File system

> *Stability: 2 - Stable*

The `fs` module enables interacting with the file system in a way modeled on standard POSIX functions.

To use the promise-based APIs:

```mjs
import * as fs from 'fs/promises';
```

```cjs
const fs = require('fs/promises');
```

To use the callback and sync APIs:

```mjs
import * as fs from 'fs';
```

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

All file system operations have synchronous, callback, and promise-based forms, and are accessible using both CommonJS syntax and ES6 Modules (ESM).

Promise example

Promise-based operations return a promise that is fulfilled when the asynchronous operation is complete.

```mjs
import { unlink } from 'fs/promises';

try {
  await unlink('/tmp/hello');
  console.log('successfully deleted /tmp/hello');
} catch (error) {
  console.error('there was an error:', error.message);
}
```

```cjs
const { unlink } = require('fs/promises');

(async function(path) {
  try {
    await unlink(path);
    console.log(`successfully deleted ${path}`);
  } catch (error) {
    console.error('there was an error:', error.message);
  }
})('/tmp/hello');
```

Callback example

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

```mjs
import { unlink } from 'fs';

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

```cjs
const { unlink } = require('fs');

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

The callback-based versions of the `fs` module APIs are preferable over the use of the promise APIs when maximal performance (both in terms of execution time and memory allocation) is required.

Synchronous example

The synchronous APIs block the Node.js event loop and further JavaScript execution until the operation is complete. Exceptions are thrown immediately and can be handled using `try…catch`, or can be allowed to bubble up.

```mjs
import { unlinkSync } from 'fs';

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

```cjs
const { unlinkSync } = require('fs');

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

Promises API

The `fs/promises` API provides asynchronous file system methods that return promises.

The promise APIs use the underlying Node.js threadpool to perform file system operations off the event loop thread. These operations are not synchronized or threadsafe. Care must be taken when performing multiple concurrent modifications on the same file or data corruption may occur.

Class: `FileHandle`

A {FileHandle} object is an object wrapper for a numeric file descriptor.

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

All {FileHandle} objects are {EventEmitter}s.

If a {FileHandle} is not closed using the `filehandle.close()` method, it will try to automatically close the file descriptor and emit a process warning, helping to prevent memory leaks. Please do not rely on this behavior because it can be unreliable and the file may not be closed. Instead, always explicitly close {FileHandle}s. Node.js may change this behavior in the future.

Event: `'close'`

The `'close'` event is emitted when the {FileHandle} has been closed and can no longer be used.

`filehandle.appendFile(data[, options])`

- `data` {string|Buffer|TypedArray|DataView|AsyncIterable|Iterable|Stream}
- `options` {Object|string}
  - `encoding` {string|null} **Default:** `'utf8'`
- Returns: {Promise} Fulfills with `undefined` upon success.

Alias of `filehandle.writeFile()`.

When operating on file handles, the mode cannot be changed from what it was set to with `fsPromises.open()`. Therefore, this is equivalent to `filehandle.writeFile()`.

`filehandle.chmod(mode)`

- `mode` {integer} the file mode bit mask.
- Returns: {Promise} Fulfills with `undefined` upon success.

Modifies the permissions on the file. See chmod(2).

`filehandle.chown(uid, gid)`

- `uid` {integer} The file's new owner's user id.
- `gid` {integer} The file's new group's group id.
- Returns: {Promise} Fulfills with `undefined` upon success.

Changes the ownership of the file. A wrapper for chown(2).

`filehandle.close()`

- Returns: {Promise} Fulfills with `undefined` upon success.

Closes the file handle after waiting for any pending operation on the handle to complete.

```
import { open } from 'fs/promises';
```

```
let filehandle;
try {
  filehandle = await open('thefile.txt', 'r');
} finally {
  await filehandle?.close();
}
```

`filehandle.createReadStream([options])`

- `options` {Object}
  - `encoding` {string} **Default:** `null`
  - `autoClose` {boolean} **Default:** `true`
  - `emitClose` {boolean} **Default:** `true`
  - `start` {integer}
  - `end` {integer} **Default:** `Infinity`
  - `highWaterMark` {integer} **Default:** `64 * 1024`
- Returns: {fs.ReadStream}

Unlike the 16 kb default `highWaterMark` for a {stream.Readable}, 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, allowed values are in the [0, `Number.MAX_SAFE_INTEGER`] range. If `start` is omitted or `undefined`, `filehandle.createReadStream()` reads sequentially from the current file position. The `encoding` can be any one of those accepted by {Buffer}.

If the `FileHandle` 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.

By default, the stream will emit a `'close'` event after it has been destroyed. Set the `emitClose` option to `false` to change this behavior.

```
import { open } from 'fs/promises';

const fd = await open('/dev/input/event0');
// Create a stream from some character device.
const stream = fd.createReadStream();
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.

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

```
import { open } from 'fs/promises';

const fd = await open('sample.txt');
fd.createReadStream({ start: 90, end: 99 });
```

`filehandle.createWriteStream([options])`

- `options` {Object}
  - `encoding` {string} **Default:** `'utf8'`
  - `autoClose` {boolean} **Default:** `true`
  - `emitClose` {boolean} **Default:** `true`
  - `start` {integer}
- Returns: {fs.WriteStream}

`options` may also include a `start` option to allow writing data at some position past the beginning of the file, allowed values are in the [0, `Number.MAX_SAFE_INTEGER`] range. Modifying a file rather than replacing it may require the `flags` `open` option to be set to `r+` rather than the default `r` . 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.

By default, the stream will emit a `'close'` event after it has been destroyed. Set the `emitClose` option to `false` to change this behavior.

`filehandle.datasync()`

- Returns: {Promise} Fulfills with `undefined` upon success.

Forces all currently queued I/O operations associated with the file to the operating system's synchronized I/O completion state. Refer to the POSIX fdatasync(2) documentation for details.

Unlike `filehandle.sync` this method does not flush modified metadata.

`filehandle.fd`

- {number} The numeric file descriptor managed by the {FileHandle} object.

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

- `buffer` {Buffer|TypedArray|DataView} A buffer that will be filled with the file data read.
- `offset` {integer} The location in the buffer at which to start filling.
- `length` {integer} The number of bytes to read.
- `position` {integer} The location where to begin reading data from the file. If `null` , data will be read from the current file position, and the position will be updated. If `position` is an integer, the current file position will remain unchanged.
- Returns: {Promise} Fulfills upon success with an object with two properties:

    - `bytesRead` {integer} The number of bytes read
    - `buffer` {Buffer|TypedArray|DataView} A reference to the passed in `buffer` argument.

Reads data from the file and stores that in the given buffer.

If the file is not modified concurrently, the end-of-file is reached when the number of bytes read is zero.

`filehandle.read([options])`

- `options` {Object}
    - `buffer` {Buffer|TypedArray|DataView} A buffer that will be filled with the file data read. **Default:** `Buffer.alloc(16384)`
    - `offset` {integer} The location in the buffer at which to start filling. **Default:** `0`
    - `length` {integer} The number of bytes to read. **Default:** `buffer.byteLength - offset`
    - `position` {integer} The location where to begin reading data from the file. If `null`, data will be read from the current file position, and the position will be updated. If `position` is an integer, the current file position will remain unchanged. **Default:**: `null`
- Returns: {Promise} Fulfills upon success with an object with two properties:
    - `bytesRead` {integer} The number of bytes read
    - `buffer` {Buffer|TypedArray|DataView} A reference to the passed in `buffer` argument.

Reads data from the file and stores that in the given buffer.

If the file is not modified concurrently, the end-of-file is reached when the number of bytes read is zero.

`filehandle.readableWebStream()`

> *Stability: 1 - Experimental*

- Returns: {ReadableStream}

Returns a `ReadableStream` that may be used to read the files data.

An error will be thrown if this method is called more than once or is called after the `FileHandle` is closed or closing.

```mjs
import {
  open,
} from 'node:fs/promises';

const file = await open('./some/file/to/read');

for await (const chunk of file.readableWebStream())
  console.log(chunk);

await file.close();
```

```cjs
const {
  open,
} = require('fs/promises');

(async () => {
```

```
  const file = await open('./some/file/to/read');

  for await (const chunk of file.readableWebStream())
    console.log(chunk);

  await file.close();
})();
```

While the `ReadableStream` will read the file to completion, it will not close the `FileHandle` automatically. User code must still call the `fileHandle.close()` method.

`filehandle.readFile(options)`

- `options` {Object|string}
  - `encoding` {string|null} **Default:** `null`
  - `signal` {AbortSignal} allows aborting an in-progress readFile
- Returns: {Promise} Fulfills upon a successful read 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.

Asynchronously reads the entire contents of a file.

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

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.readv(buffers[, position])`

- `buffers` {Buffer[]|TypedArray[]|DataView[]}
- `position` {integer} The offset from the beginning of the file where the data should be read from. If `position` is not a `number`, the data will be read from the current position.
- Returns: {Promise} Fulfills upon success an object containing two properties:
  - `bytesRead` {integer} the number of bytes read
  - `buffers` {Buffer[]|TypedArray[]|DataView[]} property containing a reference to the `buffers` input.

Read from a file and write to an array of {ArrayBufferView}s

`filehandle.stat([options])`

- `options` {Object}
  - `bigint` {boolean} Whether the numeric values in the returned {fs.Stats} object should be `bigint`. **Default:** `false`.
- Returns: {Promise} Fulfills with an {fs.Stats} for the file.

`filehandle.sync()`

- Returns: {Promise} Fufills with `undefined` upon success.

Request that all data for the open file descriptor is flushed to the storage device. The specific implementation is operating system and device specific. Refer to the POSIX fsync(2) documentation for more detail.

`filehandle.truncate(len)`

- `len` {integer} **Default:** `0`
- Returns: {Promise} Fulfills with `undefined` upon success.

Truncates the file.

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

The following example retains only the first four bytes of the file:

```
import { open } from 'fs/promises';

let filehandle = null;
try {
  filehandle = await open('temp.txt', 'r+');
  await filehandle.truncate(4);
} finally {
  await filehandle?.close();
}
```

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

If `len` is negative then `0` will be used.

`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.

`filehandle.write(buffer[, offset[, length[, position]]])`

- `buffer` {Buffer|TypedArray|DataView}
- `offset` {integer} The start position from within `buffer` where the data to write begins. **Default:** `0`
- `length` {integer} The number of bytes from `buffer` to write. **Default:** `buffer.byteLength - offset`
- `position` {integer} The offset from the beginning of the file where the data from `buffer` should be written. If `position` is not a `number`, the data will be written at the current position. See the POSIX pwrite(2) documentation for more detail.
- Returns: {Promise}

Write `buffer` to the file.

The promise is resolved with an object containing two properties:

- `bytesWritten` {integer} the number of bytes written
- `buffer` {Buffer|TypedArray|DataView} a reference to the `buffer` written.

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, use `fs.createWriteStream()`.

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} The offset from the beginning of the file where the data from `string` should be written. If `position` is not a `number` the data will be written at the current position. See the POSIX pwrite(2) documentation for more detail.
- `encoding` {string} The expected string encoding. **Default:** `'utf8'`
- Returns: {Promise}

Write `string` to the file. If `string` is not a string, the promise is rejected with an error.

The promise is resolved with an object containing two properties:

- `bytesWritten` {integer} the number of bytes written
- `buffer` {string} a reference to the `string` written.

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, use `fs.createWriteStream()`.

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|TypedArray|DataView|AsyncIterable|Iterable|Stream}
- `options` {Object|string}
  - `encoding` {string|null} The expected character encoding when `data` is a string. **Default:** `'utf8'`
- Returns: {Promise}

Asynchronously writes data to a file, replacing the file if it already exists. `data` can be a string, a buffer, an {AsyncIterable} or {Iterable} object. The promise is resolved with no arguments upon success.

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.

`filehandle.writev(buffers[, position])`

- `buffers` {Buffer[]|TypedArray[]|DataView[]}
- `position` {integer} The offset from the beginning of the file where the data from `buffers` should be written. If `position` is not a `number`, the data will be written at the current position.
- Returns: {Promise}

Write an array of {ArrayBufferView}s to the file.

The promise is resolved with an object containing a two properties:

- `bytesWritten` {integer} the number of bytes written
- `buffers` {Buffer[]|TypedArray[]|DataView[]} a reference to the `buffers` input.

It is unsafe to call `writev()` multiple times on the same file without waiting for the promise to be resolved (or rejected).

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.

`fsPromises.access(path[, mode])`

- `path` {string|Buffer|URL}
- `mode` {integer} **Default:** `fs.constants.F_OK`
- Returns: {Promise} Fulfills with `undefined` upon success.

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. `mode` should be either the value `fs.constants.F_OK` or a mask consisting of the bitwise OR of any of `fs.constants.R_OK`, `fs.constants.W_OK`, and `fs.constants.X_OK` (e.g. `fs.constants.W_OK | fs.constants.R_OK`). Check File access constants for possible values of `mode`.

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.

```
import { access } from 'fs/promises';
import { constants } from 'fs';

try {
  await access('/etc/passwd', constants.R_OK | constants.W_OK);
  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} Fulfills with `undefined` upon success.

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

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

The `mode` option only affects the newly created file. See `fs.open()` for more details.

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` {string|integer}
- Returns: {Promise} Fulfills with `undefined` upon success.

Changes the permissions of a file.

`fsPromises.chown(path, uid, gid)`

- `path` {string|Buffer|URL}
- `uid` {integer}
- `gid` {integer}
- Returns: {Promise} Fulfills with `undefined` upon success.

Changes the ownership of a file.

`fsPromises.copyFile(src, dest[, mode])`

- `src` {string|Buffer|URL} source filename to copy
- `dest` {string|Buffer|URL} destination filename of the copy operation
- `mode` {integer} Optional modifiers that specify 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`) **Default:** `0`.
  - `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.
- Returns: {Promise} Fulfills with `undefined` upon success.

Asynchronously copies `src` to `dest`. By default, `dest` is overwritten if it already exists.

No guarantees are made about the atomicity of the copy operation. If an error occurs after the destination file has been opened for writing, an attempt will be made to remove the destination.

```
import { constants } from 'fs';
import { copyFile } from 'fs/promises';

try {
  await copyFile('source.txt', 'destination.txt');
  console.log('source.txt was copied to destination.txt');
} catch {
  console.log('The file could not be copied');
```

```
}

// By using COPYFILE_EXCL, the operation will fail if destination.txt exists.
try {
  await copyFile('source.txt', 'destination.txt', constants.COPYFILE_EXCL);
  console.log('source.txt was copied to destination.txt');
} catch {
  console.log('The file could not be copied');
}
```

`fsPromises.cp(src, dest[, options])`

> *Stability: 1 - Experimental*

- `src` {string|URL} source path to copy.
- `dest` {string|URL} destination path to copy to.
- `options` {Object}
  - `dereference` {boolean} dereference symlinks. **Default:** `false`.
  - `errorOnExist` {boolean} when `force` is `false`, and the destination exists, throw an error. **Default:** `false`.
  - `filter` {Function} Function to filter copied files/directories. Return `true` to copy the item, `false` to ignore it. Can also return a `Promise` that resolves to `true` or `false` **Default:** `undefined`.
  - `force` {boolean} overwrite existing file or directory. The copy operation will ignore errors if you set this to false and the destination exists. Use the `errorOnExist` option to change this behavior. **Default:** `true`.
  - `preserveTimestamps` {boolean} When `true` timestamps from `src` will be preserved. **Default:** `false`.
  - `recursive` {boolean} copy directories recursively **Default:** `false`
  - `verbatimSymlinks` {boolean} When `true`, path resolution for symlinks will be skipped. **Default:** `false`
- Returns: {Promise} Fulfills with `undefined` upon success.

Asynchronously copies the entire directory structure from `src` to `dest`, including subdirectories and files.

When copying a directory to another directory, globs are not supported and behavior is similar to `cp dir1/ dir2/`.

`fsPromises.lchmod(path, mode)`

- `path` {string|Buffer|URL}
- `mode` {integer}
- Returns: {Promise} Fulfills with `undefined` upon success.

Changes the permissions on a symbolic link.

This method is only implemented on macOS.

`fsPromises.lchown(path, uid, gid)`

- `path` {string|Buffer|URL}

- `uid` {integer}
- `gid` {integer}
- Returns: {Promise} Fulfills with `undefined` upon success.

Changes the ownership on a symbolic link.

`fsPromises.lutimes(path, atime, mtime)`

- `path` {string|Buffer|URL}
- `atime` {number|string|Date}
- `mtime` {number|string|Date}
- Returns: {Promise} Fulfills with `undefined` upon success.

Changes the access and modification times of a file in the same way as `fsPromises.utimes()`, with the difference that if the path refers to a symbolic link, then the link is not dereferenced: instead, the timestamps of the symbolic link itself are changed.

`fsPromises.link(existingPath, newPath)`

- `existingPath` {string|Buffer|URL}
- `newPath` {string|Buffer|URL}
- Returns: {Promise} Fulfills with `undefined` upon success.

Creates a new link from the `existingPath` to the `newPath`. See the POSIX link(2) documentation for more detail.

`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} Fulfills with the {fs.Stats} object for the given symbolic link `path`.

Equivalent to `fsPromises.stat()` unless `path` refers to a symbolic link, in which case the link itself is stat-ed, not the file that it refers to. Refer to the POSIX lstat(2) document for more detail.

`fsPromises.mkdir(path[, options])`

- `path` {string|Buffer|URL}
- `options` {Object|integer}
  - `recursive` {boolean} **Default:** `false`
  - `mode` {string|integer} Not supported on Windows. **Default:** `0o777`.
- Returns: {Promise} Upon success, fulfills with `undefined` if `recursive` is `false`, or the first directory path created if `recursive` is `true`.

Asynchronously creates a directory.

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 directories should be created. Calling `fsPromises.mkdir()` when `path` is a directory that exists results in a rejection only when `recursive` is false.

`fsPromises.mkdtemp(prefix[, options])`

- `prefix` {string}
- `options` {string|Object}
  - `encoding` {string} **Default:** `'utf8'`
- Returns: {Promise} Fulfills with a string containing the filesystem path of the newly created temporary directory.

Creates a unique temporary directory. A unique directory name is generated by appending six random characters to the end of the provided `prefix`. Due to platform inconsistencies, avoid trailing `X` characters in `prefix`. Some platforms, notably the BSDs, can return more than six random characters, and replace trailing `X` characters in `prefix` with random characters.

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

```
import { mkdtemp } from 'fs/promises';

try {
  await mkdtemp(path.join(os.tmpdir(), 'foo-'));
} catch (err) {
  console.error(err);
}
```

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` {string|integer} Sets the file mode (permission and sticky bits) if the file is created. **Default:** `0o666` (readable and writable)
- Returns: {Promise} Fulfills with a {FileHandle} object.

Opens a {FileHandle}.

Refer to the POSIX open(2) documentation for more detail.

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.opendir(path[, options])`

- `path` {string|Buffer|URL}
- `options` {Object}
  - `encoding` {string|null} **Default:** `'utf8'`
  - `bufferSize` {number} Number of directory entries that are buffered internally when reading from the directory. Higher values lead to better performance but higher memory usage. **Default:** `32`

- Returns: {Promise} Fulfills with an {fs.Dir}.

Asynchronously open a directory for iterative scanning. See the POSIX opendir(3) documentation for more detail.

Creates an {fs.Dir}, which contains all further functions for reading from and cleaning up the directory.

The `encoding` option sets the encoding for the `path` while opening the directory and subsequent read operations.

Example using async iteration:

```
import { opendir } from 'fs/promises';

try {
  const dir = await opendir('./');
  for await (const dirent of dir)
    console.log(dirent.name);
} catch (err) {
  console.error(err);
}
```

When using the async iterator, the {fs.Dir} object will be automatically closed after the iterator exits.

`fsPromises.readdir(path[, options])`

- `path` {string|Buffer|URL}
- `options` {string|Object}
  - `encoding` {string} **Default:** `'utf8'`
  - `withFileTypes` {boolean} **Default:** `false`
- Returns: {Promise} Fulfills with an array of the names of the files in the directory excluding `'.'` and `'..'`.

Reads the contents of a directory.

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.

```
import { readdir } from 'fs/promises';

try {
  const files = await readdir(path);
  for (const file of files)
    console.log(file);
} catch (err) {
  console.error(err);
}
```

`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'`.
  - `signal` {AbortSignal} allows aborting an in-progress readFile
- Returns: {Promise} Fulfills with the contents of the file.

Asynchronously reads the entire contents of a 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.

It is possible to abort an ongoing `readFile` using an {AbortSignal}. If a request is aborted the promise returned is rejected with an `AbortError`:

```
import { readFile } from 'fs/promises';

try {
  const controller = new AbortController();
  const { signal } = controller;
  const promise = readFile(fileName, { signal });

  // Abort the request before the promise settles.
  controller.abort();

  await promise;
} catch (err) {
  // When a request is aborted - err is an AbortError
  console.error(err);
}
```

Aborting an ongoing request does not abort individual operating system requests but rather the internal buffering `fs.readFile` performs.

Any specified {FileHandle} has to support reading.

`fsPromises.readlink(path[, options])`

- `path` {string|Buffer|URL}
- `options` {string|Object}
  - `encoding` {string} **Default:** `'utf8'`
- Returns: {Promise} Fulfills with the `linkString` upon success.

Reads the contents of the symbolic link referred to by `path`. See the POSIX readlink(2) documentation for more detail. 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} Fulfills with the resolved path upon success.

Determines the actual location of `path` using the same semantics as the `fs.realpath.native()` function.

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} Fulfills with `undefined` upon success.

Renames `oldPath` to `newPath`.

`fsPromises.rmdir(path[, options])`

- `path` {string|Buffer|URL}
- `options` {Object}
  - `maxRetries` {integer} If an `EBUSY`, `EMFILE`, `ENFILE`, `ENOTEMPTY`, or `EPERM` error is encountered, Node.js retries the operation with a linear backoff wait of `retryDelay` milliseconds longer on each try. This option represents the number of retries. This option is ignored if the `recursive` option is not `true`. **Default:** `0`.
  - `recursive` {boolean} If `true`, perform a recursive directory removal. In recursive mode, operations are retried on failure. **Default:** `false`. **Deprecated.**
  - `retryDelay` {integer} The amount of time in milliseconds to wait between retries. This option is ignored if the `recursive` option is not `true`. **Default:** `100`.
- Returns: {Promise} Fulfills with `undefined` upon success.

Removes the directory identified by `path`.

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.

To get a behavior similar to the `rm -rf` Unix command, use `fsPromises.rm()` with options `{ recursive: true, force: true }`.

`fsPromises.rm(path[, options])`

- `path` {string|Buffer|URL}
- `options` {Object}
  - `force` {boolean} When `true`, exceptions will be ignored if `path` does not exist. **Default:** `false`.
  - `maxRetries` {integer} If an `EBUSY`, `EMFILE`, `ENFILE`, `ENOTEMPTY`, or `EPERM` error is encountered, Node.js will retry the operation with a linear backoff wait of `retryDelay` milliseconds longer on each try. This option represents the number of retries. This option is ignored if the `recursive` option is not `true`. **Default:** `0`.
  - `recursive` {boolean} If `true`, perform a recursive directory removal. In recursive mode operations are retried on failure. **Default:** `false`.
  - `retryDelay` {integer} The amount of time in milliseconds to wait between retries. This option is ignored if the `recursive` option is not `true`. **Default:** `100`.
- Returns: {Promise} Fulfills with `undefined` upon success.

Removes files and directories (modeled on the standard POSIX `rm` utility).

`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} Fulfills 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} Fulfills with `undefined` upon success.

Creates a symbolic link.

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} Fulfills with `undefined` upon success.

Truncates (shortens or extends the length) of the content at `path` to `len` bytes.

`fsPromises.unlink(path)`

- `path` {string|Buffer|URL}
- Returns: {Promise} Fulfills with `undefined` upon success.

If `path` refers to a symbolic link, then the link is removed without affecting the file or directory to which that link refers. If the `path` refers to a file path that is not a symbolic link, the file is deleted. See the POSIX unlink(2) documentation for more detail.

`fsPromises.utimes(path, atime, mtime)`

- `path` {string|Buffer|URL}
- `atime` {number|string|Date}
- `mtime` {number|string|Date}
- Returns: {Promise} Fulfills with `undefined` upon success.

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.

`fsPromises.watch(filename[, options])`

- `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'`.
  - `signal` {AbortSignal} An {AbortSignal} used to signal when the watcher should stop.
- Returns: {AsyncIterator} of objects with the properties:
  - `eventType` {string} The type of change
  - `filename` {string|Buffer} The name of the file changed.

Returns an async iterator that watches for changes on `filename`, where `filename` is either a file or a directory.

```
const { watch } = require('fs/promises');

const ac = new AbortController();
const { signal } = ac;
setTimeout(() => ac.abort(), 10000);

(async () => {
  try {
    const watcher = watch(__filename, { signal });
    for await (const event of watcher)
      console.log(event);
  } catch (err) {
    if (err.name === 'AbortError')
```

```
      return;
    throw err;
  }
})();
```

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

All the caveats for `fs.watch()` also apply to `fsPromises.watch()`.

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

- `file` {string|Buffer|URL|FileHandle} filename or `FileHandle`
- `data` {string|Buffer|TypedArray|DataView|AsyncIterable|Iterable|Stream}
- `options` {Object|string}
  - `encoding` {string|null} **Default:** `'utf8'`
  - `mode` {integer} **Default:** `0o666`
  - `flag` {string} See support of file system `flags`. **Default:** `'w'`.
  - `signal` {AbortSignal} allows aborting an in-progress writeFile
- Returns: {Promise} Fulfills with `undefined` upon success.

Asynchronously writes data to a file, replacing the file if it already exists. `data` can be a string, a buffer, an {AsyncIterable} or {Iterable} object.

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

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

The `mode` option only affects the newly created file. See `fs.open()` for more details.

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 settled.

Similarly to `fsPromises.readFile` - `fsPromises.writeFile` is a convenience method that performs multiple `write` calls internally to write the buffer passed to it. For performance sensitive code consider using `fs.createWriteStream()`.

It is possible to use an {AbortSignal} to cancel an `fsPromises.writeFile()`. Cancelation is "best effort", and some amount of data is likely still to be written.

```
import { writeFile } from 'fs/promises';
import { Buffer } from 'buffer';

try {
  const controller = new AbortController();
  const { signal } = controller;
  const data = new Uint8Array(Buffer.from('Hello Node.js'));
  const promise = writeFile('message.txt', data, { signal });

  // Abort the request before the promise settles.
  controller.abort();
```

```
  await promise;
} catch (err) {
  // When a request is aborted - err is an AbortError
  console.error(err);
}
```

Aborting an ongoing request does not abort individual operating system requests but rather the internal buffering `fs.writeFile` performs.

Callback API

The callback APIs perform all operations asynchronously, without blocking the event loop, then invoke a callback function upon completion or error.

The callback APIs use the underlying Node.js threadpool to perform file system operations off the event loop thread. These operations are not synchronized or threadsafe. Care must be taken when performing multiple concurrent modifications on the same file or data corruption may occur.

`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. `mode` should be either the value `fs.constants.F_OK` or a mask consisting of the bitwise OR of any of `fs.constants.R_OK`, `fs.constants.W_OK`, and `fs.constants.X_OK` (e.g. `fs.constants.W_OK | fs.constants.R_OK`). Check File access constants for possible values of `mode`.

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.

```
import { access, constants } from 'fs';

const file = 'package.json';

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

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

// Check if the file is writable.
```

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

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

Do not use `fs.access()` to check for the accessibility of a file before calling `fs.open()`, `fs.readFile()` or `fs.writeFile()`. 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)

```
import { access, open, close } from 'fs';

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

  open('myfile', 'wx', (err, fd) => {
    if (err) throw err;

    try {
      writeMyData(fd);
    } finally {
      close(fd, (err) => {
        if (err) throw err;
      });
    }
  });
});
```

write (RECOMMENDED)

```
import { open, close } from 'fs';

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

    throw err;
  }
```

```
  try {
    writeMyData(fd);
  } finally {
    close(fd, (err) => {
      if (err) throw err;
    });
  }
});
```

read (NOT RECOMMENDED)

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

    throw err;
  }

  open('myfile', 'r', (err, fd) => {
    if (err) throw err;

    try {
      readMyData(fd);
    } finally {
      close(fd, (err) => {
        if (err) throw err;
      });
    }
  });
});
```

read (RECOMMENDED)

```
import { open, close } from 'fs';

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

    throw err;
  }

  try {
    readMyData(fd);
```

```
  } finally {
    close(fd, (err) => {
      if (err) throw err;
    });
  }
});
```

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.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}.

The `mode` option only affects the newly created file. See `fs.open()` for more details.

```
import { appendFile } from '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:

```
import { appendFile } from '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.

```
import { open, close, appendFile } from 'fs';
```

```
function closeFd(fd) {
  close(fd, (err) => {
    if (err) throw err;
  });
}

open('message.txt', 'a', (err, fd) => {
  if (err) throw err;

  try {
    appendFile(fd, 'data to append', 'utf8', (err) => {
      closeFd(fd);
      if (err) throw err;
    });
  } catch (err) {
    closeFd(fd);
    throw err;
  }
});
```

`fs.chmod(path, mode, callback)`

- `path` {string|Buffer|URL}
- `mode` {string|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 the POSIX chmod(2) documentation for more detail.

```
import { chmod } from 'fs';

chmod('my_file.txt', 0o775, (err) => {
  if (err) throw err;
  console.log('The permissions for file "my_file.txt" have been changed!');
});
```

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.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 the POSIX chown(2) documentation for more detail.

`fs.close(fd[, callback])`

- `fd` {integer}
- `callback` {Function}
  - `err` {Error}

Closes the file descriptor. No arguments other than a possible exception are given to the completion callback.

Calling `fs.close()` on any file descriptor (`fd`) that is currently in use through any other `fs` operation may lead to undefined behavior.

See the POSIX close(2) documentation for more detail.

`fs.copyFile(src, dest[, mode], callback)`

- `src` {string|Buffer|URL} source filename to copy
- `dest` {string|Buffer|URL} destination filename of the copy operation
- `mode` {integer} 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.

`mode` 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.

```
import { copyFile, constants } from 'fs';

function callback(err) {
  if (err) throw err;
  console.log('source.txt was copied to destination.txt');
}

// destination.txt will be created or overwritten by default.
copyFile('source.txt', 'destination.txt', callback);

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

`fs.cp(src, dest[, options], callback)`

> *Stability: 1 - Experimental*

- `src` {string|URL} source path to copy.
- `dest` {string|URL} destination path to copy to.
- `options` {Object}
  - `dereference` {boolean} dereference symlinks. **Default:** `false`.
  - `errorOnExist` {boolean} when `force` is `false`, and the destination exists, throw an error. **Default:** `false`.
  - `filter` {Function} Function to filter copied files/directories. Return `true` to copy the item, `false` to ignore it. Can also return a `Promise` that resolves to `true` or `false` **Default:** `undefined`.
  - `force` {boolean} overwrite existing file or directory. The copy operation will ignore errors if you set this to false and the destination exists. Use the `errorOnExist` option to change this behavior. **Default:** `true`.
  - `preserveTimestamps` {boolean} When `true` timestamps from `src` will be preserved. **Default:** `false`.
  - `recursive` {boolean} copy directories recursively **Default:** `false`
  - `verbatimSymlinks` {boolean} When `true`, path resolution for symlinks will be skipped. **Default:** `false`
- `callback` {Function}

Asynchronously copies the entire directory structure from `src` to `dest`, including subdirectories and files.

When copying a directory to another directory, globs are not supported and behavior is similar to `cp dir1/ dir2/`.

`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|FileHandle} **Default:** `null`
  - `mode` {integer} **Default:** `0o666`
  - `autoClose` {boolean} **Default:** `true`
  - `emitClose` {boolean} **Default:** `true`
  - `start` {integer}
  - `end` {integer} **Default:** `Infinity`
  - `highWaterMark` {integer} **Default:** `64 * 1024`
  - `fs` {Object|null} **Default:** `null`
- Returns: {fs.ReadStream}

Unlike the 16 kb default `highWaterMark` for a {stream.Readable}, 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, allowed values are in the [0, `Number.MAX_SAFE_INTEGER`] range. 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.

By default, the stream will emit a `'close'` event after it has been destroyed. Set the `emitClose` option to `false` to change this behavior.

By providing the `fs` option, it is possible to override the corresponding `fs` implementations for `open`, `read`, and `close`. When providing the `fs` option, an override for `read` is required. If no `fd` is provided, an override for `open` is also required. If `autoClose` is `true`, an override for `close` is also required.

```
import { createReadStream } from 'fs';

// Create a stream from some character device.
const stream = 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:

```
import { createReadStream } from '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|FileHandle} **Default:** `null`
    - `mode` {integer} **Default:** `0o666`
    - `autoClose` {boolean} **Default:** `true`
    - `emitClose` {boolean} **Default:** `true`
    - `start` {integer}
    - `fs` {Object|null} **Default:** `null`
- Returns: {fs.WriteStream}

`options` may also include a `start` option to allow writing data at some position past the beginning of the file, allowed values are in the [0, `Number.MAX_SAFE_INTEGER`] range. Modifying a file rather than replacing it may require the `flags` option to be set to `r+` rather than the default `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.

By default, the stream will emit a `'close'` event after it has been destroyed. Set the `emitClose` option to `false` to change this behavior.

By providing the `fs` option it is possible to override the corresponding `fs` implementations for `open` , `write` , `writev` and `close` . Overriding `write()` without `writev()` can reduce performance as some optimizations (`_writev()`) will be disabled. When providing the `fs` option, overrides for at least one of `write` and `writev` are required. If no `fd` option is supplied, an override for `open` is also required. If `autoClose` is `true` , an override for `close` is also required.

Like {fs.ReadStream}, if `fd` is specified, {fs.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:

```
import { exists } from 'fs';

exists('/etc/passwd', (e) => {
  console.log(e ? 'it exists' : '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)

```
import { exists, open, close } from 'fs';

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

      try {
        writeMyData(fd);
      } finally {
        close(fd, (err) => {
          if (err) throw err;
        });
      }
    });
  }
});
```

write (RECOMMENDED)

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

    throw err;
  }

  try {
    writeMyData(fd);
  } finally {
    close(fd, (err) => {
      if (err) throw err;
    });
```

```
  }
});
```

read (NOT RECOMMENDED)

```
import { open, close, exists } from 'fs';

exists('myfile', (e) => {
  if (e) {
    open('myfile', 'r', (err, fd) => {
      if (err) throw err;

      try {
        readMyData(fd);
      } finally {
        close(fd, (err) => {
          if (err) throw err;
        });
      }
    });
  } else {
    console.error('myfile does not exist');
  }
});
```

read (RECOMMENDED)

```
import { open, close } from 'fs';

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

    throw err;
  }

  try {
    readMyData(fd);
  } finally {
    close(fd, (err) => {
      if (err) throw err;
    });
  }
});
```

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.fchmod(fd, mode, callback)`

- `fd` {integer}
- `mode` {string|integer}
- `callback` {Function}
  - `err` {Error}

Sets the permissions on the file. No arguments other than a possible exception are given to the completion callback.

See the POSIX fchmod(2) documentation for more detail.

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

- `fd` {integer}
- `uid` {integer}
- `gid` {integer}
- `callback` {Function}
  - `err` {Error}

Sets the owner of the file. No arguments other than a possible exception are given to the completion callback.

See the POSIX fchown(2) documentation for more detail.

`fs.fdatasync(fd, callback)`

- `fd` {integer}
- `callback` {Function}
  - `err` {Error}

Forces all currently queued I/O operations associated with the file to the operating system's synchronized I/O completion state. Refer to the POSIX fdatasync(2) documentation for details. No arguments other than a possible exception are given to the completion callback.

`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}

Invokes the callback with the {fs.Stats} for the file descriptor.

See the POSIX fstat(2) documentation for more detail.

`fs.fsync(fd, callback)`

- `fd` {integer}

- `callback` {Function}
  - `err` {Error}

Request that all data for the open file descriptor is flushed to the storage device. The specific implementation is operating system and device specific. Refer to the POSIX fsync(2) documentation for more detail. No arguments other than a possible exception are given to the completion callback.

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

- `fd` {integer}
- `len` {integer} **Default:** `0`
- `callback` {Function}
  - `err` {Error}

Truncates the file descriptor. No arguments other than a possible exception are given to the completion callback.

See the POSIX ftruncate(2) documentation for more detail.

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:

```
import { open, close, ftruncate } from 'fs';

function closeFd(fd) {
  close(fd, (err) => {
    if (err) throw err;
  });
}

open('temp.txt', 'r+', (err, fd) => {
  if (err) throw err;

  try {
    ftruncate(fd, 4, (err) => {
      closeFd(fd);
      if (err) throw err;
    });
  } catch (err) {
    closeFd(fd);
    if (err) throw err;
  }
});
```

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

If `len` is negative then `0` will be used.

`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()`.

`fs.lchmod(path, mode, callback)`

- `path` {string|Buffer|URL}
- `mode` {integer}
- `callback` {Function}
  - `err` {Error|AggregateError}

Changes the permissions on a symbolic link. No arguments other than a possible exception are given to the completion callback.

This method is only implemented on macOS.

See the POSIX lchmod(2) documentation for more detail.

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

- `path` {string|Buffer|URL}
- `uid` {integer}
- `gid` {integer}
- `callback` {Function}
  - `err` {Error}

Set the owner of the symbolic link. No arguments other than a possible exception are given to the completion callback.

See the POSIX lchown(2) documentation for more detail.

`fs.lutimes(path, atime, mtime, callback)`

- `path` {string|Buffer|URL}
- `atime` {number|string|Date}
- `mtime` {number|string|Date}
- `callback` {Function}
  - `err` {Error}

Changes the access and modification times of a file in the same way as `fs.utimes()`, with the difference that if the path refers to a symbolic link, then the link is not dereferenced: instead, the timestamps of the symbolic link itself are changed.

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

`fs.link(existingPath, newPath, callback)`

- `existingPath` {string|Buffer|URL}
- `newPath` {string|Buffer|URL}
- `callback` {Function}

- `err` {Error}

Creates a new link from the `existingPath` to the `newPath`. See the POSIX link(2) documentation for more detail. No arguments other than a possible exception are given to the completion callback.

`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}

Retrieves the {fs.Stats} for the symbolic link referred to by the path. 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.

See the POSIX lstat(2) documentation for more details.

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

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

Asynchronously creates a directory.

The callback is given a possible exception and, if `recursive` is `true`, the first directory path created, `(err[, path])`. `path` can still be `undefined` when `recursive` is `true`, if no directory was created.

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 directories should be created. Calling `fs.mkdir()` when `path` is a directory that exists results in an error only when `recursive` is false.

```
import { mkdir } from 'fs';

// Creates /tmp/a/apple, regardless of whether `/tmp` and /tmp/a exist.
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:

```
import { mkdir } from 'fs';
```

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

See the POSIX mkdir(2) documentation for more details.

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

- `prefix` {string}
- `options` {string|Object}
  - `encoding` {string} **Default:** `'utf8'`
- `callback` {Function}
  - `err` {Error}
  - `directory` {string}

Creates a unique temporary directory.

Generates six random characters to be appended behind a required `prefix` to create a unique temporary directory. Due to platform inconsistencies, avoid trailing `X` characters in `prefix`. Some platforms, notably the BSDs, can return more than six random characters, and replace trailing `X` characters in `prefix` with random characters.

The created directory 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.

```
import { mkdtemp } from 'fs';

mkdtemp(path.join(os.tmpdir(), 'foo-'), (err, directory) => {
  if (err) throw err;
  console.log(directory);
  // 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`).

```
import { tmpdir } from 'os';
import { mkdtemp } from 'fs';

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

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

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

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

Asynchronous file open. See the POSIX open(2) documentation for more details.

`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.opendir(path[, options], callback)`

- `path` {string|Buffer|URL}
- `options` {Object}
  - `encoding` {string|null} **Default:** `'utf8'`
  - `bufferSize` {number} Number of directory entries that are buffered internally when reading from the directory. Higher values lead to better performance but higher memory usage. **Default:** `32`
- `callback` {Function}
  - `err` {Error}
  - `dir` {fs.Dir}

Asynchronously open a directory. See the POSIX opendir(3) documentation for more details.

Creates an {fs.Dir}, which contains all further functions for reading from and cleaning up the directory.

The `encoding` option sets the encoding for the `path` while opening the directory and subsequent read operations.

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

- `fd` {integer}
- `buffer` {Buffer|TypedArray|DataView} The buffer that the data will be written to.
- `offset` {integer} The position in `buffer` to write the data to.
- `length` {integer} The number of bytes to read.
- `position` {integer|bigint} Specifies where to begin reading from in the file. If `position` is `null` or `-1` , 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 be unchanged.
- `callback` {Function}
  - `err` {Error}
  - `bytesRead` {integer}
  - `buffer` {Buffer}

Read data from the file specified by `fd` .

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

If the file is not modified concurrently, the end-of-file is reached when the number of bytes read is zero.

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

`fs.read(fd, [options,] callback)`

- `fd` {integer}
- `options` {Object}
  - `buffer` {Buffer|TypedArray|DataView} **Default:** `Buffer.alloc(16384)`
  - `offset` {integer} **Default:** `0`
  - `length` {integer} **Default:** `buffer.byteLength - offset`
  - `position` {integer|bigint} **Default:** `null`
- `callback` {Function}
  - `err` {Error}
  - `bytesRead` {integer}
  - `buffer` {Buffer}

Similar to the `fs.read()` function, this version takes an optional `options` object. If no `options` object is specified, it will default with the above values.

`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[]}

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 `'..'`.

See the POSIX readdir(3) documentation for more details.

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.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'`.
  - `signal` {AbortSignal} allows aborting an in-progress readFile
- `callback` {Function}
  - `err` {Error|AggregateError}
  - `data` {string|Buffer}

Asynchronously reads the entire contents of a file.

```
import { readFile } from '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:

```
import { readFile } from '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.

```
import { readFile } from 'fs';

// macOS, Linux, and Windows
readFile('<directory>', (err, data) => {
```

```
  // => [Error: EISDIR: illegal operation on a directory, read <directory>]
});

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

It is possible to abort an ongoing request using an `AbortSignal`. If a request is aborted the callback is called with an `AbortError`:

```
import { readFile } from 'fs';

const controller = new AbortController();
const signal = controller.signal;
readFile(fileInfo[0].name, { signal }, (err, buf) => {
  // ...
});
// When you want to abort the request
controller.abort();
```

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

Aborting an ongoing request does not abort individual operating system requests but rather the internal buffering `fs.readFile` performs.

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'`.

Performance Considerations

The `fs.readFile()` method asynchronously reads the contents of a file into memory one chunk at a time, allowing the event loop to turn between each chunk. This allows the read operation to have less impact on other activity that may be using the underlying libuv thread pool but means that it will take longer to read a complete file into memory.

The additional read overhead can vary broadly on different systems and depends on the type of file being read. If the file type is not a regular file (a pipe for instance) and Node.js is unable to determine an actual file size, each read operation will load on 64 KB of data. For regular files, each read will process 512 KB of data.

For applications that require as-fast-as-possible reading of file contents, it is better to use `fs.read()` directly and for application code to manage reading the full contents of the file itself.

The Node.js GitHub issue #25741 provides more information and a detailed analysis on the performance of `fs.readFile()` for multiple file sizes in different Node.js versions.

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

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

Reads the contents of the symbolic link referred to by `path`. The callback gets two arguments `(err, linkString)`.

See the POSIX readlink(2) documentation for more details.

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.readv(fd, buffers[, position], callback)`

- `fd` {integer}
- `buffers` {ArrayBufferView[]}
- `position` {integer}
- `callback` {Function}
  - `err` {Error}
  - `bytesRead` {integer}
  - `buffers` {ArrayBufferView[]}

Read from a file specified by `fd` and write to an array of `ArrayBufferView`s using `readv()`.

`position` is the offset from the beginning of the file from where data should be read. If `typeof position !== 'number'`, the data will be read from the current position.

The callback will be given three arguments: `err`, `bytesRead`, and `buffers`. `bytesRead` is how many bytes were read from the file.

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

`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.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. If there is a directory at `newPath`, an error will be raised instead. No arguments other than a possible exception are given to the completion callback.

See also: rename(2).

```
import { rename } from 'fs';

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

`fs.rmdir(path[, options], callback)`

- `path` {string|Buffer|URL}
- `options` {Object}
  - `maxRetries` {integer} If an `EBUSY`, `EMFILE`, `ENFILE`, `ENOTEMPTY`, or `EPERM` error is encountered, Node.js retries the operation with a linear backoff wait of `retryDelay` milliseconds longer on each try. This option represents the number of retries. This option is ignored if the `recursive` option is not `true`. **Default:** `0`.
  - `recursive` {boolean} If `true`, perform a recursive directory removal. In recursive mode, operations are retried on failure. **Default:** `false`. **Deprecated.**
  - `retryDelay` {integer} The amount of time in milliseconds to wait between retries. This option is ignored if the `recursive` option is not `true`. **Default:** `100`.
- `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.

To get a behavior similar to the `rm -rf` Unix command, use `fs.rm()` with options `{ recursive: true, force: true }`.

`fs.rm(path[, options], callback)`

- `path` {string|Buffer|URL}
- `options` {Object}
  - `force` {boolean} When `true`, exceptions will be ignored if `path` does not exist. **Default:** `false`.
  - `maxRetries` {integer} If an `EBUSY`, `EMFILE`, `ENFILE`, `ENOTEMPTY`, or `EPERM` error is encountered, Node.js will retry the operation with a linear backoff wait of `retryDelay` milliseconds longer on each try. This option represents the number of retries. This option is ignored if the `recursive` option is not `true`. **Default:** `0`.
  - `recursive` {boolean} If `true`, perform a recursive removal. In recursive mode operations are retried on failure. **Default:** `false`.
  - `retryDelay` {integer} The amount of time in milliseconds to wait between retries. This option is ignored if the `recursive` option is not `true`. **Default:** `100`.
- `callback` {Function}
  - `err` {Error}

Asynchronously removes files and directories (modeled on the standard POSIX `rm` utility). No arguments other than a possible exception are given to the completion callback.

`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.

For example, given the following directory structure:

```
- txtDir
-- file.txt
- app.js
```

The next program will check for the stats of the given paths:

```
import { stat } from 'fs';

const pathsToCheck = ['./txtDir', './txtDir/file.txt'];

for (let i = 0; i < pathsToCheck.length; i++) {
  stat(pathsToCheck[i], (err, stats) => {
    console.log(stats.isDirectory());
    console.log(stats);
  });
}
```

The resulting output will resemble:

```
true
Stats {
  dev: 16777220,
  mode: 16877,
  nlink: 3,
  uid: 501,
  gid: 20,
  rdev: 0,
  blksize: 4096,
```

```
  ino: 14214262,
  size: 96,
  blocks: 0,
  atimeMs: 1561174653071.963,
  mtimeMs: 1561174614583.3518,
  ctimeMs: 1561174626623.5366,
  birthtimeMs: 1561174126937.2893,
  atime: 2019-06-22T03:37:33.072Z,
  mtime: 2019-06-22T03:36:54.583Z,
  ctime: 2019-06-22T03:37:06.624Z,
  birthtime: 2019-06-22T03:28:46.937Z
}
false
Stats {
  dev: 16777220,
  mode: 33188,
  nlink: 1,
  uid: 501,
  gid: 20,
  rdev: 0,
  blksize: 4096,
  ino: 14214074,
  size: 8,
  blocks: 8,
  atimeMs: 1561174616618.8555,
  mtimeMs: 1561174614584,
  ctimeMs: 1561174614583.8145,
  birthtimeMs: 1561174007710.7478,
  atime: 2019-06-22T03:36:56.619Z,
  mtime: 2019-06-22T03:36:54.584Z,
  ctime: 2019-06-22T03:36:54.584Z,
  birthtime: 2019-06-22T03:26:47.711Z
}
```

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

- `target` {string|Buffer|URL}
- `path` {string|Buffer|URL}
- `type` {string}
- `callback` {Function}
  - `err` {Error}

Creates the link called `path` pointing to `target`. No arguments other than a possible exception are given to the completion callback.

See the POSIX symlink(2) documentation for more details.

The `type` argument is only available on Windows and ignored on other platforms. It can be set to `'dir'`, `'file'`, or `'junction'`. If the `type` argument is not set, Node.js will autodetect `target` type and use `'file'` or `'dir'`. If the `target` does not exist, `'file'` will be used. Windows junction points require the

destination path to be absolute. When using `'junction'`, the `target` argument will automatically be normalized to absolute path.

Relative targets are relative to the link's parent directory.

```
import { symlink } from 'fs';

symlink('./mew', './mewtwo', callback);
```

The above example creates a symbolic link `mewtwo` which points to `mew` in the same directory:

```
$ tree .
.
├── mew
└── mewtwo -> ./mew
```

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

- `path` {string|Buffer|URL}
- `len` {integer} **Default:** `0`
- `callback` {Function}
  - `err` {Error|AggregateError}

Truncates the file. 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.

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

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

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

See the POSIX truncate(2) documentation for more details.

`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.

```
import { unlink } from 'fs';
// Assuming that 'path/file.txt' is a regular file.
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 the POSIX unlink(2) documentation for more details.

`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 in seconds, `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.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'`.
    - `signal` {AbortSignal} allows closing the watcher with an AbortSignal.
- `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`.

If a `signal` is passed, aborting the corresponding AbortController will close the returned {fs.FSWatcher}.

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. An `ERR_FEATURE_UNAVAILABLE_ON_PLATFORM` exception will be thrown when the option is used on a platform that does not support it.

On Windows, no events will be emitted if the watched directory is moved or renamed. An `EPERM` error is reported when the watched directory is deleted.

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.
- On IBM i systems, this feature is not supported.

If the underlying functionality is not available for some reason, then `fs.watch()` will not be able to function and may throw an exception. 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 or Docker.

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`.

```
import { watch } from '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}
  - `bigint` {boolean} **Default:** `false`
  - `persistent` {boolean} **Default:** `true`
  - `interval` {integer} **Default:** `5007`
- `listener` {Function}
  - `current` {fs.Stats}
  - `previous` {fs.Stats}
- Returns: {fs.StatWatcher}

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:

```
import { watchFile } from '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`. If the `bigint` option is `true`, the numeric values in these objects are specified as `BigInt`s.

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

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). 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 contents of `previous` in the second callback event (the file's reappearance) will be the same as the contents of `previous` in the first callback event (its disappearance).

This happens when:

- the file is deleted, followed by a restore
- the file is renamed and then renamed a 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|Object}
- `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, or an object with an own `toString` function property, then an exception is thrown.

`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|Object}
- `options` {Object|string}
  - `encoding` {string|null} **Default:** `'utf8'`
  - `mode` {integer} **Default:** `0o666`
  - `flag` {string} See support of file system `flags`. **Default:** `'w'`.
  - `signal` {AbortSignal} allows aborting an in-progress writeFile
- `callback` {Function}
  - `err` {Error|AggregateError}

When `file` is a filename, asynchronously writes data to the file, replacing the file if it already exists. `data` can be a string or a buffer.

When `file` is a file descriptor, the behavior is similar to calling `fs.write()` directly (which is recommended). See the notes below on using a file descriptor.

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

The `mode` option only affects the newly created file. See `fs.open()` for more details.

If `data` is a plain object, it must have an own (not inherited) `toString` function property.

```
import { writeFile } from 'fs';
import { Buffer } from 'buffer';

const data = new Uint8Array(Buffer.from('Hello Node.js'));
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:

```
import { writeFile } from '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.

Similarly to `fs.readFile` - `fs.writeFile` is a convenience method that performs multiple `write` calls internally to write the buffer passed to it. For performance sensitive code consider using `fs.createWriteStream()`.

It is possible to use an {AbortSignal} to cancel an `fs.writeFile()`. Cancelation is "best effort", and some amount of data is likely still to be written.

```
import { writeFile } from 'fs';
import { Buffer } from 'buffer';

const controller = new AbortController();
const { signal } = controller;
const data = new Uint8Array(Buffer.from('Hello Node.js'));
writeFile('message.txt', data, { signal }, (err) => {
  // When a request is aborted - the callback is called with an AbortError
});
// When the request should be aborted
controller.abort();
```

Aborting an ongoing request does not abort individual operating system requests but rather the internal buffering `fs.writeFile` performs.

Using `fs.writeFile()` with file descriptors

When `file` is a file descriptor, the behavior is almost identical to directly calling `fs.write()` like:

```
import { write } from 'fs';
import { Buffer } from 'buffer';

write(fd, Buffer.from(data, options.encoding), callback);
```

The difference from directly calling `fs.write()` is that under some unusual conditions, `fs.write()` might write only part of the buffer and need to be retried to write the remaining data, whereas `fs.writeFile()` retries until the data is entirely written (or an error occurs).

The implications of this are a common source of confusion. In the file descriptor case, the file is not replaced! The data is not necessarily written to the beginning of the file, and the file's original data may remain before and/or after the newly written data.

For example, if `fs.writeFile()` is called twice in a row, first to write the string `'Hello'`, then to write the string `', World'`, the file would contain `'Hello, World'`, and might contain some of the file's original data (depending on the size of the original file, and the position of the file descriptor). If a file name had been used instead of a descriptor, the file would be guaranteed to contain only `', World'`.

`fs.writev(fd, buffers[, position], callback)`

- `fd` {integer}
- `buffers` {ArrayBufferView[]}
- `position` {integer}
- `callback` {Function}
  - `err` {Error}
  - `bytesWritten` {integer}
  - `buffers` {ArrayBufferView[]}

Write an array of `ArrayBufferView` s to the file specified by `fd` using `writev()`.

`position` is 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.

The callback will be given three arguments: `err`, `bytesWritten`, and `buffers`. `bytesWritten` is how many bytes were written from `buffers`.

If this method is `util.promisify()` ed, it returns a promise for an `Object` with `bytesWritten` and `buffers` properties.

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

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.

Synchronous API

The synchronous APIs perform all operations synchronously, blocking the event loop until the operation completes or fails.

`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. `mode` should be either the value `fs.constants.F_OK` or a mask consisting of the bitwise OR of any of `fs.constants.R_OK`, `fs.constants.W_OK`, and `fs.constants.X_OK` (e.g. `fs.constants.W_OK | fs.constants.R_OK`). Check File access constants for possible values of `mode`.

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

```
import { accessSync, constants } from 'fs';

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

`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}.

The `mode` option only affects the newly created file. See `fs.open()` for more details.

```
import { appendFileSync } from 'fs';

try {
  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:

```
import { appendFileSync } from '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.

```
import { openSync, closeSync, appendFileSync } from 'fs';

let fd;

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

`fs.chmodSync(path, mode)`

- `path` {string|Buffer|URL}
- `mode` {string|integer}

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

See the POSIX chmod(2) documentation for more detail.

`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 the POSIX chown(2) documentation for more detail.

`fs.closeSync(fd)`

- `fd` {integer}

Closes the file descriptor. Returns `undefined`.

Calling `fs.closeSync()` on any file descriptor (`fd`) that is currently in use through any other `fs` operation may lead to undefined behavior.

See the POSIX close(2) documentation for more detail.

`fs.copyFileSync(src, dest[, mode])`

- `src` {string|Buffer|URL} source filename to copy
- `dest` {string|Buffer|URL} destination filename of the copy operation
- `mode` {integer} 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.

`mode` 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.

```
import { copyFileSync, constants } from 'fs';

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

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

`fs.cpSync(src, dest[, options])`

> *Stability: 1 - Experimental*

- `src` {string|URL} source path to copy.
- `dest` {string|URL} destination path to copy to.
- `options` {Object}
  - `dereference` {boolean} dereference symlinks. **Default:** `false`.
  - `errorOnExist` {boolean} when `force` is `false`, and the destination exists, throw an error. **Default:** `false`.
  - `filter` {Function} Function to filter copied files/directories. Return `true` to copy the item, `false` to ignore it. **Default:** `undefined`
  - `force` {boolean} overwrite existing file or directory. The copy operation will ignore errors if you set this to false and the destination exists. Use the `errorOnExist` option to change this behavior. **Default:** `true`.
  - `preserveTimestamps` {boolean} When `true` timestamps from `src` will be preserved. **Default:** `false`.
  - `recursive` {boolean} copy directories recursively **Default:** `false`
  - `verbatimSymlinks` {boolean} When `true`, path resolution for symlinks will be skipped. **Default:** `false`

Synchronously copies the entire directory structure from `src` to `dest`, including subdirectories and files.

When copying a directory to another directory, globs are not supported and behavior is similar to `cp dir1/ dir2/`.

`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.

```
import { existsSync } from 'fs';

if (existsSync('/etc/passwd'))
  console.log('The path exists.');
```

`fs.fchmodSync(fd, mode)`

- `fd` {integer}
- `mode` {string|integer}

Sets the permissions on the file. Returns `undefined`.

See the POSIX fchmod(2) documentation for more detail.

`fs.fchownSync(fd, uid, gid)`

- `fd` {integer}
- `uid` {integer} The file's new owner's user id.
- `gid` {integer} The file's new group's group id.

Sets the owner of the file. Returns `undefined`.

See the POSIX fchown(2) documentation for more detail.

`fs.fdatasyncSync(fd)`

- `fd` {integer}

Forces all currently queued I/O operations associated with the file to the operating system's synchronized I/O completion state. Refer to the POSIX fdatasync(2) documentation for details. Returns `undefined`.

`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}

Retrieves the {fs.Stats} for the file descriptor.

See the POSIX fstat(2) documentation for more detail.

`fs.fsyncSync(fd)`

- `fd` {integer}

Request that all data for the open file descriptor is flushed to the storage device. The specific implementation is operating system and device specific. Refer to the POSIX fsync(2) documentation for more detail. Returns `undefined`.

`fs.ftruncateSync(fd[, len])`

- `fd` {integer}
- `len` {integer} **Default:** `0`

Truncates the file descriptor. Returns `undefined`.

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

`fs.futimesSync(fd, atime, mtime)`

- `fd` {integer}
- `atime` {number|string|Date}
- `mtime` {number|string|Date}

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

`fs.lchmodSync(path, mode)`

- `path` {string|Buffer|URL}
- `mode` {integer}

Changes the permissions on a symbolic link. Returns `undefined`.

This method is only implemented on macOS.

See the POSIX lchmod(2) documentation for more detail.

`fs.lchownSync(path, uid, gid)`

- `path` {string|Buffer|URL}
- `uid` {integer} The file's new owner's user id.
- `gid` {integer} The file's new group's group id.

Set the owner for the path. Returns `undefined`.

See the POSIX lchown(2) documentation for more details.

`fs.lutimesSync(path, atime, mtime)`

- `path` {string|Buffer|URL}
- `atime` {number|string|Date}
- `mtime` {number|string|Date}

Change the file system timestamps of the symbolic link referenced by `path`. Returns `undefined`, or throws an exception when parameters are incorrect or the operation fails. This is the synchronous version of `fs.lutimes()`.

`fs.linkSync(existingPath, newPath)`

- `existingPath` {string|Buffer|URL}
- `newPath` {string|Buffer|URL}

Creates a new link from the `existingPath` to the `newPath`. See the POSIX link(2) documentation for more detail. Returns `undefined`.

`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`.
  - `throwIfNoEntry` {boolean} Whether an exception will be thrown if no file system entry exists, rather than returning `undefined`. **Default:** `true`.
- Returns: {fs.Stats}

Retrieves the {fs.Stats} for the symbolic link referred to by `path`.

See the POSIX lstat(2) documentation for more details.

`fs.mkdirSync(path[, options])`

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

Synchronously creates a directory. Returns `undefined`, or if `recursive` is `true`, the first directory path created. This is the synchronous version of `fs.mkdir()`.

See the POSIX mkdir(2) documentation for more details.

`fs.mkdtempSync(prefix[, options])`

- `prefix` {string}
- `options` {string|Object}
  - `encoding` {string} **Default:** `'utf8'`
- Returns: {string}

Returns the created directory 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.opendirSync(path[, options])`

- `path` {string|Buffer|URL}
- `options` {Object}
  - `encoding` {string|null} **Default:** `'utf8'`
  - `bufferSize` {number} Number of directory entries that are buffered internally when reading from the directory. Higher values lead to better performance but higher memory usage. **Default:** `32`

- Returns: {fs.Dir}

Synchronously open a directory. See opendir(3).

Creates an {fs.Dir}, which contains all further functions for reading from and cleaning up the directory.

The `encoding` option sets the encoding for the `path` while opening the directory and subsequent read operations.

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

- `path` {string|Buffer|URL}
- `flags` {string|number} **Default:** `'r'`. See support of file system `flags`.
- `mode` {string|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.readdirSync(path[, options])`

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

Reads the contents of the directory.

See the POSIX readdir(3) documentation for more details.

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.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.

```
import { readFileSync } from 'fs';

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

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

`fs.readlinkSync(path[, options])`

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

Returns the symbolic link's string value.

See the POSIX readlink(2) documentation for more details.

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|bigint}
- Returns: {number}

Returns the number of `bytesRead`.

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

`fs.readSync(fd, buffer[, options])`

- `fd` {integer}
- `buffer` {Buffer|TypedArray|DataView}
- `options` {Object}
  - `offset` {integer} **Default:** `0`
  - `length` {integer} **Default:** `buffer.byteLength - offset`
  - `position` {integer|bigint} **Default:** `null`
- Returns: {number}

Returns the number of `bytesRead`.

Similar to the above `fs.readSync` function, this version takes an optional `options` object. If no `options` object is specified, it will default with the above values.

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

`fs.readvSync(fd, buffers[, position])`

- `fd` {integer}
- `buffers` {ArrayBufferView[]}
- `position` {integer}
- Returns: {number} The number of bytes read.

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

`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.renameSync(oldPath, newPath)`

- `oldPath` {string|Buffer|URL}
- `newPath` {string|Buffer|URL}

Renames the file from `oldPath` to `newPath`. Returns `undefined`.

See the POSIX rename(2) documentation for more details.

`fs.rmdirSync(path[, options])`

- `path` {string|Buffer|URL}
- `options` {Object}
  - `maxRetries` {integer} If an `EBUSY`, `EMFILE`, `ENFILE`, `ENOTEMPTY`, or `EPERM` error is encountered, Node.js retries the operation with a linear backoff wait of `retryDelay`

milliseconds longer on each try. This option represents the number of retries. This option is ignored if the `recursive` option is not `true`. **Default:** `0`.
    - `recursive` {boolean} If `true`, perform a recursive directory removal. In recursive mode, operations are retried on failure. **Default:** `false`. **Deprecated.**
    - `retryDelay` {integer} The amount of time in milliseconds to wait between retries. This option is ignored if the `recursive` option is not `true`. **Default:** `100`.

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.

To get a behavior similar to the `rm -rf` Unix command, use `fs.rmSync()` with options `{ recursive: true, force: true }`.

`fs.rmSync(path[, options])`

- `path` {string|Buffer|URL}
- `options` {Object}
    - `force` {boolean} When `true`, exceptions will be ignored if `path` does not exist. **Default:** `false`.
    - `maxRetries` {integer} If an `EBUSY`, `EMFILE`, `ENFILE`, `ENOTEMPTY`, or `EPERM` error is encountered, Node.js will retry the operation with a linear backoff wait of `retryDelay` milliseconds longer on each try. This option represents the number of retries. This option is ignored if the `recursive` option is not `true`. **Default:** `0`.
    - `recursive` {boolean} If `true`, perform a recursive directory removal. In recursive mode operations are retried on failure. **Default:** `false`.
    - `retryDelay` {integer} The amount of time in milliseconds to wait between retries. This option is ignored if the `recursive` option is not `true`. **Default:** `100`.

Synchronously removes files and directories (modeled on the standard POSIX `rm` utility). Returns `undefined`.

`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`.
    - `throwIfNoEntry` {boolean} Whether an exception will be thrown if no file system entry exists, rather than returning `undefined`. **Default:** `true`.
- Returns: {fs.Stats}

Retrieves the {fs.Stats} for the path.

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

- `target` {string|Buffer|URL}
- `path` {string|Buffer|URL}
- `type` {string}

Returns `undefined`.

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

`fs.truncateSync(path[, len])`

- `path` {string|Buffer|URL}
- `len` {integer} **Default:** `0`

Truncates the file. 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.unlinkSync(path)`

- `path` {string|Buffer|URL}

Synchronous unlink(2). Returns `undefined`.

`fs.utimesSync(path, atime, mtime)`

- `path` {string|Buffer|URL}
- `atime` {number|string|Date}
- `mtime` {number|string|Date}

Returns `undefined`.

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

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

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

Returns `undefined`.

If `data` is a plain object, it must have an own (not inherited) `toString` function property.

The `mode` option only affects the newly created file. See `fs.open()` for more details.

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.writevSync(fd, buffers[, position])`

- `fd` {integer}
- `buffers` {ArrayBufferView[]}
- `position` {integer}
- Returns: {number} The number of bytes written.

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

Common Objects

The common objects are shared by all of the file system API variants (promise, callback, and synchronous).

Class: `fs.Dir`

A class representing a directory stream.

Created by `fs.opendir()`, `fs.opendirSync()`, or `fsPromises.opendir()`.

```
import { opendir } from 'fs/promises';

try {
  const dir = await opendir('./');
  for await (const dirent of dir)
    console.log(dirent.name);
} catch (err) {
  console.error(err);
}
```

When using the async iterator, the {fs.Dir} object will be automatically closed after the iterator exits.

`dir.close()`

- Returns: {Promise}

Asynchronously close the directory's underlying resource handle. Subsequent reads will result in errors.

A promise is returned that will be resolved after the resource has been closed.

`dir.close(callback)`

- `callback` {Function}
  - `err` {Error}

Asynchronously close the directory's underlying resource handle. Subsequent reads will result in errors.

The `callback` will be called after the resource handle has been closed.

`dir.closeSync()`

Synchronously close the directory's underlying resource handle. Subsequent reads will result in errors.

`dir.path`

- {string}

The read-only path of this directory as was provided to `fs.opendir()`, `fs.opendirSync()`, or `fsPromises.opendir()`.

`dir.read()`

- Returns: {Promise} containing {fs.Dirent|null}

Asynchronously read the next directory entry via readdir(3) as an {fs.Dirent}.

A promise is returned that will be resolved with an {fs.Dirent}, or `null` if there are no more directory entries to read.

Directory entries returned by this function are in no particular order as provided by the operating system's underlying directory mechanisms. Entries added or removed while iterating over the directory might not be included in the iteration results.

`dir.read(callback)`

- `callback` {Function}
  - `err` {Error}
  - `dirent` {fs.Dirent|null}

Asynchronously read the next directory entry via readdir(3) as an {fs.Dirent}.

After the read is completed, the `callback` will be called with an {fs.Dirent}, or `null` if there are no more directory entries to read.

Directory entries returned by this function are in no particular order as provided by the operating system's underlying directory mechanisms. Entries added or removed while iterating over the directory might not be included in the iteration results.

`dir.readSync()`

- Returns: {fs.Dirent|null}

Synchronously read the next directory entry as an {fs.Dirent}. See the POSIX readdir(3) documentation for more detail.

If there are no more directory entries to read, `null` will be returned.

Directory entries returned by this function are in no particular order as provided by the operating system's underlying directory mechanisms. Entries added or removed while iterating over the directory might not be included in the iteration results.

`dir[Symbol.asyncIterator]()`

- Returns: {AsyncIterator} of {fs.Dirent}

Asynchronously iterates over the directory until all entries have been read. Refer to the POSIX readdir(3) documentation for more detail.

Entries returned by the async iterator are always an {fs.Dirent}. The `null` case from `dir.read()` is handled internally.

See {fs.Dir} for an example.

Directory entries returned by this iterator are in no particular order as provided by the operating system's underlying directory mechanisms. Entries added or removed while iterating over the directory might not be included in the iteration results.

Class: `fs.Dirent`

A representation of a directory entry, which can be a file or a subdirectory within the directory, as returned by reading from an {fs.Dir}. The directory entry is a combination of the file name and file type pairs.

Additionally, 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 {Buffer}s.

`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`

- Extends {EventEmitter}

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

All {fs.FSWatcher} objects 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.

```
import { watch } from 'fs';
// Example when handled through fs.watch() listener
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.

`watcher.ref()`

- Returns: {fs.FSWatcher}

When called, requests that the Node.js event loop *not* exit so long as the {fs.FSWatcher} is active. Calling `watcher.ref()` multiple times will have no effect.

By default, all {fs.FSWatcher} objects are "ref'ed", making it normally unnecessary to call `watcher.ref()` unless `watcher.unref()` had been called previously.

`watcher.unref()`

- Returns: {fs.FSWatcher}

When called, the active {fs.FSWatcher} object will not require the Node.js event loop to remain active. If there is no other activity keeping the event loop running, the process may exit before the {fs.FSWatcher} object's callback is invoked. Calling `watcher.unref()` multiple times will have no effect.

Class: `fs.StatWatcher`

- Extends {EventEmitter}

A successful call to `fs.watchFile()` method will return a new {fs.StatWatcher} object.

`watcher.ref()`

- Returns: {fs.StatWatcher}

When called, requests that the Node.js event loop *not* exit so long as the {fs.StatWatcher} is active. Calling `watcher.ref()` multiple times will have no effect.

By default, all {fs.StatWatcher} objects are "ref'ed", making it normally unnecessary to call `watcher.ref()` unless `watcher.unref()` had been called previously.

`watcher.unref()`

- Returns: {fs.StatWatcher}

When called, the active {fs.StatWatcher} object will not require the Node.js event loop to remain active. If there is no other activity keeping the event loop running, the process may exit before the {fs.StatWatcher} object's callback is invoked. Calling `watcher.unref()` multiple times will have no effect.

Class: `fs.ReadStream`

- Extends: {stream.Readable}

Instances of {fs.ReadStream} are created and returned using the `fs.createReadStream()` function.

Event: `'close'`

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

Event: `'open'`

- `fd` {integer} Integer file descriptor used by the {fs.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}. If `fd` is specified, then `readStream.path` will be `undefined`.

`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`, and the object will contain additional nanosecond-precision properties suffixed with `Ns`.

```
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:

```
BigIntStats {
  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,
  atimeNs: 1318289051000000000n,
  mtimeNs: 1318289051000000000n,
  ctimeNs: 1318289051000000000n,
  birthtimeNs: 1318289051000000000n,
  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.

If the {fs.Stats} object was obtained from `fs.lstat()`, this method will always return `false`. This is because `fs.lstat()` returns information about a symbolic link itself and not the path it resolves to.

`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 represents a device.

`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.atimeNs`

- {bigint}

Only present when `bigint: true` is passed into the method that generates the object. The timestamp indicating the last time this file was accessed expressed in nanoseconds since the POSIX Epoch.

`stats.mtimeNs`

- {bigint}

Only present when `bigint: true` is passed into the method that generates the object. The timestamp indicating the last time this file was modified expressed in nanoseconds since the POSIX Epoch.

`stats.ctimeNs`

- {bigint}

Only present when `bigint: true` is passed into the method that generates the object. The timestamp indicating the last time the file status was changed expressed in nanoseconds since the POSIX Epoch.

`stats.birthtimeNs`

- {bigint}

Only present when `bigint: true` is passed into the method that generates the object. The timestamp indicating the creation time of this file expressed in nanoseconds 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 numeric values that hold the corresponding times in milliseconds. Their precision is platform specific. When `bigint: true` is passed into the method that generates the object, the properties will be bigints, otherwise they will be numbers.

The `atimeNs`, `mtimeNs`, `ctimeNs`, `birthtimeNs` properties are bigints that hold the corresponding times in nanoseconds. They are only present when `bigint: true` is passed into the method that generates the object. 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`

- Extends {stream.Writable}

Instances of {fs.WriteStream} are created and returned using the `fs.createWriteStream()` function.

Event: `'close'`

Emitted when the {fs.WriteStream}'s underlying file descriptor has been closed.

Event: `'open'`

- `fd` {integer} Integer file descriptor used by the {fs.WriteStream}.

Emitted when the {fs.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.close([callback])`

- `callback` {Function}
  - `err` {Error}

Closes `writeStream`. Optionally accepts a callback that will be executed once the `writeStream` is closed.

`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.constants`

- {Object}

Returns an object containing commonly used constants for file system operations.

FS constants

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

Not every constant will be available on every operating system.

To use more than one constant, use the bitwise OR `|` operator.

Example:

```
import { open, constants } from 'fs';

const {
  O_RDWR,
  O_CREAT,
```

```
  O_EXCL
} = constants;

open('/path/to/my/file', O_RDWR | O_CREAT | O_EXCL, (err, fd) => {
  // ...
});
```

File access constants

The following constants are meant for use as the `mode` parameter passed to `fsPromises.access()`, `fs.access()`, and `fs.accessSync()`.

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.
`UV_FS_O_FILEMAP`	When set, a memory file mapping is used to access the file. This flag is available on Windows operating systems only. On other operating systems, this flag is ignored.

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.

Notes

Ordering of callback and promise-based operations

Because they are executed asynchronously by the underlying thread pool, there is no guaranteed ordering when using either the callback or promise-based methods.

For example, the following is prone to error because the `fs.stat()` operation might 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)}`);
});
```

It is important to correctly order the operations by awaiting the results of one before invoking the other:

```
import { rename, stat } from 'fs/promises';

const from = '/tmp/hello';
const to = '/tmp/world';

try {
  await rename(from, to);
```

```
  const stats = await stat(to);
  console.log(`stats: ${JSON.stringify(stats)}`);
} catch (error) {
  console.error('there was an error:', error.message);
}
```

```
const { rename, stat } = require('fs/promises');

(async function(from, to) {
  try {
    await rename(from, to);
    const stats = await stat(to);
    console.log(`stats: ${JSON.stringify(stats)}`);
  } catch (error) {
    console.error('there was an error:', error.message);
  }
})('/tmp/hello', '/tmp/world');
```

Or, when using the callback APIs, move the `fs.stat()` call into the callback of the `fs.rename()` operation:

```
import { rename, stat } from 'fs';

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

```
const { rename, stat } = require('fs/promises');

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

File paths

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

String paths

String 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 determined by calling `process.cwd()`.

Example using an absolute path on POSIX:

```
import { open } from 'fs/promises';

let fd;
try {
  fd = await open('/open/some/file.txt', 'r');
  // Do something with the file
} finally {
  await fd.close();
}
```

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

```
import { open } from 'fs/promises';

let fd;
try {
  fd = await open('file.txt', 'r');
  // Do something with the file
} finally {
  await fd.close();
}
```

File URL paths

For most `fs` module functions, the `path` or `filename` argument may be passed as a {URL} object using the `file:` protocol.

```
import { readFileSync } from 'fs';

readFileSync(new URL('file:///tmp/hello'));
```

`file:` URLs are always absolute paths.

Platform-specific considerations

On Windows, `file:` {URL}s with a host name convert to UNC paths, while `file:` {URL}s with drive letters convert to local absolute paths. `file:` {URL}s with no host name and no drive letter will result in an error:

```
import { readFileSync } from 'fs';
// On Windows :

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

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

`file:` {URL}s with drive letters must use `:` as a separator just after the drive letter. Using another separator will result in an error.

On all other platforms, `file:` {URL}s with a host name are unsupported and will result in an error:

```
import { readFileSync } from 'fs';
// On other platforms:

// - WHATWG file URLs with hostname are unsupported
// file://hostname/p/a/t/h/file => throw!
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
readFileSync(new URL('file:///tmp/hello'));
```

A `file:` {URL} having encoded slash characters will result in an error on all platforms:

```
import { readFileSync } from 'fs';

// On Windows
readFileSync(new URL('file:///C:/p/a/t/h/%2F'));
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
readFileSync(new URL('file:///p/a/t/h/%2F'));
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:` {URL}s having encoded backslash will result in an error:

```
import { readFileSync } from 'fs';

// On Windows
readFileSync(new URL('file:///C:/path/%5C'));
readFileSync(new URL('file:///C:/path/%5c'));
/* TypeError [ERR_INVALID_FILE_URL_PATH]: File URL path must not include encoded
\ or / characters */
```

Buffer paths

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:

```
import { open } from 'fs/promises';
import { Buffer } from 'buffer';

let fd;
try {
  fd = await open(Buffer.from('/open/some/file.txt'), 'r');
  // Do something with the file
} finally {
  await fd.close();
}
```

Per-drive working directories on Windows

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.

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 differences between operating systems and assigns all open files a numeric file descriptor.

The callback-based `fs.open()`, and synchronous `fs.openSync()` methods open a file and 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.

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.

```
import { open, close, fstat } from 'fs';

function closeFd(fd) {
  close(fd, (err) => {
    if (err) throw err;
  });
}

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

```
        closeFd(fd);
        throw err;
      }

      // use stat

      closeFd(fd);
    });
  } catch (err) {
    closeFd(fd);
    throw err;
  }
});
```

The promise-based APIs use a {FileHandle} object in place of the numeric file descriptor. These objects are better managed by the system to ensure that resources are not leaked. However, it is still required that they are closed when operations are completed:

```
import { open } from 'fs/promises';

let file;
try {
  file = await open('/open/some/file.txt', 'r');
  const stat = await file.stat();
  // use stat
} finally {
  await file.close();
}
```

Threadpool usage

All callback and promise-based file system APIs (with the exception of `fs.FSWatcher()`) use libuv's threadpool. This can have surprising and negative performance implications for some applications. See the `UV_THREADPOOL_SIZE` documentation for more information.

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)) causes the operation to return an error if the path already exists. On POSIX, if the path is a symbolic link, using `O_EXCL` returns an error even if the link is to a path that does not exist. The exclusive flag might 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 the `flag` option to be set to `'r+'` rather than the default `'w'`.

The behavior of some flags are platform-specific. As such, opening a directory on macOS and Linux with the `'a+'` flag, as in the 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.