os

Package os go1.15.2 Latest

Published: **Sep 9, 2020** | License: BSD-3-Clause | Standard library

Overview

Package os provides a platform-independent interface to operating system functionality. The design is Unix-like, although the error handling is Go-like; failing calls return values of type error rather than error numbers. Often, more information is available within the error. For example, if a call that takes a file name fails, such as Open or Stat, the error will include the failing file name when printed and will be of type *PathError, which may be unpacked for more information.

The os interface is intended to be uniform across all operating systems. Features not generally available appear in the system-specific package syscall.

Here is a simple example, opening a file and reading some of it.

```
file, err := os.Open("file.go") // For read access.
if err != nil {
    log.Fatal(err)
}
```

If the open fails, the error string will be self-explanatory, like

```
open file.go: no such file or directory
```

The file's data can then be read into a slice of bytes. Read and Write take their byte counts from the length of the argument slice.

```
data := make([]byte, 100)
count, err := file.Read(data)
if err != nil {
    log.Fatal(err)
}
fmt.Printf("read %d bytes: %q\n", count, data[:count])
```

Note: The maximum number of concurrent operations on a File may be limited by the OS or the system. The number should be high, but exceeding it may degrade performance or cause other issues.

Constants

```
const (
    // Exactly one of O_RDONLY, O_WRONLY, or O_RDWR must be specified.
    O_RDONLY int = syscall.O_RDONLY // open the file read-only.
```

```
    O_WRONLY int = syscall.O_WRONLY // open the file write-only.
    O_RDWR   int = syscall.O_RDWR   // open the file read-write.
    // The remaining values may be or'ed in to control behavior.
    O_APPEND int = syscall.O_APPEND // append data to the file when writing.
    O_CREATE int = syscall.O_CREAT  // create a new file if none exists.
    O_EXCL   int = syscall.O_EXCL   // used with O_CREATE, file must not exist.
    O_SYNC   int = syscall.O_SYNC   // open for synchronous I/O.
    O_TRUNC  int = syscall.O_TRUNC  // truncate regular writable file when opened.
)
```

Flags to OpenFile wrapping those of the underlying system. Not all flags may be implemented on a given system.

```
const (
    SEEK_SET int = 0 // seek relative to the origin of the file
    SEEK_CUR int = 1 // seek relative to the current offset
    SEEK_END int = 2 // seek relative to the end
)
```

Seek whence values.

Deprecated: Use io.SeekStart, io.SeekCurrent, and io.SeekEnd.

```
const (
    PathSeparator     = '/' // OS-specific path separator
    PathListSeparator = ':' // OS-specific path list separator
)
```

```
const DevNull = "/dev/null"
```

DevNull is the name of the operating system's ``null device." On Unix-like systems, it is "/dev/null"; on Windows, "NUL".

Variables

```
var (
    // ErrInvalid indicates an invalid argument.
    // Methods on File will return this error when the receiver is nil.
    ErrInvalid = errInvalid() // "invalid argument"

    ErrPermission       = errPermission()       // "permission denied"
    ErrExist            = errExist()            // "file already exists"
    ErrNotExist         = errNotExist()         // "file does not exist"
    ErrClosed           = errClosed()           // "file already closed"
    ErrNoDeadline       = errNoDeadline()       // "file type does not support deadli
    ErrDeadlineExceeded = errDeadlineExceeded() // "i/o timeout"
)
```

Portable analogs of some common system call errors.

Errors returned from this package may be tested against these errors with errors.Is.

```
var (
    Stdin  = NewFile(uintptr(syscall.Stdin), "/dev/stdin")
    Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")
    Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")
)
```

Stdin, Stdout, and Stderr are open Files pointing to the standard input, standard output, and standard error file descriptors.

Note that the Go runtime writes to standard error for panics and crashes; closing Stderr may cause those messages to go elsewhere, perhaps to a file opened later.

```
var Args []string
```

Args hold the command-line arguments, starting with the program name.

func Chdir

```
func Chdir(dir string) error
```

Chdir changes the current working directory to the named directory. If there is an error, it will be of type *PathError.

func Chmod

```
func Chmod(name string, mode FileMode) error
```

Chmod changes the mode of the named file to mode. If the file is a symbolic link, it changes the mode of the link's target. If there is an error, it will be of type *PathError.

A different subset of the mode bits are used, depending on the operating system.

On Unix, the mode's permission bits, ModeSetuid, ModeSetgid, and ModeSticky are used.

On Windows, only the 0200 bit (owner writable) of mode is used; it controls whether the file's read-only attribute is set or cleared. The other bits are currently unused. For compatibility with Go 1.12 and earlier, use a non-zero mode. Use mode 0400 for a read-only file and 0600 for a readable+writable file.

On Plan 9, the mode's permission bits, ModeAppend, ModeExclusive, and ModeTemporary are used.

func Chown

```
func Chown(name string, uid, gid int) error
```

Chown changes the numeric uid and gid of the named file. If the file is a symbolic link, it changes the uid and gid of the link's target. A uid or gid of -1 means to not change that value. If there is an error, it will be of type *PathError.

On Windows or Plan 9, Chown always returns the syscall.EWINDOWS or EPLAN9 error, wrapped in *PathError.

func Chtimes

```
func Chtimes(name string, atime time.Time, mtime time.Time) error
```

Chtimes changes the access and modification times of the named file, similar to the Unix utime() or utimes() functions.

The underlying filesystem may truncate or round the values to a less precise time unit. If there is an error, it will be of type *PathError.

func Clearenv

```
func Clearenv()
```

Clearenv deletes all environment variables.

func Environ

```
func Environ() []string
```

Environ returns a copy of strings representing the environment, in the form "key=value".

func Executable

```
func Executable() (string, error)
```

Executable returns the path name for the executable that started the current process. There is no guarantee that the path is still pointing to the correct executable. If a symlink was used to start the process, depending on the operating system, the result might be the symlink or the path it pointed to. If a stable result is needed, path/filepath.EvalSymlinks might help.

Executable returns an absolute path unless an error occurred.

The main use case is finding resources located relative to an executable.

func Exit

```
func Exit(code int)
```

Exit causes the current program to exit with the given status code. Conventionally, code zero indicates success, non-zero an error. The program terminates immediately; deferred functions are not run.

For portability, the status code should be in the range [0, 125].

func Expand

```
func Expand(s string, mapping func(string) string) string
```

Expand replaces ${var} or $var in the string based on the mapping function. For example, os.ExpandEnv(s) is equivalent to os.Expand(s, os.Getenv).

func ExpandEnv

```
func ExpandEnv(s string) string
```

ExpandEnv replaces ${var} or $var in the string according to the values of the current environment variables. References to undefined variables are replaced by the empty string.

func Getegid

```
func Getegid() int
```

Getegid returns the numeric effective group id of the caller.

On Windows, it returns -1.

func Getenv

```
func Getenv(key string) string
```

Getenv retrieves the value of the environment variable named by the key. It returns the value, which will be empty if the variable is not present. To distinguish between an empty value and an unset value, use LookupEnv.

func Geteuid

```
func Geteuid() int
```

Geteuid returns the numeric effective user id of the caller.

On Windows, it returns -1.

func Getgid

```
func Getgid() int
```

Getgid returns the numeric group id of the caller.

On Windows, it returns -1.

func Getgroups

```
func Getgroups() ([]int, error)
```

Getgroups returns a list of the numeric ids of groups that the caller belongs to.

On Windows, it returns syscall.EWINDOWS. See the os/user package for a possible alternative.

func Getpagesize

```
func Getpagesize() int
```

Getpagesize returns the underlying system's memory page size.

func Getpid

```
func Getpid() int
```

Getpid returns the process id of the caller.

func Getppid

```
func Getppid() int
```

Getppid returns the process id of the caller's parent.

func Getuid

```
func Getuid() int
```

Getuid returns the numeric user id of the caller.

On Windows, it returns -1.

func Getwd

```
func Getwd() (dir string, err error)
```

Getwd returns a rooted path name corresponding to the current directory. If the current directory can be reached via multiple paths (due to symbolic links), Getwd may return any one of them.

func Hostname

```
func Hostname() (name string, err error)
```

Hostname returns the host name reported by the kernel.

func IsExist

```
func IsExist(err error) bool
```

IsExist returns a boolean indicating whether the error is known to report that a file or directory already exists. It is satisfied by ErrExist as well as some syscall errors.

func IsNotExist

```
func IsNotExist(err error) bool
```

IsNotExist returns a boolean indicating whether the error is known to report that a file or directory does not exist. It is satisfied by ErrNotExist as well as some syscall errors.

func IsPathSeparator

```
func IsPathSeparator(c uint8) bool
```

IsPathSeparator reports whether c is a directory separator character.

func IsPermission

```
func IsPermission(err error) bool
```

IsPermission returns a boolean indicating whether the error is known to report that permission is denied. It is satisfied by ErrPermission as well as some syscall errors.

func IsTimeout

```
func IsTimeout(err error) bool
```

IsTimeout returns a boolean indicating whether the error is known to report that a timeout occurred.

func Lchown

```
func Lchown(name string, uid, gid int) error
```

Lchown changes the numeric uid and gid of the named file. If the file is a symbolic link, it changes the uid and gid of the link itself. If there is an error, it will be of type *PathError.

On Windows, it always returns the syscall.EWINDOWS error, wrapped in *PathError.

func Link

```
func Link(oldname, newname string) error
```

Link creates newname as a hard link to the oldname file. If there is an error, it will be of type *LinkError.

func LookupEnv

```
func LookupEnv(key string) (string, bool)
```

LookupEnv retrieves the value of the environment variable named by the key. If the variable is present in the environment the value (which may be empty) is returned and the boolean is true. Otherwise the returned value will be empty and the boolean will be false.

func Mkdir

```
func Mkdir(name string, perm FileMode) error
```

Mkdir creates a new directory with the specified name and permission bits (before umask). If there is an error, it will be of type *PathError.

func MkdirAll

```
func MkdirAll(path string, perm FileMode) error
```

MkdirAll creates a directory named path, along with any necessary parents, and returns nil, or else returns an error. The permission bits perm (before umask) are used for all directories that MkdirAll creates. If path is already a directory, MkdirAll does nothing and returns nil.

func NewSyscallError

```
func NewSyscallError(syscall string, err error) error
```

NewSyscallError returns, as an error, a new SyscallError with the given system call name and error details. As a convenience, if err is nil, NewSyscallError returns nil.

func Pipe

```
func Pipe() (r *File, w *File, err error)
```

Pipe returns a connected pair of Files; reads from r return bytes written to w. It returns the files and an error, if any.

func Readlink

```
func Readlink(name string) (string, error)
```

Readlink returns the destination of the named symbolic link. If there is an error, it will be of type *PathError.

func Remove

```
func Remove(name string) error
```

Remove removes the named file or (empty) directory. If there is an error, it will be of type *PathError.

func RemoveAll

```
func RemoveAll(path string) error
```

RemoveAll removes path and any children it contains. It removes everything it can but returns the first error it encounters. If the path does not exist, RemoveAll returns nil (no error). If there is an error, it will be of type *PathError.

func Rename

```
func Rename(oldpath, newpath string) error
```

Rename renames (moves) oldpath to newpath. If newpath already exists and is not a directory, Rename replaces it. OS-specific restrictions may apply when oldpath and newpath are in different directories. If there is an error, it will be of type *LinkError.

func SameFile

```
func SameFile(fi1, fi2 FileInfo) bool
```

SameFile reports whether fi1 and fi2 describe the same file. For example, on Unix this means that the device and inode fields of the two underlying structures are identical; on other systems the decision may be based on the path names. SameFile only applies to results returned by this package's Stat. It returns false in other cases.

func Setenv

```
func Setenv(key, value string) error
```

Setenv sets the value of the environment variable named by the key. It returns an error, if any.

func Symlink

```
func Symlink(oldname, newname string) error
```

Symlink creates newname as a symbolic link to oldname. If there is an error, it will be of type *LinkError.

func TempDir

```
func TempDir() string
```

TempDir returns the default directory to use for temporary files.

On Unix systems, it returns $TMPDIR if non-empty, else /tmp. On Windows, it uses GetTempPath, returning the first non-empty value from %TMP%, %TEMP%, %USERPROFILE%, or the Windows directory. On Plan 9, it returns /tmp.

The directory is neither guaranteed to exist nor have accessible permissions.

func Truncate

```
func Truncate(name string, size int64) error
```

Truncate changes the size of the named file. If the file is a symbolic link, it changes the size of the link's target. If there is an error, it will be of type *PathError.

func Unsetenv

```
func Unsetenv(key string) error
```

Unsetenv unsets a single environment variable.

func UserCacheDir

```
func UserCacheDir() (string, error)
```

UserCacheDir returns the default root directory to use for user-specific cached data. Users should create their own application-specific subdirectory within this one and use that.

On Unix systems, it returns $XDG_CACHE_HOME as specified by https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html if non-empty, else $HOME/.cache. On Darwin, it returns $HOME/Library/Caches. On Windows, it returns %LocalAppData%. On Plan 9, it returns $home/lib/cache.

If the location cannot be determined (for example, $HOME is not defined), then it will return an error.

func UserConfigDir

```
func UserConfigDir() (string, error)
```

UserConfigDir returns the default root directory to use for user-specific configuration data. Users should create their own application-specific subdirectory within this one and use that.

On Unix systems, it returns $XDG_CONFIG_HOME as specified by https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html if non-empty, else $HOME/.config. On Darwin, it returns $HOME/Library/Application Support. On Windows, it returns %AppData%. On Plan 9, it returns $home/lib.

If the location cannot be determined (for example, $HOME is not defined), then it will return an error.

func UserHomeDir

```
func UserHomeDir() (string, error)
```

UserHomeDir returns the current user's home directory.

On Unix, including macOS, it returns the $HOME environment variable. On Windows, it returns %USERPROFILE%. On Plan 9, it returns the $home environment variable.

type File

```
type File struct {
    // contains filtered or unexported fields
}
```

File represents an open file descriptor.

func Create

```
func Create(name string) (*File, error)
```

Create creates or truncates the named file. If the file already exists, it is truncated. If the file does not exist, it is created with mode 0666 (before umask). If successful, methods on the returned File can be used for I/O; the associated file descriptor has mode O_RDWR. If there is an error, it will be of type *PathError.

func NewFile

```
func NewFile(fd uintptr, name string) *File
```

NewFile returns a new File with the given file descriptor and name. The returned value will be nil if fd is not a valid file descriptor. On Unix systems, if the file descriptor is in non-blocking mode, NewFile will attempt to return a pollable File (one for which the SetDeadline methods work).

func Open

```
func Open(name string) (*File, error)
```

Open opens the named file for reading. If successful, methods on the returned file can be used for reading; the associated file descriptor has mode O_RDONLY. If there is an error, it will be of type *PathError.

func OpenFile

```
func OpenFile(name string, flag int, perm FileMode) (*File, error)
```

OpenFile is the generalized open call; most users will use Open or Create instead. It opens the named file with specified flag (O_RDONLY etc.). If the file does not exist, and the O_CREATE flag is passed, it is created with mode perm (before umask). If successful, methods on the returned File can be used for I/O. If there is an error, it will be of type *PathError.

func (*File) Chdir

```
func (f *File) Chdir() error
```

Chdir changes the current working directory to the file, which must be a directory. If there is an error, it will be of type *PathError.

func (*File) Chmod

```
func (f *File) Chmod(mode FileMode) error
```

Chmod changes the mode of the file to mode. If there is an error, it will be of type *PathError.

func (*File) Chown

```
func (f *File) Chown(uid, gid int) error
```

Chown changes the numeric uid and gid of the named file. If there is an error, it will be of type *PathError.

On Windows, it always returns the syscall.EWINDOWS error, wrapped in *PathError.

func (*File) Close

```
func (f *File) Close() error
```

Close closes the File, rendering it unusable for I/O. On files that support SetDeadline, any pending I/O operations will be canceled and return immediately with an error. Close will return an error if it has already been called.

func (*File) Fd

```
func (f *File) Fd() uintptr
```

Fd returns the integer Unix file descriptor referencing the open file. The file descriptor is valid only until f.Close is called or f is garbage collected. On Unix systems this will cause the SetDeadline methods to stop working.

func (*File) Name

```
func (f *File) Name() string
```

Name returns the name of the file as presented to Open.

func (*File) Read

```
func (f *File) Read(b []byte) (n int, err error)
```

Read reads up to len(b) bytes from the File. It returns the number of bytes read and any error encountered. At end of file, Read returns 0, io.EOF.

func (*File) ReadAt

```
func (f *File) ReadAt(b []byte, off int64) (n int, err error)
```

ReadAt reads len(b) bytes from the File starting at byte offset off. It returns the number of bytes read and the error, if any. ReadAt always returns a non-nil error when n < len(b). At end of file, that error is io.EOF.

func (*File) ReadFrom

```
func (f *File) ReadFrom(r io.Reader) (n int64, err error)
```

ReadFrom implements io.ReaderFrom.

func (*File) Readdir

```
func (f *File) Readdir(n int) ([]FileInfo, error)
```

Readdir reads the contents of the directory associated with file and returns a slice of up to n FileInfo values, as would be returned by Lstat, in directory order. Subsequent calls on the same file will yield further FileInfos.

If n > 0, Readdir returns at most n FileInfo structures. In this case, if Readdir returns an empty slice, it will return a non-nil error explaining why. At the end of a directory, the error is io.EOF.

If n <= 0, Readdir returns all the FileInfo from the directory in a single slice. In this case, if Readdir succeeds (reads all the way to the end of the directory), it returns the slice and a nil error. If it encounters an error before the end of the directory, Readdir returns the FileInfo read until that point and a non-nil error.

func (*File) Readdirnames

```
func (f *File) Readdirnames(n int) (names []string, err error)
```

Readdirnames reads the contents of the directory associated with file and returns a slice of up to n names of files in the directory, in directory order. Subsequent calls on the same file will yield further names.

If n > 0, Readdirnames returns at most n names. In this case, if Readdirnames returns an empty slice, it will return a non-nil error explaining why. At the end of a directory, the error is io.EOF.

If n <= 0, Readdirnames returns all the names from the directory in a single slice. In this case, if Readdirnames succeeds (reads all the way to the end of the directory), it returns the slice and a nil error. If it encounters an error before the end of the directory, Readdirnames returns the names read until that point and a non-nil error.

func (*File) Seek

```
func (f *File) Seek(offset int64, whence int) (ret int64, err error)
```

Seek sets the offset for the next Read or Write on file to offset, interpreted according to whence: 0 means relative to the origin of the file, 1 means relative to the current offset, and 2 means relative to the end. It returns the new offset and an error, if any. The behavior of Seek on a file opened with O_APPEND is not specified.

If f is a directory, the behavior of Seek varies by operating system; you can seek to the beginning of the directory on Unix-like operating systems, but not on Windows.

func (*File) SetDeadline

```
func (f *File) SetDeadline(t time.Time) error
```

SetDeadline sets the read and write deadlines for a File. It is equivalent to calling both SetReadDeadline and SetWriteDeadline.

Only some kinds of files support setting a deadline. Calls to SetDeadline for files that do not support deadlines will return ErrNoDeadline. On most systems ordinary files do not support deadlines, but pipes do.

A deadline is an absolute time after which I/O operations fail with an error instead of blocking. The deadline applies to all future and pending I/O, not just the immediately following call to Read or Write.

After a deadline has been exceeded, the connection can be refreshed by setting a deadline in the future.

If the deadline is exceeded a call to Read or Write or to other I/O methods will return an error that wraps ErrDeadlineExceeded. This can be tested using errors.Is(err, os.ErrDeadlineExceeded). That error implements the Timeout method, and calling the Timeout method will return true, but there are other possible errors for which the Timeout will return true even if the deadline has not been exceeded.

An idle timeout can be implemented by repeatedly extending the deadline after successful Read or Write calls.

A zero value for t means I/O operations will not time out.

func (*File) SetReadDeadline

```
func (f *File) SetReadDeadline(t time.Time) error
```

SetReadDeadline sets the deadline for future Read calls and any currently-blocked Read call. A zero value for t means Read will not time out. Not all files support setting deadlines; see SetDeadline.

func (*File) SetWriteDeadline

```
func (f *File) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline sets the deadline for any future Write calls and any currently-blocked Write call. Even if Write times out, it may return n > 0, indicating that some of the data was successfully written. A zero value for t means Write will not time out. Not all files support setting deadlines; see SetDeadline.

func (*File) Stat

```
func (f *File) Stat() (FileInfo, error)
```

Stat returns the FileInfo structure describing file. If there is an error, it will be of type *PathError.

func (*File) Sync

```
func (f *File) Sync() error
```

Sync commits the current contents of the file to stable storage. Typically, this means flushing the file system's in-memory copy of recently written data to disk.

func (*File) SyscallConn

```
func (f *File) SyscallConn() (syscall.RawConn, error)
```

SyscallConn returns a raw file. This implements the syscall.Conn interface.

func (*File) Truncate

```
func (f *File) Truncate(size int64) error
```

Truncate changes the size of the file. It does not change the I/O offset. If there is an error, it will be of type *PathError.

func (*File) Write

```
func (f *File) Write(b []byte) (n int, err error)
```

Write writes len(b) bytes to the File. It returns the number of bytes written and an error, if any. Write returns a non-nil error when n != len(b).

func (*File) WriteAt

```
func (f *File) WriteAt(b []byte, off int64) (n int, err error)
```

WriteAt writes len(b) bytes to the File starting at byte offset off. It returns the number of bytes written and an error, if any. WriteAt returns a non-nil error when n != len(b).

If file was opened with the O_APPEND flag, WriteAt returns an error.

func (*File) WriteString

```
func (f *File) WriteString(s string) (n int, err error)
```

WriteString is like Write, but writes the contents of string s rather than a slice of bytes.

type FileInfo

```
type FileInfo interface {
    Name() string       // base name of the file
    Size() int64        // length in bytes for regular files; system-dependent for ot
    Mode() FileMode     // file mode bits
    ModTime() time.Time // modification time
    IsDir() bool        // abbreviation for Mode().IsDir()
    Sys() interface{}   // underlying data source (can return nil)
}
```

A FileInfo describes a file and is returned by Stat and Lstat.

func Lstat

```
func Lstat(name string) (FileInfo, error)
```

Lstat returns a FileInfo describing the named file. If the file is a symbolic link, the returned FileInfo describes the symbolic link. Lstat makes no attempt to follow the link. If there is an error, it will be of type *PathError.

func Stat

```
func Stat(name string) (FileInfo, error)
```

Stat returns a FileInfo describing the named file. If there is an error, it will be of type *PathError.

type FileMode

```
type FileMode uint32
```

A FileMode represents a file's mode and permission bits. The bits have the same definition on all systems, so that information about files can be moved from one system to another portably. Not all bits apply to all systems. The only required bit is ModeDir for directories.

```
const (
    // The single letters are the abbreviations
    // used by the String method's formatting.
    ModeDir        FileMode = 1 << (32 - 1 - iota) // d: is a directory
    ModeAppend                                     // a: append-only
    ModeExclusive                                  // l: exclusive use
    ModeTemporary                                  // T: temporary file; Plan 9 only
    ModeSymlink                                    // L: symbolic link
    ModeDevice                                     // D: device file
    ModeNamedPipe                                  // p: named pipe (FIFO)
    ModeSocket                                     // S: Unix domain socket
    ModeSetuid                                     // u: setuid
    ModeSetgid                                     // g: setgid
    ModeCharDevice                                 // c: Unix character device, when
    ModeSticky                                     // t: sticky
    ModeIrregular                                  // ?: non-regular file; nothing el

    // Mask for the type bits. For regular files, none will be set.
    ModeType = ModeDir | ModeSymlink | ModeNamedPipe | ModeSocket | ModeDevice | Mode

    ModePerm FileMode = 0777 // Unix permission bits
)
```

The defined file mode bits are the most significant bits of the FileMode. The nine least-significant bits are the standard Unix rwxrwxrwx permissions. The values of these bits should be considered part of the public API and may be used in wire protocols or disk representations: they must not be changed, although new bits might be added.

func (FileMode) IsDir

```
func (m FileMode) IsDir() bool
```

IsDir reports whether m describes a directory. That is, it tests for the ModeDir bit being set in m.

func (FileMode) IsRegular

```
func (m FileMode) IsRegular() bool
```

IsRegular reports whether m describes a regular file. That is, it tests that no mode type bits are set.

func (FileMode) Perm

```
func (m FileMode) Perm() FileMode
```

Perm returns the Unix permission bits in m.

func (FileMode) String

```
func (m FileMode) String() string
```

type LinkError

```
type LinkError struct {
    Op  string
    Old string
    New string
    Err error
}
```

LinkError records an error during a link or symlink or rename system call and the paths that caused it.

func (*LinkError) Error

```
func (e *LinkError) Error() string
```

func (*LinkError) Unwrap

```
func (e *LinkError) Unwrap() error
```

type PathError

```
type PathError struct {
    Op   string
    Path string
    Err  error
}
```

PathError records an error and the operation and file path that caused it.

func (*PathError) Error

```
func (e *PathError) Error() string
```

func (*PathError) Timeout

```
func (e *PathError) Timeout() bool
```

Timeout reports whether this error represents a timeout.

func (*PathError) Unwrap

```
func (e *PathError) Unwrap() error
```

type ProcAttr

```
type ProcAttr struct {
    // If Dir is non-empty, the child changes into the directory before
    // creating the process.
    Dir string
    // If Env is non-nil, it gives the environment variables for the
    // new process in the form returned by Environ.
    // If it is nil, the result of Environ will be used.
    Env []string
    // Files specifies the open files inherited by the new process. The
    // first three entries correspond to standard input, standard output, and
    // standard error. An implementation may support additional entries,
    // depending on the underlying operating system. A nil entry corresponds
    // to that file being closed when the process starts.
    Files []*File

    // Operating system-specific process creation attributes.
    // Note that setting this field means that your program
    // may not execute properly or even compile on some
    // operating systems.
    Sys *syscall.SysProcAttr
}
```

ProcAttr holds the attributes that will be applied to a new process started by StartProcess.

type Process

```
type Process struct {
    Pid int
```

```
    // contains filtered or unexported fields
}
```

Process stores the information about a process created by StartProcess.

func FindProcess

```
func FindProcess(pid int) (*Process, error)
```

FindProcess looks for a running process by its pid.

The Process it returns can be used to obtain information about the underlying operating system process.

On Unix systems, FindProcess always succeeds and returns a Process for the given pid, regardless of whether the process exists.

func StartProcess

```
func StartProcess(name string, argv []string, attr *ProcAttr) (*Process, error)
```

StartProcess starts a new process with the program, arguments and attributes specified by name, argv and attr. The argv slice will become os.Args in the new process, so it normally starts with the program name.

If the calling goroutine has locked the operating system thread with runtime.LockOSThread and modified any inheritable OS-level thread state (for example, Linux or Plan 9 name spaces), the new process will inherit the caller's thread state.

StartProcess is a low-level interface. The os/exec package provides higher-level interfaces.

If there is an error, it will be of type *PathError.

func (*Process) Kill

```
func (p *Process) Kill() error
```

Kill causes the Process to exit immediately. Kill does not wait until the Process has actually exited. This only kills the Process itself, not any other processes it may have started.

func (*Process) Release

```
func (p *Process) Release() error
```

Release releases any resources associated with the Process p, rendering it unusable in the future. Release only needs to be called if Wait is not.

func (*Process) Signal

```
func (p *Process) Signal(sig Signal) error
```

Signal sends a signal to the Process. Sending Interrupt on Windows is not implemented.

func (*Process) Wait

```
func (p *Process) Wait() (*ProcessState, error)
```

Wait waits for the Process to exit, and then returns a ProcessState describing its status and an error, if any. Wait releases any resources associated with the Process. On most operating systems, the Process must be a child of the current process or an error will be returned.

type ProcessState

```
type ProcessState struct {
    // contains filtered or unexported fields
}
```

ProcessState stores information about a process, as reported by Wait.

func (*ProcessState) ExitCode

```
func (p *ProcessState) ExitCode() int
```

ExitCode returns the exit code of the exited process, or -1 if the process hasn't exited or was terminated by a signal.

func (*ProcessState) Exited

```
func (p *ProcessState) Exited() bool
```

Exited reports whether the program has exited.

func (*ProcessState) Pid

```
func (p *ProcessState) Pid() int
```

Pid returns the process id of the exited process.

func (*ProcessState) String

```
func (p *ProcessState) String() string
```

func (*ProcessState) Success

```
func (p *ProcessState) Success() bool
```

Success reports whether the program exited successfully, such as with exit status 0 on Unix.

func (*ProcessState) Sys

```
func (p *ProcessState) Sys() interface{}
```

Sys returns system-dependent exit information about the process. Convert it to the appropriate underlying type, such as syscall.WaitStatus on Unix, to access its contents.

func (*ProcessState) SysUsage

```
func (p *ProcessState) SysUsage() interface{}
```

SysUsage returns system-dependent resource usage information about the exited process. Convert it to the appropriate underlying type, such as *syscall.Rusage on Unix, to access its contents. (On Unix, *syscall.Rusage matches struct rusage as defined in the getrusage(2) manual page.)

func (*ProcessState) SystemTime

```
func (p *ProcessState) SystemTime() time.Duration
```

SystemTime returns the system CPU time of the exited process and its children.

func (*ProcessState) UserTime

```
func (p *ProcessState) UserTime() time.Duration
```

UserTime returns the user CPU time of the exited process and its children.

type Signal

```
type Signal interface {
    String() string
    Signal() // to distinguish from other Stringers
}
```

A Signal represents an operating system signal. The usual underlying implementation is operating system-dependent: on Unix it is syscall.Signal.

```
var (
    Interrupt Signal = syscall.SIGINT
```

```
    Kill      Signal = syscall.SIGKILL
)
```

The only signal values guaranteed to be present in the os package on all systems are os.Interrupt (send the process an interrupt) and os.Kill (force the process to exit). On Windows, sending os.Interrupt to a process with os.Process.Signal is not implemented; it will return an error instead of sending a signal.

type SyscallError

```
type SyscallError struct {
    Syscall string
    Err     error
}
```

SyscallError records an error from a specific system call.

func (*SyscallError) Error

```
func (e *SyscallError) Error() string
```

func (*SyscallError) Timeout

```
func (e *SyscallError) Timeout() bool
```

Timeout reports whether this error represents a timeout.

func (*SyscallError) Unwrap

```
func (e *SyscallError) Unwrap() error
```