.appendFile('message.txt', 'data to append', 'utf8', callback);
```

The `path` may be specified as a numeric file descriptor that has been opened for appending (using `fs.open()` or `fs.openSync()`). The file descriptor will not be closed automatically.

```
fs.open('message.txt', 'a', (err, fd) => {
  if (err) throw err;
  fs.appendFile(fd, 'data to append', 'utf8', (err) => {
    fs.close(fd, (err) => {
      if (err) throw err;
    });
    if (err) throw err;
  });
});
```

fs.appendFileSync(path, data[, options])

- `path` `<string>` | `<Buffer>` | `<URL>` | `<number>` filename or file descriptor
- `data` `<string>` | `<Buffer>`
- `options` `<Object>` | `<string>`
    - `encoding` `<string>` | `<null>` **Default:** `'utf8'`
    - `mode` `<integer>` **Default:** `0o666`
    - `flag` `<string>` See support of file system `flags`. **Default:** `'a'`.

Synchronously append data to a file, creating the file if it does not yet exist. `data` can be a string or a `Buffer`.

```
try {
  fs.appendFileSync('message.txt', 'data to append');
  console.log('The "data to append" was appended to file!');
} catch (err) {
  /* Handle the error */
}
```

If `options` is a string, then it specifies the encoding:

```
fs.appendFileSync('message.txt', 'data to append', 'utf8');
```

The `path` may be specified as a numeric file descriptor that has been opened for appending (using `fs.open()` or `fs.openSync()`). The file descriptor will not be closed automatically.

```
let fd;

try {
  fd = fs.openSync('message.txt', 'a');
  fs.appendFileSync(fd, 'data to append', 'utf8');
} catch (err) {
  /* Handle the error */
} finally {
  if (fd !== undefined)
    fs.closeSync(fd);
}
```

fs.chmod(path, mode, callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `mode` `<integer>`
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronously changes the permissions of a file. No arguments other than a possible exception are given to the completion callback.

See also: `chmod(2)`.

File modes

The `mode` argument used in both the `fs.chmod()` and `fs.chmodSync()` methods is a numeric bitmask created using a logical OR of the following constants:

Constant	Octal	Description
`fs.constants.S_IRUSR`	`0o400`	read by owner
`fs.constants.S_IWUSR`	`0o200`	write by owner
`fs.constants.S_IXUSR`	`0o100`	execute/search by owner
`fs.constants.S_IRGRP`	`0o40`	read by group
`fs.constants.S_IWGRP`	`0o20`	write by group
`fs.constants.S_IXGRP`	`0o10`	execute/search by group
`fs.constants.S_IROTH`	`0o4`	read by others
`fs.constants.S_IWOTH`	`0o2`	write by others
`fs.constants.S_IXOTH`	`0o1`	execute/search by others

An easier method of constructing the `mode` is to use a sequence of three octal digits (e.g. `765`). The left-most digit (`7` in the example), specifies the permissions for the file owner. The middle digit (`6` in the example), specifies permissions for the group. The right-most digit (`5` in the example), specifies the permissions for others.

Number	Description
7	read, write, and execute
6	read and write
5	read and execute
4	read only
3	write and execute
2	write only
1	execute only
0	no permission

For example, the octal value `0o765` means:

- The owner may read, write and execute the file.
- The group may read and write the file.
- Others may read and execute the file.

When using raw numbers where file modes are expected, any value larger than `0o777` may result in platform-specific behaviors that are not supported to work consistently. Therefore constants like `S_ISVTX`, `S_ISGID` or `S_ISUID` are not exposed in `fs.constants`.

Caveats: on Windows only the write permission can be changed, and the distinction among the permissions of group, owner or others is not implemented.

fs.chmodSync(path, mode)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `mode` `<integer>`

For detailed information, see the documentation of the asynchronous version of this API: `fs.chmod()`.

See also: `chmod(2)`.

fs.chown(path, uid, gid, callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `uid` `<integer>`
- `gid` `<integer>`
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronously changes owner and group of a file. No arguments other than a possible exception are given to the completion callback.

See also: `chown(2)`.

fs.chownSync(path, uid, gid)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `uid` `<integer>`
- `gid` `<integer>`

Synchronously changes owner and group of a file. Returns `undefined`. This is the synchronous version of `fs.chown()`.

See also: `chown(2)`.

fs.close(fd, callback)

- `fd` `<integer>`
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronous `close(2)`. No arguments other than a possible exception are given to the completion callback.

fs.closeSync(fd)

- `fd` `<integer>`

Synchronous `close(2)`. Returns `undefined`.

fs.constants

- `<Object>`

Returns an object containing commonly used constants for file system operations. The specific constants currently defined are described in FS Constants.

fs.copyFile(src, dest[, flags], callback)

- `src` `<string>` | `<Buffer>` | `<URL>` source filename to copy
- `dest` `<string>` | `<Buffer>` | `<URL>` destination filename of the copy operation
- `flags` `<number>` modifiers for copy operation. **Default:** `0`.
- `callback` `<Function>`

Asynchronously copies `src` to `dest`. By default, `dest` is overwritten if it already exists. No arguments other than a possible exception are given to the callback function. Node.js makes no guarantees about the atomicity of the copy operation. If an error occurs after the destination file has been opened for writing, Node.js will attempt to remove the destination.

`flags` is an optional integer that specifies the behavior of the copy operation. It is possible to create a mask consisting of the bitwise OR of two or more values (e.g. `fs.constants.COPYFILE_EXCL | fs.constants.COPYFILE_FICLONE`).

- `fs.constants.COPYFILE_EXCL` - The copy operation will fail if `dest` already exists.
- `fs.constants.COPYFILE_FICLONE` - The copy operation will attempt to create a copy-on-write reflink. If the platform does not support copy-on-write, then a fallback copy mechanism is used.
- `fs.constants.COPYFILE_FICLONE_FORCE` - The copy operation will attempt to create a copy-on-write reflink. If the platform does not support copy-on-write, then the operation will fail.

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

// destination.txt will be created or overwritten by default.
fs.copyFile('source.txt', 'destination.txt', (err) => {
  if (err) throw err;
  console.log('source.txt was copied to destination.txt');
});
```

If the third argument is a number, then it specifies `flags`:

```
const fs = require('fs');
const { COPYFILE_EXCL } = fs.constants;
```

```
// By using COPYFILE_EXCL, the operation will fail if destination.txt exists.
fs.copyFile('source.txt', 'destination.txt', COPYFILE_EXCL, callback);
```

fs.copyFileSync(src, dest[, flags])

- `src` `<string>` | `<Buffer>` | `<URL>` source filename to copy
- `dest` `<string>` | `<Buffer>` | `<URL>` destination filename of the copy operation
- `flags` `<number>` modifiers for copy operation. **Default:** `0` .

Synchronously copies `src` to `dest` . By default, `dest` is overwritten if it already exists. Returns `undefined` . Node.js makes no guarantees about the atomicity of the copy operation. If an error occurs after the destination file has been opened for writing, Node.js will attempt to remove the destination.

`flags` is an optional integer that specifies the behavior of the copy operation. It is possible to create a mask consisting of the bitwise OR of two or more values (e.g. `fs.constants.COPYFILE_EXCL | fs.constants.COPYFILE_FICLONE`).

- `fs.constants.COPYFILE_EXCL` - The copy operation will fail if `dest` already exists.
- `fs.constants.COPYFILE_FICLONE` - The copy operation will attempt to create a copy-on-write reflink. If the platform does not support copy-on-write, then a fallback copy mechanism is used.
- `fs.constants.COPYFILE_FICLONE_FORCE` - The copy operation will attempt to create a copy-on-write reflink. If the platform does not support copy-on-write, then the operation will fail.

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

// destination.txt will be created or overwritten by default.
fs.copyFileSync('source.txt', 'destination.txt');
console.log('source.txt was copied to destination.txt');
```

If the third argument is a number, then it specifies `flags` :

```
const fs = require('fs');
const { COPYFILE_EXCL } = fs.constants;

// By using COPYFILE_EXCL, the operation will fail if destination.txt exists.
fs.copyFileSync('source.txt', 'destination.txt', COPYFILE_EXCL);
```

fs.createReadStream(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
  - `flags` `<string>` See support of file system `flags` . **Default:** `'r'` .
  - `encoding` `<string>` **Default:** `null`
  - `fd` `<integer>` **Default:** `null`
  - `mode` `<integer>` **Default:** `0o666`
  - `autoClose` `<boolean>` **Default:** `true`
  - `start` `<integer>`
  - `end` `<integer>` **Default:** `Infinity`
  - `highWaterMark` `<integer>` **Default:** `64 * 1024`
- Returns: `<fs.ReadStream>` See Readable Streams .

Unlike the 16 kb default `highWaterMark` for a readable stream, the stream returned by this method has a default `highWaterMark` of 64 kb.

`options` can include `start` and `end` values to read a range of bytes from the file instead of the entire file. Both `start` and `end` are inclusive and start counting at 0. If `fd` is specified and `start` is omitted or `undefined`, `fs.createReadStream()` reads sequentially from the current file position. The `encoding` can be any one of those accepted by `Buffer`.

If `fd` is specified, `ReadStream` will ignore the `path` argument and will use the specified file descriptor. This means that no `'open'` event will be emitted. `fd` should be blocking; non-blocking `fd`s should be passed to `net.Socket`.

If `fd` points to a character device that only supports blocking reads (such as keyboard or sound card), read operations do not finish until data is available. This can prevent the process from exiting and the stream from closing naturally.

```
const fs = require('fs');
// Create a stream from some character  device.
const stream = fs.createReadStream('/dev/input/event0');
setTimeout(() => {
  stream.close(); // This may not close the stream.
  // Artificially marking end-of-stream, as if the underlying resource had
  // indicated end-of-file by itself, allows the stream to close.
  // This does not cancel pending read operations, and if there is such an
  // operation, the process may still not be able to exit successfully
  // until it finishes.
  stream.push(null);
  stream.read(0);
}, 100);
```

If `autoClose` is false, then the file descriptor won't be closed, even if there's an error. It is the application's responsibility to close it and make sure there's no file descriptor leak. If `autoClose` is set to true (default behavior), on `'error'` or `'end'` the file descriptor will be closed automatically.

`mode` sets the file mode (permission and sticky bits), but only if the file was created.

An example to read the last 10 bytes of a file which is 100 bytes long:

```
fs.createReadStream('sample.txt', { start: 90, end: 99 });
```

If `options` is a string, then it specifies the encoding.

fs.createWriteStream(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
  - `flags` `<string>` See support of file system `flags`. **Default:** `'w'`.
  - `encoding` `<string>` **Default:** `'utf8'`
  - `fd` `<integer>` **Default:** `null`
  - `mode` `<integer>` **Default:** `0o666`
  - `autoClose` `<boolean>` **Default:** `true`
  - `start` `<integer>`
- Returns: `<fs.WriteStream>` See Writable Stream.

`options` may also include a `start` option to allow writing data at some position past the beginning of the file. Modifying a file rather than replacing it may require a `flags` mode of `r+` rather than the default mode `w`. The `encoding` can be any one of those accepted by `Buffer`.

If `autoClose` is set to true (default behavior) on `'error'` or `'finish'` the file descriptor will be closed automatically. If `autoClose` is false, then the file descriptor won't be closed, even if there's an error. It is the application's responsibility to close it and make sure there's no file descriptor leak.

Like `ReadStream`, if `fd` is specified, `WriteStream` will ignore the `path` argument and will use the specified file descriptor. This means that no `'open'` event will be emitted. `fd` should be blocking; non-blocking `fd`s should be passed to `net.Socket`.

If `options` is a string, then it specifies the encoding.

fs.exists(path, callback)

Stability: 0 - Deprecated: Use `fs.stat()` or `fs.access()` instead.

- `path` `<string>` | `<Buffer>` | `<URL>`
- `callback` `<Function>`
  - `exists` `<boolean>`

Test whether or not the given path exists by checking with the file system. Then call the `callback` argument with either true or false:

```
fs.exists('/etc/passwd', (exists) => {
  console.log(exists ? 'it\'s there' : 'no passwd!');
});
```

The parameters for this callback are not consistent with other Node.js callbacks. Normally, the first parameter to a Node.js callback is an `err` parameter, optionally followed by other parameters. The `fs.exists()` callback has only one boolean parameter. This is one reason `fs.access()` is recommended instead of `fs.exists()`.

Using `fs.exists()` to check for the existence of a file before calling `fs.open()`, `fs.readFile()` or `fs.writeFile()` is not recommended. Doing so introduces a race condition, since other processes may change the file's state between the two calls. Instead, user code should open/read/write the file directly and handle the error raised if the file does not exist.

write (NOT RECOMMENDED)

```
fs.exists('myfile', (exists) => {
  if (exists) {
    console.error('myfile already exists');
  } else {
    fs.open('myfile', 'wx', (err, fd) => {
      if (err) throw err;
      writeMyData(fd);
    });
  }
});
```

write (RECOMMENDED)

```
fs.open('myfile', 'wx', (err, fd) => {
  if (err) {
    if (err.code === 'EEXIST') {
      console.error('myfile already exists');
      return;
    }

    throw err;
  }

  writeMyData(fd);
});
```

read (NOT RECOMMENDED)

```
fs.exists('myfile', (exists) => {
  if (exists) {
    fs.open('myfile', 'r', (err, fd) => {
      if (err) throw err;
      readMyData(fd);
    });
  } else {
    console.error('myfile does not exist');
  }
});
```

read (RECOMMENDED)

```
fs.open('myfile', 'r', (err, fd) => {
  if (err) {
    if (err.code === 'ENOENT') {
      console.error('myfile does not exist');
      return;
    }

    throw err;
  }

  readMyData(fd);
});
```

The "not recommended" examples above check for existence and then use the file; the "recommended" examples are better because they use the file directly and handle the error, if any.

In general, check for the existence of a file only if the file won't be used directly, for example when its existence is a signal from another process.

fs.existsSync(path)

- `path` `<string>` | `<Buffer>` | `<URL>`
- Returns: `<boolean>`

Returns `true` if the path exists, `false` otherwise.

For detailed information, see the documentation of the asynchronous version of this API: `fs.exists()`.

`fs.exists()` is deprecated, but `fs.existsSync()` is not. The `callback` parameter to `fs.exists()` accepts parameters that are inconsistent with other Node.js callbacks. `fs.existsSync()` does not use a callback.

fs.fchmod(fd, mode, callback)

- `fd` `<integer>`
- `mode` `<integer>`
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronous `fchmod(2)`. No arguments other than a possible exception are given to the completion callback.

fs.fchmodSync(fd, mode)

- `fd` `<integer>`
- `mode` `<integer>`

Synchronous `fchmod(2)`. Returns `undefined`.

fs.fchown(fd, uid, gid, callback)

- `fd` `<integer>`
- `uid` `<integer>`
- `gid` `<integer>`
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronous `fchown(2)`. No arguments other than a possible exception are given to the completion callback.

fs.fchownSync(fd, uid, gid)

- `fd` `<integer>`
- `uid` `<integer>`
- `gid` `<integer>`

Synchronous `fchown(2)`. Returns `undefined`.

fs.fdatasync(fd, callback)

- `fd` `<integer>`
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronous `fdatasync(2)`. No arguments other than a possible exception are given to the completion callback.

fs.fdatasyncSync(fd)

- `fd` `<integer>`

Synchronous `fdatasync(2)`. Returns `undefined`.

fs.fstat(fd[, options], callback)

- `fd` `<integer>`
- `options` `<Object>`
  - `bigint` `<boolean>` Whether the numeric values in the returned `fs.Stats` object should be `bigint`. **Default:** `false`.
- `callback` `<Function>`
  - `err` `<Error>`
  - `stats` `<fs.Stats>`

Asynchronous `fstat(2)`. The callback gets two arguments `(err, stats)` where `stats` is an `fs.Stats` object. `fstat()` is identical to `stat()`, except that the file to be stat-ed is specified by the file descriptor `fd`.

fs.fstatSync(fd[, options])

- `fd` `<integer>`

- `options` `<Object>`
  - `bigint` `<boolean>` Whether the numeric values in the returned `fs.Stats` object should be `bigint`. **Default:** `false`.
- Returns: `<fs.Stats>`

Synchronous `fstat(2)`.

fs.fsync(fd, callback)

- `fd` `<integer>`
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronous `fsync(2)`. No arguments other than a possible exception are given to the completion callback.

fs.fsyncSync(fd)

- `fd` `<integer>`

Synchronous `fsync(2)`. Returns `undefined`.

fs.ftruncate(fd[, len], callback)

- `fd` `<integer>`
- `len` `<integer>` **Default:** `0`
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronous `ftruncate(2)`. No arguments other than a possible exception are given to the completion callback.

If the file referred to by the file descriptor was larger than `len` bytes, only the first `len` bytes will be retained in the file.

For example, the following program retains only the first four bytes of the file:

```
console.log(fs.readFileSync('temp.txt', 'utf8'));
// Prints: Node.js

// get the file descriptor of the file to be truncated
const fd = fs.openSync('temp.txt', 'r+');

// truncate the file to first four bytes
fs.ftruncate(fd, 4, (err) => {
  assert.ifError(err);
  console.log(fs.readFileSync('temp.txt', 'utf8'));
});
// Prints: Node
```

If the file previously was shorter than `len` bytes, it is extended, and the extended part is filled with null bytes (`'\0'`):

```
console.log(fs.readFileSync('temp.txt', 'utf8'));
// Prints: Node.js

// get the file descriptor of the file to be truncated
const fd = fs.openSync('temp.txt', 'r+');

// truncate the file to 10 bytes, whereas the actual size is 7 bytes
fs.ftruncate(fd, 10, (err) => {
```

```
    assert.ifError(err);
    console.log(fs.readFileSync('temp.txt'));
});
// Prints: <Buffer 4e 6f 64 65 2e 6a 73 00 00 00>
// ('Node.js\0\0\0' in UTF8)
```

The last three bytes are null bytes (`'\0'`), to compensate the over-truncation.

fs.ftruncateSync(fd[, len])

- `fd` `<integer>`
- `len` `<integer>` **Default:** `0`

Returns `undefined` .

For detailed information, see the documentation of the asynchronous version of this API: `fs.ftruncate()` .

fs.futimes(fd, atime, mtime, callback)

- `fd` `<integer>`
- `atime` `<number>` | `<string>` | `<Date>`
- `mtime` `<number>` | `<string>` | `<Date>`
- `callback` `<Function>`
  - `err` `<Error>`

Change the file system timestamps of the object referenced by the supplied file descriptor. See `fs.utimes()` .

This function does not work on AIX versions before 7.1, it will return the error `UV_ENOSYS` .

fs.futimesSync(fd, atime, mtime)

- `fd` `<integer>`
- `atime` `<integer>`
- `mtime` `<integer>`

Synchronous version of `fs.futimes()` . Returns `undefined` .

fs.lchmod(path, mode, callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `mode` `<integer>`
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronous `lchmod(2)` . No arguments other than a possible exception are given to the completion callback.

Only available on macOS.

fs.lchmodSync(path, mode)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `mode` `<integer>`

Synchronous `lchmod(2)` . Returns `undefined` .

fs.lchown(path, uid, gid, callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `uid` `<integer>`
- `gid` `<integer>`
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronous `lchown(2)`. No arguments other than a possible exception are given to the completion callback.

fs.lchownSync(path, uid, gid)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `uid` `<integer>`
- `gid` `<integer>`

Synchronous `lchown(2)`. Returns `undefined`.

fs.link(existingPath, newPath, callback)

- `existingPath` `<string>` | `<Buffer>` | `<URL>`
- `newPath` `<string>` | `<Buffer>` | `<URL>`
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronous `link(2)`. No arguments other than a possible exception are given to the completion callback.

fs.linkSync(existingPath, newPath)

- `existingPath` `<string>` | `<Buffer>` | `<URL>`
- `newPath` `<string>` | `<Buffer>` | `<URL>`

Synchronous `link(2)`. Returns `undefined`.

fs.lstat(path[, options], callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
  - `bigint` `<boolean>` Whether the numeric values in the returned `fs.Stats` object should be `bigint`. **Default:** `false`.
- `callback` `<Function>`
  - `err` `<Error>`
  - `stats` `<fs.Stats>`

Asynchronous `lstat(2)`. The callback gets two arguments `(err, stats)` where `stats` is a `fs.Stats` object. `lstat()` is identical to `stat()`, except that if `path` is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

fs.lstatSync(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
  - `bigint` `<boolean>` Whether the numeric values in the returned `fs.Stats` object should be `bigint`. **Default:** `false`.
- Returns: `<fs.Stats>`

Synchronous `lstat(2)`.

fs.mkdir(path[, options], callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>` | `<integer>`
  - `recursive` `<boolean>` **Default:** `false`
  - `mode` `<integer>` Not supported on Windows. **Default:** `0o777`.
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronously creates a directory. No arguments other than a possible exception are given to the completion callback.

The optional `options` argument can be an integer specifying mode (permission and sticky bits), or an object with a `mode` property and a `recursive` property indicating whether parent folders should be created.

```
// Creates /tmp/a/apple, regardless of whether `/tmp` and /tmp/a exist.
fs.mkdir('/tmp/a/apple', { recursive: true }, (err) => {
  if (err) throw err;
});
```

On Windows, using `fs.mkdir()` on the root directory even with recursion will result in an error:

```
fs.mkdir('/', { recursive: true }, (err) => {
  // => [Error: EPERM: operation not permitted, mkdir 'C:\']
});
```

See also: `mkdir(2)`.

fs.mkdirSync(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>` | `<integer>`
  - `recursive` `<boolean>` **Default:** `false`
  - `mode` `<integer>` Not supported on Windows. **Default:** `0o777`.

Synchronously creates a directory. Returns `undefined`. This is the synchronous version of `fs.mkdir()`.

See also: `mkdir(2)`.

fs.mkdtemp(prefix[, options], callback)

- `prefix` `<string>`
- `options` `<string>` | `<Object>`
  - `encoding` `<string>` **Default:** `'utf8'`
- `callback` `<Function>`
  - `err` `<Error>`
  - `folder` `<string>`

Creates a unique temporary directory.

Generates six random characters to be appended behind a required `prefix` to create a unique temporary directory.

The created folder path is passed as a string to the callback's second parameter.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use.

```
fs.mkdtemp(path.join(os.tmpdir(), 'foo-'), (err, folder) => {
  if (err) throw err;
  console.log(folder);
  // Prints: /tmp/foo-itXde2 or C:\Users\...\AppData\Local\Temp\foo-itXde2
});
```

The `fs.mkdtemp()` method will append the six randomly selected characters directly to the `prefix` string. For instance, given a directory `/tmp`, if the intention is to create a temporary directory *within* `/tmp`, the `prefix` must end with a trailing platform-specific path separator (`require('path').sep`).

```
// The parent directory for the new temporary directory
const tmpDir = os.tmpdir();

// This method is *INCORRECT*:
fs.mkdtemp(tmpDir, (err, folder) => {
  if (err) throw err;
  console.log(folder);
  // Will print something similar to `/tmpabc123`.
  // A new temporary directory is created at the file system root
  // rather than *within* the /tmp directory.
});

// This method is *CORRECT*:
const { sep } = require('path');
fs.mkdtemp(`${tmpDir}${sep}`, (err, folder) => {
  if (err) throw err;
  console.log(folder);
  // Will print something similar to `/tmp/abc123`.
  // A new temporary directory is created within
  // the /tmp directory.
});
```

fs.mkdtempSync(prefix[, options])

- `prefix` `<string>`
- `options` `<string>` | `<Object>`
  - `encoding` `<string>` **Default:** `'utf8'`
- Returns: `<string>`

Returns the created folder path.

For detailed information, see the documentation of the asynchronous version of this API: `fs.mkdtemp()`.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use.

fs.open(path[, flags[, mode]], callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `flags` `<string>` | `<number>` See support of file system `flags`. **Default:** `'r'`.
- `mode` `<integer>` **Default:** `0o666` (readable and writable)
- `callback` `<Function>`
  - `err` `<Error>`
  - `fd` `<integer>`

Asynchronous file open. See `open(2)`.

`mode` sets the file mode (permission and sticky bits), but only if the file was created. On Windows, only the write permission can be manipulated; see `fs.chmod()`.

The callback gets two arguments `(err, fd)`.

Some characters (`< > : " / \ | ? *`) are reserved under Windows as documented by Naming Files, Paths, and Namespaces. Under NTFS, if the filename contains a colon, Node.js will open a file system stream, as described by this MSDN page.

Functions based on `fs.open()` exhibit this behavior as well: `fs.writeFile()`, `fs.readFile()`, etc.

fs.openSync(path[, flags, mode])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `flags` `<string>` | `<number>` **Default:** `'r'`. See support of file system `flags`.
- `mode` `<integer>` **Default:** `0o666`
- Returns: `<number>`

Returns an integer representing the file descriptor.

For detailed information, see the documentation of the asynchronous version of this API: `fs.open()`.

fs.read(fd, buffer, offset, length, position, callback)

- `fd` `<integer>`
- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>`
- `offset` `<integer>`
- `length` `<integer>`
- `position` `<integer>`
- `callback` `<Function>`
  - `err` `<Error>`
  - `bytesRead` `<integer>`
  - `buffer` `<Buffer>`

Read data from the file specified by `fd`.

`buffer` is the buffer that the data will be written to.

`offset` is the offset in the buffer to start writing at.

`length` is an integer specifying the number of bytes to read.

`position` is an argument specifying where to begin reading from in the file. If `position` is `null`, data will be read from the current file position, and the file position will be updated. If `position` is an integer, the file position will remain unchanged.

The callback is given the three arguments, `(err, bytesRead, buffer)`.

If this method is invoked as its `util.promisify()` ed version, it returns a `Promise` for an `Object` with `bytesRead` and `buffer` properties.

fs.readdir(path[, options], callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
  - `encoding` `<string>` **Default:** `'utf8'`
  - `withFileTypes` `<boolean>` **Default:** `false`
- `callback` `<Function>`
  - `err` `<Error>`
  - `files` `<string[]>` | `<Buffer[]>` | `<fs.Dirent[]>`

Asynchronous `readdir(3)` . Reads the contents of a directory. The callback gets two arguments `(err, files)` where `files` is an array of the names of the files in the directory excluding `'.'` and `'..'` .

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the filenames passed to the callback. If the `encoding` is set to `'buffer'` , the filenames returned will be passed as `Buffer` objects.

If `options.withFileTypes` is set to `true` , the `files` array will contain `fs.Dirent` objects.

fs.readdirSync(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
  - `encoding` `<string>` **Default:** `'utf8'`
  - `withFileTypes` `<boolean>` **Default:** `false`
- Returns: `<string[]>` | `<Buffer[]>` | `<fs.Dirent[]>`

Synchronous `readdir(3)` .

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the filenames returned. If the `encoding` is set to `'buffer'` , the filenames returned will be passed as `Buffer` objects.

If `options.withFileTypes` is set to `true` , the result will contain `fs.Dirent` objects.

fs.readFile(path[, options], callback)

- `path` `<string>` | `<Buffer>` | `<URL>` | `<integer>` filename or file descriptor
- `options` `<Object>` | `<string>`
  - `encoding` `<string>` | `<null>` **Default:** `null`
  - `flag` `<string>` See support of file system `flags` . **Default:** `'r'` .
- `callback` `<Function>`
  - `err` `<Error>`
  - `data` `<string>` | `<Buffer>`

Asynchronously reads the entire contents of a file.

```
fs.readFile('/etc/passwd', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

The callback is passed two arguments `(err, data)`, where `data` is the contents of the file.

If no encoding is specified, then the raw buffer is returned.

If `options` is a string, then it specifies the encoding:

```
fs.readFile('/etc/passwd', 'utf8', callback);
```

When the path is a directory, the behavior of `fs.readFile()` and `fs.readFileSync()` is platform-specific. On macOS, Linux, and Windows, an error will be returned. On FreeBSD, a representation of the directory's contents will be returned.

```
// macOS, Linux, and Windows
fs.readFile('<directory>', (err, data) => {
  // => [Error: EISDIR: illegal operation on a directory, read <directory>]
});

//  FreeBSD
fs.readFile('<directory>', (err, data) => {
  // => null, <data>
});
```

The `fs.readFile()` function buffers the entire file. To minimize memory costs, when possible prefer streaming via `fs.createReadStream()`.

File Descriptors

1. Any specified file descriptor has to support reading.
2. If a file descriptor is specified as the `path`, it will not be closed automatically.
3. The reading will begin at the current position. For example, if the file already had `'Hello World'` and six bytes are read with the file descriptor, the call to `fs.readFile()` with the same file descriptor, would give `'World'`, rather than `'Hello World'`.

fs.readFileSync(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>` | `<integer>` filename or file descriptor
- `options` `<Object>` | `<string>`
  - `encoding` `<string>` | `<null>` **Default:** `null`
  - `flag` `<string>` See support of file system `flags`. **Default:** `'r'`.
- Returns: `<string>` | `<Buffer>`

Returns the contents of the `path`.

For detailed information, see the documentation of the asynchronous version of this API: `fs.readFile()`.

If the `encoding` option is specified then this function returns a string. Otherwise it returns a buffer.

Similar to `fs.readFile()`, when the path is a directory, the behavior of `fs.readFileSync()` is platform-specific.

```
// macOS, Linux, and Windows
fs.readFileSync('<directory>');
// => [Error: EISDIR: illegal operation on a directory, read <directory>]

//  FreeBSD
fs.readFileSync('<directory>'); // => <data>
```

fs.readlink(path[, options], callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
  - `encoding` `<string>` **Default:** `'utf8'`
- `callback` `<Function>`
  - `err` `<Error>`
  - `linkString` `<string>` | `<Buffer>`

Asynchronous `readlink(2)`. The callback gets two arguments `(err, linkString)`.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the link path passed to the callback. If the `encoding` is set to `'buffer'`, the link path returned will be passed as a `Buffer` object.

fs.readlinkSync(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
  - `encoding` `<string>` **Default:** `'utf8'`
- Returns: `<string>` | `<Buffer>`

Synchronous `readlink(2)`. Returns the symbolic link's string value.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the link path returned. If the `encoding` is set to `'buffer'`, the link path returned will be passed as a `Buffer` object.

fs.readSync(fd, buffer, offset, length, position)

- `fd` `<integer>`
- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>`
- `offset` `<integer>`
- `length` `<integer>`
- `position` `<integer>`
- Returns: `<number>`

Returns the number of `bytesRead`.

For detailed information, see the documentation of the asynchronous version of this API: `fs.read()`.

fs.realpath(path[, options], callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
  - `encoding` `<string>` **Default:** `'utf8'`
- `callback` `<Function>`
  - `err` `<Error>`
  - `resolvedPath` `<string>` | `<Buffer>`

Asynchronously computes the canonical pathname by resolving `.`, `..` and symbolic links.

A canonical pathname is not necessarily unique. Hard links and bind mounts can expose a file system entity through many pathnames.

This function behaves like `realpath(3)`, with some exceptions:

1. No case conversion is performed on case-insensitive file systems.
2. The maximum number of symbolic links is platform-independent and generally (much) higher than what the native `realpath(3)` implementation supports.

The `callback` gets two arguments `(err, resolvedPath)`. May use `process.cwd` to resolve relative paths.

Only paths that can be converted to UTF8 strings are supported.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the path passed to the callback. If the `encoding` is set to `'buffer'`, the path returned will be passed as a `Buffer` object.

If `path` resolves to a socket or a pipe, the function will return a system dependent name for that object.

fs.realpath.native(path[, options], callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
  - `encoding` `<string>` **Default:** `'utf8'`
- `callback` `<Function>`
  - `err` `<Error>`
  - `resolvedPath` `<string>` | `<Buffer>`

Asynchronous `realpath(3)`.

The `callback` gets two arguments `(err, resolvedPath)`.

Only paths that can be converted to UTF8 strings are supported.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the path passed to the callback. If the `encoding` is set to `'buffer'`, the path returned will be passed as a `Buffer` object.

On Linux, when Node.js is linked against musl libc, the procfs file system must be mounted on `/proc` in order for this function to work. Glibc does not have this restriction.

fs.realpathSync(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
  - `encoding` `<string>` **Default:** `'utf8'`
- Returns: `<string>` | `<Buffer>`

Returns the resolved pathname.

For detailed information, see the documentation of the asynchronous version of this API: `fs.realpath()`.

fs.realpathSync.native(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
  - `encoding` `<string>` **Default:** `'utf8'`
- Returns: `<string>` | `<Buffer>`

Synchronous `realpath(3)`.

Only paths that can be converted to UTF8 strings are supported.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the path returned. If the `encoding` is set to `'buffer'`, the path returned will be passed as a `Buffer` object.

On Linux, when Node.js is linked against musl libc, the procfs file system must be mounted on `/proc` in order for this function to work. Glibc does not have this restriction.

fs.rename(oldPath, newPath, callback)

- `oldPath` `<string>` | `<Buffer>` | `<URL>`
- `newPath` `<string>` | `<Buffer>` | `<URL>`
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronously rename file at `oldPath` to the pathname provided as `newPath`. In the case that `newPath` already exists, it will be overwritten. No arguments other than a possible exception are given to the completion callback.

See also: `rename(2)`.

```
fs.rename('oldFile.txt', 'newFile.txt', (err) => {
  if (err) throw err;
  console.log('Rename complete!');
});
```

fs.renameSync(oldPath, newPath)

- `oldPath` `<string>` | `<Buffer>` | `<URL>`
- `newPath` `<string>` | `<Buffer>` | `<URL>`

Synchronous `rename(2)`. Returns `undefined`.

fs.rmdir(path, callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronous `rmdir(2)`. No arguments other than a possible exception are given to the completion callback.

Using `fs.rmdir()` on a file (not a directory) results in an `ENOENT` error on Windows and an `ENOTDIR` error on POSIX.

fs.rmdirSync(path)

- `path` `<string>` | `<Buffer>` | `<URL>`

Synchronous `rmdir(2)`. Returns `undefined`.

Using `fs.rmdirSync()` on a file (not a directory) results in an `ENOENT` error on Windows and an `ENOTDIR` error on POSIX.

fs.stat(path[, options], callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
  - `bigint` `<boolean>` Whether the numeric values in the returned `fs.Stats` object should be `bigint`. **Default:** `false`.

- `callback` `<Function>`
  - `err` `<Error>`
  - `stats` `<fs.Stats>`

Asynchronous `stat(2)`. The callback gets two arguments `(err, stats)` where `stats` is an `fs.Stats` object.

In case of an error, the `err.code` will be one of Common System Errors.

Using `fs.stat()` to check for the existence of a file before calling `fs.open()`, `fs.readFile()` or `fs.writeFile()` is not recommended. Instead, user code should open/read/write the file directly and handle the error raised if the file is not available.

To check if a file exists without manipulating it afterwards, `fs.access()` is recommended.

fs.statSync(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
  - `bigint` `<boolean>` Whether the numeric values in the returned `fs.Stats` object should be `bigint`. **Default:** `false`.
- Returns: `<fs.Stats>`

Synchronous `stat(2)`.

fs.symlink(target, path[, type], callback)

- `target` `<string>` | `<Buffer>` | `<URL>`
- `path` `<string>` | `<Buffer>` | `<URL>`
- `type` `<string>` **Default:** `'file'`
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronous `symlink(2)`. No arguments other than a possible exception are given to the completion callback. The `type` argument can be set to `'dir'`, `'file'`, or `'junction'` and is only available on Windows (ignored on other platforms). Windows junction points require the destination path to be absolute. When using `'junction'`, the `target` argument will automatically be normalized to absolute path.

Here is an example below:

```
fs.symlink('./foo', './new-port', callback);
```

It creates a symbolic link named "new-port" that points to "foo".

fs.symlinkSync(target, path[, type])

- `target` `<string>` | `<Buffer>` | `<URL>`
- `path` `<string>` | `<Buffer>` | `<URL>`
- `type` `<string>` **Default:** `'file'`

Returns `undefined`.

For detailed information, see the documentation of the asynchronous version of this API: `fs.symlink()`.

fs.truncate(path[, len], callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `len` `<integer>` **Default:** `0`

- `callback` `<Function>`
  - `err` `<Error>`

Asynchronous `truncate(2)`. No arguments other than a possible exception are given to the completion callback. A file descriptor can also be passed as the first argument. In this case, `fs.ftruncate()` is called.

Passing a file descriptor is deprecated and may result in an error being thrown in the future.

fs.truncateSync(path[, len])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `len` `<integer>` **Default:** `0`

Synchronous `truncate(2)`. Returns `undefined`. A file descriptor can also be passed as the first argument. In this case, `fs.ftruncateSync()` is called.

Passing a file descriptor is deprecated and may result in an error being thrown in the future.

fs.unlink(path, callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronously removes a file or symbolic link. No arguments other than a possible exception are given to the completion callback.

```
// Assuming that 'path/file.txt' is a regular file.
fs.unlink('path/file.txt', (err) => {
  if (err) throw err;
  console.log('path/file.txt was deleted');
});
```

`fs.unlink()` will not work on a directory, empty or otherwise. To remove a directory, use `fs.rmdir()`.

See also: `unlink(2)`.

fs.unlinkSync(path)

- `path` `<string>` | `<Buffer>` | `<URL>`

Synchronous `unlink(2)`. Returns `undefined`.

fs.unwatchFile(filename[, listener])

- `filename` `<string>` | `<Buffer>` | `<URL>`
- `listener` `<Function>` Optional, a listener previously attached using `fs.watchFile()`

Stop watching for changes on `filename`. If `listener` is specified, only that particular listener is removed. Otherwise, *all* listeners are removed, effectively stopping watching of `filename`.

Calling `fs.unwatchFile()` with a filename that is not being watched is a no-op, not an error.

Using `fs.watch()` is more efficient than `fs.watchFile()` and `fs.unwatchFile()`. `fs.watch()` should be used instead of `fs.watchFile()` and `fs.unwatchFile()` when possible.

fs.utimes(path, atime, mtime, callback)

- `path` `<string>` | `<Buffer>` | `<URL>`

- `atime` `<number>` | `<string>` | `<Date>`
- `mtime` `<number>` | `<string>` | `<Date>`
- `callback` `<Function>`
  - `err` `<Error>`

Change the file system timestamps of the object referenced by `path`.

The `atime` and `mtime` arguments follow these rules:

- Values can be either numbers representing Unix epoch time, `Date`s, or a numeric string like `'123456789.0'`.
- If the value can not be converted to a number, or is `NaN`, `Infinity` or `-Infinity`, an `Error` will be thrown.

fs.utimesSync(path, atime, mtime)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `atime` `<integer>`
- `mtime` `<integer>`

Returns `undefined`.

For detailed information, see the documentation of the asynchronous version of this API: `fs.utimes()`.

fs.watch(filename[, options][, listener])

- `filename` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
  - `persistent` `<boolean>` Indicates whether the process should continue to run as long as files are being watched. **Default:** `true`.
  - `recursive` `<boolean>` Indicates whether all subdirectories should be watched, or only the current directory. This applies when a directory is specified, and only on supported platforms (See Caveats). **Default:** `false`.
  - `encoding` `<string>` Specifies the character encoding to be used for the filename passed to the listener. **Default:** `'utf8'`.
- `listener` `<Function>` | `<undefined>` **Default:** `undefined`
  - `eventType` `<string>`
  - `filename` `<string>` | `<Buffer>`
- Returns: `<fs.FSWatcher>`

Watch for changes on `filename`, where `filename` is either a file or a directory.

The second argument is optional. If `options` is provided as a string, it specifies the `encoding`. Otherwise `options` should be passed as an object.

The listener callback gets two arguments `(eventType, filename)`. `eventType` is either `'rename'` or `'change'`, and `filename` is the name of the file which triggered the event.

On most platforms, `'rename'` is emitted whenever a filename appears or disappears in the directory.

The listener callback is attached to the `'change'` event fired by `fs.FSWatcher`, but it is not the same thing as the `'change'` value of `eventType`.

Caveats

The `fs.watch` API is not 100% consistent across platforms, and is unavailable in some situations.

The recursive option is only supported on macOS and Windows.

Availability

This feature depends on the underlying operating system providing a way to be notified of filesystem changes.

- On Linux systems, this uses `inotify(7)`.
- On BSD systems, this uses `kqueue(2)`.
- On macOS, this uses `kqueue(2)` for files and `FSEvents` for directories.
- On SunOS systems (including Solaris and SmartOS), this uses `event ports`.
- On Windows systems, this feature depends on `ReadDirectoryChangesW`.
- On Aix systems, this feature depends on `AHAFS`, which must be enabled.

If the underlying functionality is not available for some reason, then `fs.watch` will not be able to function. For example, watching files or directories can be unreliable, and in some cases impossible, on network file systems (NFS, SMB, etc), or host file systems when using virtualization software such as Vagrant, Docker, etc.

It is still possible to use `fs.watchFile()`, which uses stat polling, but this method is slower and less reliable.

Inodes

On Linux and macOS systems, `fs.watch()` resolves the path to an inode and watches the inode. If the watched path is deleted and recreated, it is assigned a new inode. The watch will emit an event for the delete but will continue watching the *original* inode. Events for the new inode will not be emitted. This is expected behavior.

AIX files retain the same inode for the lifetime of a file. Saving and closing a watched file on AIX will result in two notifications (one for adding new content, and one for truncation).

Filename Argument

Providing `filename` argument in the callback is only supported on Linux, macOS, Windows, and AIX. Even on supported platforms, `filename` is not always guaranteed to be provided. Therefore, don't assume that `filename` argument is always provided in the callback, and have some fallback logic if it is `null`.

```
fs.watch('somedir', (eventType, filename) => {
  console.log(`event type is: ${eventType}`);
  if (filename) {
    console.log(`filename provided: ${filename}`);
  } else {
    console.log('filename not provided');
  }
});
```

fs.watchFile(filename[, options], listener)

- `filename` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
  - `persistent` `<boolean>` **Default:** `true`
  - `interval` `<integer>` **Default:** `5007`
- `listener` `<Function>`
  - `current` `<fs.Stats>`
  - `previous` `<fs.Stats>`

Watch for changes on `filename`. The callback `listener` will be called each time the file is accessed.

The `options` argument may be omitted. If provided, it should be an object. The `options` object may contain a boolean named `persistent` that indicates whether the process should continue to run as long as files are being watched. The `options` object may specify an `interval` property indicating how often the target should be polled in milliseconds.

The `listener` gets two arguments the current stat object and the previous stat object:

```
fs.watchFile('message.text', (curr, prev) => {
  console.log(`the current mtime is: ${curr.mtime}`);
  console.log(`the previous mtime was: ${prev.mtime}`);
});
```

These stat objects are instances of `fs.Stat`.

To be notified when the file was modified, not just accessed, it is necessary to compare `curr.mtime` and `prev.mtime`.

When an `fs.watchFile` operation results in an `ENOENT` error, it will invoke the listener once, with all the fields zeroed (or, for dates, the Unix Epoch). In Windows, `blksize` and `blocks` fields will be `undefined`, instead of zero. If the file is created later on, the listener will be called again, with the latest stat objects. This is a change in functionality since v0.10.

Using `fs.watch()` is more efficient than `fs.watchFile` and `fs.unwatchFile`. `fs.watch` should be used instead of `fs.watchFile` and `fs.unwatchFile` when possible.

When a file being watched by `fs.watchFile()` disappears and reappears, then the `previousStat` reported in the second callback event (the file's reappearance) will be the same as the `previousStat` of the first callback event (its disappearance).

This happens when:

- the file is deleted, followed by a restore
- the file is renamed twice - the second time back to its original name

fs.write(fd, buffer[, offset[, length[, position]]], callback)

- `fd` `<integer>`
- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>`
- `offset` `<integer>`
- `length` `<integer>`
- `position` `<integer>`
- `callback` `<Function>`
  - `err` `<Error>`
  - `bytesWritten` `<integer>`
  - `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>`

Write `buffer` to the file specified by `fd`.

`offset` determines the part of the buffer to be written, and `length` is an integer specifying the number of bytes to write.

`position` refers to the offset from the beginning of the file where this data should be written. If `typeof position !== 'number'`, the data will be written at the current position. See `pwrite(2)`.

The callback will be given three arguments `(err, bytesWritten, buffer)` where `bytesWritten` specifies how many *bytes* were written from `buffer`.

If this method is invoked as its `util.promisify()` ed version, it returns a `Promise` for an `Object` with `bytesWritten` and `buffer` properties.

It is unsafe to use `fs.write()` multiple times on the same file without waiting for the callback. For this scenario, `fs.createWriteStream()` is recommended.

On Linux, positional writes don't work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

fs.write(fd, string[, position[, encoding]], callback)

- `fd` `<integer>`
- `string` `<string>`
- `position` `<integer>`
- `encoding` `<string>` **Default:** `'utf8'`
- `callback` `<Function>`
  - `err` `<Error>`
  - `written` `<integer>`
  - `string` `<string>`

Write `string` to the file specified by `fd`. If `string` is not a string, then the value will be coerced to one.

`position` refers to the offset from the beginning of the file where this data should be written. If `typeof position !== 'number'` the data will be written at the current position. See `pwrite(2)`.

`encoding` is the expected string encoding.

The callback will receive the arguments `(err, written, string)` where `written` specifies how many *bytes* the passed string required to be written. Bytes written is not necessarily the same as string characters written. See `Buffer.byteLength`.

It is unsafe to use `fs.write()` multiple times on the same file without waiting for the callback. For this scenario, `fs.createWriteStream()` is recommended.

On Linux, positional writes don't work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

On Windows, if the file descriptor is connected to the console (e.g. `fd == 1` or `stdout`) a string containing non-ASCII characters will not be rendered properly by default, regardless of the encoding used. It is possible to configure the console to render UTF-8 properly by changing the active codepage with the `chcp 65001` command. See the `chcp` docs for more details.

fs.writeFile(file, data[, options], callback)

- `file` `<string>` | `<Buffer>` | `<URL>` | `<integer>` filename or file descriptor
- `data` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `options` `<Object>` | `<string>`
  - `encoding` `<string>` | `<null>` **Default:** `'utf8'`
  - `mode` `<integer>` **Default:** `0o666`
  - `flag` `<string>` See support of file system `flags`. **Default:** `'w'`.
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronously writes data to a file, replacing the file if it already exists. `data` can be a string or a buffer.

The `encoding` option is ignored if `data` is a buffer.

```
const data = new Uint8Array(Buffer.from('Hello Node.js'));
fs.writeFile('message.txt', data, (err) => {
  if (err) throw err;
  console.log('The file has been saved!');
});
```

If `options` is a string, then it specifies the encoding:

```
fs.writeFile('message.txt', 'Hello Node.js', 'utf8', callback);
```

It is unsafe to use `fs.writeFile()` multiple times on the same file without waiting for the callback. For this scenario, `fs.createWriteStream()` is recommended.

File Descriptors

1. Any specified file descriptor has to support writing.
2. If a file descriptor is specified as the `file`, it will not be closed automatically.
3. The writing will begin at the beginning of the file. For example, if the file already had `'Hello World'` and the newly written content is `'Aloha'`, then the contents of the file would be `'Aloha World'`, rather than just `'Aloha'`.

fs.writeFileSync(file, data[, options])

- `file` `<string>` | `<Buffer>` | `<URL>` | `<integer>` filename or file descriptor
- `data` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `options` `<Object>` | `<string>`
  - `encoding` `<string>` | `<null>` **Default:** `'utf8'`
  - `mode` `<integer>` **Default:** `0o666`
  - `flag` `<string>` See support of file system `flags`. **Default:** `'w'`.

Returns `undefined`.

For detailed information, see the documentation of the asynchronous version of this API: `fs.writeFile()`.

fs.writeSync(fd, buffer[, offset[, length[, position]]])

- `fd` `<integer>`
- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>`
- `offset` `<integer>`
- `length` `<integer>`
- `position` `<integer>`
- Returns: `<number>` The number of bytes written.

For detailed information, see the documentation of the asynchronous version of this API: `fs.write(fd, buffer...)`.

fs.writeSync(fd, string[, position[, encoding]])

- `fd` `<integer>`
- `string` `<string>`
- `position` `<integer>`
- `encoding` `<string>`
- Returns: `<number>` The number of bytes written.

For detailed information, see the documentation of the asynchronous version of this API: `fs.write(fd, string...)`.

fs Promises API

Stability: 1 - Experimental

The `fs.promises` API provides an alternative set of asynchronous file system methods that return `Promise` objects rather than using callbacks. The API is accessible via `require('fs').promises`.

class: FileHandle

A `FileHandle` object is a wrapper for a numeric file descriptor. Instances of `FileHandle` are distinct from numeric file descriptors in that, if the `FileHandle` is not explicitly closed using the `filehandle.close()` method, they will automatically close the file descriptor and will emit a process warning, thereby helping to prevent memory leaks.

Instances of the `FileHandle` object are created internally by the `fsPromises.open()` method.

Unlike the callback-based API (`fs.fstat()`, `fs.fchown()`, `fs.fchmod()`, and so on), a numeric file descriptor is not used by the promise-based API. Instead, the promise-based API uses the `FileHandle` class in order to help avoid accidental leaking of unclosed file descriptors after a `Promise` is resolved or rejected.

filehandle.appendFile(data, options)

- `data` `<string>` | `<Buffer>`
- `options` `<Object>` | `<string>`
  - `encoding` `<string>` | `<null>` **Default:** `'utf8'`
  - `mode` `<integer>` **Default:** `0o666`
  - `flag` `<string>` See support of file system `flags`. **Default:** `'a'`.
- Returns: `<Promise>`

Asynchronously append data to this file, creating the file if it does not yet exist. `data` can be a string or a `Buffer`. The `Promise` will be resolved with no arguments upon success.

If `options` is a string, then it specifies the encoding.

The `FileHandle` must have been opened for appending.

filehandle.chmod(mode)

- `mode` `<integer>`
- Returns: `<Promise>`

Modifies the permissions on the file. The `Promise` is resolved with no arguments upon success.

filehandle.chown(uid, gid)

- `uid` `<integer>`
- `gid` `<integer>`
- Returns: `<Promise>`

Changes the ownership of the file then resolves the `Promise` with no arguments upon success.

filehandle.close()

- Returns: `<Promise>` A `Promise` that will be resolved once the underlying file descriptor is closed, or will be rejected if an error occurs while closing.

Closes the file descriptor.

```
const fsPromises = require('fs').promises;
async function openAndClose() {
  let filehandle;
  try {
    filehandle = await fsPromises.open('thefile.txt', 'r');
```

```
    } finally {
      if (filehandle !== undefined)
        await filehandle.close();
    }
  }
```

filehandle.datasync()

- Returns: <Promise>

Asynchronous fdatasync(2). The Promise is resolved with no arguments upon success.

filehandle.fd

- <number> The numeric file descriptor managed by the FileHandle object.

filehandle.read(buffer, offset, length, position)

- buffer <Buffer> | <Uint8Array>
- offset <integer>
- length <integer>
- position <integer>
- Returns: <Promise>

Read data from the file.

buffer is the buffer that the data will be written to.

offset is the offset in the buffer to start writing at.

length is an integer specifying the number of bytes to read.

position is an argument specifying where to begin reading from in the file. If position is null, data will be read from the current file position, and the file position will be updated. If position is an integer, the file position will remain unchanged.

Following successful read, the Promise is resolved with an object with a bytesRead property specifying the number of bytes read, and a buffer property that is a reference to the passed in buffer argument.

filehandle.readFile(options)

- options <Object> | <string>
  - encoding <string> | <null> **Default:** null
  - flag <string> See support of file system flags. **Default:** 'r'.
- Returns: <Promise>

Asynchronously reads the entire contents of a file.

The Promise is resolved with the contents of the file. If no encoding is specified (using options.encoding), the data is returned as a Buffer object. Otherwise, the data will be a string.

If options is a string, then it specifies the encoding.

When the path is a directory, the behavior of fsPromises.readFile() is platform-specific. On macOS, Linux, and Windows, the promise will be rejected with an error. On FreeBSD, a representation of the directory's contents will be returned.

The FileHandle has to support reading.

If one or more filehandle.read() calls are made on a file handle and then a filehandle.readFile() call is made, the data will be read from the current position till the end of the file. It doesn't always read from the beginning of the file.

filehandle.stat([options])

- `options` `<Object>`
  - `bigint` `<boolean>` Whether the numeric values in the returned `fs.Stats` object should be `bigint`. **Default:** `false`.
- Returns: `<Promise>`

Retrieves the `fs.Stats` for the file.

filehandle.sync()

- Returns: `<Promise>`

Asynchronous `fsync(2)`. The `Promise` is resolved with no arguments upon success.

filehandle.truncate(len)

- `len` `<integer>` **Default:** `0`
- Returns: `<Promise>`

Truncates the file then resolves the `Promise` with no arguments upon success.

If the file was larger than `len` bytes, only the first `len` bytes will be retained in the file.

For example, the following program retains only the first four bytes of the file:

```
const fs = require('fs');
const fsPromises = fs.promises;

console.log(fs.readFileSync('temp.txt', 'utf8'));
// Prints: Node.js

async function doTruncate() {
  let filehandle = null;
  try {
    filehandle = await fsPromises.open('temp.txt', 'r+');
    await filehandle.truncate(4);
  } finally {
    if (filehandle) {
      // close the file if it is opened.
      await filehandle.close();
    }
  }
  console.log(fs.readFileSync('temp.txt', 'utf8'));  // Prints: Node
}

doTruncate().catch(console.error);
```

If the file previously was shorter than `len` bytes, it is extended, and the extended part is filled with null bytes (`'\0'`):

```
const fs = require('fs');
const fsPromises = fs.promises;

console.log(fs.readFileSync('temp.txt', 'utf8'));
// Prints: Node.js

async function doTruncate() {
  let filehandle = null;
  try {
```

```
    filehandle = await fsPromises.open('temp.txt', 'r+');
    await filehandle.truncate(10);
  } finally {
    if (filehandle) {
      // close the file if it is opened.
      await filehandle.close();
    }
  }
  console.log(fs.readFileSync('temp.txt', 'utf8'));  // Prints Node.js\0\0\0
}

doTruncate().catch(console.error);
```

The last three bytes are null bytes (`'\0'`), to compensate the over-truncation.

filehandle.utimes(atime, mtime)

- `atime` `<number>` | `<string>` | `<Date>`
- `mtime` `<number>` | `<string>` | `<Date>`
- Returns: `<Promise>`

Change the file system timestamps of the object referenced by the `FileHandle` then resolves the `Promise` with no arguments upon success.

This function does not work on AIX versions before 7.1, it will resolve the `Promise` with an error using code `UV_ENOSYS` .

filehandle.write(buffer, offset, length, position)

- `buffer` `<Buffer>` | `<Uint8Array>`
- `offset` `<integer>`
- `length` `<integer>`
- `position` `<integer>`
- Returns: `<Promise>`

Write `buffer` to the file.

The `Promise` is resolved with an object containing a `bytesWritten` property identifying the number of bytes written, and a `buffer` property containing a reference to the `buffer` written.

`offset` determines the part of the buffer to be written, and `length` is an integer specifying the number of bytes to write.

`position` refers to the offset from the beginning of the file where this data should be written. If `typeof position !== 'number'` , the data will be written at the current position. See `pwrite(2)` .

It is unsafe to use `filehandle.write()` multiple times on the same file without waiting for the `Promise` to be resolved (or rejected). For this scenario, `fs.createWriteStream()` is strongly recommended.

On Linux, positional writes do not work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

filehandle.write(string[, position[, encoding]])

- `string` `<string>`
- `position` `<integer>`
- `encoding` `<string>` **Default:** `'utf8'`
- Returns: `<Promise>`

Write `string` to the file. If `string` is not a string, then the value will be coerced to one.

The `Promise` is resolved with an object containing a `bytesWritten` property identifying the number of bytes written, and a `buffer` property containing a reference to the `string` written.

`position` refers to the offset from the beginning of the file where this data should be written. If the type of `position` is not a `number` the data will be written at the current position. See `pwrite(2)`.

`encoding` is the expected string encoding.

It is unsafe to use `filehandle.write()` multiple times on the same file without waiting for the `Promise` to be resolved (or rejected). For this scenario, `fs.createWriteStream()` is strongly recommended.

On Linux, positional writes do not work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

filehandle.writeFile(data, options)

- `data` `<string>` | `<Buffer>` | `<Uint8Array>`
- `options` `<Object>` | `<string>`
  - `encoding` `<string>` | `<null>` **Default:** `'utf8'`
  - `mode` `<integer>` **Default:** `0o666`
  - `flag` `<string>` See support of file system `flags`. **Default:** `'w'`.
- Returns: `<Promise>`

Asynchronously writes data to a file, replacing the file if it already exists. `data` can be a string or a buffer. The `Promise` will be resolved with no arguments upon success.

The `encoding` option is ignored if `data` is a buffer.

If `options` is a string, then it specifies the encoding.

The `FileHandle` has to support writing.

It is unsafe to use `filehandle.writeFile()` multiple times on the same file without waiting for the `Promise` to be resolved (or rejected).

If one or more `filehandle.write()` calls are made on a file handle and then a `filehandle.writeFile()` call is made, the data will be written from the current position till the end of the file. It doesn't always write from the beginning of the file.

fsPromises.access(path[, mode])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `mode` `<integer>` **Default:** `fs.constants.F_OK`
- Returns: `<Promise>`

Tests a user's permissions for the file or directory specified by `path`. The `mode` argument is an optional integer that specifies the accessibility checks to be performed. Check File Access Constants for possible values of `mode`. It is possible to create a mask consisting of the bitwise OR of two or more values (e.g. `fs.constants.W_OK | fs.constants.R_OK`).

If the accessibility check is successful, the `Promise` is resolved with no value. If any of the accessibility checks fail, the `Promise` is rejected with an `Error` object. The following example checks if the file `/etc/passwd` can be read and written by the current process.

```
const fs = require('fs');
const fsPromises = fs.promises;

fsPromises.access('/etc/passwd', fs.constants.R_OK | fs.constants.W_OK)
  .then(() => console.log('can access'))
  .catch(() => console.error('cannot access'));
```

Using `fsPromises.access()` to check for the accessibility of a file before calling `fsPromises.open()` is not recommended. Doing so introduces a race condition, since other processes may change the file's state between the two calls. Instead, user code should open/read/write the file directly and handle the error raised if the file is not accessible.

fsPromises.appendFile(path, data[, options])

- `path` `<string>` | `<Buffer>` | `<URL>` | `<FileHandle>` filename or `FileHandle`
- `data` `<string>` | `<Buffer>`
- `options` `<Object>` | `<string>`
  - `encoding` `<string>` | `<null>` **Default:** `'utf8'`
  - `mode` `<integer>` **Default:** `0o666`
  - `flag` `<string>` See support of file system `flags`. **Default:** `'a'`.
- Returns: `<Promise>`

Asynchronously append data to a file, creating the file if it does not yet exist. `data` can be a string or a `Buffer`. The `Promise` will be resolved with no arguments upon success.

If `options` is a string, then it specifies the encoding.

The `path` may be specified as a `FileHandle` that has been opened for appending (using `fsPromises.open()`).

fsPromises.chmod(path, mode)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `mode` `<integer>`
- Returns: `<Promise>`

Changes the permissions of a file then resolves the `Promise` with no arguments upon succces.

fsPromises.chown(path, uid, gid)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `uid` `<integer>`
- `gid` `<integer>`
- Returns: `<Promise>`

Changes the ownership of a file then resolves the `Promise` with no arguments upon success.

fsPromises.copyFile(src, dest[, flags])

- `src` `<string>` | `<Buffer>` | `<URL>` source filename to copy
- `dest` `<string>` | `<Buffer>` | `<URL>` destination filename of the copy operation
- `flags` `<number>` modifiers for copy operation. **Default:** `0`.
- Returns: `<Promise>`

Asynchronously copies `src` to `dest`. By default, `dest` is overwritten if it already exists. The `Promise` will be resolved with no arguments upon success.

Node.js makes no guarantees about the atomicity of the copy operation. If an error occurs after the destination file has been opened for writing, Node.js will attempt to remove the destination.

`flags` is an optional integer that specifies the behavior of the copy operation. It is possible to create a mask consisting of the bitwise OR of two or more values (e.g. `fs.constants.COPYFILE_EXCL | fs.constants.COPYFILE_FICLONE`).

- `fs.constants.COPYFILE_EXCL` - The copy operation will fail if `dest` already exists.

- `fs.constants.COPYFILE_FICLONE` - The copy operation will attempt to create a copy-on-write reflink. If the platform does not support copy-on-write, then a fallback copy mechanism is used.
- `fs.constants.COPYFILE_FICLONE_FORCE` - The copy operation will attempt to create a copy-on-write reflink. If the platform does not support copy-on-write, then the operation will fail.

```
const fsPromises = require('fs').promises;

// destination.txt will be created or overwritten by default.
fsPromises.copyFile('source.txt', 'destination.txt')
  .then(() => console.log('source.txt was copied to destination.txt'))
  .catch(() => console.log('The file could not be copied'));
```

If the third argument is a number, then it specifies `flags`:

```
const fs = require('fs');
const fsPromises = fs.promises;
const { COPYFILE_EXCL } = fs.constants;

// By using COPYFILE_EXCL, the operation will fail if destination.txt exists.
fsPromises.copyFile('source.txt', 'destination.txt', COPYFILE_EXCL)
  .then(() => console.log('source.txt was copied to destination.txt'))
  .catch(() => console.log('The file could not be copied'));
```

fsPromises.lchmod(path, mode)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `mode` `<integer>`
- Returns: `<Promise>`

Changes the permissions on a symbolic link then resolves the `Promise` with no arguments upon success. This method is only implemented on macOS.

fsPromises.lchown(path, uid, gid)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `uid` `<integer>`
- `gid` `<integer>`
- Returns: `<Promise>`

Changes the ownership on a symbolic link then resolves the `Promise` with no arguments upon success.

fsPromises.link(existingPath, newPath)

- `existingPath` `<string>` | `<Buffer>` | `<URL>`
- `newPath` `<string>` | `<Buffer>` | `<URL>`
- Returns: `<Promise>`

Asynchronous `link(2)`. The `Promise` is resolved with no arguments upon success.

fsPromises.lstat(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
  - `bigint` `<boolean>` Whether the numeric values in the returned `fs.Stats` object should be `bigint`. **Default:** `false`.

- Returns: `<Promise>`

Asynchronous `lstat(2)`. The `Promise` is resolved with the `fs.Stats` object for the given symbolic link `path`.

fsPromises.mkdir(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>` | `<integer>`
  - `recursive` `<boolean>` **Default:** `false`
  - `mode` `<integer>` Not supported on Windows. **Default:** `0o777`.
- Returns: `<Promise>`

Asynchronously creates a directory then resolves the `Promise` with no arguments upon success.

The optional `options` argument can be an integer specifying mode (permission and sticky bits), or an object with a `mode` property and a `recursive` property indicating whether parent folders should be created.

fsPromises.mkdtemp(prefix[, options])

- `prefix` `<string>`
- `options` `<string>` | `<Object>`
  - `encoding` `<string>` **Default:** `'utf8'`
- Returns: `<Promise>`

Creates a unique temporary directory and resolves the `Promise` with the created folder path. A unique directory name is generated by appending six random characters to the end of the provided `prefix`.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use.

```
fsPromises.mkdtemp(path.join(os.tmpdir(), 'foo-'))
  .catch(console.error);
```

The `fsPromises.mkdtemp()` method will append the six randomly selected characters directly to the `prefix` string. For instance, given a directory `/tmp`, if the intention is to create a temporary directory *within* `/tmp`, the `prefix` must end with a trailing platform-specific path separator (`require('path').sep`).

fsPromises.open(path, flags[, mode])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `flags` `<string>` | `<number>` See support of file system `flags`. **Default:** `'r'`.
- `mode` `<integer>` **Default:** `0o666` (readable and writable)
- Returns: `<Promise>`

Asynchronous file open that returns a `Promise` that, when resolved, yields a `FileHandle` object. See `open(2)`.

`mode` sets the file mode (permission and sticky bits), but only if the file was created.

Some characters (`< > : " / \ | ? *`) are reserved under Windows as documented by Naming Files, Paths, and Namespaces. Under NTFS, if the filename contains a colon, Node.js will open a file system stream, as described by this MSDN page.

fsPromises.readdir(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`

    - `encoding` `<string>` **Default:** `'utf8'`
    - `withFileTypes` `<boolean>` **Default:** `false`
- Returns: `<Promise>`

Reads the contents of a directory then resolves the `Promise` with an array of the names of the files in the directory excluding `'.'` and `'..'`.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the filenames. If the `encoding` is set to `'buffer'`, the filenames returned will be passed as `Buffer` objects.

If `options.withFileTypes` is set to `true`, the resolved array will contain `fs.Dirent` objects.

fsPromises.readFile(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>` | `<FileHandle>` filename or `FileHandle`
- `options` `<Object>` | `<string>`
    - `encoding` `<string>` | `<null>` **Default:** `null`
    - `flag` `<string>` See support of file system `flags`. **Default:** `'r'`.
- Returns: `<Promise>`

Asynchronously reads the entire contents of a file.

The `Promise` is resolved with the contents of the file. If no encoding is specified (using `options.encoding`), the data is returned as a `Buffer` object. Otherwise, the data will be a string.

If `options` is a string, then it specifies the encoding.

When the `path` is a directory, the behavior of `fsPromises.readFile()` is platform-specific. On macOS, Linux, and Windows, the promise will be rejected with an error. On FreeBSD, a representation of the directory's contents will be returned.

Any specified `FileHandle` has to support reading.

fsPromises.readlink(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
    - `encoding` `<string>` **Default:** `'utf8'`
- Returns: `<Promise>`

Asynchronous `readlink(2)`. The `Promise` is resolved with the `linkString` upon success.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the link path returned. If the `encoding` is set to `'buffer'`, the link path returned will be passed as a `Buffer` object.

fsPromises.realpath(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
    - `encoding` `<string>` **Default:** `'utf8'`
- Returns: `<Promise>`

Determines the actual location of `path` using the same semantics as the `fs.realpath.native()` function then resolves the `Promise` with the resolved path.

Only paths that can be converted to UTF8 strings are supported.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the path. If the `encoding` is set to `'buffer'`, the path returned will be passed as a `Buffer` object.

On Linux, when Node.js is linked against musl libc, the procfs file system must be mounted on `/proc` in order for this function to work. Glibc does not have this restriction.

fsPromises.rename(oldPath, newPath)

- `oldPath` `<string>` | `<Buffer>` | `<URL>`
- `newPath` `<string>` | `<Buffer>` | `<URL>`
- Returns: `<Promise>`

Renames `oldPath` to `newPath` and resolves the `Promise` with no arguments upon success.

fsPromises.rmdir(path)

- `path` `<string>` | `<Buffer>` | `<URL>`
- Returns: `<Promise>`

Removes the directory identified by `path` then resolves the `Promise` with no arguments upon success.

Using `fsPromises.rmdir()` on a file (not a directory) results in the `Promise` being rejected with an `ENOENT` error on Windows and an `ENOTDIR` error on POSIX.

fsPromises.stat(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
  - `bigint` `<boolean>` Whether the numeric values in the returned `fs.Stats` object should be `bigint`. **Default:** `false`.
- Returns: `<Promise>`

The `Promise` is resolved with the `fs.Stats` object for the given `path`.

fsPromises.symlink(target, path[, type])

- `target` `<string>` | `<Buffer>` | `<URL>`
- `path` `<string>` | `<Buffer>` | `<URL>`
- `type` `<string>` **Default:** `'file'`
- Returns: `<Promise>`

Creates a symbolic link then resolves the `Promise` with no arguments upon success.

The `type` argument is only used on Windows platforms and can be one of `'dir'`, `'file'`, or `'junction'`. Windows junction points require the destination path to be absolute. When using `'junction'`, the `target` argument will automatically be normalized to absolute path.

fsPromises.truncate(path[, len])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `len` `<integer>` **Default:** `0`
- Returns: `<Promise>`

Truncates the `path` then resolves the `Promise` with no arguments upon success. The `path` *must* be a string or `Buffer`.

fsPromises.unlink(path)

- `path` `<string>` | `<Buffer>` | `<URL>`

- Returns: `<Promise>`

Asynchronous `unlink(2)`. The `Promise` is resolved with no arguments upon success.

fsPromises.utimes(path, atime, mtime)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `atime` `<number>` | `<string>` | `<Date>`
- `mtime` `<number>` | `<string>` | `<Date>`
- Returns: `<Promise>`

Change the file system timestamps of the object referenced by `path` then resolves the `Promise` with no arguments upon success.

The `atime` and `mtime` arguments follow these rules:

- Values can be either numbers representing Unix epoch time, `Date`s, or a numeric string like `'123456789.0'`.
- If the value can not be converted to a number, or is `NaN`, `Infinity` or `-Infinity`, an `Error` will be thrown.

fsPromises.writeFile(file, data[, options])

- `file` `<string>` | `<Buffer>` | `<URL>` | `<FileHandle>` filename or `FileHandle`
- `data` `<string>` | `<Buffer>` | `<Uint8Array>`
- `options` `<Object>` | `<string>`
  - `encoding` `<string>` | `<null>` **Default:** `'utf8'`
  - `mode` `<integer>` **Default:** `0o666`
  - `flag` `<string>` See support of file system `flags`. **Default:** `'w'`.
- Returns: `<Promise>`

Asynchronously writes data to a file, replacing the file if it already exists. `data` can be a string or a buffer. The `Promise` will be resolved with no arguments upon success.

The `encoding` option is ignored if `data` is a buffer.

If `options` is a string, then it specifies the encoding.

Any specified `FileHandle` has to support writing.

It is unsafe to use `fsPromises.writeFile()` multiple times on the same file without waiting for the `Promise` to be resolved (or rejected).

FS Constants

The following constants are exported by `fs.constants`.

Not every constant will be available on every operating system.

File Access Constants

The following constants are meant for use with `fs.access()`.

Constant	Description
`F_OK`	Flag indicating that the file is visible to the calling process. This is useful for determining if a file exists, but says nothing about `rwx` permissions. Default if no mode is specified.
`R_OK`	Flag indicating that the file can be read by the calling process.
`W_OK`	Flag indicating that the file can be written by the calling process.

`X_OK`	Flag indicating that the file can be executed by the calling process. This has no effect on Windows (will behave like `fs.constants.F_OK`).

File Copy Constants

The following constants are meant for use with `fs.copyFile()`.

Constant	Description
`COPYFILE_EXCL`	If present, the copy operation will fail with an error if the destination path already exists.
`COPYFILE_FICLONE`	If present, the copy operation will attempt to create a copy-on-write reflink. If the underlying platform does not support copy-on-write, then a fallback copy mechanism is used.
`COPYFILE_FICLONE_FORCE`	If present, the copy operation will attempt to create a copy-on-write reflink. If the underlying platform does not support copy-on-write, then the operation will fail with an error.

File Open Constants

The following constants are meant for use with `fs.open()`.

Constant	Description
`O_RDONLY`	Flag indicating to open a file for read-only access.
`O_WRONLY`	Flag indicating to open a file for write-only access.
`O_RDWR`	Flag indicating to open a file for read-write access.
`O_CREAT`	Flag indicating to create the file if it does not already exist.
`O_EXCL`	Flag indicating that opening a file should fail if the `O_CREAT` flag is set and the file already exists.
`O_NOCTTY`	Flag indicating that if path identifies a terminal device, opening the path shall not cause that terminal to become the controlling terminal for the process (if the process does not already have one).
`O_TRUNC`	Flag indicating that if the file exists and is a regular file, and the file is opened successfully for write access, its length shall be truncated to zero.
`O_APPEND`	Flag indicating that data will be appended to the end of the file.
`O_DIRECTORY`	Flag indicating that the open should fail if the path is not a directory.
`O_NOATIME`	Flag indicating reading accesses to the file system will no longer result in an update to the `atime` information associated with the file. This flag is available on Linux operating systems only.
`O_NOFOLLOW`	Flag indicating that the open should fail if the path is a symbolic link.
`O_SYNC`	Flag indicating that the file is opened for synchronized I/O with write operations waiting for file integrity.
`O_DSYNC`	Flag indicating that the file is opened for synchronized I/O with write operations waiting for data integrity.
`O_SYMLINK`	Flag indicating to open the symbolic link itself rather than the resource it is pointing to.
`O_DIRECT`	When set, an attempt will be made to minimize caching effects of file I/O.

`O_NONBLOCK`	Flag indicating to open the file in nonblocking mode when possible.

File Type Constants

The following constants are meant for use with the `fs.Stats` object's `mode` property for determining a file's type.

Constant	Description
`S_IFMT`	Bit mask used to extract the file type code.
`S_IFREG`	File type constant for a regular file.
`S_IFDIR`	File type constant for a directory.
`S_IFCHR`	File type constant for a character-oriented device file.
`S_IFBLK`	File type constant for a block-oriented device file.
`S_IFIFO`	File type constant for a FIFO/pipe.
`S_IFLNK`	File type constant for a symbolic link.
`S_IFSOCK`	File type constant for a socket.

File Mode Constants

The following constants are meant for use with the `fs.Stats` object's `mode` property for determining the access permissions for a file.

Constant	Description
`S_IRWXU`	File mode indicating readable, writable, and executable by owner.
`S_IRUSR`	File mode indicating readable by owner.
`S_IWUSR`	File mode indicating writable by owner.
`S_IXUSR`	File mode indicating executable by owner.
`S_IRWXG`	File mode indicating readable, writable, and executable by group.
`S_IRGRP`	File mode indicating readable by group.
`S_IWGRP`	File mode indicating writable by group.
`S_IXGRP`	File mode indicating executable by group.
`S_IRWXO`	File mode indicating readable, writable, and executable by others.
`S_IROTH`	File mode indicating readable by others.
`S_IWOTH`	File mode indicating writable by others.
`S_IXOTH`	File mode indicating executable by others.

File System Flags

The following flags are available wherever the `flag` option takes a string:

- `'a'` - Open file for appending. The file is created if it does not exist.
- `'ax'` - Like `'a'` but fails if the path exists.
- `'a+'` - Open file for reading and appending. The file is created if it does not exist.
- `'ax+'` - Like `'a+'` but fails if the path exists.
- `'as'` - Open file for appending in synchronous mode. The file is created if it does not exist.
- `'as+'` - Open file for reading and appending in synchronous mode. The file is created if it does not exist.
- `'r'` - Open file for reading. An exception occurs if the file does not exist.
- `'r+'` - Open file for reading and writing. An exception occurs if the file does not exist.
- `'rs+'` - Open file for reading and writing in synchronous mode. Instructs the operating system to bypass the local file system cache.

  This is primarily useful for opening files on NFS mounts as it allows skipping the potentially stale local cache. It has a very real impact on I/O performance so using this flag is not recommended unless it is needed.

  This doesn't turn `fs.open()` or `fsPromises.open()` into a synchronous blocking call. If synchronous operation is desired, something like `fs.openSync()` should be used.
- `'w'` - Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx'` - Like `'w'` but fails if the path exists.
- `'w+'` - Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx+'` - Like `'w+'` but fails if the path exists.

`flag` can also be a number as documented by `open(2)`; commonly used constants are available from `fs.constants`. On Windows, flags are translated to their equivalent ones where applicable, e.g. `O_WRONLY` to `FILE_GENERIC_WRITE`, or `O_EXCL|O_CREAT` to `CREATE_NEW`, as accepted by `CreateFileW`.

The exclusive flag `'x'` (`O_EXCL` flag in `open(2)`) ensures that path is newly created. On POSIX systems, path is considered to exist even if it is a symlink to a non-existent file. The exclusive flag may or may not work with network file systems.

On Linux, positional writes don't work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

Modifying a file rather than replacing it may require a flags mode of `'r+'` rather than the default mode `'w'`.

The behavior of some flags are platform-specific. As such, opening a directory on macOS and Linux with the `'a+'` flag - see example below - will return an error. In contrast, on Windows and FreeBSD, a file descriptor or a `FileHandle` will be returned.

```
// macOS and Linux
fs.open('<directory>', 'a+', (err, fd) => {
  // => [Error: EISDIR: illegal operation on a directory, open <directory>]
});

// Windows and FreeBSD
fs.open('<directory>', 'a+', (err, fd) => {
  // => null, <fd>
});
```

On Windows, opening an existing hidden file using the `'w'` flag (either through `fs.open()` or `fs.writeFile()` or `fsPromises.open()`) will fail with `EPERM`. Existing hidden files can be opened for writing with the `'r+'` flag.

A call to `fs.ftruncate()` or `filehandle.truncate()` can be used to reset the file contents.