/\*\*

\* 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.

\* @since v10.0.0

\*/

declare module 'fs/promises' {

import { Abortable } from 'node:events';

import { Stream } from 'node:stream';

import {

Stats,

BigIntStats,

StatOptions,

WriteVResult,

ReadVResult,

PathLike,

RmDirOptions,

RmOptions,

MakeDirectoryOptions,

Dirent,

OpenDirOptions,

Dir,

ObjectEncodingOptions,

BufferEncodingOption,

OpenMode,

Mode,

WatchOptions,

WatchEventType,

CopyOptions,

ReadStream,

WriteStream,

} from 'node:fs';

interface FileChangeInfo<T extends string | Buffer> {

eventType: WatchEventType;

filename: T;

}

interface FlagAndOpenMode {

mode?: Mode | undefined;

flag?: OpenMode | undefined;

}

interface FileReadResult<T extends NodeJS.ArrayBufferView> {

bytesRead: number;

buffer: T;

}

interface FileReadOptions<T extends NodeJS.ArrayBufferView = Buffer> {

/\*\*

\* @default `Buffer.alloc(0xffff)`

\*/

buffer?: T;

/\*\*

\* @default 0

\*/

offset?: number | null;

/\*\*

\* @default `buffer.byteLength`

\*/

length?: number | null;

position?: number | null;

}

interface CreateReadStreamOptions {

encoding?: BufferEncoding | null | undefined;

autoClose?: boolean | undefined;

emitClose?: boolean | undefined;

start?: number | undefined;

end?: number | undefined;

highWaterMark?: number | undefined;

}

interface CreateWriteStreamOptions {

encoding?: BufferEncoding | null | undefined;

autoClose?: boolean | undefined;

emitClose?: boolean | undefined;

start?: number | undefined;

}

// TODO: Add `EventEmitter` close

interface FileHandle {

/\*\*

\* The numeric file descriptor managed by the {FileHandle} object.

\* @since v10.0.0

\*/

readonly fd: number;

/\*\*

\* 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()`.

\* @since v10.0.0

\* @return Fulfills with `undefined` upon success.

\*/

appendFile(data: string | Uint8Array, options?: (ObjectEncodingOptions & FlagAndOpenMode) | BufferEncoding | null): Promise<void>;

/\*\*

\* Changes the ownership of the file. A wrapper for [`chown(2)`](http://man7.org/linux/man-pages/man2/chown.2.html).

\* @since v10.0.0

\* @param uid The file's new owner's user id.

\* @param gid The file's new group's group id.

\* @return Fulfills with `undefined` upon success.

\*/

chown(uid: number, gid: number): Promise<void>;

/\*\*

\* Modifies the permissions on the file. See [`chmod(2)`](http://man7.org/linux/man-pages/man2/chmod.2.html).

\* @since v10.0.0

\* @param mode the file mode bit mask.

\* @return Fulfills with `undefined` upon success.

\*/

chmod(mode: Mode): Promise<void>;

/\*\*

\* 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

\* \[0, [`Number.MAX\_SAFE\_INTEGER`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Number/MAX\_SAFE\_INTEGER)\] 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.

\*

\* ```js

\* 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:

\*

\* ```js

\* import { open } from 'fs/promises';

\*

\* const fd = await open('sample.txt');

\* fd.createReadStream({ start: 90, end: 99 });

\* ```

\* @since v16.11.0

\*/

createReadStream(options?: CreateReadStreamOptions): ReadStream;

/\*\*

\* `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

\* \[0, [`Number.MAX\_SAFE\_INTEGER`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Number/MAX\_SAFE\_INTEGER)\] 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.

\* @since v16.11.0

\*/

createWriteStream(options?: CreateWriteStreamOptions): WriteStream;

/\*\*

\* 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)`](http://man7.org/linux/man-pages/man2/fdatasync.2.html) documentation for details.

\*

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

\* @since v10.0.0

\* @return Fulfills with `undefined` upon success.

\*/

datasync(): Promise<void>;

/\*\*

\* 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)`](http://man7.org/linux/man-pages/man2/fsync.2.html) documentation for more detail.

\* @since v10.0.0

\* @return Fufills with `undefined` upon success.

\*/

sync(): Promise<void>;

/\*\*

\* 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.

\* @since v10.0.0

\* @param buffer A buffer that will be filled with the file data read.

\* @param offset The location in the buffer at which to start filling.

\* @param length The number of bytes to read.

\* @param position 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.

\* @return Fulfills upon success with an object with two properties:

\*/

read<T extends NodeJS.ArrayBufferView>(buffer: T, offset?: number | null, length?: number | null, position?: number | null): Promise<FileReadResult<T>>;

read<T extends NodeJS.ArrayBufferView = Buffer>(options?: FileReadOptions<T>): Promise<FileReadResult<T>>;

/\*\*

\* 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.

\* @since v10.0.0

\* @return 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.

\*/

readFile(

options?: {

encoding?: null | undefined;

flag?: OpenMode | undefined;

} | null

): Promise<Buffer>;

/\*\*

\* Asynchronously reads the entire contents of a file. The underlying file will \_not\_ be closed automatically.

\* The `FileHandle` must have been opened for reading.

\* @param options An object that may contain an optional flag.

\* If a flag is not provided, it defaults to `'r'`.

\*/

readFile(

options:

| {

encoding: BufferEncoding;

flag?: OpenMode | undefined;

}

| BufferEncoding

): Promise<string>;

/\*\*

\* Asynchronously reads the entire contents of a file. The underlying file will \_not\_ be closed automatically.

\* The `FileHandle` must have been opened for reading.

\* @param options An object that may contain an optional flag.

\* If a flag is not provided, it defaults to `'r'`.

\*/

readFile(

options?:

| (ObjectEncodingOptions & {

flag?: OpenMode | undefined;

})

| BufferEncoding

| null

): Promise<string | Buffer>;

/\*\*

\* @since v10.0.0

\* @return Fulfills with an {fs.Stats} for the file.

\*/

stat(

opts?: StatOptions & {

bigint?: false | undefined;

}

): Promise<Stats>;

stat(

opts: StatOptions & {

bigint: true;

}

): Promise<BigIntStats>;

stat(opts?: StatOptions): Promise<Stats | BigIntStats>;

/\*\*

\* 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:

\*

\* ```js

\* 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.

\* @since v10.0.0

\* @param [len=0]

\* @return Fulfills with `undefined` upon success.

\*/

truncate(len?: number): Promise<void>;

/\*\*

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

\* @since v10.0.0

\*/

utimes(atime: string | number | Date, mtime: string | number | Date): Promise<void>;

/\*\*

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

\* [AsyncIterable](https://tc39.github.io/ecma262/#sec-asynciterable-interface) or

\* [Iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration\_protocols#The\_iterable\_protocol) object, or an

\* object with an own `toString` function

\* property. 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.

\* @since v10.0.0

\*/

writeFile(data: string | Uint8Array, options?: (ObjectEncodingOptions & FlagAndOpenMode & Abortable) | BufferEncoding | null): Promise<void>;

/\*\*

\* Write `buffer` to the file.

\*

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

\*

\* The promise is resolved with an object containing two properties:

\*

\* 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.

\* @since v10.0.0

\* @param [offset=0] The start position from within `buffer` where the data to write begins.

\* @param [length=buffer.byteLength] The number of bytes from `buffer` to write.

\* @param position 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.

\*/

write<TBuffer extends Uint8Array>(

buffer: TBuffer,

offset?: number | null,

length?: number | null,

position?: number | null

): Promise<{

bytesWritten: number;

buffer: TBuffer;

}>;

write(

data: string,

position?: number | null,

encoding?: BufferEncoding | null

): Promise<{

bytesWritten: number;

buffer: string;

}>;

/\*\*

\* Write an array of [ArrayBufferView](https://developer.mozilla.org/en-US/docs/Web/API/ArrayBufferView) s to the file.

\*

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

\*

\* 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.

\* @since v12.9.0

\* @param position 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.

\*/

writev(buffers: ReadonlyArray<NodeJS.ArrayBufferView>, position?: number): Promise<WriteVResult>;

/\*\*

\* Read from a file and write to an array of [ArrayBufferView](https://developer.mozilla.org/en-US/docs/Web/API/ArrayBufferView) s

\* @since v13.13.0, v12.17.0

\* @param position 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.

\* @return Fulfills upon success an object containing two properties:

\*/

readv(buffers: ReadonlyArray<NodeJS.ArrayBufferView>, position?: number): Promise<ReadVResult>;

/\*\*

\* Closes the file handle after waiting for any pending operation on the handle to

\* complete.

\*

\* ```js

\* import { open } from 'fs/promises';

\*

\* let filehandle;

\* try {

\* filehandle = await open('thefile.txt', 'r');

\* } finally {

\* await filehandle?.close();

\* }

\* ```

\* @since v10.0.0

\* @return Fulfills with `undefined` upon success.

\*/

close(): Promise<void>;

}

/\*\*

\* Tests a user's permissions for the file or directory specified by `path`.

\* The `mode` argument is an optional integer that specifies the accessibility

\* checks to be performed. Check `File access constants` for possible values

\* of `mode`. It is possible to create a mask consisting of the bitwise OR of

\* two or more values (e.g. `fs.constants.W\_OK | fs.constants.R\_OK`).

\*

\* If the accessibility check is successful, the promise is resolved with no

\* value. If any of the accessibility checks fail, the promise is rejected

\* with an [Error](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Error) object. The following example checks if the file`/etc/passwd` can be read and

\* written by the current process.

\*

\* ```js

\* 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.

\* @since v10.0.0

\* @param [mode=fs.constants.F\_OK]

\* @return Fulfills with `undefined` upon success.

\*/

function access(path: PathLike, mode?: number): Promise<void>;

/\*\*

\* 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.

\*

\* ```js

\* 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');

\* }

\* ```

\* @since v10.0.0

\* @param src source filename to copy

\* @param dest destination filename of the copy operation

\* @param [mode=0] 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`)

\* @return Fulfills with `undefined` upon success.

\*/

function copyFile(src: PathLike, dest: PathLike, mode?: number): Promise<void>;

/\*\*

\* Opens a `FileHandle`.

\*

\* Refer to the POSIX [`open(2)`](http://man7.org/linux/man-pages/man2/open.2.html) documentation for more detail.

\*

\* Some characters (`< > : " / \ | ? \*`) are reserved under Windows as documented

\* by [Naming Files, Paths, and Namespaces](https://docs.microsoft.com/en-us/windows/desktop/FileIO/naming-a-file). Under NTFS, if the filename contains

\* a colon, Node.js will open a file system stream, as described by [this MSDN page](https://docs.microsoft.com/en-us/windows/desktop/FileIO/using-streams).

\* @since v10.0.0

\* @param [flags='r'] See `support of file system `flags``.

\* @param [mode=0o666] Sets the file mode (permission and sticky bits) if the file is created.

\* @return Fulfills with a {FileHandle} object.

\*/

function open(path: PathLike, flags: string | number, mode?: Mode): Promise<FileHandle>;

/\*\*

\* Renames `oldPath` to `newPath`.

\* @since v10.0.0

\* @return Fulfills with `undefined` upon success.

\*/

function rename(oldPath: PathLike, newPath: PathLike): Promise<void>;

/\*\*

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

\* @since v10.0.0

\* @param [len=0]

\* @return Fulfills with `undefined` upon success.

\*/

function truncate(path: PathLike, len?: number): Promise<void>;

/\*\*

\* 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 }`.

\* @since v10.0.0

\* @return Fulfills with `undefined` upon success.

\*/

function rmdir(path: PathLike, options?: RmDirOptions): Promise<void>;

/\*\*

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

\* @since v14.14.0

\* @return Fulfills with `undefined` upon success.

\*/

function rm(path: PathLike, options?: RmOptions): Promise<void>;

/\*\*

\* 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.

\* @since v10.0.0

\* @return Upon success, fulfills with `undefined` if `recursive` is `false`, or the first directory path created if `recursive` is `true`.

\*/

function mkdir(

path: PathLike,

options: MakeDirectoryOptions & {

recursive: true;

}

): Promise<string | undefined>;

/\*\*

\* Asynchronous mkdir(2) - create a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options Either the file mode, or an object optionally specifying the file mode and whether parent folders

\* should be created. If a string is passed, it is parsed as an octal integer. If not specified, defaults to `0o777`.

\*/

function mkdir(

path: PathLike,

options?:

| Mode

| (MakeDirectoryOptions & {

recursive?: false | undefined;

})

| null

): Promise<void>;

/\*\*

\* Asynchronous mkdir(2) - create a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options Either the file mode, or an object optionally specifying the file mode and whether parent folders

\* should be created. If a string is passed, it is parsed as an octal integer. If not specified, defaults to `0o777`.

\*/

function mkdir(path: PathLike, options?: Mode | MakeDirectoryOptions | null): Promise<string | undefined>;

/\*\*

\* 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.

\*

\* ```js

\* import { readdir } from 'fs/promises';

\*

\* try {

\* const files = await readdir(path);

\* for (const file of files)

\* console.log(file);

\* } catch (err) {

\* console.error(err);

\* }

\* ```

\* @since v10.0.0

\* @return Fulfills with an array of the names of the files in the directory excluding `'.'` and `'..'`.

\*/

function readdir(

path: PathLike,

options?:

| (ObjectEncodingOptions & {

withFileTypes?: false | undefined;

})

| BufferEncoding

| null

): Promise<string[]>;

/\*\*

\* Asynchronous readdir(3) - read a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function readdir(

path: PathLike,

options:

| {

encoding: 'buffer';

withFileTypes?: false | undefined;

}

| 'buffer'

): Promise<Buffer[]>;

/\*\*

\* Asynchronous readdir(3) - read a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function readdir(

path: PathLike,

options?:

| (ObjectEncodingOptions & {

withFileTypes?: false | undefined;

})

| BufferEncoding

| null

): Promise<string[] | Buffer[]>;

/\*\*

\* Asynchronous readdir(3) - read a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options If called with `withFileTypes: true` the result data will be an array of Dirent.

\*/

function readdir(

path: PathLike,

options: ObjectEncodingOptions & {

withFileTypes: true;

}

): Promise<Dirent[]>;

/\*\*

\* Reads the contents of the symbolic link referred to by `path`. See the POSIX [`readlink(2)`](http://man7.org/linux/man-pages/man2/readlink.2.html) 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.

\* @since v10.0.0

\* @return Fulfills with the `linkString` upon success.

\*/

function readlink(path: PathLike, options?: ObjectEncodingOptions | BufferEncoding | null): Promise<string>;

/\*\*

\* Asynchronous readlink(2) - read value of a symbolic link.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function readlink(path: PathLike, options: BufferEncodingOption): Promise<Buffer>;

/\*\*

\* Asynchronous readlink(2) - read value of a symbolic link.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function readlink(path: PathLike, options?: ObjectEncodingOptions | string | null): Promise<string | Buffer>;

/\*\*

\* 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.

\* @since v10.0.0

\* @param [type='file']

\* @return Fulfills with `undefined` upon success.

\*/

function symlink(target: PathLike, path: PathLike, type?: string | null): Promise<void>;

/\*\*

\* 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)`](http://man7.org/linux/man-pages/man2/lstat.2.html) document for more detail.

\* @since v10.0.0

\* @return Fulfills with the {fs.Stats} object for the given symbolic link `path`.

\*/

function lstat(

path: PathLike,

opts?: StatOptions & {

bigint?: false | undefined;

}

): Promise<Stats>;

function lstat(

path: PathLike,

opts: StatOptions & {

bigint: true;

}

): Promise<BigIntStats>;

function lstat(path: PathLike, opts?: StatOptions): Promise<Stats | BigIntStats>;

/\*\*

\* @since v10.0.0

\* @return Fulfills with the {fs.Stats} object for the given `path`.

\*/

function stat(

path: PathLike,

opts?: StatOptions & {

bigint?: false | undefined;

}

): Promise<Stats>;

function stat(

path: PathLike,

opts: StatOptions & {

bigint: true;

}

): Promise<BigIntStats>;

function stat(path: PathLike, opts?: StatOptions): Promise<Stats | BigIntStats>;

/\*\*

\* Creates a new link from the `existingPath` to the `newPath`. See the POSIX [`link(2)`](http://man7.org/linux/man-pages/man2/link.2.html) documentation for more detail.

\* @since v10.0.0

\* @return Fulfills with `undefined` upon success.

\*/

function link(existingPath: PathLike, newPath: PathLike): Promise<void>;

/\*\*

\* 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)`](http://man7.org/linux/man-pages/man2/unlink.2.html) documentation for more detail.

\* @since v10.0.0

\* @return Fulfills with `undefined` upon success.

\*/

function unlink(path: PathLike): Promise<void>;

/\*\*

\* Changes the permissions of a file.

\* @since v10.0.0

\* @return Fulfills with `undefined` upon success.

\*/

function chmod(path: PathLike, mode: Mode): Promise<void>;

/\*\*

\* Changes the permissions on a symbolic link.

\*

\* This method is only implemented on macOS.

\* @deprecated Since v10.0.0

\* @return Fulfills with `undefined` upon success.

\*/

function lchmod(path: PathLike, mode: Mode): Promise<void>;

/\*\*

\* Changes the ownership on a symbolic link.

\* @since v10.0.0

\* @return Fulfills with `undefined` upon success.

\*/

function lchown(path: PathLike, uid: number, gid: number): Promise<void>;

/\*\*

\* 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.

\* @since v14.5.0, v12.19.0

\* @return Fulfills with `undefined` upon success.

\*/

function lutimes(path: PathLike, atime: string | number | Date, mtime: string | number | Date): Promise<void>;

/\*\*

\* Changes the ownership of a file.

\* @since v10.0.0

\* @return Fulfills with `undefined` upon success.

\*/

function chown(path: PathLike, uid: number, gid: number): Promise<void>;

/\*\*

\* 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.

\* @since v10.0.0

\* @return Fulfills with `undefined` upon success.

\*/

function utimes(path: PathLike, atime: string | number | Date, mtime: string | number | Date): Promise<void>;

/\*\*

\* 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.

\* @since v10.0.0

\* @return Fulfills with the resolved path upon success.

\*/

function realpath(path: PathLike, options?: ObjectEncodingOptions | BufferEncoding | null): Promise<string>;

/\*\*

\* Asynchronous realpath(3) - return the canonicalized absolute pathname.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function realpath(path: PathLike, options: BufferEncodingOption): Promise<Buffer>;

/\*\*

\* Asynchronous realpath(3) - return the canonicalized absolute pathname.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function realpath(path: PathLike, options?: ObjectEncodingOptions | BufferEncoding | null): Promise<string | Buffer>;

/\*\*

\* 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.

\*

\* ```js

\* 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`).

\* @since v10.0.0

\* @return Fulfills with a string containing the filesystem path of the newly created temporary directory.

\*/

function mkdtemp(prefix: string, options?: ObjectEncodingOptions | BufferEncoding | null): Promise<string>;

/\*\*

\* Asynchronously creates a unique temporary directory.

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

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function mkdtemp(prefix: string, options: BufferEncodingOption): Promise<Buffer>;

/\*\*

\* Asynchronously creates a unique temporary directory.

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

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function mkdtemp(prefix: string, options?: ObjectEncodingOptions | BufferEncoding | null): Promise<string | Buffer>;

/\*\*

\* Asynchronously writes data to a file, replacing the file if it already exists.`data` can be a string, a `Buffer`, or, an object with an own (not inherited)`toString` function property.

\*

\* 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.

\*

\* ```js

\* 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.

\* @since v10.0.0

\* @param file filename or `FileHandle`

\* @return Fulfills with `undefined` upon success.

\*/

function writeFile(

file: PathLike | FileHandle,

data: string | NodeJS.ArrayBufferView | Iterable<string | NodeJS.ArrayBufferView> | AsyncIterable<string | NodeJS.ArrayBufferView> | Stream,

options?:

| (ObjectEncodingOptions & {

mode?: Mode | undefined;

flag?: OpenMode | undefined;

} & Abortable)

| BufferEncoding

| null

): Promise<void>;

/\*\*

\* 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()`).

\* @since v10.0.0

\* @param path filename or {FileHandle}

\* @return Fulfills with `undefined` upon success.

\*/

function appendFile(path: PathLike | FileHandle, data: string | Uint8Array, options?: (ObjectEncodingOptions & FlagAndOpenMode) | BufferEncoding | null): Promise<void>;

/\*\*

\* 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`:

\*

\* ```js

\* 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.

\* @since v10.0.0

\* @param path filename or `FileHandle`

\* @return Fulfills with the contents of the file.

\*/

function readFile(

path: PathLike | FileHandle,

options?:

| ({

encoding?: null | undefined;

flag?: OpenMode | undefined;

} & Abortable)

| null

): Promise<Buffer>;

/\*\*

\* Asynchronously reads the entire contents of a file.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* If a `FileHandle` is provided, the underlying file will \_not\_ be closed automatically.

\* @param options An object that may contain an optional flag.

\* If a flag is not provided, it defaults to `'r'`.

\*/

function readFile(

path: PathLike | FileHandle,

options:

| ({

encoding: BufferEncoding;

flag?: OpenMode | undefined;

} & Abortable)

| BufferEncoding

): Promise<string>;

/\*\*

\* Asynchronously reads the entire contents of a file.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* If a `FileHandle` is provided, the underlying file will \_not\_ be closed automatically.

\* @param options An object that may contain an optional flag.

\* If a flag is not provided, it defaults to `'r'`.

\*/

function readFile(

path: PathLike | FileHandle,

options?:

| (ObjectEncodingOptions &

Abortable & {

flag?: OpenMode | undefined;

})

| BufferEncoding

| null

): Promise<string | Buffer>;

/\*\*

\* Asynchronously open a directory for iterative scanning. See the POSIX [`opendir(3)`](http://man7.org/linux/man-pages/man3/opendir.3.html) 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:

\*

\* ```js

\* 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.

\* @since v12.12.0

\* @return Fulfills with an {fs.Dir}.

\*/

function opendir(path: PathLike, options?: OpenDirOptions): Promise<Dir>;

/\*\*

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

\*

\* ```js

\* 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()`.

\* @since v15.9.0, v14.18.0

\* @return of objects with the properties:

\*/

function watch(

filename: PathLike,

options:

| (WatchOptions & {

encoding: 'buffer';

})

| 'buffer'

): AsyncIterable<FileChangeInfo<Buffer>>;

/\*\*

\* Watch for changes on `filename`, where `filename` is either a file or a directory, returning an `FSWatcher`.

\* @param filename A path to a file or directory. If a URL is provided, it must use the `file:` protocol.

\* @param options Either the encoding for the filename provided to the listener, or an object optionally specifying encoding, persistent, and recursive options.

\* If `encoding` is not supplied, the default of `'utf8'` is used.

\* If `persistent` is not supplied, the default of `true` is used.

\* If `recursive` is not supplied, the default of `false` is used.

\*/

function watch(filename: PathLike, options?: WatchOptions | BufferEncoding): AsyncIterable<FileChangeInfo<string>>;

/\*\*

\* Watch for changes on `filename`, where `filename` is either a file or a directory, returning an `FSWatcher`.

\* @param filename A path to a file or directory. If a URL is provided, it must use the `file:` protocol.

\* @param options Either the encoding for the filename provided to the listener, or an object optionally specifying encoding, persistent, and recursive options.

\* If `encoding` is not supplied, the default of `'utf8'` is used.

\* If `persistent` is not supplied, the default of `true` is used.

\* If `recursive` is not supplied, the default of `false` is used.

\*/

function watch(filename: PathLike, options: WatchOptions | string): AsyncIterable<FileChangeInfo<string>> | AsyncIterable<FileChangeInfo<Buffer>>;

/\*\*

\* 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/`.

\* @since v16.7.0

\* @experimental

\* @param src source path to copy.

\* @param dest destination path to copy to.

\* @return Fulfills with `undefined` upon success.

\*/

function cp(source: string, destination: string, opts?: CopyOptions): Promise<void>;

}

declare module 'node:fs/promises' {

export \* from 'fs/promises';

}